Compiler Design - Cheatsheet

Simplified Compiler Structure

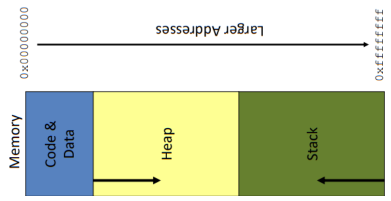
Source Code $\rightarrow$

$\underbrace{\text{Lexical Analysis} \rightarrow^{\text{Token Stream}} \text{Parsing} \rightarrow^{\text{Abstract Syntax Tree}}}_{\text{Front End}}$

$\underbrace{\text{Intermediate Code Generation} \rightarrow^{\text{Intermediate Code}}}_{\text{Middle End}}$

$\underbrace{\text{Code Generation}}_{\text{Back End}}$

Stack layout



Suffiexes

1. q = quadword (4 words)
2. l = long (2 words)
3. w = word (16-bit)
4. b = byte (8-bit)

movq-instruction

movq SRC, DST = $DST \leftarrow SRC$

Flags

e (equality)	ZF is set
ne (inequality)	(not ZF)
g (greater than)	(not ZF) and (SF = OF)
l (less than)	SF <> OF
ge (greater or equal)	(SF = OF)
e (less than or equal)	SF <> OF or Z

Addressing

−8 (%rsp)	rsp – 8
(%rax, %rcx)	rax + 8·rcx
8(%rax, %rcx)	rax + 8·rcx + 8

leaq vs. movq

In leaq we just compute the address, in movq we dereference the address.

getelementptr instruction

The first argument is always a type used as the basis for the calculations. The second argument is always a pointer or a vector of pointers, and is the base address to start from. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. GEP never dereferences the address it's calculating.

Regular Expressions

Regular expressions precisely describe sets of strings. A regular expression R has one of the following forms:

1. ϵ: Epsilon stands for the empty string
2. $'a'$: An ordinary character stands for itself
3. $R_1|R_2$: Alternatives, stands for choice of R_1 or R_2
4. R_1R_2: Concatenation, stands for R_1 followed by R_2
5. R^*: Kleene star, stands for zero or more repetitions of R

Useful extensions:

1. $''foo''$: Strings, equivalent to $'f' \, 'o' \, 'o'$
2. R^+: One or more repetitions of R, equivalent to RR^*
3. $R?$: Zero or one occurrences of R, equivalent to $(\epsilon|R)$
4. $['a' - 'z']$: One of a or b or c or $\ldots z$, equivalent to $(a|b|\ldots|z)$
5. $[\hat{}\,'0' - '9']$: Any character except 0 through 9
6. R as x: Name the string matched by R as x

Chomsky Hierarchy

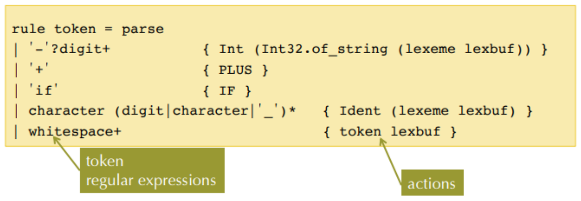
Regular $\subset$ Context-Free $\subset$ Context Sensitive $\subset$ Recursively Enumerable

Matching Rule for Lexer

Most languages choose "longest match".

Lexer Generator

1. Reads a list of regular expressions: $R_1, \ldots, R_n$, one per token
2. Each token has an attached "action" A_i (just a piece of code to run when the regular expression is matched)



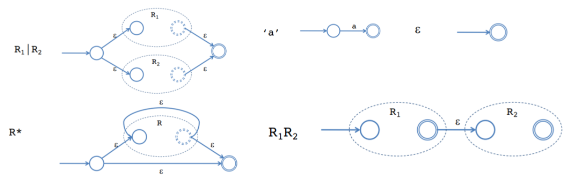
```
rule token = parse
| '-'?digit+              { Int (Int32.of_string (lexeme lexbuf)) }
| '+'                     { PLUS }
| 'if'                    { IF }
| character (digit|character|'_')*   { Ident (lexeme lexbuf) }
| whitespace+                        { token lexbuf }
```

Generates scanning code that:

1. Decides whether the input is of the form $(R_1|\ldots|R_n)^*$
2. After matching a (longest) token, runs the associated action

Implementation Strategies

We can use a finite automaton since one must exist for a regular expression.



DFA vs. NFA

DFA:

1. Action of the automaton for each input is fully determined
2. Accepts if the input is consumed upon reaching an accepting state
3. Obvious table-based implementation

NFA:

1. Automaton potentially has a choice at every step
2. Accepts an input if there exists a way to reach an accepting state
3. Less obvious how to implement efficiently

Parsing

Def.: Finding Syntactic Structure
Limits of regular expressions:

1. DFA's have only finite # of states (i.e., finite memory)
2. So, DFA's can't count

Therefore we need something more powerful than DFA's.

CONTEXT FREE GRAMMARS

Def. recursive: S mentions itself, e.g. $S \mapsto (S)S$ A Context-free Grammar (CFG) consists of

1. A set of terminals (e.g., a lexical token, ϵ is not a terminal)
2. A set of nonterminals (e.g., S and other syntactic variables)
3. A designated nonterminal called the start symbol
4. A set of productions: $LHS \mapsto RHS$

single step: For arbitrary strings α, β, γ and production rule $A \mapsto \beta$ a single step of the derivation is $\alpha A \gamma \mapsto \alpha \beta \gamma$
Parse Tree: Leaves are terminals and Internal nodes are nonterminals.

Derivation Orders

Productions of the grammar can be applied in any order. Both strategies (and any other) yield the same parse tree!

1. Leftmost derivation: Find the left-most nonterminal and apply a production to it e.g. $S \mapsto \underline{\mathbf{E}} + S$
2. Rightmost derivation: Find the right-most nonterminal and apply a production there e.g. $S \mapsto E + \underline{\mathbf{S}}$

productive and nonproductive: This grammar has nonterminal definitions that don't mention any terminal symbols.
finite: There is a finite derivation starting from start symbol.
Example of right associative: $S \mapsto E + S|E$, $E \mapsto \text{number}|(S)$
Example of Ambiguity: $S \mapsto S + S|(S)|\text{number}$, accepts the same set of strings as the previous one but there are two leftmost derivations. Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their precedence.
Eliminating Ambiguity:

1. by adding nonterminals and allowing recursion only on the left (or right)
2. Higher-precedence operators go farther from the start symbol

Example of Eliminating Ambiguity: $S \mapsto S + S|(S)|\text{number}$ becomes $S_0 \mapsto S_0 + S_1|S_1$, $S_1 \mapsto S_2 * S_1|S_2$ and $S_2 \mapsto \text{number}|(S_0)$

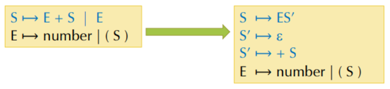
LL(1) GRAMMARS

Top-down: Start from the start symbol (root of the parse tree), and go down. Not all grammars can be parsed top-down"with a single lookahead.
LL(1) means:

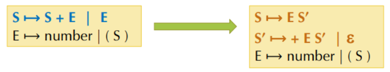
1. Left-to-right scanning
2. Left-most derivation
3. **1** lookahead symbol

Making a grammar LL(1)

Problem: We can't decide which S production to apply until we see the symbol after the first expression **Solution:** "Left-factor" the grammar. There is a common S prefix for each choice, so add a new non-terminal S' at the decision point.



Also need to eliminate left-recursion.
S $\mapsto$ S$\alpha_1 | \ldots |$S$\alpha_n | \beta_1 | \ldots | \beta_m$ becomes S $\mapsto \beta_1 S' | \ldots | \beta_m S'$
$S' \mapsto \alpha_1 S' | \ldots | \alpha_n S' \mid \varepsilon$



Predictive Parsing

For a given nonterminal, the lookahead symbol uniquely determines the production to apply. Therefore we get a table with nonterminal $\times$ input token $\rightarrow$ production

Construction of the parse table:

Consider a given production: $A \mapsto \gamma$.

1. (Case 1) Construct the set of all input tokens that may appear first in strings that can be derived from γ. Add the production $\mapsto \gamma$ to the entry (A, token) for each such token.
2. (Case 2) If γ can derive ϵ (the empty string), then we construct the set of all input tokens that may follow the nonterminal A in the grammar. Add the production $\mapsto \gamma$ to the entry (A, token) for each such token.

Example: We have
$T \mapsto S\$, S \mapsto ES', S' \mapsto e, S' \mapsto +S, E \mapsto \text{number}|(S)$. We get

1. First(S\$) = First(S) = First(E′) = First(E) = { number, ′(′ }
2. First(ϵ) = { ϵ }
3. First(+S) = { + }
4. First(number) = { number }
5. First((S)) = { ′(′ }
6. Follow(S′) = Follow(S)
7. Follow(S) = { \$, ′)′ } ∪ Follow(S′)

	number	+	(	)	\$ (EOF)
T	⟼ S\$		⟼S\$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ number		⟼ (S)		

LR GRAMMARS

LR(k) parser

1. Bottom-up Parsing
2. Left-to-right scanning
3. **R**ightmost derivation
4. **k** lookahead symbols

LR grammars are more expressive than LL. Can handle left-recursive (and right recursive) grammars.

Shift/Reduce Parsing

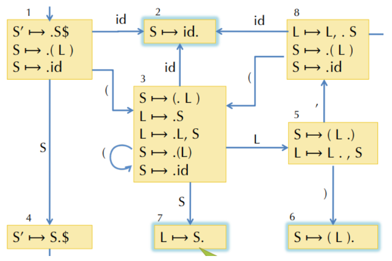
Shift: Move look-ahead token to the stack
Reduce:: Replace symbols γ at top of stack with nonterminal X s.t. $X \mapsto \gamma$ is a production

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

LR(0) state: items to track progress on possible upcoming reductions
LR(0) item: a production with an extra separator ".ïn the RHS, e.g. $S \mapsto .(L)$ or $S \mapsto (L)$. Idea is stuff before the '.' is already on the stack and stuff after the '.' is what might be seen next.

Constructing the DFA: Start state & Closure

1. Start state of the DFA = empty stack, so it contains the item $S' \mapsto .S\$$
2. 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 $S' \mapsto .S\$$). The added items have the '.' located at the beginning (no symbols for those items have been added to the stack yet), e.g. CLOSURE$(\{S' \mapsto .S\$\}) = \{S' \mapsto .S\$, S \mapsto .(L), S \mapsto .\text{id}\}$
3. Next we add the transitions, e.g after reading id we get $S \mapsto$ id.
4. Finally, for each new state, we take the closure
5. If a reduce state is reached, reduce otherwise, if the next token matches an outgoing edge, shift



Implementing the Parsing Table

Entries for the "action table" specify two kinds of actions: Shift and go to state n and Reduce using reduction $X \mapsto \gamma$. We only have reduction when we don't have any outgoing edges otherwise we shift.

	(	)	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		

LR(0) Limitations

shift/reduce: $S \mapsto (L)$. and $L \mapsto .L, S$ yield a problem in a state since with (L). we are allowed to reduce but with $.L$ we are also allowed to shift. **reduce/reduce:** $S \mapsto L, S$. and $S \mapsto, S$. yield a problem in a state since there exists two diffrent reduction rules for S.

LR(1) Parsing

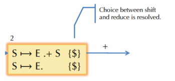
1. LR(1) state = set of LR(1) items
2. An LR(1) item is an LR(0) item + a set of look-ahead symbols $A \mapsto \alpha.\beta, \mathcal{L}$

LR(1) closure:

1. Form the set of items just as for LR(0) algorithm
2. Whenever a new item $C \mapsto .\gamma$ is added because $A \mapsto \beta.C\delta, \mathcal{L}$ is already in the set, we need to compute its look-ahead set $\mathcal{M}$
   (a) The look-ahead set $\mathcal{M}$ includes FIRST(δ)
   (b) If δ is or can derive ϵ, then the look-ahead $\mathcal{M}$ also contains $\mathcal{L}$

Example Closure: Asume we have
$S' \mapsto S\$, S \mapsto E + S|E, E \mapsto \text{number}|S$

- Start item: $S' \mapsto S\$, \{\}$
- Since S is to the right of a '.', add $S \mapsto .E + S, \{\$\}; S \mapsto .E, \{\$\}$
- Need to keep closing, since E appears to the right of a '.' in '.E + S', $E \mapsto .\text{number}, \{+\}; E \mapsto .(S), \{+\}$
- Because E also appears to the right of . in .E we get: $E \mapsto .\text{number}, \{\$\}; E \mapsto .(S), \{\$\}$
- All items are distinct, so we're done



	+	\$	E
1			g2
2	s3	S ⟼ E	

Ambiguity with if else:

```
if (E1) if (E2) S1 else S2
```

This is known as the dangling else problem. What should the right answer be? We know two solutions: the simple one would just require {} and another one could use the grammar

1. $S \mapsto M|U$ // M = matched, U = unmatched
2. $U \mapsto$ if (E) S // Unmatched if
3. $U \mapsto$ if (E) M else U // Nested if is matched
4. $M \mapsto$ if (E) M else M // Matched if
5. $M \mapsto$ X = E // Other statements

Lambda Calculus

The lambda calculus is a minimal programming language. It has variables, functions, and function application. It's Turing Complete. Concrete syntax:

```
exp ::=
    | x               //variables
    | fun x -> exp    //functions
    | exp1 exp2       //function application
    | ( exp )         //parentheses
```

The only values of the lambda calculus are (closed) functions that is val ::= fun x → exp.
substitute: Replace all free occurrences of x in e by v also written as $e\{v/\}$ If we try to substitute a variable which is not free, the expression remains the same.
x{v/x} = v (replace the free x by v)
$y\{v/x\}$ $= y$ (assuming $y \neq x$)
(fun $x \to$ exp)$\{v/x\}$ = (fun $x \to$ exp) (x is bound in exp)
(fun $y \to$ exp)$\{v/x\}$ = (fun $y \to$ exp$\{v/x\}$) (assuming $y \neq x$)
(e_1e_2) {v/x} = (e_1{v/x}e_2{v/x}) (substitute everywhere)
free variable: We say variable x is free in fun y → x + y. Free variables are defined in an outer scope
bound variable: We say variable y is bound by fun y. Its scope is the body x + y in fun y → x + y
closed: A term with no free variables is called closed.
open: A term with one or more free variables is called open. **Free Variable Calculation:**
$fv(x) = \{x\}$
$fv(\text{fun}\, x \to \text{exp}) = fv(\text{exp})\backslash\{x\}$
$fv(\text{exp}_1\, \text{exp}_2) = fv(\text{exp}_1) \cup fv(\text{exp}_2)$
Variable Capture: In
$(\text{fun}\, x \to (xy))\{(\text{fun}\, z \to x)/y\} = \text{fun}\, x \to (x(\text{fun}\, z \to x))$ x is captured. Usually not the desired behavior. This property is sometimes called dynamic scoping. The meaning of x is determined by where it is bound dynamically.
Alpha Equivalence: Two terms that differ only by consistent renaming of bound variables are called alpha equivalent, e.g. $(\text{fun}\, x \to yx)$ the same as $(\text{fun}\, z \to yz)$

Operational Semantics

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

$$\frac{\text{exp}_1 \Downarrow (\text{ fun } x \to \text{exp}_3) \quad \text{exp}_2 \Downarrow v \quad \text{exp}_3\{v/x\} \Downarrow w}{\text{exp}_1\ \text{exp}_2 \Downarrow w}$$

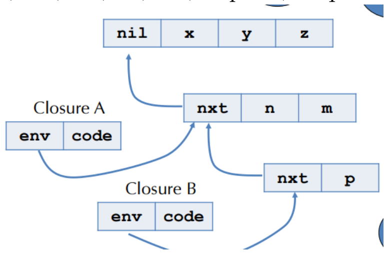
Y Combinator & Factorial

Y combinator: $Y == \backslash f.(\backslash x.f(x\ x))(\backslash x.f(x\ x))$. Y F = F (Y F) for any term F. **Example factorial function:**

- Typical recursive definition: $fac = \backslash n.if(n = 0)1(n * fac(n-1))$
- Abstract it: $F = \backslash f.\backslash n.if(n = 0)1(n * f(n-1))$
- Thus, fac = F fac (i.e., fac is a fixpoint of F)
- Y F being fixpoint of F can thus be viewed as the factorial function!

CLOSURE CONVERSION

The closure is a pair of the environment and a code pointer: Code(env, y, body). We get fun (env, y) → let x = nth env 0 in body, where y is possibly in body.
Array-based Closures with N-ary Functions: We have (fun (x y z) → (fun (n m) → (fun p → (fun q → n + z) x)))



Theorem: (simply typed lambda calculus with integers): If $\vdash e : t$, then there exists a value v such that $e \Downarrow v$
Theorem: (Type Safety) If $\vdash P : t$ is a well-typed program, then either

1. the program terminates in a well-defined way, or
2. the program continues computing forever

type: A type is just a predicate on the set of values in a system. E.g., 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.
subtype: This subset relation gives rise to a subtype relation: if $Pos <: Int$, then Pos is a subtype of Int
LUB: least upper bound, for statically unknown conditionals, we want the return value to be the LUB of the types of the branches. LUB is also called the join operation.
subtyping relation is a **partial order**, that is:

1. Reflexive: T ¡: T for any type T
2. Transitive: T1 ¡: T2 and T2 ¡: T3 then T1 ¡: T3
3. Antisymmetric: It T1 ¡: T2 and T2 ¡: T1 then T1 = T2

Soundness of Subtyping Relations: A subtyping relation T1 ¡: T2 is sound if it approximates the underlying semantic subset relation, that is let be $[\![T]\!] = \{v \mid \vdash v : T\}$. If T1 ¡: T2 implies $[\![T1]\!] \subseteq [\![T2]\!]$, then T1 ¡: T2 is sound. Whenever we have a sound subtyping relation, it follows that $[\![LUB(\text{T}_1, \text{T}_2)]\!] \supseteq [\![\text{T}_1]\!] \cup [\![\text{T}_2]\!]$. Using LUBs in the typing rules yields sound approximations of the program behavior (as if the IF-B rule)
Subsumption Rule: $\frac{E \vdash e:T \quad T <: S}{E \vdash e:S}$

Subtyping for Function Types: Need to convert an S1 to a T1 and T2 to S2, so the argument type is contravariant and the output type is covariant: $\frac{S1 <: T1 \quad T2 <: S2}{(T1 \to T2) <: (S1 \to S2)}$
reference types: Are not covariant because of

```
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 *)
}
```

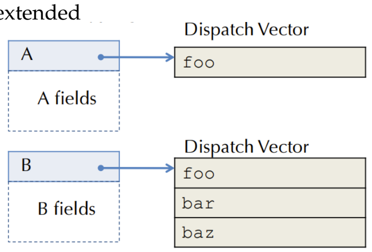
and not contravariant because of

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

Immutable Record Subtyping: it holds $\{x : int, y : int\} \neq \{y : int, x : int\}$ and $\{x : int, y : int, z : int\} <: \{x : int, y : int\}$
Mutable Structures: Mutable structures are invariant that is $T_1 ref <: T_2 ref \implies T_1 = T_2$
Structural vs. Nominal Typing: Checking against the name is nominal typing and checking against the structure is structural typing.

Compiling Objects

Objects contain a pointer to a dispatch vector (also called a virtual table or vtable) with pointers to method code
Single Inheritance: every method has its own small integer index, Index is used to look up the method in the dispatch vector. Each interface and class gives rise to a dispatch vector layout. DV layout of new method is appended to the class which is being extended



Multiple Inheritance: Multiple Inheritance approaches: Allow multiple DV tables (C++) Choose which DV to use based on static types, casting requires runtime operations; Use a level of indirection: Map method identifiers to code pointers using a hash table, search up through the class hierarchy; Give up separate compilation: Now we have access to the whole class hierarchy. Use sparse dispatch vectors or binary decision trees.

Optimization

An optimization must always be safe i.e. it must not change the behavior of the program.

- Problem: many optimizations trade space for time (e.g. Loop unrolling)
- Constant Folding: If operands are statically known, compute value at compile-time (has to preformed at every stage of optimization - constant expressions can be created by translation or earlier optimizations / can enable further optimizations). Also: Algebraic Simplification (Use mathematical identities e.g $a \cdot 1 \rightarrow a$)
- Copy Propagation: For x = y replace uses of x with y
- Dead Code Elimination: If side-effect free code can never be observed, safe to eliminate it
- Inlining: Replace a function call with the body of the function (arguments are rewritten)
- Code Specialization: Create Specialized versions of a function that is called form different places with different arguments.
- Common Subexpression Elimination: In some sense, it is the opposite of inlining: fold redundant computations together
- Loop Optimizations
  - Hot spots often occur in loops (esp. inner loops)
  - Loop Invariant Code Motion
  - Strength Reduction (replace expensive ops by cheap ones by creating a dependent induction variable)
  - Loop unrolling

Code Analysis

Algorithm for Dataflow Analysis:

```
  let w = new set with all nodes
  repeat until w is empty
    let n = w.pop()
    old_out = out[n]
    let in = combine(preds[n])
    out[n] := flow[n](in)
    if (!equal old_out out[n]),
      for all m in succs[n], w.add(m)
end
```

- Liveness
  - Observation: uid1 and uid2 can be assigned to the same register if their values will not be needed at the same time. Liveness property is more fine grained than scope.
  - Variable v is live at a program point L if v is defined before L and v is used after L
  - Liveness analysis is one example of dataflow analysis: 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
  - Dataflow: Compute information for all variables simultaneously. 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 nor more refinement is possible.

Liveness: backward, may $(out[n] = \bigcup_{n' \in succ[n]} in[n'], in[n] = gen[n] \cup (out[n] \setminus kill[n])), gen[n] = use[n], kill[n] = def[n]$
Reaching Definition: (for Constant and Copy propagation)forward, may $(in[n] = \bigcup_{n' \in pred[n]} out[n'], out[n] = gen[n] \cup (out[n] \setminus kill[n])), gen[n] = \{n\}, kill[n] = defs[a] \setminus \{n\}$
Available Expressions: forward (replace expression with res e.g. a = x + 1, b = x + 1 → b = a), must $(in[n] = \bigcap_{n' \in pred[n]} out[n'], out[n] = gen[n] \cup (out[n] \setminus kill[n])), gen[n] = \{n\} \setminus kill[n], kill[n] = uses[a]$
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, background, must $(out[n] = \bigcap_{n' \in pred[n]} out[n'], in = genunion(out \setminus kill))$
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

Register Allocation

Linear-Scan Register: Compute liveness information for each temp, try to use a reg if not possible than spill

- Register Allocation: Compute liveness information for each temp, create an infereence graph, try to color the graph.
- Kempe: 1. Find a node with degree ¡ k and cut it out of the graph, 2. Recursively k-color the remaining subgraph, 3. When remaining graph is colored, there must be at least one free color available for the deleted node. If the graph cannot be colored we spill that node.
- Precolored nodes: Certain variables must be pre-assigned to registers (call, imul, caller-save registers)

Coalescing Interference Graphs: A more aggressive strategy is to coalesce nodes of the interference graph if they are connected by move-related edges (if t1 and t2 in movq t1, t2 can be assigned the same register (color)) this move is redundant and can be eliminated
Brigg's strategy: It's safe to coalesce x & y if the resulting node will have fewer than k neighbors that have degree ¿= k.
George's strategy: We can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree ¡ k.

Loops

Identify loops & nesting structure in a CFG

- A loop is a strongly connected component (head reachable from each node)
- Concept of dominators: A dominates B = if the only way to reach B from start node is via A. A loop contains at least 1 back edge. (back edge = target dominates the source)
- dom is transitive and anti-symmetric, can be computed as a forward dataflow analysis
  $in[n] = \bigcap_{n' \in pred[n]} out[n'], out[n] = in[n] \cup \{n\}, \bigsqcap = \cap$

Single Static Assignment (SSA)

- %uids on RHS of phi nodes can be defined "later" in CFG
- An alloca inst. is promotable if the address of the variable is not taken or is not passed by reference
- ϕ nodes can be placed at dominator tree join nodes
- eliminate phi nodes after optmization

Algorithm For Computing Dominance Frontiers:

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

Garbage Collector

- Garbage: An object x is reachable iff a register contains a pointer to x or another reachable object y contains a pointer to x
- reachable objects can be found by starting from registers and following all pointers
- Mark and Sweep

  When memory runs out, GC executes two phases: mark phase: trace reachable objects; sweep phase: collects garbage objects (extra bit reserved for memory management)

  pointer reversal can be used to allow auxiliary data to be stored in the objects.
- Stop and Copy

  Memory is organized into two areas: Old space (used for allocation), new space (use as a reserve for GC)

  When old space is full all reachable objects are moved, old and new are swapped.
- Reference Counting

  Store number of references in the object itself, assignments modify that number. Cannot collect circular structures.