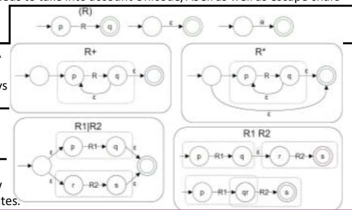
COMPILERS

Regex: Expressions to match strings, non-recursive.
Precedence (H → L) = grouping, repetition, concat, alternation.
Rules, Derived Rules
a = match symbol
\symbol = esc regex char
ε = empty string
R_1R_2 = match adjacent
$R_1|R_2$ = alternation
(R) = grouping
R+ = 1 or more reps
R* = 0 or more reps
R? = 0 or 1
. = match any char
.* = match every string
[abcd] = match any in set
[0-9] = match chars in range
[^abc] = match not in set

Lexical Anal: stream of chars → stream of tokens based on formal description of tokens of input lang. (Chars+Regex) → Tokens. **Identifier Tokens**: lex anal needs to id keywords quickly so uses fast string lookup using perfect hash func (no possible collisions). *Keyword Identifiers*: have special meaning in prog lang and normally rep by own token (class → CLASS, while → WHILE). *Non-Keyword Identifiers*: prog def identifiers e.g. var names, typically generic token used ("var1" → IDENT("var1"). **Literal Tokens**: const vals embedded in input prog. *Unsigned Ints*: literal token for ints – tokeniser needs to account for –ve ints and varying int sizes to prevent overflow ('123' → INTEGER(123), '1e400' → BIGINT(1e400)). *Unsigned Reals*: literal token for floating point vals, tokeniser must take into account large&-ve floats ('17.33' → FLOAT(17.33)). *Strings*: string tokens containing string, tokeniser needs to take into account Unicode/ASCII as well as escape chars ("hello" → STRING("hello")). **Other Tokens**: *Operators*: 1/2 char symbols (+ → PLUS, <= → LESSEQUAL). *Whitespace*: removed unless in string literal, may saved for nice errors ('pub mod' → [PUBLIC, MODULE], 'pubmod' → INDENT("pubmod")) *Comments*: removed. *Pre-Processing Directives and Macros*: normally removed in stage before lexical anal.



Lex Anal Impl: *Manually Impl*: **A** = relatively easy to write from scratch. **D** = more err prone, can be difficult to change rules for tokens. *Lex Anal Gens*: e.g. Logos which takes programmer defined token structures and functions to consume matching sequences by regex. Generate tokeniser. **Regex → NFA → DFA → Min State DFA (optimisation) → Transition Table + GetToken func.**

Disambiguation: when more than 1 expr matches, choose longest matching seq.

Finite Automata (FSM): Start state has transition to it, double circle = accepting state, each non-accepting state should have transition for every symbol (could be to error state) but omit for conciseness here. **NFA** (Non-deterministic Finite Automata): allow choice of transitions from each state. **DFA** (Deterministic Finite Automata): no two transitions leaving a state have the same symbol.

Subset Construction: NFA traversal requires backtracking (slow) – DFA traversal is faster (no BT) so convert NFA → DFA (remove ε trans). DFA needs more mem than NFA (up to 2^n states for n state NFA, always used as rare to be an issue). DFA states are subsets of NFA ones.

	Space	Time
NFA	$O(len\ R)$	$O(len\ R \times len\ X)$
DFA	$O(2^{len\ R})$	$O(len\ X)$

ε-Closure(s) = set of all states reachable from s with only ε transitions (inc s).

NFA → DFA: 1. ε closure of start state, 2. For each symbol, list states current state reaches, 3. Find ε closure (new state), 4. repeat with new state as start, 5. Mark states which inc accepting state as accepting states.

Chomsky Hierarchy: Hierarchy of grammars and machines required to parse them. **R** = non-terminal, **t** = token seq, **α+β+ϕ** = terminal+non-terminal seq. **Type 3 = R → t**, regular, DFA. **Type 2 = R → α**, Context Free, Pushdown automata (DFA w/ stack+read-only tape). **Type 1 = αRβ → αϕβ where R → ϕ**, Context Sensitive, Linear Bounded Automata (Turing machine w/ limited tape). **Type 0 = α → β**, Unrestricted/recursively enumerable, Turing machine. **regular ⊂ context-free ⊂ context-sensitive ⊂ recursively enumerable.**

Parsing (Syntax Anal) Types: CFGs are parsed with complexity O(n^3), LL and LR are subsets of CFG that can be parsed in O(n). LL(n) is subset of LR(n). Higher lookahead → more powerful+complex grammars at cost of performance (particularly mem). **LL** (build AST from root to leaves) → **LL(k)** = left-to-right scan, Leftmost derivation, k-token lookahead. **LL(0)** = doesn't exist. **LL(1)** = most popular (recursive descent or automaton impl). **LL(2)** = sometimes useful. **LL(k > 2)** = rarely needed. **LR** (build AST from leaves to root) → **LR(k)** = Left-to-right scan, rightmost derivation (in reverse) with k-token lookahead. **LR(0)** = weak, used in edu. **SLR(1)** = stronger than LR(0), superseded by LALR(1). **LALR(1)** = fast, popular comparable in power to LR(1). **LR(1)** = powerful, high mem usage. **LR(k > 1)** = possible, rarely used.

LR Parsers: Too complex to build by hand – need generator. Can be impl efficiently in O(n) time. **Parser Operation**: *shift (Sn)* = push state n onto stack and move to next token, *reduce (Rn)* = 1. use rule n to remove m states from stack (m = len of RHS of rule n), 2. get rule assoc with new top of stack (perform goto), 3. push back LHS of rule n, 4. generate new AST node for rule, *accept (a)* = parse successful, *error* = error, *goto (Gn)* = used in reduce case (not directly). **$** → for LR parsers an end of input symbol $ is always required (E' → E $) – if not present it's presence is implied.

LR(0): Rules in grammar with • (curr pos of parser). X → •ABC = init, X → A • BC (mid-way), X → ABC • (complete item – matched whole rule). Use LR(0) items as states in NFA (track progress of parser through rule).

$$X \to A \bullet BC \xrightarrow{B} X \to AB \bullet C$$
(current → new)

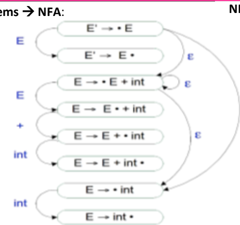
If B is non-terminal, for each rule B → •D add ε trans:

$$X \to A \bullet BC \xrightarrow{\epsilon} B \to \bullet D$$
(current → new)

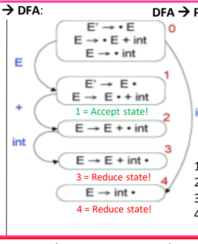
Making LR(0) Parser:
1. CFG → LR(0) items
2. Items → NFA
3. NFA → DFA
4. Generate parsing table from DFA

If there is more than one action for a cell in an LR(0) table, the grammar is not LR(0).

Grammar: **Items → NFA**:
E' → E
E → E '+' int
E → int
Items:
(A states inc…)
E' → • E
E' → E •
E → • E '+' int
E → E • '+' int
E → E '+' • int
E → E '+' int •
E → • int
E → int •



NFA → DFA:



DFA → Parsing Table:

State	ACTION			GOTO
	int	+	$	E
0	s4			g1
1		s2	a	
2	s3			
3	r1	r1	r1	
4	r2	r2	r2	

1. X → Terminal = s[new state]
2. X → Non-T = g[new state]
3. R' → … • = a (accept :D)
4. R → … • = r[rule to reduce]

First Set: Initially FIRST(N) = {}
1. FIRST(ε) = ε
2. FIRST(t) = t
3. N → $Y_1Y_2…Y_n$: FIRST(N) = FIRST(Y_1)
If ε ∈ FIRST(Y_1), FIRST(N) += FIRST(Y_2) etc

Follow Set: Initially FOLLOW(X) = {}
Only focus on RHS! FOLLOW *never* contains ε.
1. FOLLOW(Start) += {$}
2. N → X: FOLLOW(X) = FOLLOW(N)
3. N → X ε: FOLLOW(X) = FOLLOW(N)
4. N → X t: FOLLOW(X) = t
5. N → X $Y_1Y_2…Y_n$: FOLLOW(X) = FIRST(Y_1) (if ε ∈ FIRST(Y_1), FOLLOW(X) += FIRST(Y_2) then go to FIRST(Y_2) – if reach end apply rule 3).

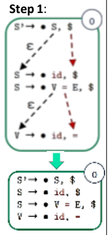
LR(1) Items: a pair **[LR(0) item, lookahead token t]**. Given item **[X → A • B C, t]**: **A** is top of stack, only recognise **B** if followed by a string derivable from **Ct**, **B** is top of stack only if followed by **Ct** if the **current token ∈ FIRST(Ct)**. In below, **u = FIRST(Ct)**.
NFA Transitions: Given **[X → A • B C, t]**: Add initial item for **[S' → • S, $]** (to start DFA construction)

$$[X \to A \bullet B\,C, t] \xrightarrow{B} [X \to A\,B \bullet C, t] \qquad [X \to A \bullet B\,C, t] \xrightarrow{\epsilon} [B \to \bullet D, u]$$

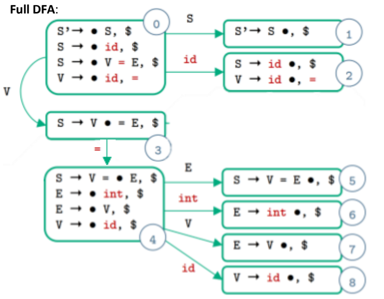
DFA Construction: 1. Write all initial states in one box and expand all non-terminals into their initial state in the box too, 2. all transitions out with diff symbols – remember above about changing lookahead, 3. for each new state expand non-terminals and repeat until reach reduction states.
Parsing Table: can now have several different rules per row (same state, diff curr token)

Grammar:
S → id
  | V '=' E
V → id
E → V | int
Expanded:
S' → S $
S → id
S → V '=' E
V → id
E → V
E → int

Step 1:



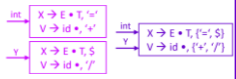
Full DFA:



State	ACTION				GOTO		
	id	int	=	$	E	V	S
0	s2					g3	g1
1				a			
2			r3	r1			
3			s4				
4	s8	s6			g5	g7	
5				r2			
6				r5			
7				r4			
8				r3			

Conflicts and Ambiguity (LR): Ambiguous grammar is one where more than 1 parse tree can be created for some input – cannot be LR(k) as parser won't know when to shift/reduce. Can be solved by rewriting grammar/augmenting grammar with additional rules. **Shift-Reduce Conflict**: cannot decide whether to reduce tokens to LHS of rule or shift another token, e.g. *Expr → Num* and *Expr → Num '+' Num*. Sol: *Expr → Num* and *AddExpr → Num '+' Num*. **Reduce-Reduce Conflict**: 2 rules which have same RHS – can't determine which to apply, e.g. *Expr → Expr '+' Expr | Expr '*' Expr | '(' Expr ')'*. Sol: 1. prioritise earliest rule from grammar source, 2. Use several rules: *Expr → Expr '+' Term | Term and Term → Term '*' Factor | Factor and Factor → '(' expr ')' | int*.

LALR(1) Parsers: Merge LR(1) states which have the same LR(0) items but differ in lookahead (red mem usage). However some reductions can occur before an error is detected (LR(1) would have picked up on them straight away). E.g.: **A → '(' A ')', A → '+'**. Input +) will be caught after shifting + in LR(1), but LALR(1) will shift + and do reduction A → + before catching error (if lookahead set for A → + is {$,)}).



LL Parsing: Top-Down using either recursive descent (can be hand-coded) or an LL(1) pushdown automaton (generated by LL(1) parser-generator (e.g. ANTLR))

LL(k) Grammar: If a k-token lookahead is sufficient to determine which alternative of a rule to use when parsing. LL(1) uses current token only. Can be impl as DFA or parse by recursive descent – we consider recursive descent.

LL(1) Grammar: For rule A → α | β if:
first(α) ∩ first(β) = ∅
∧ ε ∈ first(α) ⇒ (first(β) ∩ follow(A) = ∅)
∧ ε ∈ first(β) ⇒ (first(α) ∩ follow(A) = ∅)

EBNF: **{α}** = 0 or more occurrences of α, **[α]** = 0 or 1 occurrences of α, **(…)** = grouping elements. *Expr → Term '+' Expr | Term* becomes *Expr → Term ['+' Expr]*. Removing left-recursion: *Seq → Seq ';' Stat | Stat* becomes *Seq → Stat {';' Stat}*.

Recursive Descent Parser: Set of parse functions for each rule (take tokens, return remaining tokens + generated AST). Code to get next token, if match expected pop from list else throw error:
def match(expected: token):
    if lexical_analyser.next() == expected:
        lexical_analyser.pop()
    else: \n raise error("unexpected token!")

CFG → LL(1) Conversion: transformations must be applied to non-LL(1) CFG, which can't always be automated. Semantics must be maintained and readability is key.

Left Factorisation:

non-LL(1)	EBNF LL(1)	BNF LL(1)
$A \to B\ C \mid B\ D$	$A \to B\ (C \mid D)$	$A \to B\ X$ $X \to C \mid D$
$A \to B\ C \mid B$	$A \to B\ [C]$	$A \to B\ X$ $X \to C \mid \epsilon$

Substitution: sub a rule/non-terminal with it's alternatives (can find conflicts which need left-factoring).
A → B | C   A → "b" | "b" "c"
B → "b", C → "c"   A → "b" ["c"]

Left Recursion Removal: Can't be LL(1) with left recursion.

A → X | A Y => A → X {Y}
Expr → Expr ('+' | '-') Term | Term => Expr → Term {('+' | '-') Term}
Term → Term ('*' | '/') | Factor => Term → Factor {('*' | '/') Factor}
Factor → '(' Expr ')' | int => Factor → '(' Expr ')' | int

AST Construction: Class hierarchies organise nodes into variants of given types.

```
def scalartype(type, name): # check type is standard type
    return type == TOP_ST.lookup(name):
class IfAST extends AST:
    ExpressionAST E # Syntactic value
    tatementAST S1 # Syntactic value (then)
    StatementAST S2 # Syntactic value (else)
def check ():
    E.check() # E.check() also records the type of E
    if ! scalartype(E.type, "boolean"):
        error("if cond expression not a boolean")
    S1.check()
    if S2 != None: S2.check()
```

Statements Example:
Stat → IfStat | BeginStat
IfStat → 'if' Expr 'then' Stat ['else' Stat] 'fi'
BeginStat → 'begin' Stat {';' Stat} 'end'

```
def Stat() -> Statement:
  if next_token == IF: return IfStat()
  elif next_token == BEGIN: return BeginStat()
  else: raise error("Expected statement")
def IfStat() -> Statement:
  match(IF)
  cond = Expr()
  match(THEN)
  if_branch = Stat()
  else_branch = none
  if next_token() == ELSE:
    match(ELSE)
    else_branch = Stat()
  match(FI)
  return IfStatement(cond, if_branch, else_branch)

def BeginStat() -> Statement:
  stats = []
  match(BEGIN)
  stats.append(Stat())
  while next_token == SEMICOLON:
    match(SEMICOLON)
    stats.append(Stat())
  match(END)
  return Block(stats)
```

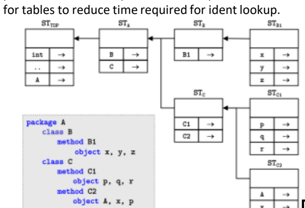
Error Recovery: Useful error messages, error recovery can be used to gen more errors (parsing or semantic).
Panic-Mode Error Recovery: each parse func gets extra param of synchronising tokens (syncset). More tokens added as more parse funcs called. When err, skip ahead discarding tokens until elem of syncset seen. Compiler must know when not to panic (phase-level error recovery). *Common heuristic*: add FOLLOW(A) to syncset for A (if A is in outer construct add FIRST of outer).

```
def match(expected):
  if token == expected:
    token = lexical_analyser.get_token()
  else print('unexpected token %s encountered at %s.
    Expecting %s', (token, token.position(), expected))
def skipto(syncset):
  while token not in syncset and token != EOF:
    token = lexical_analyser.get_token()
def check(expectset, syncset, errormessage):
  if token not in expectset:
    print(errormessage)
    skipto(expectset union syncset)
```

Type Checking in Languages: *Static v Dynamic Typing*: types checked at compile or runtime? *Strong v Weak Typing*: can types be easily coerced? *Type Inference*: can infer types at compile time. *Function Overloading*: multiple different funcs with same name in same scope. *Polymorphic Typing*: some langs allow types to be based off of interfaces/parent classes/generics. *Assignment-Compatibility and Type-Equivalence*: rules concerning how/what types are/can be matched. *Type Coercion/Casting*: some allow implicit, some explicit. *Primitive Sizes*: some langs allow multiple size ints/floats, some have set sizes, each lang differs in those set sizes/options. Python: dynamic, strongly typed. C: static, weakly typed.

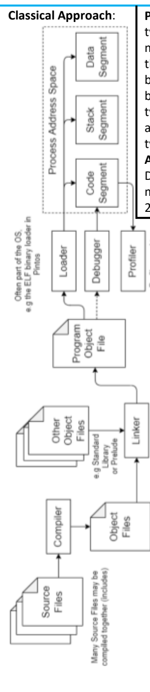
Semantic Analysis: checks statically at compile time if the prog is semantically valid within lang rules. Provides info for next phases (codegen) e.g. creates symbol table of identifiers and their types. Sem analysers can be gen from attribute grammars (assoc sem rules with AST node types) however usually hand-written.
Typical Checks: *Variable Decl*: type in scope, can you declare this type (e.g. can't decl void), check identifier (shadowing?). *Assignment*: is identifier valid (check scope whether declared, rules on variable types), check expression, can we assign to vals of this type, check range (e.g. u8 can't hold 256544). *Array Declaration*: check declared type valid (prev checks + can we make arrays of this type?), size (potentially only allow const, no negative sizes), scoping (shadowing?), array size warnings (some may be too large and cause stack overflow – do we warn about this?). *Array Elem Assignment*: is variable ident in scope, can we index the thing being treated as array, is index valid (in bound, positive), do we allow array slices? *Function Declaration*: check return type+params (as above), check func name declared (overloading/nesting?), scoping rules (what is parameter scope?), check returns (are there implicit returns, does body return correct type?). *Function Call*: check args are param compatible, check return type (is it assigned to var? if so is it matching type?).
AST + Coercion: can embed pointers to identifier objects in AST. When type checking, can use ad-hoc coercion rules (e.g. assign int to double). May want to use class hierarchy (if no match, check superclasses for matches). Can then put basic types, standard types, functions (etc) in top-level symbol table.

Symbol Tables: Instead of storing semantic info for identifiers in AST nodes (lengthy lookup during codegen) a structure is used to identify identifiers with semantic info. For scoping, use tree of symbol tables (could have flat table with idents translated to non-shadowed form – no name conflicts). Many langs idents declared before use so single pass on AST to build symbol table. Map data structures used for tables to reduce time required for ident lookup.




```
class SymbolTable:
  SymbolTable encSymTable # Ref to enclosing symbol table
  Dictionary dict # Maps names to objects
  def SymbolTable(SymbolTable st): # Create new symbol table
    dict = Dictionary(); # Initialise dictionary
    encSymTable = st; # Reference enclosing symbol table
  def add(name, obj):
    return dict.add(name, obj) # add name&obj to dictionary
  def lookup(name): # return object else None if name not
    return dict.get(name) # in dict
  def lookupAll(name): # Lookup name in current&enclosing
    S = self # symbol tables
    while S != None:
      obj = S.lookup(name)
      if obj != None: return obj # name found, return obj
      S = S.encSymTable # name not found, move to enclosing ST
    return None # name not found, return None
```

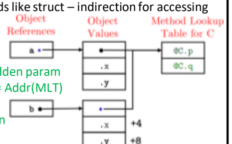
Classical Approach:



Primitive Types: most basic types (can be stored in mem/regs) supported by arch the compiler targets e.g. bool (1 byte) & float (IEEE 4, 8, 16 bytes). Compilers may align types for more optimal mem access (sometimes enforced by target arch). Some langs support pointer types (makes type checking difficult). More modern langs use type-checkable object refs.
Alignment: some addrs are considered n-byte aligned when it is a multiple of n bytes. Data aligned by size of data processor can load in single instr (ensures min number of mem accesses). Some archs enforce alignment (particularly for stack). When aligned to 2^k, first k bits are 0 (can be used for other info in PTEs e.g. supervisor bits, R/W etc).
Structs/Records: data type consisting of groups of fields/members of different types. Fields typically allocated contiguous mem blocks, alignment spaces fields, some order fields by order in code, some optimise order, tuples effectively anonymous structs. *Record* = $(F_1, …, F_n)$. *Size(Record)* = *Size*(F_1) + … + *Size*(F_n). *Addr*(F_k) = *StartAddr(Record)* + *Size*(F_1) + … + *Size*(F_{k-1}). **Arrays**: contiguous section of memory populated by n variables (elems) of same type.
Can have aligned elems, some langs associate auxiliary data (e.g. size) with the arr. *Arr[Type]* = *Elem[Type]*$_1$, …, *Elem[Type]*$_n$ (indexed 0 → n – 1). *Size(Arr)* = *Size(Type)* * n. *Addr*$(Elem_k)$ = *StartAddr(Arr)* + k * *Size(Type)*. **Objects**: impl as reference to record with pointer to method lookup table (MLT) for the class (need to consider inheritance). When calling method of obj, traverse MLT then jump to method after placing first arg as pointer to obj. Access fields like struct – indirection for accessing vals/calling methods adds overhead.

```
class C {
  int x
  int y
  method p ...
  method q ...
}
C a = new C()
C b = new C()
```

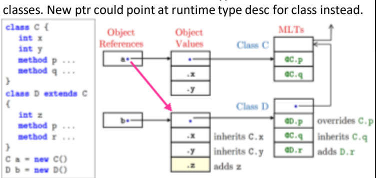
ASM Method Call b.q():
mov eax, <b>
push eax   # objref = hidden param
mov eax, [eax]   # eax = Addr(MLT)
call [eax + 4]
call q at byte offset 4 in MLT for class C



ASM (interface) f = (class) b:
# static check suffices to check
# b impls F
mov [ecx], [ebx] # b's obj ref
# into 1st field of f
mov [ecx+4], @B.F # copy addr
# of B's F interface MLT to
# second field of F

ASM a = (A) f [casting]:
# check f's object in A's class
mov [edx], [ecx] # edx will traverse
# MLT hierarchy
CheckClass:
mov edx, [edx] # get MLT addr
cmp edx, @A # is MLT class A's?
je Okay
cmp edx, NULL
je ClassError
jne CheckClass # if not err,
check superclass
Okay:
Mov [eax], [ecx] # f refers to an A
# obj. Copy 1st field of f into 'a'

Class Inheritance: Chained MLTs allow type-tests for inherited classes. New ptr could point at runtime type desc for class instead.



Dynamic Binding: Only append fields+entries to MLT when interiting – can set var of type parent class to val of type child class. Note original "a" now inaccessible.

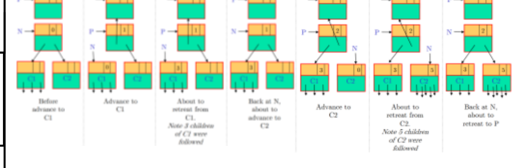
Program Address Space: **[Code | Stack | Data@[Static (Global Vars) | Heap]]**. Separating into diff pages means each can have different perms (e.g. code R/X only). *Static area*: globals (in fixed locations), MLTs, constants. *Stack*: args sent to func, local vars, addr of obj for methods, EBP = start of stack reg (frame pointer reg). **Heap**: Can alloc vars (aka heap/dynamic vars), can remain here for any period of time and be ref'd from any point in the prog (*new* in Java or *malloc* in C). Managed in 2 ways: *Explicit*: programmer decides when mem freed (*free* in C). GC: compiler checks when objs no longer in use and frees them (Java). **Explicit Heap Alloc**: maintain structure containing free blocks+info about them (free list, impl as linked list). Several lists for diff sizes reduces search time, alloc blocks just large enough, alloc with expectation more mem will be used (if needed), must consider alignment of vals if required, heap alloc worse for locality (worse cache locality).

Profilers: Give performance info on prog – interr func periodically and record info about methods/variables e.g. time spent in diff funcs. Typically used once code is correct, optimised by compiler, and algorithmically optimal.

Garbage Collection: ensures alloc vars are de-alloc'd when no longer used. Reqs: correctness (don't dealloc live data), performance (fast + low mem overhead to red impact on prog performance), compiler supported (c must provide info for which vars point to heap and pointers in each block via special subroutines for GC or as part of type desc for obj data). Can be run 1-shot (pause prog), on-the-fly (as prog is running) or concurrent (diff thread). **Heap Compaction GC**: after GC deallocs many blocks, heap may be fragmented so mark live blocks, merge free blocks, then update pointers. **Reference Counting GC**: block management housekeeping info contains ref count (ref made, ref++, when ref removed, ref--). **A** = simple+efficient (GC ad prog runs). **D** = compiler must gen code to correctly track refs, in ref cycles blocks kept alive indefinitely. **Mark-Sweep GC**: pause prog + collect blocks Not pointed to by non-heap val. **A** = Finds dead blocks/garbage (unlike ref count). **D** = overhead, each block needs space for mark bit. *Mark*: for all stack vars, mark heap mem they point to as live. *Sweep*: go through blocks, if not marked live, collect+dealloc. **Two-Space GC**: heap split into from- and to-space. Blocks alloc from "from" – when no more free all live blocks moved to "to", then spaces swap (from → to, to → from). **A** = few complex ptr manips, automatically compact mem when copying, can place linked blocks close together when copy (better cache locality). **Generational GC**: Heap split into several areas based on age of blocks. Alloc new blocks from youngest, if full move oldest young blocks to older area. GC used more freq on younger areas, can apply diff GC to diff areas.

```
def New(size):
  if FreeList == None:
    Mark_Sweep()
  return Allocate(size)
def Mark_Sweep()
  foreach GlobalorLocal heapRef Var:
    Mark(HeapRef.block) # as live
  Sweep()
  if FreeList == None:
    Error("Out of memory")
def Mark(Block):
  if Block.MarkBit == 0: # 0=not marked
    Block.MarkBit = 1     # 1=marked(live)
    foreach ChildRef in Block.Children:
      Mark(ChildRef.Block)
def Sweep():
  Block = HeapStart
  while Block < HeapEnd:
    if Block.MarkBit == 0:   # dead block
      Deallocate(Block)      # reclaim
    else Block.MarkBit = 0   # reset
    Block = Block + Block.size
```

Pointer-Reversal Marking: recursive traversal of pointers requires a lot of mem – GC must work when mem low. By storing previously visited in the ptr we are traversing through we require no extra stack space. When find already marked block, use reverse ptrs to backtrack.



Debugger: *Post-Mortem*: Core dump. Reverse PC to source line → stack traceback returns method name, vals of local vars, name of calling method, it's locals, params. Contents of global/dynamic vars (vars may be in regs at various times – debugger keeps track over time). *Interactive Debugger*: for natively compiled progs, need support from arch/OS to set breakpoints on machine instrs. Don't affect program behaviour (besides execution time). Require info about idents and mapping PC to source lines. Optimising compilers and debuggers rarely work together.

ASM for Switch Statement: Ass no states fall through:

```
code for expression
# assume result is in R1. If constant expression then
# just emit instructions for corresponding statement
# jump to statement if expression equal to label x:
foreach case_label x do
    CMP R1, x
# jump to STATEMENT<n>, corresponding to label x:
    JEQ STATEMENT<n>
Endfor
# jump to suitable error message and exit:
jmp CASE_ERROR
# statements for case clauses:
foreach STATEMENT<n> do
# label for STATEMENT<n> instructions:
    STATEMENT<n>:
    code for STATEMENT<n>
# not needed for last STATEMENT<x>:
    JMP ENDCASE
endfor
ENDCASE:
# assembly instructions for statements
# following switch statement
```

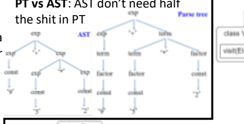
ASM for Jump Table:

```
code for expression # result in R1
cmp R1, lowest_label # check expression is in range
jlt CASE_ERROR # jump to suitable error message and exit
cmp R1, highest_label
jgt CASE_ERROR # jump to suitable error message and exit
sub R1, lowest_label # change expression to 0..high-low ignoring overflow
mul R1, 4 # change to address offset (if 4-byte addresses)
jmp [R1 + JUMP_TABLE] # jump to statement
JUMP_TABLE: # addresses of statements
Address of statement code for lowest_label
...
Address of statement code for highest_label
# rest as to the left (from line 7)
```

Compilers: particular class of programs called language processors. Cs can be: **Processes**: programs written in some lang. **Writes**: prog which writes progs. **Translates** progs from one lang to equiv in other. **Basic Structure**: input → analysis (create internal rep, do lexical, syntax, and semantic anal) → synthesis (use IR to create program in target lang – intermediate codegen → optimisation) → output.

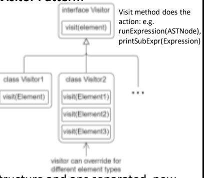
Parse Tree: How string derived from start symbol – graphical proof that a sentence is within a grammar. Can express grammar as tuple **G(S, P, t, nt)**: **S** = start symbol, **P** = productions, **t** = terminals, **nt** = non-terminals, and S ∈ nt. If S is entirely ts, use productions+pattern matching to anal. **Sentence**: string derived only consisting of ts. **Language**: set of all sentences which can be derived from S.

PT vs AST: AST don't need half the shit in PT



Syntax: Grammatical structure of lang expressed through rules. **Semantics**: meaning assoc with prog.

Visitor Pattern:



A = data structure and ops separated, new ops easily added by creating+passing new visitor which is able to op on structure. E.g. turtle ops on AST of turtle program (visitors for colour/languages etc.)

```
data Stat = Assign name Exp |
            Seq Stat Stat |
            ForLoop Name Exp Exp Stat
data Exp = Binop Op Exp Exp |
           Unop Op Exp |
           Ident Name |
           Const Int
data Op = Plus | Minus |
          Times | Divide | Minus
type Name = [Char]
```

```
data Instruction
 = Add | Sub | Mul | Div            (as before)
 | PushImm Int    (push constant onto stack)
 | PushAbs Name   (push variable at given location onto stack)
 | Pop Name       (remove of top of stack & store it at given loc'n)
 | CompEq         (subtract top two elements of stack, and
                   replace with 1 if the result was zero, 0 otherwise)
 | JTrue Label    (remove top item from stack; if 1 jump to label)
 | JFalse Label   (jump if stack top is 0)
 | Define Label   (set up destination for jump)
```

```
transStat (Assign id exp)
  = transExp exp ++ [Pop id]
transStat (Seq s1 s2)
  = transStat s1 ++ transStat s2
transExp :: Exp -> [Instruction]
transExp (Binop op e1 e2)
  = transExp e1 ++
    transExp e2 ++
    transOp op
transExp (Unop op e)
  = transExp e ++
    transUnop op
transExp (Ident id) = [PushAbs id]
transExp (Const v) = [PushImm v]
transOp Plus = [Add]
transOp Minus = [Sub]
transOp Times = [Mul]
transOp Divide = [Div]
transUnop Minus = [Negate]
```

```
transStat (ForLoop id e1 e2 body)
 = transExp e1 ++ [Pop id] ++
   [Define label1] ++
   transExp e2 ++ [PushAbs id] ++ [CompGt] ++
   [JTrue label2] ++
   transStat body ++
   [PushAbs id] ++ [PushImm 1] ++ [Add] ++ [Pop id]++
   [Jump label1] ++
   [Define label2]
```

Unbounded:

Stack slot:	3	2	1	0
PushImm 100,				100
PushImm 3,			3	100
Mul,				300

Use regs where things end up on stack → more efficient use. TransExp needs param r to put res in (tracks SP too by above if stack slots = regs).

```
transExp (Const n) r = [LoadImm r n]
transExp (Ident x) r = [Load r x]
transExp (Binop op e1 e2) r
  = transExp e1 r ++
    transExp e2 (r+1) ++
    transBinop op r (r+1)
where
  transBinop Plus r1 r2 = [Add r1 r2]
  transBinop Minus r1 r2 = [Sub r1 r2]
```

Improvement with Immediate Operands: transBinOpImm just calls AddImm instead of Add etc.

```
transExp (Binop op (Const n) e2) r
  | commutative op = transExp e2 r ++ transBinopImm op r n
transExp (Binop op e1 (Const n)) r
  = transExp e1 r ++ transBinopImm op r n
where
  transBinopImm Plus r n = [AddImm r n]
```

Accumulator Machine: 1 reg (acc) to which ops can be applied

Instruction	Acc	Stack 1	Stack 0
LoadImm 500	500		
MulImm 3	1500		
AddImm 400	1900		
Push	1900		1900

```
Add:
  Acc:=Acc+Store[SP]; SP:=SP+1;
Push:
  SP:=SP-1; Store[SP]:=Acc;
Pop:
  Acc:=Store[SP]; SP:=SP+1;
Load name:
  Acc:=Store[name];
Store name:
  Store[name]:=Acc;
```

```
transExp (Const n) = [LoadImm n]
transExp (Ident x) = [Load x]
transExp (Binop op e1 e2)
  = transExp e2 ++
    [Push] ++
    transExp e1 ++
    transBinop op
where
  transBinop Plus = [Add]
```

Note e2 must be eval before e1 so it forms the RH op of the binop. For e1 + e2, push e2 onto stack while eval e1

Machine with Limited Regs: When free regs remain, use RM strat, when limit reached (1 reg left) use acc strat.

Enough regs (arith instr taking reg ops):

```
transBinop Plus r1 r2   = [Add r1 r2]
transBinop Minus r1 r2  = [Sub r1 r2]
```

Acc strat (use AddStack):

```
transBinopStack Add r = [AddStack r]
transBinopStack Sub r = [SubStack r]
```

The **AddStack r** instruction is simply:
r := r+Store[SP]; SP:=SP+1;

```
data Instruction =
  Add reg reg          (Add r1 r2:  r1:=r1+r2)
 | Sub reg reg | ...   (similar)
 | Load reg name       (reg := value at location name)
 | LoadImm reg num     (load constant into reg)
 | Store reg name      (store reg at location name)
 | Push reg            (push reg onto stack)
 | Pop reg             (remove value from stack & put it in the reg)
 | CompEq reg reg      (subtract reg from reg & set reg to 1
                        if the result was zero, 0 otherwise)
 | JTrue reg label     (if reg = 1 jump to label)
 | JFalse reg label    (if reg = 0 jump to label)
 | Define label        (set up destination for jump)
```

Register Use Data Type:

Acc Machine same type just without regs

```
transExp (Const n) r = [LoadImm r n]
transExp (Ident x) r = [Load r x]
transExp (Binop op e1 e2) r
  = if r == MAXREG then
      transExp e2 r ++
      [Push r] ++
      transExp e1 r ++
      transBinopStack op r
    elseif r < MAXREG
      transExp e1 r ++
      transExp e2 (r+1) ++
      transBinop op r (r+1)
```

If no more registers, use stack for e2 while evaluating e1

Arithmetic instruction gets one operand from stack (next slide)

If there are more registers, use register r to hold e1. Evaluate e2 leaving result in register r+1

Arithmetic instruction gets both operands from registers

```
data Statement
  = Assign name expression |
    Compound [Statement] |
    IfThen expression Statement |
    IfThenElse expression Statement Statement |
    While expression Statement
```

```
translate_statement :: Statement -> [Instruction]
translate_statement (Compound stats)
  = foldr (++) [] (map translate_statement stats)
translate_statement
  (IfThenElse exp thenbody elsebody)
  = translate_expression exp ++
    [JFalse elselabel] ++
    translate_statement thenbody ++
    [Jump endlabel] ++
    [Define elselabel] ++
    translate_statement elsebody ++
    [Define endlabel]
  where
    endlabel = newlabel
    elselabel = newlabel
```

```
[Pop var]
translate_expression exp ++
  translate_expression cond_exp ++
  [JFalse label] ++
  translate_statement body ++
  [Define label]
where    (assume we have func newlabel)
  label = newlabel
translate_statement (While exp body)
  = [Jump condlabel] ++        "rotated" while loop
    [Define startlabel] ++
    translate_statement body ++
    [Define condlabel] ++
    translate_expression exp ++
    [JTrue startlabel]
```

Register Usage: regs fast R/W, multi-ported (2+ can be read per clock cycle), specified by small field of instr (room for imm operands+other data). CPU can optimise at runtime and use reg access+data deps to optimise instr ordering. Hence should aim for minimal reg/other mem hierarchy use. **x + (3 + (y * 2))** uses 4 regs, **((y * 2) + 3) + x** uses only 2.

Sethi-Ullman Weights: given E_1 and E_2, always eval subexpr which uses most regs first. When second expr is eval it has 1 less reg (as res of first must be stored). **E_1 first: max(E_1, E_2 + 1), E_2 first: max(E_1 + 1, E_2)**. Non commutative ops need order maintained. Use weight func to get order.

Register Targeting: To avoid clobbering, give transExp list of regs it can use (it leaves res in first).

Action	Registers
Evaluate A	n_A
Result of A stored in a reg	1
Evaluate B while storing result of A	$n_B + 1$
Result of B stored in a reg	2
Operate on subexpression results	1

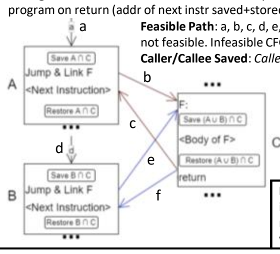
```
weight :: Exp -> Int
weight (Const n) = 1    Ass loading
weight (Ident x) = 1    const/var into reg
weight (Binop op e1 e2 )
= min [cost1 , cost2 ]
  where
    cost1 (if we choose to do e1 first)
      = max [weight e1, (weight e2)+1]
    cost2 (if we choose to do e2 first)
      = max [(weight e1)+1, weight e2 ]
```

```
transExp :: Exp → [Register] → [Instruction]
transExp (Const n) (destreg:restofregs)
  = [LoadImm destreg n]
transExp (Ident x) (destreg:restofregs)
  = [LoadAbs destreg x]
transExp (Binop op e1 e2) (dstreg:nxtreg:regs)
= if weight e1 > weight e2 then
    transExp e1 (dstreg:nxtreg:regs) ++
    transExp e2 (nxtreg:regs) ++
    transBinop op dstreg nxtreg
  else
    transExp e2 (nxtreg:dstreg:regs) ++
    transExp e1 (dstreg:regs) ++
    transBinop op dstreg nxtreg
```

Effectiveness: worst case = perfectly balanced tree (uneven = always eval in order which reduces reg demand). k vals, k/2 – 1 ops, roof($\log_2$k) regs required. Worst case: N regs support 2^N terms (does not account for reused cars/vals). Most comps use this + GC as well. **Limitations**: fails to exploit context of generated code (no attempt to use regs to keep val from stat-to-stat, no use of regs to store vars, doesn't handle repeated use of vars or intermediate vals in computation).

Register Alloc for Func Calls: need to know where params are when passed (stack/regs). Changing eval order can change res due to side effects. Need register targeting. At point of call, several regs may already be in use (can be diff from diff call sites.) **Function Call Eval Order**: C++, order undefined, Java order L→R. eval order depends on context (need to watch calling convention).

Calling Convention: funcs can be called in many diff places (call sites), must go to correct pos in program on return (addr of next instr saved+stored).

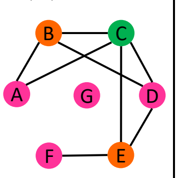


Feasible Path: a, b, c, d, e, f, g. **Infeasible Path**: a, b, f, g – valid in graph but not feasible. Infeasible CFG problem becomes difficult with many call sites. **Caller/Callee Saved**: *Caller*: caller saves regs it is using to preserve them (callee can't clobber). *Callee*: callee saves regs it needs to use. Both can end up saving redundant regs as they don't know which regs the other may use (esp problem in separate compilation of libs etc). Some make reg preservation decisions at runtime, others define *Application Binary Interface* to ensure linked lib calls save right regs.

Moore's Law: every 2 years, density of transistors in integrated circuits doubles. This is slowing down recently. More optimised compilers are now the way forward for progress.

Graph Colouring: **1. Use simple traversal to generate intermediate (3AC) code.** Named locations for all temporaries allow for considering all vals including intermediates. **2. Construct interference graph.** Each node is temp location, edges connect simultaneously live locations. **3. Attempt to colour nodes.** Regs which simultaneously store vals (directly connected) must be diff colours. **Spilling**: when colouring not possible. **1. Find an edge connecting problematic var and break it.** Done by adding code to store it to mem and reload later. **2. Redo analysis to see if it works now.** Avoid adding spill code to innermost loop (prioritise values by nesting depth). Split live range that will allow colouring.

```
S1:   A = 100;
P1:
S2:   B = 200;
P2:
S3:   C = A + B;
P3:
S4:   D = A * 2;
P4:
S5:   E = B * 2;
P5:
S6:   F = D - C;
P6:
S7:   G = E + F;
P7:
```



Optimisations: *Without*: program is slower but easier to debug (more direct translation). *With*: opts such as loop hoisting, constant propagation, inlining, and smart reg alloc makes prog execute quicker but debugger less useful as statements reordered/removed. When using opts, don't re-impl things like memcpy (already have opts done behind the scenes). **High-Level Optimisations**: Using HL info encoded in program (types, function anal). E.g. *Function Inlining*: replace call site with copy of func body. **A** = avoid call/return overhead, creates further opt opportunities. **D** = can require static anal (calling virtual/overloaded func requires type info). **Low-Level Optimisations**: Use LL info (instr types, ISA, order of IR instrs) to opt output. Note can lose HL info as we get lower. E.g. *Instr Sched*: in pipelined arch, instr order can impact processing speed so reorder instrs to allow parallel processing. **D** = requires some dependency anal. **Peephole Optimisation**: scan through assembly in order, looking for obvious cases to opt. Can catch some worst cases (e.g. store followed by load of same location). Very easy to impl (at smallest, just consider 2 adj instrs). **D** = phase ordering problem – what order should ops be applied to get best res?

Local Basic Blocks Fast+Easy to Validate

Global Procedures

Intraprocedural Whole Program Slow, very complex + rare

Local	Optimisation on the level of basic blocks (single entry adn exit points) (e.g expressions)	Runs quickly and easy to validate.
Global	Optimisation on the scale of a whole procedure	Can have worse than $O(n)$ (Linear) complexity given n represents instructions, basic blocks or variables.
Intraprocedural	Optimisation over the whole program	Rare and hard to avoid excessive compile times.

Interference Graphs from Live Ranges: Use the LiveOut sets from each block to construct interference graph:

[(T0,[T0,T1,T2,T3]), (T3,[T0,T3,T2,T1,T4]),
(T1,[T1,T0,T2,T3,T4]), (T4,[T3,T4,T2,T1])]
(T2,[T1,T2,T0,T3,T4]), Then do graph colouring :)

Loop Optimisations: **Loop Invariant Code Motion**: an instr is loop-invariant if all operands arrive from outside the loop (moving it out doesn't change the semantics).

```
for (int b = 0; b < 4; b++) { int c = a + 3 * 2 }
→ int c = a + 3 * 2; for (int b = 0; b < 4; b++)
```

Strength Reduction: replacing a complex op with a smaller, simpler one. An induction var is one which inc/dec by a loop invariant amt each iteration. We can use less costly instrs to compute their value.

```
int a = 7; for (int b = 0; b < 4; b++) { a += 4 }
→ int a = 23; for (int b = 0; b < 4; b++)
```

Note: 23 = 7 + 4 * 4

```
while (some_pred) {
    b = some_func(b);
    for (int i = 0; i < b; b++) {
        a++
...
→ while (some_pred) {
    b = some_func(b);
    a += b;
...
```

Control Var Selection: replace loop control var with one of the induction vars used. Exit cond for loop must be changed accordingly. E.g.:

```
int *a = malloc (15 * sizeof(int));
for (int i = 0; i < 15; i++) {
    a[i] = 5;
}
→ int *a = malloc (15 * sizeof(int));
int *end = a + 15;
for (   ; a != end; a++) {
    *a = 5;
}
```

Dead Code Elim: code that does not produce a result can be eliminated. Many other opts result in dead code (e.g. inlining func where not all return vals/optional args used).

Intermediate Representation: an IR is used in code synthesis+opt. Must rep all primitive ops needed for exec, be easy to analyse+manip, indep of target instr set (allowing many diff ISAs to be targeted by backend using IR), use temp vars to store intermediate vals to allow for opts+reg assignment. **Lowering Rep**: taking HL features → LL reps, e.g. array → pointer arith/addr calcs. When lowering, lose HL info e.g. values which are part of arr, but can opt LL rep (addr calcs). Opt compiler synthesis can have several IRs + anal stages: IR $\xrightarrow{\text{Analyse, Optimise}}$ IR $\xrightarrow{\text{Analyse, Optimise}}$ IR $\xrightarrow{\text{Analyse, Optimise}}$ IR

Data Flow Analysis for Live Ranges: Live range = range of instrs for which temp val must be maintained. Live range starts at def, ends when var is used/immediately if val never used. **CFG**: Can generate graph of simple IR (ideally 3AC) instructions in prog: → [Instr num | Instr | Regs used | Regs def | Successors] →. **Live Range Defs**: *point* = any location between adj nodes, *path* = seq of points traversing through CFG, *live var (live-out)* = if the var may be used along some path going through the point (can be found via DFS but we want faster method).

LiveIn[B] = set of live vars at beginning of B

$$LiveIn(n) = uses(n) \cup (LiveOut(n) - defines(n))$$

LiveOut[B] = set of live vars at end of B

$$LiveOut(n) = \bigcup_{s \in succ(n)} LiveIn(s)$$

Uses/Gen[B] = set of vars used in B (on RHS but not LHS of *prior* statement *in* B).

Defines/Kill[B] = vars assigned value in B (LHS).

Iterative Steps for LiveIn and LiveOut:

1. Find use and def sets for each node
2. Iter 1: LiveIn= uses, LiveOut as normal (U LiveIn)
3. Rest of Iterations: use previous values to get new LiveIn/LiveOut sets
4. Stop when sets stop changing :)

Improvements to Iterative Methods: To reduce req iters, best to update nodes last as defs depend on succs. We consider it a *backward anal*. Time complexity dep on num nodes/instrs/regs.

Time Tutorial Q: Assuming all ops take unit time 1 and infinite time is for nodes never reached:

timeOut(n) = timeIn(n) + 1,

timeIn(n) = $\min_{p \in pred(n)}$ timeOut(p)

Use iteration to solve system of DFA equations, initial assignments of timeIn/timeOut are infinity. The entry node needs to be init as 0 however (remember to add a start node to CFGs!!)

This is a forward dataflow analysis as info is propagated from start forwards (timeIn depends on predecessors not successors).

```
   start:
1:     a = b + c
2:     d = a < 10        ← Code
   L0:
3:     if d goto L1 else L2
4:     b = a - 1
   L1:
5:     a = f(d,e)
6:     if a goto L0 else L2
   L2:
7:     a = b
end
```

↓ CFG (as table)

Node id	Instruction	Uses	Defs	Succs	Preds
0:	start			1	
1:	a = b + c	b,c	a	2	0
2:	d = a < 10	a	d	3	1
3:	if d goto L1 else L2	d		5,7	2,6
4:	b = a - 1	a	b	5	
5:	a = f(d,e)	d,e	a	6	4,3
6:	if a goto L0 else L2	a		3,7	5
7:	a = b	b	a	8	3,6

Available Expressions Tutorial Q: Available expressions allow for common sub-expression elimination.

Define a set of all expressions in the prog:

$$U = \{a + b, d + e, g + h, x * 2, \}$$

Know that the following must hold:

$$AvailIn(n) = \bigcap_{p \in preds(n)} AvailOut(p)$$

$$AvailOut(n) = gen(n) \cup (AvailIn(n) - kill(n))$$

We know that: $AvailOut(0) = \emptyset$

Node 0 = entry point of the prog's CFG.

Iteration:

$AvailOut(0) = \emptyset$
for $n \in N - \{0\}$
  $AvailIn(n) = AvailOut(n) = U$
do {
  for $n \in N - \{0\}$
    $AvailIn(n) = \bigcap_{p \in preds(n)} AvailOut(p)$
    $AvailOut(n) = gen(n) \cup (AvailIn(n) - kill(n))$
} while any $AvailOut(n)$ changes

Note we init entry point differently (all others start with all possible exprs). Iteration causes avail set to shrink (find largest set satisfying eqns). This is forward analysis.

Loop Invariant Code Motion: **Loop Invariant**: d is loop-inv if every def of a u_i ∈ uses(d) that reaches d is outside the loop (all vals it uses unchanging in loop). Hence it's val is loop-inv so can move d out. **Finding Loop-Inv Instrs**: attempt to find nodes of the form [Binop | Copy op | Const]. **Reach**: def d reaches p if there is a path d → p where d is not killed. For node n we have several sets: **Gen(n)** = {n} = the set of defs gen by the node. **Kill(n)** = Set of all defs of t except for n. **ReachIn(n)** = Set of defs reaching up to n. **ReachOut(n)** = Set of defs reaching after n. Calculate sets as follows (same iter pattern):

$$ReachIn(n) \triangleq \bigcup_{p \in Pred(n)} ReachOut(p)$$

$$ReachOut(n) \triangleq Gen(n) \cup (ReachIn(n) \setminus Kill(n))$$

This is forward anal as defs reach forwards. For reaching defs can reduce each set to relevant reaching defs but consider-ing only ReachIns that are actually operands of instr. Note here that 4's set does not inc 6 (the loop cond check).

	Reaching def (ReachIn):
1: x=1	1: []
2: w=100	2: [1]
3: z=200	3: [1,2]
4: x=x+1	4: [1,2,3,4,5]
5: y=w+z	5: [2,3,4,5]
6: if (x<10)	6: [2,3,4,5]

Identifying Loops: given set S of nodes which are part of a loop: there is a single header node h ∈ S. There is a path from h to any other node in S. There is a path from any node in S to h. All non-loop nodes can only *directly* connect to h (and no others in S). **Dominator**: node d dominates n if every path from start node to n goes through d (every node dominates itself). To find nodes which dominate curr node: 1. Set Dom to set of all nodes, 2. Apply rules (as below) as start node has set {start} and will propagate, reducing other set sizes, 3. Once sets stop changing stop. $Doms(start) = \{start\}$

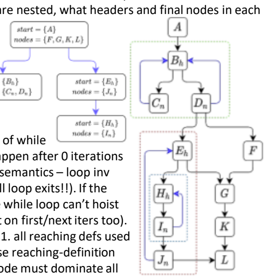
$$Doms(n) = \{n\} \cup \left(\bigcap_{p \in preds(n)} Doms(p)\right)$$

Back Edges: edge from n → h where h dominates n as back edge. **Natural Loop**: natural loop of a back-edge (n, h) is the set of nodes S s.t.: all nodes x ∈ S are dominated by h, and for all x there is a path x → n which does not contain h. This represents a loop with header node h.

Multiple Loops: Often 2 natural loops will arise from one source code loop (still consider this as 1 loop). When we have nested loops, nodes of inner loop are a proper subset of those in the outer loop. **Control Tree**: construct tree to show which loops are nested, what headers and final nodes in each loop are.

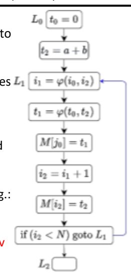
Hoisting Instructions:

Pre-Header: A node inserted imm-ediately before the header node of a natural loop.



Caution: When pulling out of while loop, consider what will happen after 0 iterations (pulling out likely changes semantics – loop inv node does not dominate all loop exits!!). If the value is altered later in the while loop can't hoist (second maybe dependent on first/next iters too). **Conditions for Hoisting d**: 1. all reaching defs used by d occur outside loop (use reaching-definition anal for this), 2. loop inv node must dominate all loop exits as pre-header also does (use dominators anal for this), 3. only one definition of t (count the definitions), 4. t cannot be LiveOut from loop's pre-header (if it is then t is used in loop – use live range anal). **Basic Hoisting Process**: 1. compute dominance sets for each node, 2. use dom sets to id natural loops+headers, 3. compute reaching sets for nodes, 4. use relevant reaching defs to id loop-inv code, 5. attempt to move to pre-header, 6. check semantics not altered. Can be repeated process.

Single Static Assignment (SSA): Representation avoiding side conditions by only allowing single ass to each temporary (temps also immutable). **A** = simplifies many opt probs, makes many opt passes more efficient, widely used in compilers. **D** = requires potentially complex transition to+from SSA. Each reassignment being renamed splits live ranges, so each temp has only one reaching def. **Phi Function**: Used for branching - $\varphi(a_1, a_2)$ means either a_1 or a_2 could be used. By renaming vars we elim many loop hoisting issues (particularly for variable re-def in loops). Phi has no direct translation in programs. E.g.: Once in SSA we re-evaluate **hoisting requirements**: 1. All reaching defs used by d occur outside loop (same as prior to SSA, reaching def anal), 2. Loop inv node must dom all loop exits (no longer issue), 3. Only 1 def of t (guaranteed by SSA), 4. t cannot be LiveOut from loop pre-header (cannot occur with SSA due to single ass).



Other Compiler Optimisations: Conventional compilers reduce work done at runtime, but restructuring compilers determine optimal order to do computation, e.g. adding parallelism to loops. Things: Induction Variables, Strength Reduction, Induction variable Elimination, Rewriting Comparisons, Array Bounds Check Elimination, Common Sub-Expressions, Partial Redundance Elimination, Loop Unrolling, Inlining, Dead Code Elimination Rematerialisation, Tail-Call Optimisation. **Higher-Level Optimisations**: subtype polymorphism (may be able to determine method called at runtime so can inline), pattern matching (compiler determine optimal order of tests that still respects matching order semantics), mem management, lazy eval (moving eval of exprs to their uses so no redundant comp done), arrays (optimising overloads and allow array slicing without lots of data copying).

Points-To Anal Tutorial Q: in program with *new* or pointer vars/assignments we want to know what each pointer var in the prog might point to. Each *new* in the code corresponds to an allocation site (the id of the CFG node where the alloc occurred). The PointsIn(n) for a node n is a set of pairs {(reg,id)...}, indicating that just before n, reg might point to a heap cell allocated at node id. The effect of each instr on the points-to set is the diff between PointsIn(n) and PointsOut(n). After *new* dest reg may point to something alloc'd.

$$pointsIn(n) = \bigcup_{p \in pred(n)} pointsOut(p)$$

$$pointsOut(n) = effect(pointsIn(n))(instruction_n)$$

```
effect :: PointsToSet → CFGNode → PointsToSet
effect pts (Node id (Cmp r1 r2)) = pts
effect pts (Node id (Bgt label)) = pts
effect pts (Node id (New n r)) = pts ∪ {(r, id)}
effect pts (Node id (Mov r1 r2)) =
  (removeTargets r2 pts) `union` [(r2, id) | (r1,id) <- pts]
    removeTargets r2 pts = [(r, id) | (r, id) <- pts, r != r2]
```

```
1:     New 8 D0
2:     New 8 D3
3:     Cmp D1 D2
4:     Bgt L
5:     Mov D0 D3
6:  L:
```

PointsIn(6) = {(D0,1),(D3,2)}
PointsOut(6) = {(D0,1),(D3,1),(D3,2)}