Compilers –Lexical & Syntax Analysis (Producing an AST) -> Semantic Analysis (Producing a Symbol Table / Typed AST) -> Runtime Memory organization -> IR Generation (Making an IR of assembly code. Optimizations) -> Assembly Generation (writing the assembly string based on our IR).

1) Compilers – Lexical Analysis

Transforming a stream of characters into tokens, based on the **formal description of tokens**.
Lexical Analyser made of **Regular Expressions.**

1.1) Identifier Tokens (Part of Lexical Analysis):

LA uses fast string lookup with hash function.
1) Keyword Identifiers: Special reserved words in language e.g 'class' → CLASS, 'while' → WHILE, etc. 2)
2) Non-Keyword Identifiers Programmer defined identifiers, such as varnames. A generic token used that uses the provided string name. 'var1' → IDENT("var1")
Literal Tokens are constant values embedded in the program (INTEGER(13), FLOAT(17.03), STRING("hi"))
Other tokens: Operators, whitespace, comments, pre-processing directives and Macros (think back to WACC).

1.2) Regular Expressions Rules

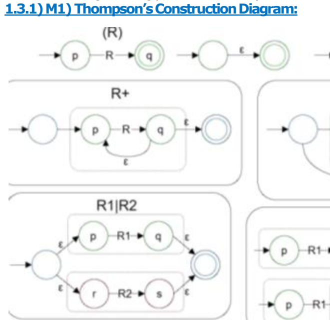
Can't be recursive. Regexes match strings.
Rule – Description – Example:
1) `a` - matches a symbol - `x` matches 'x' only.
2) `\symbol` - Escapes a regex character. `\(` matches '(' only
3) ε - matches the empty string. ε - matches "" only
4) R_1R_2 – matches adjacent as a combined rule - `ab89` matches 'ab89' only
5) $R_1|R_2$ – alternation, match one regex or the other. abc|1 matches "abc" or "1"
6) (R) – group regexes together, (a|b)c matches 'ac' and 'bc'
7) R+ - matches one or more repetitions of R, (a|b|j)+ matches all non-empty strings of a,b & j (eg: 'abbjab')
8) R* - Zero or more repetitions. R* equivalent to (R+)|ε
Precedence from highest to lowest: grouping, repetition, concatenation, alternation. We can derive these compound rules which are much more useful:
Rule – Description – Example:
1) R? – Zero or One occurrence – a? matches " and "a".
2) `.` - wildcard, matches any possible string.
3) `[abcd]` – Character set, match any chars in set.
4) `[0-9]` or `[a-zA-z]`– match a single number from 0 to 9, or any alphabet character.
5) `[^abc]` – match any character *except* those in the set

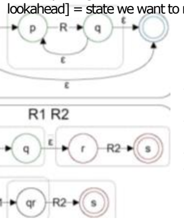
We can use these in **production rules:**
`SignedInt` → (+|−)? Int
`Keyword` → ' if' | 'while' | 'do'
When one or more expression matches, we choose the **longest matching character sequence.** Else, regex rules are ordered, with earlier rules taking precedence.
Lexical Analysers using Regex like this are quite easy to write, but it becomes hard to change token rules. In practice we use **Lexical Analyser Generators**, which take programmer defined token structures and functions based on the formal language definition to produce a **tokenizer** (program input → AST).
To generate a Lexical Analyser, we first write our *Regular Expressions* and then use **Thompson's Construction** to convert to a *Non-deterministic Finite Automata.* Then use **Subset Construction** to convert to a *Deterministic Finite Automata*. We can do **further optimizations** to get a *Minimum State DFA* and from here we can get a *Transition Table + GetToken function* - by **mapping our graph (our FSM) into a table,** which is our *Lexical Analyser.*

1.3) Finite Automata (Finite State Machines)

Arrows denote transitions between states.
Start state has an unlabelled transition to it.
Accepting states are double circles.
Will stop when no transition can be made (as a result matches the longest string possible to a state).

1.3.1) M1) Thompson's Construction Diagram:





NFAs require backtracking when using since multiple eq- transitions from states. So, we convert NFAs to DFAs.
1) **NFA**: **Space**: O(len R), **Time:** O(len R × len X)*
2) **DFA**: **Space**: $O(2^{len R})$, **Time:** O(len X)

1.4) M2) NFAs to DFA (subset construction)

1) Start at the NFA start state.
2) Compute the ε-Closure for each state. These are the set of all states that can be reached from the current state by any number of ε transitions. Set the new name of each node to be it's ε-Closure.
3) Now, from our grouped node, consider all A transitions. Take the epsilon closure of all nodes we can go to with A, as a node. Do this for B and so on.
4) For all the new nodes we constructed, carry out step 2 and 3. **Note: Be very careful when adding transitions – remember we can loop to ourselves if we had a connection from one node in our epsilon closure to another!! Also, any node that contains an accepting state in its epsilon closure is now an accepting state in our DFA. This accepting state can still have transitions.**
Context Free Grammars are transformed to NFAs. CFGs take the form rule -> (rule|token)*

2) Parsing - LR Parsing

2.1) Chomsky Hierarchy Given:

R = non-terminal   *t* = sequence of tokens
α, β, φ sequences of terminals and non-terminals
regular ⊂ con-free ⊂ con-sensitive ⊂ rec. enumerable

2.2) Parser for Context Free Grammars

Context-free grammars parsed with complexity $O(n^3)$.
LL and LR grammars subsets of CFG, parsed in O(n), which is why we use them. LL(n) is a subset of LR(n). Must be generated. **LR Parsers** are bottom up, building the AST from the leaves to the root. **LL Parser** are top down, building the AST from root to leaves. LL/LR(k) denotes an LL/LR parser with k token lookahead.

2.3) LR(0) Parsers. 2.3.1) LR(0) Items

LR(0) parsers don't use the current token to perform a reduction. • represents the current position of the parser.
Initial Item - we haven't parsed any part of a rule yet,
Complete (Reduce) Item – we've fully matched a rule and can be reduced.
The rule X → AB has 3 LR(0) items, the initial item •AB, A•B, and the reduce item AB•. We add a **start rule:** E' → E $ - where $ is end of input. If omitted, its implied.

2.3.2) NFA from LR(0)Items

– we can create transitions between LR(0) Items 1) Given an item X -> A•BC $\to_B$ X → AB•C if we have a rule for B.
2) If B is non terminal (no rule matches), for each rule B → •D, we add a new ε transition (we can't reduce yet).
(X →A •BC) $\to_\epsilon$ (B → •D) **This means if we're about to process a token that's a rule in a DFA state, we have to add the productions of this rule to our DFA state.** To convert to the **DFA** from here, we do subset construction, but its simple enough right away.

2.3.3) DFA to LR(0) Parsing Table

The parsing table describes the rules to apply, and states to move to when a given item is encountered. It is generated from a DFA using the rules:
1) For each Terminal translation X $\to_T$ Y
Add P[X, T] = sY (shift Y, we cannot reduce yet)
2) For each Non-terminal translation
X $\to_N$ Y Add P[X, N] = gY (Goto Y)
3) **For each State X containing item R' → · · · •**
Add P[X, $] = a (accept) (end of input)
4)For each State X containing item R → · · · •
Add P[X, T] = rN (reduce) (use rule rN to reduce)
An empty cell indicates an error.
We have a table of our states as rows, and each possible input as our columns (including $, and Goto for ele-ments of S / whatever grammar rule we're parsing).

2.3.4) LR Parser Operation

For action, we write T[state, lookahead] = what we do. We start with 0 (state 0) on the stack. $ ends our tokens
1) shift S_n **2)** accept a **3)** error – report error.
4) goto G_n **5)** reduce r_n. 1. Use rule n to pop m states from stack; m is length RHS of rule n
2. Push the state we have at the top now (after the pops), with the LHS of the rule we had matched, and then compute where that takes us as normal (e.g push[2,7,S] = g8)
4. Generate AST node for rule
We used parsing table to do work. We write T[state, lookahead] = state we want to move to.
We should end with an accepting state. Remember after a rule is resolved, we jump to another state and pop n states, then add that state onto the stack!! Also we don't touch remaining tokens.

$\to_\epsilon$ Zero or more derivations.
$\to_+$ One or more derivations.
A → ε Non-terminal A is nullable.
A → AB|BC AB and BC are alternatives of A.

M3) Computing the First Set:

We can construct first set from the rule by going through each alternative. In each alternative we iterate through each terminal/non-terminal B, if ε ∈ first(B) then we include first(B) \ {ε} in the first set and move on to the next in the sequence.

$$\text{first}(\alpha) = \{t : \alpha \to^* t\beta\} \cup \begin{cases} \{\epsilon\} & \text{if } \alpha \to *\epsilon \\ \{\} & otherwise \end{cases}$$

Example: 1) Factor → '(' Expr ')' | id
2)Term2 → '*' Factor Term2 | ε
3) Term → Factor Term2
4) Expr2 → '+' Term Expr2 | ε
5) Expr → Term Expr2
6) Expr' → Expr $
FIRST(Factor) = {'(', id}
FIRST(Term2) = {"*", ε}
FIRST(Term) = FIRST(Factor) = {'(', id}

M4) Computing the Follow Set:

The follow set for a non-terminal rule is the **set of all tokens (terminals)** that could immediately follow, **and $ if the end of the input could follow.** Hence we must check each rule which contains the non-terminal. Given rule B → C A D we have follow(A) =

$$\text{first}(D) \setminus \{\epsilon\} \cup \begin{cases} \text{follow}(B) & \text{if } D \to^* \epsilon \\ \{\} & otherwise \end{cases}$$

C can be empty here, since it is at the start of the rule, it has no bearing on the follow set. If A can end the input, we include $ in follow set.
Example for the rules above:
Follow(Expr) = {')', $}
Follow(Term) = FIRST(Expr2) + Follow(Expr) + Follow(Expr2) (because Expr2 can be a null string and thus we need the follow of Expr2) = {+,), $}
Follow(Term2) = Follow(Term) + Follow(Term2), but that has the same items so it's just Follow(Term) = {+,), $}.

2.4.1) LR(1) Items

A pair [LR(0) item, lookahead token t], hence if we have the LR Item [X → A • B C, t]:
1) A is on top of the stack
2) **We only want to recognise B** if it is followed by a string derivable from Ct (aka: the current token is in the first set of Ct)

2.4.2) NFA from LR(1) Items

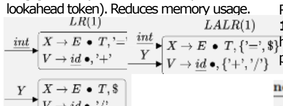
Given a state [X → A • B C, t] we add the transition: [X → A • B C, t]$\to_B$[X → A B • C, t].
When we do this, **If B is a rule/non-terminal** then for every rule of form B → D we add an ε transition and a new state for every token u in first(Ct)
[X → A • B C, t] $\to_\epsilon$ [B → • D, u]
We also need an initial item for the rule [X' → • X, $] (start of text, has end of input token $).

2.43) LR(1) Table

Our parsing table can now contain several different rules per row (same state, different current token). **We only perform reduction of a rule [A → A •, t] when the current token is t (equal to lookahead tok)** So in practice our table doesn't always r_n splayed out across a row, usually just 1 column in the row.

2.5) LALR(1) Parsers

LALR(1) Parsers are similar to LR(1) but we **merge LR(1) states that have the same LR(0) items** (and thus differ only in their lookahead token). Reduces memory usage.



Some reductions can occur before an error is detected as a result, which an LR(1) would have immediately detected but they will later be detected by LALR(1).
Ambiguous Grammars produce more than one parse tree for some inputs

2.5) M6) Fixing Shift Reduce Conflicts

Caused by ambiguity where parser can't decide to reduce tokens to the LHS of a rule, or shift another token. For example this grammar Expr → Num and Expr → Num + Num
We can fix by rewriting the grammar, encoding precedence, introducing some other rule:
E → - E | E – Int | Int becomes (separate out)
E → E – Int | T   T → - T | Int.

For **Reduce-Reduce** conflicts, caused by two rules having the same RHS, we add precedence to some rules (e.g earliest).

2.6) Parse Trees

Leaf nodes built on a shift operations
Non-Leaf nodes created on a reduce.
To make AST we do a pass over Parse Tree, or have AST construction rules to build AST while parsing. To make these properly, just look at our input and figure out how we could construct it (what rules were taken and uses).

2.7) LL Parsing

Top-down – recursive descent or DFA. Handcoded or generated.
LL(k) Grammar – a grammar is LL(k) if a k token lookahead is sufficient to determine which alternative of a rule to use when parsing.
LL(1) uses the current token **ONLY**. LL(0) could not exist as it doesn't use ANY TOKENS. **Leaf Nodes are constructed from the root.**

2.7.1) LL(1) Grammar

A grammar is LL(1) if for a rule
A → α | β (A is non-terminal):
first(α) ∩ first(β) = ∅ ∧
ε ∈ first(α) ⇒ (first(β) ∩ follow(A) = ∅) ∧
ε ∈ first(β) ⇒ (first(α) ∩ follow(A) = ∅)
AKA – if a and B don't share first tokens, or if epsilon is in the first of A or B, then the first of B/a and follow A don't share anything, makes sense!

2.7.2) Extended Backus Naur Form (EBNF)

Used to improve CFGs. **{α} = 0 or more occs of α**
[α] = 0 or 1. (...) **groups elems together.**
We left factor with alternatives with intersecting first sets: Expr → Term '+' Expr | Term becomes Expr → Term ['+' Expr]
We can remove left recursion:
Sequence → Sequence ';' Statement | Statement
→ Sequence → Statement {';' Statement}
Encoding Precedence:
1. Define language literals.
2. Start by using top precedence operators first.
3. Use the next higher operators, lowest last.
4. Ensure our grammar is LL(1) – using the LL(1) definition above in the LL(1) section.
Defining LL(1) bool grammar, or < and < not prec.
boolexpr → andexpr { or andexpr }
andexpr → notexpr { and notexpr }
notexpr → {not} operand
operand → true | false | '(' boolexpr ')'

2.7.3) Writing a Recursive Descent Parser

Recursive Descent Parsers for LL(1) consist of:
1)A **parse function for each rule**, e.g: A()
2) The current input token, a global variable token
3) A token match and advance function:

```
def match(expected):
  if token == expected:
    token = lexical_analyser.get_token()
  else error("unexpected token found.")
```

We can easily better error messages/error recovery by saying what we encountered, the position, what we expected and then returning an encompassing ERROR_TOKEN.
Example Parsing Patterns:

```
1) A B   A() B()
2) A | B if next_token() in first(A): A()
         elif next_token() in first(B): B()
3) {A} while next_token() in first(A): A()
4) [A] if next_token() in first(A): A()
```

2.7.4) AST Construction

Class hierarchies organize nodes into variants of a given type (e.g if statements, print statements and assignments are all statements, so they inherit from some Statement class).

2.7.5) CFG to LL

Left Factorization, Substitution and Left Recursion Removal are typically used transformations.
1) Left Factorization: Two or more alternatives have a common prefix. We factor this to be parsed before deciding which alternative to parse:

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

with its alternatives, can make left factors obvious:
A → B | C   B → 'hello'
C → 'hello' 'there'
We have an indirect conflict as both alternatives for A start with 'hello'. We can directly substitute instead: A → 'hello' | 'hello' 'there'
Now we can left factor: A → 'hello' ['there']
3) Left Recursion Removal: Must be done to make grammars LL(1):
A → X | A Y ⇒ A → X{Y}

As an example of Left Recursion Removal:
Expr → Expr ('+' | '-') Term | Term
Term → Term ('*' | '/') | F actor
Factor → '(' Expr ')' | int
Parse tree may no longer represent the associativity, hence we will need to ensure the arithmetic is still associative when we construct the AST:
Expr → Term {('+' | '-') Term}
Term → Factor {('*' | '/') Factor}
Factor → '(' Expr ')' | int

2.7.6) Error Recovery

We need to emit good error messages, and skip as little code as possible to recover. **Panic Mode Recovery:** Each parse function has a syncset of tokens. When an error occurs the parser skips forward until it encounters one of these tokens.

2.7.6.1) Panic Mode Error Recovery

Panic Mode Recovery: Each parse function has a syncset of tokens **(usually the follow set – provided as extra args to the function).** When an error occurs the parser skips forward until it encounters one of these tokens (thus repairing the state).
We improve our check function:

```
def check (expectset, syncset, error):
    if next_token() not in expectset:
      add_error(next_token(),
parser_pos(),
                    expectset, error)
      skipto(expectset + syncset)
```

Skipto simply pop_token()s while our next token isn't in syncset or EOF.
We use check at the top of our higher parsers to make sure things are ok. We use match in our rules parsers to match:
def Statement(syncset):
check ({IF, PRINT, BEGIN}, syncset, "Error ...")
    if token == IF: IfStatement(syncset)
    ....
def Case():
  match(CASE)
  Token[] labels = []
  labels.append(match(INT)) while match(COMMA)
  match(COLON)
  StatementAST s = Statement()
  return CaseAST(labels, s)   ...

3) Semantic Analysis

Checks if parsed prog semantically valid according to language rules. Produces a symbol table of identifiers and their types. Hand written usually.

3.1) Typical Semantic Checks

1) **Variable Declaration:** <Type> = <Id>
We check if type is in scope (declared).
We check if type declaration is valid (void a isn't)
Check if identifier already in scope.
2) **Assignment:** <id>=<Expr>
Check if the identifier is valid, check the expr, check the types of the left and right lineup.
3) **Array Declaration:** <Type> <Id>
[<Size>]...
Check type, identifier in scope, size, some arrays are too large so raise warning if num too big.
4) **ArrElem Assign:**<Id>[<Index>]
...=<Expr>
Check if the var id is in scope, check id's type is indexable, and index is int. Check expr match arr type.
5) **Function Declaration**
<RetType><Name>(<ParmType><ParamId>,...){...}
Check type of ret and params, check funcname declared, check scoping (argScopeCanBeShadowed), has returns?
6) **Function Call** <Fun Name>(<Expr>, ...)
Check if args are valid exprs, and check their types. Check return type = LHS type.

3.2) Symbol Tables

Rather than store Semantic info in AST, we do it in a **Symbol Table** instead. We use a tree of symbol tables for scoping, or use renaming for one symbol table. In our symbol tables, we map our identifiers (names) to EntryTypes: which could be any Type, or things outside of types: Packages, Functions, NewScopes (these have their own sym tables as children). What we add to the table can either be all the information from the AST node or a pointer to it.

4) Runtime Memory Organization.

4.1) Alignment

Char = 1,2 Bytes (Unicode)
Pointer Types in some languages. Usually 4 Bytes. When aligned to a multiple of two 2^k, the first k bits of the address will be zero. For example alignment of pages in memory allow for page table entries to use the bits that will always be zero (due to alignment) to be used for other information (e.g reference, r, read/w)

4.1) Structs and Records

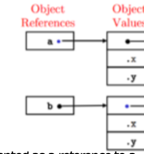
1) Allocated in contiguous block of memory.
2) Alignment is used to space fields out correctly.
3) C orders fields by their code position, others optimize (its important for us in C to optimize this)
4) Tuples are anonymous structs.
If no padding/alignment/reordering is done as in C then:
1) Record = ($Field_1$, $Field_2$, ... , $Field_n$)
2) Size(Record) = Size($Field_1$) + Size($Field_2$) + ··· + Size($Field_n$)
Address($Field_k$) = StartAddress(Record) + Size($Field_1$) + ··· + Size($Field_{k-1}$)

4.2) Arrays

A contiguous section of memory populated by n variables (elements) of the same type.
1)Can have elements aligned
2) Some languages associated arrays with auxiliary data (e.g length for bounds checking).
Without any alignment or padding:
Array[Type] = $Element[Type]_1$, ... , $Element[Type]_n$
Size(Array) = Size(Type) × n
Address($Element_k$) = StartAddress(Array) + k × Size(Type)

4.3) Objects

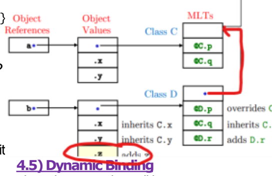
```
class C {
  int x
  int y
  method p ...
  method q ...
}
C a = new C()
C b = new C()
```



Objects are implemented as a reference to a record with an additional pointer to an MLT for the class (**inheritance**)
When we call an object method, traverse to the object's MLT. We jump to the method, having placed the first argument as a pointer to the object.
We access fields just like a struct:
Object.field = Mem[Mem[ObjectRef] + OffsetToField]. To call methods: CALL Mem[Mem[Mem[@ObjRef] + OffsetToMLEntry]]]
e.g: b.q() is: mov eax, <b>
push eax
mov eax, [eax]
call [eax+4] #q is at offset 4.
The indirection required for accessing values, calling methods adds overhead

4.4) Inheritance and Overriding

For single inheritance we include the parents fields in the struct and methods in MLT.
Then, we just append the extra fields normally at the bottom of the struct variables. The methods of C are overridden in D or inherited – so we still access Class C's methods in the exact same way in D (same offsets). To add methods, we just append them at the bottom of our subclass MLT. We have a pointer pointing to the **super class MLT** for type testing.



4.5) Dynamic Binding

This scheme works well because we can set a = b, where b is a subclass of a. We only have the fields which a had visible – so we lose access to the extra fields b appended but this is okay – our b can act as an a. The key is what happens in the **MLT** - since all of C's methods exist in D we can call them as if we were C and still work (though they do different things).

4.6) Program Address Space

1) Code Segment, (read-execute)
2) Stack Segment, has local variables, func args, data. Stack Frame = FP or EBP in IA32.3) Data Segment which contains
1) Static Area for Global Variables, pointer to MLT (in Heap Memory). Global Variables are placed in order of the allocation, as with MLT. 2) Heap Memory.

M7) Modelling Runtime Memory

```
class Triple {int p, q, r}
class Alpha {
  int ca
  int cb[10]
  method M(int u, v) {
    int la
    int lb[10]
    struct {int x, y, z} lc
    Triple id = new Triple()
  }
}
```

We want to model runtime memory of: Alpha object = new Alpha() object.M(2, 3).
I have the stack growing **downwards.**

```
Stack: 0 object addr → object Ref
       4 return addr
       8 caller FP
       12 la
       16 lb[0]
       ...
       52 lb[9]
       56 lc.x
       60 lc.y
       64 lc.z
       object ld → Id Ref
```

Id Ref = [0: Triple @MLT, 4: .p, 8: .q, 12: .r]
Object Ref = [0: Alpha @MLT, 4: ca, 8: cb[0]..., 44: cb[9]]

4.7) Dynamic/Heap Variables

4.7.1) Explicit Heap Allocation

1) We maintain a structure containing free blocks and info of each block – e.g a linkedlist (think to PintOS).
2) Should allocate blocks just big enough.
3) For extensible arrays (vectors, mutable strings) we should alloc more mem than curr needed, as we should expect growth. (performance, otherwise need to delete and realloc bigger space).
4) Consider alignment of values.
5) Heap allocation is worse for locality (e.g than the stack) and hence can potentially have worse cache locality, affecting performance.
Malloc(int N(size)) Pseudocode:
SEARCH a List of "Free Memory Blocks" for a block of size N or a bigger block
IF a free block of exactly size N is found:
Remove block from FreeList
Return start address of block
ELSE IF a free block bigger than N is found:
Split block into an N-sized block and a residual block
Leave residual block in Free List
Return start address of N-sized block
OTHERWISE if the Search did not return a suitable free block:
Request more heap memory from the Operating System & allocate
IF the OS does not comply:
Report failure e.g. return Null or raise an exception
Free(address) Pseudocode:
RETURN block at given address to list of "Free Memory Blocks"
COALESCE returned block with adjacent free block(s) if possible => reduces fragmentation
Housekeeping: When allocating heap objects we can have some extra information before the struct fields (e.g array length).
Optimizations: Have multiple free lists for diff allocations. Faster searches. Allocate blocks just big enough.

4.8) Garbage Collection (Free Redundant Heap Blocks!)

Correctness, Performance. The GC must be compiler supported.
After lots of blocks are freed, we should compact the heap.

4.8.1) Reference Counting

Block management/housekeeping info keeps a reference count. When a reference is made to the object, the refcount is increased, when it is removed it is decreased.
When the refcount is zero, free. Simple, efficient, but requires compiler to track refs, and we must resolve reference cycles (A references B, B references A, nobody references them. We must collect them still). Refs increase during assignment or allocation (if our ref object is on the RHS). Refs decrease if they're on LHS of an assign.

4.8.2) Mark Sweep Garbage Collection

Pause the program and collect all blocks not pointed to. Finds **ALL** dead blocks unlike refcounting. Significant Overhead.
1) Mark: For all stack vars, if they point to the heap, then recursively traverse and mark each block they point to as live.
2) Sweep: Go through each block. If not live, then collect and deallocate it.

4.8.3) Pointer Reversal Marking

Cheap way of traversing nodes.
We have two pointers, P and C.
When traversing blocks, C is at the block, the block has pointer Cp1 pointing to Child1.
To check child 1, we set the block at pointer Cp1 to P, and put P as C, then put C as Child1. We do our work in child1 (find children), and then we set C = P, restoring C, set P to Cp1 restoring P. We then restore the pointer Cp1 to point to child1.

4.8.4) Two Space Garbage Collector

Heap is split into two, **from-space** and a **two-space**.
1) Blocks are allocated from the from-space.
2) When there are no more free blocks, all live (reachable from a non-heap pointer) blocks are copied to the to-space and then the to-space becomes the from-space and vice-versa.
3) Fast, there are few complex pointer manipulations.
4) Automatically compacts memory when copying.
5) Can place linked blocks close together in memory when copying to allow for better cache locality.

4.8.5) Generational Garbage Collector

The heap is split into several areas based on the age of blocks.
1) Allocate new blocks from the youngest, if full, move the oldest young blocks to an older area.
2) GC is used more frequently on the younger areas. Can apply different GC techniques to the different areas

4.9) Debuggers

Inspect program behaviour.
1) We must maintain program info when its being debugged.
2) Should add useful info (func/var names, source line mappings) in our program.
3) Usually compile with optimizations off so its mirroring the source code.
Interactive Debuggers: GDB – inspect state, variables, funcs, mem, pause execution. For natively compiled (Rust, C, Go) OS must support **breakpoints.** For bytecode compiled (Java), the interpreter supports this.
Post-Mortem / Core Dump e.g backtrace Provides debugging information upon failure by doing a **reverse lookup from the program counter.**
1) Stack Backtrace shows the methods called, their local variables, arguments etc.
2) Contents of global and dynamic/heap variables.
Requires the debugger to work out where variables are stored (register, stack, heap) and to map them back to names from the source using debugging information embedded at compile time.

4.10) Profilers

Provide performance information on a program.
1)Typically used once code is correct, algorithmically optimal and optimised by the compiler.
2) Can determine the time spent in functions, or how much of a program is spent is certain sections of code.
For example a profiler could interrupt execution periodically (some number of ms), identify the method being used, and increment a counter for it. Hence after profiling we can get a breakdown of the proportion of program time spent in each function.

5) Compilers – Backend

5.1) Backus Naur Form

A CFG specifying the syntactic structure of a language
A CFG is a set of **Productions**, associated with a set of **tokens** (terminals), non-terminals (**rule**) & start symbol
Each production is of the form:
non-terminal → String of terminals & non terminals

1) Productions: A way to expand a non-terminal symbol into a string of terminals & non terminals
2) Terminals: Symbols that can't be further expanded (tokens genned from Lex. Analysis)
3) Non-Terminal: Symbols that can be expanded further – outlined in a Production.
3) Parse Trees show how a string is derived from the start symbol.

5.1.1) Associativity Associativity can be enforced by using left or right recursive productions:

term → const | ident   Define a base term.
expr → expr - term   Left associative
expr → term - expr   Right associative
expr → term

5.1.2) Precedence

To enforce precedence, we can consider **levels.** We factor those of highest precedence to the lowest level.
exp → exp + term | exp – term | term
term → term * factor | term / factor | factor
factor → const | ident
To prove something is a member of our grammar we can construct the derivation or a Parse Tree.

5.1.3) Production Choice

We may have a grammar where we cannot determine which production for a non-terminal token to use based on the first symbol.

```
stat → 'loop' statlist 'until' expr
stat → 'loop' statlist 'while' expr
stat → 'loop' statlist 'forever'
```

When we have token 'loop' we cannot determine which production to use. Methods to deal with this:
1) Delay the choice
Delay creating this tree (from stat) until it is known which production matches. It is still possible to create the statlist inside while doing so.
2) Modify the grammar
Change the grammar to factor out the difference.
stat → 'loop' statlist loopstat
loopstat → 'until' expr
loopstat → 'while' expr
loopstat → 'forever'
However there are more difficult problems, which can be more easily fixed with bottom-up parsing.
Top down parsing is done by Rec. Desc. Parsers, which can't deal with left recursion.
Bottom-up Parsing
The grammar's productions are used right → left.
Input is compared against the right hand side to produce a non-terminal on the left.
Parsing is complete when the whole input is replaced by the start symbol. Bottom up parsers are difficult to implement, so parser generators are recommended.

5.2) Visitor Pattern

The visitor pattern is a design pattern that is commonly used in object-oriented programming to separate algorithms from the objects they operate on. The idea behind the pattern is to create a separate class (the visitor) that can traverse a complex object structure and perform operations on its elements.
In the context of compilers, the visitor pattern can be used to implement the different phases of a compiler as separate visitor classes. For example, a lexer could be implemented as a visitor that traverses the source code and generates a stream of tokens, while a parser could be implemented as another visitor that takes this stream of tokens and generates an abstract syntax tree (AST).

6) Code Generation

1) The language:
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
type Name = [Char]
2) Assembly Instructions:
data Instruction = Add | Sub | Mul | Div |
PushImm Int |
PushAbs Name |
Pop Name | CompEq |
Jump Label | JTrue Label |
JFalse Label | Define Label

3) Assembly Pseudocode

```
DD / MINUS / MUL / DIV:
    T := store[SP]
    SP := SP + 4
    T := store[SP] [+=*/] T
    store[SP] := T
PUSHIMM:
    SP := SP - 4
    store[SP] := operand(IR)
PUSHABS:
    T := store[operand(IR)]
    SP := SP - 4
    store[SP] := T
POP:
    T := store[SP]
    SP := SP + 4
    store[operand(IR)] := T
COMPEQ:
    T := store[SP]
    SP := SP + 4
    T := store[SP] - T
    store[SP] = T=0 ? 1 : 0

JTRUE / JFALSE:
    T := store[SP]
    SP := SP + 4
    PC := T=1 / 0 ? operand(IR) : PC
```

4) Translate Functions for Exp and Stat

```
transExp::Exp →[Instruction]
transExp(BinOp op e1 e2) =
  transExp e1 ++ transExp e2 ++ [case op of
                    Plus → Add
                    Minus → Sub
                    Times → Mul
                    Divide → Div]
transExp(Unop Minus e)
  = transExp e ++ [PushImm (-1), Mul]
transExp(Unop _) = error "(transExp)
  Only '-' unary operator supported"
transExp(Ident id) = [PushAbs id]
transExp(Const n) = [PushImm n]

transStat::Stat →[Instruction]
transStat(Assign id exp) = transExp exp ++
                           [Pop id]
transStat(Seq s1 s2) =
              transStat s1 ++ transStat s2
transStat(ForLoop x e1 e2 body)
 = transExp e1 ++ [Pop x] ++ [Define "loop"]
 ++ transExp e2 ++ [CompEq:JTrue "break"]
 ++ transStat body ++ [PushImm 1, Add, Pop
x, Jump "loop", Define "break"]
```

7) Improving our Assembly

1) Using Immediate Instructions
movl $3, %eax
imull $3, %eax
addl $3, eax
Rather than moving into registers first and then doing work.
We simply add new AddImm, SubImm, etc data types.
The translateFunctions for these are very simple.

2) Dealing with Bounded Numbers of Registers

1) Accumulator machines have 1 register. We store the accumulated value in there.

```
data Instruction = Add | Sub | Mul | Div —
  | AddImm Int | SubImm Int | MulImm Int | DivImm Int —
  | CompEq       — CompEq   -> Acc :=
  | Push         — Push     -> SP−−; mem[SP] := Acc
  | Pop          — Pop      -> Acc := mem[SP]; SP++
  | Load Name    — Load n   -> Acc := mem[n]
  | LoadImm Int  — Load i   -> Acc := i
  | Store Name   — Store n  -> mem[n] := Acc
  | Jump Label   — Jump l   -> PC := l
  | JTrue Label  — JTrue l  -> IF Acc = 1 THEN JUMP TO l
  | JFalse Label — JFalse l -> IF Acc = 0 THEN JUMP TO l
  | Define Label — Assembler directive to set up label
```

```
transOpImm :: Op -> (Int -> Instruction)
transOpImm Plus = AddImm
transOpImm Minus = SubImm
transOpImm Times = MulImm
transOpImm Divide = DivImm
```

transOp :: Op → Instruction is obvious

```
transExp :: Exp -> [Instruction]
transExp (Const n) = [LoadImm n]
transExp (Ident x) = [Load x]
transExp (Unop Minus e) = transExp e ++ [MulImm (−1)]
```

```
— Can only use Minus unary operator (e.g −3)
transExp (Unop _ _)
  = error "(transExp) Only '−' unary operator supported"

— If constant on the left, can use immediate operand
transExp (BinOp op e (Const n)) = transExp e ++ [transOpImm op n]

— With commutative operator, can switch order to use immediate operand
transExp (BinOp Times (Const n) e) = transExp e ++ [MulImm n]
transExp (BinOp Plus (Const n) e) = transExp e ++ [AddImm n]

— General case for two expressions
transExp (BinOp op e1 e2) = transExp e2 ++ Push : transExp e1 ++ [transOp op]
```

2) Combine the register and accumulator approach: Use registers as normal, and when we run out of registers take the Accumulator machine approach.
Our only code change from the before is the BinOp case with two exprs in transExp (and we pass in the register we want to store on in transExp):
(final case)

```
transExp (BinOp op e1 e2) r
 | r == maxReg = transExp e2 r ++
   [Push r] ++ transExp e1 r ++
   [translateOpStack op r]
 | otherwise = transExp e1 r ++
   transExp e2 (r + 1) ++
   [translateOp op r (r + 1)]
```

7.1) Register Usage

1) Sethi Ullman Weights

Given an expression E1 op E2 always evaluate the subexpression that uses most registers 1st
1) If E1 evaluated first, registers needed is max(E1, E2 + 1)
2) If E2 evaluated first, registers needed is max(E1 + 1, E2)
This is nicely encoded by the *weight* function

```
weight :: Exp -> Int
weight (Const) = 1
weight (Ident) = 1
weight (Unop Minus e) = weight e
weight (Unop e) = error "only takes unary"
weight (BinOp Plus (Const) e) = weight e
weight (BinOp Times (Const) e) = weight e
weight (BinOp e (Const)) = weight e
-- Use maximum of either
weight (BinOp e1 e2) = min e1_f e2_f
  where
    e1_f = max (weight e1) (weight e2 + 1)
    e2_f = max (weight e2) (weight e1 + 1)
```

This doesn't work for divide or subtraction – as they can't be reordered, instead we use register targeting.

2) Register Targetting

We give transExp a list of all the registers that are free and by convention we want the result of the operation to be stored into the first register in the list:

```
transExp :: Exp -> [Register]->[Instruction]
transExp (Const n) (dst:rst)=[LoadImm dst n]
transExp (Ident x) (dst:rst) = [Load dst x]
```

The other cases are standard. Binop case:

```
transExp :: Exp -> [Register]->[Instruction]
transExp (Const n) (dst:rst)=[LoadImm dst n]
transExp (Ident x) (dst:rst) = [Load dst x]
transExp (BinOp op e1 e2) [dst] =
  transExp e2 [dst] ++ [Push dst] ++
  transExp e1 [dst]++[transOpStack op dst]
transExp (BinOp op e1 e2) (dst:nxt:rst)
  | weight e1 > weight e2 =
    transExp e1 (dst:nxt:rst) ++
    transExp e2 (nxt:rst) ++
    [transOp op dst nxt]
  | otherwise =
    transExp e2 (next:dest:rest) ++
    transExp e1 (dest:rest) ++
    [transOp op dest next]
```

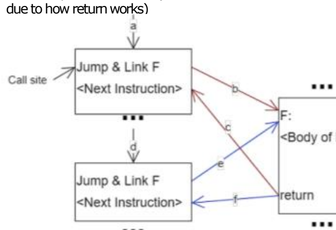
This works well, as the expression size accommodated by N registers, is 2^N.

7.2) Register Allocation for Function Calls

Need to know where parameters are when passed.
(f(x) + 1) + (1 * (a + j)) Which side of the + should be evaluated first depends on the context (e.g registers that need to be saved at the call site, and registers used by the callee, calling convention).

7.2.1) Infeasible Control Path Problem

Control Flow Graphs capture control flow inside functions and methods but not between them. We could have infeasible paths as follows: (a,b,f invalid, a,d,e,c invalid due to how return works)



saveRegs unusedRS = [Mov (Reg x) Push | x<- usedRs]
where usedRs = allRegs \ unusedRs

restoreRegs unusedRS=[Mov Pop (Reg x)|x<-revUsedRs]
where revUsedRs = reverse (allRegs \ unusedRs)

7.2.2) Caller and Callee Saving

We must enforce a **calling convention** to ensure registers are not **clobbered** (e.g non-argument or return registers are changed).
1) Caller Saved: save registers used by the caller in case the callee clobbers them.
(e.g: Save used registers by caller that callee also uses, call method, restore used registers by caller that callee also uses. **Problem:** We have to know what registers the callee uses.)
2) Callee Saved: Save registers that the callee uses only.
(e.g: In the called method, save registers that the callee uses if they are also used by the caller – at the end of our method restore them. **Problem:** We have to know what registers the caller uses.)

7.2.3) IA 32 Calling Convention solves this

Caller-Saved	Callee-Saved	Stack Pointer	Frame Pointer
%eax %edx %ecx	%ebx %esi %edi	%esp	%ebp

7.3) Register Allocation by Graph Colouring

1) **Use simple traversal to generate intermediate code** Temporary values are always saved in a named location. (e.g t0…). This way we can consider all values including intermediate ones.
2) **Construct an Inference Graph** each node is a temporary location, each edge connects simultaneously live locations. Registers that need to simultaneously store values must be different colours (different registers).
3) **Attempt To Colour Nodes:** If colouring is not possible spilling occurs.
(a) Find an edge, to remove it either split the live range (e.g temporarily put to memory).
(b) Redo the analysis to determine if the graph can now be coloured.
When choosing which values to spill it is important to consider how often a variable is used. e.g avoid spilling from innermost loop. While NP hard, heuristics exist.

8) Optimization

High level optimizations use **high-level info** encoded in the program: (types, func analysis), e.g: Function Inlining.
Low level optimizations use **low-level info** (instruction types, the ISA, the order of instructions in the IR, etc) to optimise the output. e.g: Instruction Scheduling.

8.1) Peephole Optimization

Scan through the assembly in order, *looking for obvious cases to optimise.*
1) Can catch some of the worst cases (e.g store followed by load of the same location).
2) Very easy to implement (at smallest just consider two adjacent instructions).
3) Phase ordering problem in what order should the optimisations be applied to get the best result?

8.2) Lowering Representation

Taking high-level features and converting them into lower-level representations. For example taking arrays and converting them into pointer arithmetic/address calculation. When lowering you loose high-level information (e.g that values are part of an array), but can optimise the lower level representation (optimise address calculations). We usually start with high level IRs, analyse, optimize, then move to lower IRs, optimizing based on the info we have on each level.

8.3) Other Optimizations

1) **Induction Variable** – a variable which increases / decreases by a (loop invariant) constant on each iteration.
2) **Strength Reduction** – an optimization where we calculate induction variables by breaking it down into a single addition rather than a compound expr which might have multiplication – which is expensive. (in general, replace a complex operation with a simpler one).
3) **Control variable selection** – replace loop control variable (as in the i in "for i in range") with an induction variable in our loops instead, and then rework the bounds check to work with the values of this induction variable (so we have less increments / variables).
4) **Dead Code Elimination** Code that does not produce a used result can be eliminated. many other optimisations result in dead code (e.g inlining a function where not all the function's returned values or optional arguments are used.)

8.4) Data Flow Analysis for Live Ranges

Live Range - the range of instructions for which a temporary value must be maintained. A live range starts at a definition, and ends when either the variable is used, or immediately if the value is never used. Like with Graph Colouring we have a similar process:
1. Generate code using temporaries T0… instead of regs.
2. For each temporary T_i, find T_i's live range – set of instructions for which T_i must reside in a register.
3. If liveRange(T_i) intersects liveRange(T_j) then they must be allocated to different registers – **they interfere.**
4. Assemble the Register Inference Graph (RIG).
5. Colour the RIG. If successful replace temporaries with register and generate code. If Graph can't be recoloured, then find a temporary to spill, retry. Num colours = regs.

```
data CFG = ControlFlowGraph [CFGNode]
data CFGNode = Node Id Instruction [Register] [Register] [Id]
type Id = Int                          ^uses ^defs ^succs
data Register = D Int | T Int
buildCFG :: [Instruction] -> CFG
```

List of Nodes in our graph. The nodes contain an ID, an instruction, the temporaries used by the instruction, and the ones defined by it, and the successors (the ids after the node – edges).
First we build the CFG. **Things to note**
1) Succs refers to all the possible paths that can be taken from the current node – e.g
IR Code, **CFG (line, instruction, uses, defs, succs):**

IR Code	line	instruction	uses	defs	succs
Bra L2	1	Bra L2	[]	[]	[10]
L1:					
cmp b a	2	cmp b a	[b, a]	[]	[3]
bge L3	3	bge L3	[]	[]	[4, 8]
mul #7 a	4	mul #7 a	[a]	[a]	[5]
mov a b	5	mov a b	[a]	[b]	[6]
add #1 b	6	add #1 b	[b]	[b]	[7]
bra L4	7	bra L4	[]	[]	[10]
L3:					
mov b a	8	mov b a	[b]	[a]	[9]
sub #1 a	9	sub #1 a	[a]	[a]	[10]
L4:					
L2:					
Cmp b #10	10	cmp b #10	[b]	[]	[11]
Blt L1	11	blt L1	[]	[]	[2, 12]

1) Point: any location between adjacent nodes.
2) Path: a sequence of points traversing through CFG.
3) Live: A variable is live immediately after a node n if it is live before any of n's successors. A variable is live before a node n if it's used by n, or it's alive after n and **IS NOT** overwritten by n.
4) Live Out: If the temporary is live / used in any nodes afterwards: $LiveOut(n) = \bigcup_{s \in succ(n)} LiveIn(s)$
"Do we need to keep the temporary alive after this node?"
5) Live In: If the temporary is used by our current node, or is liveout after the current node unless our current node defines that temporary:
LiveIn(n) = uses(n) ∪ (LiveOut(n) – defines(n))
Examples: b is liveout from live 1, as used by line 10. b is live from 2, as its used on the path following 8. b is **NOT** live out from node 4 as we do mov a b – overwriting our old b (so it's GONE). The new b we use on that path is a different one.

The whole point is we define a set for each node,
LiveIn(n) (temporaries alive immediately before n) and
LiveOut(n) (temporaries alive immediately after n).
Iterative code for Live Ranges

```
for node in CFG {
  LiveIn(node) = {}; LiveOut(node) = {}
}
repeat {
  for each n in CFG {
    LiveIn(n) = uses(n) U (LiveOut(n) - defs(n))
    LiveOut(n) = U_{s ∈ succ(n)} LiveIn(s)
  }
} until LiveIn and LiveOut do not change
```

To improve this method, to update the nodes from last → first (as data propagates from back to front – as we use successors – so this is more efficient).

9) Loop Invariant Code Motion

An instruction is loop-invariant if its operands are only defined **outside of the loop.** Hence the value it defines is loop-invariant (same for every iteration) and hence it may be possible to move instruction outside the loop.

9.1) Finding Reaching Definitions (Forward DFA)

Formally we attempt to find definition nodes of the form:

$$\underset{\text{Node ID}}{d} : t_d := (\underset{\text{Binary Op}}{u_1 \bullet u_2} \mid \underset{\text{Copy Op}}{u_1} \mid \underset{\text{Constant}}{c})$$

Where t_d is the destination, and u_i is for temporary variables used. d is **loop-invariant** if every definition of a $u_i \in$ uses(d) that **reaches** d is outside the loop.
Reaching definitions: A definition d reaches p if there is a path d→p where d isn't killed.
Gen(n) = {n} = set of defs generated by the node
Kill(n) = Set of all def of t except for n
ReachIn(n) = Set of definitions reaching up to n
ReachOut(n) = Set of definitions reaching after n

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

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

Informal Algorithm
Initialise ReachIn(n) and ReachOut(n) to { }
Iterate, updating ReachIn(n) and ReachOut(n) using definitions above, until convergence.
At each step, the sets increase in size

From our reaching definitions, we can reduce each set to the **relevant reaching definitions**, by considering only the reachins that are actually used by the instruction (for operands)

Lines of Code	Reaching definitions (RDs)	Relevant RDs
1. x=1	[]	[]
2: w=100	[1]	[]
3: z=200	[3, 1, 2]	[]
4: x=x+1	[1, 2, 3, 4, 5]	[1, 4]
5: y=w+z	**[2, 3, 4, 5]**	**[2, 3]**
6: if (x<10)	[2, 3, 4, 5]	[4]

go back to 4
else continue
We just found that all the definitions used by node 5 lie **outside the loop**!! We can hoist.

9.2) Identifying Loops

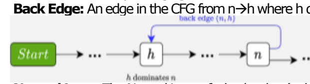
A loop in a control flow graph is a set of nodes S including a header node h, with the following properties: 1) From any node in S there is a path leading to h
2) There is a path from h to any node in S
3) There is no edge from any node outside S to any node in S other than h
So there's only way in, through a node in S, and h can lead to all of the nodes back in
Dominators: A node d dominates a node n if every path from the CFG's start node to n must go through d. Every node dominates itself.

M8) Finding all the dominators of a node

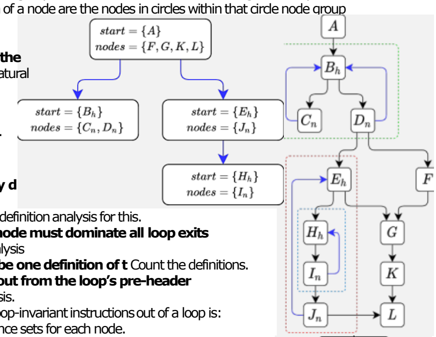
1. Set all Dom sets to the set of all nodes. Set the start node's dom to be itself.
2. Apply the Doms rule.

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

As the start node will have a set of {start} this will propagate, reducing the sizes of the sets for other nodes.
3. Once the sets stop changing, we have our solution.
Back Edge: An edge in the CFG from n→h where h dominates n is a back edge:



Natural Loop: The Natural Loop of a back edge (n, h) is the set of nodes S such that:
1) All nodes x ∈ S are dominated by h
2) For all nodes x ∈ S (except h), there is a path from x → n that does not contain h. This represents a loop, with the header node h.
Multiple Loops can share the same header. But this is not obvious from the CFG.
If we have a natural loop in another, then this is a nested loop.
Control Tree
We can construct a tree to show which loops are nested, what the headers and final nodes in each loop are. We then group our nodes with the closest enclosing circle they're and draw the control tree (children of a node are the nodes in circles within that circle node group



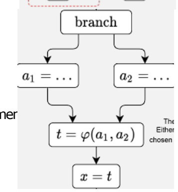
Pre Header:
A node inserted immediately **before the header node** of a natural loop.

9.3) Hoisting Instructions

The Conditions for Hoisting are:
1. **All reaching definitions used by d occur outside the Loop:** Use reaching definition analysis for this.
2. Loop invariant node must dominate all loop exits
Use dominators analysis
3. There can only be one definition of t Count the definitions.
4. t cannot be liveout from the loop's pre-header
Use live range analysis.
Process of hoisting loop-invariant instructions out of a loop is:
1. Compute dominance sets for each node.
2. Use dominance sets to identify natural loop and their headers.
3. Compute the reaching sets for nodes.
4. Use relevant reaching definitions to identify loop-invariant code.
5. Attempt the loop invariant code to a pre-header.
6. Check that the semantics of the program are not altered.

9.3) Static Single Assignment (SSA)

An IR which avoids side conditions by only allowing a single assignment temporary. Each **Reassignment of a variable is renamed**, this splits all live ranges. *Each variable has only one reaching definition.*
The phi function is used for branching. A phi statement ϕ(a1, a2) means either a1 or a2 could be used.



Once in SSA form, we can reassess the requirements for hoisting:
1. All reaching definitions used by d occur outside the loop (Same as prior to SSA). Use reaching definitions analysis for this.
2. Loop invariant node must dominate all loop exits No longer an issue.
3. There can only be one definition of t guaranteed by SSA form.
4. t cannot be liveout from the loop's pre-header Cannot occur with SSA due to single assignment.