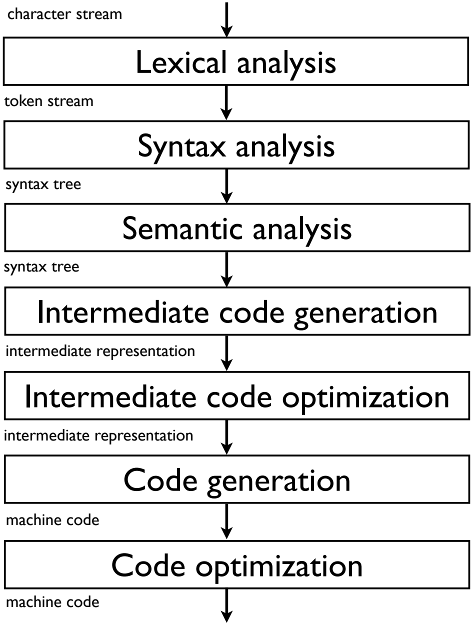
Part 6

Code generation

Structure of a compiler



Final code generation

- At this point, we have optimized intermediate code, from which we would like to generate the final code
- By final code, we typically mean assembly language of the target machine
- Goal of this stage:
    - Choose the appropriate machine instructions to translate each intermediate representation instruction
    - Handle finite machine resources (registers, memory, etc.)
    - Implement low-level details of the run-time environment
    - Implement machine-specific code optimization
- This step is very machine-specific
- In this course, we will only mention some typical and general problems

Short tour on machine code

- RISC (Reduced Instruction Set Computer)
  - E.g.: PowerPC, Sparc, MIPS (embedded systems), ARM...
  - Many registers, 3-address instructions, relatively simple instruction sets
- CISC (Complex Instruction Set Computer)
  - E.g.: x86, x86-64, amd64...
  - Few registers, 2-address instructions, complex instruction sets
- Stack-based computer:
  - E.g.: Not really used anymore but Java's virtual machine is stack-based
  - No register, zero address instructions (operands on the stack)
- Accumulator-based computer:
  - E.g.: First IBM computers were accumulator-based
  - One special register (the accumulator), one address instructions, other registers used in loops and address specification

Outline

Instruction selection

- One needs to map one or several instructions of the intermediate representation into one or several instructions of the machine language
- Complexity of the task depends on:
    - the level of the IR
    - the nature of the instruction-set architecture
    - the desired quality of the generated code
- Examples of problems:
    - Conditional jumps
    - Constants
    - Complex instructions

Example: Conditional jumps

- Conditional jumps in our intermediate language are of the form:

  IF id relop *Atom* THEN labelid ELSE labelid

- Conditional jumps might be different on some machines:
  - One-way branch instead of two-way branches

    IF c THEN l_t ELSE l_f

    ```
    branch_if_c   l_r
    jump          l_f
    ```
  - Condition such as "id relop *Atom*" might not be allowed. Then, compute the condition and store it in a register
  - There might exist special registers for conditions
  - ...

Example: Constants

- IR allows arbitrary constants as operands to binary or unary operators
- This is not always the case in machine code
  - MIPS allows only 16-bit constants in operands (even though integers are 32 bits)
  - On the ARM, a constant can only be a 8-bit number positioned at any even bit boundary (within a 32-bit word)
- If a constant is too big, translation requires to build the constant into some register
- If the constant is used within a loop, its computation should be moved outside

Exploiting complex instructions

- If we do not care about efficiency, instruction selection is straightforward:
    - Writes a code skeleton for every IR instructions
    - Example in MIPS assembly:

      $$t_2 := t_1 + 116 \quad \Rightarrow \quad \text{addi r2,r1,116}$$

      (where r2 and r1 are the registers chosen for t_2 and t_1)

- Most processors (even RISC-based) have complex instructions that can translate several IR instructions at once
    - Examples in MIPS assembly:

      $$\begin{array}{lll} t_2 := t_1 + 116 & \Rightarrow & \text{lw r3, 116(r1)} \\ t_3 := M[t_2] & & \end{array}$$

      (where r3 and r1 are the registers chosen for t_3 and t_1 resp. and assuming that t_2 will not be used later)

- For efficiency reason, one should exploit them

Code generation principle

- Determine for each variable whether it is dead after a particular use (liveness analysis, see later)

$$\begin{aligned} t_2 &:= t_1 + 116 \\ t_3 &:= M[t_2^{last}] \end{aligned}$$

- Associate an address (register, memory location...) to each variable (register allocation, see later)
- Define an instruction set description, i.e., a list of pairs of:
  - pattern: a sequence of IR instructions

$$\begin{aligned} t &:= r_s + k \\ r_t &:= M[t^{last}] \end{aligned}$$

  - replacement: a sequence of machine-code instruction translating the pattern

$$\text{lw } r_t, k(r_s)$$

- Use pattern matching to do the translation

Illustration

Pattern/replacement pairs for a subset of the MIPS instruction set

$t := r_s + k,$ $r_t := M[t^{last}]$	`lw`	$r_t,\ k(r_s)$
$r_t := M[r_s]$	`lw`	$r_t,\ 0(r_s)$
$r_t := M[k]$	`lw`	$r_t,\ k(\texttt{R0})$
$t := r_s + k,$ $M[t^{last}] := r_t$	`sw`	$r_t,\ k(r_s)$
$M[r_s] := r_t$	`sw`	$r_t,\ 0(r_s)$
$M[k] := r_t$	`sw`	$r_t,\ k(\texttt{R0})$
$r_d := r_s + r_t$	`add`	r_d, r_s, r_t
$r_d := r_t$	`add`	$r_d, \texttt{R0}, r_t$
$r_d := r_s + k$	`addi`	r_d, r_s, k
$r_d := k$	`addi`	$r_d, \texttt{R0}, k$

MIPS instructions:

- lw r,k(s): $r = M[s+k]$
- sw r,k(s): $M[s+k] = r$
- add r,s,t: $r = s + t$
- addi r,s,k: $r = s + k$ where k is a constant
- R0: a register containing the constant 0

(Mogensen)

Illustration

IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$,		beq	$r_s, r_t, label_t$
LABEL $label_f$	$label_f$:		
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$,		bne	$r_s, r_t, label_f$
LABEL $label_t$	$label_t$:		
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$		beq	$r_s, r_t, label_t$
		j	$label_f$
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$,		slt	r_d, r_s, r_t
LABEL $label_f$		bne	r_d, R0, $label_t$
	$label_f$:		
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$,		slt	r_d, r_s, r_t
LABEL $label_t$		beq	r_d, R0, $label_f$
	$label_t$:		
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$		slt	r_d, r_s, r_t
		bne	r_d, R0, $label_t$
		j	$label_f$
LABEL $label$	$label$:		

MIPS instructions:

- beq r,s,lab: branch to l if r=s
- bnq r,s,lab: branch to l if r≠s
- slt r,s,t: $d = (s < t)$
- j l: unconditional jump

(Mogensen)

Pattern matching

- A pattern should be defined for every single IR instruction (otherwise it would not be possible to translate some IR code)
- A *last* in a pattern can only be matched by a *last* in the IR code
- But any variable in a pattern can match a *last* in the IR code
- If patterns overlap, there are potentially several translations for the same IR code
- On wants to find the best possible translation (e.g., the shortest or the fastest)
- Two approaches:
  - Greedy: order the pairs so that longer patterns are listed before shorter ones and at each step, use the first pattern that matches a prefix of the IR code
  - Optimal: associate a cost to each replacement and find the translation that minimizes the total translation cost, e.g. using dynamic programming

Illustration

Using the greedy approach:

IR code			MIPS code
$a := a + b^{last}$			add a, a, b
$d := c + 8$			sw $a, 8(c)$
$M[d^{last}] := a$	$\Rightarrow$		
IF $a = c$ THEN $label_1$ ELSE $label_2$			beq $a, c, label$
LABEL $label_2$		$label_2$:	

Outline

Register allocation

- In the IR, we assumed an unlimited number of registers (to ease IR code generation)
- This is obviously not the case on a physical machine (typically, from 5 to 10 general-purpose registers)

- Registers can be accessed quickly and operations can be performed on them directly
- Using registers intelligently is therefore a critical step in any compiler (can make a difference in orders of magnitude)

- Register allocation is the process of assigning variables to registers and managing data transfer in and out of the registers

Challenges in register allocation

- Registers are scarce
  - Often substantially more IR variables than registers
  - Need to find a way to reuse registers whenever possible
- Register management is sometimes complicated
  - Each register is made of several small registers (x86)
  - There are specific registers which need to be used for some instructions (x86)
  - Some registers are reserved for the assembler or operating systems (MIPS)
  - Some registers must be reserved to handle function calls (all)
- Here, we assume only some number of indivisible, general-purpose registers (MIPS-style)

A direct solution

- Idea: store every value in main memory, loading values only when they are needed.
- To generate a code that performs some computation:
  - Generate load instructions to retrieve the values from main memory into registers
  - Generate code to perform the computation on the registers
  - Generate store instructions to store the result back into main memory
- Example: (with a,b,c,d stored resp. at fp-8, fp-12, fp-16, fp-20)

$$\begin{array}{l} a := b + c \\ d := a \\ c := a + d \end{array} \quad \Rightarrow \quad \begin{array}{l} \text{lw } t_0, -12(fp) \\ \text{lw } t_1, -16(fp) \\ \text{add } t_2, t_0, t_1 \\ \text{sw } t_2, -8(fp) \\ \hline \text{lw } t_0, -8(fp) \\ \text{sw } t_0, -20(fp) \\ \hline \text{lw } t_0, -8(fp) \\ \text{lw } t_1, -20(fp) \\ \text{add } t_2, t_0, t_1 \\ \text{sw } t_2, -16(f) \end{array}$$

A direct solution

- Advantage: very simple, translation is straighforward, never runout of registers
- Disadvantage: very inefficient, waste space and time

- Better allocator should:
  - try to reduces memory load/store
  - reduce total memory usage
- Need to answer two questions:
  - Which register do we put variables in?
  - What do we do when we run out of registers?

Liveness analysis

- A variable is live at some point in the program if its value may be read later before it is written. It is dead if there is no way its value can be used in the future.
- Two variables can share a register if there is no point in the program where they are both live
- Liveness analysis is the process of determining the live or dead statuses of all variables throughout the (IR) program

- Informally: For an instruction I and a variable t
  - If t is used in I, then t is live at the start of I
  - If t is assigned a value in I (and does not appear in the RHS of I), then t is dead at the start of the I
  - If t is live at the end of I and I does not assign a value to t, then t is live at the start of I
  - t is live at the end of I if it is live at the start of any of the immediately succeding instructions

Liveness analysis: control-flow graph

First step: construct the control-flow graph

- For each instruction numbered i, one defines $succ[i]$ as follows:
  - If instruction j is just after i and j is neither a GOTO or IF-THEN-ELSE instruction, then j is in $succ[i]$
  - If i is of the form GOTO l, the instruction with label l is in $succ[i]$.
  - If i is IF p THEN l_t ELSE l_f, instructions with label l_t and l_f are both in $succ[i]$

- The third rule loosely assumes that both outcomes of the IF-THEN-ELSE are possible, meaning that some variables will be claimed live while they are dead (not really a probblem)

Liveness analysis: control-flow graph

Example (Computation of Fibonacci(n) in a)

```
1:   a := 0
2:   b := 1
3:   z := 0
4:   LABEL loop
5:   IF n = z THEN end ELSE body
6:   LABEL body
7:   t := a + b
8:   a := b
9:   b := t
10:  n := n − 1
11:  z := 0
12:  GOTO loop
13:  LABEL end
```

i	$succ[i]$
1	2
2	3
3	4
4	5
5	6, 13
6	7
7	8
8	9
9	10
10	11
11	12
12	4
13	

Liveness analysis: *gen* and *kill*

For each IR instruction, we define two functions:

- *gen*[*i*]: set of variables that may be read by instruction *i*
- *kill*[*i*]: set of variables that may be assigned a value by instruction *i*

Instruction i	$gen[i]$	$kill[i]$
LABEL l	$\emptyset$	$\emptyset$
$x := y$	$\{y\}$	$\{x\}$
$x := k$	$\emptyset$	$\{x\}$
$x :=$ **unop** y	$\{y\}$	$\{x\}$
$x :=$ **unop** k	$\emptyset$	$\{x\}$
$x := y$ **binop** z	$\{y,z\}$	$\{x\}$
$x := y$ **binop** k	$\{y\}$	$\{x\}$
$x := M[y]$	$\{y\}$	$\{x\}$
$x := M[k]$	$\emptyset$	$\{x\}$
$M[x]$:= y	$\{x,y\}$	$\emptyset$
$M[k]$:= y	$\{y\}$	$\emptyset$
GOTO l	$\emptyset$	$\emptyset$
IF x **relop** y THEN l_t ELSE l_f	$\{x,y\}$	$\emptyset$
$x :=$ CALL $f(args)$	$args$	$\{x\}$

Liveness analysis: *in* and *out*

- For each program instruction i, we use two sets to hold liveness information:
  - $in[i]$: the variables that are live before instruction i
  - $out[i]$: the variables that are live at the end of i
- *in* and *out* are defined by these two equations:

$$\begin{aligned} in[i] &= gen[i] \cup (out[i] \setminus kill[i]) \\ out[i] &= \bigcup_{j \in succ[i]} in[j] \end{aligned}$$

- These equations can be solved by fixed-point iterations:
  - Initialize $in[i]$ and $out[i]$ to empty sets
  - Iterate over instructions (in reverse order, evaluating *out* first) until convergence (i.e., no change)
- For the last instruction ($succ[i] = \emptyset$), $out[i]$ is set of variables that are live at the end of the program (i.e., used subsequently)

Illustration

```
1:   $a := 0$
2:   $b := 1$
3:   $z := 0$
4:   LABEL $loop$
5:   IF $n = z$ THEN $end$ ELSE $body$
6:   LABEL $body$
7:   $t := a + b$
8:   $a := b$
9:   $b := t$
10:  $n := n - 1$
11:  $z := 0$
12:  GOTO $loop$
13:  LABEL $end$
```

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2		a
2	3		b
3	4		z
4	5		
5	6, 13	n, z	
6	7		
7	8	a, b	t
8	9	b	a
9	10	t	b
10	11	n	n
11	12		z
12	4		
13			

(Mogensen)

Illustration

	Initial		Iteration 1		Iteration 2		Iteration 3	
i	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$
1			n,a	n	n,a	n	n,a	n
2			n,a,b	n,a	n,a,b	n,a	n,a,b	n,a
3			n,z,a,b	n,a,b	n,z,a,b	n,a,b	n,z,a,b	n,a,b
4			n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b
5			a,b,n	n,z,a,b	a,b,n	n,z,a,b	a,b,n	n,z,a,b
6			a,b,n	a,b,n	a,b,n	a,b,n	a,b,n	a,b,n
7			b,t,n	a,b,n	b,t,n	a,b,n	b,t,n	a,b,n
8			t,n	b,t,n	t,n,a	b,t,n	t,n,a	b,t,n
9			n	t,n	n,a,b	t,n,a	n,a,b	t,n,a
10				n	n,a,b	n,a,b	n,a,b	n,a,b
11					n,z,a,b	n,a,b	n,z,a,b	n,a,b
12					n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b
13			a	a	a	a	a	a

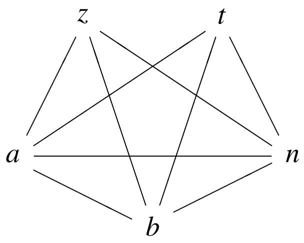
(Mogensen)

Interference

- A variable x interferes with another variable y if there is an instruction i such that $x \in kill[i]$, $y \in out[i]$ and instruction i is not $x := y$
- Note:
  - Different from $x \in out[i]$ and $y \in out[i]$:
  - if x is in $kill[i]$ and not in $out[i]$ (because x is never used after an assignment), then it should interfere with $y \in out[i]$ (to allow side-effects)
- Interference graph: undirected graph where each node is a variable and two variables are connected if they interfere

Illustration

Instruction	Left-hand side	Interferes with
1	a	n
2	b	n, a
3	z	n, a, b
7	t	b, n
8	a	t, n
9	b	n, a
10	n	a, b
11	z	n, a, b



(Mogensen)

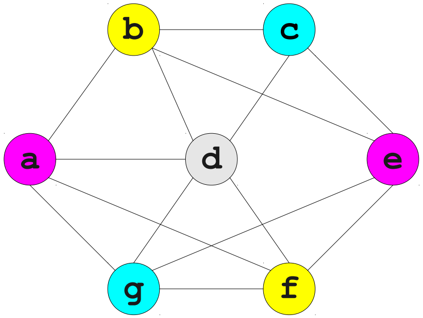
Register allocation

- Global register allocation: we assign to a variable the same register throughout the program (or procedure)
- How to do it? Assign a register number (among N) to each node of the interference graph such that
  - Two nodes that are connected have different register numbers
  - The total number of different register is no higher than the number of available registers
- This is a problem of graph colouring (where colour number = register number), which is known to be NP-complete
- Several heuristics have been proposed

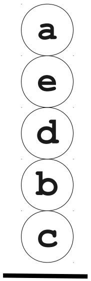
Chaitin's algorithm

- A heuristic linear algorithm for k-coloring a graph
- Algorithm:
  - Select a node with fewer than k outgoing edges
  - Remove it from the graph
  - Recursively color the rest of the graph
  - Add the node back in
  - Assign it a valid color
- Last step is always possible since the removed node has less than k neighbors in the graph
- Implementation: nodes are pushed on a stack as soon as they are selected

Illustration



Stack of nodes



Registers

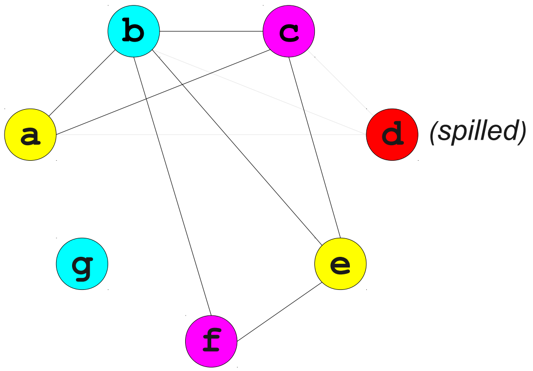
R_0	R_1	R_2	R_3

(Keith Schwarz)

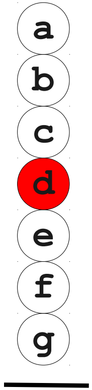
Chaitin's algorithm

- What if we can not find a node with less than k neighbors?
- Choose and remove an arbitrary node, marking it as "troublesome"
- When adding node back in, it may still be possible to find a valid color
- Otherwise, we will have to store it in memory.
    - This is called spilling.

Illustration



Stack of nodes



Registers

R_0	R_1	R_2

(Keith Schwarz)

Spilling

- A spilled variable is stored in memory
- When we need a register for a spilled variable v, temporarily evict a register to memory (since registers are supposed to be exhausted)
- When done with that register, write its value to the storage spot for v (if necessary) and load the old value back

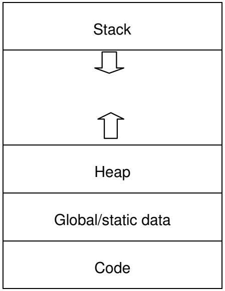
- Heuristics to choose the variable/node to spill:
  - Pick one with close to N neighbors (increasing the chance to color it)
  - Choose a node with many neighbors with close to N neighbors (increase the chance of less spilling afterwards)
  - Choose a variable that's not costly to spill (by looking at the program)

Register allocation

- We only scratched the surface of register allocation
- Many heuristics exist as well as different approaches (not using graph coloring)
- GCC uses a variant of Chaitin's algorithm

Outline

Memory organization



Memory is generally divided into four main parts:

- Code: contains the code of the program
- Static data: contains static data allocated at compile-time
- Stack: used for function calls and local variables
- Heap: for the rest (e.g., data allocated at run-time)

Computers have registers that contain addresses that delimits these different parts

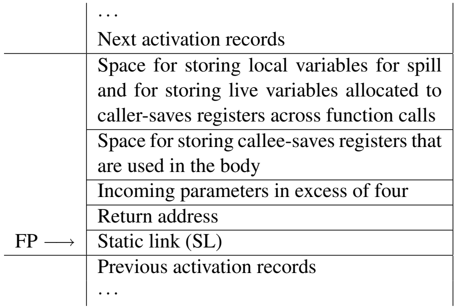
Static data

- Contains data allocated at compile-time
- Address of such data is then hardwired in the generated code
- Used e.g. in C to allocate global variables
- There are facilities in assemblers to allocate such space:
  - Example to allocate an array of 4000 bytes

```
        .data          # go to data area for allocation
baseofA:               # label for array A
        .space 4000    # move current-address pointer up 4000 bytes
        .text          # go back to text area for code generation
```

- Limitations:
  - size of the data must be known at compile-time
  - Never freed even if the data is only used a fraction of time

Stack


...
Next activation records
Space for storing local variables for spill and for storing live variables allocated to caller-saves registers across function calls
Space for storing callee-saves registers that are used in the body
Incoming parameters in excess of four
Return address
FP ⟶
Static link (SL)
Previous activation records
...

- Mainly used to store activation records for function calls
- But can be used to allocate arrays and other data structures (e.g., in C, to allocate *local* arrays)

- Allocation is quick and easy
- But sizes of arrays need to be known at compile-time and can only be used for local variables (space is freed when the function returns)

Heap

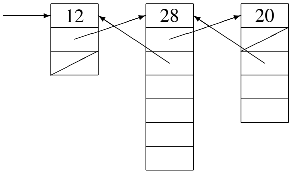
- Used for dynamic memory allocations
- Size of arrays or structures need not to be known at compile-time
- Array sizes can be increased dynamically
- Two ways to manage data allocation/deallocation:
  - Manual memory management
  - Automatic memory management (or garbage collection)

Manual memory management

- The user is responsible for both data allocation and deallocation
  - In C: malloc and free
  - In object oriented languages: object constructors and destructors
- Advantages:
  - Easier to implement than garbage collection
  - The programmer can exercice precise control over memory usage (allows better performances)
- Limitations
  - The programmer *has to* exercice precise control over memory usage (tedious)
  - Easily leads to troublesome bugs: memory leaks, double frees, use-after-frees...

A simple implementation

- Space is allocated by the operating system and then managed by the program (through library functions such as malloc and free in C)
- A free list is maintained with all current free memory blocks (initially, one big block)



- Malloc:
  - Search through the free list for a block of sufficient size
  - If found, it is possibly split in two with one removed from free list
  - If not found, ask operating system for a new chunck of memory
- Free:
  - Insert the block back into the free list
- Allocation is linear in the size of the free list, deallocation is done in constant time

A simple implementation

- Block splitting leads to memory fragmentation
  - The free list will eventually accumulate many small blocks
  - Can be solved by joining consecutive freed blocks
  - Makes free linear in free list size
- Complexity of malloc can be reduced
  - Limit block sizes to power of 2 and have a free list for each size
  - Makes malloc logarithmic in heap size
- Array resizing can be allowed by using indirection nodes
  - When array is resized, it is copied into a new (bigger) block
  - Indirection node address is updated accordingly

Garbage collection

- Allocation is still done with malloc or object constructors but memory is automatically reclaimed
  - Data/Objects that won't be used again are called garbage
  - Reclaiming garbage objects automatically is called garbage collection

- Advantages:
  - Programmer does not have to worry about freeing unused resources
- Limitations:
  - Programmer can't reclaim unused resources
  - Difficult to implement and add a significant overhead

Implementation 1: reference counting

- Idea: if no pointer to a block exists, the block can safely be freed
- Add an extra field in each memory block (of the free list) with a count of the incoming pointers
  - When creating an object, set its counter to 0
  - When creating a reference to an object, increment its counter
  - When removing a reference, decrement its counter.
  - If zero, remove all outgoing references from that object and reclaim the memory

Reference counting: illustration

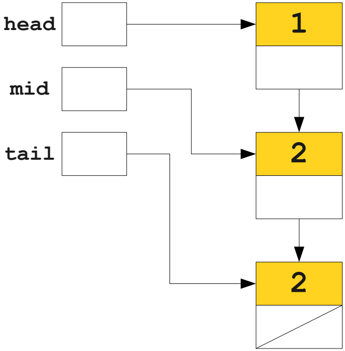
```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
   ......................................
    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



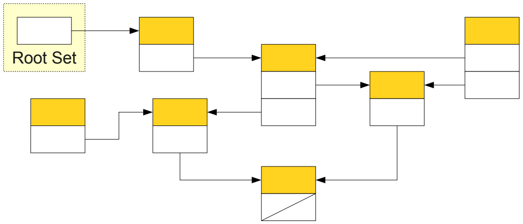
(Keith Schwarz)

Reference counting

- Straightforward to implement and can be combined with manual memory management
- Significant overhead when doing assignements for incrementing counters
- Impose constraints on the language
  - No pointer to the middle of an object, should be able to distinguish pointers from integers...
- Can not handle circular data structures
  - As counters will never be zero
  - E.g., doubly-linked lists

Implementation 2: tracing garbage collectors

- Idea: find all reachable blocks from the knowledge of what is immediately accessible (the root set) and free all other blocks
- The root set is the set of memory locations that are known to be reachable
    - all variables in the program: registers, stack-allocated, global variables...
- Any objects (resp. not) reachable from the root set are (resp. not) reachable



Tracing garbage collection: mark-and-sweep

- Mark-and-sweep garbage collection:
    - Add a flag to each block
    - Marking phase: go through the graph, e.g., depth-first, setting the flag for all reached blocks
    - Sweeping phase: go through the list of blocks and free all unflagged ones
- Implementation of the mark stage with a stack:
    - Initialized to the root set
    - Retaining reachable blocks that have not yet been visited
- Tracing GC is typically called only when a malloc fails to avoid pauses in the program

- Problem: stack requires memory (and a malloc has just failed)
    - Marking phase can be implemented without a stack (at the expense of computing times)
    - Typically by adding descriptors within blocks

Implementation: tracing garbage collection

- Advantage:
  - More precise than reference counting
  - Can handle circular references
  - Run time can be made proportional to the number of reachable objects (typically much lower than number of free blocks)
- Disadvantages:
  - Introduce huge pause times
  - Consume lots of memory

Garbage collection

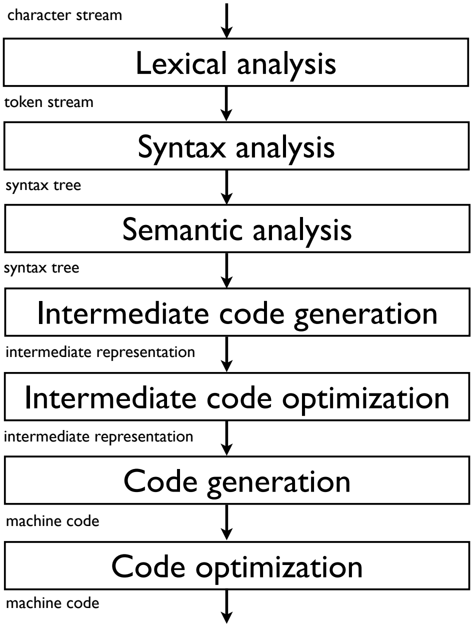
Other garbage collection methods:

- Two-space collection (stop-and-copying):
  - Alternative to free lists
  - Two allocation spaces of same size are maintained
  - Blocks are always allocated in one space until full
  - Garbage collection then copies all live objects to the other space and swap their roles
- Generational collection:
  - Maintain several spaces for different generations of objects, with these spaces of increasing sizes
  - Optimized according to the "objects die young" principle
- Concurrent and incremental collectors
  - Perform collection incrementally or concurrently during execution of the program
  - Avoid long pauses but can reduce the total throughput

Part 7

Conclusion

Structure of a compiler



Summary

- Part 1, Introduction:
  - Overview and motivation...
- Part 2, Lexical analysis:
  - Regular expression, finite automata, implementation, Flex...
- Part 3, Syntax analysis:
  - Context-free grammar, top-down (predictive) parsing, bottom-up parsing (SLR and operator precedence parsing)...
- Part 4, Semantic analysis:
  - Syntax-directed translation, abstract syntax tree, type and scope checking...
- Part 5, Intermediate code generation and optimization:
  - Intermediate representations, IR code generation, optimization...
- Part 6, Code generation:
  - Instruction selection, register allocation, liveliness analysis, memory management...

More on compilers

- Our treatment of each compiler stage was superficial
- See reference books for more details (Transp. 4)

- Some things we have not discussed at all:
  - Specificities of object-oriented or functional programming languages
  - Machine dependent code optimization
  - Parallelism
  - ...

- Related topics:
  - Natural language processing
  - Domain-specific languages
  - ...