Compiler Construction Lent Term 2016

Part III : Lectures 13 – 16

- 13 : Compilers in their OS context
- 14 : Assorted Topics
- 15 : Runtime memory management
- 16 : Bootstrapping a compiler

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

Lecture 13

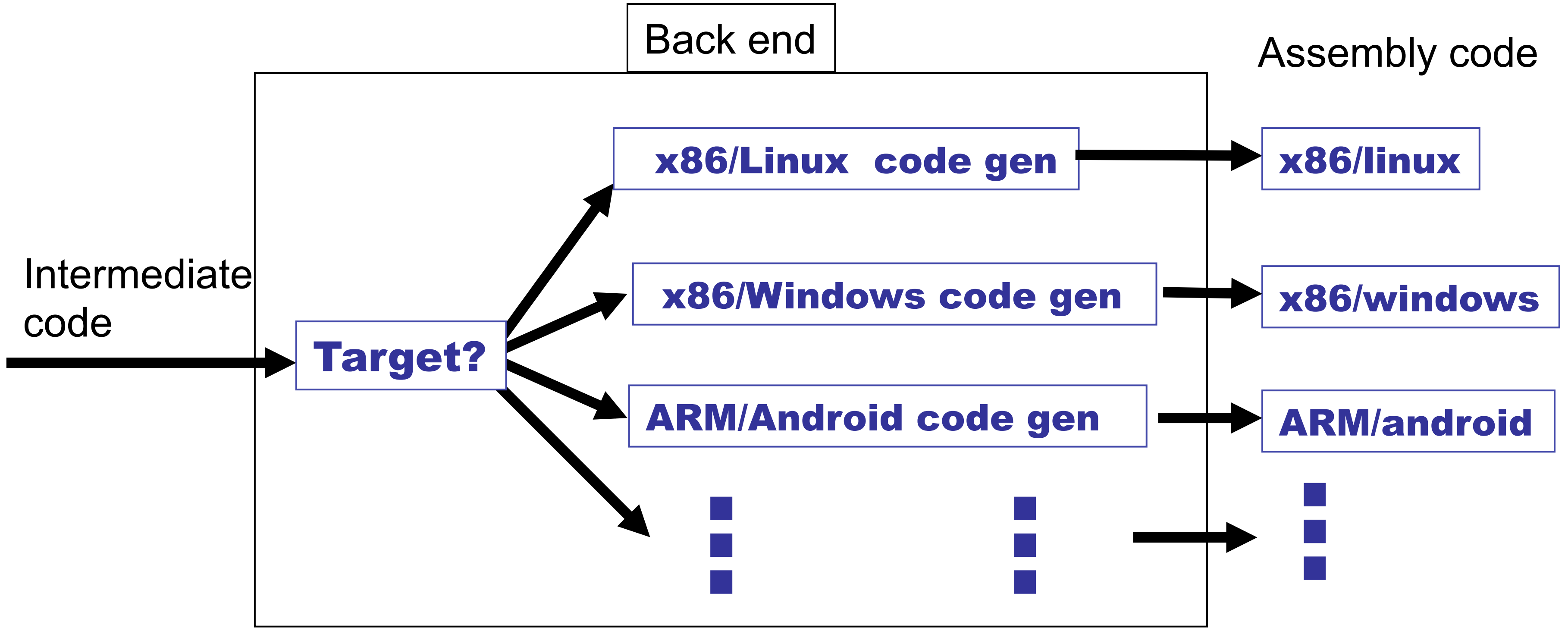
- Code generation for multiple platforms.
- Assembly code
- Linking and loading
- The Application Binary Interface (ABI)
- Object file format (only ELF covered)
- A crash course in x86 architecture and instruction set
- Naïve generation of x86 code from Jargon VM instructions

We could implement a Jargon byte code interpreter …

```
…
…
void vsm_execute_instruction(vsm_state *state, bytecode instruction)
{
 opcode code   = instruction.code;
 argument arg1 = instruction.arg1;
 switch (code) {
      case PUSH: { state->stack[state->sp++] = arg1; state->pc++; break; }
      case POP : { state->sp--; state->pc++; break; }
      case GOTO: { state->pc = arg1; break; }
      case STACK_LOOKUP: {
               state->stack[state->sp++] =
             state->stack[state->fp + arg1];
               state->pc++;  break; }


      …
      …
   }
}
…
…
```

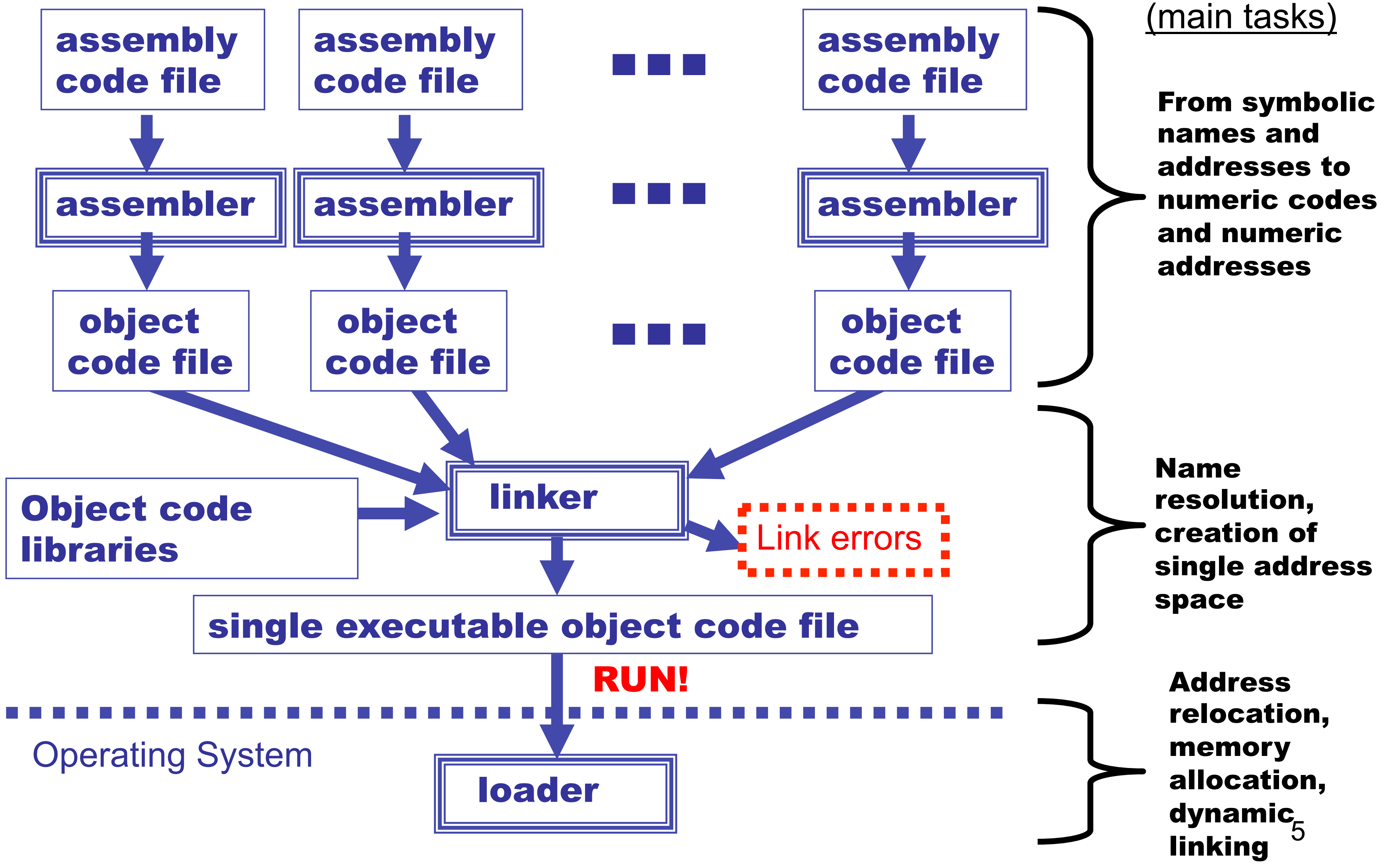
- Generate compact byte code for each Jargon instruction.
- Compiler writes byte codes to a file.
- Implement an interpreter in C or C++ for these byte codes.
- Execution is much faster than our jargon.ml implementation.
- **Or, we could generate assembly code from Jargon instructions ….**

Backend could target multiple platforms

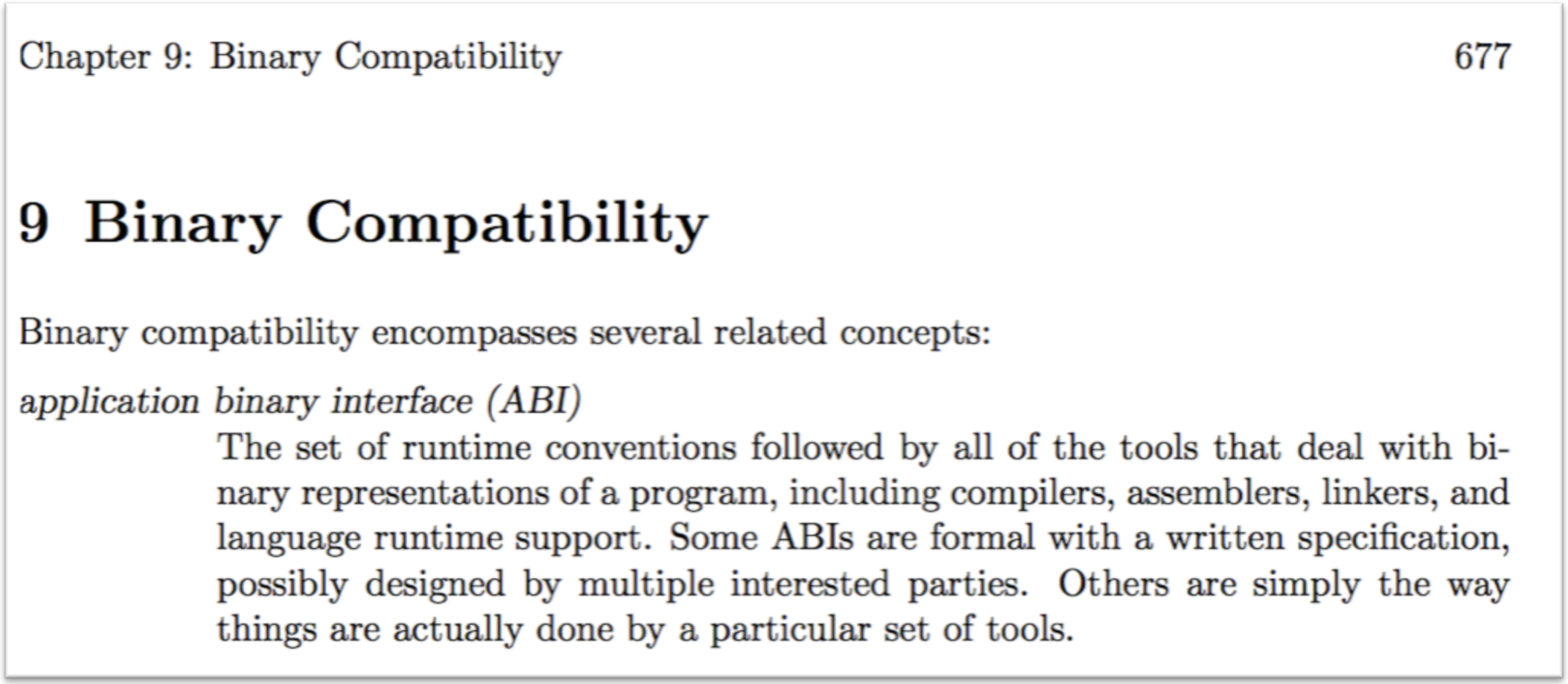


One of the great benefits of Virtual Machines is their portability. However, for more efficient code we may want to compile to assembler. Lost portability can be regained through the extra effort of implementing code generation for every desired target platform.

Assembly, Linking, Loading


assembly code file
assembly code file
assembly code file
assembler
assembler
assembler
object code file
object code file
object code file
Object code libraries
linker
Link errors
single executable object code file
RUN!
Operating System
loader
(main tasks)
From symbolic names and addresses to numeric codes and numeric addresses
Name resolution, creation of single address space
Address relocation, memory allocation, dynamic linking

The gcc manual (810 pages)
https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc.pdf



Chapter 9: Binary Compatibility 677

9 Binary Compatibility

Binary compatibility encompasses several related concepts:

application binary interface (ABI)
The set of runtime conventions followed by all of the tools that deal with binary representations of a program, including compilers, assemblers, linkers, and language runtime support. Some ABIs are formal with a written specification, possibly designed by multiple interested parties. Others are simply the way things are actually done by a particular set of tools.

Applications Binary Interface (ABI)

We will use x86/Unix as our running example.
Specifies many things, including the following.

- C calling conventions used for systems calls or calls to compiled C code.
  - Register usage and stack frame layout
  - How parameters are passed, results returned
  - Caller/callee responsibilities for placement and cleanup
- Byte-level layout and semantics of object files.
  - Executable and Linkable Format (ELF). Formerly known as Extensible Linking Format.
- Linking, loading, and name mangling

Note: the conventions are required for portable interaction with compiled C. Your compiled language does not have to follow the same conventions!

Object files

Must contain at least

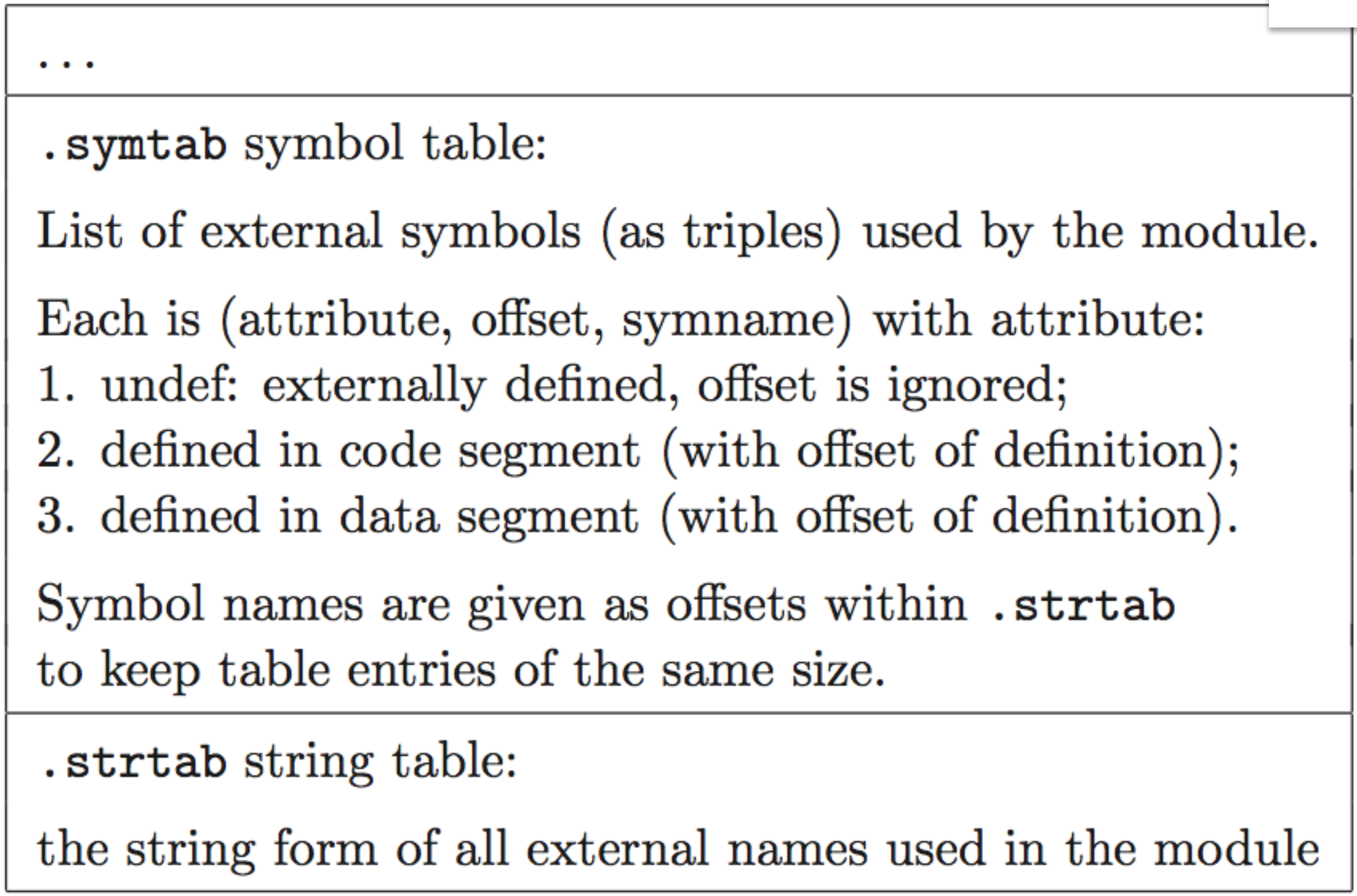
- Program instructions
- Symbols being exported
- Symbols being imported
- Constants used in the program (such as strings)

Executable and Linkable Format (ELF) is a common format for both linker input and output.

ELF details (1)

Header information; positions and sizes of sections
`.text` segment (code segment): binary data
`.data` segment: binary data
`.rela.text` code segment relocation table: list of (offset,symbol) pairs giving: (i) offset within `.text` to be relocated; and $(ii\,i)$ by which symbol
`.rela.data` data segment relocation table: list of (offset,symbol) pairs giving: (i) offset within `.data` to be relocated; and $(ii\,i)$ by which symbol
...

ELF details (2)



. . .
`.symtab` symbol table: List of external symbols (as triples) used by the module. Each is (attribute, offset, symname) with attribute: 1. undef: externally defined, offset is ignored; 2. defined in code segment (with offset of definition); 3. defined in data segment (with offset of definition). Symbol names are given as offsets within **`.strtab`** to keep table entries of the same size.
`.strtab` string table: the string form of all external names used in the module

The Linker

What does a linker do?

- takes some object files as input, notes all undefined symbols.
- recursively searches libraries adding ELF files which define such symbols until all names defined (“library search”).
- whinges if any symbol is undefined or multiply defined.

Then what?

- concatenates all code segments (forming the output code segment).
- concatenates all data segments.
- performs relocations (updates code/data segments at specified offsets.

Recently there had been renewed interest in optimization at this stage.

Dynamic vs. Static Loading

There are two approaches to linking:

Static linking (described on previous slide).

Problem: a simple “hello world” program may give a 10MB executable if it refers to a big graphics or other library.

Dynamic linking

Don’t incorporate big libraries as part of the executable, but load them into memory on demand. Such libraries are held as “.DLL” (Windows) or ”.so” (Linux) files.

Pros and Cons of dynamic linking:

(+) Executables are smaller

(+) Bug fixes to a library don’t require re-linking as the new versior is automatically demand-loaded every time the program is run.

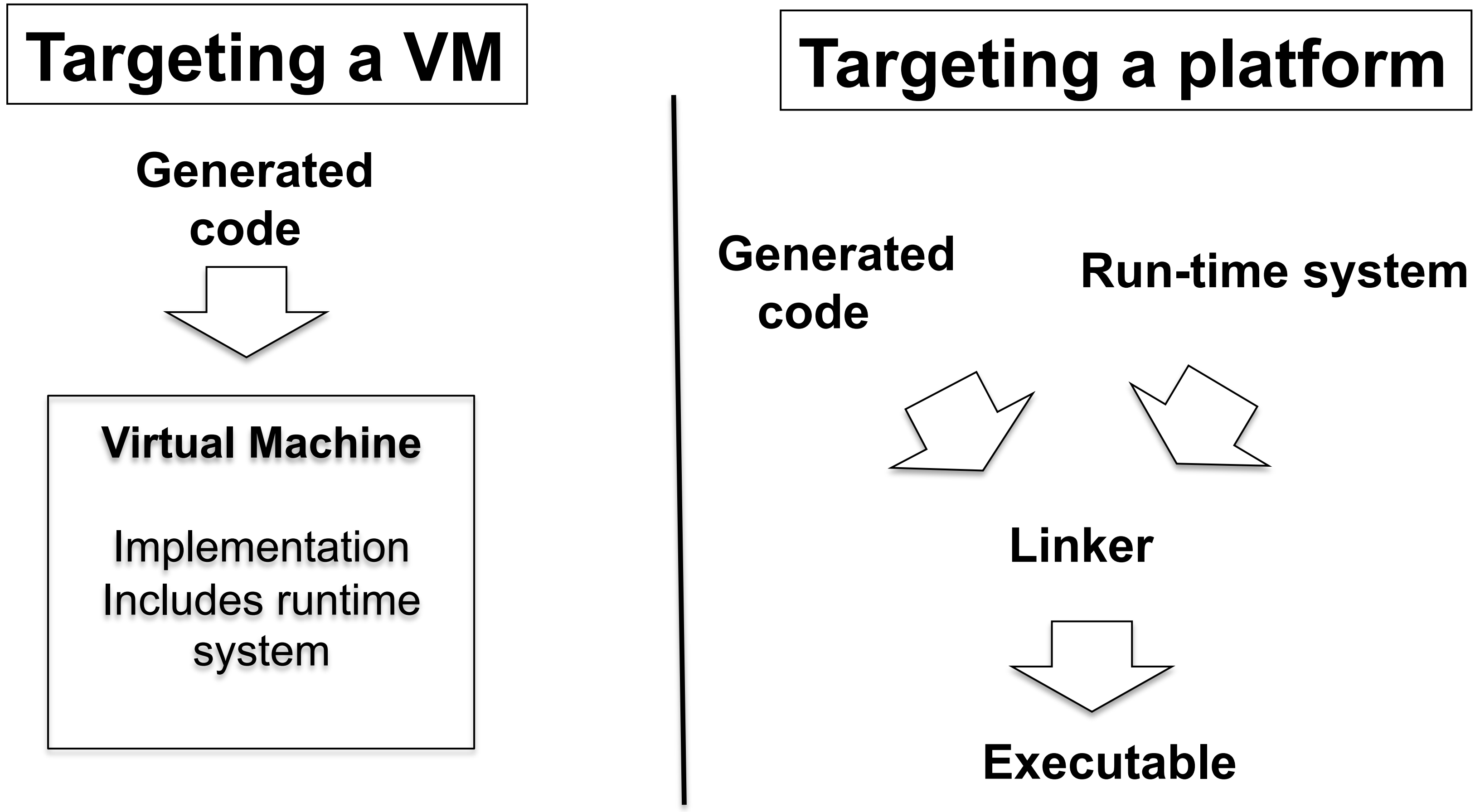
(-) Non-compatible changes to a library wreck previously working programs “DLL hell”.

A “runtime system”

A library implementing functionality needed to run compiled code on a given operating system. Normally tailored to the language being compiled.

- Implements interface between OS and language.
- May implement memory management.
- May implement “foreign function” interface (say we want to call compiled C code from Slang code, or vice versa).
- May include efficient implementations of primitive operations defined in the compiled language.
- For some languages, the runtime system may perform runtime type checking, method lookup, security checks, and so on.
- …

Runtime system

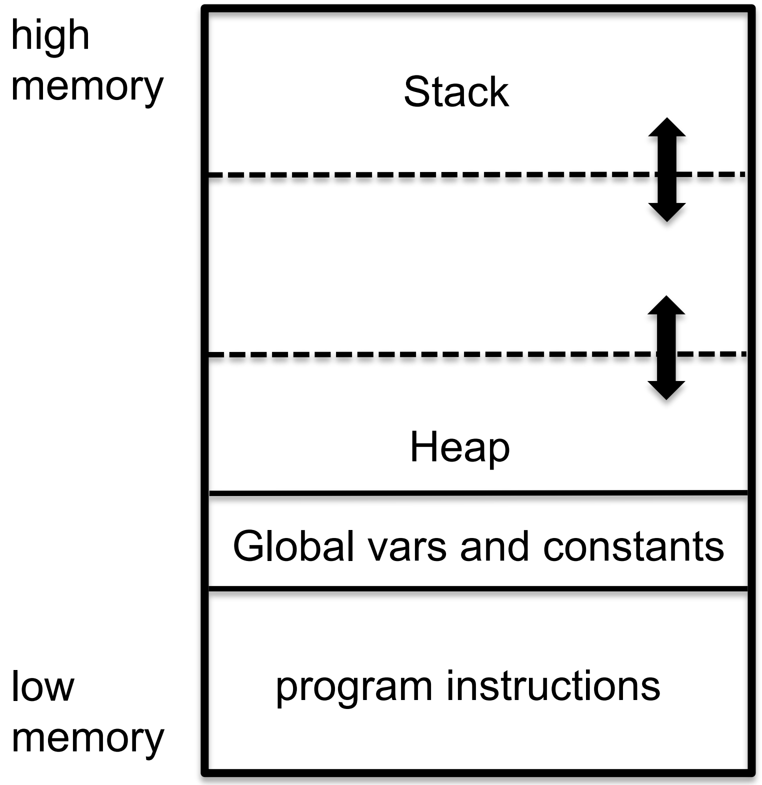


In either case, implementers of the compiler and the runtime system must agree on many low-level details of memory layout and data representation.

Typical (Low-Level) Memory Layout (UNIX)

Rough schematic of traditional layout in (virtual) memory.

Dealing with Virtual Machines allows us to ignore some of the low-level details….



The heap is used for dynamically allocating memory. Typically either for very large objects or for those objects that are returned by functions/procedures and must outlive the associated activation record.

In languages like Java and ML, the heap is managed automatically ("garbage collection")

A Crash Course in x86 assembler

- A CISC architecture
- There are 16, 32 and 64 bit versions
- 32 bit version :
  - General purpose registers : EAX EBX ECX EDX
  - Special purpose registers : ESI EDI EBP EIP ESP
    - EBP : normally used as the frame pointer
    - ESP : normally used as the stack pointer
    - EDI : often used to pass (first) argument
    - EIP : the code pointer
  - Segment and flag registers that we will ignore …
- 64 bit version:
  - Rename 32-bit registers with “R” (RAX, RBX, RCX, …)
  - More general registers: R8 R9 R10 R11 R12 R13 R14 R15

Register names can indicate “width” of a value.

rax : 64 bit version
eax : 32 bit version (or lower 32 bits of **rax**)
ax : 16 bit version (or lower 16 bits of **eax**)
al : lower 8 bits of ax
ah : upper 8 bits of ax

See https://en.wikibooks.org/wiki/X86_Assembly

The syntax of x86 assembler comes in several flavours.
Here are two examples of “put integer 4 into register eax”:

```
movl $4, %eax         // GAS (aka AT&T) notation
mov  eax, 4           // Intel notation
```

I will (mostly) use the GAS syntax, where a suffix is used to indicate width of arguments:

- b (byte) = 8 bits
- w (word) = 16 bits
- l (long) = 32 bits
- q (quad) = 64 bits

For example, we have movb, movw movl, and movq.

Examples (in GAS notation)

```
movl $4, %eax       # put 32 bit integer 4 in register eax
movw $4, %eax       # put 16 bit integer 4 in lower 16 bits of eax
movb $4, %eax       # put 4 bit integer 4 in lowest 4 bits of eax
movl  %esp, %ebp  # put the contents of esp into ebp
movl  (%esp), %ebp  # interpret contents of esp as a memory
                    # address. Copy the value at that addres
                    # into register ebp
movl  %esp, (%ebp)  # interpret contents of ebp as a memory
                    # address. Copy the value in esp to
                    # that address.
movl  %esp, 4(%ebp)  # interpret contents of ebp as a memor
                     # address. Add 4 to that address. Copy
                     # the value in esp to this new address.
```

A few more examples

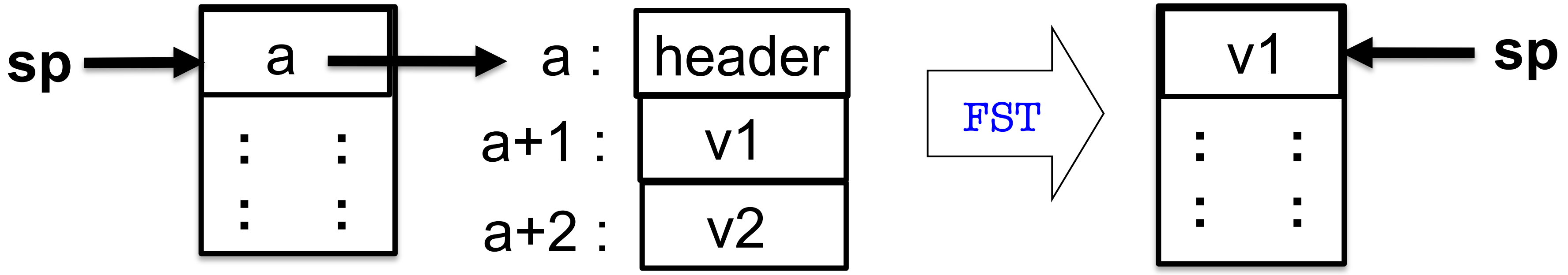
```
call label  # push return address on stack and jump to label
ret         # pop return address off stack and jump there
            # NOTE: managing other bits of the stack frame
            # such as stack and frame pointer must be done
            # explicitly
subl $4, %esp    # subtract 4 from esp. That is, adjust the
                 # stack pointer to make room for one 32-bit
                 # (4 byte) value. (stack grows downward!)
```

Assume that we have implemented a procedure in C called allocate that will manage heap memory. We will compile and link this in with code generated by the slang compiler. At the x86 level, allocate will expect a header in **edi** and return a heap pointer in **eax**.

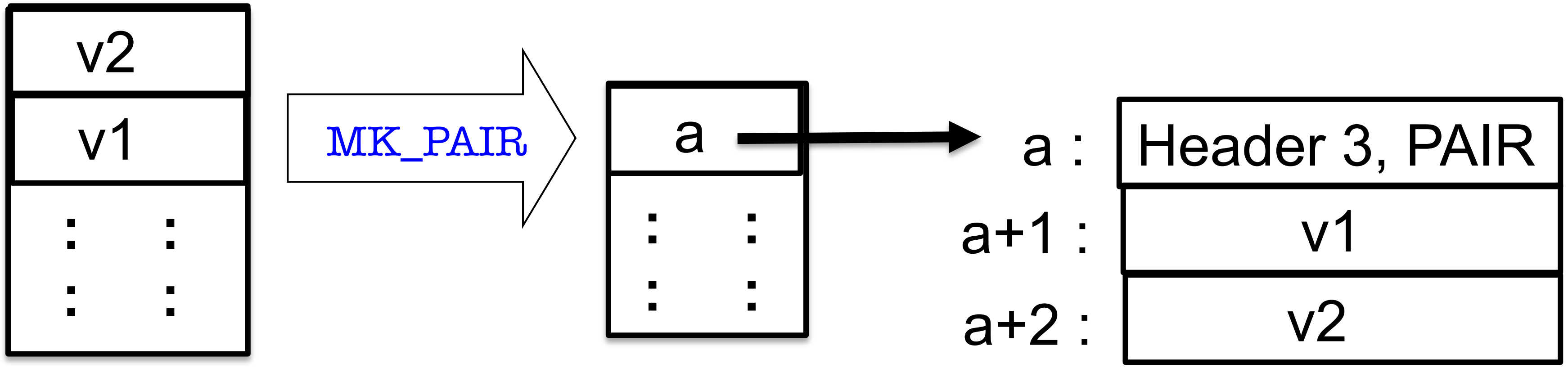
Some Jargon VM instructions are "easy" to translate

Remember: X86 is CISC, so RISC architectures may require more instructions ..

GOTO loc	jmp loc	
POP	addl $4, %esp	// move stack pointer 1 word = 4 bytes
PUSH v	subl $4, %esp movl $i, (%esp)	// make room on top of stack // where i is an integer representing v
FST	movl 4(%esp), %edx movl %edx, (%esp)	// 4 bytes, 1 word, after header // replace "a" with "v1" at top of stack
SND	movl 8(%esp), %edx movl %edx, (%esp)	// 8 bytes, 2 words, after header // replace "a" with "v2" at top of stack



… while others require more work



One possible x86 (32 bit) implementation of MK_PAIR:

```
movl $3, %edi              // construct header in edi
shr $16, %edi,             // ... put size in upper 16 bits (shift right)
movw $PAIR, %di            // ... put type in lower 16 bits of edi
call allocate              // input: header in ebi, output: “a” in eax
movl (%esp), %edx          // move “v2” to the heap,
movl %edx, 8(%eax)         //  ...  using temporary register edx
addl $4, %esp              // adjust stack pointer (pop “v2”)
movl (%esp), %edx          // move “v1” to the heap
movl %edx, 4(%eax)         //  ...  using temporary register edx
movl %eax, (%esp)          // copy value “a” to top of stack
```

Left as exercises for you :

LOOKUP APPLY RETURN CASE TEST ASSIGN REF

Here's a hint. For things you don't understand, just experiment! OK, you need to pull an address out of a closure and call it. Hmm, how does something similar get compiled from C?

```
int func ( int (*f)(int) ) { return (*f)(17); } /* pass a function pointer and apply it /*
```

X86, 64 bit

without –O2

```
_func:
        pushq   %rbp             # save frame pointer
        movq    %rsp, %rbp       # set frame pointer to stack pointer
        subq    $16, %rsp        # make some room on stack
        movl    $17, %eax        # put 17 in argument register eax
        movq    %rdi, -8(%rbp)   # rdi contains the argument f
        movl    %eax, %edi       # put 17 in register edi, so f will get it
        callq   *-8(%rbp)        # WOW, a computed address for function call!
        addq    $16, %rsp        # restore stack pointer
        popq    %rbp             # restore old frame pointer
        ret                      # restore stack
```

What about arithmetic?

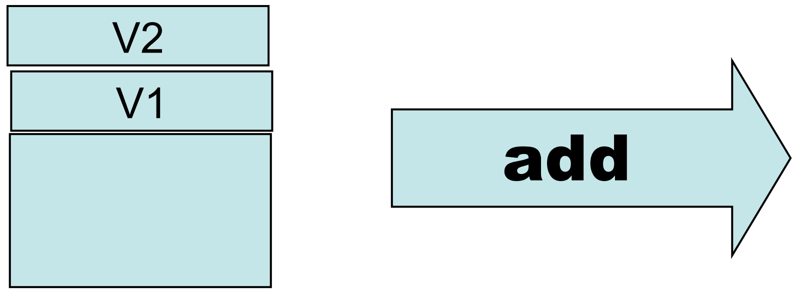
Houston, we have a problem….

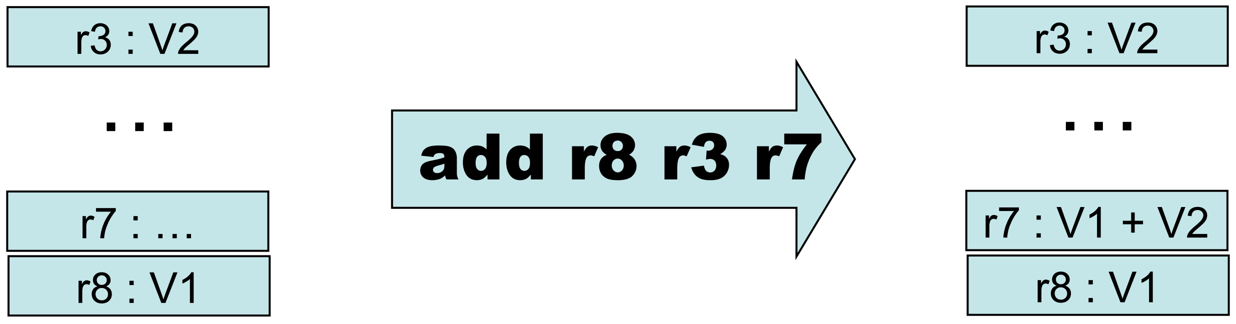
- It may not be obvious now, but if we want to have automated memory management we need to be able to distinguish between values (say integers) and pointers at runtime.
- Have you ever noticed that integers in SML or Ocaml are either 31 (or 63) bits rather than the native 32 (or 64) bits?
  - That is because these compilers use a the least significant bit to distinguish integers (bit = 1) from pointers (bit = 0).
  - OK, this works. But it may complicate every arithmetic operation!
  - This is another exercise left for you to ponder …

Lecture 14
Assorted Topics

1. Stacks are slow, registers are fast
   1. Stack frames still needed …
   2. … but try to shift work into registers
   3. Caller/callee save/restore policies
   4. Register spilling
2. Simple optimisations
   1. Peep hole (sliding window)
   2. Constant propagation
   3. Inlining
3. Representing objects (as in OOP)
   1. At first glance objects look like a closure containing multiple function (methods) …
   2. … but complications arise with method dispatch
4. Implementing exception handling on the stack

Stack vs regsisters





Stack-oriented:
(+) argument locations is implicit, so instructions are smaller.
(---) Execution is slower

Register-oriented:
(+++) Execution MUCH faster
(-) argument location is explicit, so instructions are larger

Main dilemma : registers are fast, but are fixed in number. And that number is rather small.

- Manipulating the stack involves RAM access, which can be orders of magnitude slower than register access (the "von Neumann Bottleneck")
- Fast registers are (today) a scarce resource, shared by many code fragments
- How can registers be used most effectively?
  - Requires a careful examination of a program's structure
  - Analysis phase: building data structures (typically directed graphs) that capture definition/use relationships
  - Transformation phase : using this information to rewrite code, attempting to most efficiently utilise registers
  - Problem is NP-complete
  - One of the central topics of Part II Optimising Compilers.
- Here we focus <u>only</u> on general issues : <u>calling conventions</u> and <u>register spilling</u>

Caller/callee conventions

- Caller and callee code may use overlapping sets of registers
- An agreement is needed concerning use of registers
  - Are some arguments passed in specific registers?
  - Is the result returned in a specific register?
  - If the caller and callee are both using a set of registers for “scratch space” then caller or callee must save and restore these registers so that the caller’s registers are not obliterated by the callee.
- Standard calling conventions identify specific subsets of registers as “caller saved” or “callee saved”
  - **Caller saved**: if caller cares about the value in a register, then must save it before making any call
  - **Callee saved**: The caller can be assured that the callee will leave the register intact (perhaps by saving and restoring it)

Another C example. X86, 64 bit, with gcc

```
int
callee(int, int,int,
        int,int,int,int);

int caller(void)
{
  int ret;
  ret = callee(1,2,3,4,5,6,7);
  ret += 5;
  return ret;
}
```

```
_caller:
        pushq    %rbp          # save frame pointer
        movq     %rsp, %rbp  # set new frame pointer
        subq     $16, %rsp   # make room on stack
        movl     $7, (%rsp)  # put 7th arg on stack
        movl     $1, %edi     # put 1st arg on in edi
        movl     $2, %esi     # put 2nd arg on in esi
        movl     $3, %edx     # put 3rd arg on in edx
        movl     $4, %ecx     # put 4th arg on in ecx
        movl     $5, %r8d     # put 5th arg on in r8d
        movl     $6, %r9d     # put 6th arg on in r9d
        callq    _callee       #will put resut in eax
        addl     $5, %eax     # add 5
        addq     $16, %rsp   # adjust stack
        popq     %rbp         # restore  frame pointer
        ret          # pop return address, go there
```

Regsiter spilling

- What happens when all registers are in use?
- Could use the stack for scratch space …
- … or (1) move some register values to the stack, (2) use the registers for computation, (3) restore the registers to their original value
- This is called register spilling

Simple optimisations. Inline expansion

```
fun f(x) = x + 1
fun g(x) = x - 1
…
…
fun h(x) = f(x) + g(x)
```

inline f and g

```
fun f(x) = x + 1
fun g(x) = x - 1
…
…
fun h(x) = (x+1) + (x-1)
```

(+) Avoid building activation records at runtime

(+) May allow further optimisations

(-) May lead to "code bloat" (apply only to functions with "small" bodies?)

Question: if we inline all occurrences of a function, can we delete its definition from the code?
What if it is needed at link time?

Be careful with variable scope

Inline g in h

```
let val x = 1
    fun g(y) = x + y
    fun h(x) = g(x) + 1
in
   h(17)
end
```

NO

```
let val x = 1
    fun g(y) = x + y
    fun h(x) = x + y + 1
in
   h(17)
end
```

YES

```
let val x = 1
    fun g(y) = x + y
    fun h(z) = x + z + 1
in
   h(17)
end
```

What kind of care might be needed will depend on the representation level of the Intermediate code involved.

(b) Constant propagation, constant folding

```
let x = 2
let y = x - 1
let z = y * 17
```

```
let x = 2
let y = 2 - 1
let z = y * 17
```

```
let x = 2
let y = 1
let z = y * 17
```

```
let x = 2
let y = 1
let z = 1 * 17
```

```
let x = 2
let y = 1
let z = 17
```

Propagate constants and evaluate simple expressions at compile-time

Note : opportunities are often exposed by inline expansion!

David Gries :
"Never put off till run-time what you can do at compile-time."

But be careful

How about this?

Replace

x * 0

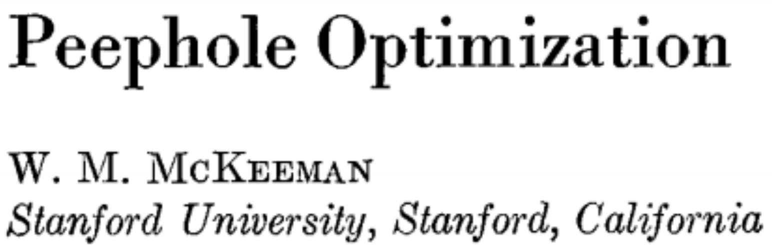
with

0

OOPS, not if x has type float!

NAN*0 = NAN,

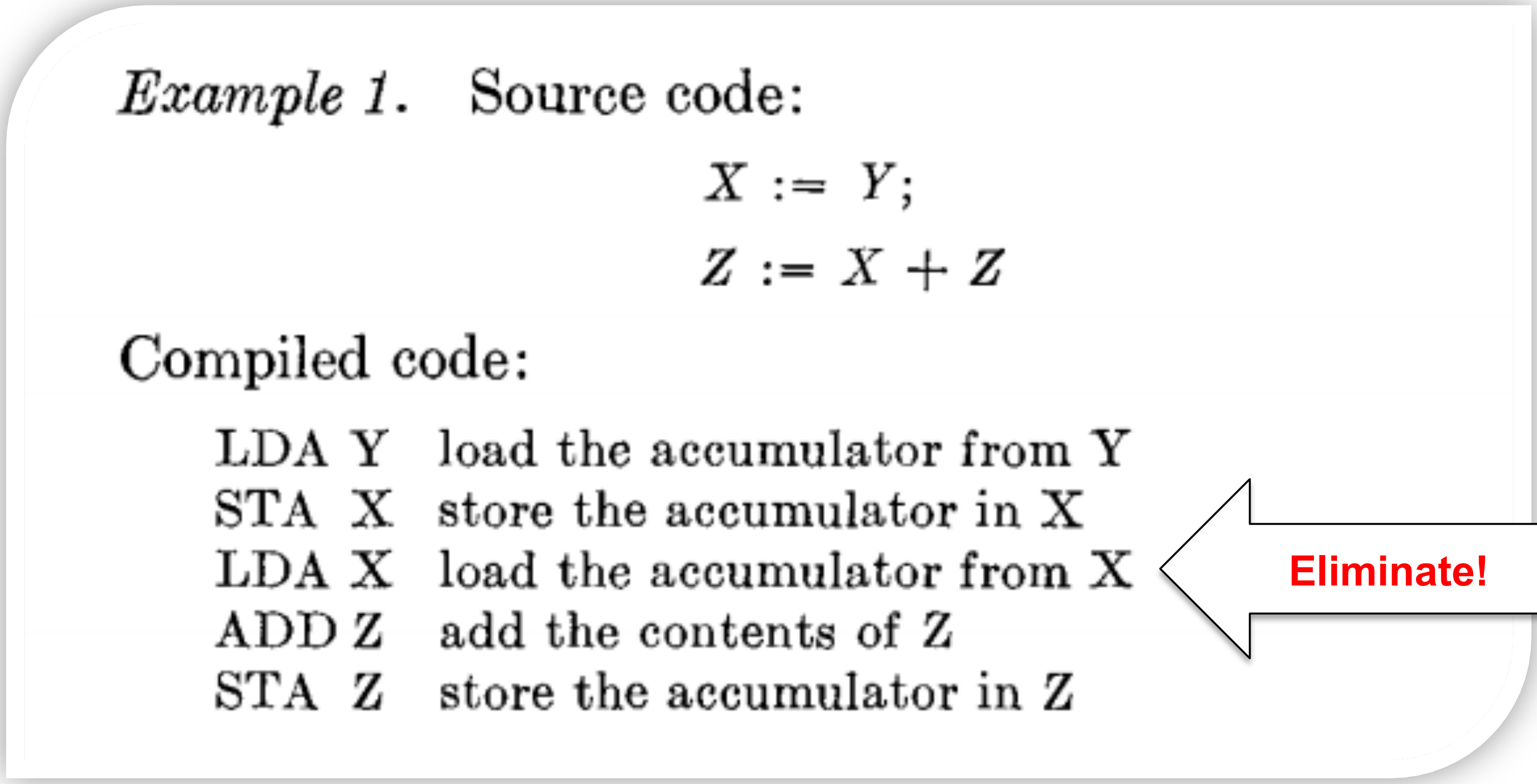
(c) peephole optimisation



Peephole Optimization

W. M. McKeeman
Stanford University, Stanford, California

Communications of the ACM, July 1965



Example 1. Source code:

$$X := Y;$$
$$Z := X + Z$$

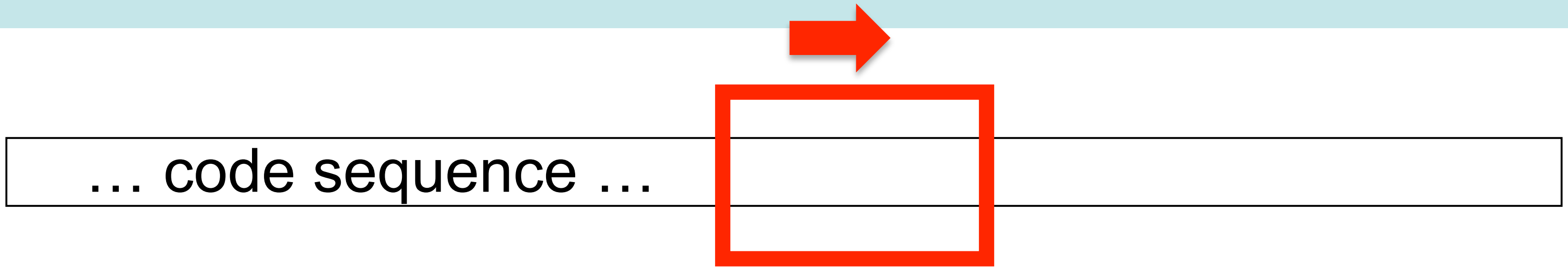
Compiled code:

```
LDA Y   load the accumulator from Y
STA  X   store the accumulator in X
LDA X   load the accumulator from X
ADD Z   add the contents of Z
STA  Z   store the accumulator in Z
```

Eliminate!

Results for syntax-directed code generation.

peephole optimisation



Sweep a window over the code sequence looking for instances of simple code patterns that can be rewritten to better code … (might be combined with constant folding, etc, and employ multiple passes)

Examples
-- eliminate useless combinations (push 0; pop)
-- introduce machine-specific instructions
-- improve control flow. For example: rewrite
"GOTO L1 … L1: GOTO L2"
to
"GOTO L2 … L1 : GOTO L2")

gcc example.
-O<m> turns on optimisation to level m

g.c

```
int h(int n)  { return (0 < n) ? n : 101 ; }

int g(int n)  { return 12 * h(n + 17); }
```

```
gcc –O2 –S –c g.c
```

g.s (fragment)

```
_g:
          .cfi_startproc
          pushq   %rbp
          movq    %rsp, %rbp
          addl    $17, %edi
          imull   $12, %edi, %ecx
          testl   %edi, %edi
          movl    $1212, %eax
          cmovgl  %ecx, %eax
          popq    %rbp
          ret
          .cfi_endproc
```

Wait. What happened to the call to h???

GNU AS (GAS) Syntax
x86, 64 bit

gcc example (-O<m> turns on optimisation)

g.c

```
int h(int n)  { return (0 < n) ? n : 101 ; }

int g(int n)  { return 12 * h(n + 17); }
```

The compiler must have done something similar to this:

```
int g(int n)  { return 12 * h(n + 17); }
➔
 int g(int n)  { int t := n+ 17; return 12 * h(t); }
➔
int g(int n)  { int t := n+ 17; return 12 *((0 < t) ? t : 101 ); }
➔
int g(int n)  { int t := n+ 17; return (0 < t) ? 12 * t : 1212 ; }
➔ …
```

New Topic:
OOP Objects (single inheritance)

```
let start := 10

   class Vehicle extends Object {
      var position := start
      method move(int x) = {position := position + x}
   }
   class Car extends Vehicle {
      var passengers := 0
      method await(v : Vehicle) =
         if (v.position < position)
         then v.move(position – v.position)
         else self.move(10)
   }
   class Truck extends Vehicle {
      method move(int x) =
         if x <= 55 then position := position +x
   }
   var t := new Truck
   var c := new Car
   var v : Vehicle := c
in
   c.passengers := 2;
   c.move(60);
   v.move(70);
   c.await(t)
end
```

method override

subtyping allows a
Truck or Car to be viewed and
used as a Vehicle

Object Implementation?

- how do we access object fields?
  - both inherited fields and fields for the current object?
- how do we access method code?
  - if the current class does not define a particular method, where do we go to get the inherited method code?
  - how do we handle method override?
- How do we implement subtyping ("object polymorphism")?
  - If B is derived from A, then need to be able to treat a pointer to a B-object as if it were an A-object.

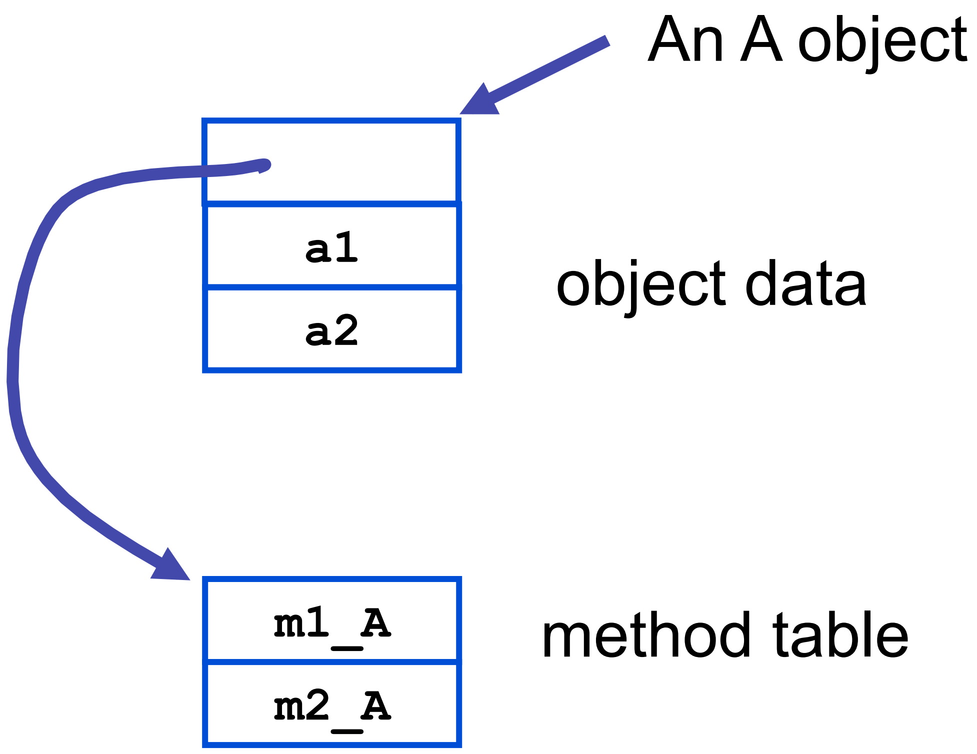
Another OO Feature

- Protection mechanisms
  - to encapsulate local state within an object, Java has “private” “protected” and “public” qualifiers
    - private methods/fields can’t be called/used outside of the class in which they are defined
  - This is really a scope/visibility issue! Front-end during semantic analysis (type checking and so on), the compiler maintains this information in the symbol table for each class and enforces visibility rules.

Object representation

C++

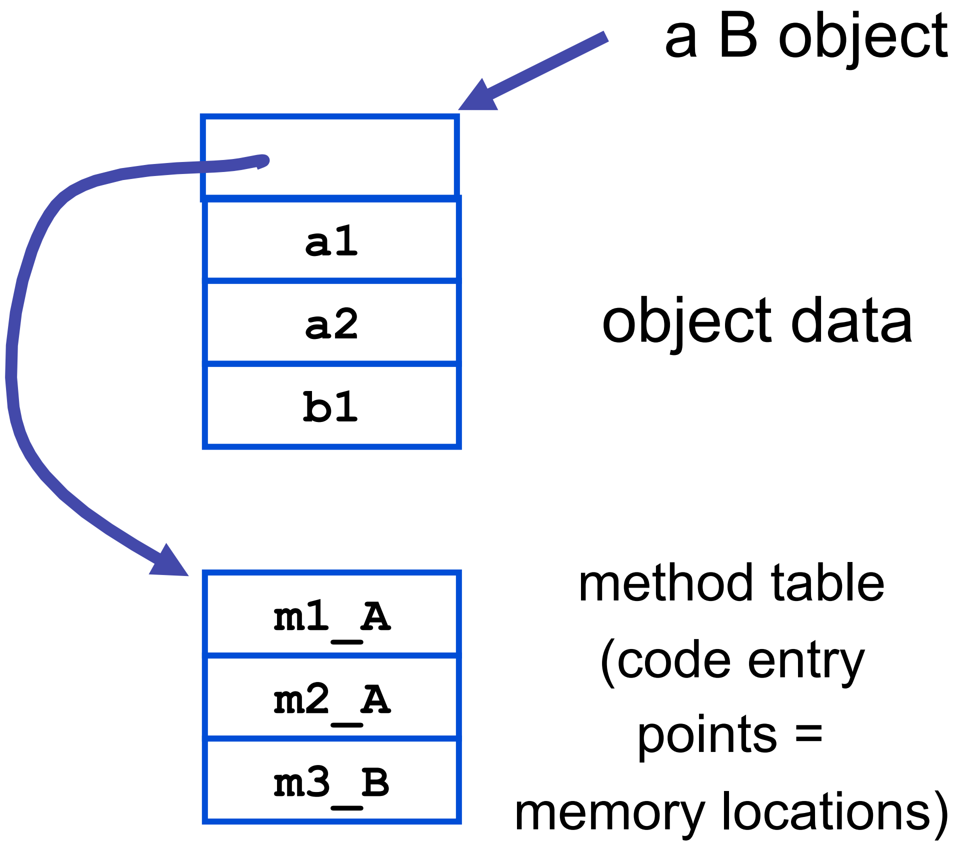
```
class A {
public:
    int a1, a2;

    void m1(int i) {
        a1 = i;
    }
    void m2(int i) {
        a2 = a1 + i;
    }
}
```



NB: a compiler typically generates methods with an extra argument representing the object (self) and used to access object data.

Inheritance ("pointer polymorphism")

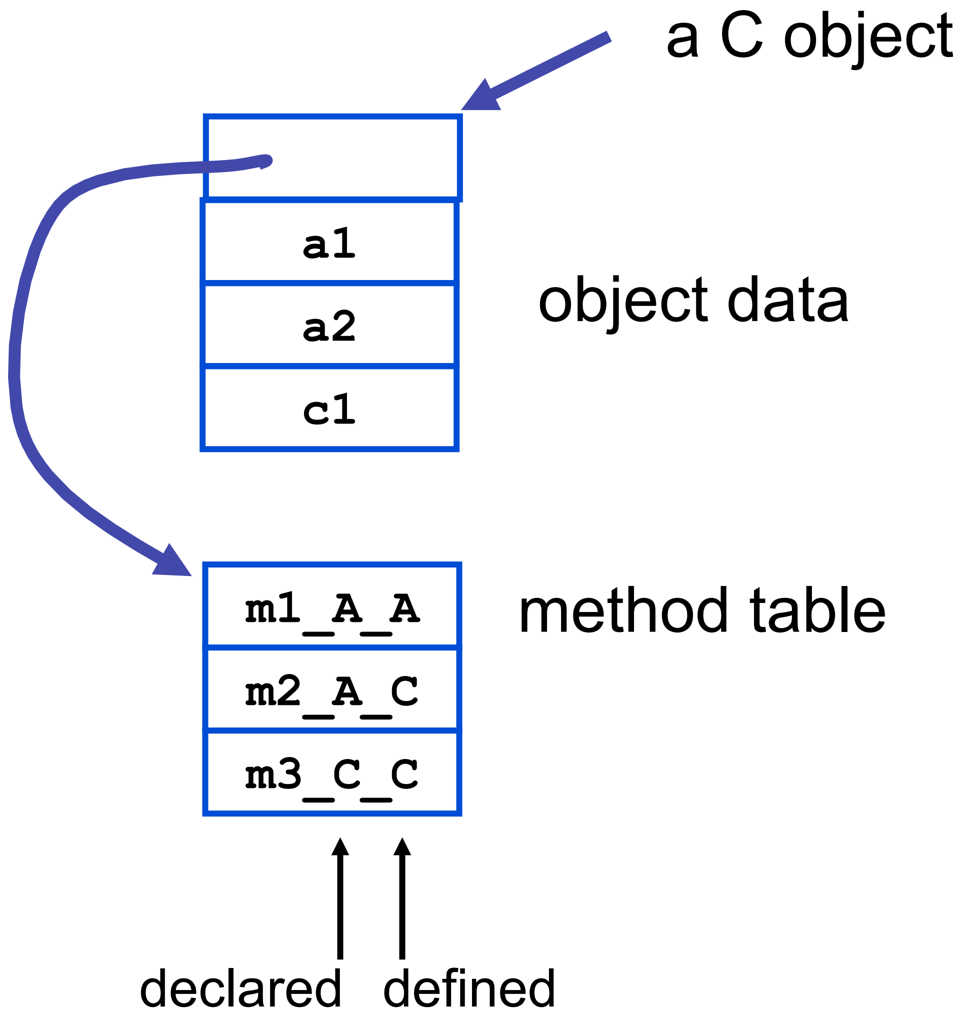
```
class B : public A {
public:
   int b1;

   void m3(void) {
      b1 = a1 + a2;
   }
}
```



Note that a pointer to a B object can be treated as if it were a pointer to an A object!

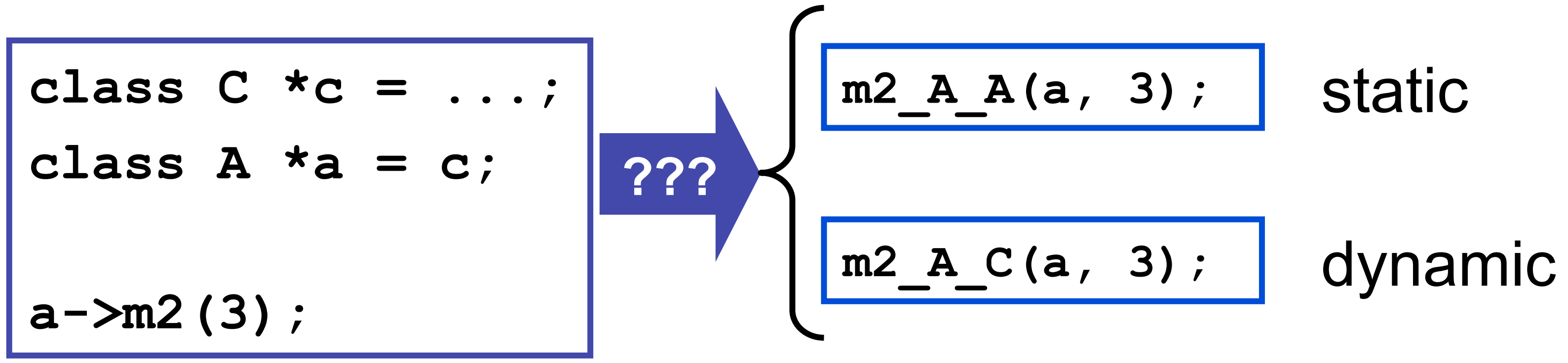
Method overriding

```
class C : public A {
public:
   int c1;

   void m3(void) {
      b1 = a1 + a2;
   }
   void m2(int i) {
      a2 = c1 + i;
   }
}
```



Static vs. Dynamic

- which method to invoke on overloaded polymorphic types?



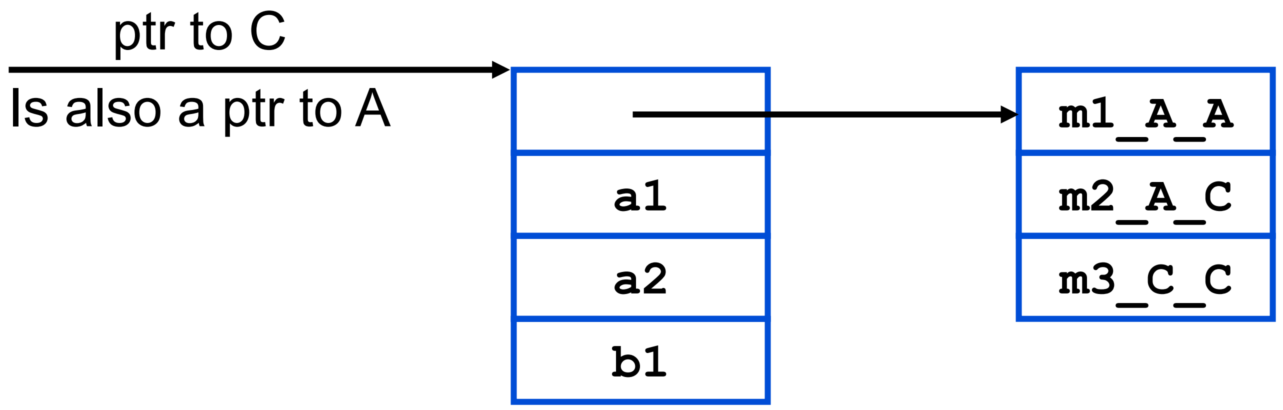
```
class C *c = ...;
class A *a = c;

a->m2(3);
```

```
m2_A_A(a, 3);
```

```
m2_A_C(a, 3);
```

Dynamic dispatch

- implementation: dispatch tables



```
class C *c = ...;
class A *a = c;

a->m2(3);
```

⇒

```
*(a->dispatch_table[1])(a, 3);
```

This implicitly uses some form of pointer subtyping

```
void m2(int i) {
        a2 = c1 + i;
}
```

```
void m2_A_C(class_A *this_A, int i) {
    class_C *this = convert_ptrA_to_ptrC(this_A);

    this->a2 = this->c1 + i;
}
```

Topic 1 : Exceptions (informal description)

```
e handle f
```

If expression e evaluates "normally" to value v, then v is the result of the entire expression.

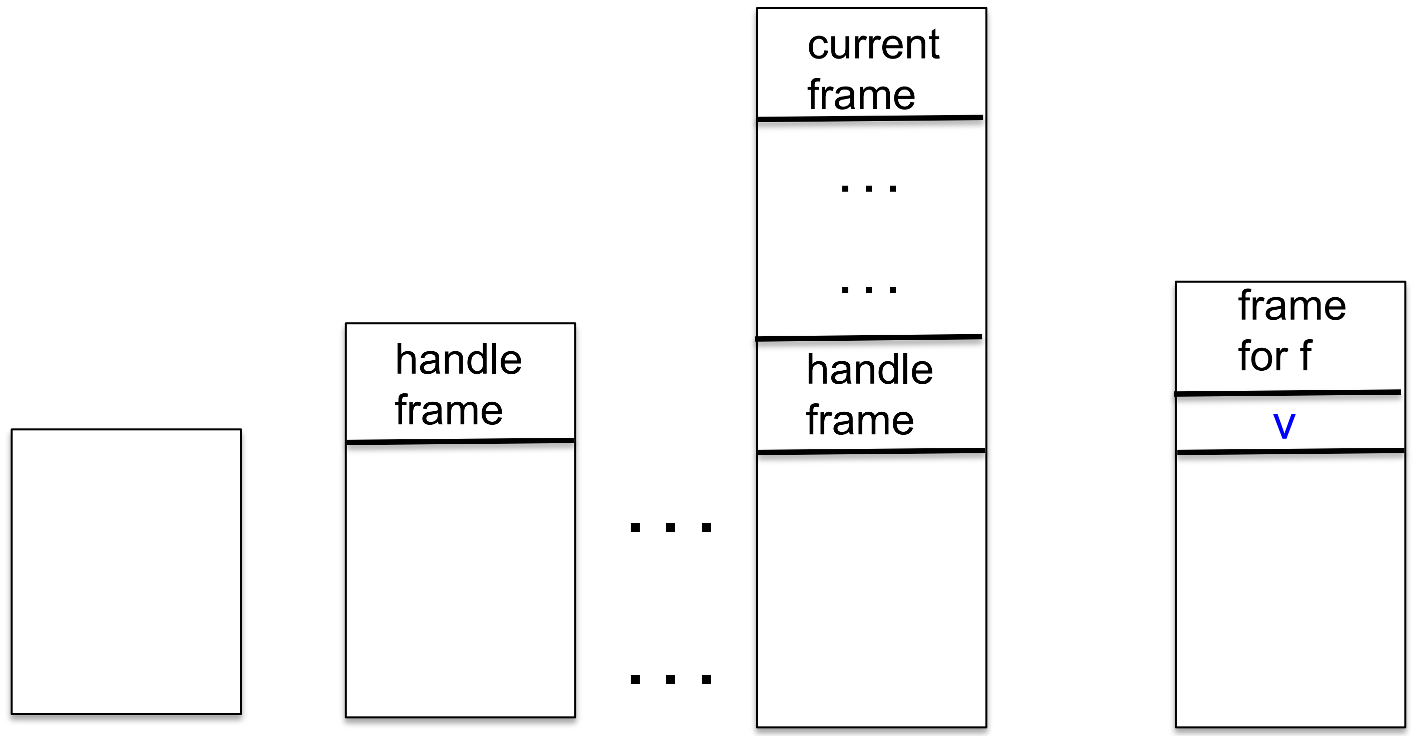
Otherwise, an exceptional value v' is "raised" in the evaluation of e, then result is (f v')

```
raise e
```

Evaluate expression e to value v, and then raise v as an exceptional value, which can only be "handled".

Implementation of exceptions may require a lot of language-specific consideration and care. Exceptions can interact in powerful and unexpected ways with other language features. Think of C++ and class destructors, for example.

Viewed from the call stack



Call stack just before evaluating code for

`e handle f`

Push a special frame for the handle

"`raise v`" is encountered while evaluating a function body associated with top-most frame

"Unwind" call stack. Depending on language, this may involve some "clean up" to free resources.

Possible pseudo-code implementation

```
e handle f
```

```
let fun _h27 () =
  build special “handle frame”
  save address of f in frame;
  … code for e …
  return value of e
in _h27 () end
```

```
raise e
```

```
… code for e …
save v, the value of e;
unwind stack until first
fp found pointing at a handle frame;
Replace handle frame with frame
for call to (extracted) f using
v as argument.
```

Lecture 15
Automating run-time memory management

- Managing the heap
- Garbage collection
  - Reference counting
  - Mark and sweep
  - Copy collection
  - Generational collection

Read Chapter 12 of
Basics of Compiler Design
(T. Mogensen)

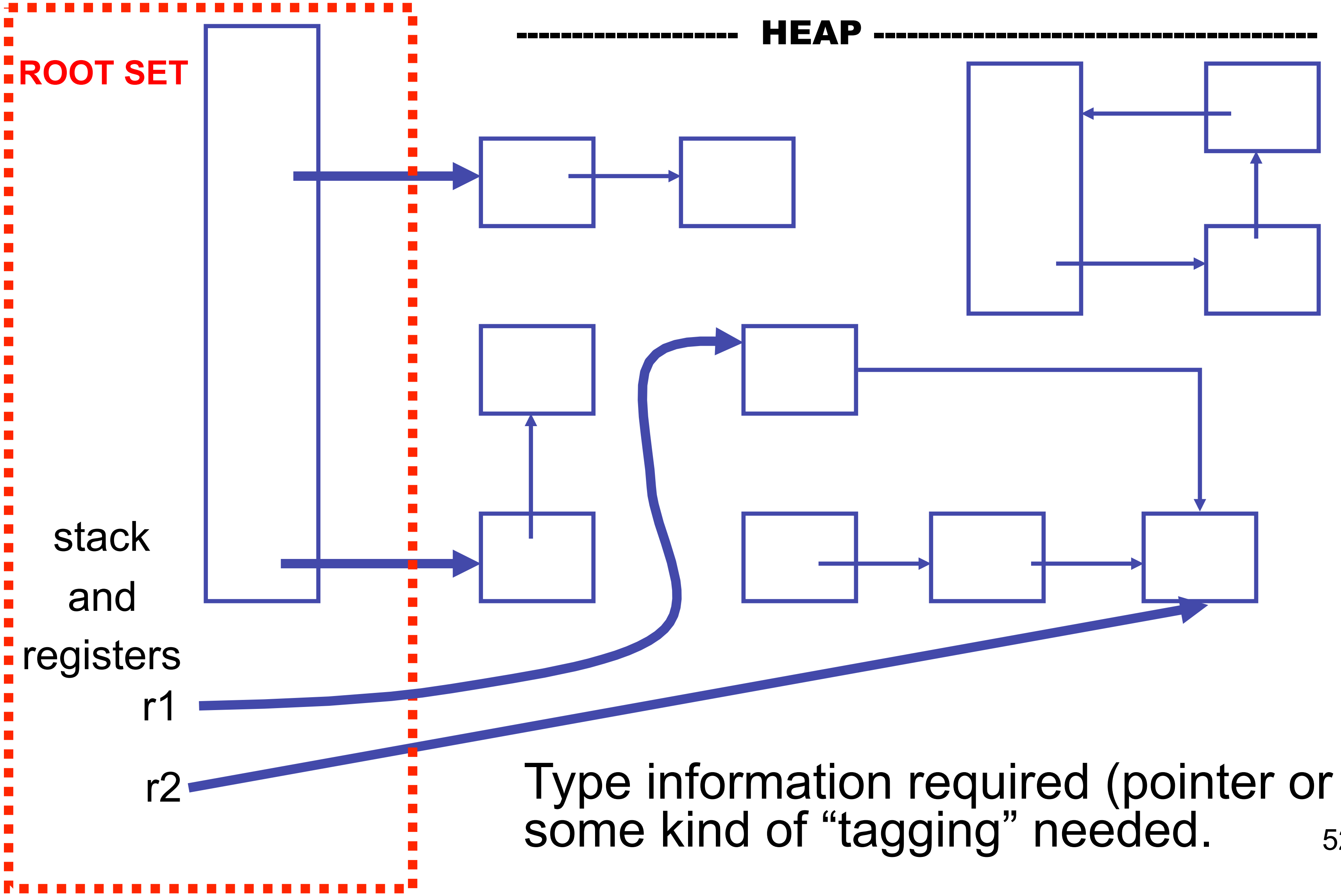
Explicit (manual) memory management

- User library manages memory; programmer decides when and where to allocate and de-allocate
  - void* malloc(long n)
  - void free(void *addr)
  - Library calls OS for more pages when necessary
  - Advantage: Gives programmer a lot of control.
  - Disadvantage: people too clever and make mistakes. Getting it right can be costly. And don't we want to automate-away tedium?
  - Advantage: With these procedures we can implement memory management for "higher level" languages ;-)

Memory Management

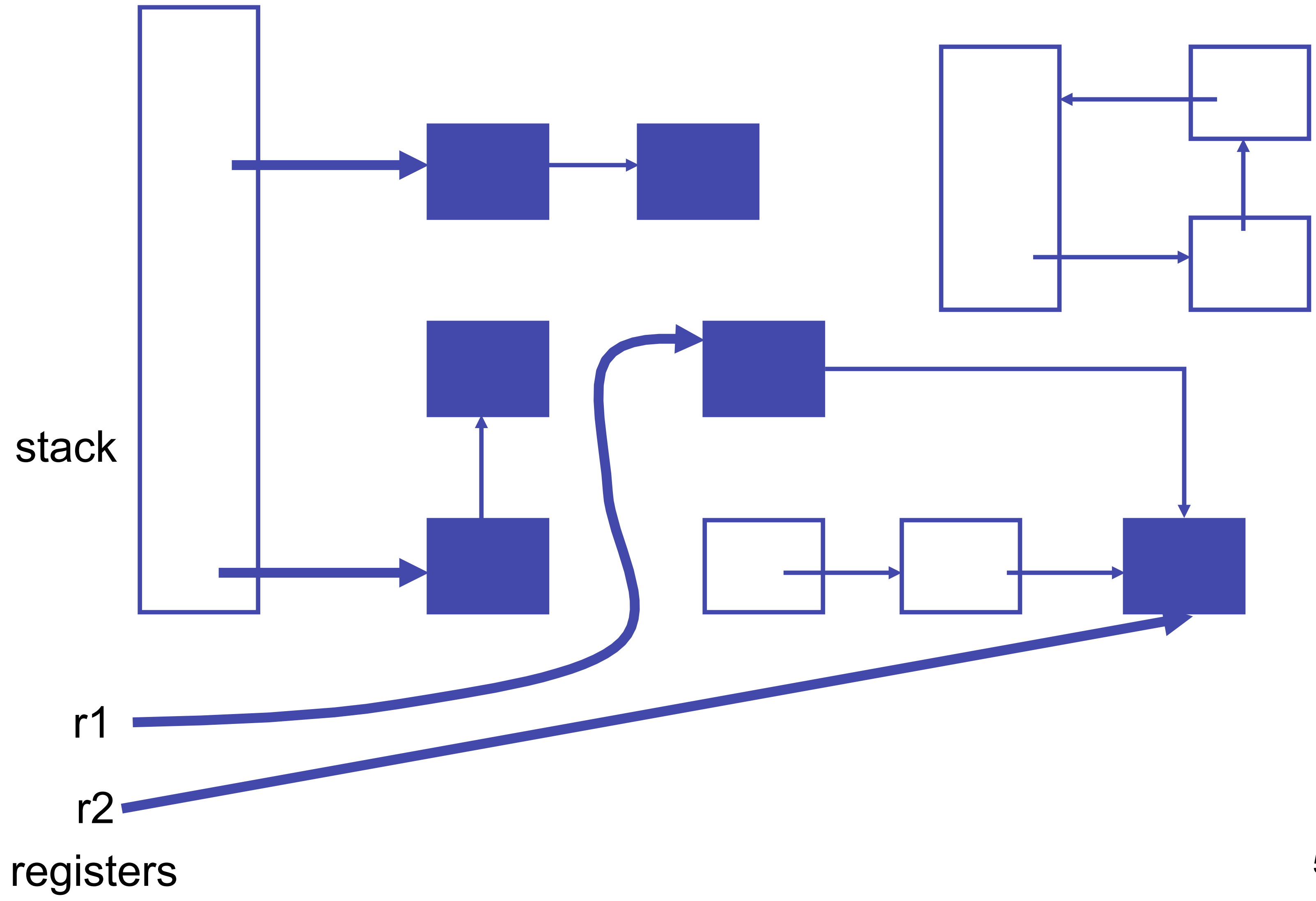
- Many programming languages allow programmers to (implicitly) allocate new storage dynamically, with no need to worry about reclaiming space no longer used.
  - New records, arrays, tuples, objects, closures, etc.
  - Java, SML, OCaml, Python, JavaScript, Python, Ruby, Go, Swift, SmallTalk, …
- Memory could easily be exhausted without some method of reclaiming and recycling the storage that will no longer be used.
  - Often called “garbage collection”
  - Is really “automated memory management” since it deals with allocation, de-allocation, compaction, and memory-related interactions with the OS.

Automation is based on an approximation : if data can be reached from a root set, then it is not “garbage”

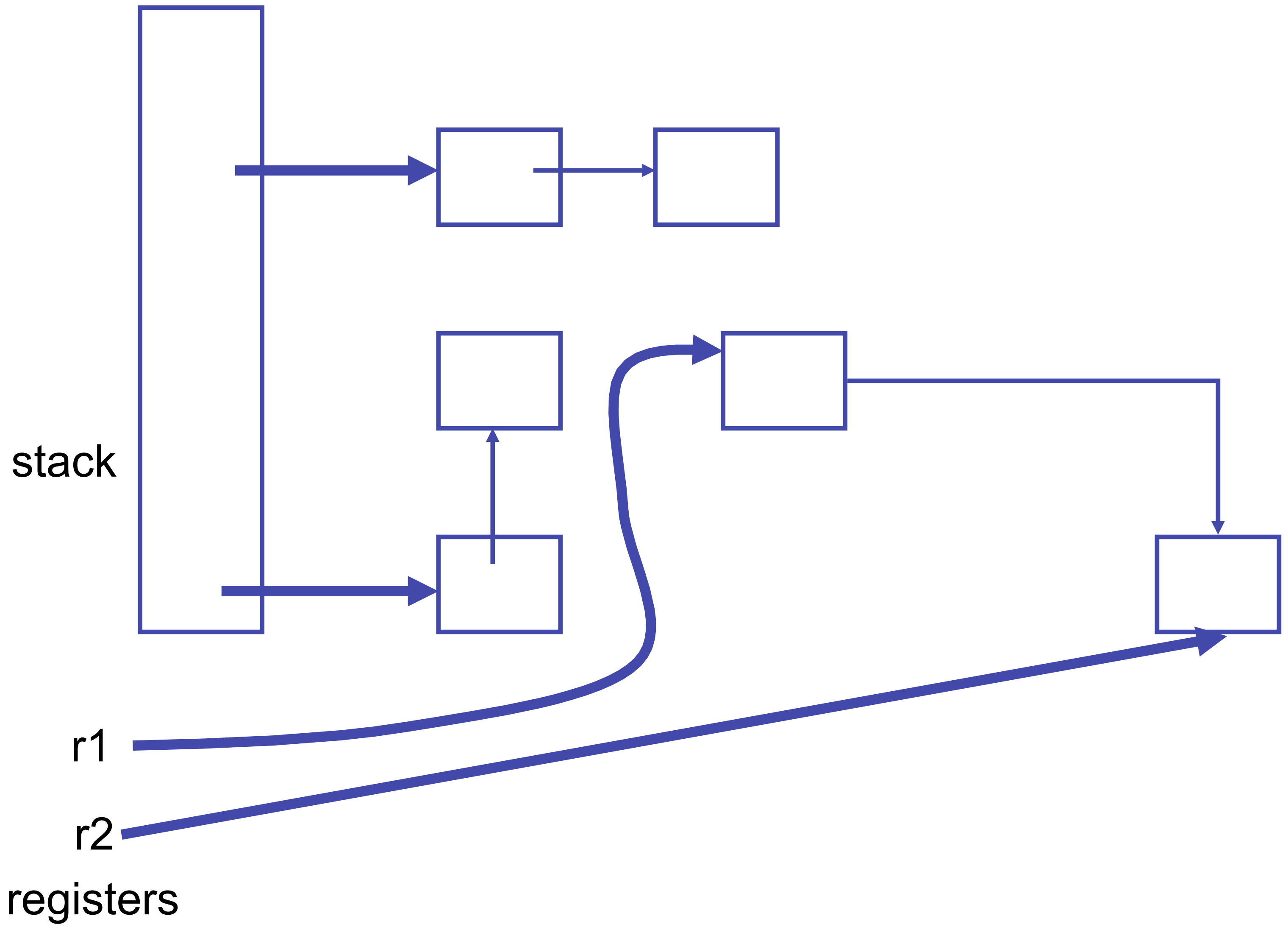

ROOT SET
-------------------- HEAP --
stack
and
registers
r1
r2

Type information required (pointer or not), some kind of “tagging” needed.

... Identify Cells Reachable From Root Set...



… reclaim unreachable cells



But How? Two basic techniques, and many variations

- **Reference counting** : Keep a reference count with each object that represents the number of pointers to it. Is garbage when count is 0.
- **Tracing** : find all objects reachable from root set. Basically transitive close of pointer graph.

For a very interesting (non-examinable) treatment of this subject see

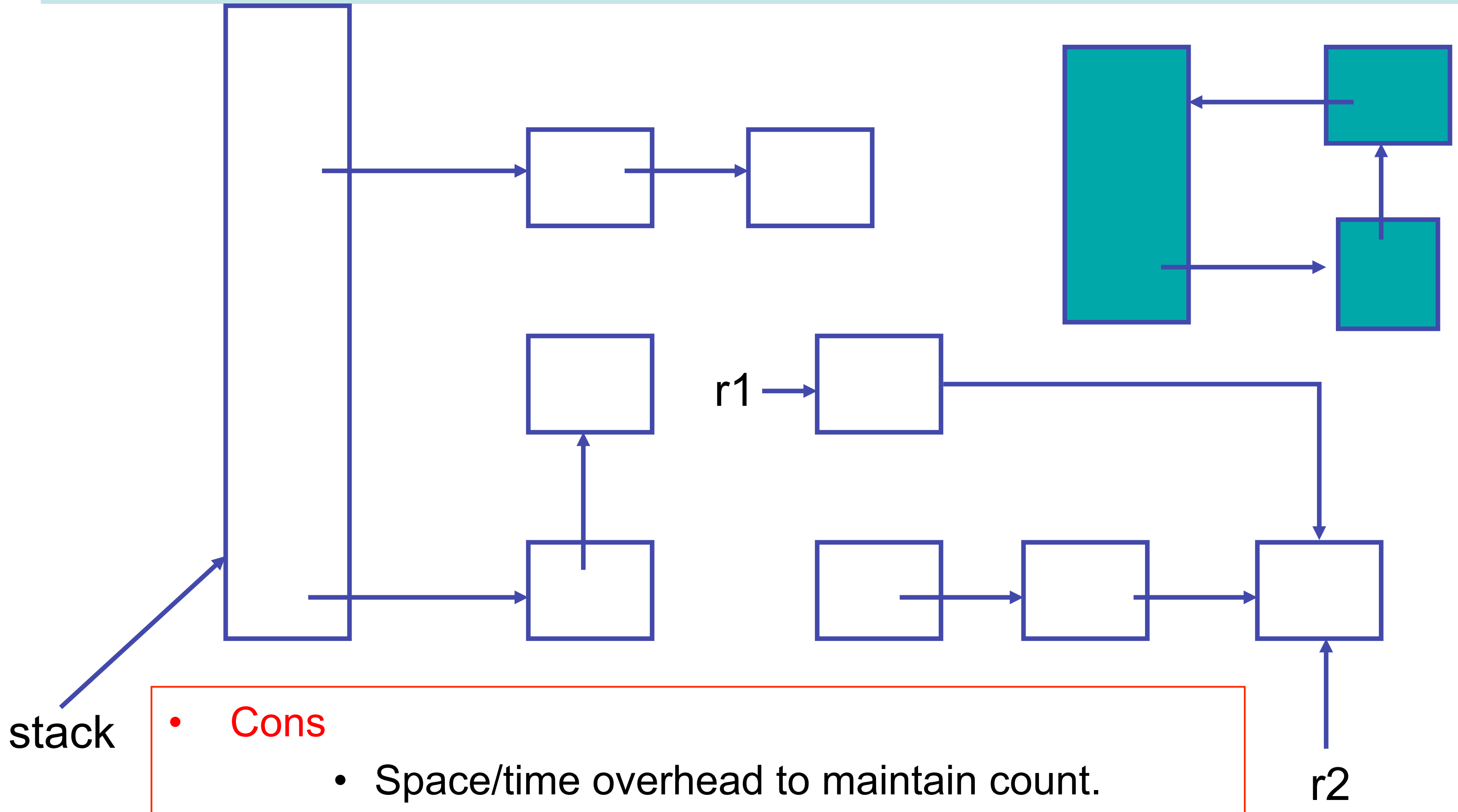
A Unified Theory of Garbage Collection.
David F. Bacon, Perry Cheng, V.T. Rajan.
OOPSLA 2004.

In that paper reference counting and tracing are presented as "dual" approaches, and other techniques are hybrids of the two.

Reference Counting, basic idea:

- Keep track of the number of pointers to each object (the reference count).
- When Object is created, set count to 1.
- Every time a new pointer to the object is created, increment the count.
- Every time an existing pointer to an object is destroyed, decrement the count
- When the reference count goes to 0, the object is unreachable garbage

Reference counting can’t detect cycles!



- Cons
  - Space/time overhead to maintain count.
  - Memory leakage when have cycles in data.
- Pros
  - Incremental (no long pauses to collect…)

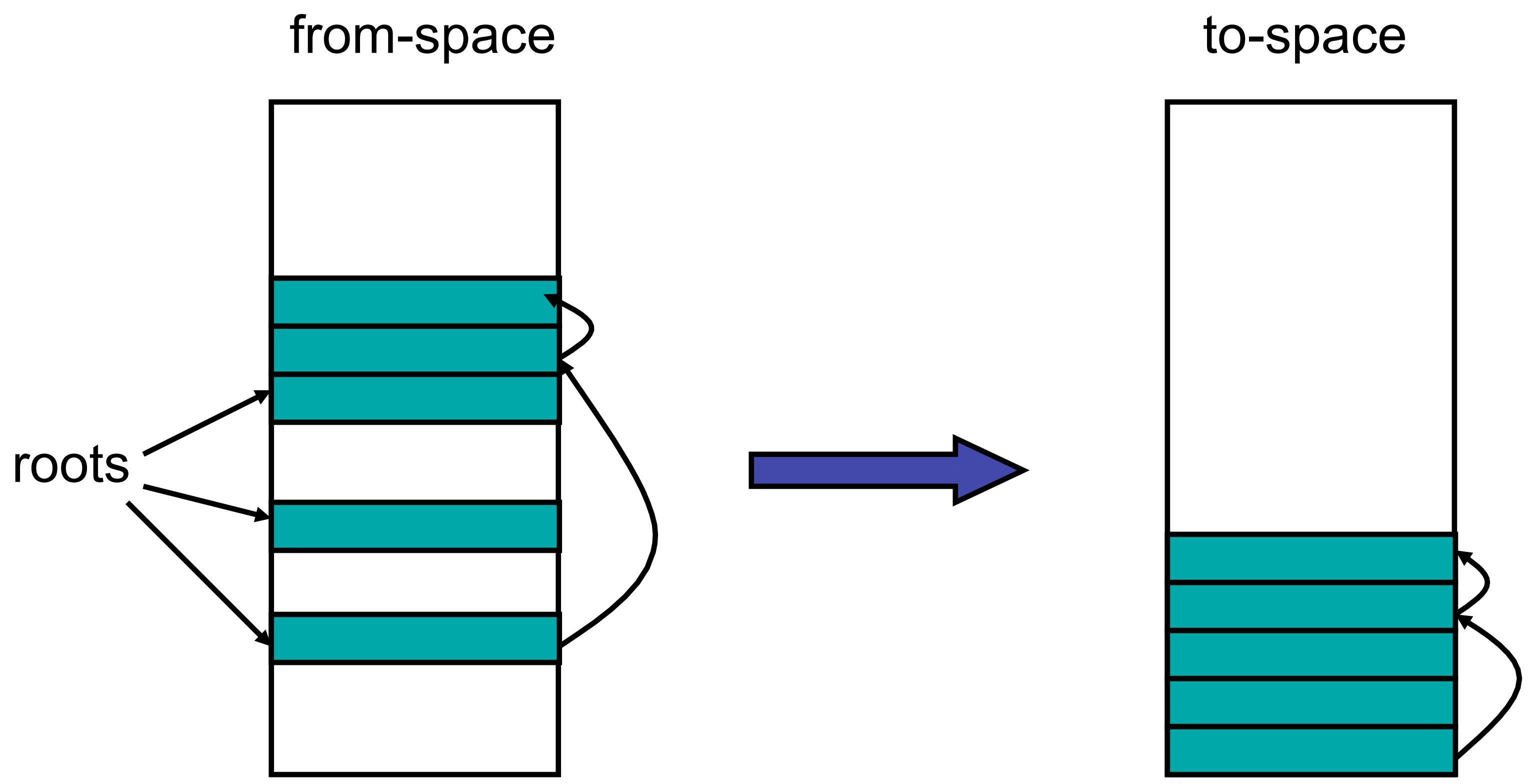
Mark and Sweep

- A two-phase algorithm
  - Mark phase: Depth first traversal of object graph from the roots to mark live data
  - Sweep phase: iterate over entire heap, adding the unmarked data back onto the free list

Copying Collection

- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- GC:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

Copying Collection



Copying GC

- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation
- Cons
  - Twice as much memory used as program requires
    - Usually, we anticipate live data will only be a small fragment of store
    - Allocate until 70% full
    - From-space = 70% heap; to-space = 30%
  - Long GC pauses = bad for interactive, real-time apps

OBSERVATION: for a copying garbage collector

- 80% to 98% new objects die very quickly.
- An object that has survived several collections has a bigger chance to become a long-lived one.
- It’s a inefficient that long-lived objects be copied over and over.

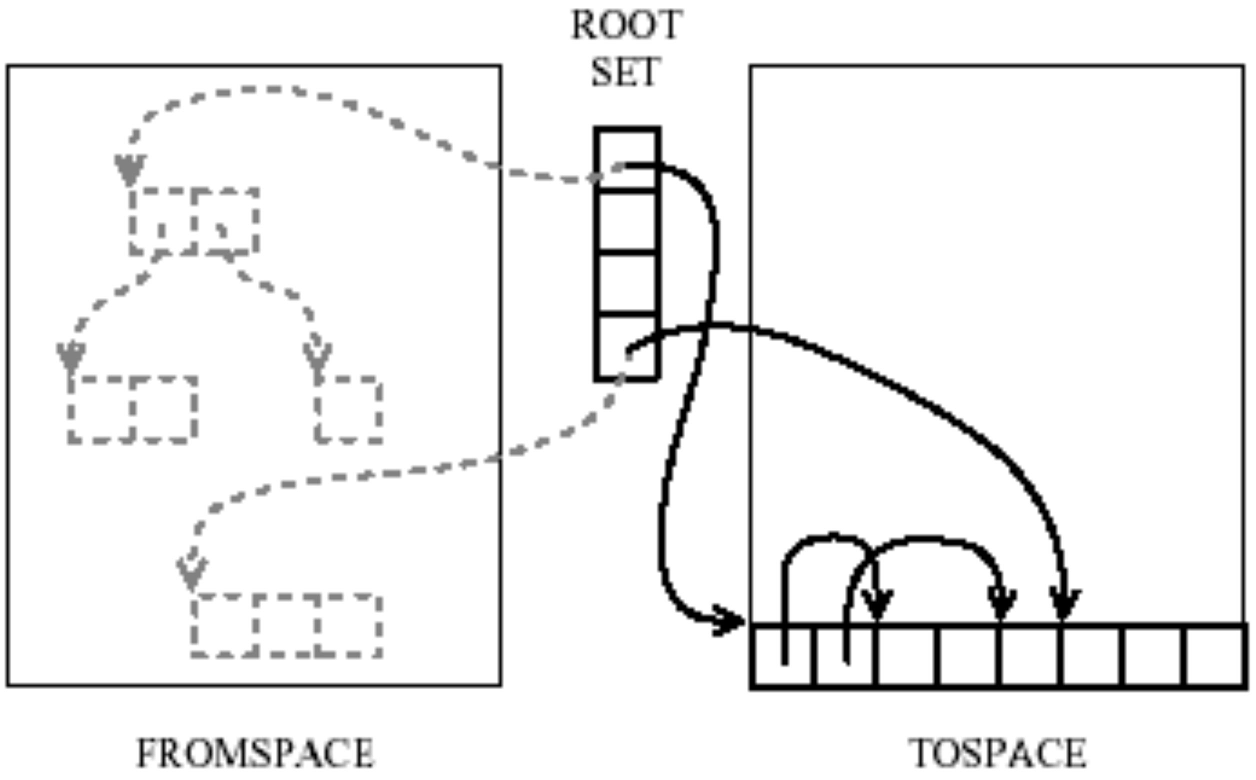


Diagram from Andrew Appel’s **Modern Compiler Implementation**

IDEA: Generational garbage collection

Segregate objects into multiple areas by age, and collect areas containing older objects less often than the younger ones.

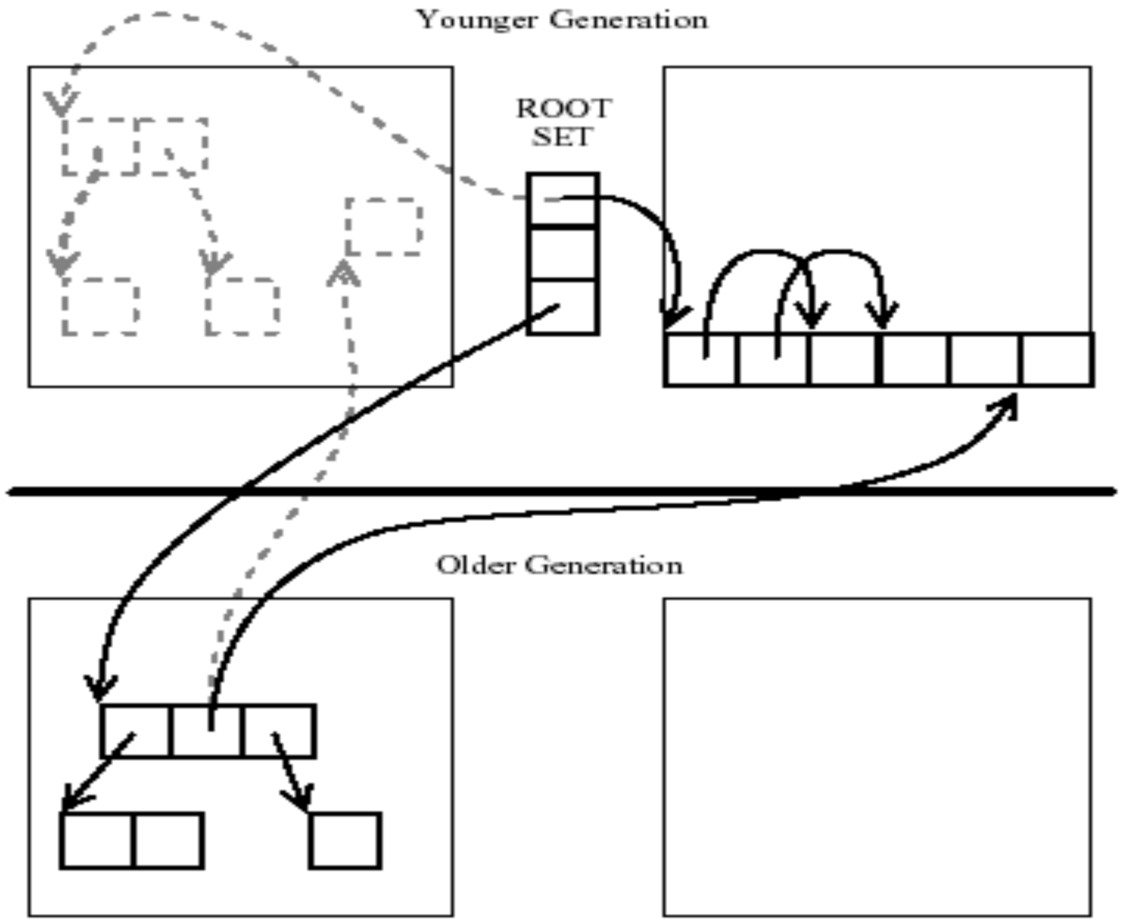


Diagram from Andrew Appel's **Modern Compiler Implementation**

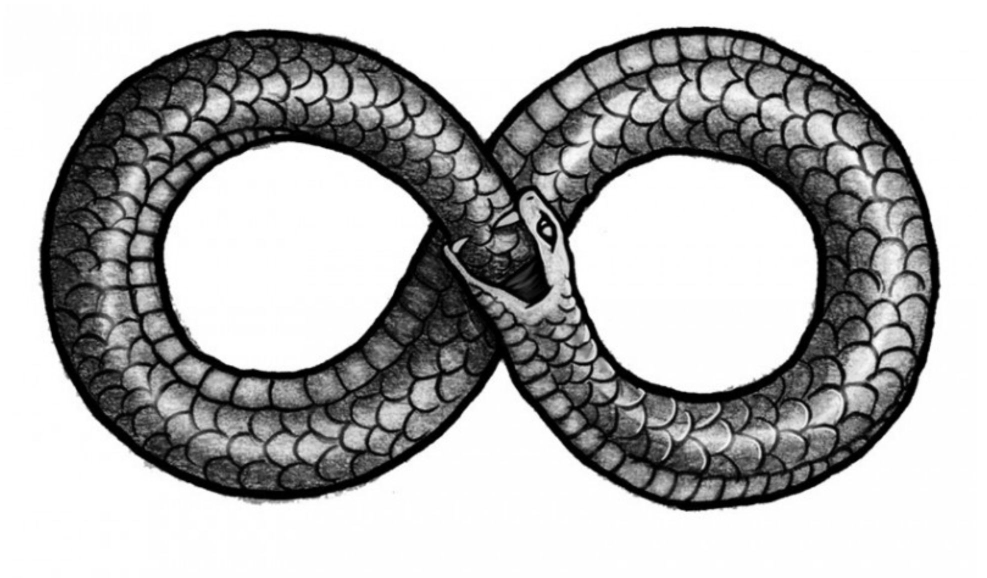
Other issues...

- When do we promote objects from young generation to old generation
  - Usually after an object survives a collection, it will be promoted
- Need to keep track of older objects pointing to newer ones!
- How big should the generations be?
  - When do we collect the old generation?
  - After several minor collections, we do a major collection
- Sometimes different GC algorithms are used for the new and older generations.
  - Why? Because the have different characteristics
  - Copying collection for the new
    - Less than 10% of the new data is usually live
    - Copying collection cost is proportional to the live data
  - Mark-sweep for the old

LECTURE 16
Bootstrapping a compiler

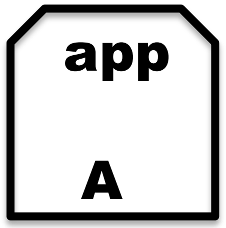
- Compilers compiling themselves!
- Read Chapter 13 Of
  - Basics of Compiler Design
  - by Torben Mogensen

    http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/

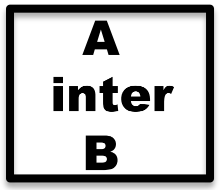


http://mythologian.net/ouroboros-symbol-of-infinity/

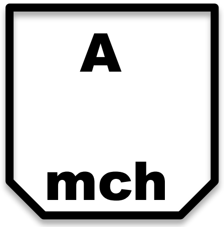
Bootstrapping. We need some notation . . .



An application called **app** written in language **A**

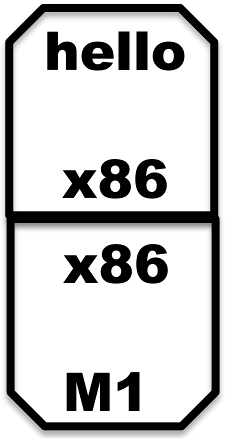


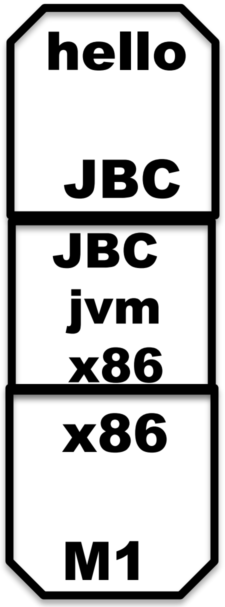
An interpreter or VM for language **A** Written in language **B**



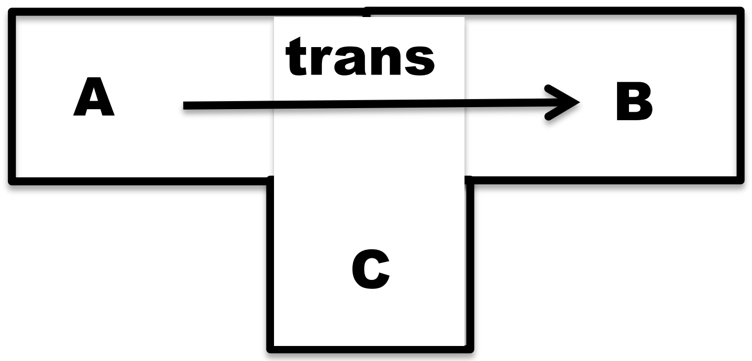
A machine called **mch** running language **A** natively.

Simple Examples



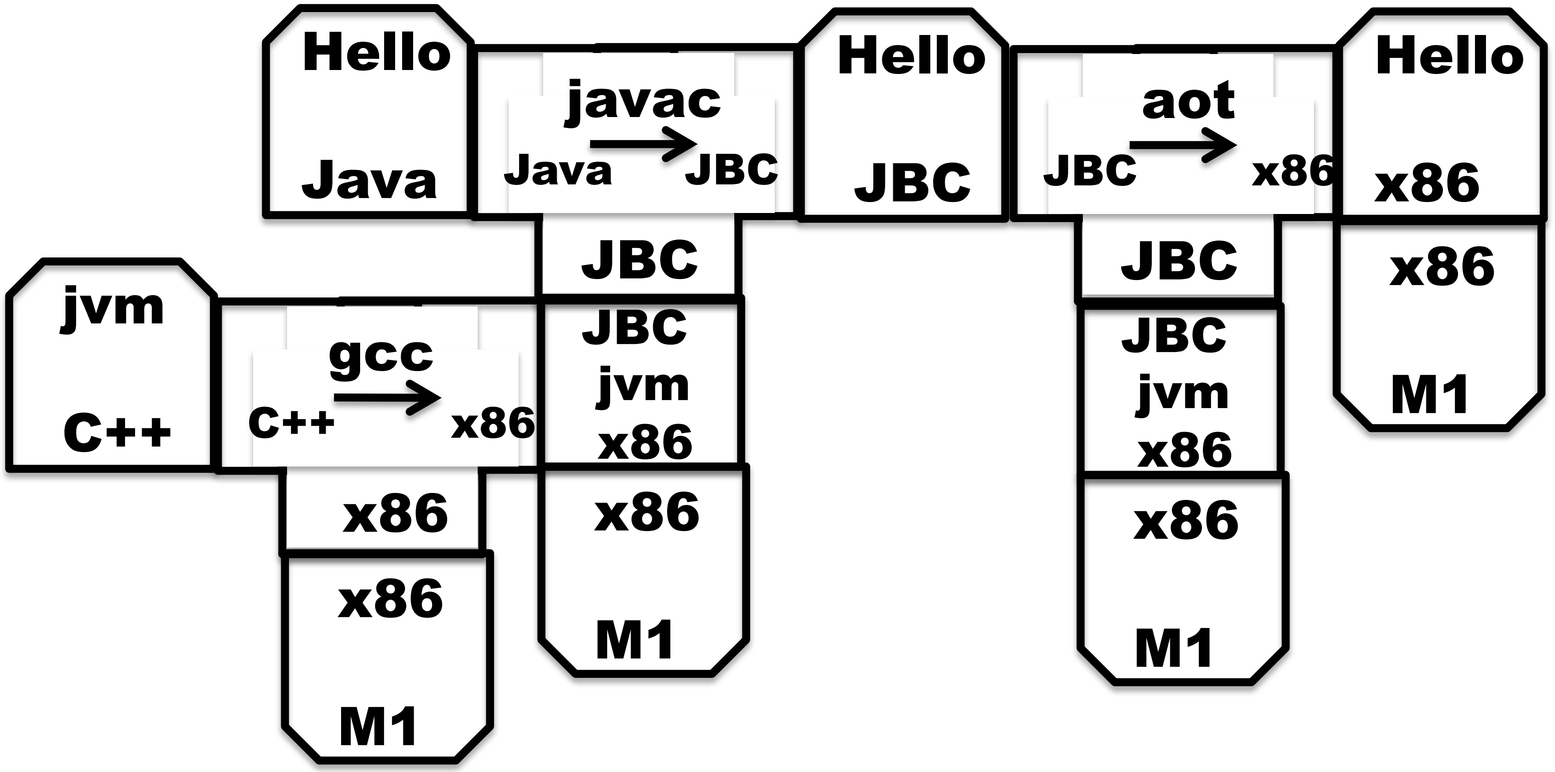


Tombstones



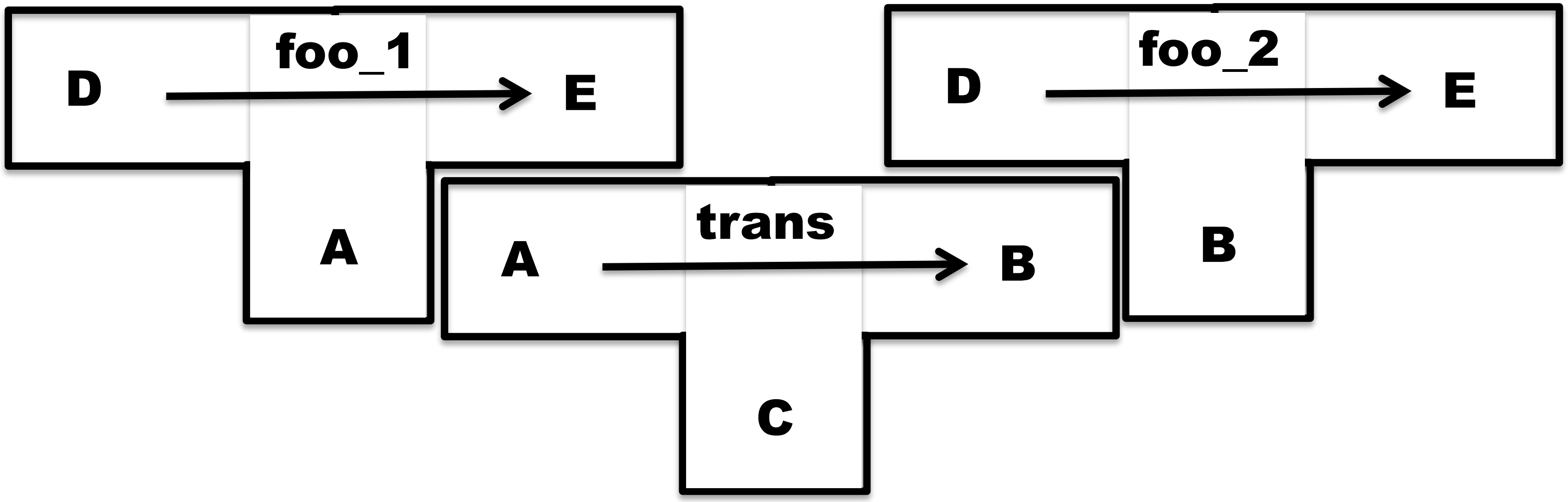
This is an application called **trans** that translates programs in language **A** into programs in language **B**, and it is written in language **C**.

Ahead-of-time compilation



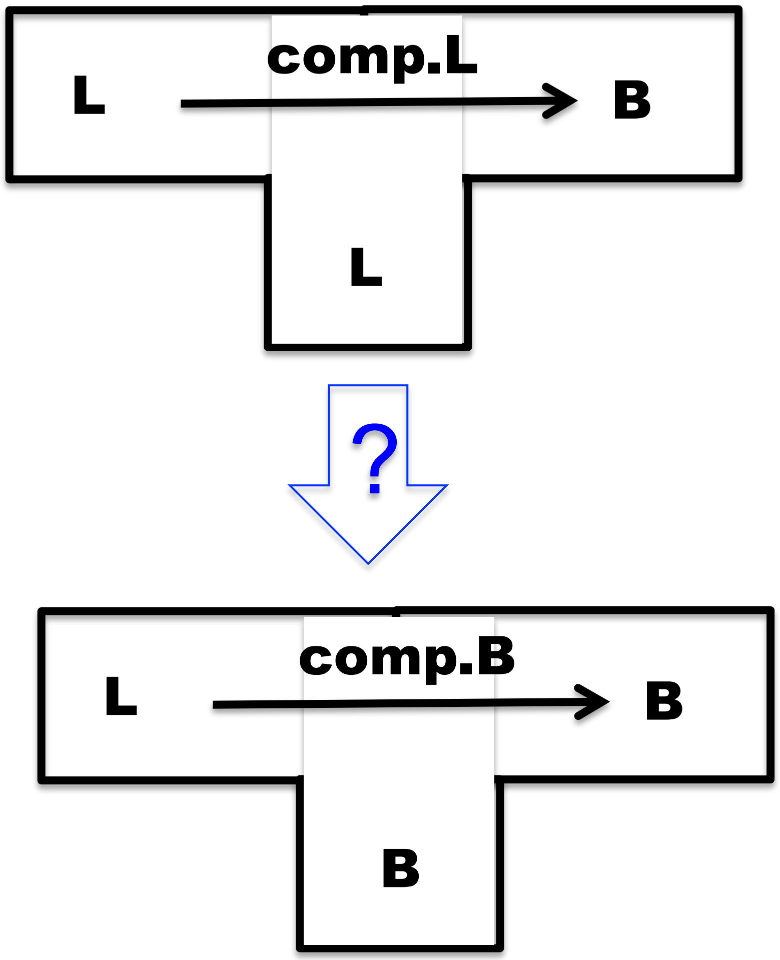
Thanks to David Greaves
for the example.

Of course translators can be translated



Translator **foo_2** is produced as output from **trans** when given **foo_1** as input.

Our seemingly impossible task



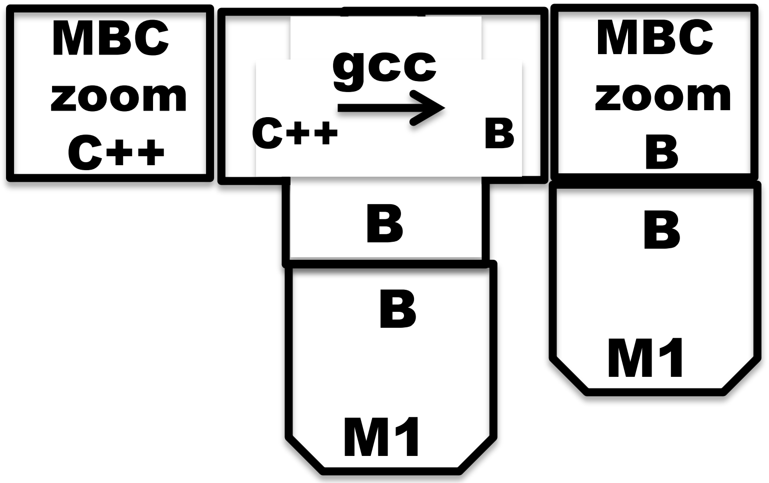
We have just invented a really great new language **L** (in fact we claim that "**L** is far superior to C++"). To prove how great **L** is we write a compiler for **L** in **L** (of course!). This compiler produces machine code **B** for a widely used instruction set (say **B** = x86).

Furthermore, we want to compile our compiler so that it can run on a machine running **B.**
Our compiler is written in L!
How can we compiler our compiler?

There are many many ways we could go about this task.
The following slides simply sketch out one plausible route to fame and fortune.

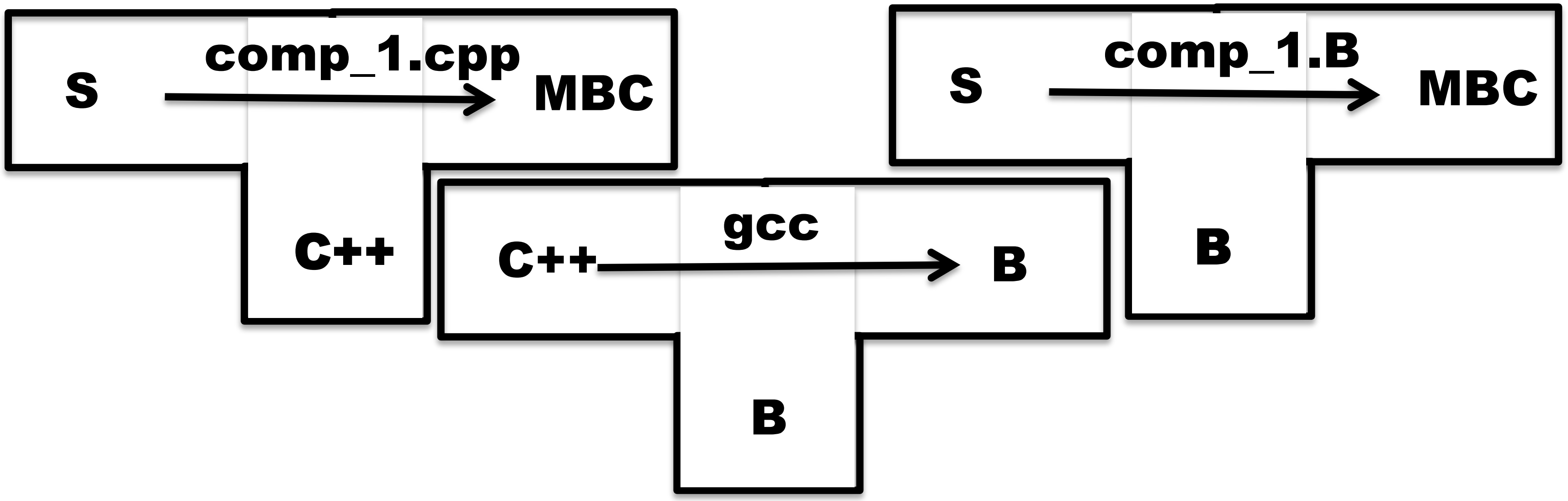
Step 1
Write a small interpreter (VM) for a small language of byte codes

MBC = My Byte Codes



The **zoom** machine!

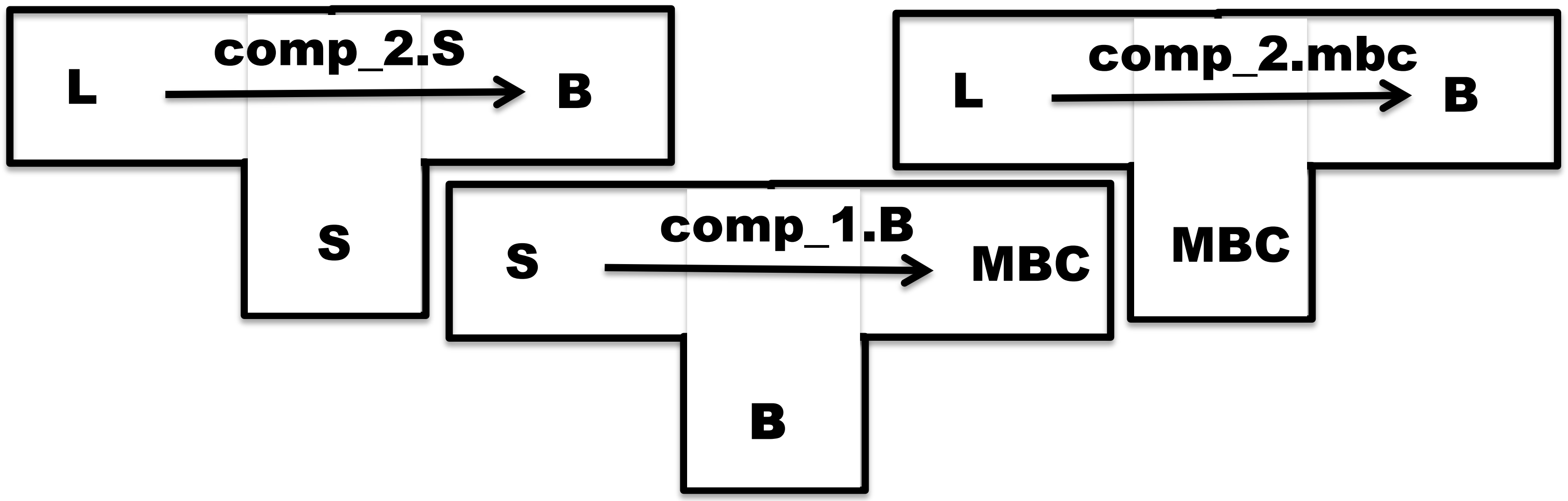
Step 2
Pick a small subset S of L and write a translator from S to MBC



Write **comp_1.cpp** by hand. (It sure would be nice if we could hide the fact that this is written is C++.)

Compiler **comp_1.B** is produced as output from **gcc** when **comp_1.cpp** is given as input.

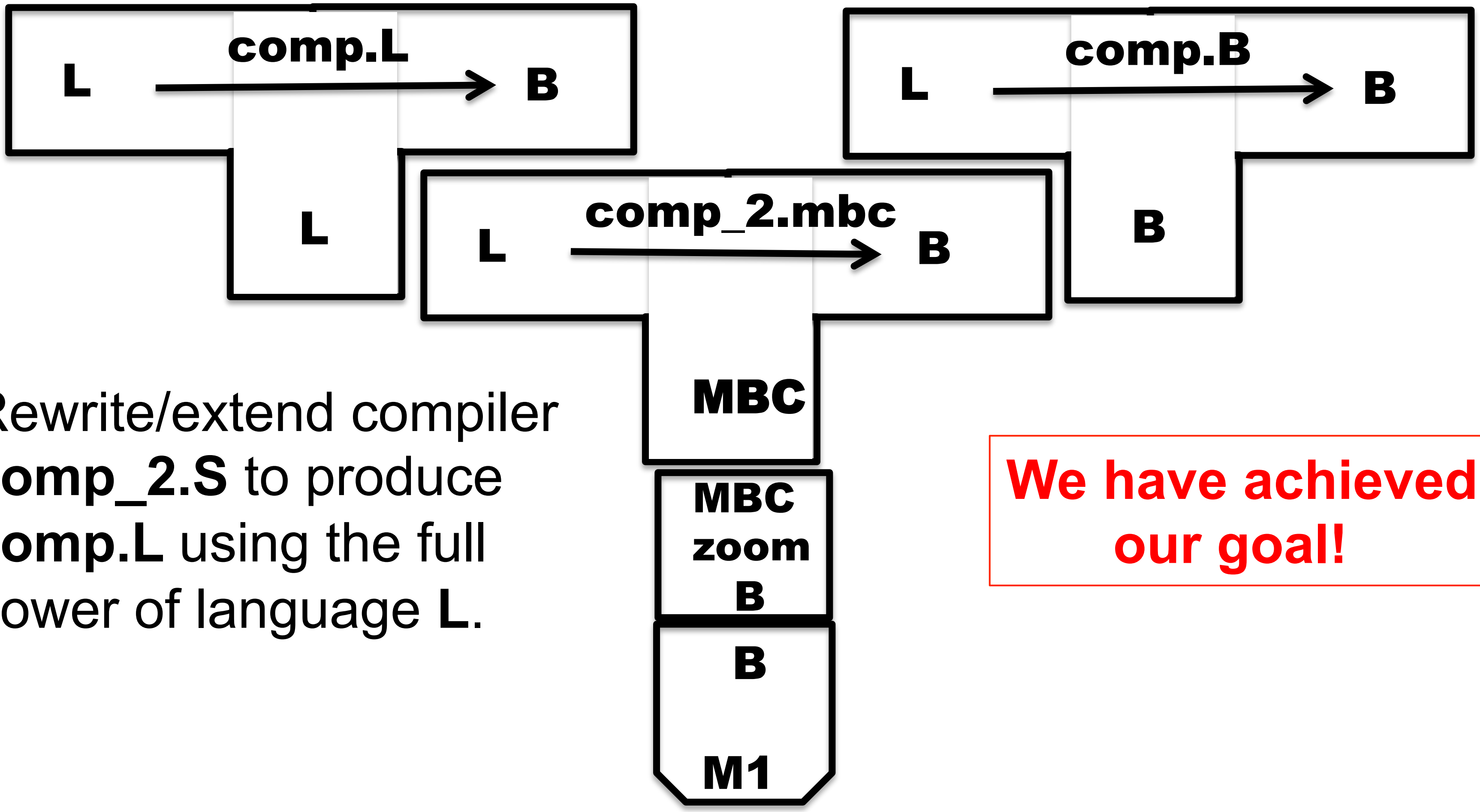
Step 3
Write a compiler for L in S



Write a compiler **comp_2.S** for the full language **L**, but written only in the sub-language **S**.

Compile **comp_2.S** using **comp_1.B** to produce **comp_2.mbc**

Step 4
Write a compiler for L in L, and then compile it!

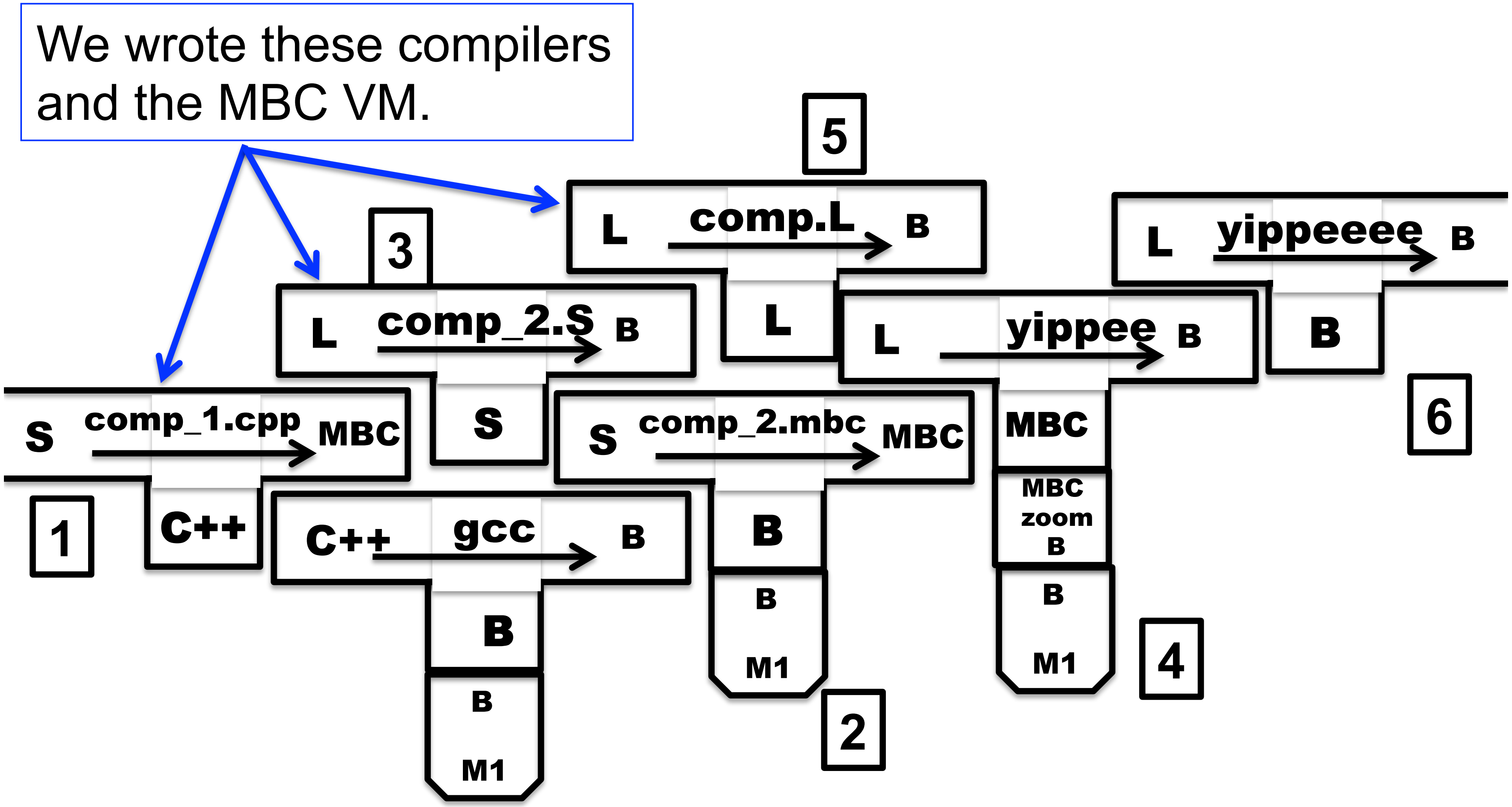


Rewrite/extend compiler **comp_2.S** to produce **comp.L** using the full power of language **L**.

We have achieved our goal!

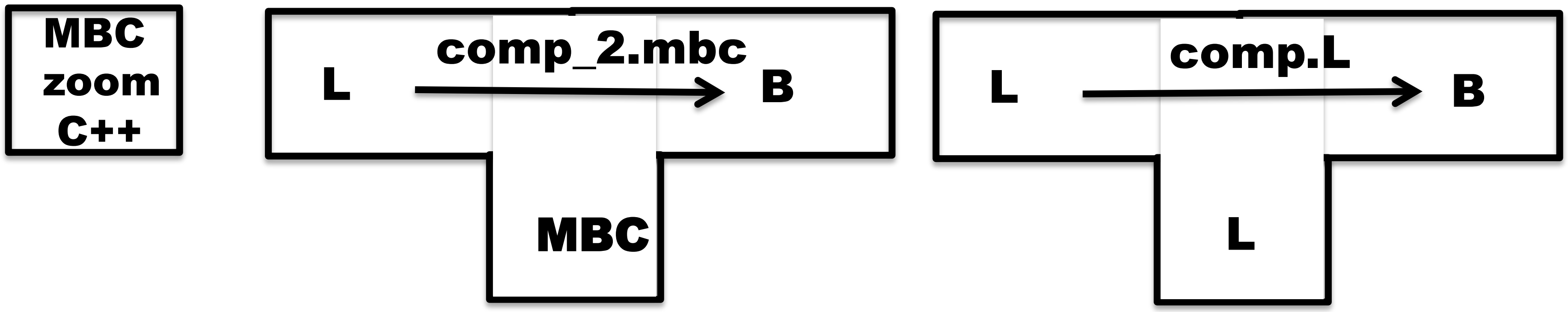
Putting it all together

We wrote these compilers and the MBC VM.



Step 5 : Cover our tracks and leave the world mystified and amazed!

Our **L** compiler download site contains only three components:



comp_2.mbc is a just file of **bytes**. We give it the mysterious and intimidating name : **voodoo**

Shhhh! Don't tell anyone that we wrote the first compiler in C++

Our instructions:

1. Use **gcc** to compile the **zoom** interpreter
2. Use **zoom** to run **voodoo** with input **comp.L** to output the compiler **comp.B**. MAGIC!

Another example (Mogensen, Page 285)

Solving a different problem.

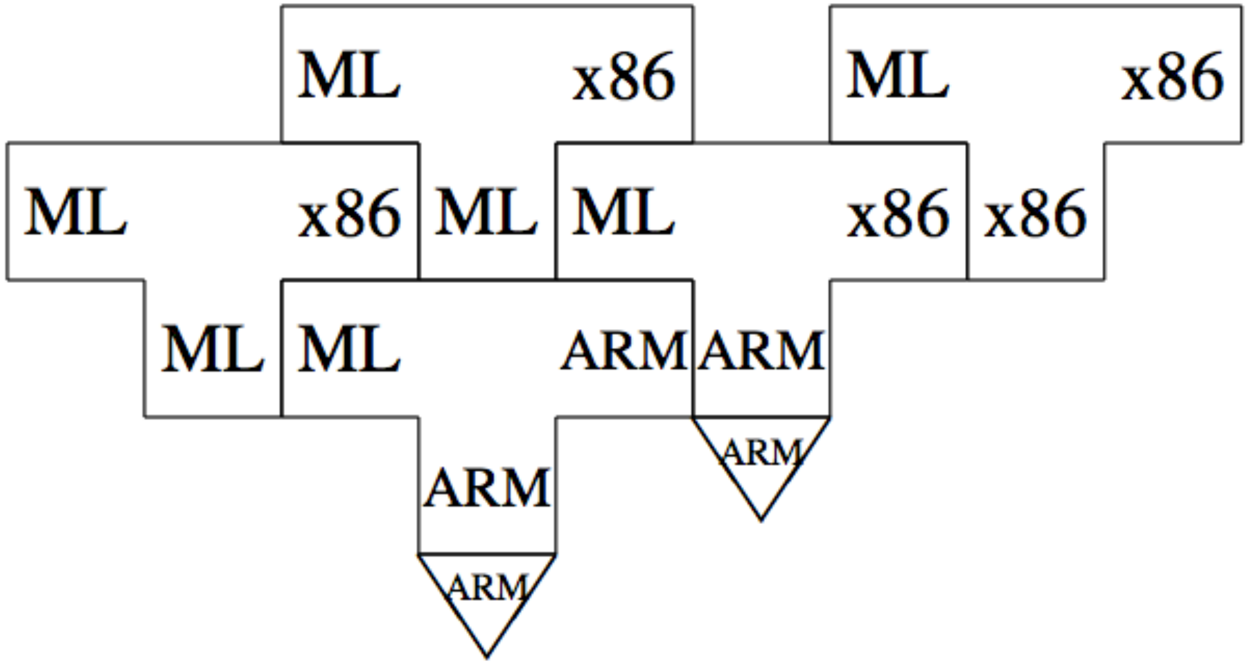
You have:

(1) An ML compiler on ARM. Who knows where it came from.

(2) An ML compiler written in ML, generating x86 code.

You want:

An ML compiler generating x86 and running on an x86 platform.


ML
x86
ML
x86
ML
x86
ML
ML
x86
x86
ML
ML
ARM
ARM
ARM
ARM
ARM