Compiler Construction
Lent Term 2013
Lectures 9 & 10 (of 16)

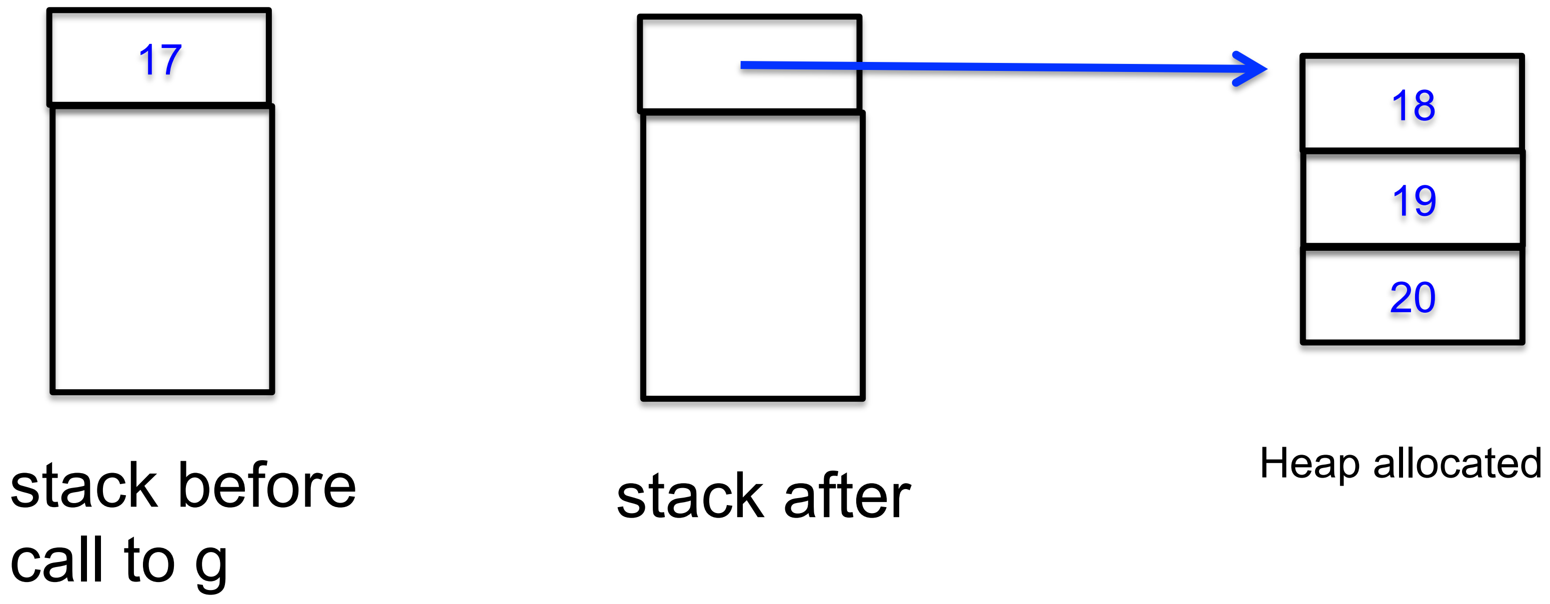
- **Assorted topics**
  - **Tuples/records**
  - **A peek at universal polymorphism**
  - **exceptions**
  - **linking and loading**
  - **bootstrapping**

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

Tuples (in ML-like, L3-like languages)

```
g: int -> int * int * int

fun g x = (x+1, x+2, x+3)

   . . . (g 17) . . .
```

Not showing "header" needed for garbage collector

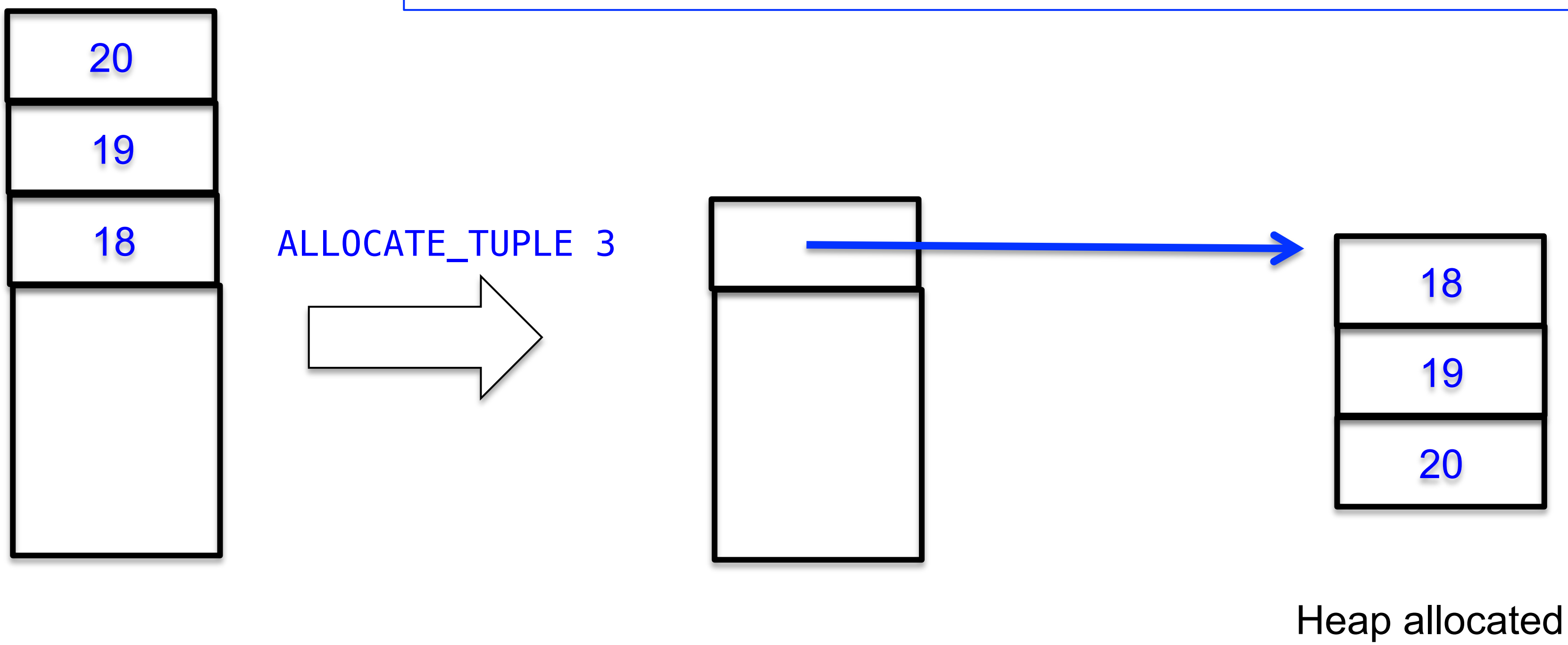


Tuples (in ML-like, L3-like languages)

```
fun g x = (x+1, x+1, x+3)
```

Some IR

```
fun g x =
   let val y1 = x+1
       val y2 = x+2
       val y3 = x+3
   in return (ALLOCATE_TUPLE 3) end
```



Tuples (in ML-like, L3-like languages)

```
fun g x = (x+1, x+1, x+3)

fun f (u, v, w) = u + v + w

   . . . f (g 17) . . .
```

- Does function f take 3 arguments or 1?
- How would you inline f?

Tuples (in ML-like, L3-like languages)

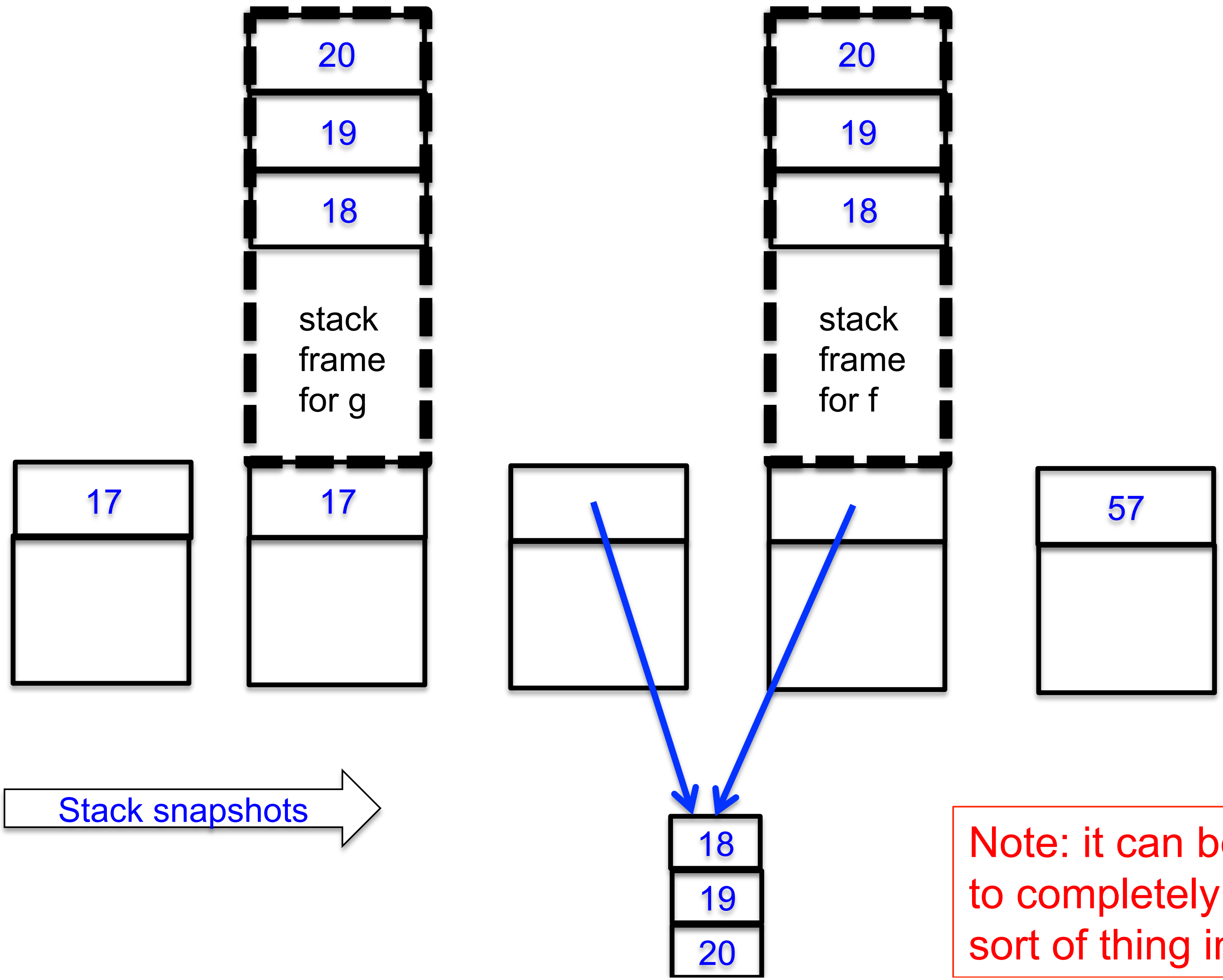
```
fun g x = (x+1, x+1, x+3)

fun f (u, v, w) = u + v + w

   . . . f (g 17) . . .
```

f takes a single argument of type `int * int * int`

Some IR

```
fun f t = let u = EXTRACT_FIELD(t, 1)
              v = EXTRACT_FIELD(t, 2)
              w = EXTRACT_FIELD(t, 3)
          in u + v + w end
```

Naïve evaluation of `f(g 17)`



Note: it can be very hard to completely avoid this sort of thing in general …

Inline (g 17)

```
let val y1 = 17+1
    val y2 = 17+2
    val y3 = 17+3
in return (ALLOCATE_TUPLE 3) end
```

⇩ "constant folding"

```
let val y1 = 18
    val y2 = 19
    val y3 = 20
in return (ALLOCATE_TUPLE 3) end
```

Inline f and g ?

```
let t = let val y1 = 18
             val y2 = 19
             val y3 = 20
            in return (ALLOCATE_TUPLE 3) end
in
   let u = EXTRACT_FIELD(t, 1)
                v = EXTRACT_FIELD(t, 2)
                w = EXTRACT_FIELD(t, 3)
   in u + v + w end
end
```

Hmm, with enough equations we should be able to rewrite this at compile time to 57 !

A peek at universal polymorphism
(not examinable)

```
map : ('a -> 'b) -> 'a list -> 'b list

fun map f [] = []
  | map f (a::rest) = (f a) :: (map f rest)
```

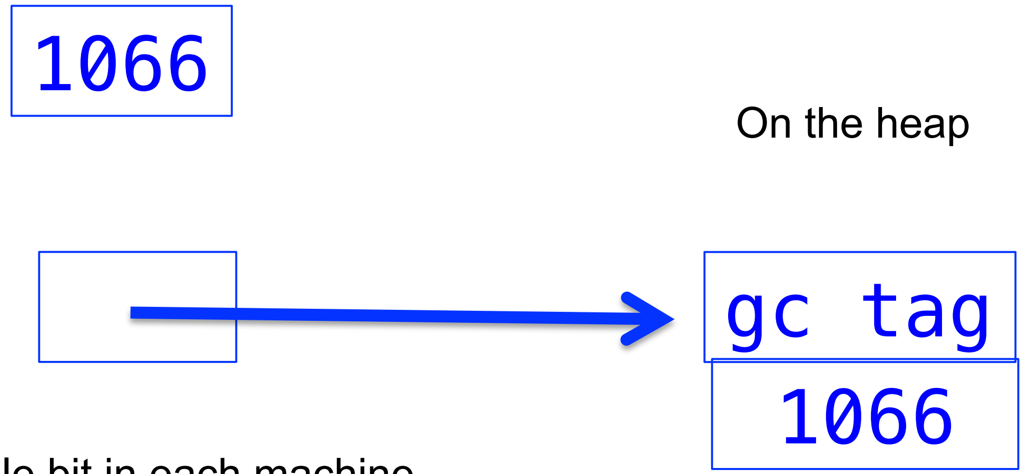
The code generated for `map` must work
for any times `'a` and `'b`.

So it seems that all values of any type must
be represented by objects of <u>the same size</u>.

Boxing and Unboxing (not examinable)

An unboxed integer :

A boxed integer :



Some ML compilers use a single bit in each machine word to distinguish boxed from unboxed values.

It is better to work with unboxed values than with boxed values.

Compilers for ML-like languages must Expend a good deal of effort trying to Find good optimizations for boxed/unboxed choices.

See Appel Chapter 16.

Similar terminology is used Java for putting a value in a container class (boxing) and taking it out (unboxing)

For example, put an int into the Integer container class.

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')

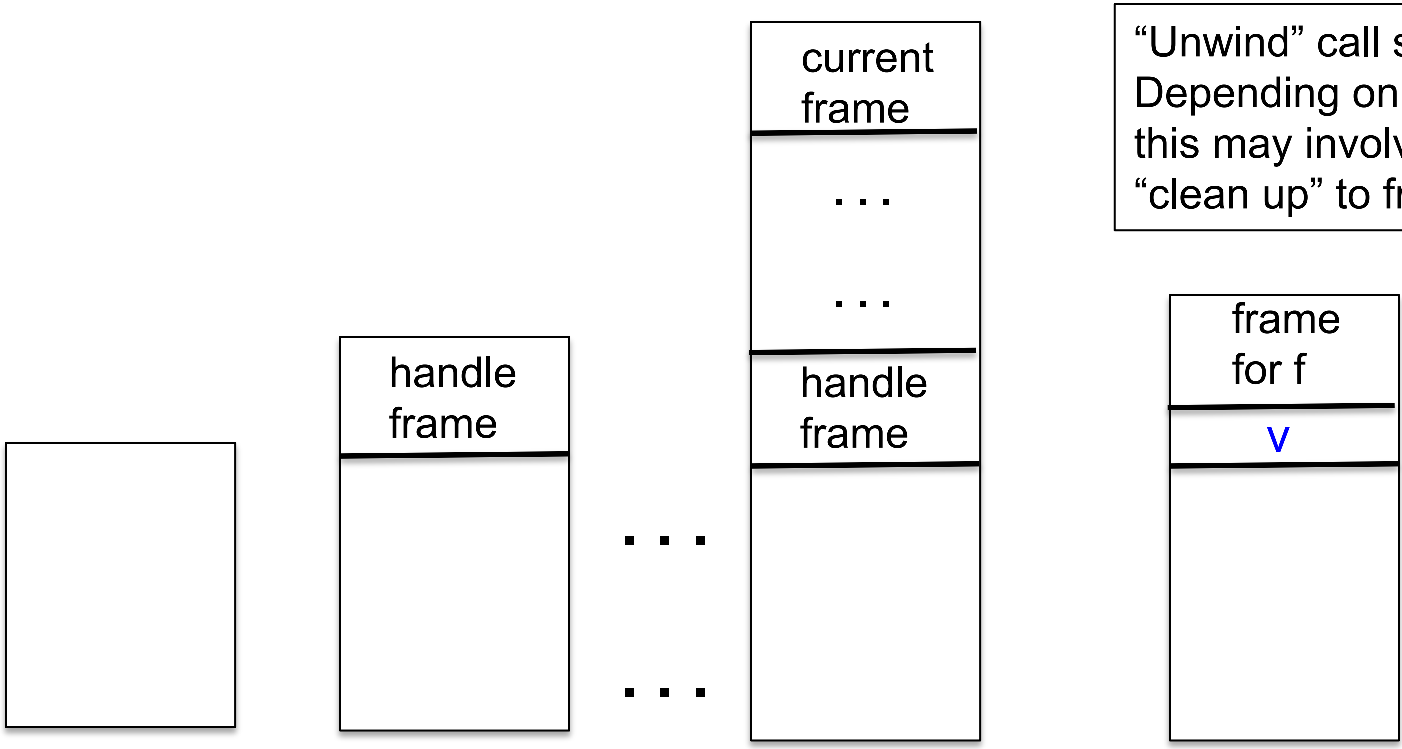
`raise e`

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

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



Call stack just before evaluating code for

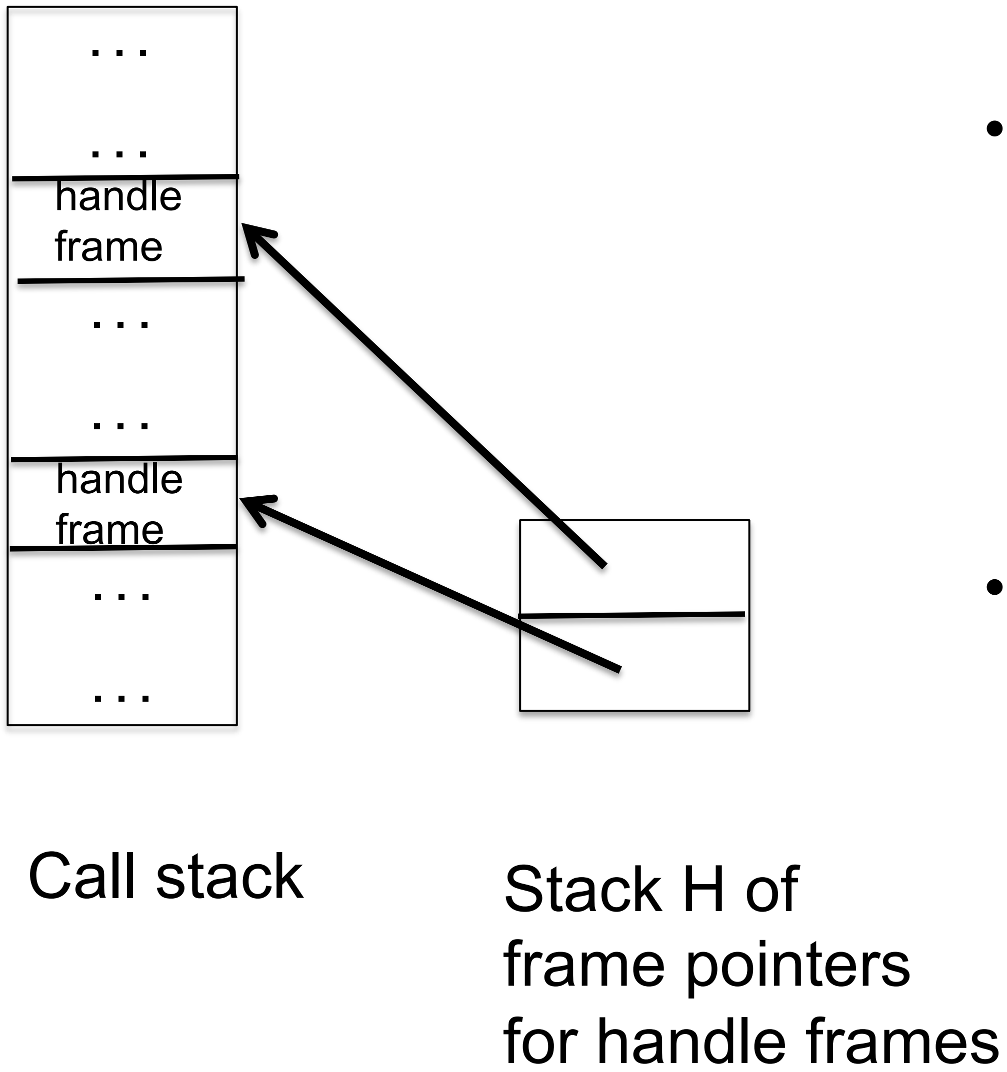
`e handle f`

Push a special frame for the handle

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

This assumes that the unwound stack contains no other handle frames

One possible implementation: use an auxiliary stack of frame pointers that LIFO records the handle frames



- The address (or closure pointer)of each handle function could be saved in the associated stack frame or in H
- An alternative to H is to have a chain of frame pointers linking the handle frames together.

Possible implementation

```
e handle f
```

```
fun _h27 () = push address of f;
              push current fp on H;
              code for e;
              return top-of-stack
```

```
raise v
```

```
h_fp := pop (H);
fp := h_fp
f  := content of fp + offset,
      the saved address of
      handler code ;
restore frame previous to fp;
push v;
call f;
```

Exceptions

For example given **exception foo;** we could implement

```
try C1 except foo => C2 end; C3
```

as (using H as a stack of active exception labels)

```
    push(H, L2);
    C1
    pop(H);
    goto L3:
L2: if (raised_exc != foo) doraise(raised_exc);
    C2;
L3: C3;
```

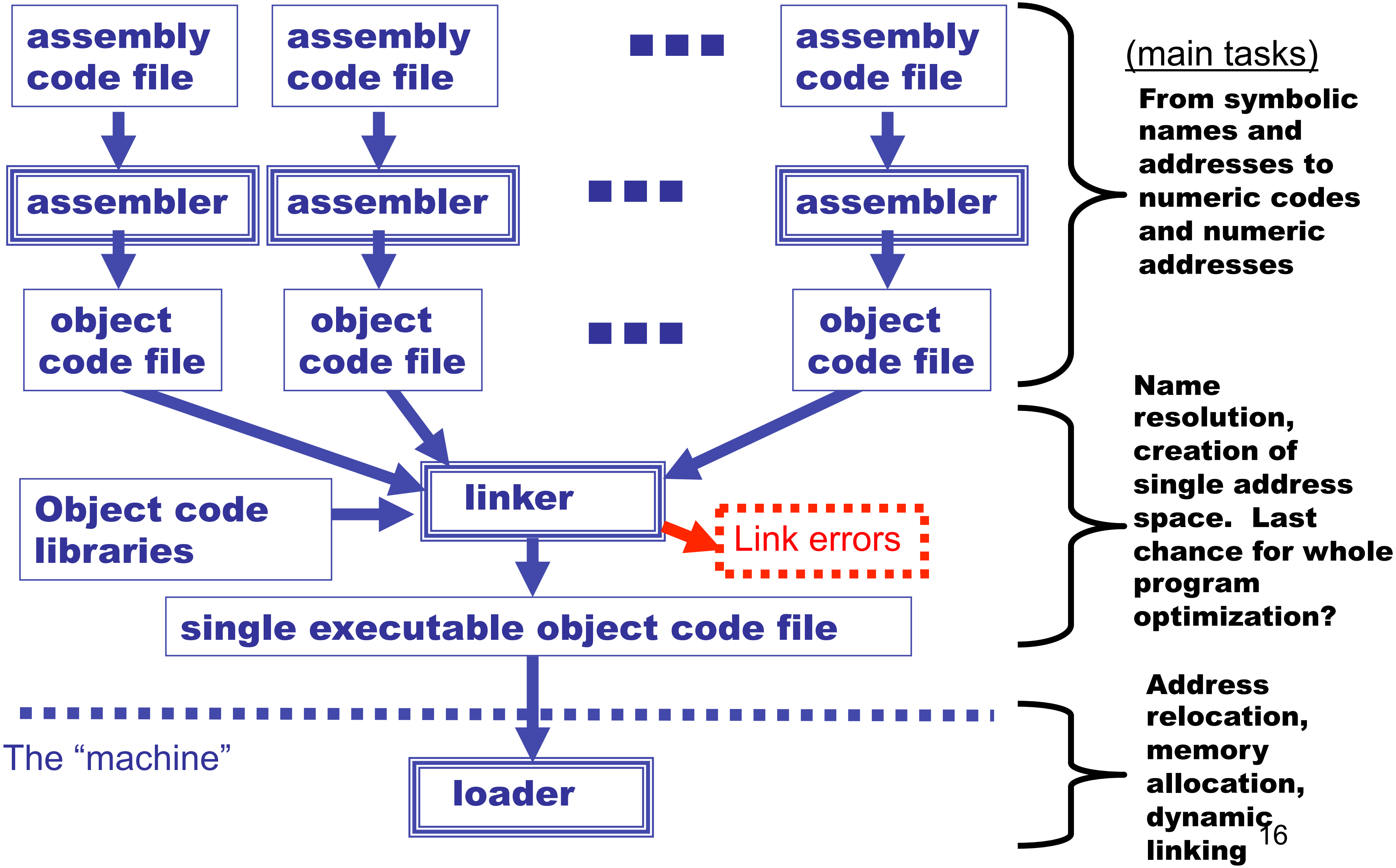
and the **doraise()** function looks like

```
void doraise(exc)
{   raised_exc = exc;
    goto pop(H);
}
```

From lecture notes of 2012

Linking and Loading

This functionality may or may not be implemented in “the compiler”.



Object files

Must contain at least

- Program instructions
- Symbols being exported
- Symbols being imported

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) by which symbol
`.rela.data` data segment relocation table: list of (offset,symbol) pairs giving: (i) offset within `.data` to be relocated; and (ii) 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.

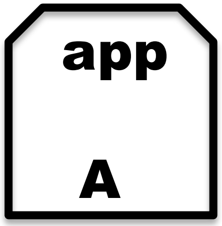
Pros and Cons of dynamic linking:

(+) Executables are smaller

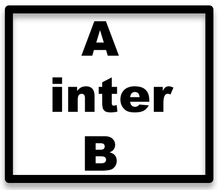
(+) Bug fixes to a library don’t require re-linking as the new versior is automatically demand-loaded every time the program is run.

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

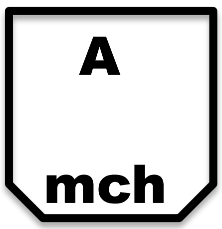
Bootstrapping. We need some notation . . .



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

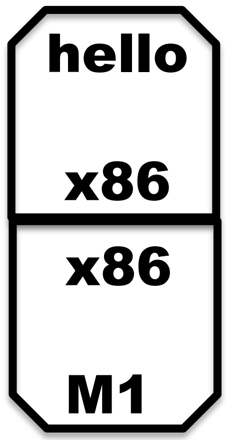


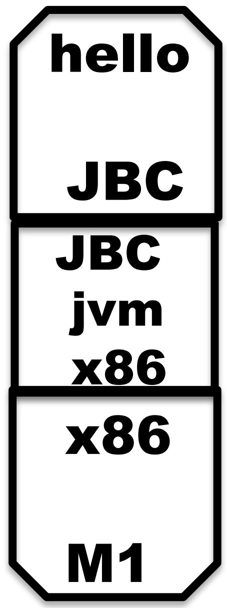
An interpreter or VM for language **A** Written in language **B**



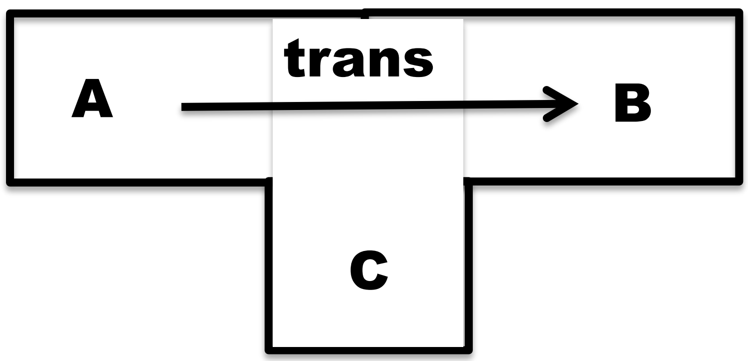
A machine called **mch** running language **A** natively.

Simple Examples



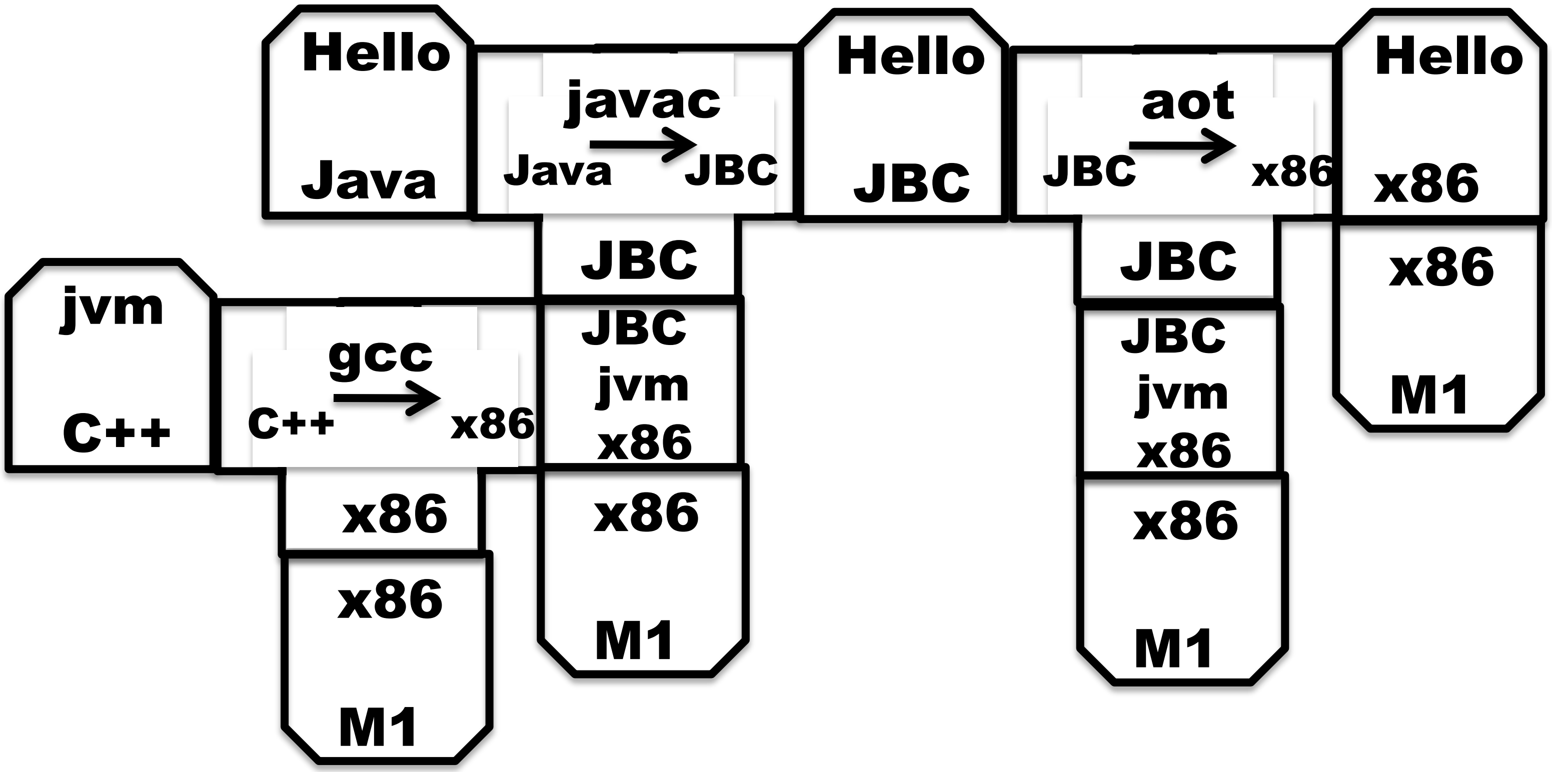


Tombstones



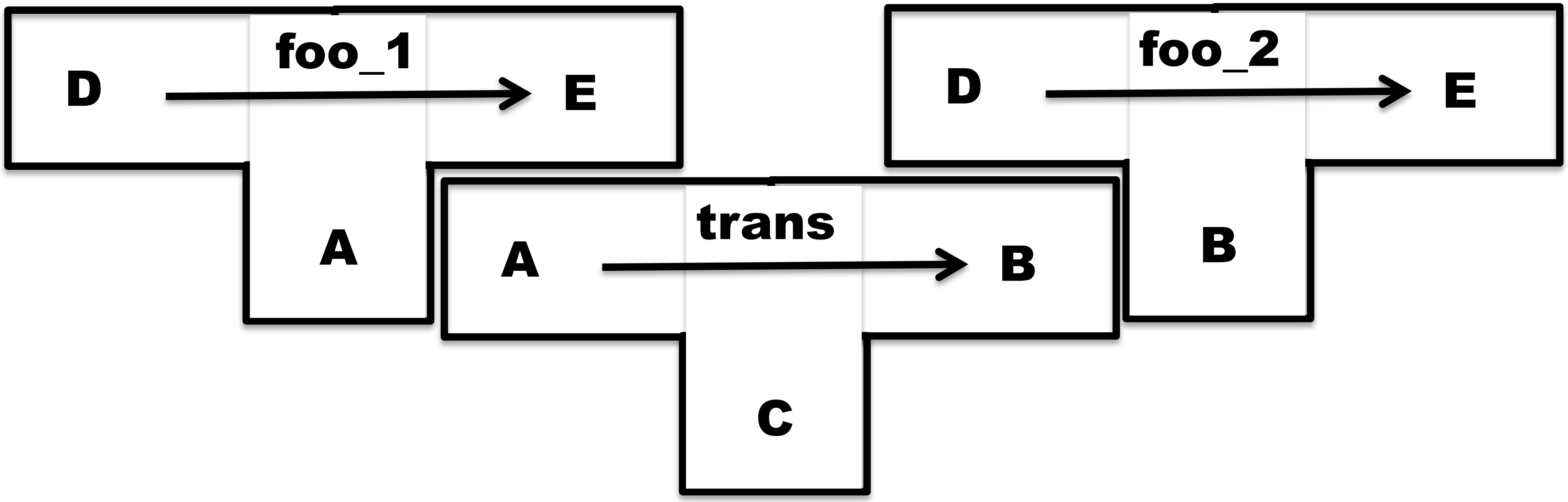
This is an application called **trans** that translates programs in language **A** into programs in language **B**, and it is written in language **C**.

Ahead-of-time compilation



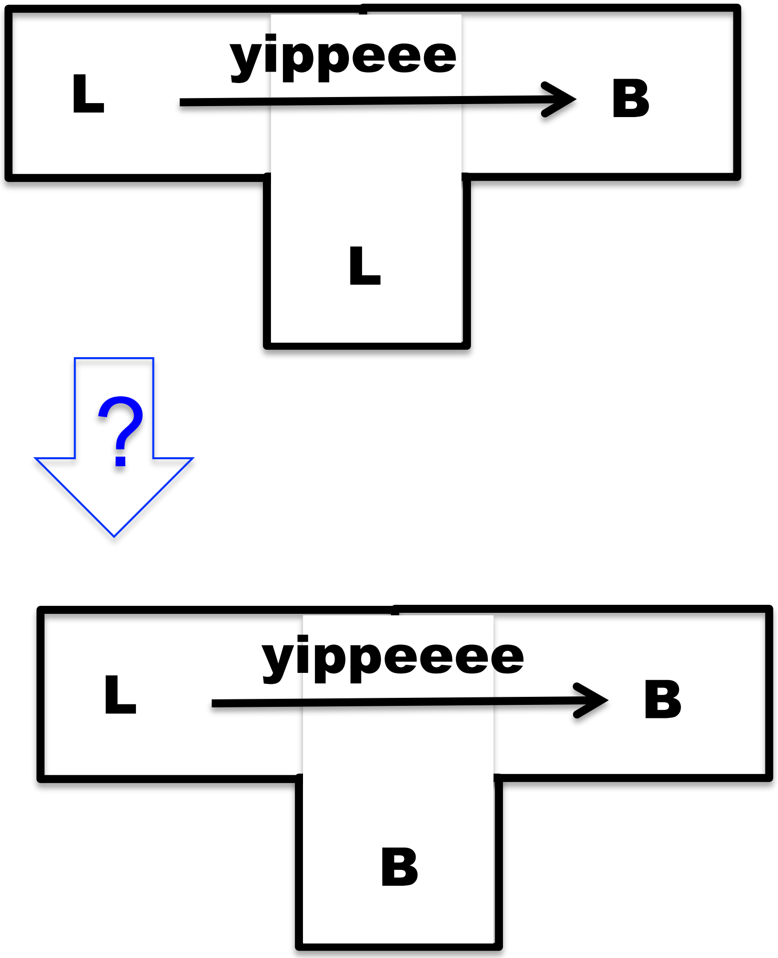
Thanks to David Greaves
for the example.

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).

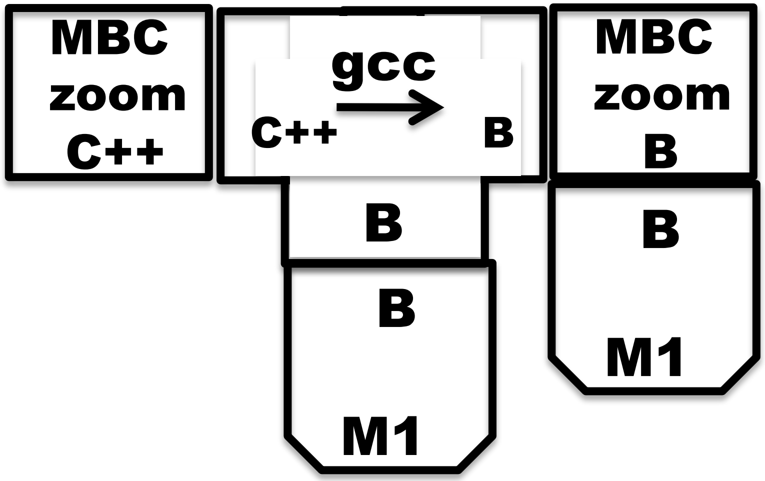
Furthermore, we want to compile our compiler so that it can run on a machine running **B.**

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.

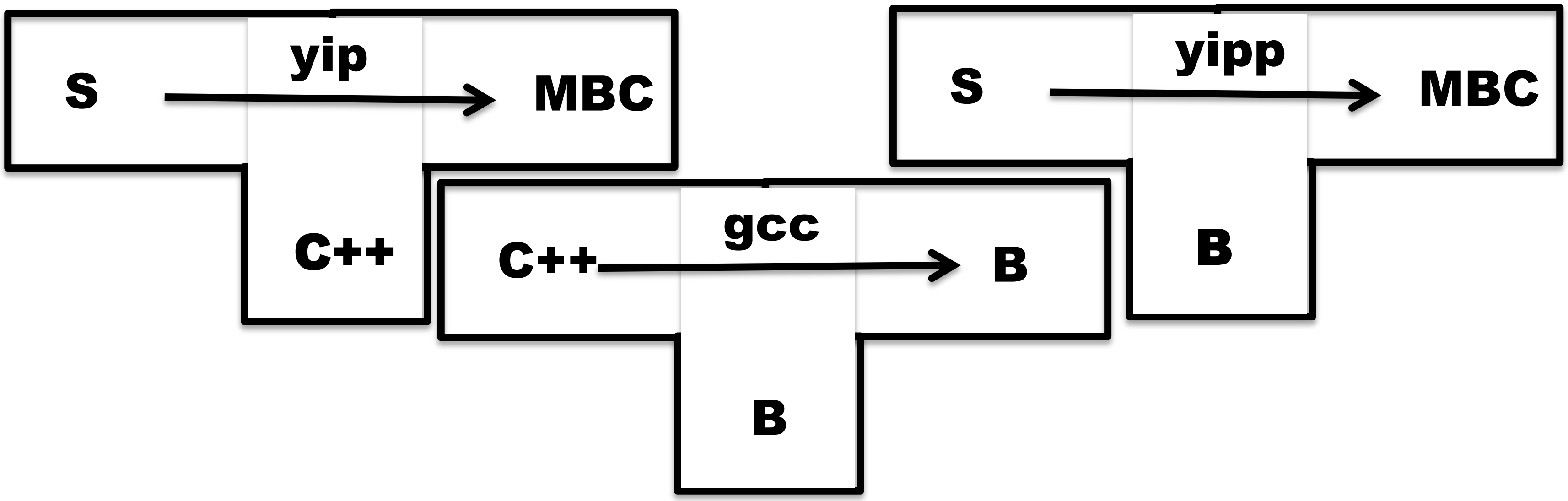
Step 1
Write a small interpreter (VM) for a small language of byte codes

MBC = My Byte Codes



The **zoom** machine!

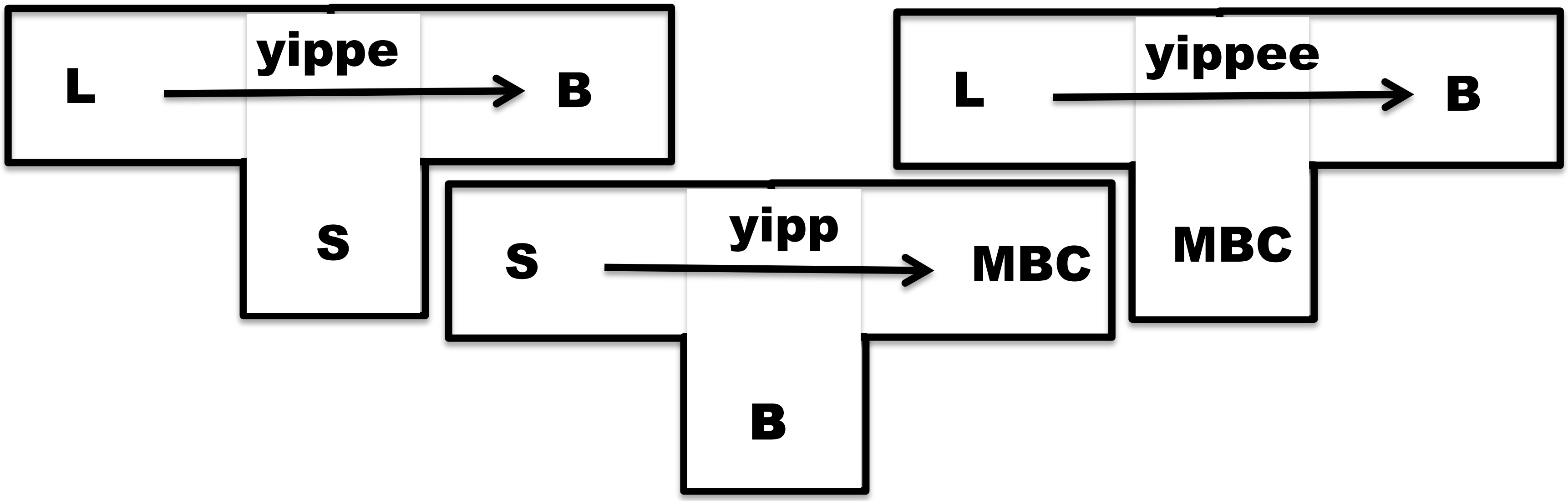
Step 2
Pick a small subset S of L and write a translator from S to MBC



Write **yip** by hand. (We are rather ashamed that this is written is C++.)

Translator **yipp** is produced as output from **gcc** when **yip** is given as input.

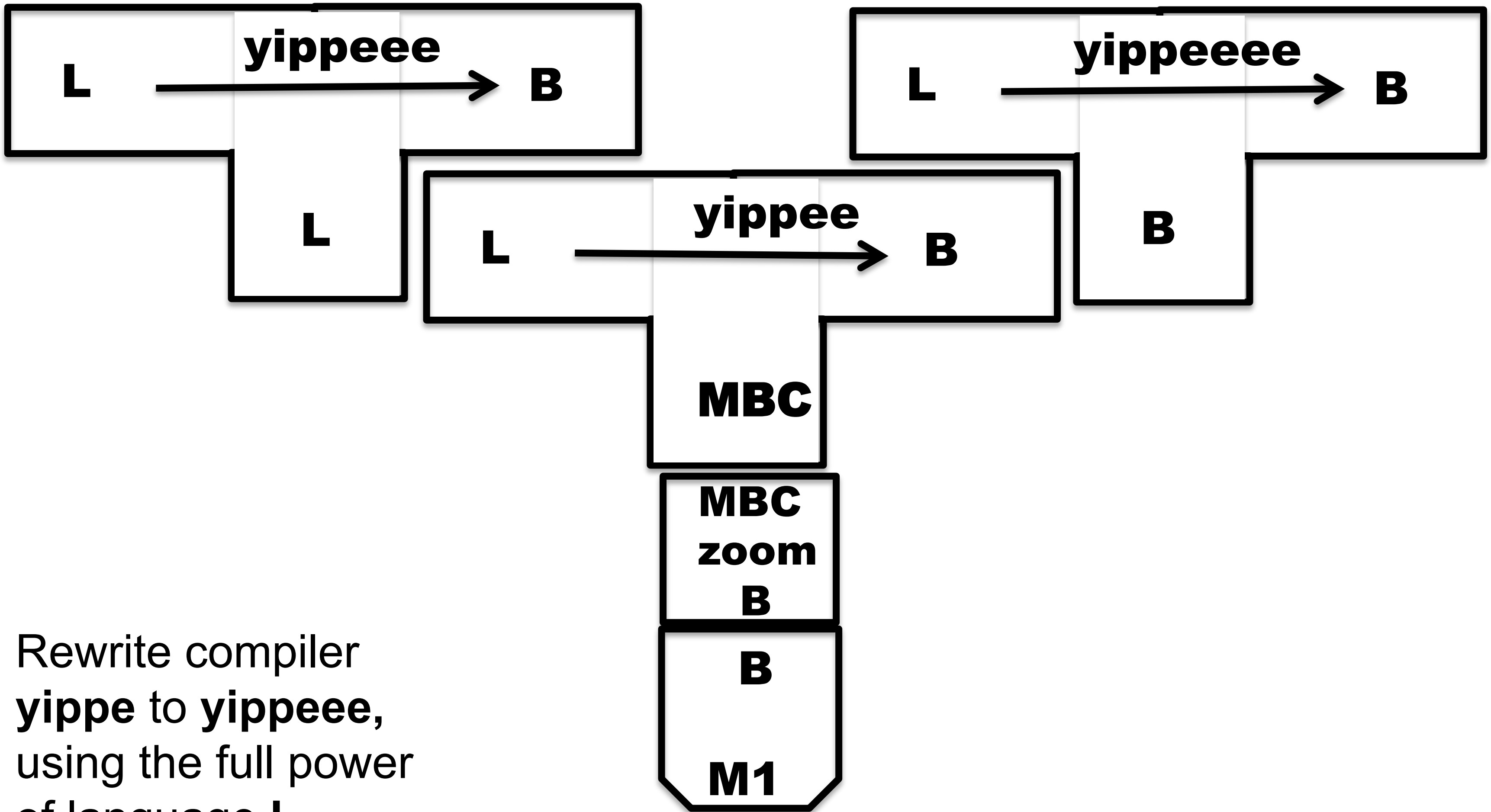
Step 3
Write a compiler for L in S



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

Compile **yippe** using **yipp** to produce **yippee**

Step 4
Write a compiler for L in L

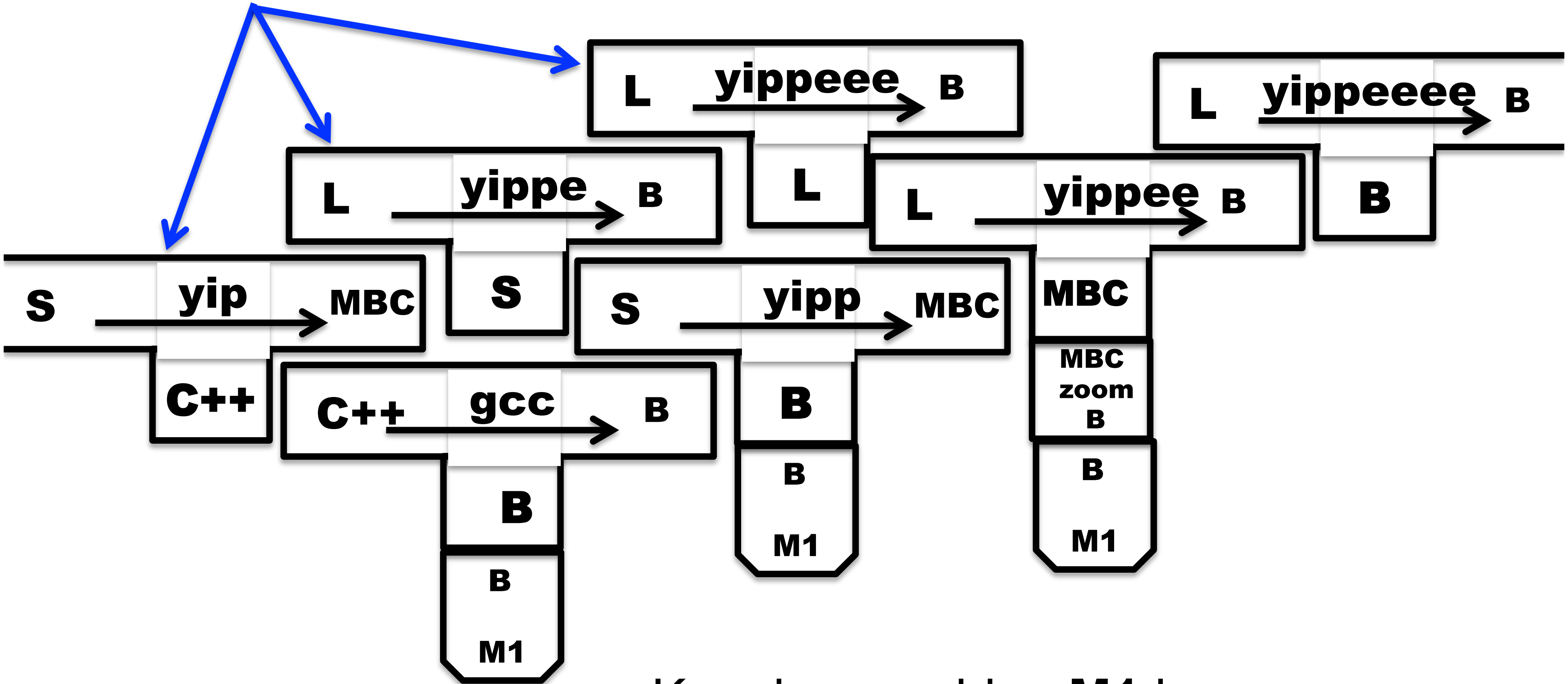


Rewrite compiler **yippe** to **yippeee,** using the full power of language **L**.

Now compile this using **yippee** to obtain our goal!

Putting it all together

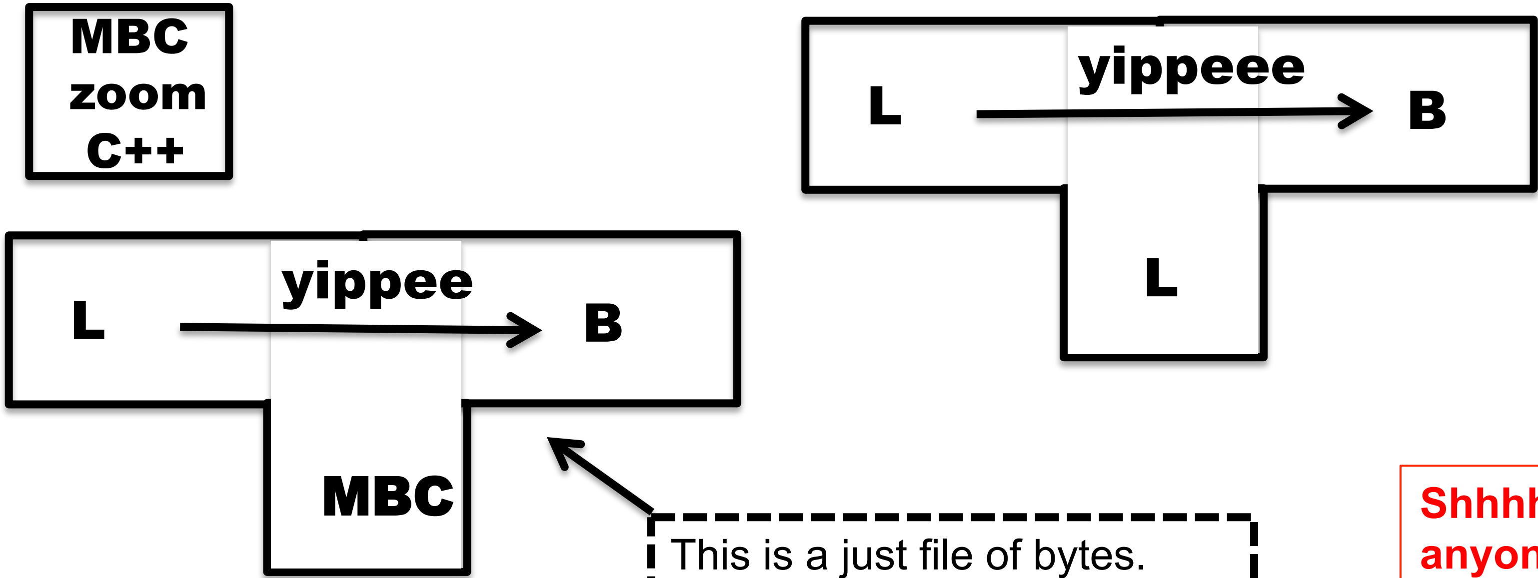
We wrote only these compilers and the MBC VM.



Keeping machine **M1** busy . . .

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

Our **L** compiler download site contains <u>only three</u> components:



This 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 **yippeee** to produce output the compiler **yippeeee**