Compiler Construction
Lent Term 2017

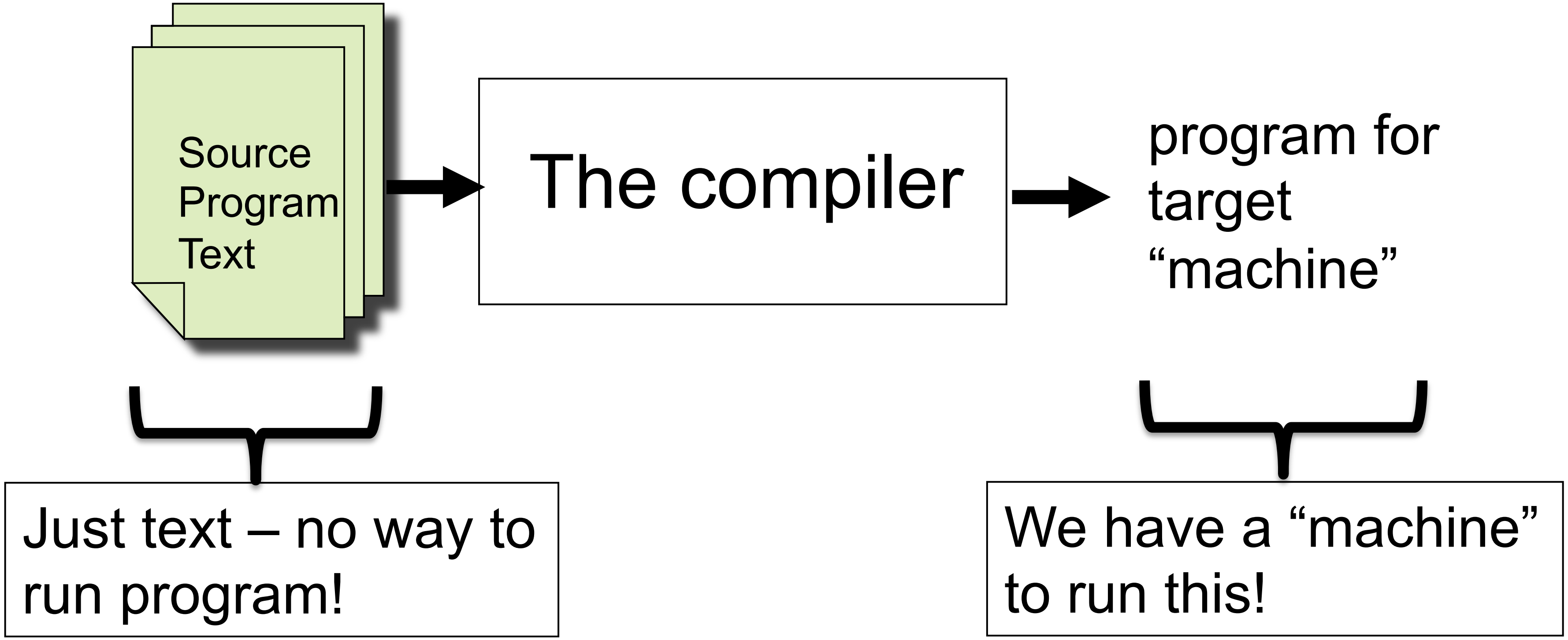
Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge

Why Study Compilers?

- **Although many of the basic ideas were developed over 50 years ago, compiler construction is still an evolving and active area of research and development.**
- **Compilers are intimately related to programming language design and evolution.**
- **Compilers are a Computer Science success story illustrating the hallmarks of our field --- higher-level abstractions implemented with lower-level abstractions.**
- **Every Computer Scientist should have a basic understanding of how compilers work.**

Compilation is a special kind of translation



A good compiler should ...

- **be correct in the sense that meaning is preserved**
- **produce usable error messages**
- **generate efficient code**
- **itself be efficient**
- **be well-structured and maintainable**

This course! (first two items)

OptComp, Part II (generate efficient code)

Pick any 2?

Just 1?

Mind The Gap

High Level Language	Typical Target Language
• “Machine” independent	• “Machine” specific
• Complex syntax	• Simple syntax
• Complex type system	• Simple types
• Variables	• memory, registers, words
• Nested scope	• Single flat scope
• Procedures, functions	
• Objects	
• Modules	
• …	

Help!!! Where do we begin???

The Gap, illustrated

```
public class Fibonacci {
   public static long fib(int m) {
      if (m == 0) return 1;
      else if (m == 1) return 1;
          else return
                 fib(m - 1) + fib(m - 2);
   }
   public static void
      main(String[] args) {
      int m =
          Integer.parseInt(args[0]);
      System.out.println(
         fib(m) + "\n");
   }
}
```

javac Fibonacci.java
javap –c Fibonacci.class

```
public class Fibonacci {
 public Fibonacci();
  Code:
    0: aload_0
    1: invokespecial #1
    4: return
 public static long fib(int);
  Code:
    0: iload_0
    1: ifne        6
    4: lconst_1
    5: lreturn
    6: iload_0
    7: iconst_1
    8: if_icmpne     13
   11: lconst_1
   12: lreturn
   13: iload_0
   14: iconst_1
   15: isub
   16: invokestatic  #2
   19: iload_0
   20: iconst_2
   21: isub
   22: invokestatic  #2
   25: ladd
   26: lreturn
 public static void
   main(java.lang.String[]);
  Code:
    0: aload_0
    1: iconst_0
    2: aaload
    3: invokestatic  #3
    6: istore_1
    7: getstatic     #4
   10: new           #5
   13: dup
   14: invokespecial #6
   17: iload_1
   18: invokestatic  #2
   21: invokevirtual #7
   24: ldc           #8
   26: invokevirtual #9
   29: invokevirtual #10
   32: invokevirtual #11
   35: return
}
```

JVM bytecodes

The Gap, illustrated

fib.ml

```
(* fib : int -> int *)
let rec fib m =
   if m = 0
   then 1
   else if m = 1
        then 1
        else fib(m - 1) + fib (m - 2)
```

ocamlc –dinstr fib.ml

```
        branch L2
L1:     acc 0
        push
        const 0
        eqint
        branchifnot L4
        const 1
        return 1
L4:     acc 0
        push
        const 1
        eqint
        branchifnot L3
        const 1
        return 1
L3:     acc 0
        offsetint -2
        push
        offsetclosure 0
        apply 1
        push
        acc 1
        offsetint -1
        push
        offsetclosure 0
        apply 1
        addint
        return 1
L2:     closurerec 1, 0
        acc 0
        makeblock 1, 0
        pop 1
        setglobal Fib!
```

OCaml VM bytecodes

The Gap, illustrated

fib.c

```
#include<stdio.h>

int Fibonacci(int);
int main()
{
  int n;
  scanf("%d",&n);
  printf("%d\n", Fibonacci(n));
  return 0;
}

int Fibonacci(int n)
{
  if ( n == 0 ) return 0;
  else if ( n == 1 ) return 1;
  else return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

gcc –S fib.c

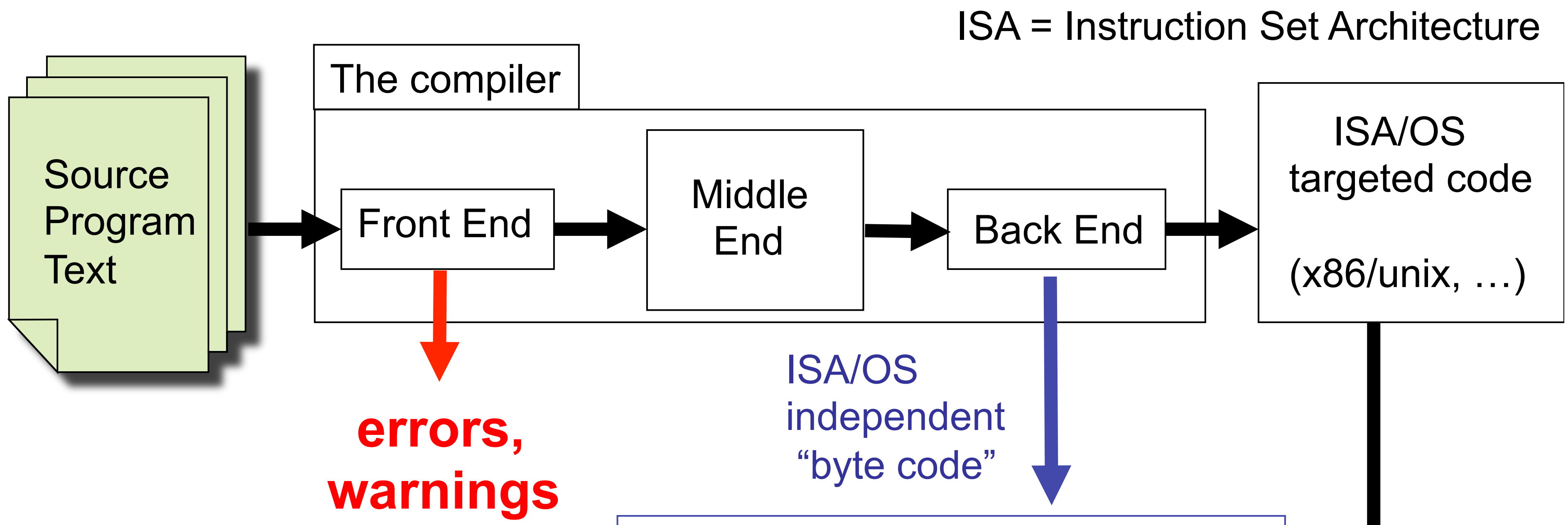
The Gap, illustrated

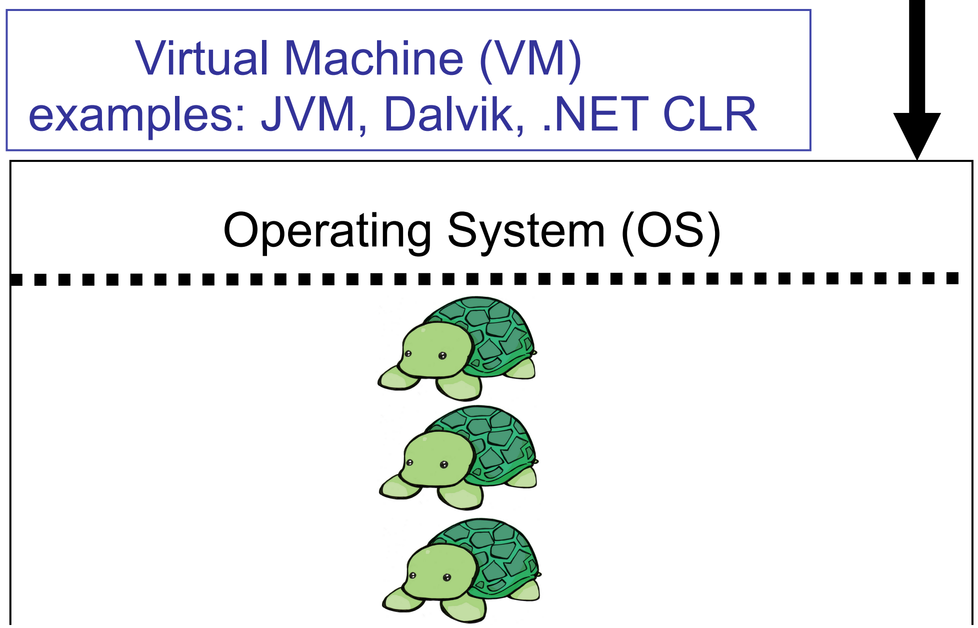
```
                .section        __TEXT,__text,regular,pure_instructions
                .globl          _main
                .align          4, 0x90
_main:                  ## @main
                .cfi_startproc
## BB#0:
                pushq           %rbp
Ltmp2:
                .cfi_def_cfa_offset 16
Ltmp3:
                .cfi_offset %rbp, -16
                movq            %rsp, %rbp
Ltmp4:
                .cfi_def_cfa_register %rbp
                subq            $16, %rsp
                leaq            L_.str(%rip), %rdi
                leaq            -8(%rbp), %rsi
                movl            $0, -4(%rbp)
                movb            $0, %al
                callq           _scanf
                movl            -8(%rbp), %edi
                movl            %eax, -12(%rbp)      ## 4-byte Spill
                callq           _Fibonacci
                leaq            L_.str1(%rip), %rdi
                movl            %eax, %esi
                movb            $0, %al
                callq           _printf
                movl            $0, %esi
                movl            %eax, -16(%rbp)      ## 4-byte Spill
                movl            %esi, %eax
                addq            $16, %rsp
                popq            %rbp
                ret
                .cfi_endproc

                .globl          _Fibonacci
                .align          4, 0x90
_Fibonacci:                 ## @Fibonacci
                .cfi_startproc
## BB#0:
                pushq           %rbp
Ltmp7:
                .cfi_def_cfa_offset 16
Ltmp8:
                .cfi_offset %rbp, -16
                movq            %rsp, %rbp
Ltmp9:
```

```
                .cfi_def_cfa_register %rbp
                subq            $16, %rsp
                movl            %edi, -8(%rbp)
                cmpl            $0, -8(%rbp)
                jne             LBB1_2
## BB#1:
                movl            $0, -4(%rbp)
                jmp             LBB1_5
LBB1_2:
                cmpl            $1, -8(%rbp)
                jne             LBB1_4
## BB#3:
                movl            $1, -4(%rbp)
                jmp             LBB1_5
LBB1_4:
                movl            -8(%rbp), %eax
                subl            $1, %eax
                movl            %eax, %edi
                callq           _Fibonacci
                movl            -8(%rbp), %edi
                subl            $2, %edi
                movl            %eax, -12(%rbp)      ## 4-byte Spill
                callq           _Fibonacci
                movl            -12(%rbp), %edi      ## 4-byte Reload
                addl            %eax, %edi
                movl            %edi, -4(%rbp)
LBB1_5:
                movl            -4(%rbp), %eax
                addq            $16, %rsp
                popq            %rbp
                ret
                .cfi_endproc

                .section        __TEXT,__cstring,cstring_literals
L_.str:                 ## @.str
                .asciz          "%d"

L_.str1:                ## @.str1
                .asciz          "%d\n"


.subsections_via_symbols
```

x86/Mac OS

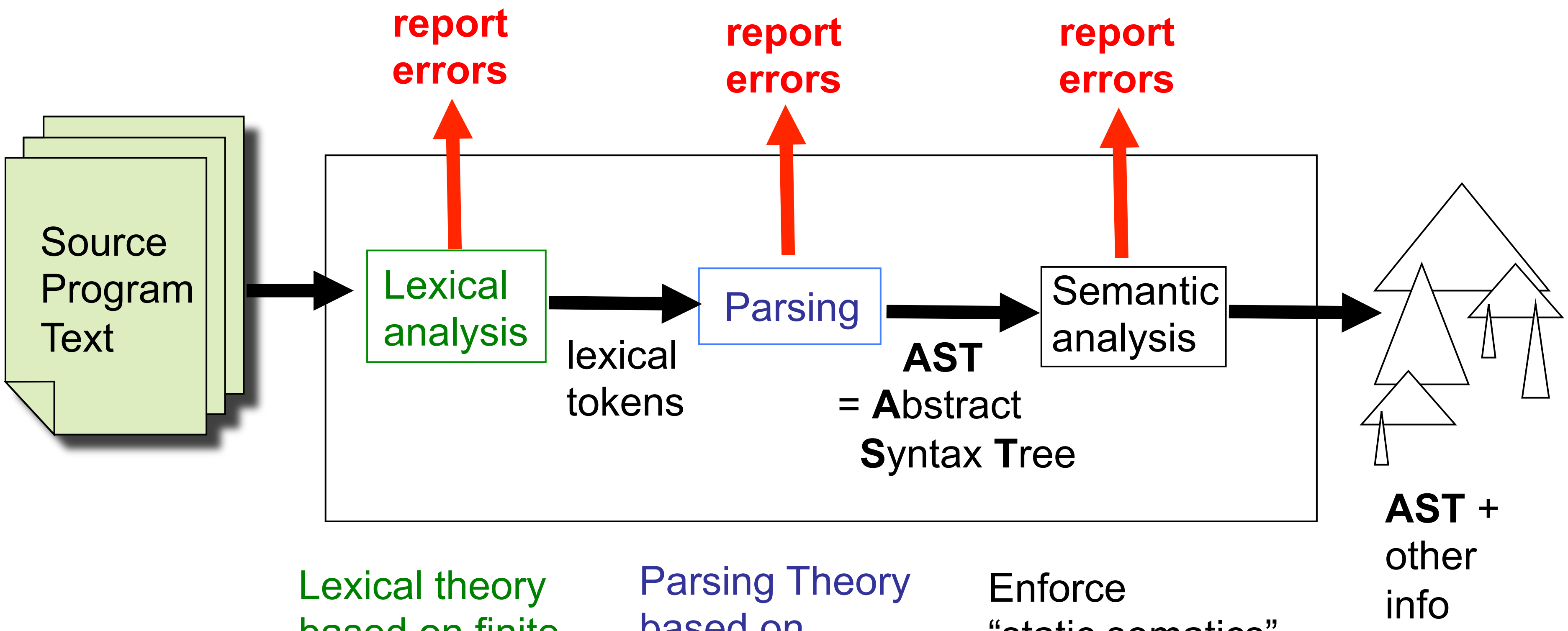
Conceptual view of a typical compiler





Key to bridging The Gap : divide and conquer. The Big Leap is broken into small steps. Each step broken into yet smaller steps …

The shape of a typical "front end"



Lexical theory based on finite automaton and regular expressions

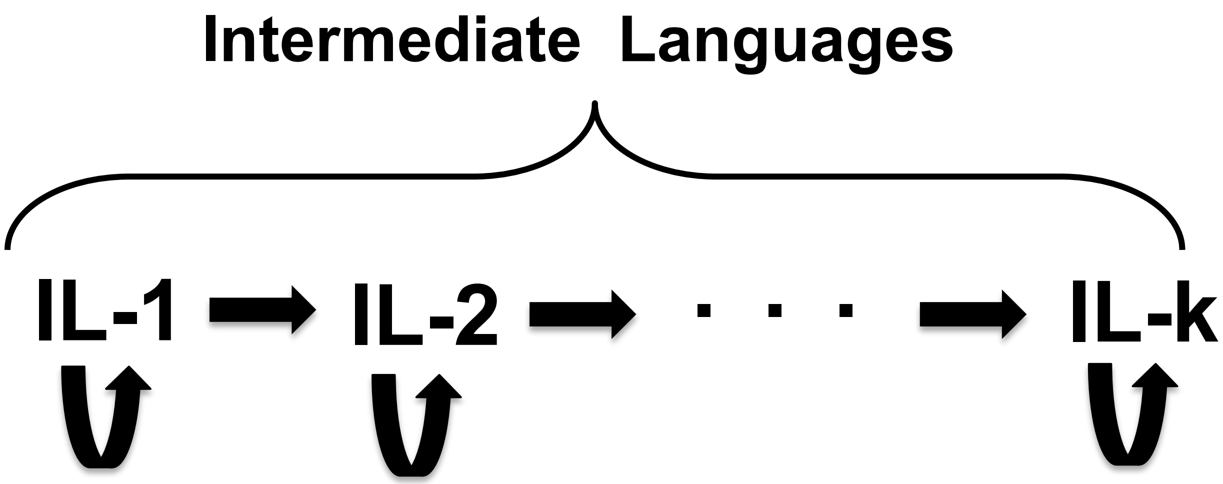
Parsing Theory based on push-down automaton and context-free grammars

Enforce "static sematics" of language: type checking, def/use rules, and so on (**SPL**!)

The AST output from the front-end should represent a legal program in the source language. ("Legal" of course does not mean "bug-free"!)

SPL = Semantics of Programming Languages, Part 1B

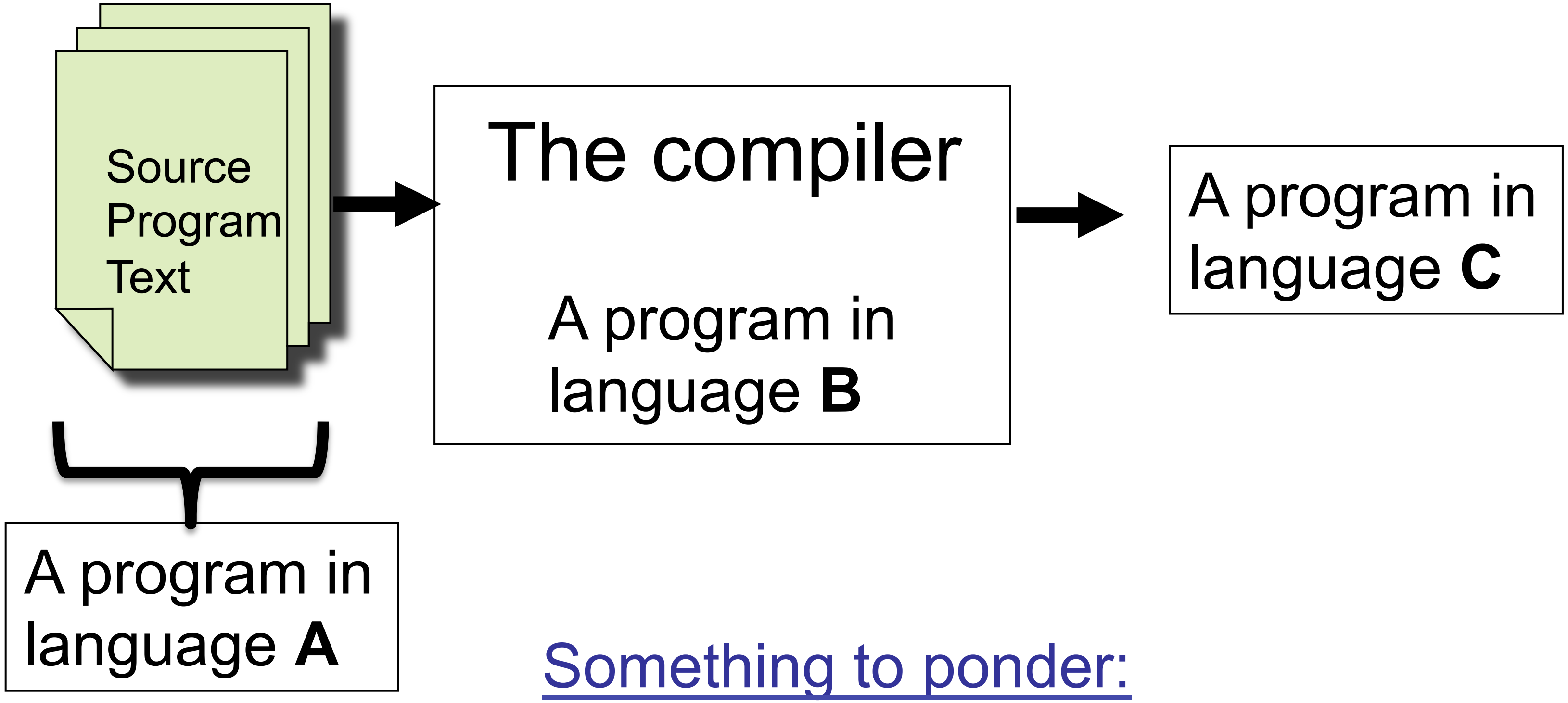
Our view of the middle- and back-ends : a sequence of small transformations



Of course industrial-strength compilers may collapse many small-steps …

- Each **IL** has its own semantics (perhaps informal)
- Each transformation (➡) preserves semantics (**SPL**!)
- Each transformation eliminates only a few aspects of **the gap**
- Each transformation is fairly easy to understand
- Some transformations can be described as “optimizations”
- We will associate each **IL** with its own interpreter/VM. (Again, not something typically done in “industrial-strength” compilers.)

Compilers must be compiled



<u>Something to ponder:</u>
A compiler is just a program.
But how did it get compiled?
The OCaml compiler is written in OCaml.

How was the compiler compiled?

Approach Taken

- We will develop a compiler for a fragment of L3 introduced in Semantics of Programming Languages, Part 1B.
- We will pay special attention to the correctness.
- We will compile only to Virtual Machines (VMs) of various kinds. See Part II optimising compilers for generating lower-level code.
- Our toy compiler is available on the course web site.
- We will be using the **OCaml** dialect of ML.

- Install from https://ocaml.org.
- See OCaml Labs : http://www.cl.cam.ac.uk/projects/ocamllabs.
- A side-by-side comparison of SML and OCaml Syntax: http://www.mpi-sws.org/~rossberg/sml-vs-ocaml.html

SML Syntax vs. OCaml Syntax

```
datatype 'a tree =
   Leaf of 'a
  | Node of 'a * ('a tree) * ('a tree)

fun map_tree f (Leaf a) = Leaf (f a)
    | map_tree f (Node (a, left, right)) =
       Node(f a, map_tree f left, map_tree f right)


let val l =
     map_tree (fn a => [a]) [Leaf 17, Leaf 21]
in
     List.rev l
end
```

```
type 'a tree =
   Leaf of 'a
  | Node of 'a * ('a tree) * ('a tree)

let rec map_tree f = function
  | Leaf a -> Leaf (f a)
  | Node (a, left, right)  ->
     Node(f a, map_tree f left, map_tree f right)

let l =
    map_tree (fun a -> [a]) [Leaf 17; Leaf 21]
in
    List.rev l
```

The Shape of this Course

1. Overview
2. Slang Front-end, Slang demo. Code tour.
3. Lexical analysis : application of Theory of Regular Languages and Finite Automata
4. Generating Recursive descent parsers
5. Beyond Recursive Descent Parsing I
6. Beyond Recursive Descent Parsing II
7. High-level "definitional" interpreter (interpreter 0). Make the stack explicit and derive interpreter 2
8. Flatten code into linear array, derive interpreter 3
9. Move complex data from stack into the heap, derive the Jargon Virtual Machine (interpreter 4)
10. More on Jargon VM. Environment management. Static links on stack. Closures.
11. A few program transformations. Tail Recursion Elimination (TRE), Continuation Passing Style (CPS). Defunctionalisation (DFC)
12. CPS+TRE+DFC provides a formal way of understanding how we went from interpreter 0 to interpreter 2. We fill the gap with interpreter 1
13. Assorted topics : compilation units, linking. From Jargon to x86
14. Assorted topics : simple optimisations, OOP object representation
15. Run-time environments, automated memory management ("garbage collection")
16. Bootstrapping a compiler

LECTURE 2
Slang Front End

- **Slang (= Simple LANGuage)**
  - **A subset of L3 from Semantics …**
  - **… with very ugly concrete syntax**
  - **You are invited to experiment with improvements to this concrete syntax.**
- **Slang : concrete syntax, types**
- **Abstract Syntax Trees (ASTs)**
- **The Front End**
- **A short in-lecture demo of slang and a brief tour of the code …**

Clunky Slang Syntax (informal)

```
uop := - | ~                                        (~ is boolean negation)

bop ::= + | - | * | < | = | && | ||

t ::= bool | int | unit | (t) | t * t | t + t | t -> t | t ref

e ::= () | n | true | false | x | (e) | ? |         (? requests an integer
      e bop e |  uop e |                              input from terminal)
      if e then else e end |
      e e | fun (x : t) -> e end |
      let x : t = e in e end |
      let f(x : t) : t = e in e end |
      !e | ref e | e := e | while e do e end |
      begin e; e; … e end |
      (e, e) | snd e | fst e |
      inl t e | inr t e |                           (notice type annotation
      case e of inl(x : t) -> e | inr(x:t) -> e end   on inl and inr constructs)
```

From slang/examples

```
let fib( m : int) : int =
   if m = 0
   then 1
   else if m = 1
         then 1
          else fib (m - 1) +
                fib (m -2)
          end
    end
in
   fib(?)
end
```

```
let gcd( p : int * int) : int =
   let m : int = fst p
   in let  n : int = snd p
   in  if m = n
       then m
       else if m < n
            then gcd(m, n - m)
            else  gcd(m - n, n)
            end
        end
      end
   end
in gcd(?, ?) end
```

The ? requests an integer input from the terminal

Slang Front End

Input file foo.slang

Parse (we use Ocaml versions of LEX and YACC, covered in Lectures 3 --- 6)

Parsed AST (Past.expr)

Static analysis : check types, and context-sensitive rules, resolve overloaded operators

Parsed AST (Past.expr)

Remove "syntactic sugar", file location information, and most type information

Intermediate AST (Ast.expr)

Parsed AST (past.ml)

```
type var = string

type loc = Lexing.position

type type_expr =
   | TEint
   | TEbool
   | TEunit
   | TEref of type_expr
   | TEarrow of type_expr * type_expr
   | TEproduct of type_expr * type_expr
   | TEunion of type_expr * type_expr

type oper = ADD | MUL | SUB | LT |
            AND | OR | EQ | EQB | EQI

type unary_oper = NEG | NOT
```

Locations (loc) are used in generating error messages.

```
type expr =
     | Unit of loc
     | What of loc
     | Var of loc * var
     | Integer of loc * int
     | Boolean of loc * bool
     | UnaryOp of loc * unary_oper * expr
     | Op of loc * expr * oper * expr
     | If of loc * expr * expr * expr
     | Pair of loc * expr * expr
     | Fst of loc * expr
     | Snd of loc * expr
     | Inl of loc * type_expr * expr
     | Inr of loc * type_expr * expr
     | Case of loc * expr * lambda * lambda
     | While of loc * expr * expr
     | Seq of loc * (expr list)
     | Ref of loc * expr
     | Deref of loc * expr
     | Assign of loc * expr * expr
     | Lambda of loc * lambda
     | App of loc * expr * expr
     | Let of loc * var * type_expr * expr * expr
     | LetFun of loc * var * lambda
                 * type_expr * expr
     | LetRecFun of loc * var * lambda
                 * type_expr * expr
```

static.mli, static.ml

```
val infer : (Past.var * Past.type_expr) list -> (Past.expr * Past.type_expr)

val check : Past.expr -> Past.expr   (* infer on empty environment *)
```

- Check type correctness
- Rewrite expressions to resolve EQ to EQI (for integers) or EQB (for bools).
- Only LetFun is returned by parser. Rewrite to LetRecFun when function is actually recursive.

Lesson : while enforcing "context-sensitive rules" we can resolve ambiguities that cannot be specified in context-free grammars.

Internal AST (ast.ml)

```
type var = string

type oper = ADD | MUL | SUB | LT |
            AND | OR | EQB | EQI

type unary_oper = NEG | NOT | READ
```

No locations, types.
No Let, EQ.

Is getting rid of types a bad idea? Perhaps a full answer would be language-dependent…

```
type expr =
     | Unit
     | Var of var
     | Integer of int
     | Boolean of bool
     | UnaryOp of unary_oper * expr
     | Op of expr * oper * expr
     | If of expr * expr * expr
     | Pair of expr * expr
     | Fst of expr
     | Snd of expr
     | Inl of expr
     | Inr of expr
     | Case of expr * lambda * lambda
     | While of expr * expr
     | Seq of (expr list)
     | Ref of expr
     | Deref of expr
     | Assign of expr * expr
     | Lambda of lambda
     | App of expr * expr
     | LetFun of var * lambda * expr
     | LetRecFun of var * lambda * expr

and lambda = var * expr
```

past_to_ast.ml

```
val translate_expr : Past.expr -> Ast.expr
```

```
let x : t = e1 in e2 end
```

```
(fun (x: t) -> e2 end) e1
```

This is done to simplify some of our code.
Is it a good idea? Perhaps not.

Lecture 3, 4, 5, 6
Lexical Analysis and Parsing

1. Theory of Regular Languages and Finite Automata applied to lexical analysis.
2. Context-free grammars
3. The ambiguity problem
4. Generating Recursive descent parsers
5. Beyond Recursive Descent Parsing I
6. Beyond Recursive Descent Parsing II

What problem are we solving?

Translate a sequence of characters

```
if m = 0 then 1 else if m = 1 then 1 else fib (m - 1) +  fib (m -2)
```

into a sequence of **tokens**

```
IF, IDENT “m”, EQUAL, INT 0, THEN, INT 1, ELSE, IF,
IDENT “m”, EQUAL, INT 1, THEN, INT 1, ELSE, IDENT “fib”,
LPAREN, IDENT “m”, SUB, INT 1, RPAREN, ADD,
IDENT “fib”, LPAREN, IDENT “m”, SUB, INT 2, RPAREN
```

implemented with some data type

```
type token =
    | INT of int| IDENT of string | LPAREN | RPAREN
    | ADD | SUB | EQUAL | IF | THEN | ELSE
    | ...
```

Recall from Discrete Mathematics (Part 1A)

Regular expressions (concrete syntax)

over a given alphabet Σ. $\{\varepsilon, \emptyset, |, *, (,)\}$

Let Σ' be the ~~4~~ 6-element set ~~$\{\epsilon, \emptyset, |, *\}$~~ (assumed disjoint from Σ)

$$U = (\Sigma \cup \Sigma')^*$$

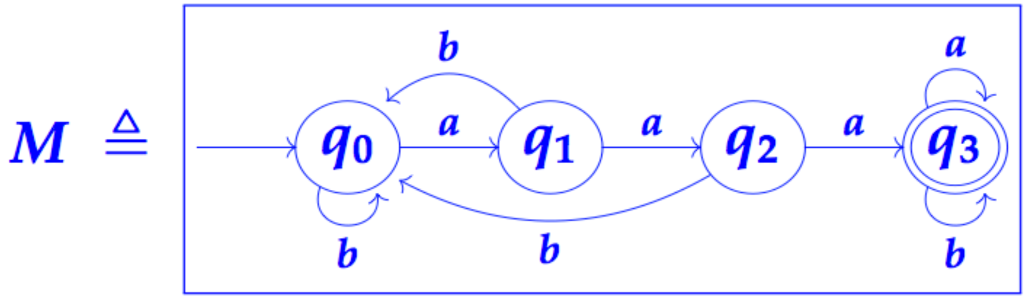
axioms: $\dfrac{}{a}$ $\quad$ $\dfrac{}{\epsilon}$ $\quad$ $\dfrac{}{\emptyset}$

rules: $\dfrac{r}{(r)}$ $\quad$ $\dfrac{r \quad s}{r|s}$ $\quad$ $\dfrac{r \quad s}{rs}$ $\quad$ $\dfrac{r}{r^*}$

(where $a \in \Sigma$ and $r, s \in U$)

Recall from Discrete Mathematics (Part 1A)

Example of a finite automaton



- set of states: $\{q_0, q_1, q_2, q_3\}$
- input alphabet: $\{a, b\}$
- transitions, labelled by input symbols: as indicated by the above directed graph
- start state: q_0
- accepting state(s): q_3

Recall from Discrete Mathematics (Part 1A)

Kleene's Theorem

Definition. A language is **regular** iff it is equal to $L(M)$, the set of strings accepted by some deterministic finite automaton M.

Theorem.

(a) For any regular expression r, the set $L(r)$ of strings matching r is a regular language.

(b) Conversely, every regular language is the form $L(r)$ for some regular expression r.

Traditional Regular Language Problem

Given a regular expression,

$$e$$

and an input string w, determine if $w \in L(e)$

.

Construct a DFA M from *e* and test if it accepts *w*.

Recall construction : regular expression → NFA → DFA

Something closer to the "lexing problem"

Given an ordered list of regular expressions,

$$e_1 \quad e_2 \quad \cdots \quad e_k$$

and an input string w, find a list of pairs

$$(i_1, w_1), (i_2, w_2), \ldots (i_n, w_n)$$

such that

$$1) \quad w = w_1 w_2 \ldots w_n$$

$$2) \quad w_j \in L(e_{i_j})$$

$$3) \quad w_j \in L(e_s) \rightarrow i_j \leq s \quad \text{(priority rule)}$$

.

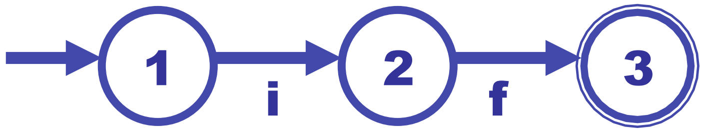
$$4) \quad \forall j : \forall u \in \text{prefix}(w_{j+1} w_{j+2} \cdots w_n) : u \neq \varepsilon$$
$$\rightarrow \forall s : w_j u \notin L(e_s) \quad \text{(longest match)}$$

Why ordered? Is "if" a variable or a keyword? Need priority to resolve ambiguity.

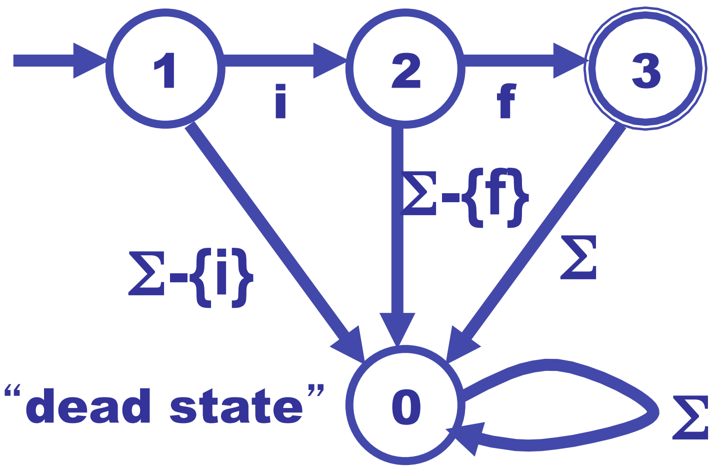
Why longest match? Is "ifif" a variable or two "if" keywords?

Define Tokens with Regular Expressions (Finite Automata)

Keyword: if



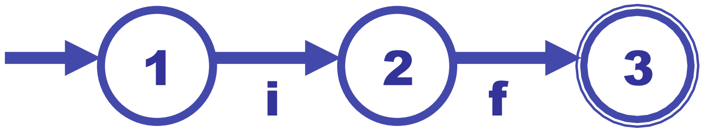
This FA is really shorthand for:



Define Tokens with Regular Expressions (Finite Automata)

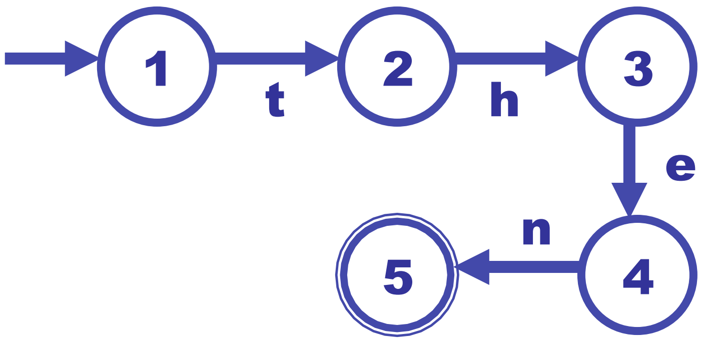
Regular Expression	Finite Automata	Token

Keyword:
if



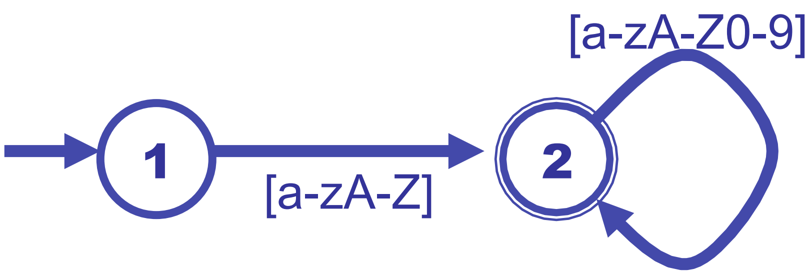
KEY(IF)

Keyword:
then



KEY(then)

Identifier:
[a-zA-Z][a-zA-Z0-9]*

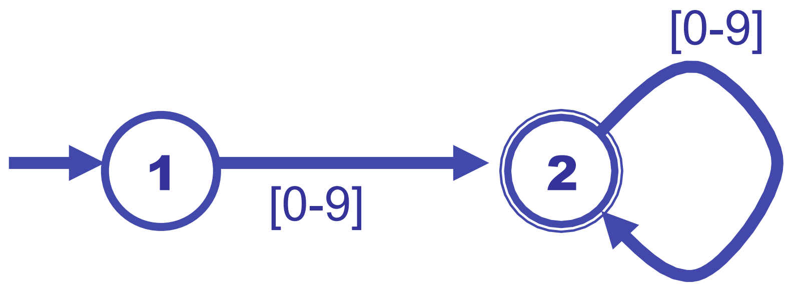


ID(s)

Define Tokens with Regular Expressions (Finite Automata)

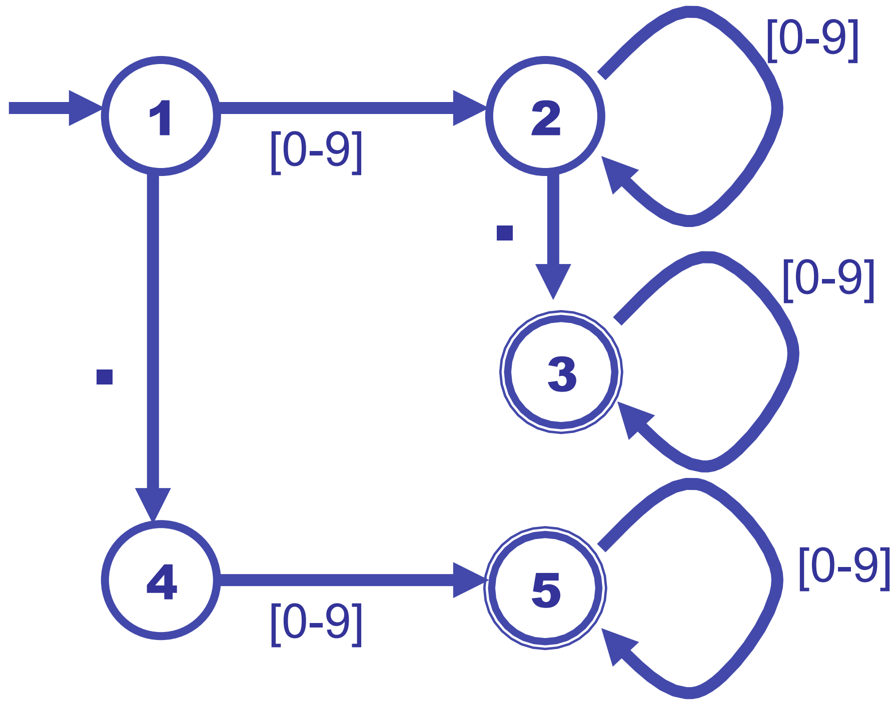
Regular Expression	Finite Automata	Token

number:
[0-9][0-9]*



NUM(n)

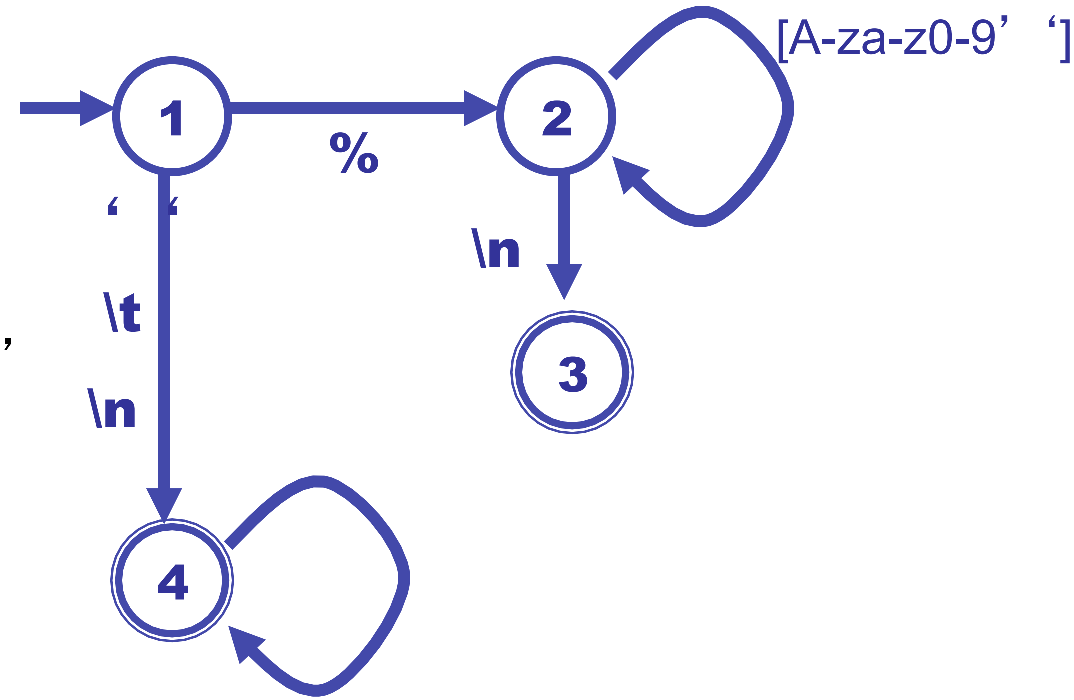
real:
([0-9]+ ‘.’ [0-9]*)
| ([0-9]* ‘.’ [0-9]+)



NUM(n)

No Tokens for "White-Space"

White-space:
(' ' | '\n' | '\t')+
| '%' [A-Za-z0-9' ']+' \n'



Constructing a Lexer

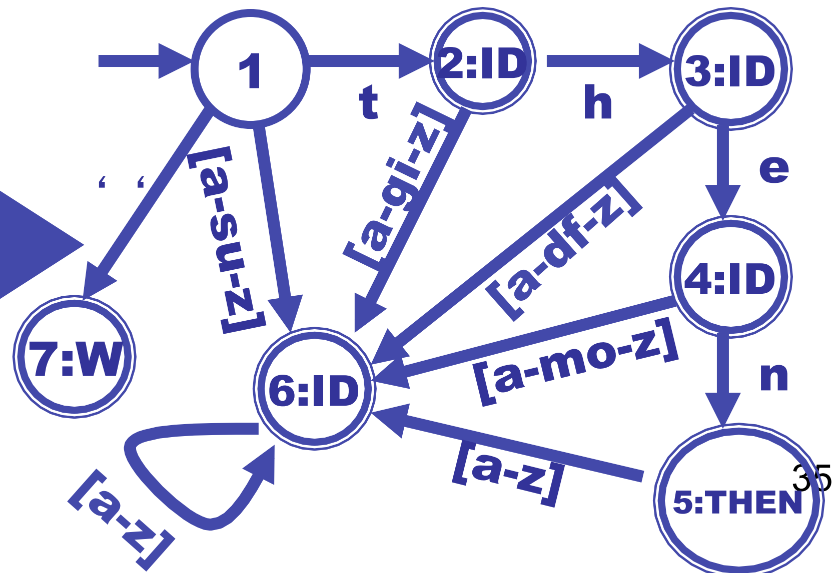
INPUT: an **ordered** list of regular expressions		Construct all corresponding finite automata		Construct a single non-deterministic finite automata		Construct a single deterministic finite automata
e_1		NFA_1				
e_2	→	NFA_2	use priority →	NFA	→	DFA
⋮		⋮				
e_k		NFA_k				

---

(1) Keyword : then

(2) Ident : [a-z][a-z]*

(2) White-space: ‘ ’

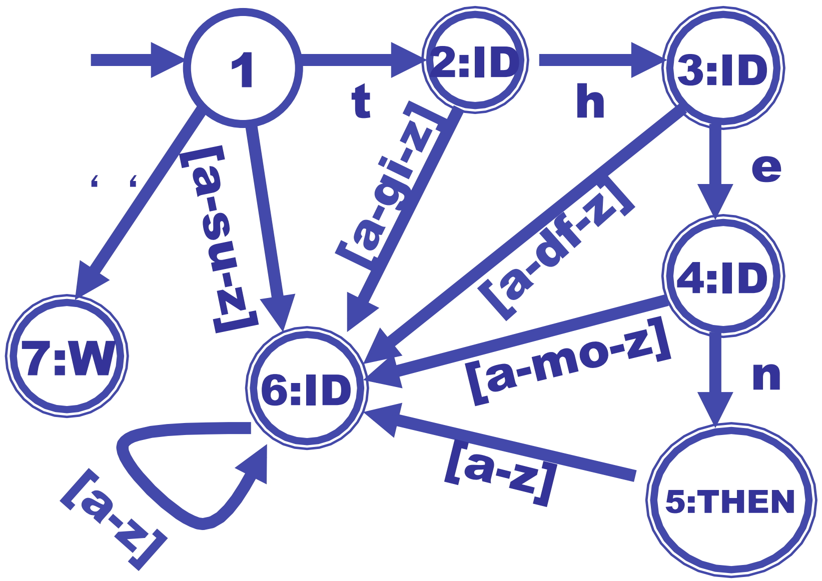


What about longest match?

Start in initial state,
Repeat:

(1) read input until dead state is reached. Emit token associated with last accepting state.

(2) reset state to start state

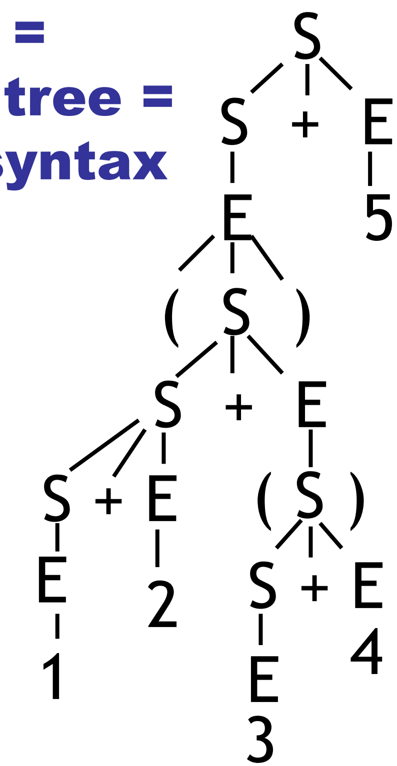


| = current position, $ = EOF

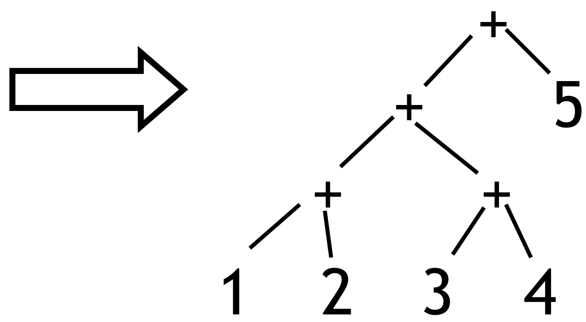
<table>
<tr><th>Input</th><th>current state</th><th>last accepting state</th></tr>
<tr><td>|then thenx$</td><td>1</td><td>0</td></tr>
<tr><td>t|hen thenx$</td><td>2</td><td>2</td></tr>
<tr><td>th|en thenx$</td><td>3</td><td>3</td></tr>
<tr><td>the|n thenx$</td><td>4</td><td>4</td></tr>
<tr><td>then| thenx$</td><td>5</td><td>5</td></tr>
<tr><td>then |thenx$</td><td>0</td><td>5 EMIT KEY(THEN)</td></tr>
<tr><td>then| thenx$</td><td>1</td><td>0 RESET</td></tr>
<tr><td>then |thenx$</td><td>7</td><td>7</td></tr>
<tr><td>then t|henx$</td><td>0</td><td>7 EMIT WHITE(‘ ‘)</td></tr>
<tr><td>then |thenx$</td><td>1</td><td>0 RESET</td></tr>
<tr><td>then t|henx$</td><td>2</td><td>2</td></tr>
<tr><td>then th|enx$</td><td>3</td><td>3</td></tr>
<tr><td>then the|nx$</td><td>4</td><td>4</td></tr>
<tr><td>then then|x$</td><td>5</td><td>5</td></tr>
<tr><td>then thenx|$</td><td>6</td><td>6</td></tr>
<tr><td>then thenx$|</td><td>0</td><td>6 EMIT ID(thenx)</td></tr>
</table>

Concrete vs. Abstract Syntax Trees

parse tree = derivation tree = concrete syntax tree



Abstract Syntax Tree (AST)



An AST contains only the information needed to generate an intermediate representation

Normally a compiler constructs the concrete syntax tree only implicitly (in the parsing process) and explicitly constructs an AST.

On to Context Free Grammars (CFGs)

E ::= ID

E ::= NUM

E ::= E * E

E ::= E / E

E ::= E + E

E ::= E – E

E ::= (E)

E is a *non-terminal symbol*

ID and NUM are *lexical classes*

*, (,), +, and – are *terminal symbols*.

E ::= E + E is called a *production rule*.

Usually will write this way

E ::= ID | NUM | E * E | E / E | E + E | E – E | (E)

CFG Derivations

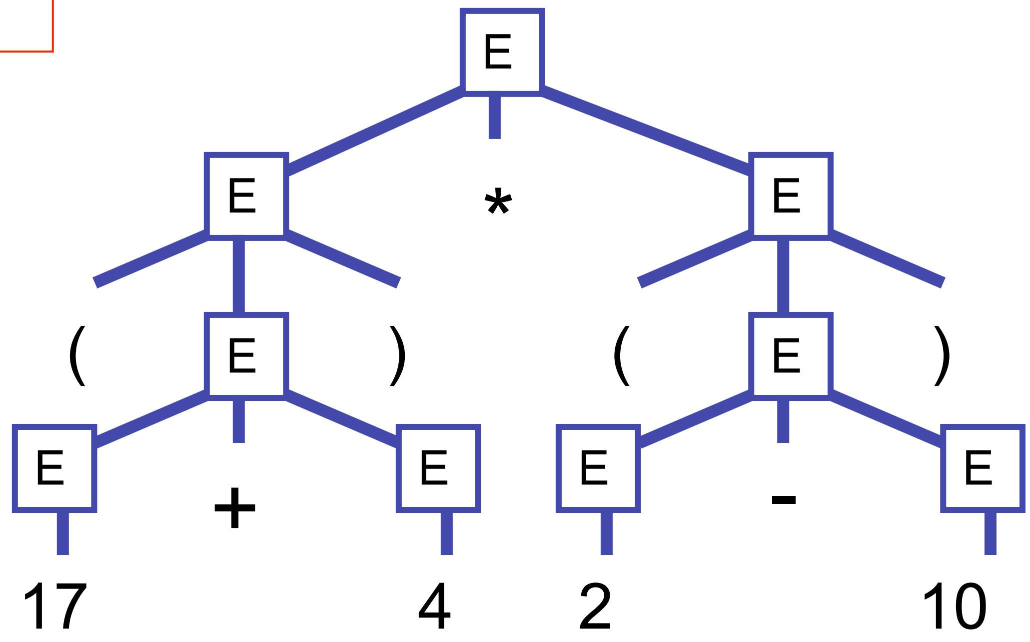
(G1) E ::= ID | NUM | ID | E * E | E / E | E + E | E – E | (E)

Rightmost derivation

E → E * E
→ E * (E)
→ E * (E – E)
→ E * (E – 10)
→ E * (2 – 10)
→ (E) * (2 – 10)
→ (E + E) * (2 – 10)
→ (E + 4) * (2 – E)
→ (17 + 4) * (2 – 10)

Leftmost derivation

E → E * E
→ (**E**) * E
→ (**E** + E) * E
→ (17 + E) * E
→ (17 + 4) * E
→ (17 + 4) * (E)
→ (17 + 4) * (**E** – E)
→ (17 + 4) * (2 – **E**)
→ (17 + 4) * (2 – 10)



The Derivation Tree for
(17 + 4) * (2 – 10)

More formally, ...

- **A CFG is a quadruple G = (N, T, R, S) where**
  - **N is the set of *non-terminal symbols***
  - **T is the set of *terminal symbols* (N and T disjoint)**
  - **$S \in N$ is the *start symbol***
  - **$R \subseteq N \times (N \cup T)^*$ is a set of rules**
- **Example: The grammar of nested parentheses G = (N, T, R, S) where**
  - **N = {S}**
  - **T ={ (,) }**
  - **R ={ (S, (S)) , (S, SS), (S,) }**

We will normally write R as

S ::= (S) | SS |

Derivations, more formally…

- **Start from start symbol (*S*)**
- **Productions are used to derive a sequence of tokens from the start symbol**
- **For arbitrary strings α, β and γ comprised of both terminal and non-terminal symbols,**
  and a production $A \rightarrow \beta$,
  a single step of derivation is
  $\alpha A\gamma \Rightarrow \alpha\beta\gamma$
  - ***i.e.*, substitute β for an occurrence of A**
- **$\alpha \Rightarrow^* \beta$ means that b can be derived from a in 0 or more single steps**
- **$\alpha \Rightarrow^+ \beta$ means that b can be derived from a in 1 or more single steps**

L(G) = The Language Generated by Grammar G

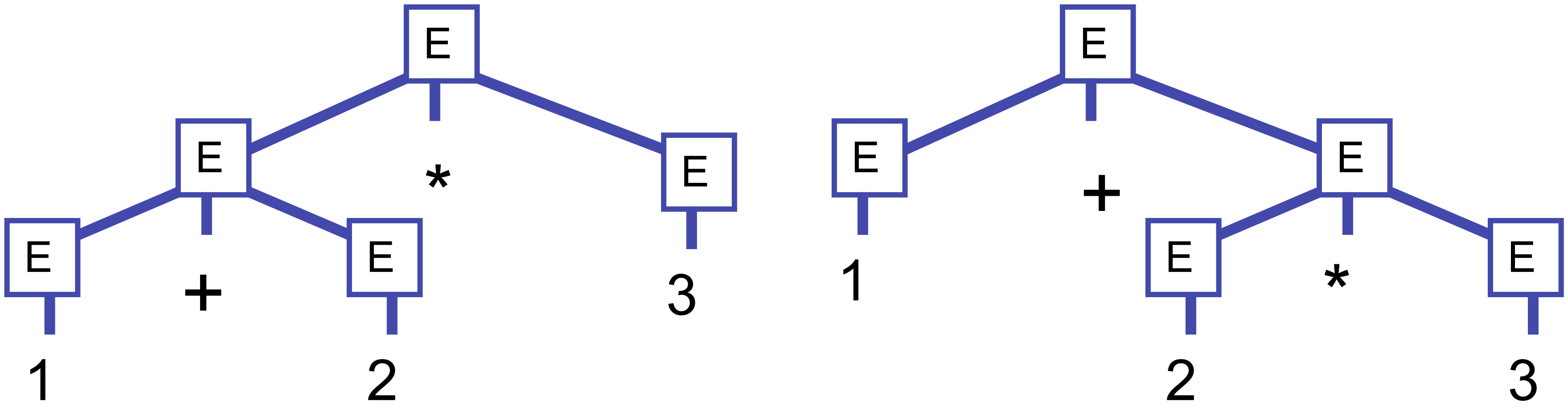
The language generated by G is the set of all terminal strings derivable from the start symbol S:

$$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$

For any subset W of T*, if there exists a CFG G such that L(G) = W, then W is called a Context-Free Language (CFL) over T.

Ambiguity

(G1) E ::= ID | NUM | ID | E * E | E / E | E + E | E – E | (E)



Both derivation trees correspond to the string

1 + 2 * 3

This type of ambiguity will cause problems when we try to go from strings to derivation trees!

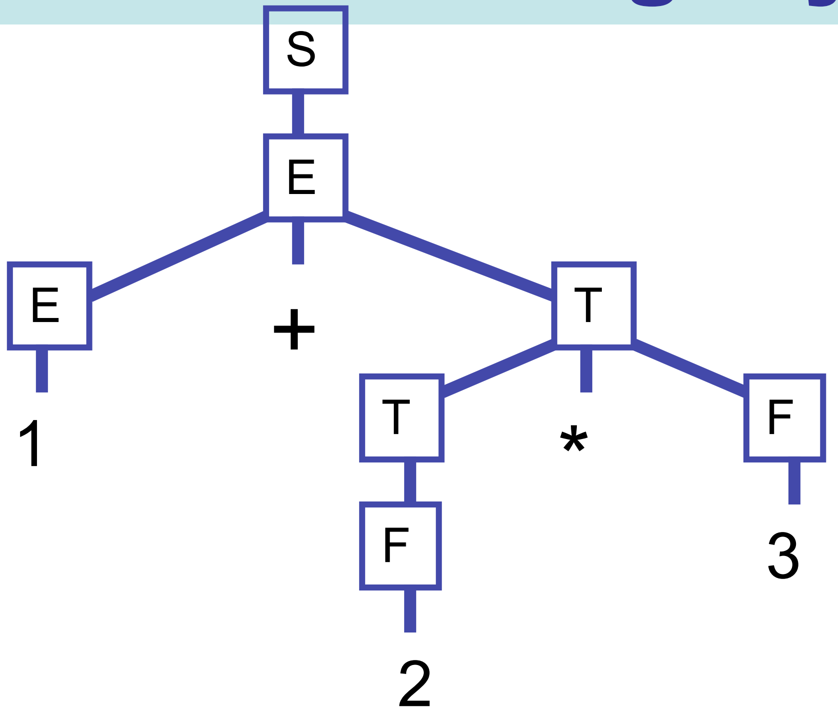
Problem: Generation vs. Parsing

- **Context-Free Grammars (CFGs) describe how to to *generate***
- ***Parsing* is the inverse of generation,**
  - **Given an input string, is it in the language generated by a CFG?**
  - **If so, construct a derivation tree (normally called a *parse tree*).**
  - **Ambiguity is a big problem**

Note : recent work on Parsing Expression Grammars (PEGs) represents an attempt to develop a formalism that describes parsing directly. This is beyond the scope of these lectures …

We can often modify the grammar in order to eliminate ambiguity

```
(G2)
 S :: = E$               (start, $ = EOF)

 E ::= E + T             (expressions)
     |  E – T
     |  T

T ::= T * F              (terms)
     |  T / F
     |  F

F ::= NUM                (factors)
     | ID
     | ( E )
```



This is the *unique* derivation tree for the string

1 + 2 * 3$

Note: L(G1) = L(G2).
Can you prove it?

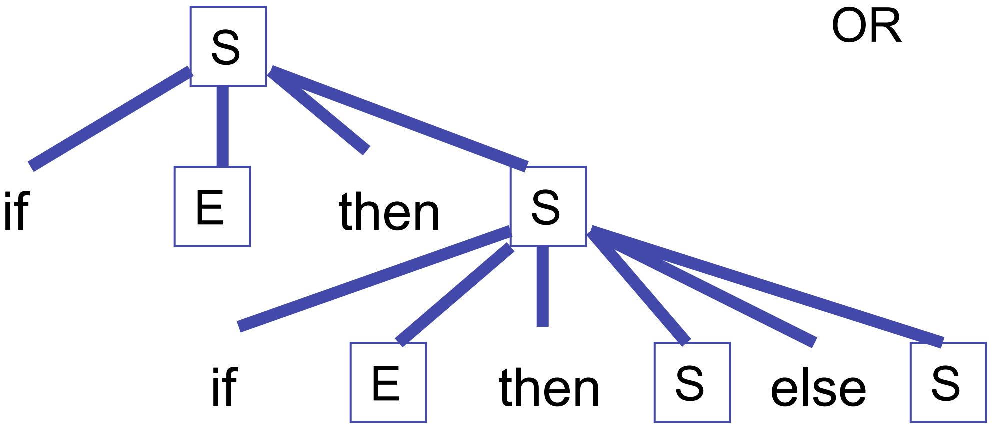
Famously Ambiguous

(G3) S ::= if E then S else S | if E then S | blah-blah

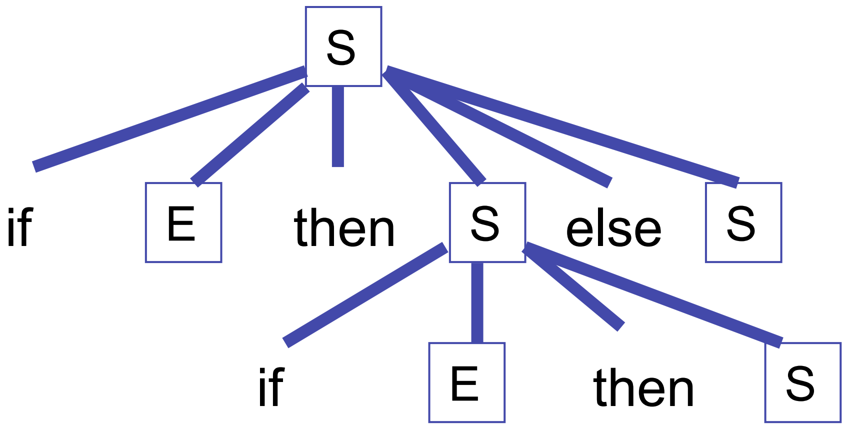
What does

if e1 then if e2 then s1 else s3

mean?



OR



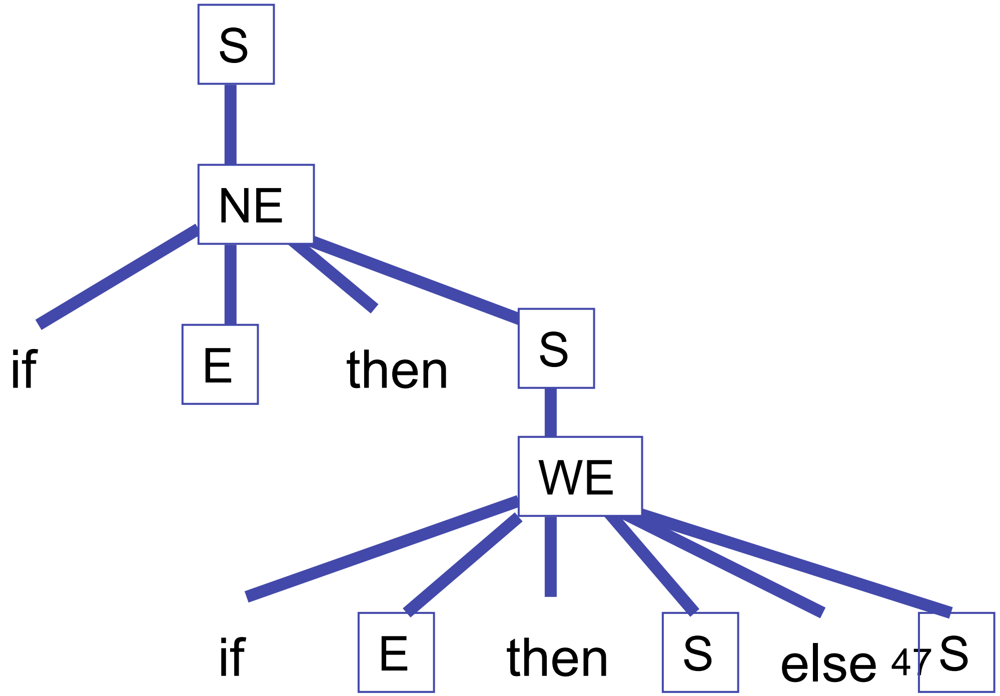
Rewrite?

(G4)
S ::= WE | NE
WE ::= **if** E **then** WE **else** WE | **blah-blah**
NE ::= **if** E **then** S
| **if** E **then** WE **else** NE

Now,

if e1 then if e2 then s1 else s3

has a unique derivation.



Note: L(G3) = L(G4).
Can you prove it?

Fun Fun Facts

See Hopcroft and Ullman, "Introduction to Automata Theory, Languages, and Computation"

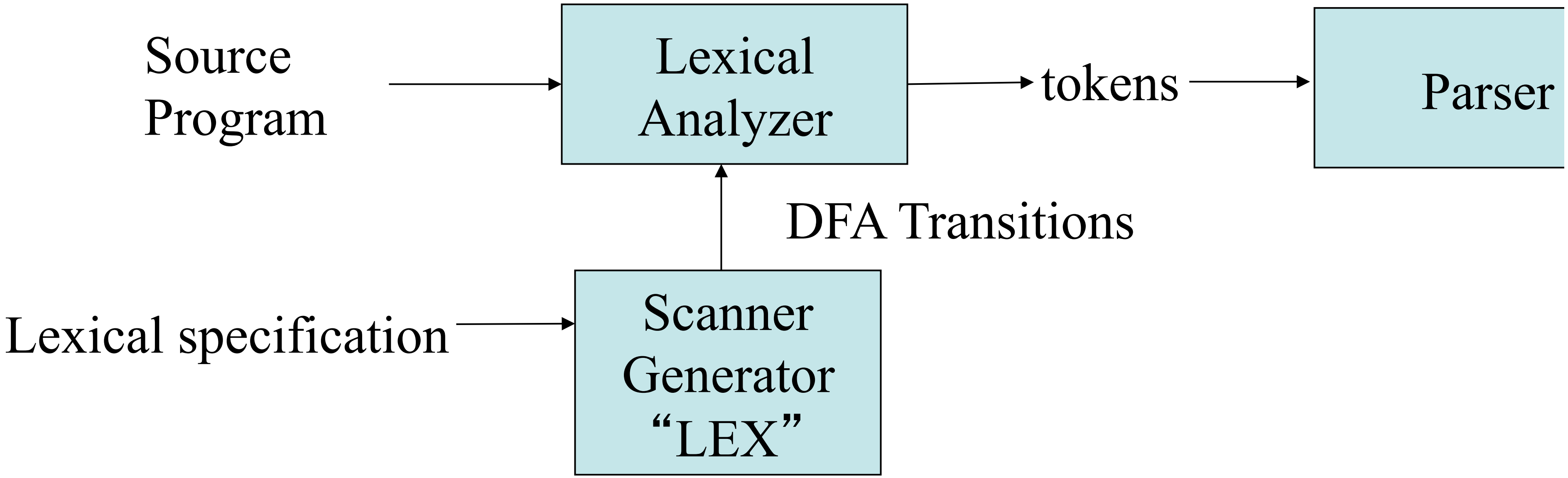
(1) Some context free languages are *inherently ambiguous* --- every context-free grammar will be ambiguous. For example:

$$L = \{a^n b^n c^m d^m \mid m \geq 1, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m \geq 1, n \geq 1\}$$

(2) Checking for ambiguity in an arbitrary context-free grammar is not decidable! Ouch!

(3) Given two grammars G1 and G2, checking L(G1) = L(G2) is not decidable! Ouch!

Generating Lexical Analyzers



The idea : use regular expressions as the basis of a lexical specification. The core of the lexical analyzer is then a deterministic finite automata (DFA)

Predictive (Recursive Descent) Parsing
Can we automate this?

(G5)

S :: = if E then S else S
  | begin S L
  | print E

E ::= NUM = NUM

L ::= end
  | ; S L

```
int tok = getToken();

void advance() {tok = getToken();}
void eat (int t) {if (tok == t) advance(); else error();}

void S() {switch(tok) {
      case IF:    eat(IF); E(); eat(THEN);
                  S(); eat(ELSE); S(); break;
      case BEGIN: eat(BEGIN); S(); L(); break;
      case PRINT: eat(PRINT); E(); break;
      default: error();
     }}

void L() {switch(tok) {
      case END:  eat(END); break;
      case SEMI: eat(SEMI); S(); L(); break;
      default: error();
     }}

void E() {eat(NUM) ; eat(EQ); eat(NUM); }
```

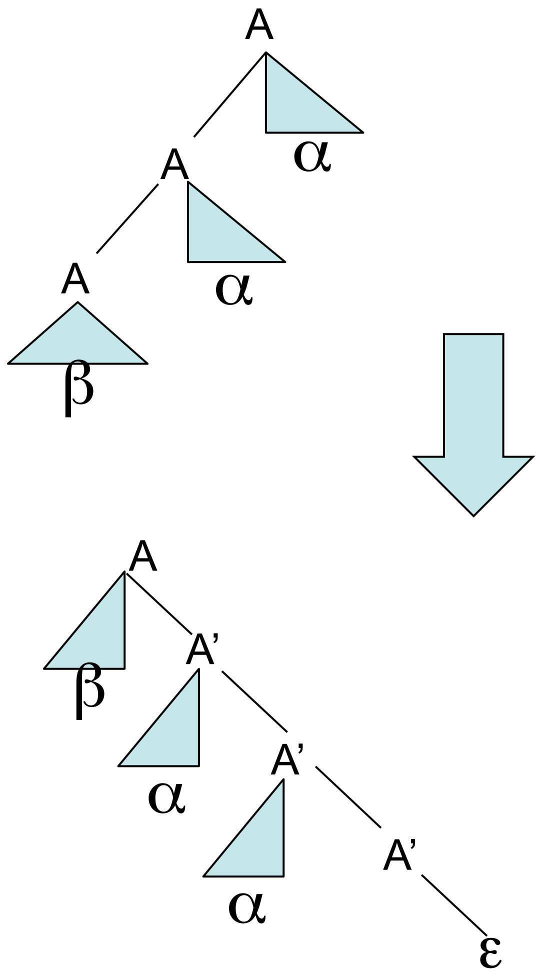
Parse corresponds to a left-most derivation constructed in a “top-down” manner

From Andrew Appel, “Modern Compiler Implementation in Java” page 46

Eliminate Left-Recursion

Immediate left-recursion

$$A ::= A\alpha 1 \mid A\alpha 2 \mid \ldots \mid A\alpha k \mid \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$$



$$A ::= \beta 1\, A' \mid \beta 2\, A' \mid \ldots \mid \beta n\, A'$$

$$A' ::= \alpha 1\, A' \mid \alpha 2\, A' \mid \ldots \mid \alpha k\, A' \mid \varepsilon$$

For eliminating left-recursion in general, see Aho and Ullman.

Eliminating Left Recursion

(G2)
S :: = E$

E ::= E + T
| E – T
| T

T ::= T * F
| T / F
| F

F ::= NUM
| ID
| (E)

Note that
E ::= T and
E ::= E + T
will cause problems
since FIRST(T) will be included
in FIRST(E + T) ---- so how can
we decide which poduction
To use based on next token?

Solution: eliminate "left recursion"!

E ::= T E'

E' ::= + T E'
|

Eliminate left recursion

(G6)
S :: = E$

E ::= T E'

E' ::= + T E'
| – T E'
|

T ::= F T'

T' ::= * F T'
| / F T'
|

F ::= NUM
| ID
| (E)

FIRST and FOLLOW

For each non-terminal X we need to compute

FIRST[X] = the set of terminal symbols that can begin strings derived from X

FOLLOW[X] = the set of terminal symbols that can immediately follow X in some derivation

nullable[X] = true of X can derive the empty string, false otherwise

nullable[Z] = false, for Z in T

nullable[Y1 Y2 … Yk] = nullable[Y1] and … nullable[Yk], for Y(i) in N union T.

FIRST[Z] = {Z}, for Z in T

FIRST[X Y1 Y2 … Yk] = FIRST[X] if not nullable[X]

FIRST[X Y1 Y2 … Yk] =FIRST[X] union FIRST[Y1 … Yk] otherwise

Computing First, Follow, and nullable

```
For each terminal symbol Z
   FIRST[Z] := {Z};
   nullable[Z] := false;

For each non-terminal symbol X
  FIRST[X] := FOLLOW[X] := {};
  nullable[X] := false;

repeat
   for each production X → Y1 Y2 … Yk
      if Y1, … Yk are all nullable, or k = 0
         then nullable[X] := true
      for each i from 1 to k, each j from i + l to k
         if Y1 … Y(i-1) are all nullable or i = 1
            then  FIRST[X] := FIRST[X] union FIRST[Y(i)]
         if Y(i+1) … Yk are all nullable or if i = k
            then FOLLOW[Y(i)] := FOLLOW[Y(i)] union FOLLOW[X]
         if Y(i+1) … Y(j-1) are all nullable or i+1 = j
            then FOLLOW[Y(i)] := FOLLOW[Y(i)] union FIRST[Y(j)]
until there is no change
```

First, Follow, nullable table for G6

	Nullable	FIRST	FOLLOW
S	False	{ (, ID, NUM }	{}
E	False	{ (, ID, NUM }	{), $ }
E’	True	{ +, - }	{), $ }
T	False	{ (, ID, NUM }	{), +, -, $ }
T’	True	{ *, / }	{), +, -, $ }
F	False	{ (, ID, NUM }	{), *, /, +, -, $ }

(G6)
S :: = E$

E ::= T E’

E’ ::= + T E’
| – T E’
|

T ::= F T’

T’ ::= * F T’
| / F T’
|

F ::= NUM
| ID
| (E)

Predictive Parsing Table for G6

Table[X, T] = Set of productions

X ::= Y1…Yk in Table[X, T]
if T in FIRST[Y1 … Yk]
or if (T in FOLLOW[X] and nullable[Y1 … Yk])

NOTE: this could lead to more than one entry! If so, out of luck --- can't do recursive descent parsing!

	+	*	(	)	ID	NUM	$
S			S ::= E$		S ::= E$	S ::= E$	
E			E ::= T E'		E ::= T E'	E ::= T E'	
E'	E' ::= + T E'			E' ::=			E' ::=
T			T ::= F T'		T ::= F T'	T ::= F T'	
T'	T' ::=	T' ::= * F T'		T' ::=			T' ::=
F			F ::= (E)		F ::= ID	F ::= NUM	

(entries for /, - are similar…)

Left-most derivation is constructed by recursive descent

(G6)
S :: = E$

E ::= T E'

E' ::= + T E'
| – T E'
|

T ::= F T'

T' ::= * F T'
| / F T'
|

F ::= NUM
| ID
| (E)

Left-most derivation

S → E$
→ T E' $
→ F T' E' $
→ (E) T' E' $
→ (T E') T' E' $
→ (F T' E') T' E' $
→ (17 T' E') T' E' $
→ (17 E') T' E' $
→ (17 + T E') T' E' $
→ (17 + F T' E') T' E' $
→ (17 + 4 T' E') T' E' $
→ (17 + 4 E') T' E' $
→ (17 + 4) T' E' $
→ (17 + 4) * F T' E' $
→ ...
→ ...
→ (17 + 4) * (2 – 10) T' E' $
→ (17 + 4) * (2 – 10) E' $
→ (17 + 4) * (2 – 10)

call S()
on '(' call E()
on '(' call T()
.l..
...

As a stack machine

S → E$
→ T E' $
→ F T' E' $
→ (E) T' E' $
→ (T E') T' E' $
→ (F T' E') T' E' $
→ (17 T' E') T' E' $
→ (17 E') T' E' $
→ (17 + T E') T' E' $
→ (17 + F T' E') T' E' $
→ (17 + 4 T' E') T' E' $
→ (17 + 4 E') T' E' $
→ (17 + 4) T' E' $
→ (17 + 4) * F T' E' $
→ …
→ …
→ (17 + 4) * (2 – 10) T' E' $
→ (17 + 4) * (2 – 10) E' $
→ (17 + 4) * (2 – 10)

	E$
	T E' $
	F T' E' $
(	E) T' E' $
(	T E') T' E' $
(	F T' E') T' E' $
(17	T' E') T' E' $
(17	E') T' E' $
(17 +	T E') T' E' $
(17 +	F T' E') T' E' $
(17 + 4	T' E') T' E' $
(17 + 4	E') T' E' $
(17 + 4)	T' E' $
(17 + 4) *	F T' E' $
…	
…	
(17 + 4) * (2 – 10)	T' E' $
(17 + 4) * (2 – 10)	E' $
(17 + 4) * (2 – 10)	

But wait! What if there are conflicts in the predictive parsing table?

(G7)

S ::= d | X Y S

Y ::= c |

X ::= Y | a

	Nullable	FIRST	FOLLOW
S	false	{ c,d ,a}	{ }
Y	true	{ c }	{ c,d,a }
X	true	{ c,a }	{ c, a,d }

The resulting "predictive" table is not so predictive....

	a	c	d
S	{ S ::= X Y S }	{ S ::= X Y S }	{ S ::= X Y S, S ::= d }
Y	{ Y ::= }	{ Y ::= , Y ::= c}	{ Y ::= }
X	{ X ::= a, X ::= Y }	{ X ::= Y }	{ X ::= Y }

LL(1), LL(k), LR(0), LR(1), ...

- LL(k) : (L)eft-to-right parse, (L)eft-most derivation, k-symbol lookahead. Based on looking at the next k tokens, an LL(k) parser must *predict* the next production. We have been looking at LL(1).
- LR(k) : (L)eft-to-right parse, (R)ight-most derivation, k-symbol lookahead. Postpone production selection until *the entire* right-hand-side has been seen (and as many as k symbols beyond).
- LALR(1) : A special subclass of LR(1).

Example

(G8)

S :: = S ; S | ID = E | print (L)

E ::= ID | NUM | E + E | (S, E)

L ::= E | L, E

To be consistent, I should write the following, but I won't…

(G8)

S :: = S SEMI S | ID EQUAL E | PRINT LPAREN L RPAREN

E ::= ID | NUM | E PLUS E | LPAREN S COMMA E RPAREN

L ::= E | L COMMA E

A right-most derivation ...

(G8)

S ::= S ; S
    | ID = E
    | print (L)

E ::= ID
    | NUM
    | E + E
    | (S, E)

L ::= E
    | L, E

S
→ S ; S
→ S ; ID = E
→ S ; ID = E + E
→ S ; ID = E + (S, E)
→ S ; ID = E + (S, ID)
→ S ; ID = E + (S, d)
→ S ; ID = E + (ID = E, d)
→ S ; ID = E + (ID = E + E, d)
→ S ; ID = E + (ID = E + NUM, d)
→ S ; ID = E + (ID = E + 6, d)
→ S ; ID = E + (ID = NUM + 6, d)
→ S ; ID = E + (ID = 5 + 6, d)
→ S ; ID = E + (d = 5 + 6, d)
→ S ; ID = ID + (d = 5 + 6, d)
→ S ; ID = c + (d = 5 + 6, d)
→ S ; b = c + (d = 5 + 6, d)
→ ID = E ; b = c + (d = 5 + 6, d)
→ ID = NUM ; b = c + (d = 5 + 6, d)
→ ID = 7 ; b = c + (d = 5 + 6, d)
→ a = 7 ; b = c + (d = 5 + 6, d)

Now, turn it upside down …

→ a = 7 ; b = c + (d = 5 + 6, d)
→ ID = 7 ; b = c + (d = 5 + 6, d)
→ ID = NUM; b = c + (d = 5 + 6, d)
→ ID = E ; b = c + (d = 5 + 6, d)
→ S ; b = c + (d = 5 + 6, d)
→ S ; ID = c + (d = 5 + 6, d)
→ S ; ID = ID + (d = 5 + 6, d)
→ S ; ID = E + (d = 5 + 6, d)
→ S ; ID = E + (ID = 5 + 6, d)
→ S ; ID = E + (ID = NUM + 6, d)
→ S ; ID = E + (ID = E + 6, d)
→ S ; ID = E + (ID = E + NUM, d)
→ S ; ID = E + (ID = E + E, d)
→ S ; ID = E + (ID = E, d)
→ S ; ID = E + (S, d)
→ S ; ID = E + (S, ID)
→ S ; ID = E + (S, E)
→ S ; ID = E + E
→ S ; ID = E
→ S ; S
S

Now, slice it down the middle...

	a = 7 ; b = c + (d = 5 + 6, d)
ID	= 7 ; b = c + (d = 5 + 6, d)
ID = NUM	; b = c + (d = 5 + 6, d)
ID = E	; b = c + (d = 5 + 6, d)
S	; b = c + (d = 5 + 6, d)
S ; ID	= c + (d = 5 + 6, d)
S ; ID = ID	+ (d = 5 + 6, d)
S ; ID = E	+ (d = 5 + 6, d)
S ; ID = E + (ID	= 5 + 6, d)
S ; ID = E + (ID = NUM	+ 6, d)
S ; ID = E + (ID = E	+ 6, d)
S ; ID = E + (ID = E + NUM	, d)
S ; ID = E + (ID = E + E	, d)
S ; ID = E + (ID = E	, d)
S ; ID = E + (S	, d)
S ; ID = E + (S, ID	)
S ; ID = E + (S, E)	
S ; ID = E + E	
S ; ID = E	
S ; S	
S	

A stack of terminals and non-terminals

The rest of the input string

Now, add some actions. s = SHIFT, r = REDUCE

	a = 7 ; b = c + (d = 5 + 6, d)	s
ID	= 7 ; b = c + (d = 5 + 6, d)	s, s
ID = NUM	; b = c + (d = 5 + 6, d)	r E ::= NUM
ID = E	; b = c + (d = 5 + 6, d)	r S ::= ID = E
S	; b = c + (d = 5 + 6, d)	s, s
S ; ID	= c + (d = 5 + 6, d)	s, s
S ; ID = ID	+ (d = 5 + 6, d)	r E ::= ID
S ; ID = E	+ (d = 5 + 6, d)	s, s, s
S ; ID = E + (ID	= 5 + 6, d)	s, s
S ; ID = E + (ID = NUM	+ 6, d)	r E ::= NUM
S ; ID = E + (ID = E	+ 6, d)	s, s
S ; ID = E + (ID = E + NUM	, d)	r E ::= NUM
S ; ID = E + (ID = E + E	, d)	r E ::= E+E, s, s
S ; ID = E + (ID = E	, d)	r S ::= ID = E
S ; ID = E + (S	)	R E::= ID
S ; ID = E + (S, ID	)	s, r E ::= (S, E)
S ; ID = E + (S, E)		r E ::= E + E
S ; ID = E + E		r S ::= ID = E
S ; ID = E		r S ::= S ; S
S ; S		
S		

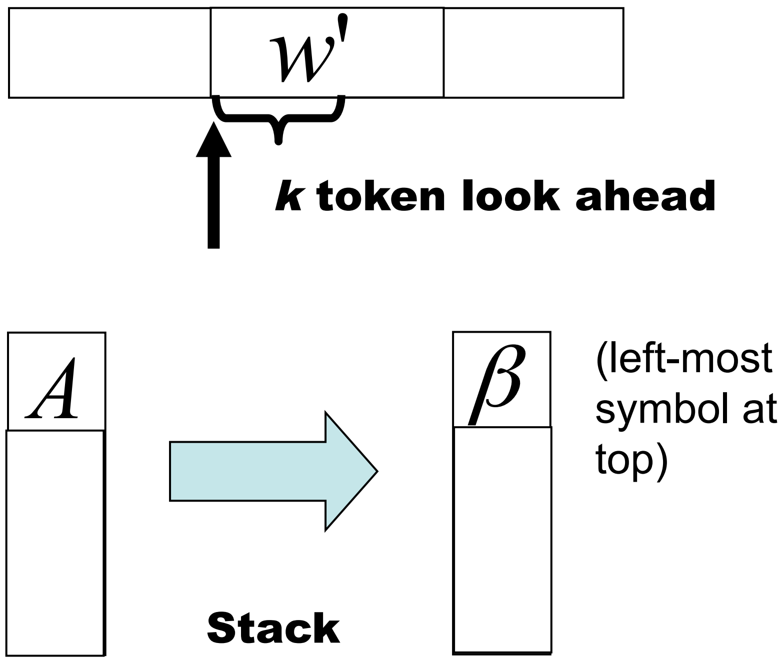
SHIFT = LEX + move token to stack

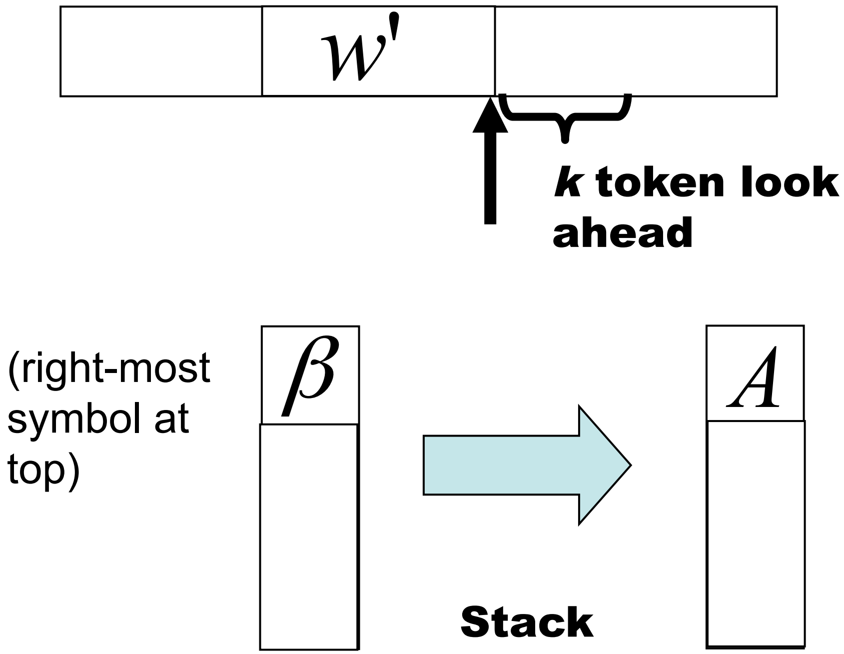
ACTIONS

LL(k) vs. LR(k) reductions

$$A \rightarrow \beta \Rightarrow^{*} w' \qquad (\beta \in (T \cup N)^{*}, \quad w' \in T^{*})$$

$LL(k)$	$LR(k)$





The language of this Stack IS REGULAR!

Q: How do we know when to shift and when to reduce? A: Build a FSA from LR(0) Items!

(G10)

S ::= A $

A ::= (A)
| ( )

If

X ::= $\alpha\beta$

is a production, then

X ::= $\alpha \bullet \beta$

is an LR(0) item.

S ::= • A $
S ::= A • $
A ::= • (A)
A ::= (• A)
A ::= (A •)
A ::= (A) •
A ::= • ( )
A ::= (•)
A ::= ( ) •

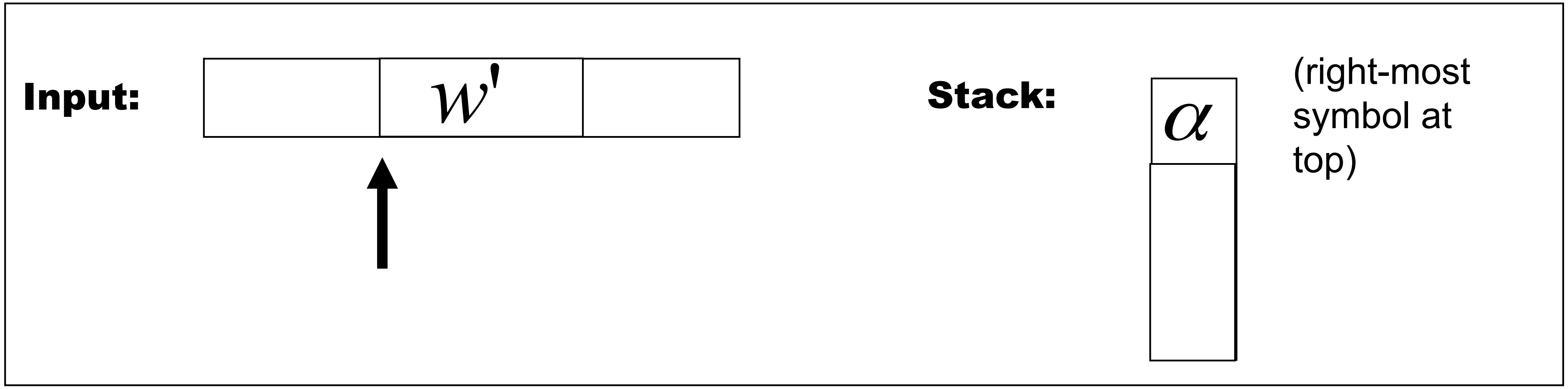
LR(0) items indicate what is on the stack (to the left of the •) and what is still in the input stream (to the right of the •)

LR(k) states (non-deterministic)

The state

$$(A \rightarrow \alpha \bullet \beta,\ a_1 a_2 \cdots a_k)$$

should represent this situation:



with

$$\beta a_1 a_2 \cdots a_k \Rightarrow^{*} w'$$

Key idea behind LR(0) items

- If the "current state" contains the item
  A ::= α • c β and the current symbol in the input buffer is c
  - the state prompts parser to perform a shift action
  - next state will contain A ::= α c • β
- If the "state" contains the item **A ::=** α •
  - the state prompts parser to perform a reduce action
- If the "state" contains the item S ::= α • $
  and the input buffer is empty
  - the state prompts parser to accept
- But How about A ::= α • X β where X is a nonterminal?

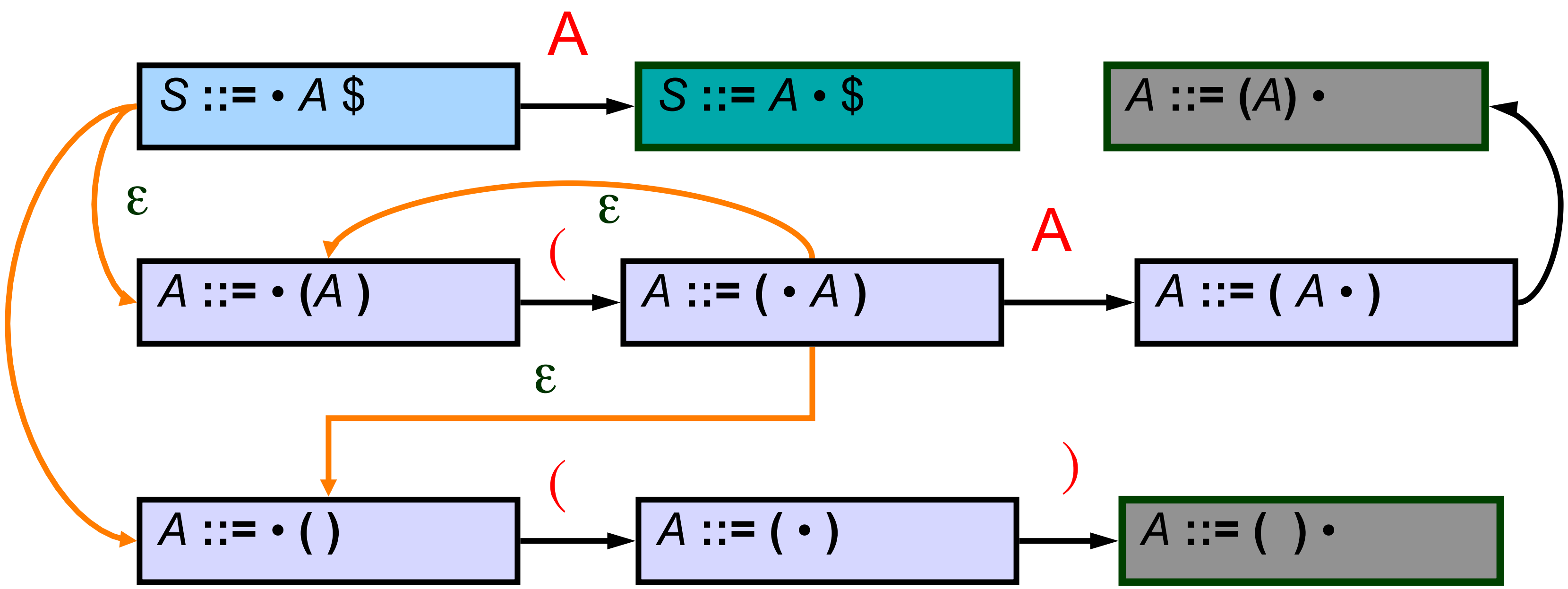
The NFA for LR(0) items

- The transition of LR(0) items can be represented by an NFA, in which
  - 1. each LR(0) item is a state,
  - 2. there is a transition from item A ::= α • c β

    to item A ::= αc • β with label c, where c is a terminal symbol
  - 3. there is an ε-transition from item A ::= α • X β to X ::= • γ, where X is a non-terminal
  - 4. S ::= • A $ is the start state
  - 5. A ::= α • is a final state.

Example NFA for Items

S ::= • A $	**S ::= A • $**	**A ::= • (A)**
A ::= (• A)	**A ::= (A •)**	**A ::= (A) •**
A ::= • ( )	**A ::= (•)**	**A ::= ( ) •**



The DFA from LR(0) items

- After the NFA for LR(0) is constructed, the resulting DFA for LR(0) parsing can be obtained by the usual NFA2DFA construction.

- we thus require
  - ε-closure (I)
  - move(S, a)

Fixed Point Algorithm for Closure(**I**)

- Every item in **I** is also an item in Closure(**I**)
- If A ::= α • B β is in Closure(**I**) and B ::= • γ is an item, then add B ::= • γ to Closure(**I**)
- Repeat until no more new items can be added to Closure(**I**)

Examples of Closure

Closure({A ::= (• A)}) =

$$\left\{\begin{array}{l} A ::= (\ \bullet A) \\ A ::= \bullet\ (A) \\ A ::= \bullet\ (\ \ ) \end{array}\right\}$$

- closure({S ::= • ***A*** **$**})

$$\left\{\begin{array}{l} S ::= \bullet\ \boldsymbol{A}\ \$ \\ \boldsymbol{A} ::= \bullet\ (\boldsymbol{A}) \\ \boldsymbol{A} ::= \bullet\ (\ \ ) \end{array}\right\}$$

S ::= • A $
S ::= A • $
A ::= • (A)
A ::= (• A)
A ::= (A •)
A ::= (A) •
A ::= • ( )
A ::= (•)
A ::= ( ) •

Goto() of a set of items

- Goto finds the new state after consuming a grammar symbol while in the current state

- Algorithm for *Goto*(I, X)
  where I is a set of items
  and X is a non-terminal

$$\text{Goto}(I, X) = \text{Closure}(\ \{\ A ::= \ \alpha\ X \bullet \beta \ \big|\ A ::= \ \alpha \bullet X\ \beta \text{ in } I\ \}\ )$$

- goto is the new set obtained by "moving the dot" over X

Examples of Goto

- Goto ({*A* ::= •(*A*)}, ( )

$$\left\{\begin{array}{l} A ::= (\ \bullet A) \\ A ::= \bullet\ (A) \\ A ::= \bullet\ (\ \ ) \end{array}\right\}$$

- Goto ({*A* ::= (• *A*)}, *A*)

$$\left\{ A ::= (A \bullet\ \ ) \right\}$$

S ::= • A $

S ::= A • $

A ::= • (A)

A ::= (• A)

A ::= (A •)

A ::= (A) •

A ::= • ( )

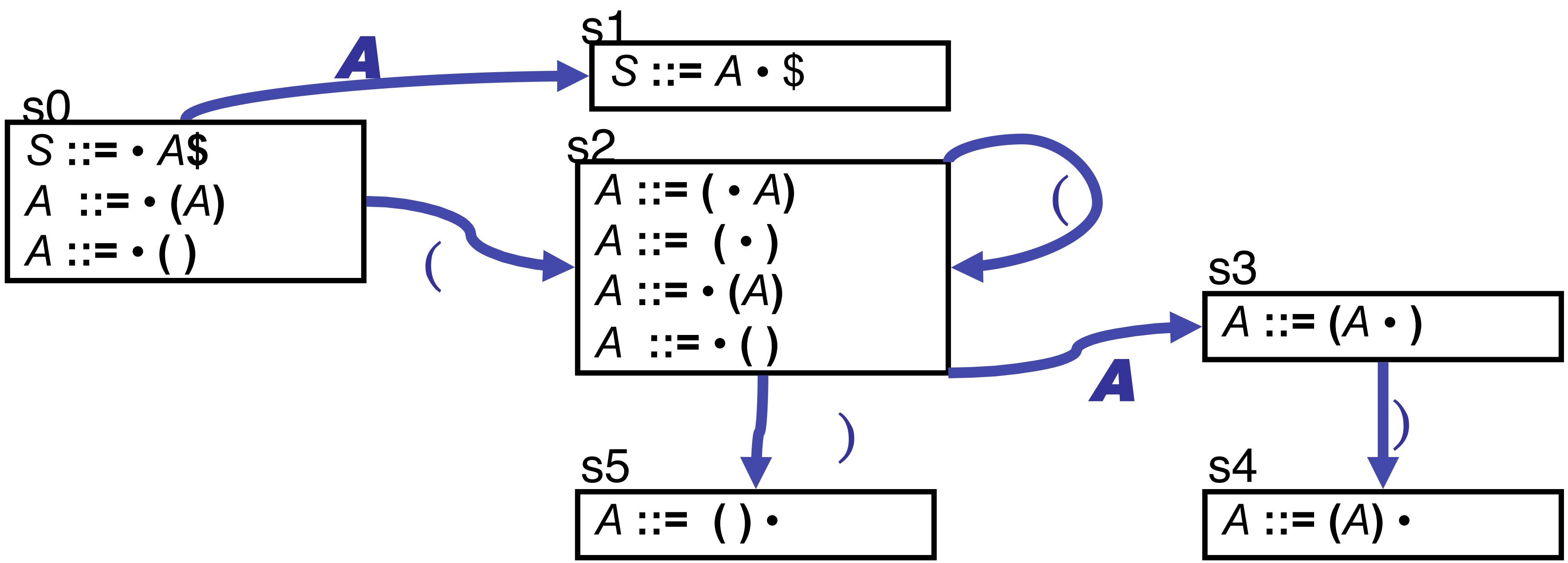
A ::= (•)

A ::= ( ) •

Building the DFA states

- Essentially the usual NFA2DFA construction!!
- Let A be the start symbol and S a new start symbol.
- Create a new rule S ::= A $
- Create the first state to be Closure({ *S* **::= •** ***A*** $})
- Pick a state **I**
  - for each item A **::=** α • X β in **I**
    - find Goto**(I, X)**
    - if Goto**(I, X)** is not already a state, make one
    - Add an edge X from state **I** to Goto**(I, X)** state
- Repeat until no more additions possible

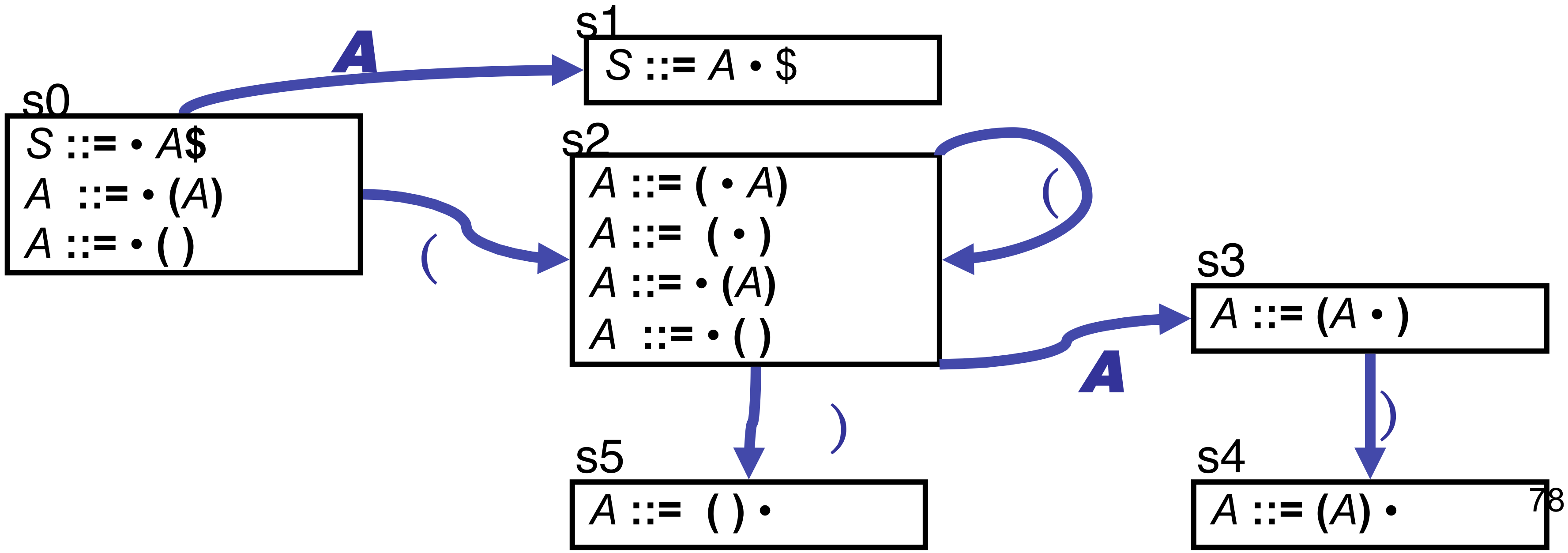
DFA Example



Creating the Parse Table(s)

(G10)
(1) S ::= A$
(2) A ::= (A)
(3) A ::= ( )

State	(	)	$	A
s0	shift to s2			goto s1
s1			accept	
s2	shift to s2	shift to s5		goto s3
s3		shift to s4		
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	



Parsing with an LR Table

Use table and top-of-stack and input symbol to get action:

If action is

shift sn : advance input one token,
push sn on stack

reduce X ::= α : pop stack 2* $|\alpha|$ times (grammar symbols are paired with states). In the state now on top of stack,
use goto table to get next state sn,
push it on top of stack

accept : stop and accept

error : weep (actually, produce a good error message)

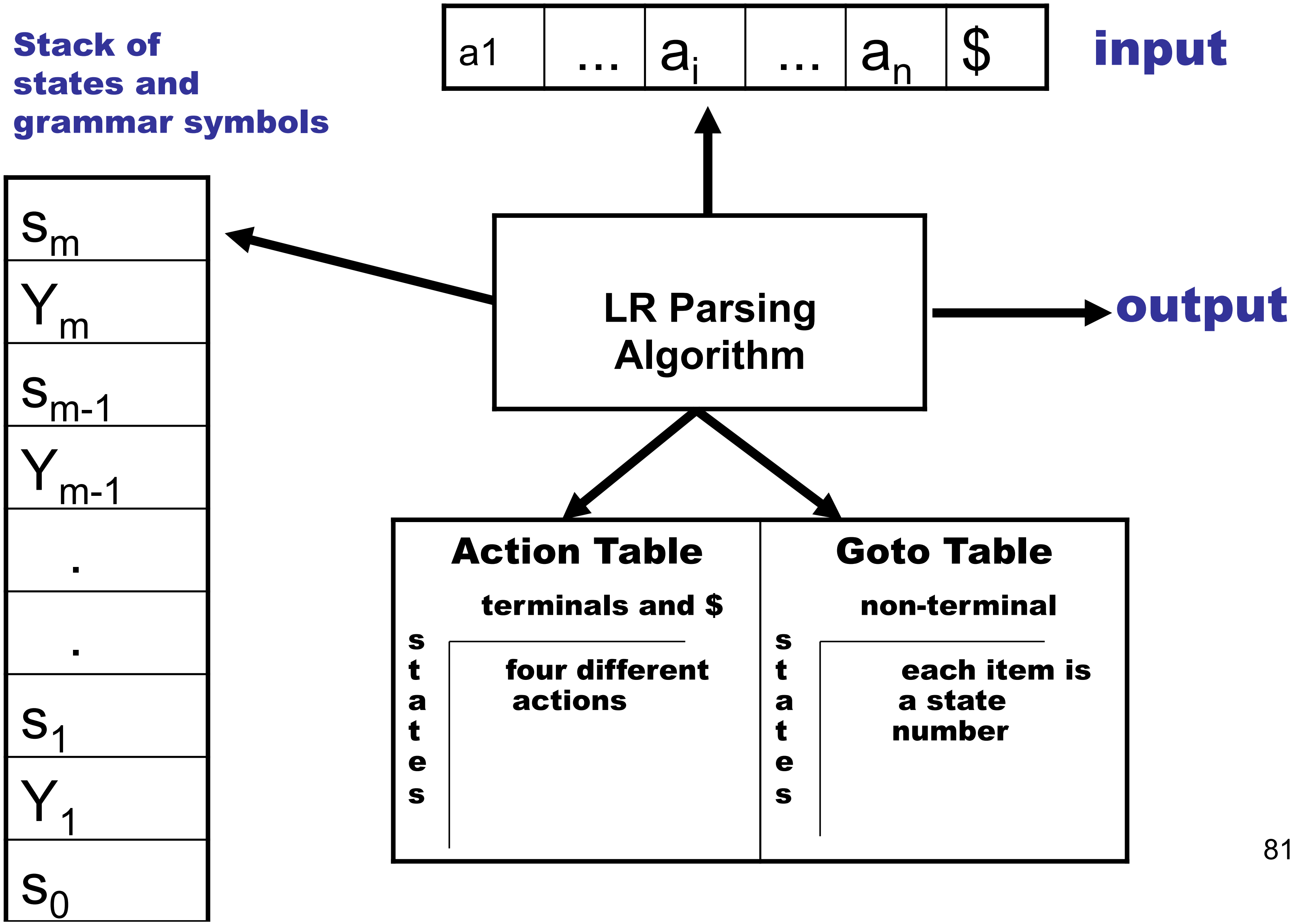
Parsing, again…

(G10)
(1) S ::= A$
(2) A ::= (A)
(3) A ::= ( )

		ACTION		Goto
State	**(**	**)**	**$**	**A**
s0	shift to s2			goto s1
s1			accept	
s2	shift to s2	shift to s5		goto s3
s3		shift to s4		
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

s0	**(())$**	**shift s2**
s0 (s2	**())$**	**shift s2**
s0 (s2 (s2	**))$**	**shift s5**
s0 (s2 (s2) s5	**)$**	**reduce A ::= ()**
s0 (s2 A	**)$**	**goto s3**
s0 (s2 A s3	**)$**	**shift s4**
s0 (s2 A s3) s4	**$**	**reduce A::= (A)**
s0 A	**$**	**goto s1**
s0 A s1	**$**	**ACCEPT!**

LR Parsing Algorithm



Problem With LR(0) Parsing

- **No lookahead**
- **Vulnerable to unnecessary conflicts**
  - **Shift/Reduce Conflicts (may reduce too soon in some cases)**
  - **Reduce/Reduce Conflicts**
- **Solutions:**
  - **LR(1) parsing - systematic lookahead**

LR(1) Items

- An LR(1) item is a pair:

  $(X ::= \alpha \,.\, \beta,\ a)$

  - $X ::= \alpha\beta$ is a production
  - a is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal

- $[X ::= \alpha \,.\, \beta, a]$ describes a context of the parser
  - We are trying to find an X followed by an a, and
  - We have (at least) α already on top of the stack
  - Thus we need to see next a prefix derived from βa

The Closure Operation

- Need to modify closure operation:.

Closure(Items) =
    repeat
        for each [X ::= α . Yβ, a] in Items
            for each production Y ::= γ
                for each b in First(βa)
                    add [Y ::= .γ, b] to Items
    until Items is unchanged

Constructing the Parsing DFA (2)

- A DFA state is a closed set of LR(1) items
- The start state contains (S' ::= .S$, dummy)
- A state that contains [X ::= α., b] is labeled with "reduce with X ::= α on lookahead b"
- And now the transitions …

The DFA Transitions

- A state s that contains [X ::= α.Yβ, b] has a transition labeled y to the state obtained from Transition(s, Y)
  - Y can be a terminal or a non-terminal

```
Transition(s, Y)
  Items = {}
  for each [X ::= α.Yβ, b] in s
      add [X ! αY.β, b] to Items
  return Closure(Items)
```

LR(1)-the parse table

- Shift and goto as before
- Reduce
  - state I with item ($A \rightarrow \alpha.$, z) gives a reduce $A \rightarrow \alpha$ if z is the next character in the input.

- LR(1)-parse tables are very big

LR(1)-DFA

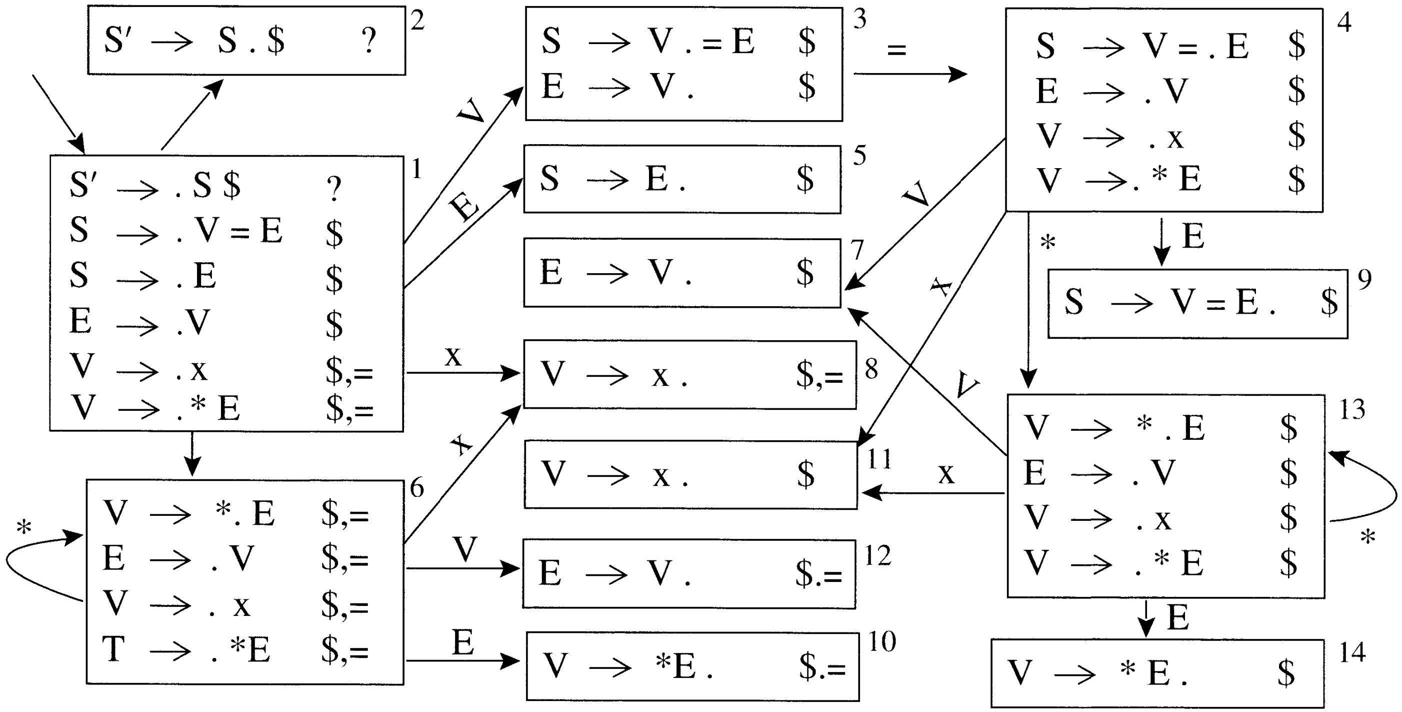
(G11)

S' ::= S$

S ::= V = E
| E

E ::= V

V ::= x
| *E



From Andrew Appel, “Modern Compiler Implementation in Java” page 65

LR(1)-parse table

	x	*	=	$	S	E	V		x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3	8			r4	r4			
2				acc				9				r1			
3			s4	r3				10			r5	r5			
4	s11	s13				g9	g7	11				r4			
5				r2				12			r3	r3			
6	s8	s6				g10	g12	13	s11	s13				g14	g7
7				r3				14				r5			

LALR States

- Consider for example the LR(1) states

$$\{[X ::= \alpha. , a], [Y ::= \beta. , c]\}$$

$$\{[X ::= \alpha. , b], [Y ::= \beta. , d]\}$$

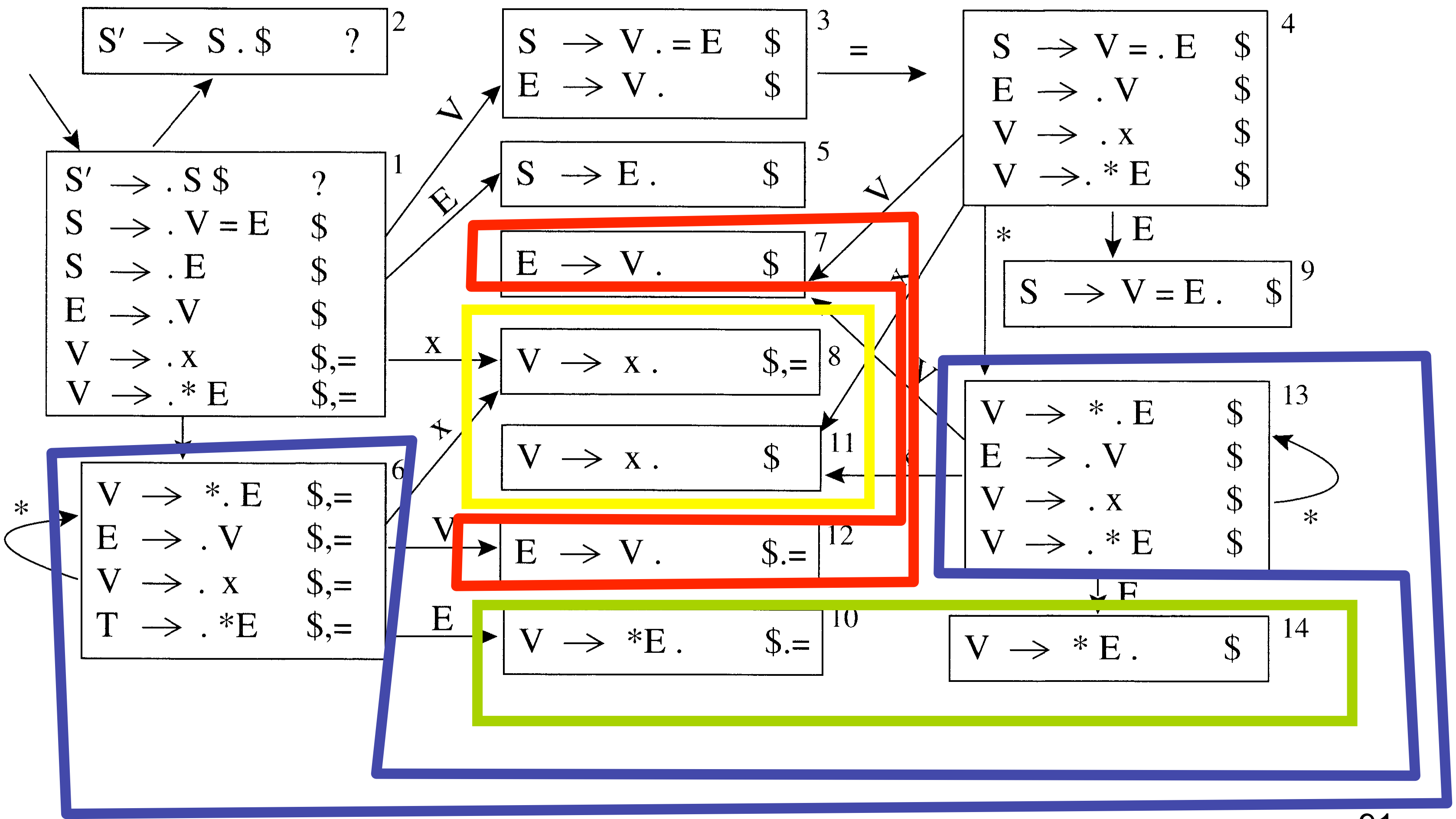
- They have the same *core* and can be merged to the state

$$\{[X ::= \alpha. , a/b], [Y ::= \beta. , c/d]\}$$

- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

For LALR(1), Collapse States ...

Combine states 6 and 13, 7 and 12, 8 and 11, 10 and 14.



LALR(1)-parse-table

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				acc			
3			s4	r3			
4	s8	s6				g9	g7
5							
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

LALR vs. LR Parsing

- LALR languages are not "natural"
  - They are an efficiency hack on LR languages

- You may see claims that any reasonable programming language has a LALR(1) grammar, {Arguably this is done by defining languages without an LALR(1) grammar as unreasonable ☺ }.

- In any case, LALR(1) has become a standard for programming languages and for parser generators, in spite of its apparent complexity.

Compiler Construction
Lent Term 2017

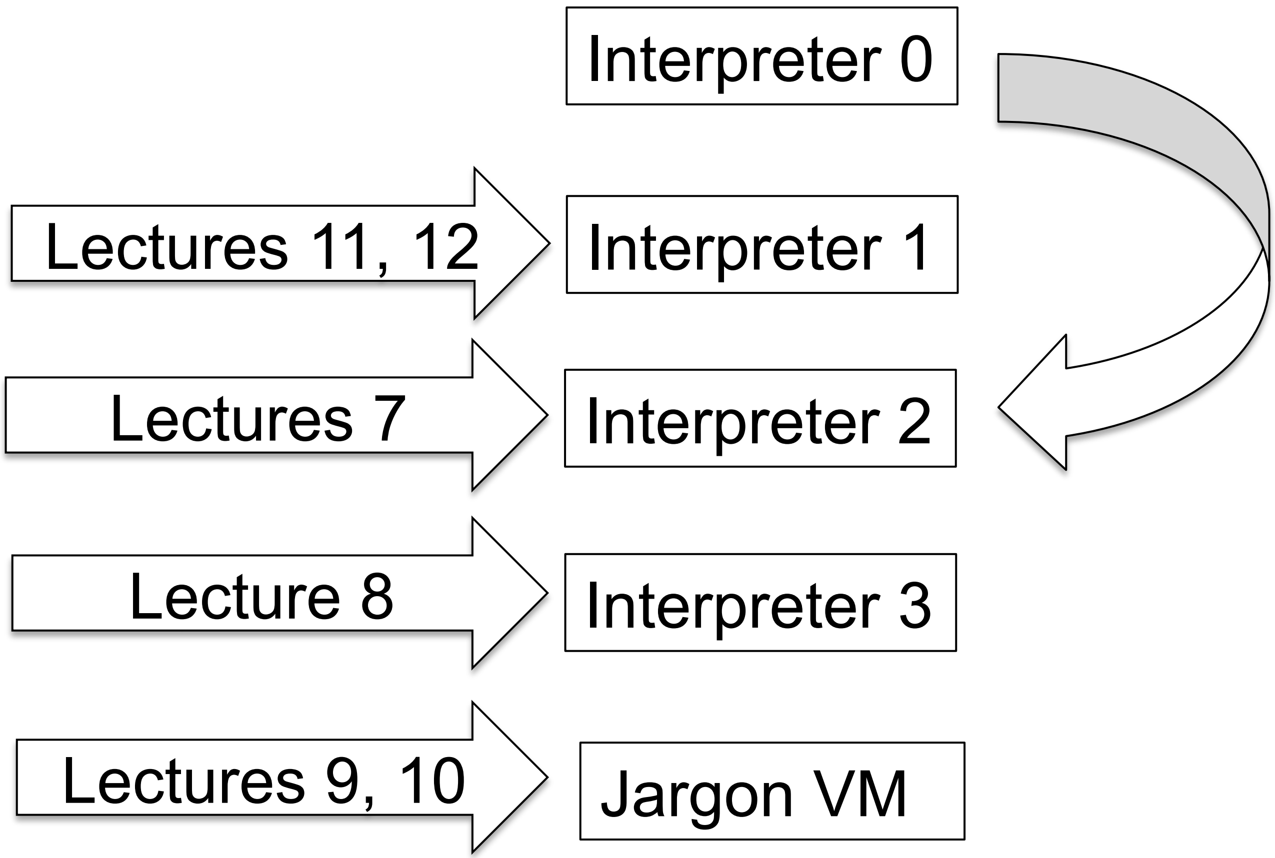
Part II : Lectures 7 – 12 (of 16)

Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge

Roadmap

Starting from a direct implementation of Slang/L3 semantics, we will **DERIVE** a Virtual Machine in a step-by-step manner. The correctness of each step is (more or less) easy to check.



Lecture 7:
We make this leap using intuition.

Later we will understand it more formally…

LECTURE 7
Interpreter 0, Interpreter 2

1. Interpreter 0 : The high-level “definitional” interpreter
   1. Slang/L3 values represented directly as OCaml values
   2. Recursive interpreter implements a denotational semantics
   3. The interpreter implicitly uses OCaml’s runtime stack
2. Interpreter 2: A high-level stack-oriented machine
   1. Makes the Ocaml runtime stack explicit
   2. Complex values pushed onto stacks
   3. One stack for values and environments
   4. One stack for instructions
   5. Heap used only for references
   6. Instructions have tree-like structure

Approaches to Mathematical Semantics

- Axiomatic: Meaning defined through logical specifications of behaviour.
  - Hoare Logic (Part II)
  - Separation Logic
- Operational: Meaning defined in terms of transition relations on states in an abstract machine.
  - Semantics (Part 1B)
- Denotational: Meaning is defined in terms of mathematical objects such as functions.
  - Denotational Semantics (Part II)

A denotational semantics for L3?

N = set of integers **B** = set of booleans **A** = set of addresses

I = set of identifiers Expr = set of L3 expressions

E = set of environments = **I** → **V** **S** = set of stores = **A** → **V**

V = set of value
≈ **A**
+ **N**
+ **B**
+ { () }
+ **V** × **V**
+ (**V** + **V**)
+ (**V** × **S**) → (**V** × **S**)

Set of values **V** solves this "domain equation" (here + means disjoint union).

Solving such equations is where some difficult maths is required …

M = the meaning function

M : (Expr × **E** × **S**) → (**V** × **S**)

Not examinable!!

Our shabby OCaml approximation

A = set of addresses

S = set of stores = **A** → **V**

V = set of value
≈ **A**
+ **N**
+ **B**
+ { () }
+ **V** × **V**
+ (**V** + **V**)
+ (**V** × **S**) → (**V** × **S**)

E = set of environments = **A** → **V**

M = the meaning function

M : (Expr × **E** × **S**) → (**V** × **S**)

```
type address

type store = address -> value

and value =
    | REF of address
    | INT of int
    | BOOL of bool
    | UNIT
    | PAIR of value * value
    | INL of value
    | INR of value
    | FUN of ((value * store)
                     -> (value * store))

type env = Ast.var -> value

val interpret :
    Ast.expr * env * store
                     -> (value * store)
```

Most of the code is obvious!

```
let rec interpret (e, env, store) =
  match e with
  | If(e1, e2, e3) ->
    let (v, store') = interpret(e1, env, store) in
        (match v with
         | BOOL true -> interpret(e2, env, store')
         | BOOL false -> interpret(e3, env, store')
         | v -> complain "runtime error.  Expecting a boolean!")
  | Pair(e1, e2)  ->
    let (v1, store1) = interpret(e1, env, store) in
    let (v2, store2) = interpret(e2, env, store1) in (PAIR(v1, v2), store2)
  | Fst e ->
     (match interpret(e, env, store) with
      | (PAIR (v1, _), store') -> (v1, store')
      | (v, _) -> complain "runtime error.  Expecting a pair!")
  | Snd e  ->
    (match interpret(e, env, store) with
      | (PAIR (_, v2), store') -> (v2, store')
      | (v, _) -> complain "runtime error.  Expecting a pair!")
  | Inl e   -> let (v, store') = interpret(e, env, store) in (INL v, store')
  | Inr e  -> let (v, store') = interpret(e, env, store) in (INR v, store')
  :
  .
```

Tricky bits : Slang functions mapped to OCaml functions!

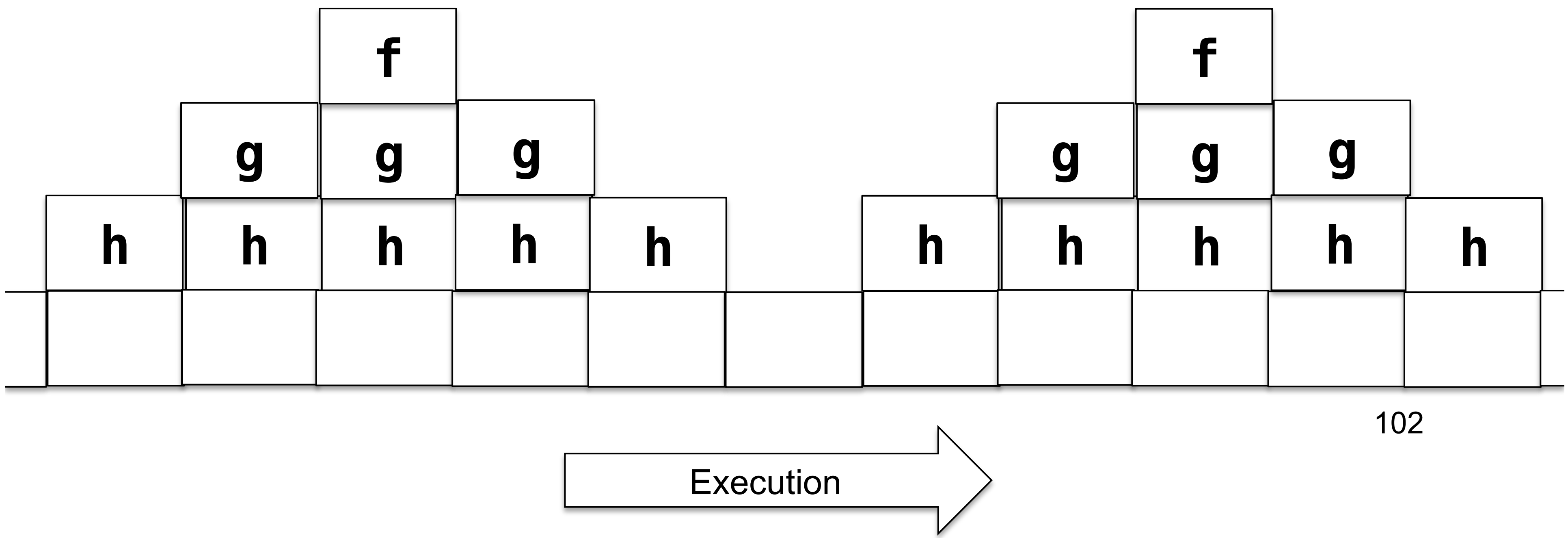
```
let rec interpret (e, env, store) =
    match e with
    :
    :
    | Lambda(x, e)  -> (FUN (fun (v, s) -> interpret(e, update(env, (x, v)), s)), store)
    | App(e1, e2) -> (* I chose to evaluate argument first!  *)
      let (v2, store1) = interpret(e2, env, store) in
      let (v1, store2) =  interpret(e1, env, store1) in
           (match v1 with
           | FUN f -> f (v2, store2)
           | v -> complain "runtime error.  Expecting a function!")
    | LetFun(f, (x, body), e) ->
       let new_env =
            update(env, (f, FUN (fun (v, s) -> interpret(body, update(env, (x, v)), s))))
       in interpret(e, new_env, store)
    | LetRecFun(f, (x, body), e) ->
       let rec new_env g = (* a recursive environment!!! *)
           if g = f then FUN (fun (v, s) -> interpret(body, update(new_env, (x, v)), s))
                    else env g
       in interpret(e, new_env, store)
```

update : env * (var * value) -> env

Typical implementation of function calls

```
let fun f (x) = x + 1
    fun g(y) = f(y+2)+2
    fun h(w) = g(w+1)+3
in
   h(h(17))
end
```

The run-time data structure is the <u>call stack</u> containing an <u>activation record</u> for each function invocation.



interpret is implicitly using Ocaml's runtime stack

```
let rec interpret (e, env, store) =
   match e with
    | Integer n           -> (INT n, store)
    | Op(e1, op, e2)   ->
      let (v1, store1) = interpret(e1, env, store) in
      let (v2, store2) = interpret(e2, env, store1) in
          (do_oper(op, v1, v2), store2)
   :
   :
```

- Every invocation of interpret is building an activation record on Ocaml's runtime stack.
- **We will now define interpreter 2 which makes this stack explicit**

Inpterp_2 data types

```
type address

type store = address -> value

and value =
     | REF of address
     | INT of int
     | BOOL of bool
     | UNIT
     | PAIR of value * value
     | INL of value
     | INR of value
     | FUN of ((value * store)
                    -> (value * store))

type env = Ast.var -> value
```

Interp_0

```
type address = int

type value =
     | REF of address
     | INT of int
     | BOOL of bool
     | UNIT
     | PAIR of value * value
     | INL of value
     | INR of value
     | CLOSURE of bool *
                     closure

and closure = code * env
```

Interp_2

```
and instruction =
   | PUSH of value
   | LOOKUP of var
   | UNARY of unary_oper
   | OPER of oper
   | ASSIGN
   | SWAP
   | POP
   | BIND of var
   | FST
   | SND
   | DEREF
   | APPLY
   | MK_PAIR
   | MK_INL
   | MK_INR
   | MK_REF
   | MK_CLOSURE of code
   | MK_REC of var * code
   | TEST of code * code
   | CASE of code * code
   | WHILE of code * code
```

Interp_2.ml : The Abstract Machine

```
and code = instruction list

and binding = var * value

and env = binding list

type env_or_value = EV of env | V of value

type env_value_stack = env_or_value list

type state = code * env_value_stack

val step : state -> state

val driver : state -> value

val compile : expr -> code

val interpret : expr -> value
```

The state is actually comprised of a heap --- a global array of values --- a pair of the form

(code, evn_value_stack)

Interpreter 2: The Abstract Machine

```
type state = code * env_value_stack

val step : state -> state
```

The state transition function.

```
let step = function
(* (code stack,          value/env stack) -> (code stack,  value/env stack) *)
 | ((PUSH v) :: ds,                    evs) -> (ds, (V v) :: evs)
 | (POP :: ds,                    s :: evs) -> (ds, evs)
 | (SWAP :: ds,             s1 :: s2 :: evs) -> (ds, s2 :: s1 :: evs)
 | ((BIND x) :: ds,            (V v) :: evs) -> (ds, EV([(x, v)]) :: evs)
 | ((LOOKUP x) :: ds,                  evs) -> (ds, V(search(evs, x)) :: evs)
 | ((UNARY op) :: ds,          (V v) :: evs) -> (ds, V(do_unary(op, v)) :: evs)
 | ((OPER op) :: ds,   (V v2) :: (V v1) :: evs) -> (ds, V(do_oper(op, v1, v2)) :: evs)
 | (MK_PAIR :: ds,     (V v2) :: (V v1) :: evs) -> (ds, V(PAIR(v1, v2)) :: evs)
 | (FST :: ds,            V(PAIR (v, _)) :: evs) -> (ds, (V v) :: evs)
 | (SND :: ds,            V(PAIR (_, v)) :: evs) -> (ds, (V v) :: evs)
 | (MK_INL :: ds,              (V v) :: evs) -> (ds, V(INL v) :: evs)
 | (MK_INR :: ds,              (V v) :: evs) -> (ds, V(INR v) :: evs)
 | (CASE (c1,  _) :: ds,        V(INL v)::evs) -> (c1 @ ds, (V v) :: evs)
 | (CASE ( _, c2) :: ds,        V(INR v)::evs) -> (c2 @ ds, (V v) :: evs)
 | ((TEST(c1, c2)) :: ds,  V(BOOL true) :: evs) -> (c1 @ ds, evs)
 | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)
 | (ASSIGN :: ds,  (V v) :: (V (REF a)) :: evs) -> (heap.(a) <- v; (ds, V(UNIT) :: evs))
 | (DEREF :: ds,           (V (REF a)) :: evs) -> (ds, V(heap.(a)) :: evs)
 | (MK_REF :: ds,              (V v) :: evs) -> let a = allocate () in (heap.(a) <- v;
                                                (ds, V(REF a) :: evs))
□| ((WHILE(c1, c2)) :: ds,V(BOOL false) :: evs) -> (ds, evs)
 | ((WHILE(c1, c2)) :: ds, V(BOOL true) :: evs) -> (c1 @ [WHILE(c1, c2)] @ ds, evs)
 | ((MK_CLOSURE c) :: ds,              evs) -> (ds,  V(mk_fun(c, evs_to_env evs)) :: evs)
 | (MK_REC(f, c) :: ds,                evs) -> (ds,  V(mk_rec(f, c, evs_to_env evs)) :: evs)
 | (APPLY :: ds,  V(CLOSURE (_, (c, env))) :: (V v) :: evs)
                                           -> (c @ ds, (V v) :: (EV env) :: evs)
 | state -> complain ("step : bad state = " ^ (string_of_state state) ^ "\n")
```

The driver. Correctness

```
(* val driver : state -> value *)
let rec driver state =
   match state with
   | ([], [V v]) -> v
   | _            -> driver (step state)
```

In other words, evaluating
  compile e
should leave the value of e on top of the stack

```
val compile : expr -> code
```

The idea: if e passes the frond-end and
    Interp_0.interpret e = v
then
    driver (compile e, []) = v'
where v' (somehow) represents v.

Implement inter_0 in interp_2

interp_0.ml

```
let rec interpret (e, env, store) =
   match e with
| Pair(e1, e2)  ->
     let (v1, store1) = interpret(e1, env, store) in
     let (v2, store2) = interpret(e2, env, store1) in (PAIR(v1, v2), store2)
   | Fst e ->
      (match interpret(e, env, store) with
      | (PAIR (v1, _), store') -> (v1, store')
      | (v, _) -> complain "runtime error.  Expecting a pair!")
   :
```

```
let step = function
 | (MK_PAIR :: ds,  (V v2) :: (V v1) :: evs)  ->  (ds,   V(PAIR(v1, v2)) :: evs)
 | (FST :: ds,            V(PAIR (v, _)) :: evs)  ->  (ds,   (V v) :: evs)
 :

let rec compile = function
 | Pair(e1, e2)   -> (compile e1) @ (compile e2) @ [MK_PAIR]
 | Fst e              -> (compile e) @ [FST]
 :
```

interp_2.ml

Implement inter_0 in interp_2

interp_0.ml

```
let rec interpret (e, env, store) =
   match e with
   | If(e1, e2, e3) ->
     let (v, store') = interpret(e1, env, store) in
         (match v with
          | BOOL true -> interpret(e2, env, store')
          | BOOL false -> interpret(e3, env, store')
          | v -> complain "runtime error.  Expecting a boolean!")
    :
```

```
let step = function
 | ((TEST(c1, c2)) :: ds,  V(BOOL true) :: evs) -> (c1 @ ds, evs)
 | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)
 :

let rec compile = function
 | If(e1, e2, e3) -> (compile e1) @ [TEST(compile e2, compile e3)]
 :
```

interp_2.ml

Tricky bits again!

interp_0.ml

```
let rec interpret (e, env, store) =
   match e with
   | Lambda(x, e)  -> (FUN (fun (v, s) -> interpret(e, update(env, (x, v)), s)), store)
   | App(e1, e2) -> (* I chose to evaluate argument first!  *)
     let (v2, store1) = interpret(e2, env, store) in
     let (v1, store2) =  interpret(e1, env, store1) in
         (match v1 with
         | FUN f -> f (v2, store2)
         | v -> complain "runtime error.  Expecting a function!")
   :
```

interp_2.ml

```
let step = function
 | (POP :: ds,                          s :: evs) -> (ds,  evs)
 | (SWAP :: ds,                   s1 :: s2 :: evs) -> (ds,  s2 :: s1 :: evs)
 | ((BIND x) :: ds,                  (V v) :: evs) -> (ds,  EV([(x, v)]) :: evs)
 | ((MK_CLOSURE c) :: ds,                   evs) -> (ds,   V(mk_fun(c, evs_to_env evs)) :: evs)
 | (APPLY :: ds,  V(CLOSURE (_, (c, env))) :: (V v) :: evs)
                                              -> (c @ ds,  (V v) :: (EV env) :: evs)
let rec compile = function
 | Lambda(x, e)   -> [MK_CLOSURE((BIND x) :: (compile e) @ [SWAP; POP])]
 | App(e1, e2)      -> (compile e2) @ (compile e1) @ [APPLY; SWAP; POP]
 :
```

Example : Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int = (snd p, fst p)
in
   rev_pair (21, 17)
end
```

```
MK_CLOSURE([BIND p; LOOKUP p; SND; LOOKUP p; FST; MK_PAIR; SWAP; POP]);
 BIND rev_pair;
 PUSH 21;
 PUSH 17;
 MK_PAIR;
 LOOKUP rev_pair;
 APPLY;
 SWAP;
 POP;
 SWAP;
 POP
```

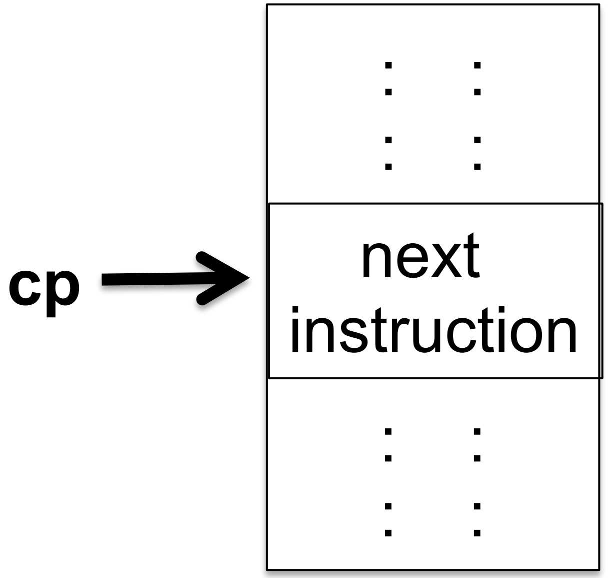
DEMO TIME!!!

LECTURE 8
Derive Interpreter 3

1. “Flatten” code into linear array
2. Add “code pointer” (cp) to machine state
3. New instructions : LABEL, GOTO, RETURN
4. “Compile away” conditionals and while loops

Linearise code

Interpreter 2 copies code
on the code stack.
We want to introduce one
global array of instructions
indexed by a code pointer (**cp**).
At runtime the **cp** points at the
next instruction to be executed.



This will require two new instructions:

LABEL L : Associate label L with this location in the code array

GOTO L : Set the **cp** to the code address associated with L

Compile conditionals, loops

`If(e1, e2, e3)`

code for `e1`
TEST k
code for `e2`
GOTO m
k: code for `e3`
m:

`While(e1, e2)`

m: code for `e1`
TEST k
code for `e2`
GOTO m
k:

If ? = 0 Then 17 else 21 end

interp_2

```
PUSH UNIT;
UNARY READ;
PUSH 0;
OPER EQI;
TEST(
   [PUSH 17],
   [PUSH 21]
)
```

interp_3

```
PUSH UNIT;
UNARY READ;
PUSH 0;
OPER EQI;
TEST L0;
PUSH 17;
GOTO L1;
LABEL L0;
PUSH 21;
LABEL L1;
HALT
```

Symbolic code locations

interp_3 (loaded)

```
0: PUSH UNIT;
1: UNARY READ;
2: PUSH 0;
3: OPER EQI;
4: TEST L0 = 7;
5: PUSH 17;
6: GOTO L1 = 9;
7: LABEL L0;
8: PUSH 21;
9: LABEL L1;
10: HALT
```

Numeric code locations

Implement inter_2 in interp_3

```
let step = function
 | ((TEST(c1, c2)) :: ds,  V(BOOL true) :: evs) -> (c1 @ ds, evs)
 | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)
 :
```

interp_2.ml

```
let step (cp, evs) =
 match (get_instruction cp, evs) with
 | (TEST (_, Some _),  V(BOOL true) :: evs)  ->  (cp + 1, evs)
 | (TEST (_, Some i),  V(BOOL false) :: evs) ->  (i,      evs)
 | (LABEL l,                            evs) ->  (cp + 1, evs)
 | (GOTO (_, Some i),                   evs) ->  (i,      evs)
 :
```

Interp_3.ml

Code locations are represented as

(“L”, None) : not yet loaded (assigned numeric address)

(“L”, Some i) : label “L” has been assigned numeric address i

Tricky bits again!

interp_2.ml

```
let step = function
 | (POP :: ds,                        s :: evs) -> (ds,  evs)
 | (SWAP :: ds,                 s1 :: s2 :: evs) -> (ds,  s2 :: s1 :: evs)
 | ((BIND x) :: ds,                 (V v) :: evs) -> (ds,  EV([(x, v)]) :: evs)
 | ((MK_CLOSURE c) :: ds,                   evs) -> (ds,  V(mk_fun(c, evs_to_env evs)) :: evs)
 | (APPLY :: ds,  V(CLOSURE (_, (c, env))) :: (V v) :: evs)
                                     -> (c @ ds, (V v) :: (EV env) :: evs)
```

interp_3.ml

```
let step (cp, evs) =
 match (get_instruction cp, evs) with
 | (POP,                              s :: evs) -> (cp + 1, evs)
 | (SWAP,                       s1 :: s2 :: evs) -> (cp + 1, s2 :: s1 :: evs)
 | (BIND x,                       (V v) :: evs) -> (cp + 1, EV([(x, v)]) :: evs)
 | (MK_CLOSURE loc,                        evs) -> (cp + 1,
                                       V(CLOSURE(loc, evs_to_env evs)) :: evs)
 | (RETURN,   (V v) :: _ :: (RA i) :: evs)  -> (i, (V v) :: evs)
 | (APPLY,  V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
                                ->   (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
```

Note that in interp_2 the body of a closure is consumed from the code stack. But in interp_3 we need to save the return address on the stack (here i is the location of the closure's code).

Tricky bits again!

interp_2.ml

```
let rec compile = function
 | Lambda(x, e)   -> [MK_CLOSURE((BIND x) :: (compile e) @ [SWAP; POP])]
 | App(e1, e2)     -> (compile e2) @ (compile e1) @ [APPLY; SWAP; POP]
 :
```

Interp_3.ml

```
let rec comp = function
 | App(e1, e2)   ->
  let (defs1, c1) = comp e1 in
  let (defs2, c2) = comp e2 in
      (defs1 @ defs2, c2 @ c1 @ [APPLY])
 | Lambda(x, e)   ->
  let (defs, c) = comp e in
  let f = new_label () in
  let def = [LABEL f ; BIND x] @ c @ [SWAP; POP; RETURN] in
     (def @ defs, [MK_CLOSURE((f, None))])
```

Interp_3.ml

```
let compile e =
   let (defs, c) = comp e in
     c              (* body of program *)
    @ [HALT]        (* stop the interpreter *)
    @ defs          (*  function definitions *)
```

Interpreter 3
(very similar to interpreter 2)

```
let step (cp, evs) =
 match (get_instruction cp, evs) with
 | (PUSH v,                           evs) -> (cp + 1, (V v) :: evs)
 | (POP,                         s :: evs) -> (cp + 1, evs)
 | (SWAP,                  s1 :: s2 :: evs) -> (cp + 1, s2 :: s1 :: evs)
 | (BIND x,                  (V v) :: evs) -> (cp + 1, EV([(x, v)]) :: evs)
 | (LOOKUP x,                         evs) -> (cp + 1, V(search(evs, x)) :: evs)
 | (UNARY op,                (V v) :: evs) -> (cp + 1, V(do_unary(op, v)) :: evs)
 | (OPER op,        (V v2) :: (V v1) :: evs) -> (cp + 1, V(do_oper(op, v1, v2)) :: evs)
 | (MK_PAIR,        (V v2) :: (V v1) :: evs) -> (cp + 1, V(PAIR(v1, v2)) :: evs)
 | (FST,               V(PAIR (v, _)) :: evs) -> (cp + 1, (V v) :: evs)
 | (SND,               V(PAIR (_, v)) :: evs) -> (cp + 1, (V v) :: evs)
 | (MK_INL,                  (V v) :: evs) -> (cp + 1, V(INL v) :: evs)
 | (MK_INR,                  (V v) :: evs) -> (cp + 1, V(INR v) :: evs)
 | (CASE (_, Some _),         V(INL v)::evs) -> (cp + 1, (V v) :: evs)
 | (CASE (_, Some i),         V(INR v)::evs) -> (i,      (V v) :: evs)
 | (TEST (_, Some _),   V(BOOL true) :: evs) -> (cp + 1, evs)
 | (TEST (_, Some i),  V(BOOL false) :: evs) -> (i,      evs)
 | (ASSIGN,     (V v) :: (V (REF a)) :: evs) -> (heap.(a) <- v; (cp + 1, V(UNIT) :: evs))
 | (DEREF,             (V (REF a)) :: evs) -> (cp + 1, V(heap.(a)) :: evs)
 | (MK_REF,                  (V v) :: evs) -> let a = new_address () in (heap.(a) <- v;
                                              (cp + 1, V(REF a) :: evs))
 | (MK_CLOSURE loc,                   evs) -> (cp + 1, V(CLOSURE(loc, evs_to_env evs)) :: evs)
 | (APPLY,  V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
                                           -> (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
(* new intructions *)
 | (RETURN,     (V v) :: _ :: (RA i) :: evs) -> (i, (V v) :: evs)
 | (LABEL l,                          evs) -> (cp + 1, evs)
 | (HALT,                             evs) -> (cp, evs)
 | (GOTO (_, Some i),                 evs) -> (i, evs)
 | _ -> complain ("step : bad state = " ^ (string_of_state (cp, evs)) ^ "\n")
```

Some observations

- A very clean machine!
- But it still has a **very** inefficient treatment of environments.
- Also, pushing complex values on the stack is not what most virtual machines do. In fact, we are still using OCaml's runtime memory management to manipulate complex values.

Example : Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int  = (snd p, fst p)
in
   rev_pair (21, 17)
end
```

```
MK_CLOSURE(
   [BIND p; LOOKUP p; SND;
    LOOKUP p; FST; MK_PAIR;
    SWAP; POP]);
 BIND rev_pair;
 PUSH 21;
 PUSH 17;
 MK_PAIR;
 LOOKUP rev_pair;
 APPLY;
 SWAP;
 POP;
 SWAP;
 POP
```

Interp_2

```
MK_CLOSURE(rev_pair)
 BIND rev_pair
 PUSH 21
 PUSH 17
 MK_PAIR
 LOOKUP rev_pair
 APPLY
 SWAP
 POP
 HALT
```

```
LABEL rev_pair
 BIND p
 LOOKUP p
 SND
 LOOKUP p
 FST
 MK_PAIR
 SWAP
 POP
 RETURN
```

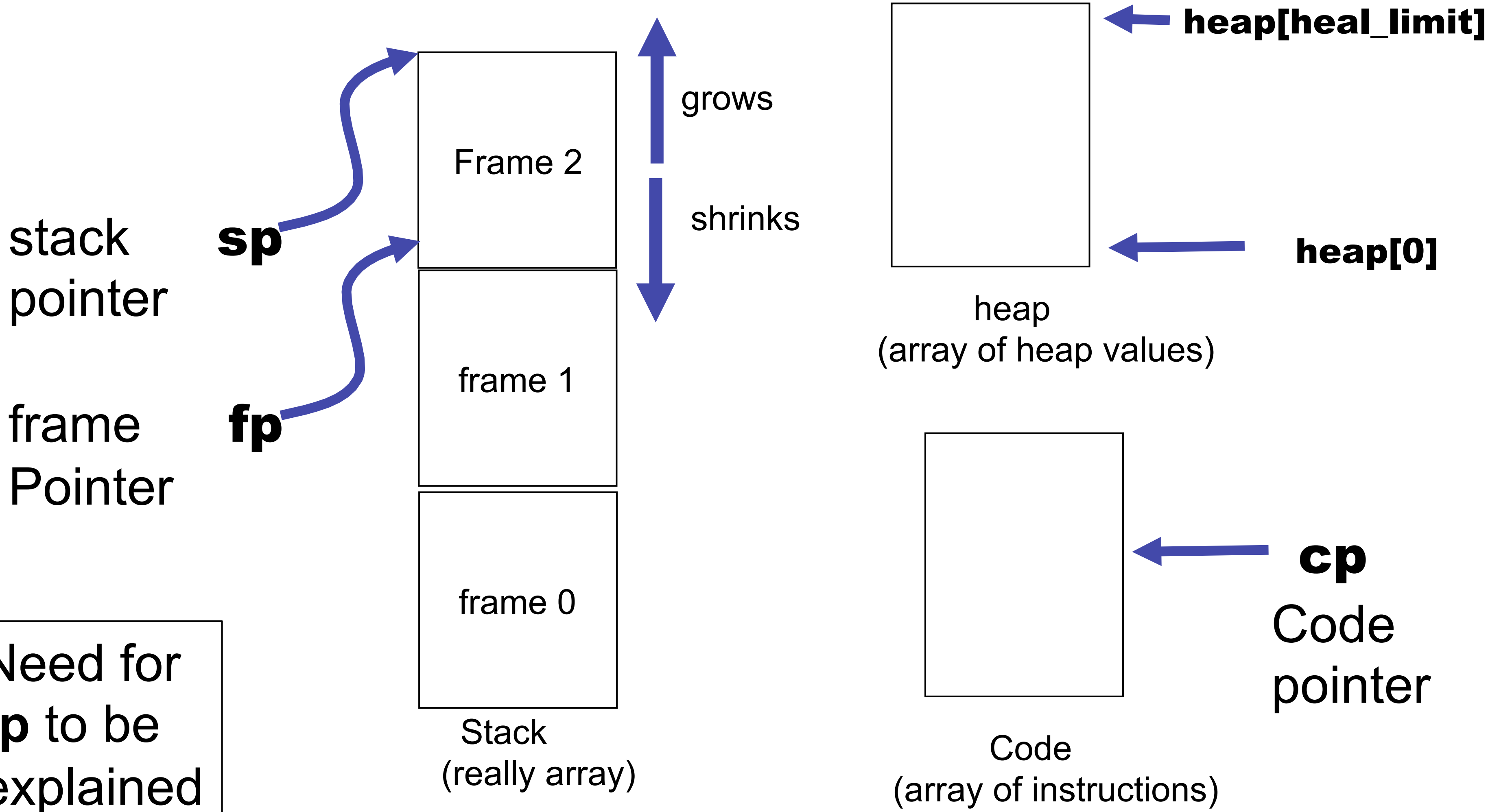
Interp_3

DEMO TIME!!!

LECTURES 9, 10
Deriving The Jargon VM
(interpreter 4)

1. **First change**: Introduce an **addressable stack.**
2. Replace variable lookup by a (relative) location on the stack or heap determined at **compile time**.
3. Relative to what? A **frame pointer** (**fp**) pointing into the stack is needed to keep track of the current **activation record.**
4. **Second change**: Optimise the representation of closures so that they contain **only** the values associated with the **free variables** of the closure and a pointer to code.
5. **Third change**: Restrict values on stack to be simple (ints, bools, heap addresses, etc). Complex data is moved to the heap, leaving pointers into the heap on the stack.
6. How might things look different in a language without first-class functions? In a language with multiple arguments to function calls?

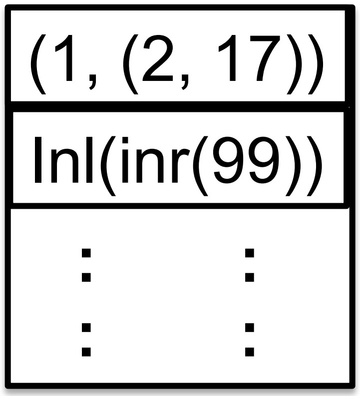
Jargon Virtual Machine



Need for **fp** to be explained soon …

The stack in interpreter 3

A stack
in interpreter 3



"All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections."
--- David Wheeler

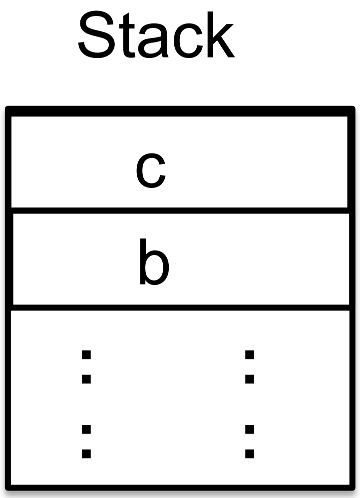
Stack elements in interpreter 3 are not of fixed size.

Virtual machines (JVM, etc) typically restrict stack elements to be of a fixed size

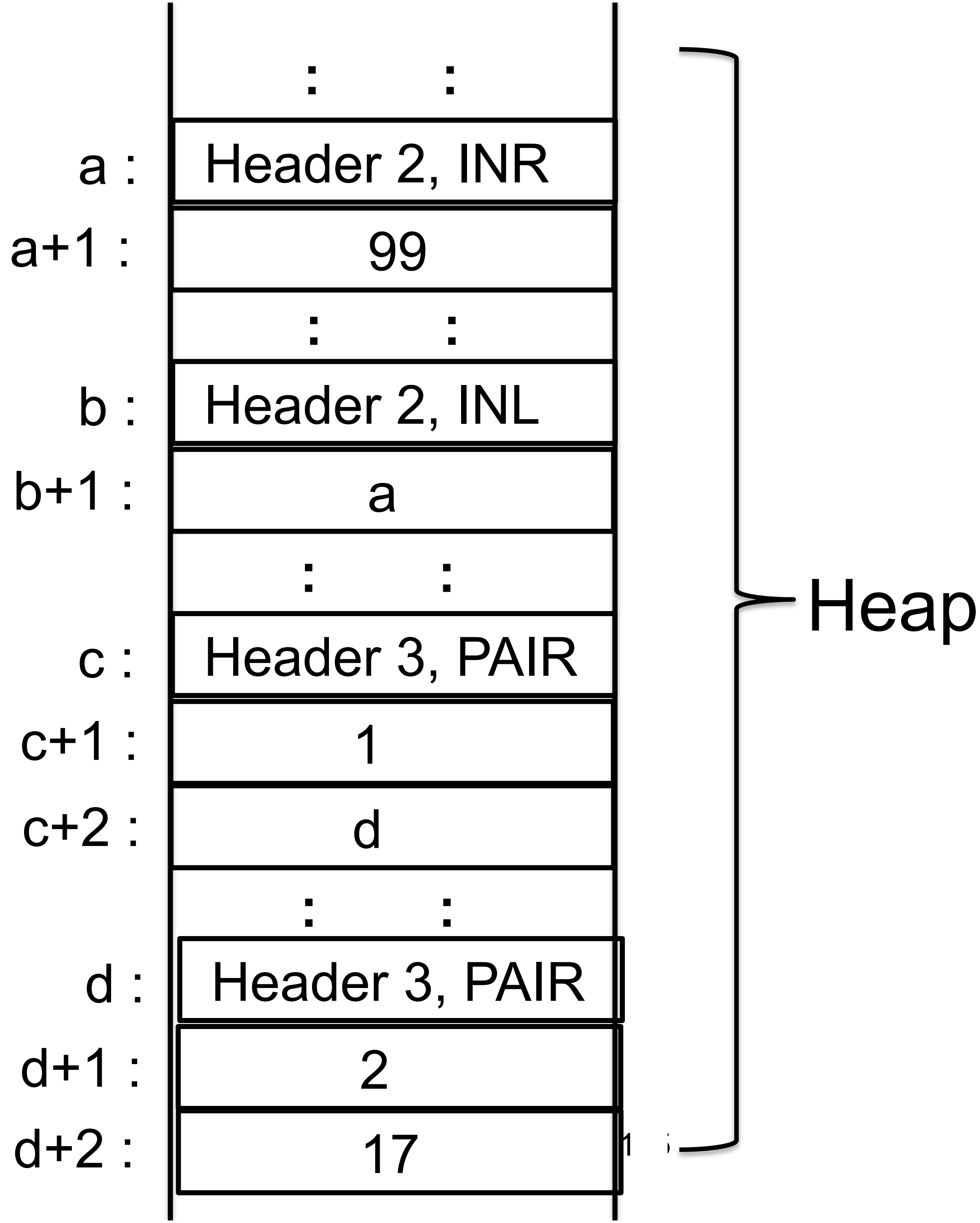
We need to shift data from the high-level stack of interpreter 3 to a lower-level stack with fixed size elements.

Solution : put the data in the heap. Place pointers to the heap on the stack.

The Jargon VM stack



Some stack elements represent pointers into the heap



interp_3.mli

Small change to instructions

jargon.mli

```
type instruction =
 | PUSH of value
 | LOOKUP of Ast.var
 | UNARY of Ast.unary_oper
 | OPER of Ast.oper
 | ASSIGN
 | SWAP
 | POP
 | BIND of Ast.var
 | FST
 | SND
 | DEREF
 | APPLY
 | RETURN
 | MK_PAIR
 | MK_INL
 | MK_INR
 | MK_REF
 | MK_CLOSURE of location
 | TEST of location
 | CASE of location
 | GOTO of location
 | LABEL of label
 | HALT
```

```
type instruction =
 | PUSH of stack_item              (* modified *)
 | LOOKUP of value_path            (* modified *)
 | UNARY of Ast.unary_oper
 | OPER of Ast.oper
 | ASSIGN
 | SWAP
 | POP
(* | BIND of var                   not needed *)
 | FST
 | SND
 | DEREF
 | APPLY
 | RETURN
 | MK_PAIR
 | MK_INL
 | MK_INR
 | MK_REF
 | MK_CLOSURE of location * int    (* modified *)
 | TEST of location
 | CASE of location
 | GOTO of location
 | LABEL of label
 | HALT
```

A word about implementation

Interpreter 3

```
type value = | REF of address | INT of int | BOOL of bool | UNIT
  | PAIR of value * value | INL of value | INR of value | CLOSURE of location * env
type env_or_value = | EV of env | V of value | RA of address
type env_value_stack = env_or_value list
```

Jargon VM

```
type stack_item =
  | STACK_INT of int
  | STACK_BOOL of bool
  | STACK_UNIT
  | STACK_HI of heap_index     (* Heap Index              *)
  | STACK_RA of code_index     (* Return Address          *)
  | STACK_FP of stack_index    (* (saved) Frame Pointer *)
```

```
type heap_type =
  | HT_PAIR
  | HT_INL
  | HT_INR
  | HT_CLOSURE
```

```
type heap_item =
  | HEAP_INT of int
  | HEAP_BOOL of bool
  | HEAP_UNIT
  | HEAP_HI of heap_index              (* Heap  Index                          *)
  | HEAP_CI of code_index              (* Code pointer for closures            *)
  | HEAP_HEADER of int * heap_type (* int is number items in heap block *)
```

The headers will be essential for garbage collection!

MK_INR (MK_INL is similar)

In interpreter 3

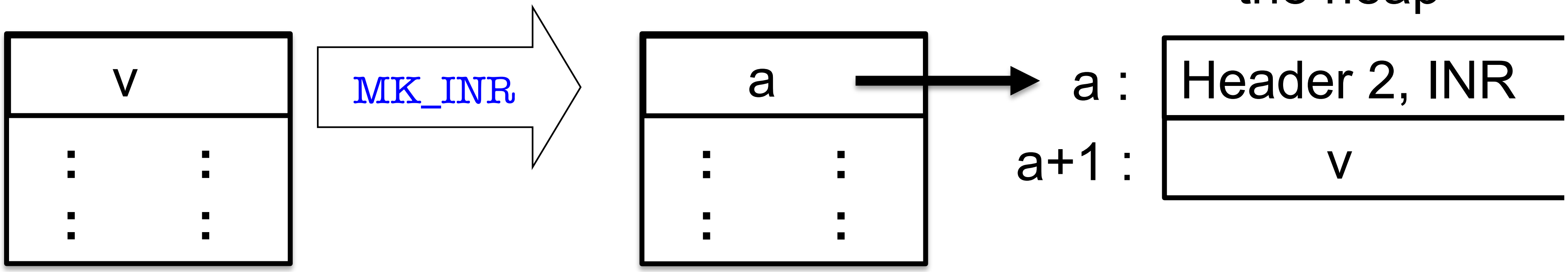
(MK_INR, (V v) :: evs) -> (cp + 1, V(INR(v)) :: evs)

Jargon VM

The stack before

The stack after

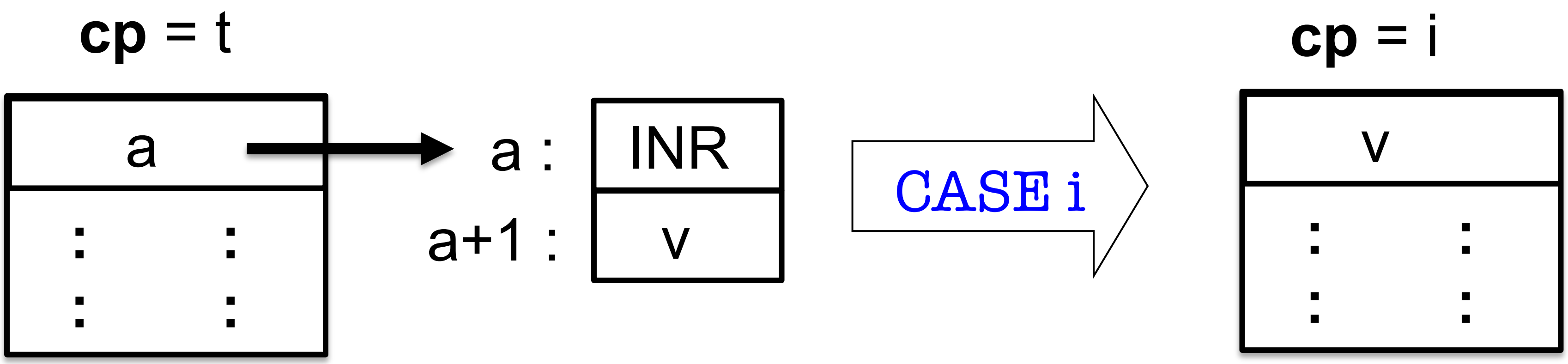
Newly allocated locations in the heap

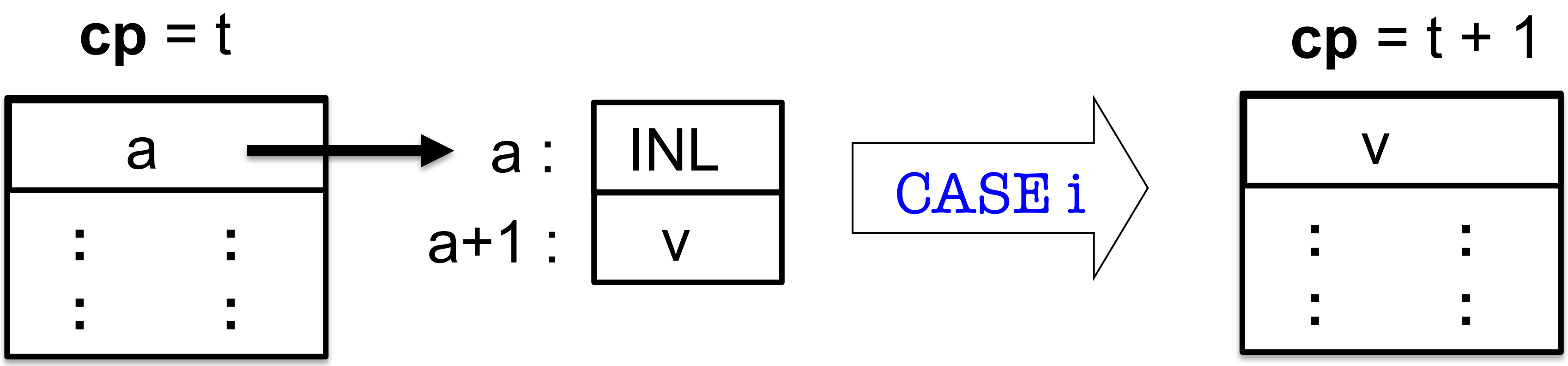


Note: The header types are not really required. We could instead add an extra field here (for example, 0 or 1). However, header types aid in understanding the code and traces of runtime execution.

CASE (TEST is similar)

(CASE (_, Some _), V(INL v)::evs) -> (cp + 1, (V v) :: evs)
(CASE (_, Some i), V(INR v)::evs) -> (i, (V v) :: evs)



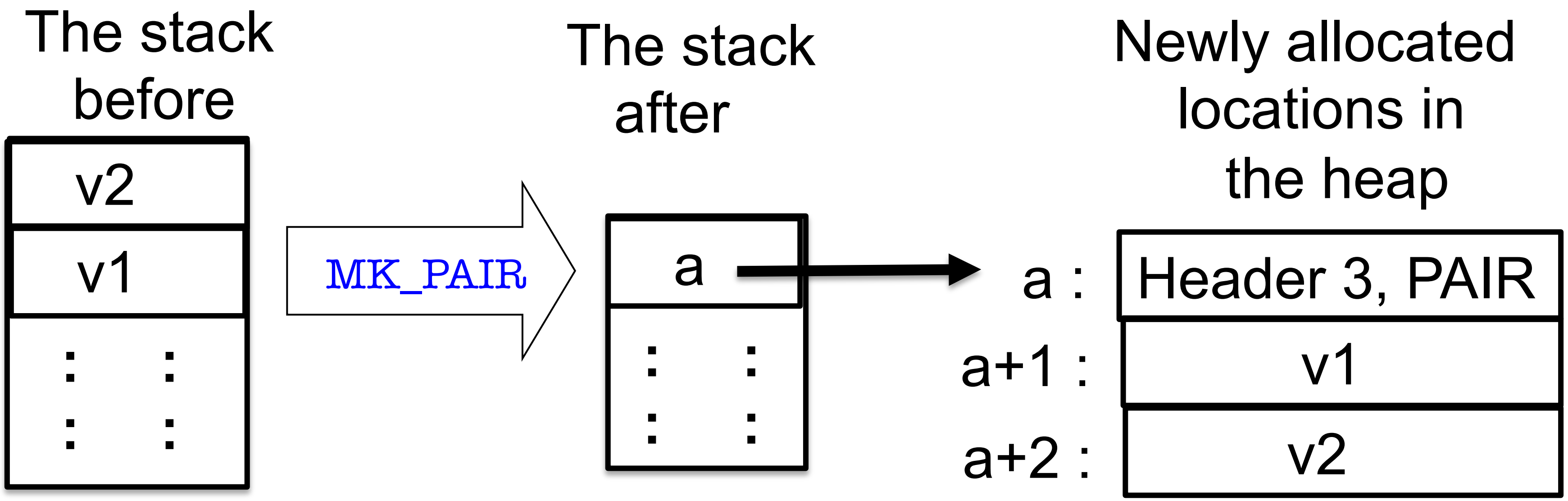


MK_PAIR

In interpreter 3:

(MK_PAIR, (V v2) :: (V v1) :: evs) -> (cp + 1, V(PAIR(v1, v2)) :: evs)

In Jargon VM:

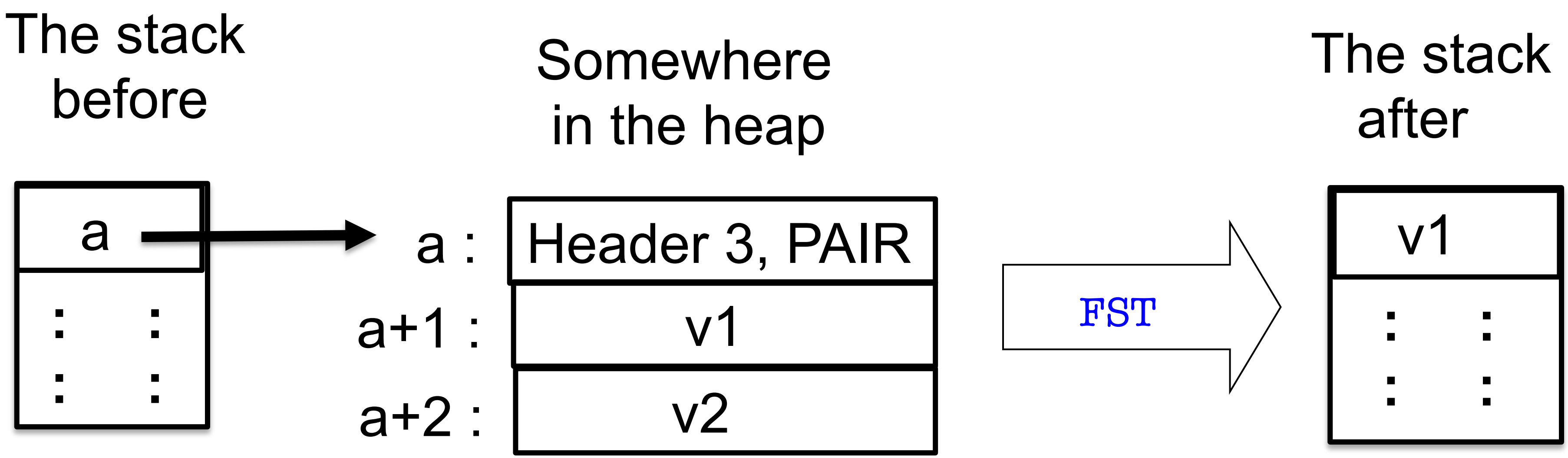


FST (similar for SND)

In interpreter 3:

(FST, V (PAIR(v1, v2)) :: evs) -> (cp + 1, v1 :: evs)

In Jargon VM:



Note that v1 could be a simple value (int or bool), or aother heap address.

These require more care …

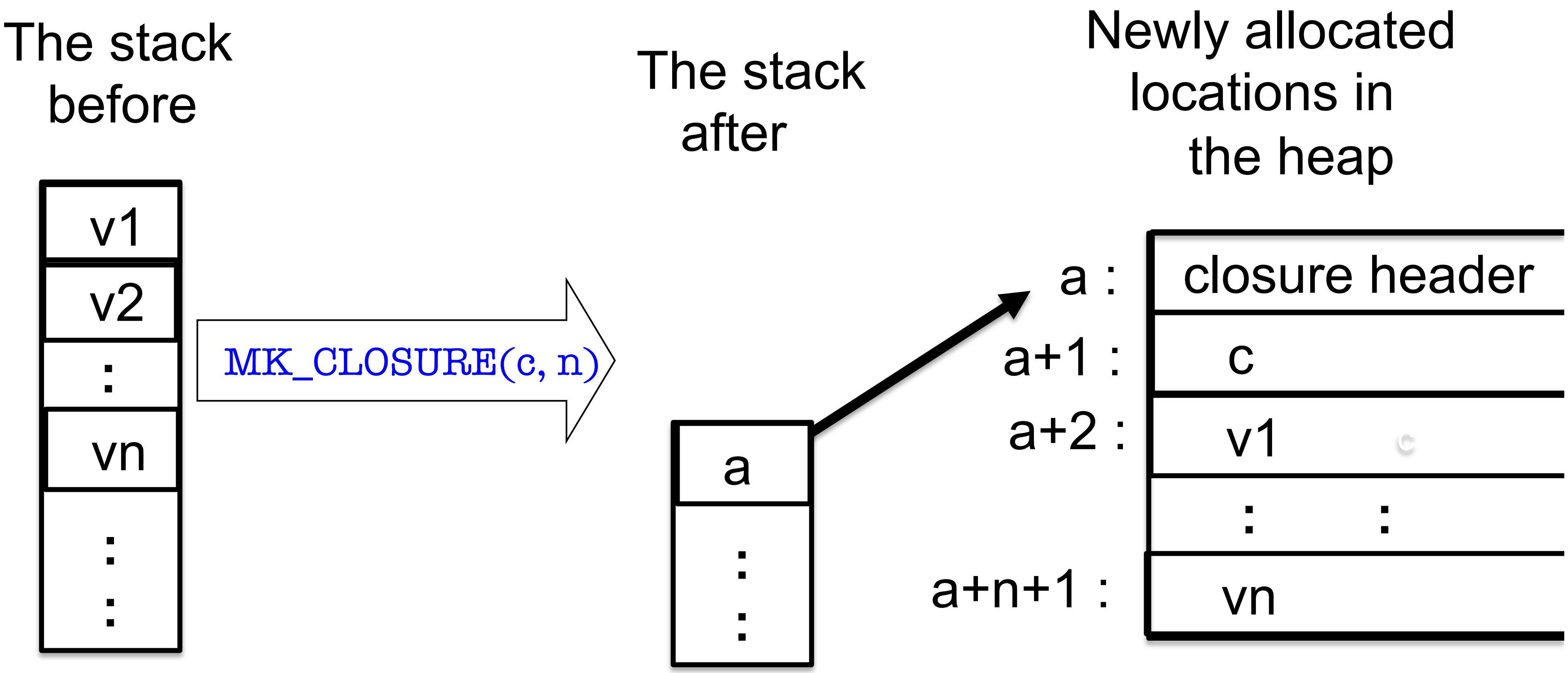
In interpreter 3:

```
let step (cp, evs) =
 match (get_instruction cp, evs) with
 | (MK_CLOSURE loc,   evs)
   -> (cp + 1, V(CLOSURE(loc, evs_to_env evs)) :: evs)

 | (APPLY,   V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
   -> (i,  (V v) :: (EV env) :: (RA (cp + 1)) :: evs)

 | (RETURN,   (V v) :: _ :: (RA i) :: evs)
   -> (i,  (V v) :: evs)
```

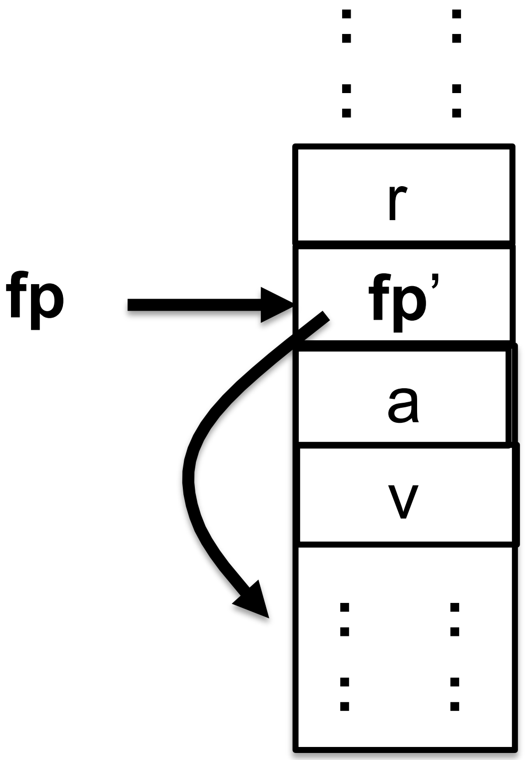
MK_CLOSURE(c, n)

c = code location of start of instructions for closure,
n = number of free variables in the body of closure.

Put values associated with **free variables** on stack,
then construct the closure on the heap



A stack frame



Return address

Saved frame pointer

Pointer to closure

Argument value

Stack frame. (Boundary May vary in the literature.)

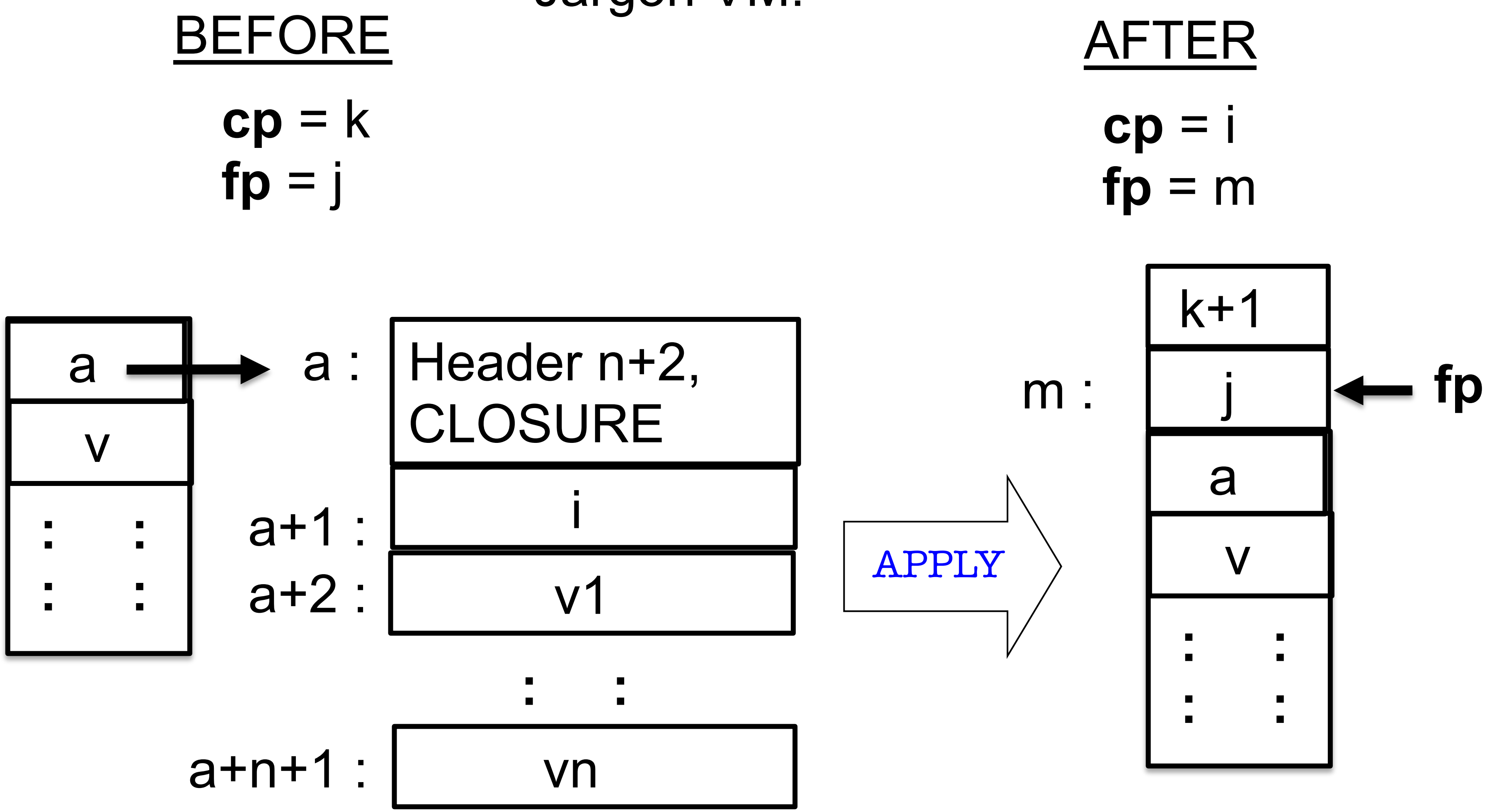
Currently executing code for the closure at heap address "a" after it was applied to argument v.

APPLY

Interpreter 3:

(APPLY, V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
-> (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)

Jargon VM:

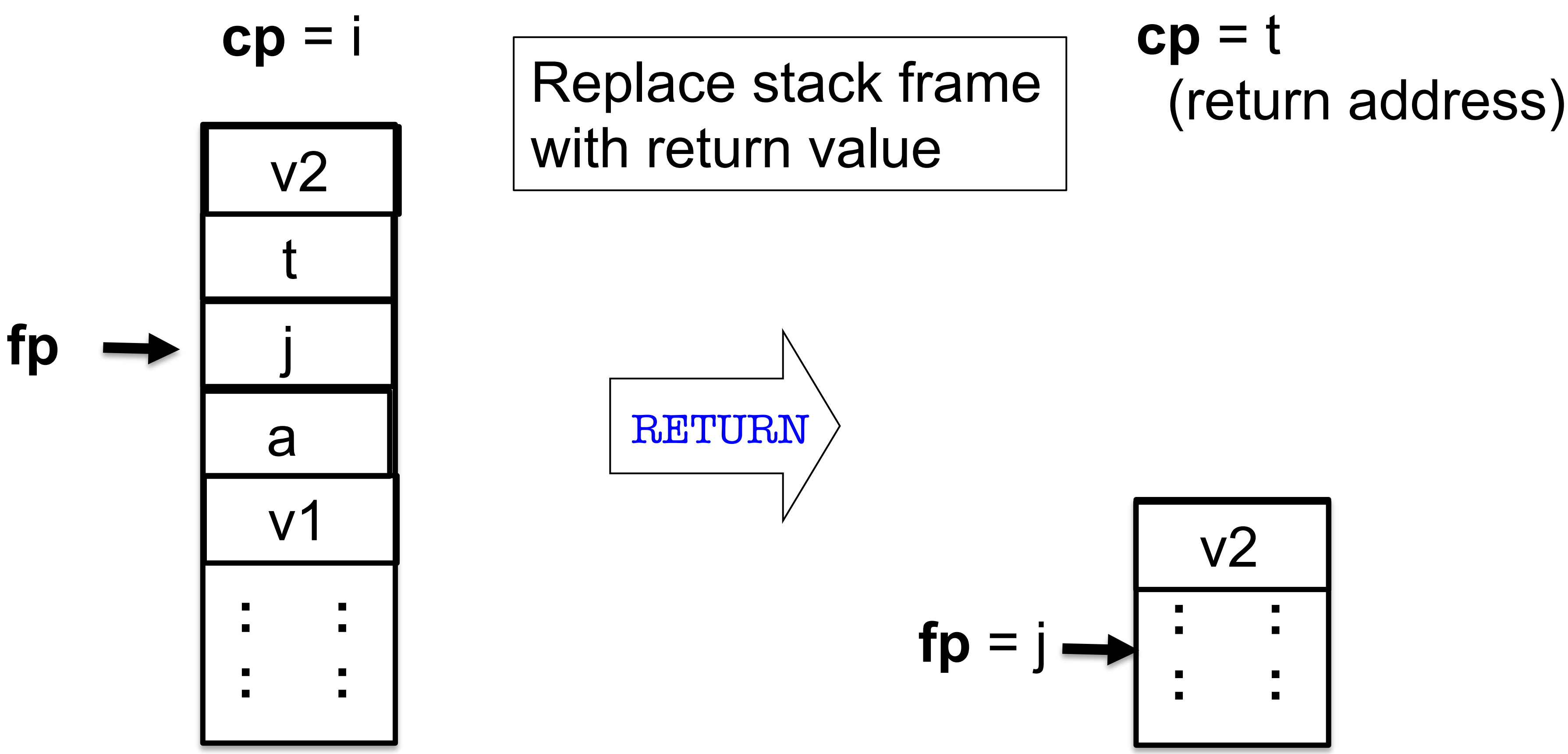


RETURN

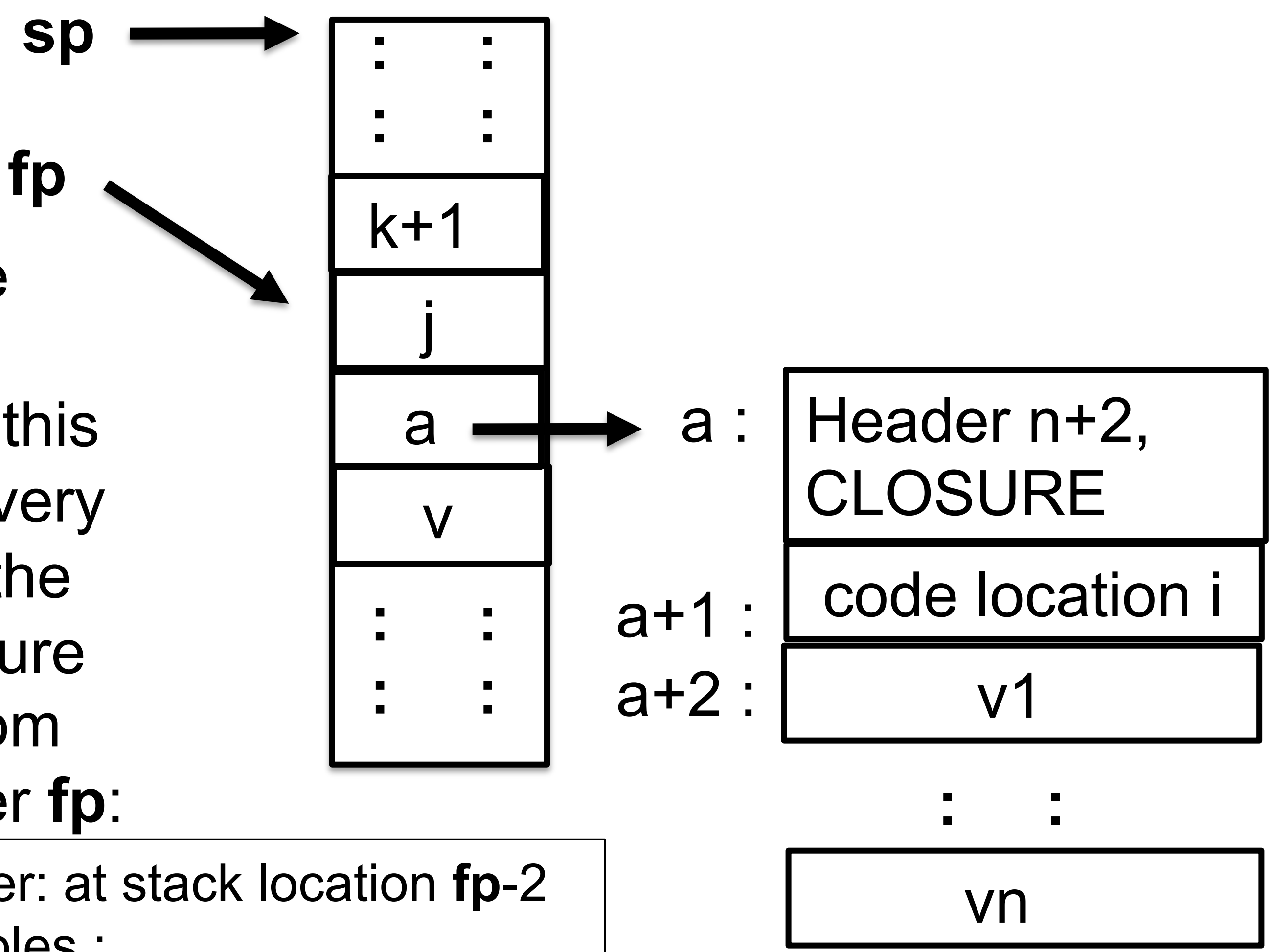
Interpreter 3:

(RETURN, (V v) :: _ :: (RA i) :: evs) -> (i, (V v) :: evs)

BEFORE Jargon VM: AFTER



Finding a variable's value at runtime



Suppose we are executing code associated with this closure. Then every free variable in the body of the closure can be found from the frame pointer **fp**:

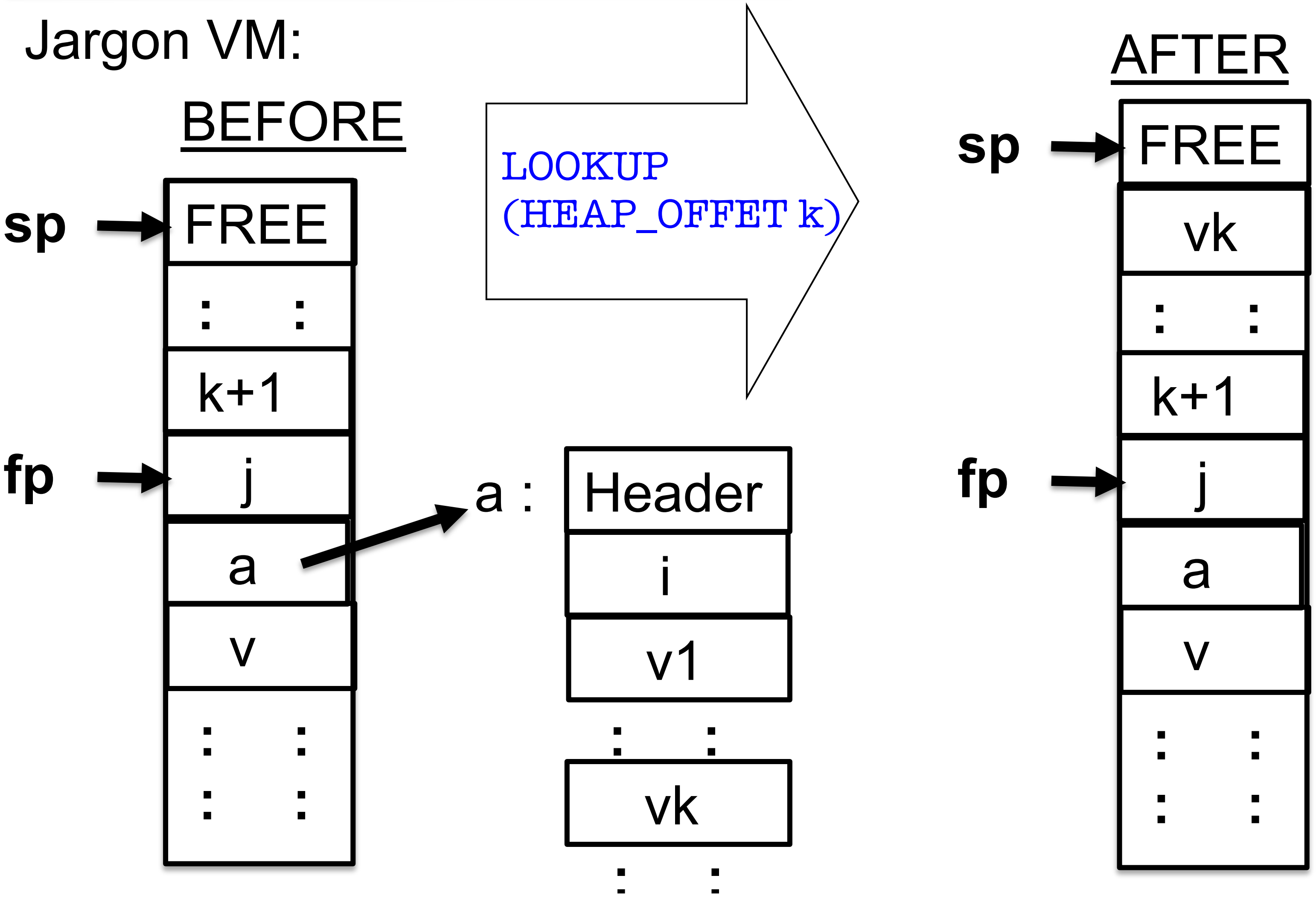
- Formal parameter: at stack location **fp**-2
- Other free variables :
    - Follow heap pointer found at **fp** -1
    - Each free variable can be associated with a fixed offset from this heap address

LOOKUP (HEAP_OFFSET k)

Interpreter 3:

(LOOKUP x, evs) -> (cp + 1, V(search(evs, x)) :: evs)

Jargon VM:

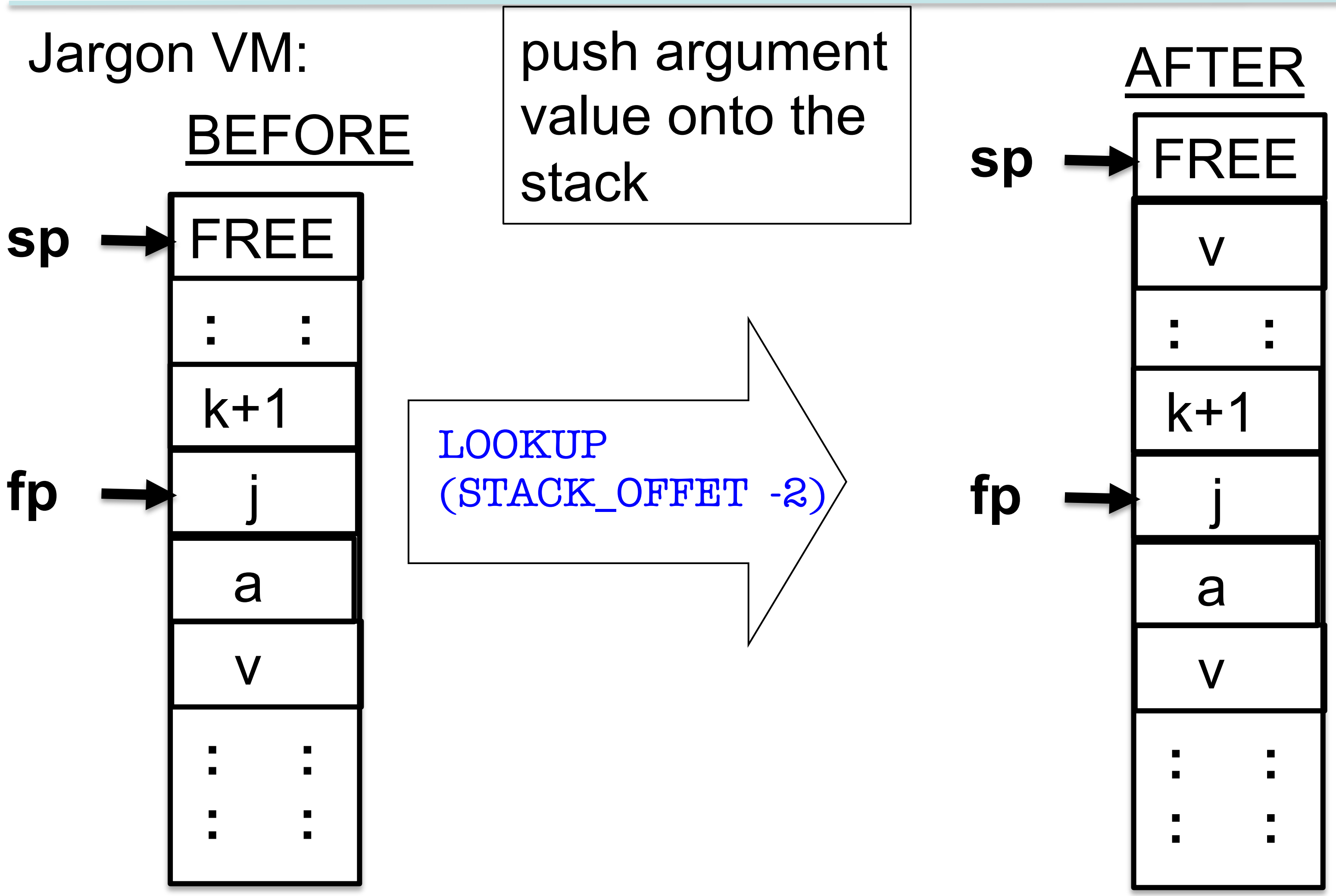


LOOKUP (STACK_OFFSET -2)

Interpreter 3:

(LOOKUP x, evs) -> (cp + 1, V(search(evs, x)) :: evs)

Jargon VM:



Oh, one problem

interpreter 3

```
let rec comp = function
 :
 | LetFun(f, (x, e1), e2) ->
             let (defs1, c1) = comp e1 in
             let (defs2, c2) = comp e2 in
             let def = [LABEL f; BIND x] @ c1 @ [SWAP; POP; RETURN] in
                (def @ defs1 @ defs2,
                 [MK_CLOSURE((f, None)); BIND f] @ c2 @ [SWAP; POP])
 :
```

Problem: Code c2 can be anything --- how are we going to find the closure for f when we need it? It has to be a fixed offset from a frame pointer --- we no longer scan the stack for bindings!

Solution in Jargon VM

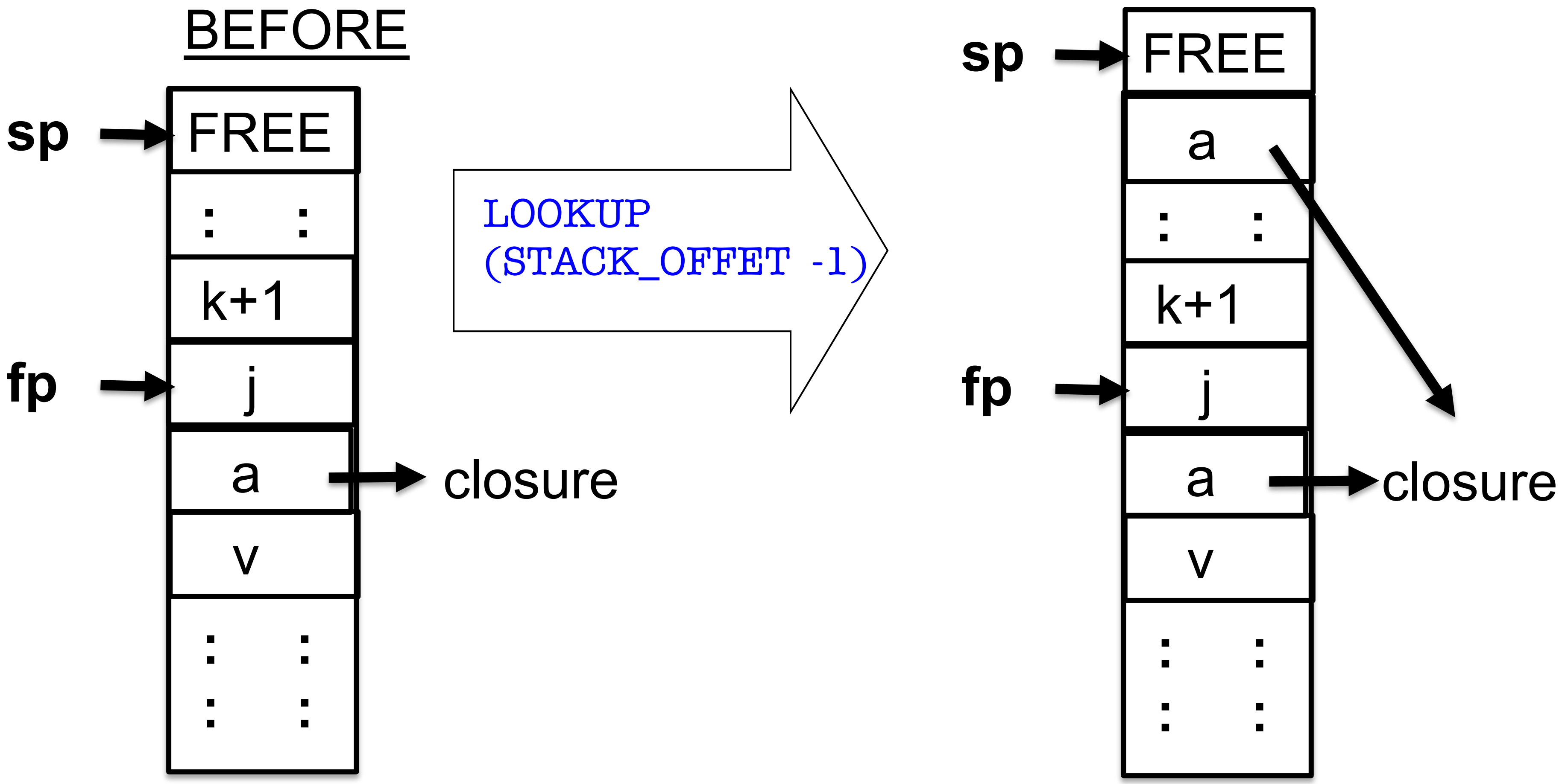
```
let rec comp vmap = function
 :
| LetFun(f, (x, e1), e2) -> comp vmap (App(Lambda(f, e2), Lambda(x, e1)))
:
```

Similar trick for LetRecFun

LOOKUP (STACK_OFFSET -1)

For recursive function calls,
push current closure on to the stack.

Jargon VM:



Example : Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int  = (snd p, fst p)
in
   rev_pair (21, 17)
end
```

After the front-end, compile treats this as follows.

```
App(
  Lambda(
    "rev_pair",
     App(Var "rev_pair",  Pair (Integer 21, Integer 17))),
  Lambda("p", Pair(Snd (Var "p"), Fst (Var "p"))))
```

Example : Compiled code for rev_pair.slang

```
App(
  Lambda("rev_pair",
            App(Var "rev_pair",  Pair (Integer 21, Integer 17))),
  Lambda("p", Pair(Snd (Var "p"), Fst (Var "p"))))
```

"first lambda"

"second lambda'

```
          MK_CLOSURE(L1, 0)              -- Make closure for second lambda
          MK_CLOSURE(L0, 0)              -- Make closure for first lambda
          APPLY                          -- do application
          HALT                           -- the end!
L0 :      PUSH STACK_INT 21              -- code for first lambda, push 21
          PUSH STACK_INT 17              -- push 17
          MK_PAIR                        -- make the pair on the heap
          LOOKUP STACK_LOCATION -2       -- push closure for second lambda on stack
          APPLY                          -- apply first lambda
          RETURN                         -- return from first lambda
L1 :      LOOKUP STACK_LOCATION -2       -- code for second lambda, push arg on stack
          SND                            -- extract second part of pair
          LOOKUP STACK_LOCATION -2       -- push arg on stack again
          FST                            -- extract first part of pair
          MK_PAIR                        -- construct a new pair
          RETURN                         -- return from second lambda
```

Example : trace of rev_pair.slang execution

```
Installed Code =
0: MK_CLOSURE(L1 = 11, 0)
1: MK_CLOSURE(L0 = 4, 0)
2: APPLY
3: HALT
4: LABEL L0
5: PUSH STACK_INT 21
6: PUSH STACK_INT 17
7: MK_PAIR
8: LOOKUP STACK_LOCATION-2
9: APPLY
10: RETURN
11: LABEL L1
12: LOOKUP STACK_LOCATION-2
13: SND
14: LOOKUP STACK_LOCATION-2
15: FST
16: MK_PAIR
17: RETURN
```

```
========== state 1 ==========
cp = 0 -> MK_CLOSURE(L1 = 11, 0)
fp = 0
Stack =
1: STACK_RA 0
0: STACK_FP 0

========== state 2 ==========
cp = 1 -> MK_CLOSURE(L0 = 4, 0)
fp = 0
Stack =
2: STACK_HI 0
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11

......
```

Example : trace of rev_pair.slang execution

```
========== state 15 ==========
cp = 16 -> MK_PAIR
fp = 8
Stack =
11: STACK_INT 21
10: STACK_INT 17
9: STACK_RA 10
8: STACK_FP 4
7: STACK_HI 0
6: STACK_HI 4
5: STACK_RA 3
4: STACK_FP 0
3: STACK_HI 2
2: STACK_HI 0
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11
2 -> HEAP_HEADER(2, HT_CLOSURE)
3 -> HEAP_CI 4
4 -> HEAP_HEADER(3, HT_PAIR)
5 -> HEAP_INT 21
6 -> HEAP_INT 17
```

```
========== state 19 ==========
cp = 3 -> HALT
fp = 0
Stack =
2: STACK_HI 7
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11
2 -> HEAP_HEADER(2, HT_CLOSURE)
3 -> HEAP_CI 4
4 -> HEAP_HEADER(3, HT_PAIR)
5 -> HEAP_INT 21
6 -> HEAP_INT 17
7 -> HEAP_HEADER(3, HT_PAIR)
8 -> HEAP_INT 17
9 -> HEAP_INT 21


Jargon VM :
output> (17, 21)
```

Example : closure_add.slang

```
let f(y : int) : int -> int = let g(x :int) : int = y + x  in g end
in let add21 : int -> int  = f(21)
    in let add17 : int -> int  = f(17)
        in add17(3) + add21(10)
        end
   end
end
```

Note : we really do need closures on the heap here — the values 21 and 17 do not exist on the stack at this point in the execution.

After the front-end, this becomes represented as follows.

```
App(Lambda(f, App(Lambda(add21,
                         App(Lambda(add17,
                                    Op(App(Var(add17), Integer(3)),
                                       ADD,
                                       App(Var(add21), Integer(10)))),
                             App(Var(f), Integer(17)))),
                  App(Var(f), Integer(21))))),
    Lambda(y, App(Lambda(g, Var(g)), Lambda(x, Op(Var(y), ADD, Var(x))))))
```

Can we make sense of this?

```
        MK_CLOSURE(L3, 0)
        MK_CLOSURE(L0, 0)
        APPLY
        HALT
L0 :    PUSH STACK_INT 21
        LOOKUP STACK_LOCATION -2
        APPLY
        LOOKUP STACK_LOCATION -2
        MK_CLOSURE(L1, 1)
        APPLY
        RETURN
L1 :    PUSH STACK_INT 17
        LOOKUP HEAP_LOCATION 1
        APPLY
        LOOKUP STACK_LOCATION -2
        MK_CLOSURE(L2, 1)
        APPLY
        RETURN
L2 :    PUSH STACK_INT 3
        LOOKUP STACK_LOCATION -2
        APPLY
        PUSH STACK_INT 10
        LOOKUP HEAP_LOCATION 1
        APPLY
        OPER ADD
        RETURN
L3 :    LOOKUP STACK_LOCATION -2
        MK_CLOSURE(L5, 1)
        MK_CLOSURE(L4, 0)
        APPLY
        RETURN
L4 :    LOOKUP STACK_LOCATION -2
        RETURN
L5 :    LOOKUP HEAP_LOCATION 1
        LOOKUP STACK_LOCATION -2
        OPER ADD
        RETURN
```

The Gap, illustrated

fib.slang

```
let fib (m :int) : int  =
   if m = 0
   then 1
   else if m = 1
         then 1
         else fib(m - 1) + fib (m - 2)
         end
    end
in fib (?) end
```

slang.byte –c –i4 fib.slang

```
            MK_CLOSURE(fib, 0)
            MK_CLOSURE(L0, 0)
            APPLY
            HALT
L0 :        PUSH STACK_UNIT
            UNARY READ
            LOOKUP STACK_LOCATION -2
            APPLY
            RETURN
fib :       LOOKUP STACK_LOCATION -2
            PUSH STACK_INT 0
            OPER EQI
            TEST L1
            PUSH STACK_INT 1
            GOTO L2
L1 :        LOOKUP STACK_LOCATION -2
            PUSH STACK_INT 1
            OPER EQI
            TEST L3
            PUSH STACK_INT 1
            GOTO L4
L3 :        LOOKUP STACK_LOCATION -2
            PUSH STACK_INT 1
            OPER SUB
            LOOKUP STACK_LOCATION -1
            APPLY
            LOOKUP STACK_LOCATION -2
            PUSH STACK_INT 2
            OPER SUB
            LOOKUP STACK_LOCATION -1
            APPLY
            OPER ADD
L4 :
L2 :        RETURN
```

Jargon VM code

Remarks

1. The semantic GAP between a Slang/L3 program and a low-level translation (say x86/Unix) has been significantly reduced.
2. Implementing the Jargon VM at a lower-level of abstraction (in C?, JVM bytecodes? X86/Unix? …) looks like a relatively easy programming problem.
3. However, using a lower-level implementation (say x86, exploiting fast registers) to generate very efficient code is not so easy. See Part II Optimising Compilers.

Verification of compilers is an active area of research. See CompCert, CakeML, and DeepSpec.

What about languages other than Slang/L3?

- Many textbooks on compilers treat only languages with first-order functions --- that is, functions cannot be passes as an argument or returned as a result. In this case, we can avoid allocating environments on the heap since all values associated with free variables will be somewhere on the stack!
- But how do we find these values? We optimise stack search by following a chain of **static links**. Static links are added to every stack frame and the point to the stack frame of the last invocation of the defining function.
- One other thing: most languages take multiple arguments for a function/procedure call.

Terminology: Caller and Callee

```
fun f (x, y) = e1

…

fun g(w, v) =
    w + f(v, v)
```

For this invocation of the function f, we say that g is the <u>caller</u> while f is the callee

Recursive functions can play both roles at the same time …

Nesting depth

Pseudo-code

```
fun b(z) = e

 fun g(x1) =
   fun h(x2) =
     fun f(x3) = e3(x1, x2, x3, b, g h, f)
     in
        e2(x1, x2, b, g, h, f)
      end
   in
      e1(x1, b, g, h)
  end
...
b(g(17))
...
```

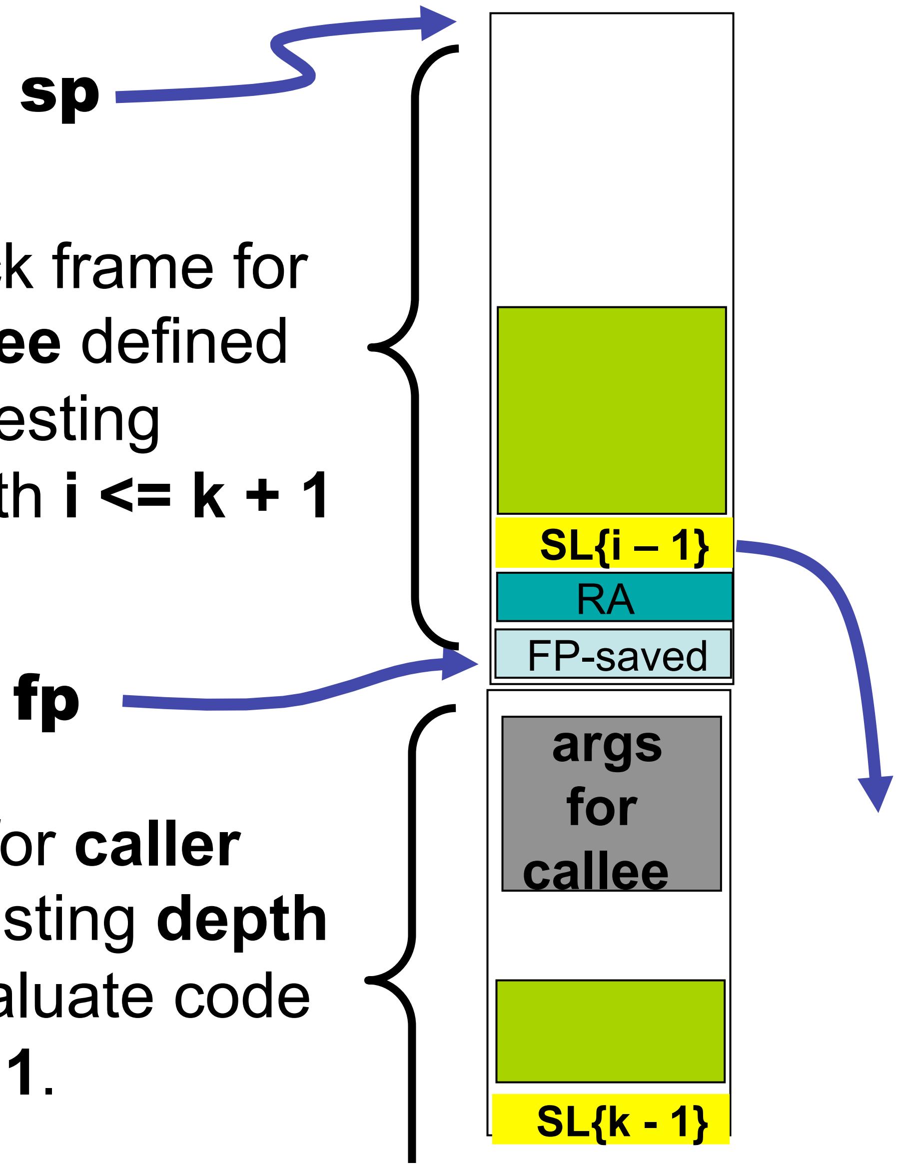
Nesting depth

code in big box is at nesting depth k

```
fun b(z) = e    nesting depth k + 1

 fun g(x1) =
   fun h(x2) =
     fun f(x3) = e3(x1, x2, x3, b, g h, f)      nesting depth k + 3
     in
        e2(x1, x2, b, g, h, f)
      end                                        nesting depth k + 2
   in
      e1(x1, b, g, h)
  end                                            nesting depth k + 1
...
b(g(17))
...
```


Function g is the **definer** of h. Functions g and b must share a definer defined at depth k-1

Stack with static links and variable number of arguments

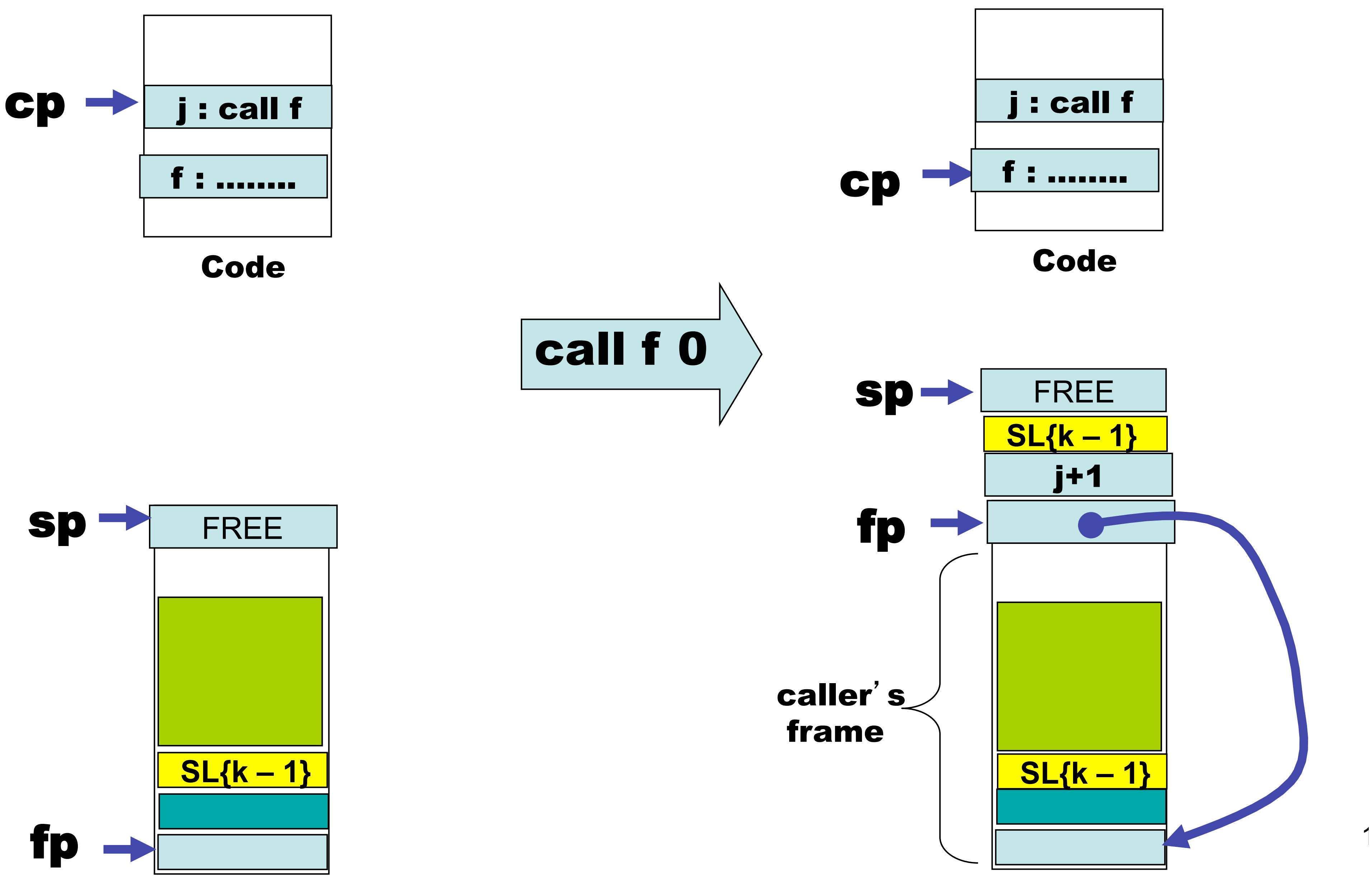


stack frame for **callee** defined at nesting depth **i <= k + 1**

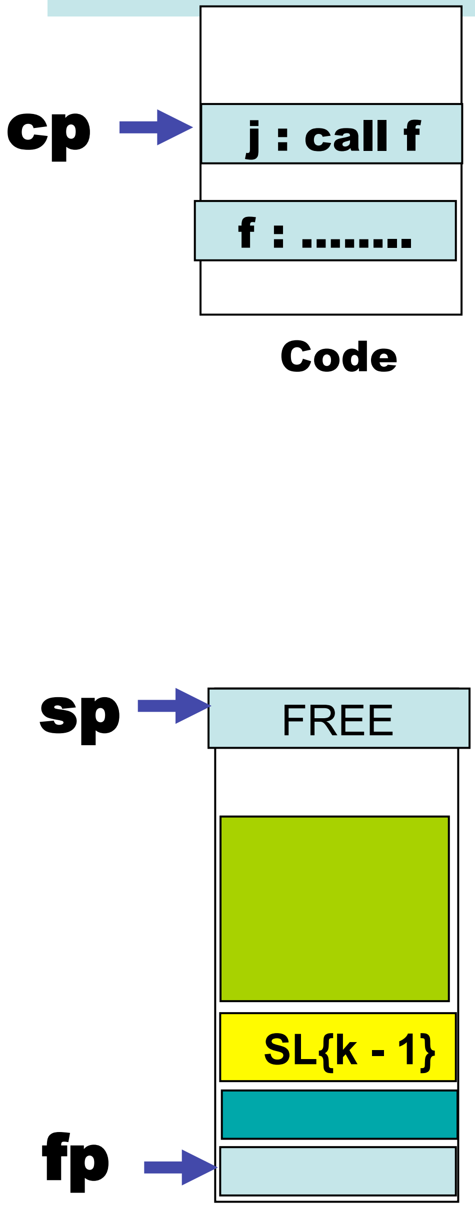
stack frame for **caller** defined at nesting **depth k** used to evaluate code at **depth k + 1**.

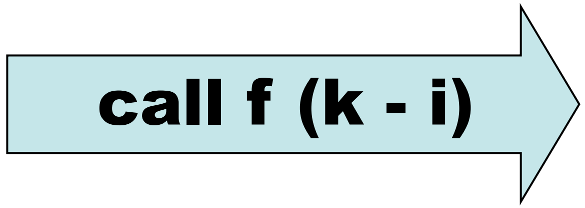
The static link points down to the closest frame **of definer** at nesting depth **i - 1**

caller and callee at same nesting depth k

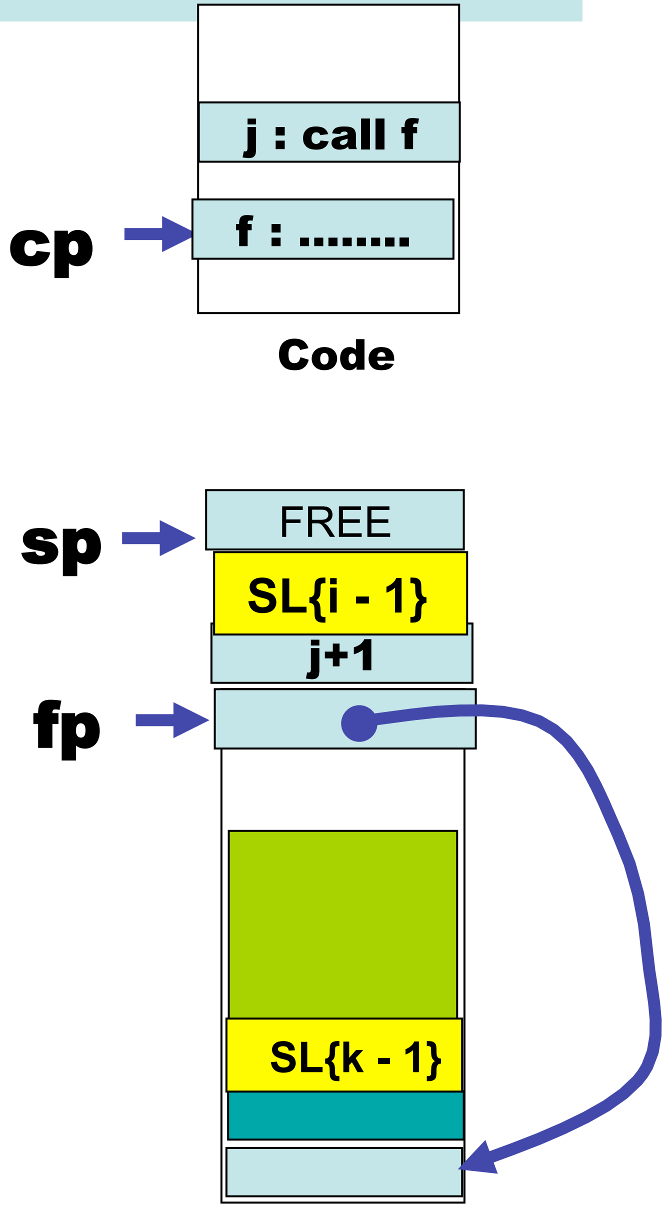


caller at depth k and callee at depth i < k

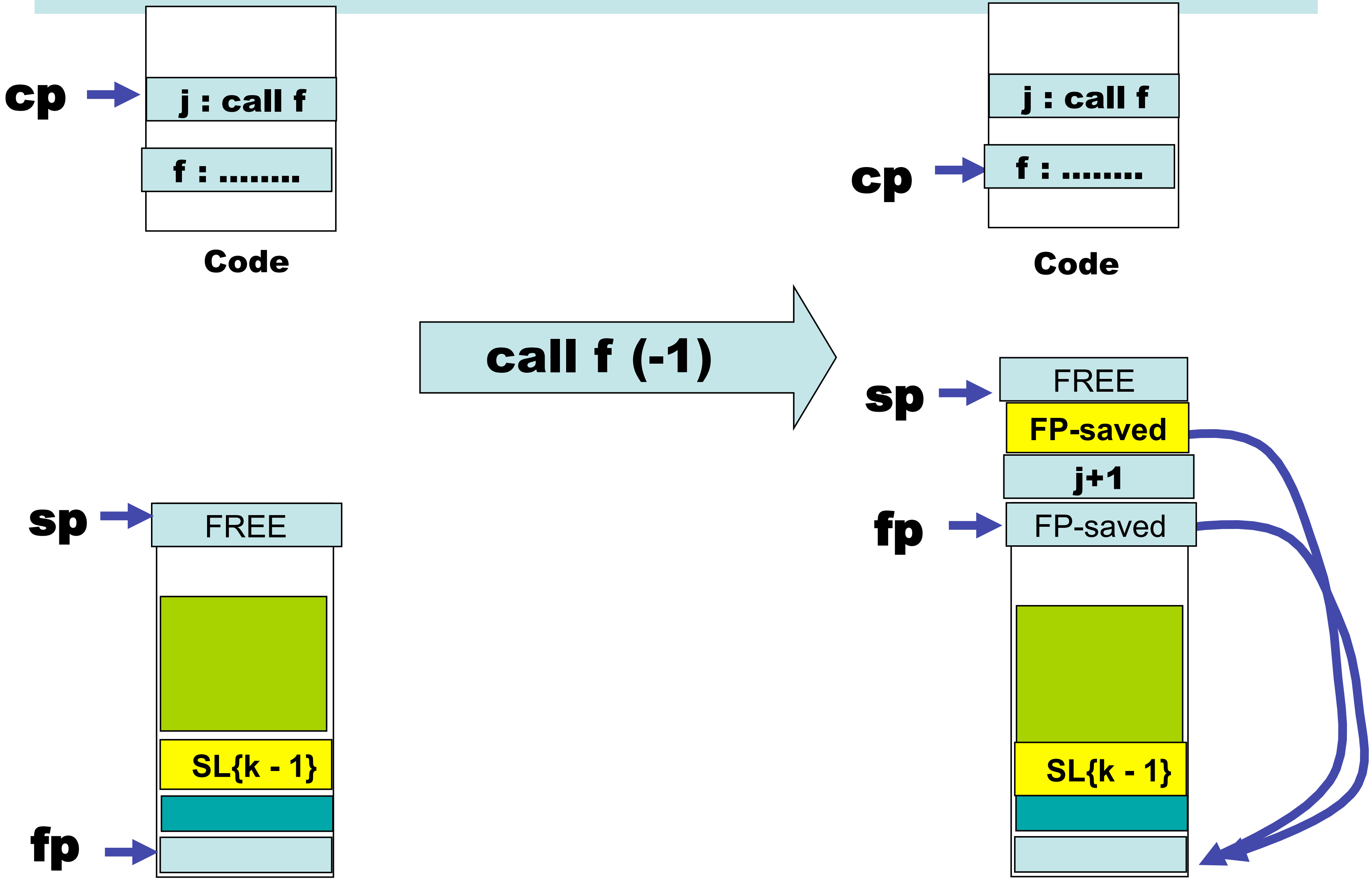




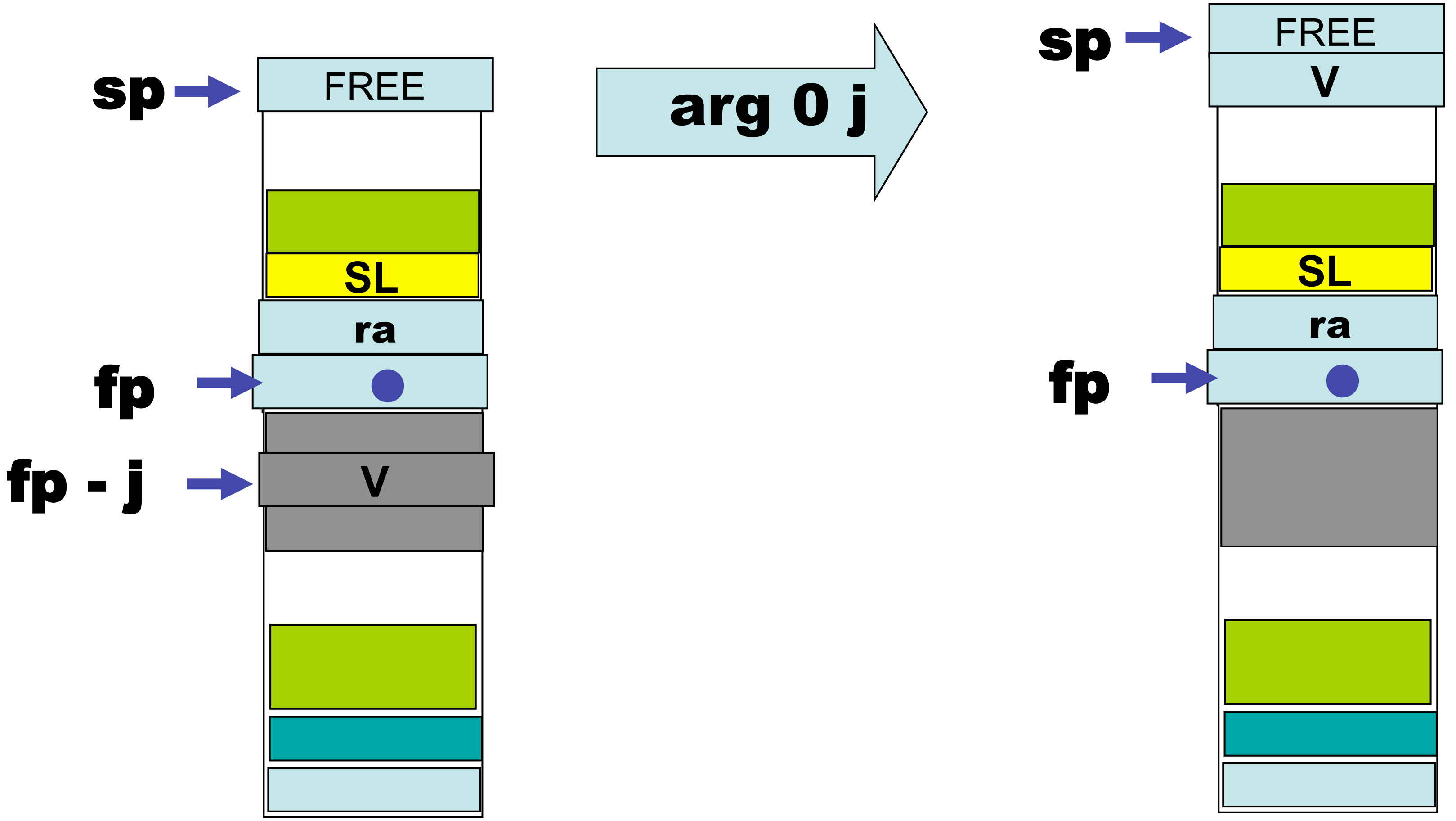

```
p := !(fp + 2);
for c = 1 to k - i
{
    p := !(p + 2);
}
SL{i-1} := p;
```



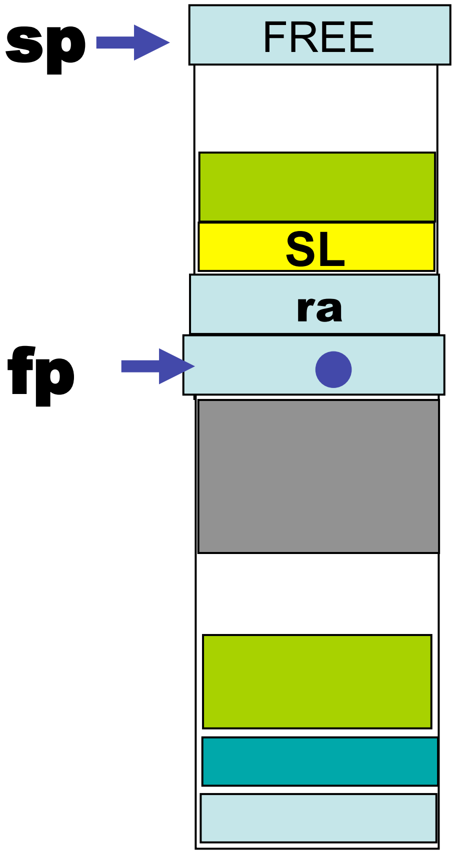
caller at depth k and callee at depth k + 1

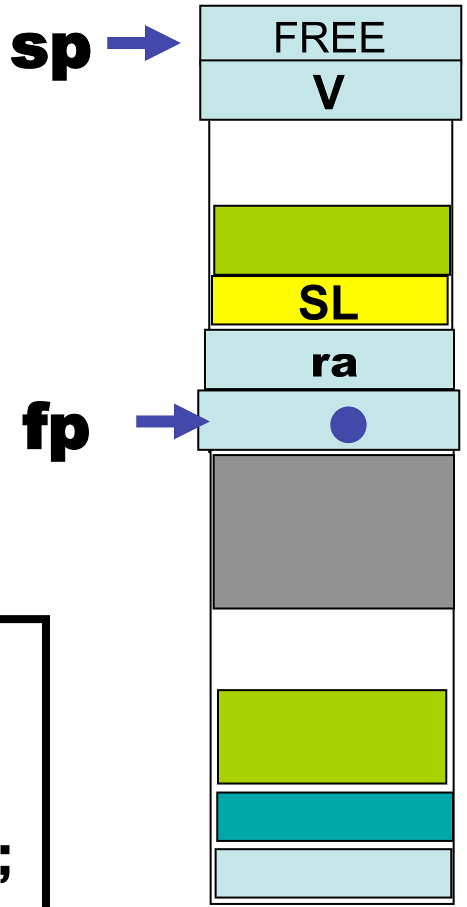


Access to argument values at static distance 0



Access to argument values at static distance d, 0 < d






```
p := !(fp + 2);
for c = 1 to d
{
   p := !(p + 2);
}
v := !(p - j);
```

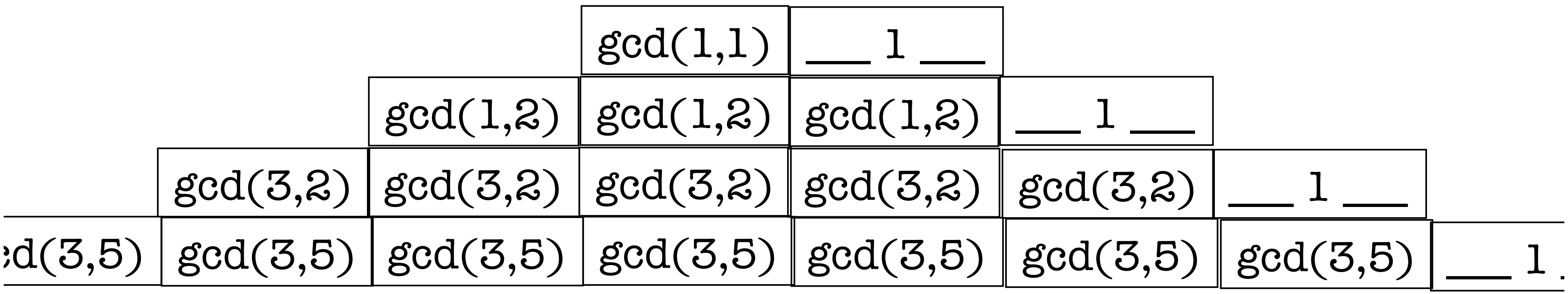
LECTUREs 11, 12
What about Interpreter 1?

- Evaluation using a stack
- Recursion using a stack
- Tail recursion elimination: from recursion to iteration
- Continuation Passing Style (CPS) : transform any recursive function to a tail-recursive function
- "Defunctionalisation" (DFC) : replace higher-order functions with a data structure
- Putting it all together:
  - Derive the Fibonacci Machine
  - Derive the Expression Machine, and "compiler"!
- This provides a roadmap for the interp_0 → interp_1 → interp_2 derivations.

Example of tail-recursion : gcd

```
(* gcd : int * int -> int *)
let rec gcd(m, n) =
   if m = n
   then m
   else if m < n
      then gcd(m,     n - m)
      else  gcd(m - n,      n)
```

Compared to fib, this function uses recursion in a different way. It is **tail-recursive**. If implemented with a stack, then the "call stack" (at least with respect to gcd) will simply grow and then shrink. No "ups and downs" in between.



Tail-recursive code can be replaced by iterative code that does not require a "call stack" (constant space)

gcd_iter : gcd without recursion!

```
(* gcd : int * int -> int *)
let rec gcd(m, n) =
   if m = n
   then m
   else if m < n
      then gcd(m,     n - m)
      else  gcd(m - n,     n)
```

Here we have illustrated tail-recursion elimination as a source-to-source transformation. However, the OCaml compiler will do something similar to a lower-level intermediate representation. Upshot : we will consider all tail-recursive OCaml functions as representing <u>iterative</u> programs.

```
(* gcd_iter : int * int -> int *)
let gcd_iter (m, n) =
   let rm = ref m
   in let rn = ref n
   in let result = ref 0
   in let not_done = ref true
   in let _ =
      while !not_done
       do
          if !rm = !rn
          then (not_done := false;
                 result := !rm)
          else if !rm < !rn
               then rn := !rn - !rm
               else rm := !rm - !rn
       done
   in !result
```

Familiar examples : fold_left, fold_right

From ocaml-4.01.0/stdlib/list.ml :

```
(* fold_left :  ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

   fold_left f a [b1; ...; bn]]  = f (... (f (f a b1) b2) ...) bn
*)
let rec fold_left f a l =
 match l with
  | []          -> a
  | b :: rest -> fold_left f (f a b) rest

(* fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

   fold_right f [a1; ...; an] b = f a1 (f a2 (... (f an b) ...))
 *)
let rec fold_right f l b =
 match l with
  | []         -> b
  | a::rest -> f a (fold_right f rest b)
```

This is tail recursive

This is NOT tail recursive

Question: can we transform any recursive function into a tail recursive function?

The answer is YES!

- **We add an extra argument, called a *continuation*, that represents "the rest of the computation"**
- **This is called the Continuation Passing Style (CPS) transformation.**
- **We will then "defunctionalize" (DFC) these continuations and represent them with a stack.**
- **Finally, we obtain a tail recursive function that carries its own stack as an extra argument!**

We will apply this kind of transformation to <u>the code of interpreter 0</u> as the first steps towards deriving interpreter 1.

(CPS) transformation of fib

```
(* fib : int -> int *)
let rec fib m =
   if m = 0
   then 1
   else if m = 1
          then 1
          else fib(m - 1) + fib (m - 2)

(* fib_cps : int * (int -> int)  -> int *)
 let rec fib_cps (m,  cnt) =
   if m = 0
   then cnt 1
   else if m = 1
         then cnt 1
         else fib_cps(m -1,  fun a -> fib_cps(m - 2 , fun b  -> cnt (a + b)))
```

A closer look

The rest of the computation after computing "fib(m)". That is, cnt is a function expecting the result of "fib(m)" as its argument.

```
let rec fib_cps (m,  cnt) =
   if m = 0
   then cnt 1
   else if m = 1
        then cnt 1
        else fib_cps(m -1,  fun a -> fib_cps(m - 2 , fun b  -> cnt (a + b)))
```

The computation waiting for the result of "fib(m-1)"

The computation waiting for the result of "fib(m-2)"

This makes explicit the order of evaluation that is implicit in the original "fib(m-1) + fib(m-2)" :
-- first compute fib(m-1)
-- then compute fib(m-1)
-- then add results together
-- then return

Expressed with “let” rather than “fun”

```
(* fib_cps_v2 : (int -> int) * int -> int *)
let rec fib_cps_v2 (m, cnt) =
   if m = 0
   then cnt 1
   else if m = 1
         then cnt 1
         else let cnt2 a b = cnt (a + b)
              in let cnt1 a = fib_cps_v2(m - 2, cnt2 a)
              in fib_cps_v2(m - 1, cnt1)
```

Some prefer writing CPS forms without explicit funs ….

Use the identity continuation …

```
(* fib_cps : int * (int -> int)  -> int *)
 let rec fib_cps (m, cnt) =
   if m = 0
   then cnt 1
   else if m = 1
        then cnt 1
        else fib_cps(m -1,  fun a -> fib_cps(m - 2 , fun b  -> cnt (a + b)))


let id (x : int) = x

let fib_1 x = fib_cps(x, id)
```

```
List.map fib_1  [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;

  =  [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

Correctness?

NB: This proof pretends that we can treat OCaml functions as ideal mathematical functions, which of course we cannot. OCaml functions might raise exceptions like "stack overflow" or "you burned my toast", and so on. But this is a convenient fiction as long as we remember to be careful.

For all c : int -> int, for all m, 0 <= m, we have, c(fib m) = fib_cps(m, c).

Proof: assume c : int -> int. By Induction on m. Base case : m = 0:

fib_cps(0, c) = c(1) = c(fib(0).

Induction step: Assume for all n < m, c(fib n) = fib_cps(n, c).
(That is, we need course-of-values induction!)

```
fib_cps(m + 1, c)
= if m + 1 = 1
  then c 1
  else fib_cps((m+1) -1, fun a -> fib_cps((m+1) -2, fun b -> c (a + b)))
= if m + 1 = 1
  then c 1
  else fib_cps(m, fun a -> fib_cps(m-1, fun b -> c (a + b)))
= (by induction)
  if m + 1 = 1
  then c 1
  else (fun a -> fib_cps(m -1, fun b -> c (a + b))) (fib m)
```

Correctness?

```
= if m + 1 = 1
  then c 1
  else fib_cps(m-1, fun b -> c ((fib m) + b))
= (by induction)
  if m + 1 = 1
  then c 1
  else (fun b -> c ((fib m) + b)) (fib (m-1))
 = if m + 1 = 1
  then c 1
  else c ((fib m) + (fib (m-1)))
 = c (if m + 1 = 1
     then 1
     else ((fib m) + (fib (m-1))))
 = c(if m +1 = 1
     then 1
     else fib((m + 1) - 1) + fib ((m + 1) - 2))
 = c (fib(m + 1))
```


QED.

Can with express fib_cps without a functional argument ?

```
(* fib_cps_v2 : (int -> int) * int -> int *)
let rec fib_cps_v2 (m, cnt) =
   if m = 0
   then cnt 1
   else if m = 1
        then cnt 1
        else let cnt2 a b = cnt (a + b)
             in let cnt1 a = fib_cps_v2(m - 2, cnt2 a)
             in fib_cps_v2(m - 1, cnt1)
```

Idea of "defunctonalisation" (DFC): replace id, cnt1 and cnt2 with instances of a new data type:

```
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt
```

Now we need an "apply" function of type `cnt * int -> int`

"Defunctionalised" version of fib_cps

```
(* datatype to represent continuations *)
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt

(* apply_cnt : cnt * int -> int *)
let rec apply_cnt = function
  | (ID, a)                -> a
  | (CNT1 (m, cnt), a) -> fib_cps_dfc(m - 2, CNT2 (a, cnt))
  | (CNT2 (a, cnt), b)  -> apply_cnt (cnt, a + b)

(*  fib_cps_dfc : (cnt * int) -> int *)
and fib_cps_dfc (m, cnt) =
   if m = 0
   then apply_cnt(cnt, 1)
   else if m = 1
         then apply_cnt(cnt, 1)
         else fib_cps_dfc(m -1, CNT1(m, cnt))

(*  fib_2 : int -> int *)
let fib_2 m = fib_cps_dfc(m, ID)
```

Correctness?

Let < c > be of type cnt representing
a continuation c : int -> int constructed by fib_cps.

Then
    apply_cnt(< c >, m) = c(m)
and
    fib_cps(n, c) = fib_cps_dfc(n, < c >).

Proof left as an exercise!

Functional continuation c	Representation < c >
fun a -> fib_cps(m - 2 , fun b -> cnt (a + b))	CNT1(m, < cnt >)
fun b -> cnt (a + b)	CNT2(a, < cnt >)
fun x -> x	ID

Eureka! Continuations are just lists (used like a stack)

```
type int_list = NIL | CONS of int * int_list
```

```
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt
```

Think nil

Think cons type1

Think cons type2

Replace the above continuations with lists! (I’ve selected more suggestive names for the constructors.)

```
type tag = SUB2 of int | PLUS of int

type tag_list_cnt = tag list
```

The continuation lists are used like a stack!

```
type tag = SUB2 of int | PLUS of int
type tag_list_cnt = tag list

(* apply_tag_list_cnt : tag_list_cnt * int -> int *)
let rec apply_tag_list_cnt = function
  | ([], a)                  -> a
  | ((SUB2 m) :: cnt, a) -> fib_cps_dfc_tags(m - 2, (PLUS a):: cnt)
  | ((PLUS a) :: cnt, b)  -> apply_tag_list_cnt (cnt, a + b)

(* fib_cps_dfc_tags : (tag_list_cnt * int) -> int *)
and fib_cps_dfc_tags (m, cnt) =
   if m = 0
   then apply_tag_list_cnt(cnt, 1)
   else if m = 1
         then apply_tag_list_cnt(cnt, 1)
         else fib_cps_dfc_tags(m - 1, (SUB2 m) :: cnt)

(*  fib_3 : int -> int *)
let fib_3 m = fib_cps_dfc_tags(m, [])
```

Combine Mutually tail-recursive functions into a single function

```
type state_type =
  | SUB1 (* for right-hand-sides starting with fib_   *)
  | APPL  (* for right-hand-sides starting with apply_ *)

type state = (state_type * int * tag_list_cnt) -> int

(* eval : state -> int              A two-state transition function*)
let rec eval = function
  | (SUB1, 0,                cnt) -> eval (APPL, 1,                 cnt)
  | (SUB1, 1,                cnt) -> eval (APPL, 1,                 cnt)
  | (SUB1, m,                cnt) -> eval (SUB1, (m-1), (SUB2 m) :: cnt)
  | (APPL, a, (SUB2 m) :: cnt) -> eval (SUB1, (m-2), (PLUS a) :: cnt)
  | (APPL, b,  (PLUS a) :: cnt) -> eval (APPL, (a+b),               cnt)
  | (APPL, a,                  [])  -> a

(*  fib_4 : int -> int *)
let fib_4 m = eval (SUB1, m, [])
```

Eliminate tail recursion to obtain The Fibonacci Machine!

```
(* step : state -> state *)
let step = function
  | (SUB1, 0,                cnt) -> (APPL, 1,                    cnt)
  | (SUB1, 1,                cnt) -> (APPL, 1,                    cnt)
  | (SUB1, m,                cnt) -> (SUB1, (m-1), (SUB2 m) :: cnt)
  | (APPL, a, (SUB2 m) :: cnt) -> (SUB1, (m-2),  (PLUS a) :: cnt)
  | (APPL, b,  (PLUS a) :: cnt) -> (APPL, (a+b),                  cnt)
  | _ -> failwith "step : runtime error!"

(* clearly TAIL RECURSIVE! *)
let rec driver state = function
   | (APPL, a, []) -> a
   |  state          -> driver (step state)

(*  fib_5 : int -> int *)
let fib_5 m = driver  (SUB1, m, [])
```

In this version we have simply made the tail-recursive structure very explicit.

Here is a trace of fib_5 6.

```
1 SUB1 || 6 || []
2 SUB1 || 5 || [SUB2 6]
3 SUB1 || 4 || [SUB2 6, SUB2 5]
4 SUB1 || 3 || [SUB2 6, SUB2 5, SUB2 4]
5 SUB1 || 2 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3]
6 SUB1 || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, SUB2 2]
7 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, SUB2 2]
8 SUB1 || 0 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, PLUS 1]
9 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, PLUS 1]
10 APPL || 2 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3]
11 SUB1 || 1 || [SUB2 6, SUB2 5, SUB2 4, PLUS 2]
12 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, PLUS 2]
13 APPL || 3 || [SUB2 6, SUB2 5, SUB2 4]
14 SUB1 || 2 || [SUB2 6, SUB2 5, PLUS 3]
15 SUB1 || 1 || [SUB2 6, SUB2 5, PLUS 3, SUB2 2]
16 APPL || 1 || [SUB2 6, SUB2 5, PLUS 3, SUB2 2]
17 SUB1 || 0 || [SUB2 6, SUB2 5, PLUS 3, PLUS 1]
18 APPL || 1 || [SUB2 6, SUB2 5, PLUS 3, PLUS 1]
19 APPL || 2 || [SUB2 6, SUB2 5, PLUS 3]
20 APPL || 5 || [SUB2 6, SUB2 5]
21 SUB1 || 3 || [SUB2 6, PLUS 5]
22 SUB1 || 2 || [SUB2 6, PLUS 5, SUB2 3]
23 SUB1 || 1 || [SUB2 6, PLUS 5, SUB2 3, SUB2 2]
24 APPL || 1 || [SUB2 6, PLUS 5, SUB2 3, SUB2 2]
25 SUB1 || 0 || [SUB2 6, PLUS 5, SUB2 3, PLUS 1]
26 APPL || 1 || [SUB2 6, PLUS 5, SUB2 3, PLUS 1]
27 APPL || 2 || [SUB2 6, PLUS 5, SUB2 3]
28 SUB1 || 1 || [SUB2 6, PLUS 5, PLUS 2]
29 APPL || 1 || [SUB2 6, PLUS 5, PLUS 2]
30 APPL || 3 || [SUB2 6, PLUS 5]
31 APPL || 8 || [SUB2 6]
32 SUB1 || 4 || [PLUS 8]
33 SUB1 || 3 || [PLUS 8, SUB2 4]
34 SUB1 || 2 || [PLUS 8, SUB2 4, SUB2 3]
35 SUB1 || 1 || [PLUS 8, SUB2 4, SUB2 3, SUB2 2]
36 APPL || 1 || [PLUS 8, SUB2 4, SUB2 3, SUB2 2]
37 SUB1 || 0 || [PLUS 8, SUB2 4, SUB2 3, PLUS 1]
38 APPL || 1 || [PLUS 8, SUB2 4, SUB2 3, PLUS 1]
39 APPL || 2 || [PLUS 8, SUB2 4, SUB2 3]
40 SUB1 || 1 || [PLUS 8, SUB2 4, PLUS 2]
41 APPL || 1 || [PLUS 8, SUB2 4, PLUS 2]
42 APPL || 3 || [PLUS 8, SUB2 4]
43 SUB1 || 2 || [PLUS 8, PLUS 3]
44 SUB1 || 1 || [PLUS 8, PLUS 3, SUB2 2]
45 APPL || 1 || [PLUS 8, PLUS 3, SUB2 2]
46 SUB1 || 0 || [PLUS 8, PLUS 3, PLUS 1]
47 APPL || 1 || [PLUS 8, PLUS 3, PLUS 1]
48 APPL || 2 || [PLUS 8, PLUS 3]
49 APPL || 5 || [PLUS 8]
50 APPL ||13|| []
```

The OCaml file in basic_transformations/fibonacci_machine.ml contains some code for pretty printing such traces….

Pause to reflect

- **What have we accomplished?**
- **We have taken a recursive function and turned it into an iterative function that does not require "stack space" for its evaluation (in OCaml)**
- **However, this function now carries its own evaluation stack as an extra argument!**
- **We have derived this iterative function in a step-by-step manner where each tiny step is easily proved correct.**
- **Wow!**

That was fun! Let's do it again!

```
type expr =
   | INT of int
   | PLUS of expr * expr
   | SUBT of expr * expr
   | MULT of expr * expr
```

This time we will derive a stack-machine AND a "compiler" that translates expressions into a list of instructions for the machine.

```
(* eval : expr -> int
   a simple recusive evaluator for expressions *)
let rec eval = function
   | INT a            -> a
   | PLUS(e1, e2)   -> (eval e1) + (eval e2)
   | SUBT(e1, e2)   -> (eval e1) - (eval e2)
   | MULT(e1, e2)   -> (eval e1) * (eval e2)
```

Here we go again : CPS

```
type cnt_2  = int -> int

type state_2 = expr * cnt_2

(* eval_aux_2 : state_2 -> int *)
let rec eval_aux_2 (e, cnt) =
  match e with
   | INT a       -> cnt a
   | PLUS(e1, e2) ->
     eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 + v2)))
   | SUBT(e1, e2) ->
     eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 - v2)))
   | MULT(e1, e2) ->
     eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 * v2)))

 (* id_cnt : cnt_2 *)
let id_cnt (x : int) = x

(*  eval_2 : expr -> int *)
let eval_2 e = eval_aux_2(e, id_cnt)
```

Defunctionalise!

```
type cnt_3 =
  | ID
  | OUTER_PLUS of expr * cnt_3
  | OUTER_SUBT of expr * cnt_3
  | OUTER_MULT of expr * cnt_3
  | INNER_PLUS of int * cnt_3
  | INNER_SUBT of int * cnt_3
  | INNER_MULT of int * cnt_3

type state_3 = expr * cnt_3

(* apply_3 : cnt_3 * int -> int *)
let rec apply_3 = function
   | (ID,                  v)                    -> v
   | (OUTER_PLUS(e2, cnt), v1) -> eval_aux_3(e2, INNER_PLUS(v1, cnt))
   | (OUTER_SUBT(e2, cnt), v1) -> eval_aux_3(e2, INNER_SUBT(v1, cnt))
   | (OUTER_MULT(e2, cnt), v1) -> eval_aux_3(e2, INNER_MULT(v1, cnt)
   | (INNER_PLUS(v1, cnt), v2) -> apply_3(cnt, v1 + v2)
   | (INNER_SUBT(v1, cnt), v2) -> apply_3(cnt, v1 - v2)
   | (INNER_MULT(v1, cnt), v2) -> apply_3(cnt, v1 * v2)
```

Defunctionalise!

```
(* eval_aux_2 : state_3 -> int *)
and eval_aux_3 (e, cnt) =
  match e with
   | INT a        -> apply_3(cnt, a)
   | PLUS(e1, e2) -> eval_aux_3(e1, OUTER_PLUS(e2, cnt))
   | SUBT(e1, e2) -> eval_aux_3(e1, OUTER_SUBT(e2, cnt))
   | MULT(e1, e2) -> eval_aux_3(e1, OUTER_MULT(e2, cnt))

(* eval_3 : expr -> int *)
let eval_3 e = eval_aux_3(e, ID)
```

Eureka! Again we have a stack!

```
type tag =
  | O_PLUS of expr
  | I_PLUS of int
  | O_SUBT of expr
  | I_SUBT of int
  | O_MULT of expr
  | I_MULT of int

type cnt_4 = tag list
type state_4 = expr * cnt_4

(* apply_4 : cnt_4 * int -> int *)
let rec apply_4 = function
   | ([],          v)                 -> v
   | ((O_PLUS e2) :: cnt, v1) -> eval_aux_4(e2, (I_PLUS v1) :: cnt)
   | ((O_SUBT e2) :: cnt, v1) -> eval_aux_4(e2, (I_SUBT v1) :: cnt)
   | ((O_MULT e2) :: cnt, v1) -> eval_aux_4(e2, (I_MULT v1) :: cnt)
   | ((I_PLUS v1) :: cnt, v2) -> apply_4(cnt, v1 + v2)
   | ((I_SUBT v1) :: cnt, v2) -> apply_4(cnt, v1 - v2)
   | ((I_MULT v1) :: cnt, v2) -> apply_4(cnt, v1 * v2)
```

Eureka! Again we have a stack!

```
(* eval_aux_4 : state_4 -> int *)
and eval_aux_4 (e, cnt) =
  match e with
   | INT a            -> apply_4(cnt, a)
   | PLUS(e1, e2) -> eval_aux_4(e1, O_PLUS(e2) :: cnt)
   | SUBT(e1, e2) -> eval_aux_4(e1, O_SUBT(e2) :: cnt)
   | MULT(e1, e2) -> eval_aux_4(e1, O_MULT(e2) :: cnt)

(* eval_4 : expr -> int *)
let eval_4 e = eval_aux_4(e, [])
```

Eureka! Can combine apply_4 and eval_aux_4

```
type acc =
  | A_INT of int
  | A_EXP of expr

type cnt_5 = cnt_4

type state_5 = cnt_5 * acc

val : step : state_5 -> state_5

val driver : state_5 -> int

val eval_5 : expr -> int
```

Type of an "accumulator" that contains either an int or an expression.

The driver will be clearly tail-recursive …

Rewrite to use driver, accumulator

```
let step_5 = function
  | (cnt,              A_EXP (INT a)) -> (cnt, A_INT a)
  | (cnt,     A_EXP (PLUS(e1, e2))) -> (O_PLUS(e2) :: cnt, A_EXP e1)
  | (cnt,     A_EXP (SUBT(e1, e2))) -> (O_SUBT(e2) :: cnt, A_EXP e1)
  | (cnt,     A_EXP (MULT(e1, e2))) -> (O_MULT(e2) :: cnt, A_EXP e1)
  | ((O_PLUS e2) :: cnt,  A_INT v1) -> ((I_PLUS v1) :: cnt, A_EXP e2)
  | ((O_SUBT e2) :: cnt,  A_INT v1) -> ((I_SUBT v1) :: cnt, A_EXP e2)
  | ((O_MULT e2) :: cnt, A_INT v1) -> ((I_MULT v1) :: cnt, A_EXP e2)
  | ((I_PLUS v1) :: cnt,   A_INT v2) -> (cnt, A_INT (v1 + v2))
  | ((I_SUBT v1) :: cnt,   A_INT v2) -> (cnt, A_INT (v1 - v2))
  | ((I_MULT v1) :: cnt,   A_INT v2) -> (cnt, A_INT (v1  * v2))
  | ([],                        A_INT v) -> ([], A_INT v)

let rec driver_5 = function
  | ([], A_INT v) -> v
  | state         -> driver_5 (step_5 state)

let eval_5 e = driver_5([], A_EXP e)
```

Eureka! There are really two independent stacks here --- one for "expressions" and one for values

```
type directive =
  | E of expr
  | DO_PLUS
  | DO_SUBT
  | DO_MULT

type directive_stack = directive list

type value_stack = int list

type state_6 = directive_stack * value_stack

val step_6 : state_6 -> state_6

val driver_6 : state_6 -> int

val exp_6 : expr -> int
```

The state is now two stacks!

Split into two stacks

```
let step_6 = function
   | (E(INT v) :: ds,                vs) -> (ds, v :: vs)
   | (E(PLUS(e1, e2)) :: ds,    vs) -> ((E e1) :: (E e2) :: DO_PLUS :: ds, vs)
   | (E(SUBT(e1, e2)) :: ds,    vs) -> ((E e1) :: (E e2) :: DO_SUBT :: ds, vs)
   | (E(MULT(e1, e2)) :: ds,    vs) -> ((E e1) :: (E e2) :: DO_MULT :: ds, vs)

   | (DO_PLUS :: ds, v2 :: v1 :: vs) -> (ds, (v1 + v2) :: vs)
   | (DO_SUBT :: ds, v2 :: v1 :: vs) -> (ds, (v1 - v2) :: vs)
   | (DO_MULT :: ds, v2 :: v1 :: vs) -> (ds, (v1 * v2) :: vs)
   | _ -> failwith "eval : runtime error!"

let rec driver_6 = function
    | ([], [v]) -> v
    | state    -> driver_6 (step_6 state)

let eval_6 e = driver_6 ([E e], [])
```

An eval_6 trace

e = PLUS(MULT(INT 89, INT 2), SUBT(INT 10, INT 4))

inspect

```
state 1  DS = [E(PLUS(MULT(INT(89), INT(2)), SUBT(INT(10), INT(4))))]
         VS = []
state 2  DS = [DO_PLUS; E(SUBT(INT(10), INT(4))); E(MULT(INT(89), INT(2)))]
         VS = []
state 3  DS = [DO_PLUS; E(SUBT(INT(10), INT(4))); DO_MULT; E(INT(2)); E(INT(89))]
         VS = []
state 4  DS = [DO_PLUS; E(SUBT(INT(10), INT(4))); DO_MULT; E(INT(2))]
         VS = [89]
```

compute

```
state 5  DS = [DO_PLUS; E(SUBT(INT(10), INT(4))); DO_MULT]
         VS = [89; 2]
state 6  DS = [DO_PLUS; E(SUBT(INT(10), INT(4)))]
         VS = [178]
```

inspect

```
state 7  DS = [DO_PLUS; DO_SUBT; E(INT(4)); E(INT(10))]
         VS = [178]
state 8  DS = [DO_PLUS; DO_SUBT; E(INT(4))]
         VS = [178; 10]
```

compute

```
state 9  DS = [DO_PLUS; DO_SUBT]
         VS = [178; 10; 4]
state 10DS = [DO_PLUS]
         VS = [178; 6]
state 11DS = []
         VS = [184]
```

Top of each stack is on the right

Key insight

This evaluator is interleaving two distinct computations:

(1) decomposition of the input expression into sub-expressions
(2) the computation of +, -, and *.

Idea: why not do the decomposition BEFORE the computation?

Key insight: An interpreter can (usually) be **refactored** into a translation (compilation!) followed by a lower-level interpreter.

Interpret_higher (e) = interpret_lower(compile(e))

Note : this can occur at many levels of abstraction: think of machine code being interpreted in micro-code …

Refactor --- compile!

```
(* low-level instructions *)
type instr =
  | Ipush of int
  | Iplus
  | Isubt
  | Imult

type code = instr list

type state_7 = code * value_stack

(* compile : expr -> code *)
let rec compile = function
   | INT a          -> [Ipush a]
   | PLUS(e1, e2)   -> (compile e1) @ (compile e2) @ [Iplus]
   | SUBT(e1, e2)   -> (compile e1) @ (compile e2) @ [Isubt]
   | MULT(e1, e2)   -> (compile e1) @ (compile e2) @ [Imult]
```

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

Evaluate compiled code.

```
(* step_7 : state_7 -> state_7 *)
let step_7 = function
   | (Ipush v :: is,         vs) ->  (is, v :: vs)
   | (Iplus :: is, v2::v1::vs) -> (is, (v1 + v2) :: vs)
   | (Isubt :: is, v2::v1::vs) -> (is, (v1 - v2) :: vs)
   | (Imult :: is, v2::v1::vs) -> (is, (v1 * v2) :: vs)
   | _ -> failwith "eval : runtime error!"

let rec driver_7 = function
    | ([], [v]) -> v
    | _ -> driver_7 (step_7 state)

let eval_7 e = driver_7  (compile e, []) 1
```

An eval_7 trace

inspect

```
compile (PLUS(MULT(INT 89, INT 2), SUBT(INT 10, INT 4)))
  = [push 89; push 2; mult; push 10; push 4; subt; plus]
```

compute

state 1 IS = [add; sub; push 4; push 10; mul; push 2; push 89]
VS = []

state 2 IS = [add; sub; push 4; push 10; mul; push 2]
VS = [89]

state 3 IS = [add; sub; push 4; push 10; mul]
VS = [89; 2]

state 4 IS = [add; sub; push 4; push 10]
VS = [178]

state 5 IS = [add; sub; push 4]
VS = [178; 10]

state 6 IS = [add; sub]
VS = [178; 10; 4]

state 7 IS = [add]
VS = [178; 6]

state 8 IS = []
VS = [184]

Top of each stack is on the right

Interp_0.ml → interp_1.ml → interp_2.ml

The derivation from eval to compile+eval_7 can be used as a guide to a derivation from Interpreter 0 to interpreter 2.

1. Apply CPS to the code of Interpreter 0
2. Defunctionalise
3. Arrive at interpreter 1, which has a single continuation stack containing expressions, values and environments
4. Spit this stack into two stacks : one for instructions and the other for values and environments
5. Refactor into compiler + lower-level interpreter
6. Arrive at interpreter 2.

Taking stock

Starting from a direct implementation of Slang/L3 semantics, we have **DERIVED** a Virtual Machine in a step-by-step manner. The correctness of aach step is (more or less) easy to check.

Interpreter 0

Explicit stack via CPS+DFS → Interpreter 1

Split stack into two, refactor → Interpreter 2

Linearise code → Interpreter 3

Low-level addressable stack → Jargon VM

Compiler Construction
Lent Term 2017

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

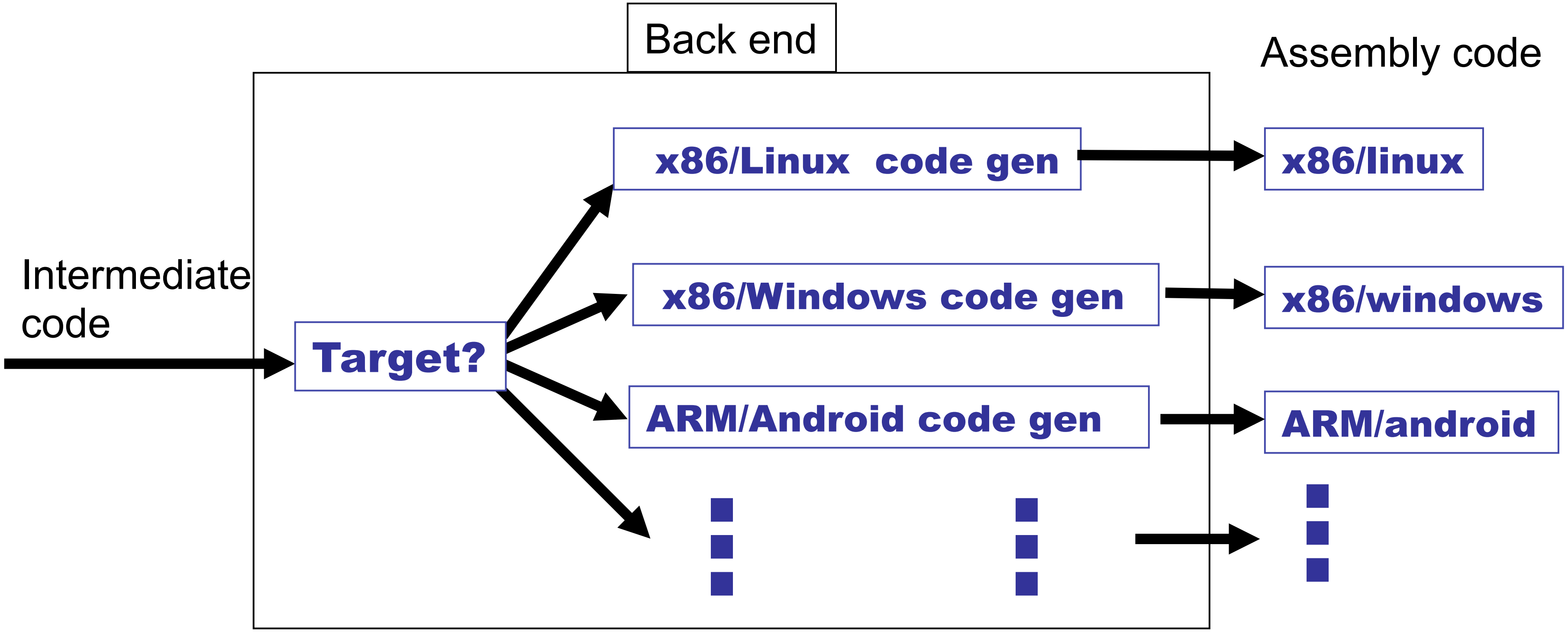
- Code generation for multiple platforms.
- Assembly code
- Linking and loading
- The Application Binary Interface (ABI)
- Object file format (only ELF covered)
- A crash course in x86 architecture and instruction set
- Naïve generation of x86 code from Jargon VM instructions

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; }

    …
    …
  }
}
…
…
```

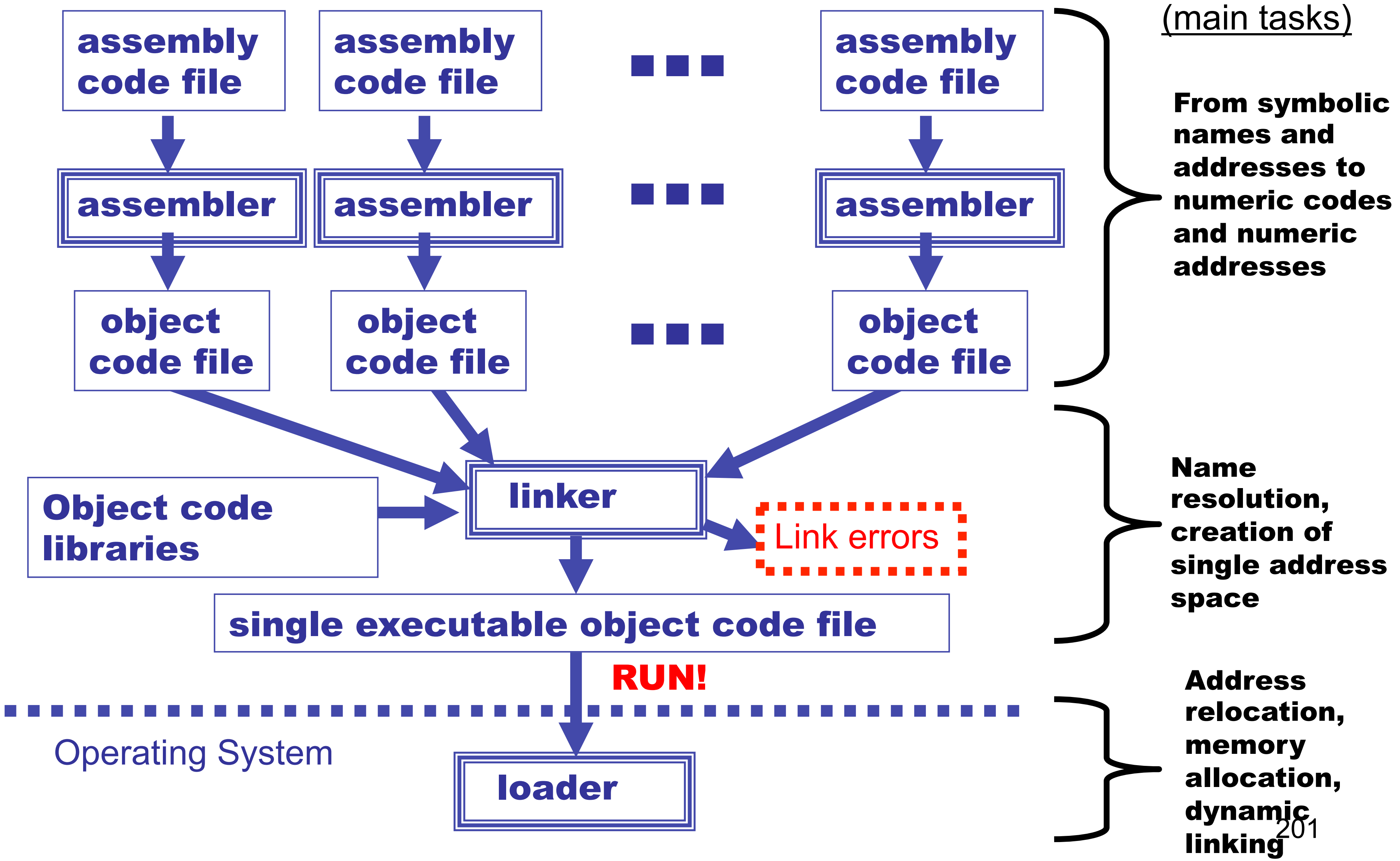
- Generate compact byte code for each Jargon instruction.
- Compiler writes byte codes to a file.
- Implement an interpreter in C or C++ for these byte codes.
- Execution is much faster than our jargon.ml implementation.
- **Or, we could generate assembly code from Jargon instructions ….**

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



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) 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.

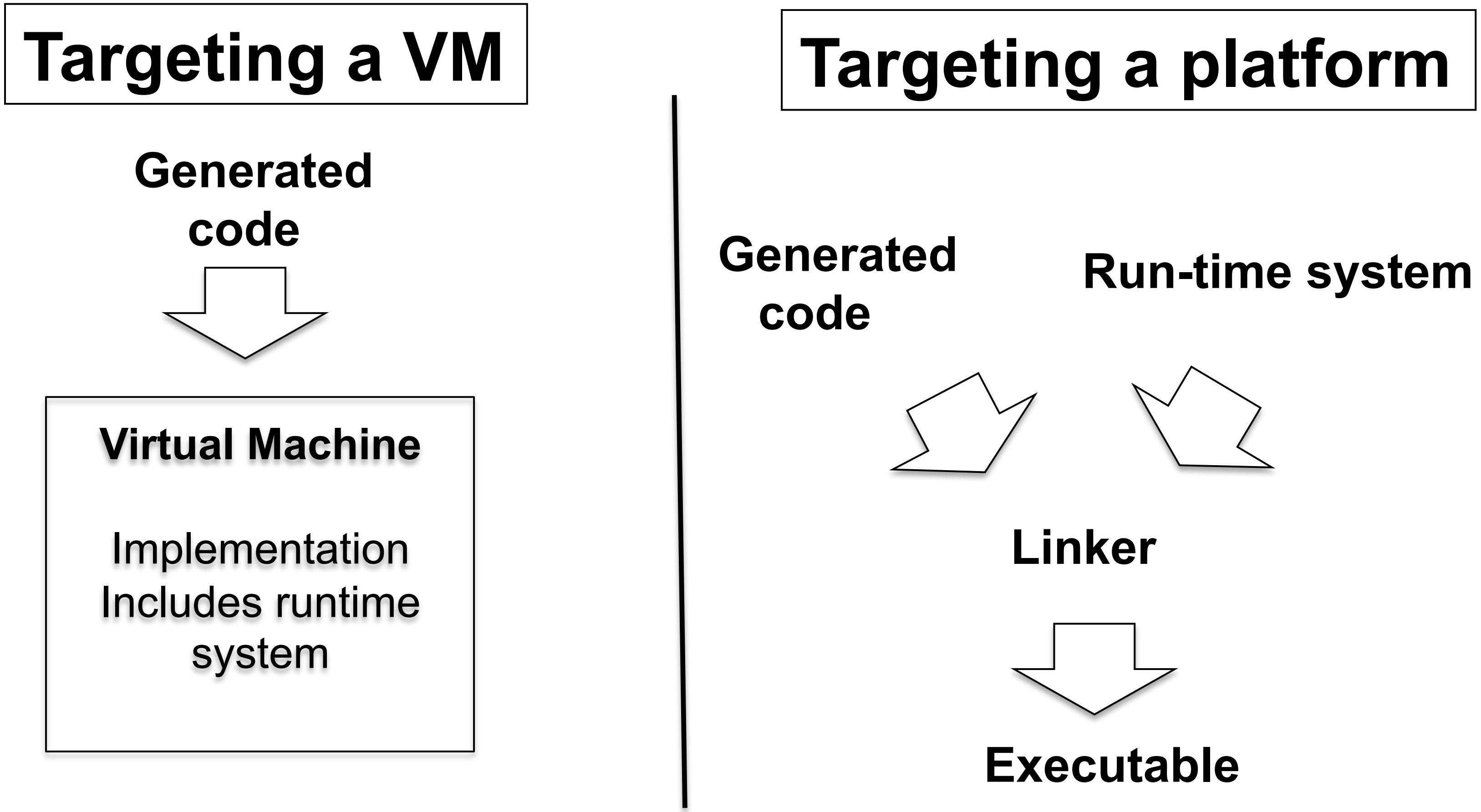
(-) Non-compatible changes to a library wreck previously working programs "DLL hell".

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

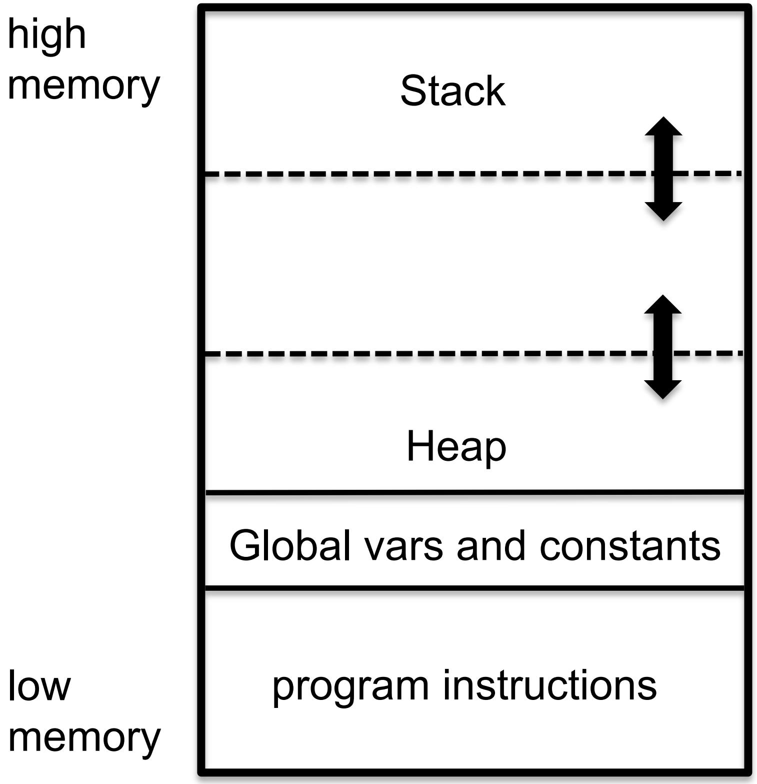


In either case, implementers of the compiler and the runtime system must agree on many low-level details of memory layout and data representation.

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 address
                    # 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 memory
                     # address. Add 4 to that address. Copy
                     # the value in esp to this new address.
```

A few more examples

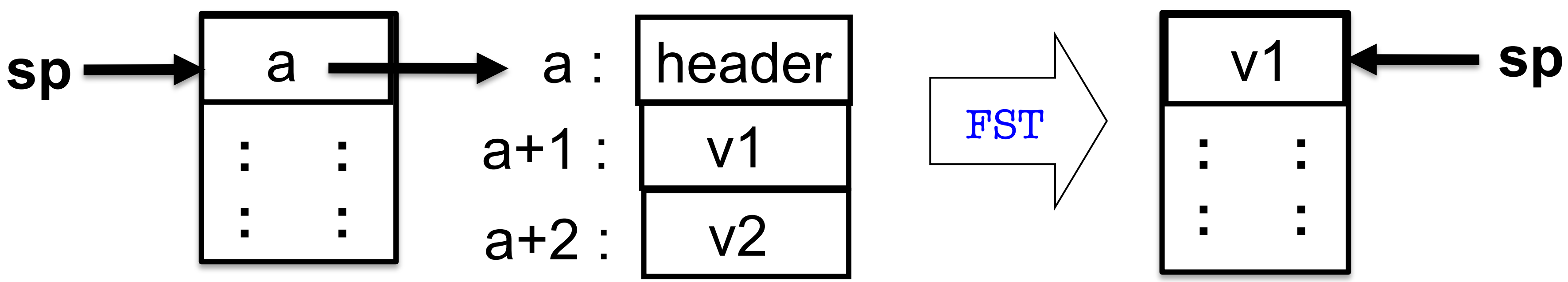
```
call label  # push return address on stack and jump to label
ret         # pop return address off stack and jump there
            # NOTE: managing other bits of the stack frame
            # such as stack and frame pointer must be done
            # explicitly
subl $4, %esp   # subtract 4 from esp. 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**.

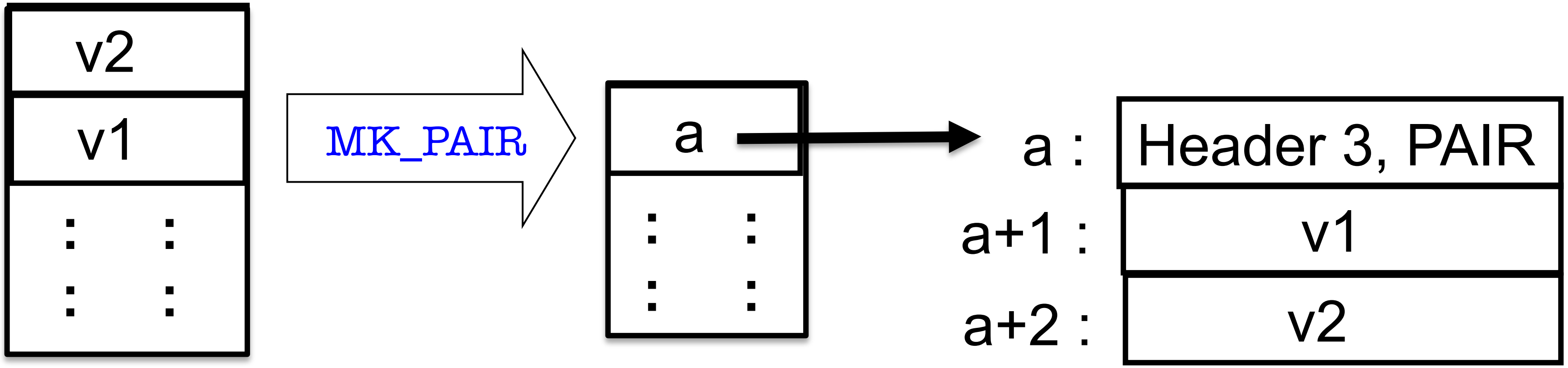
Some Jargon VM instructions are "easy" to translate

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?

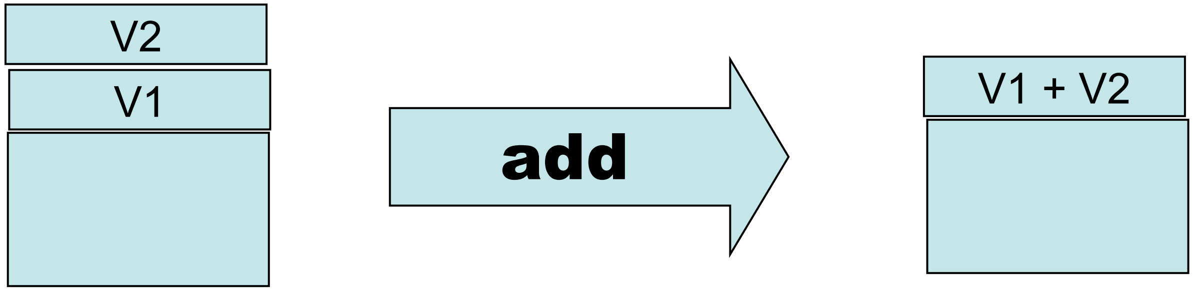
Houston, we have a problem….

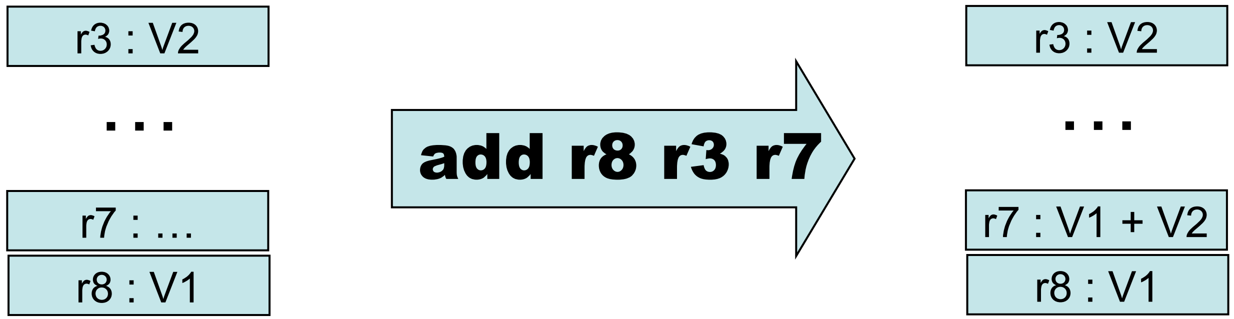
- It may not be obvious now, but if we want to have automated memory management we need to be able to distinguish between values (say integers) and pointers at runtime.
- Have you ever noticed that integers in SML or Ocaml are either 31 (or 63) bits rather than the native 32 (or 64) bits?
  - That is 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:

$$X := Y;$$

$$Z := X + Z$$

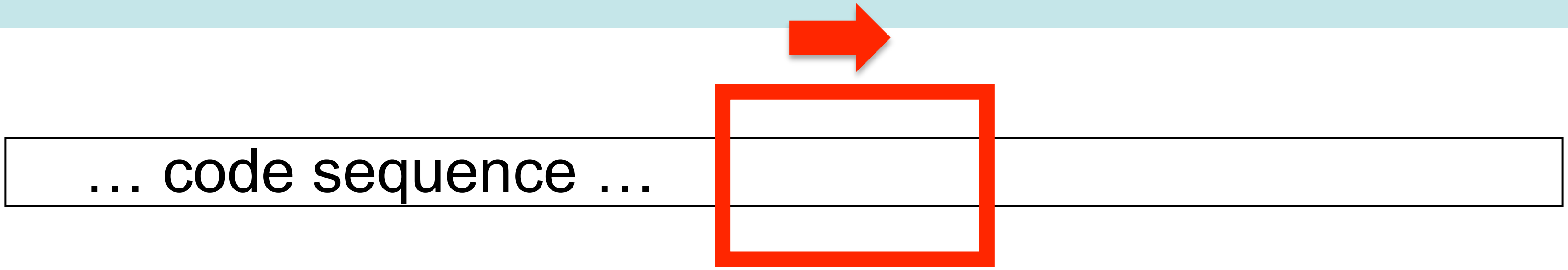
Compiled code:

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.

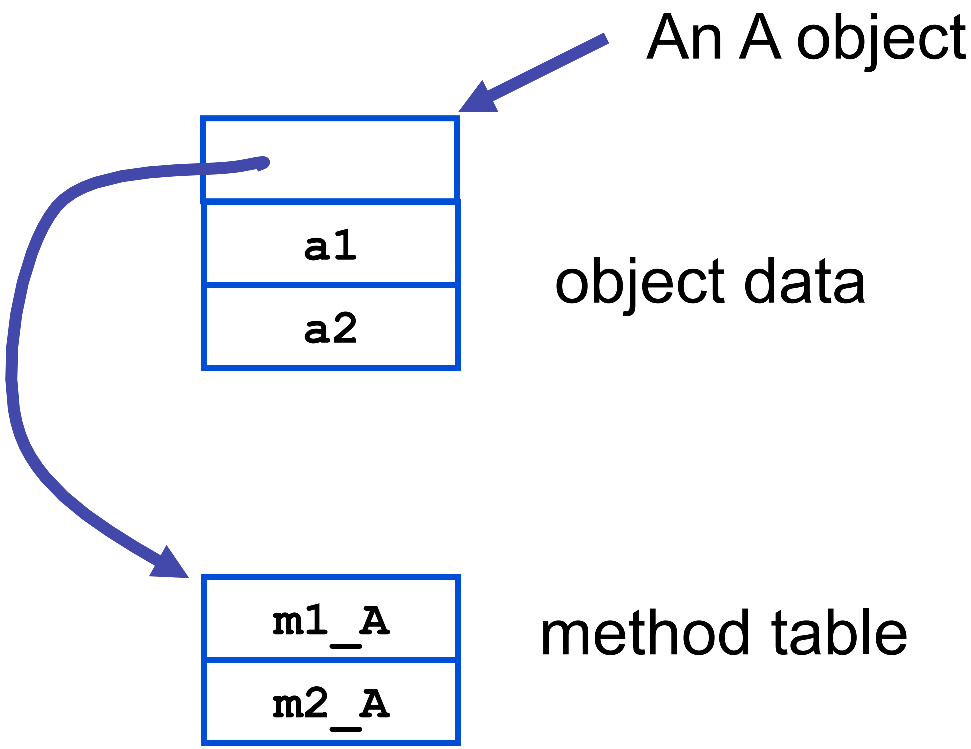
Another OO Feature

- 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++

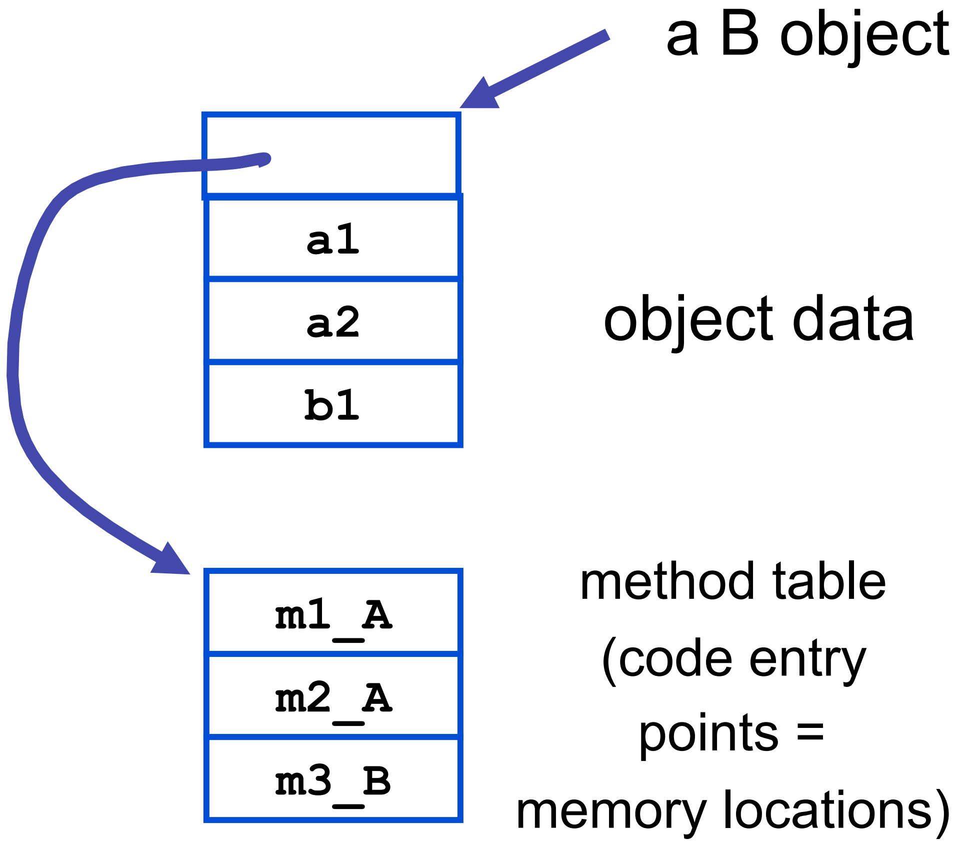
```
class A {
public:
    int a1, a2;

    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")

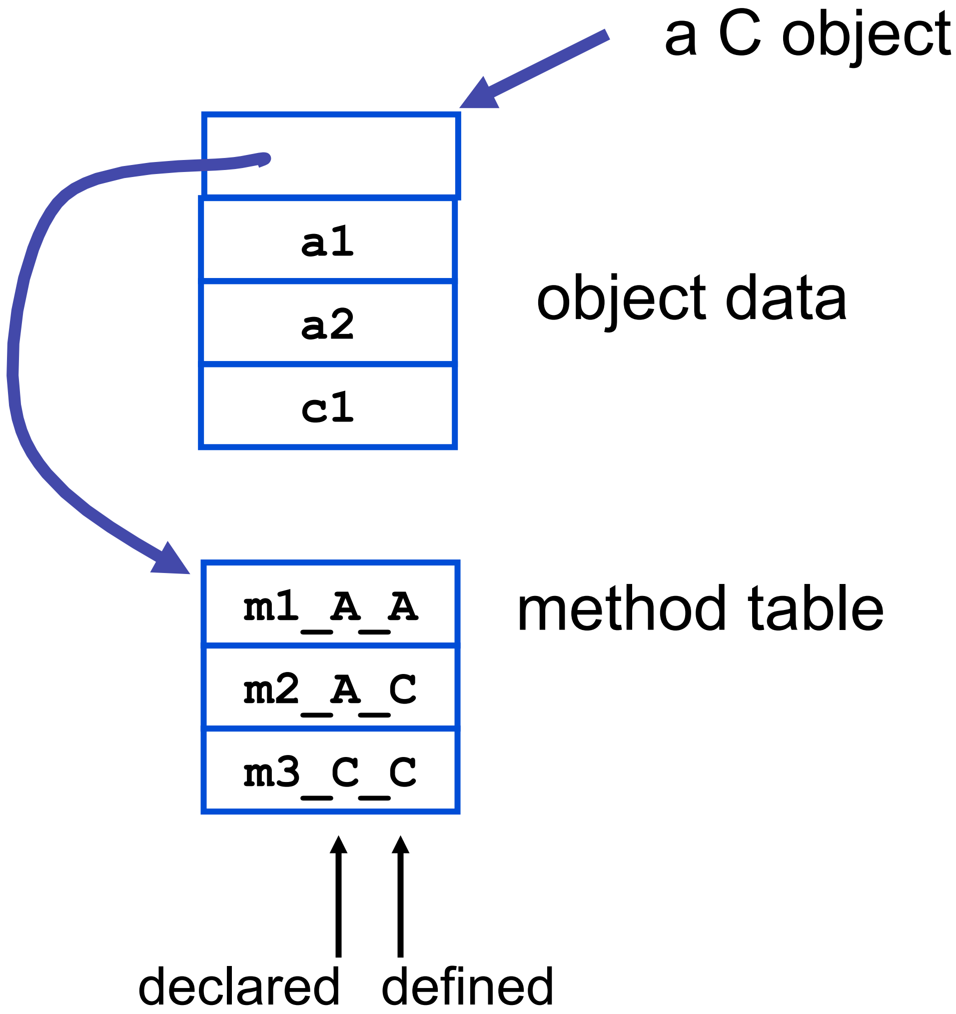
```
class B : public A {
public:
   int b1;

   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);
```

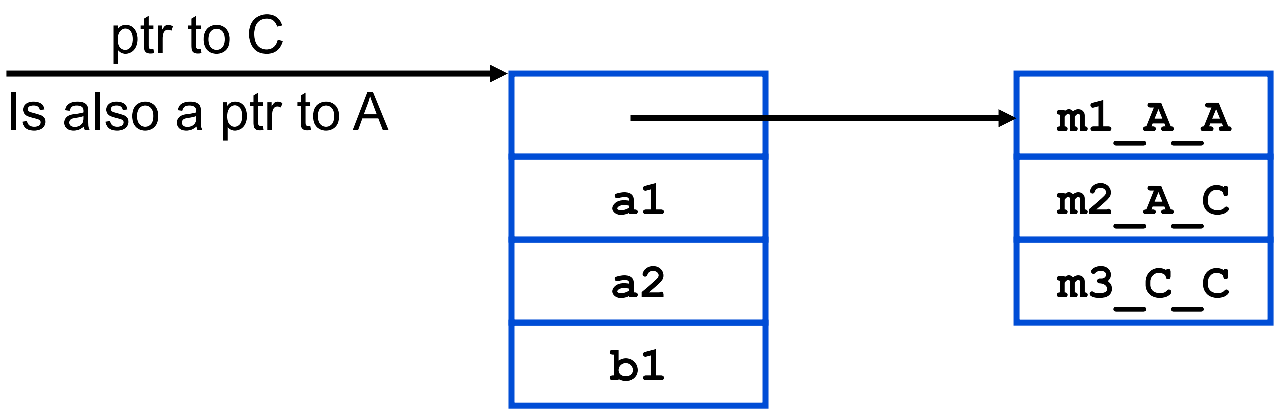
static

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

dynamic

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.

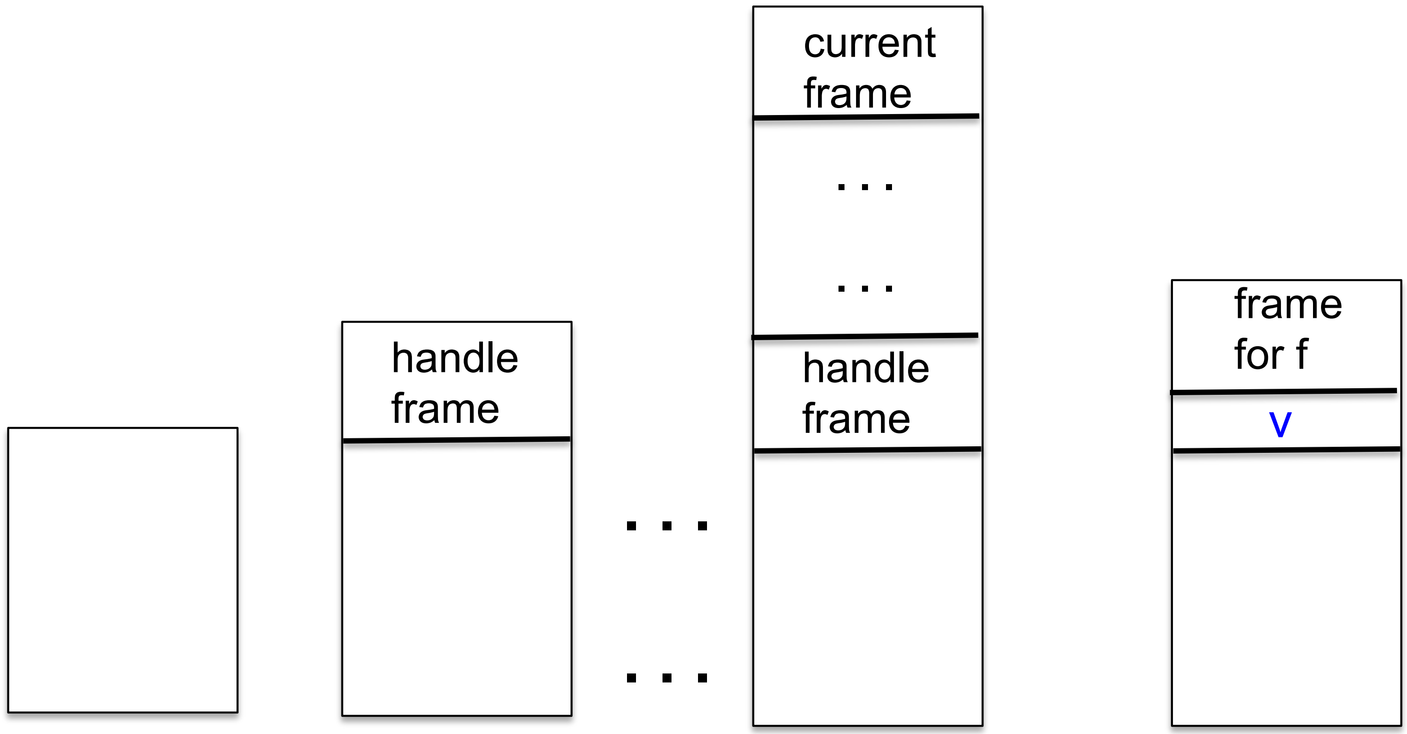
Otherwise, an exceptional value v' is "raised" in the evaluation of e, then result is (f v')

`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)

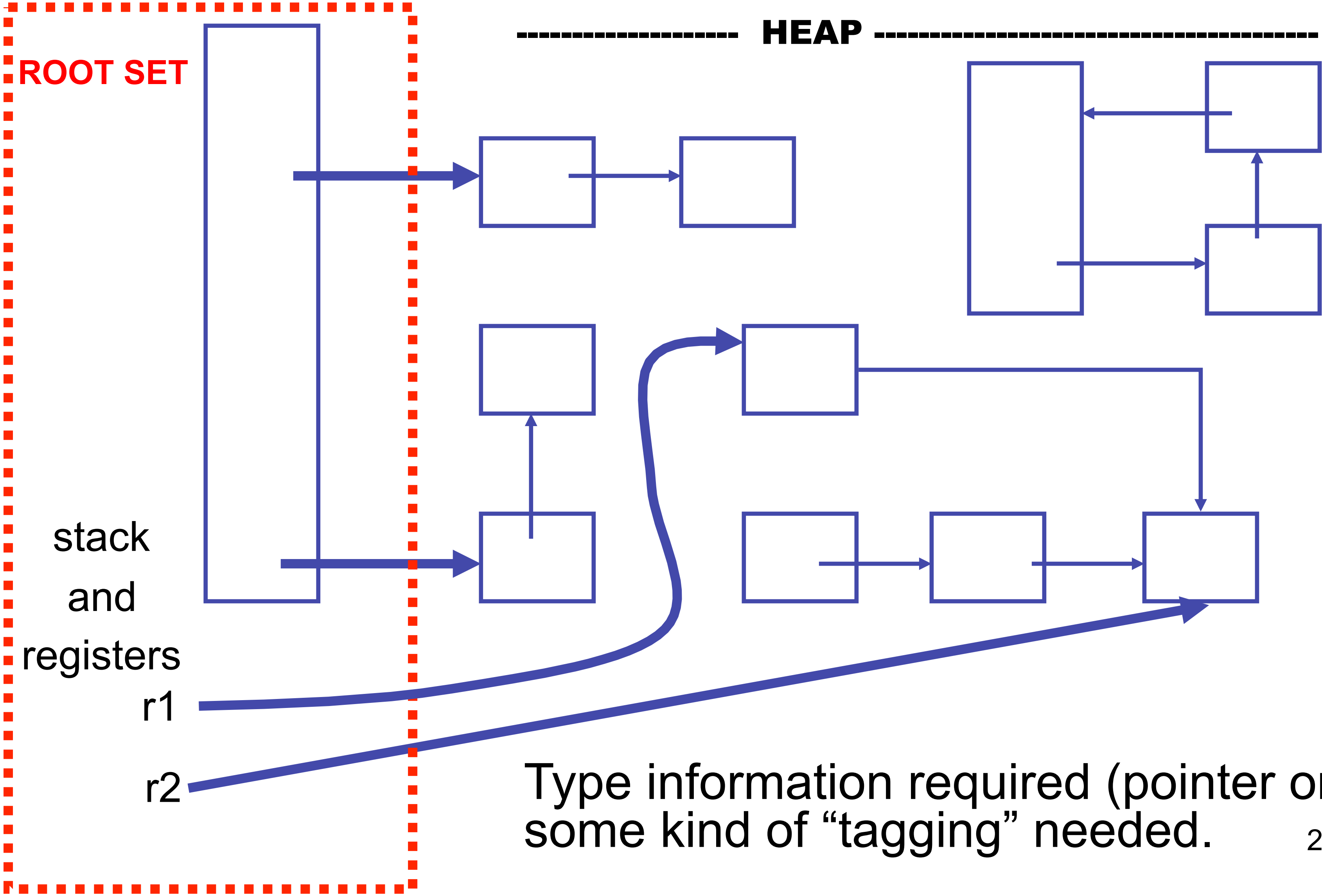
Explicit (manual) memory management

- 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?
  - <u>Advantage</u>: With these procedures we can implement memory management for "higher level" languages ;-)

Memory Management

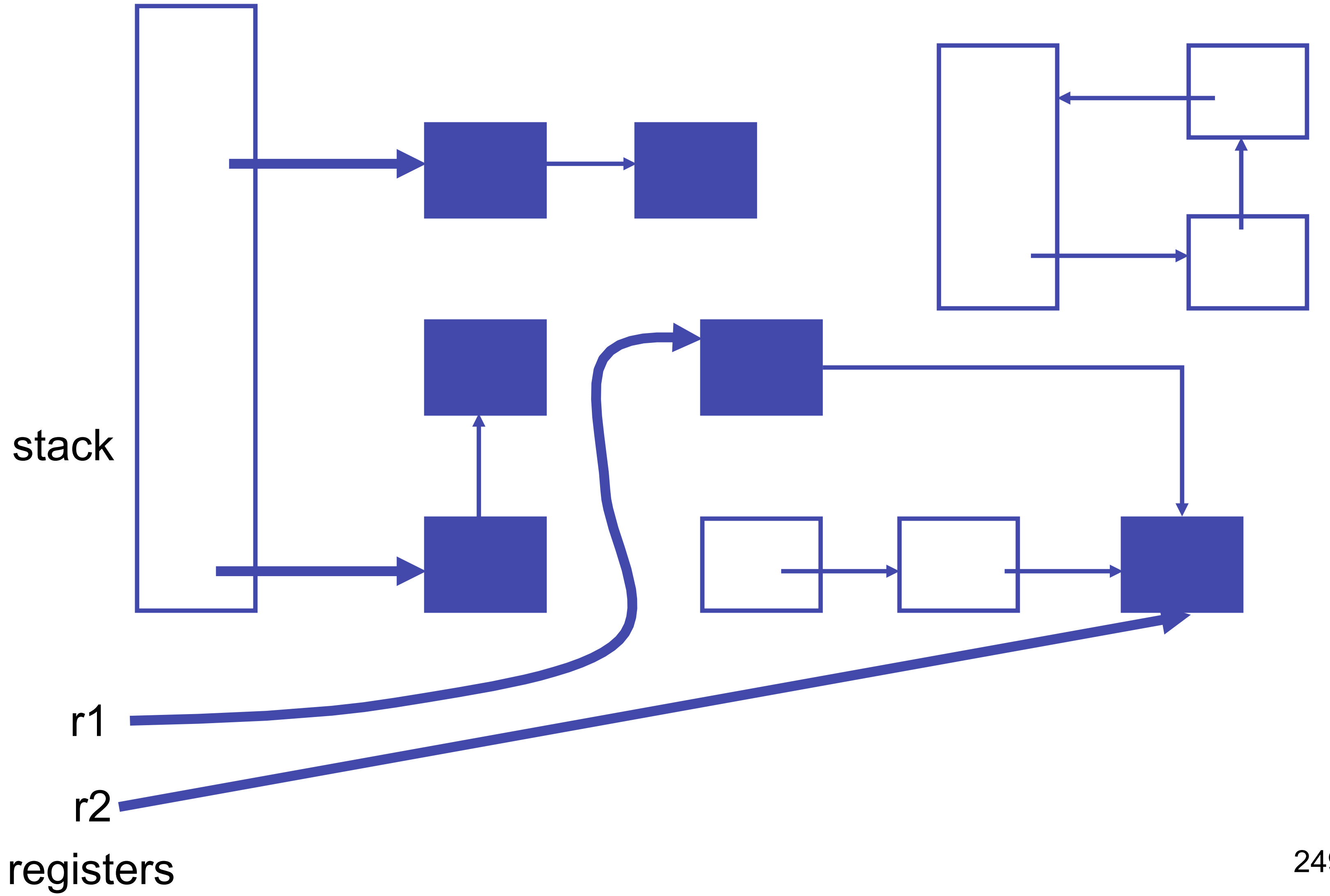
- Many programming languages allow programmers to (implicitly) allocate new storage dynamically, with no need to worry about reclaiming space no longer used.
  - New records, arrays, tuples, objects, closures, etc.
  - Java, SML, OCaml, Python, JavaScript, Python, Ruby, Go, Swift, SmallTalk, …
- Memory 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"

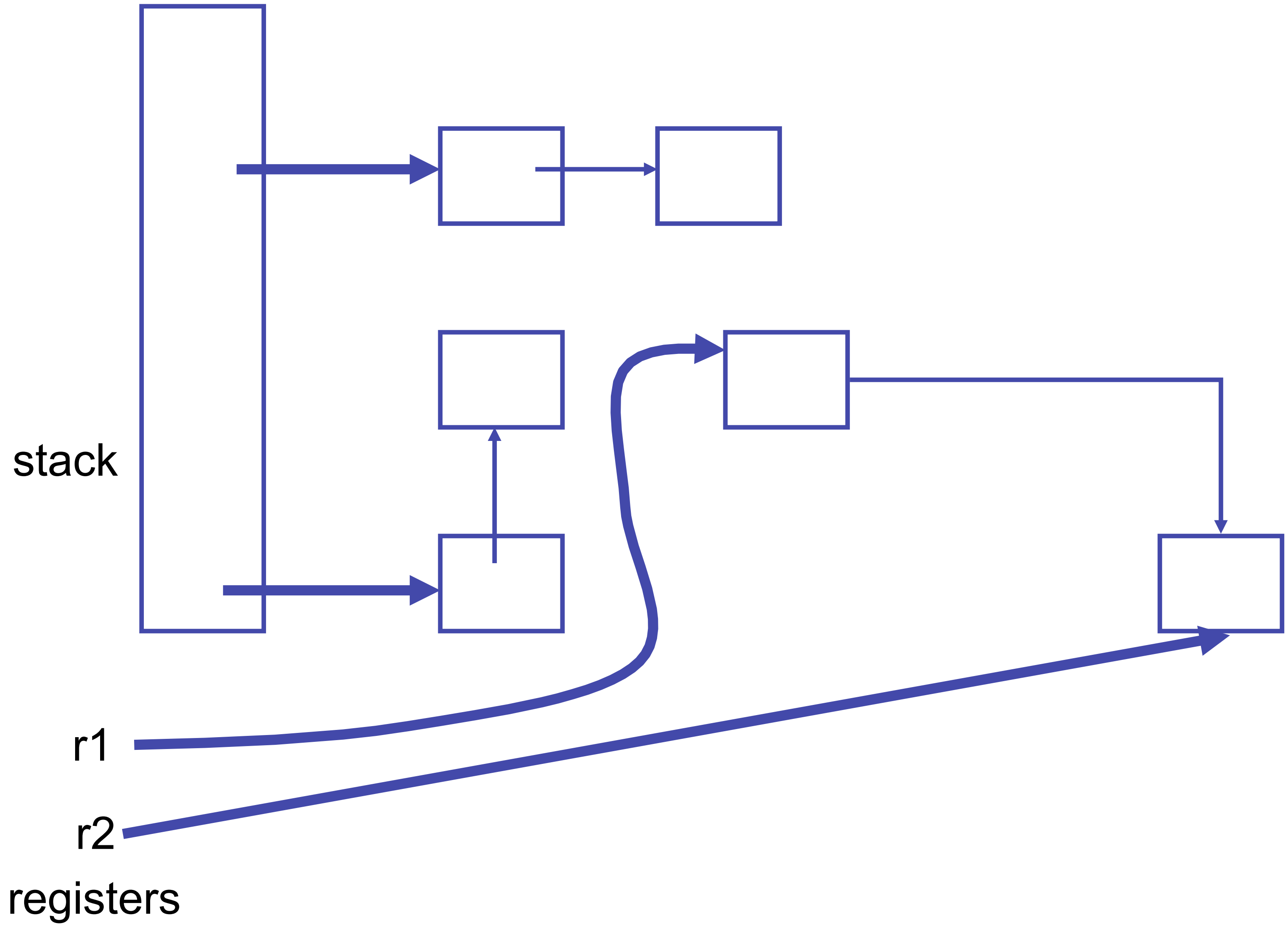


Type information required (pointer or not), some kind of "tagging" needed.

… 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

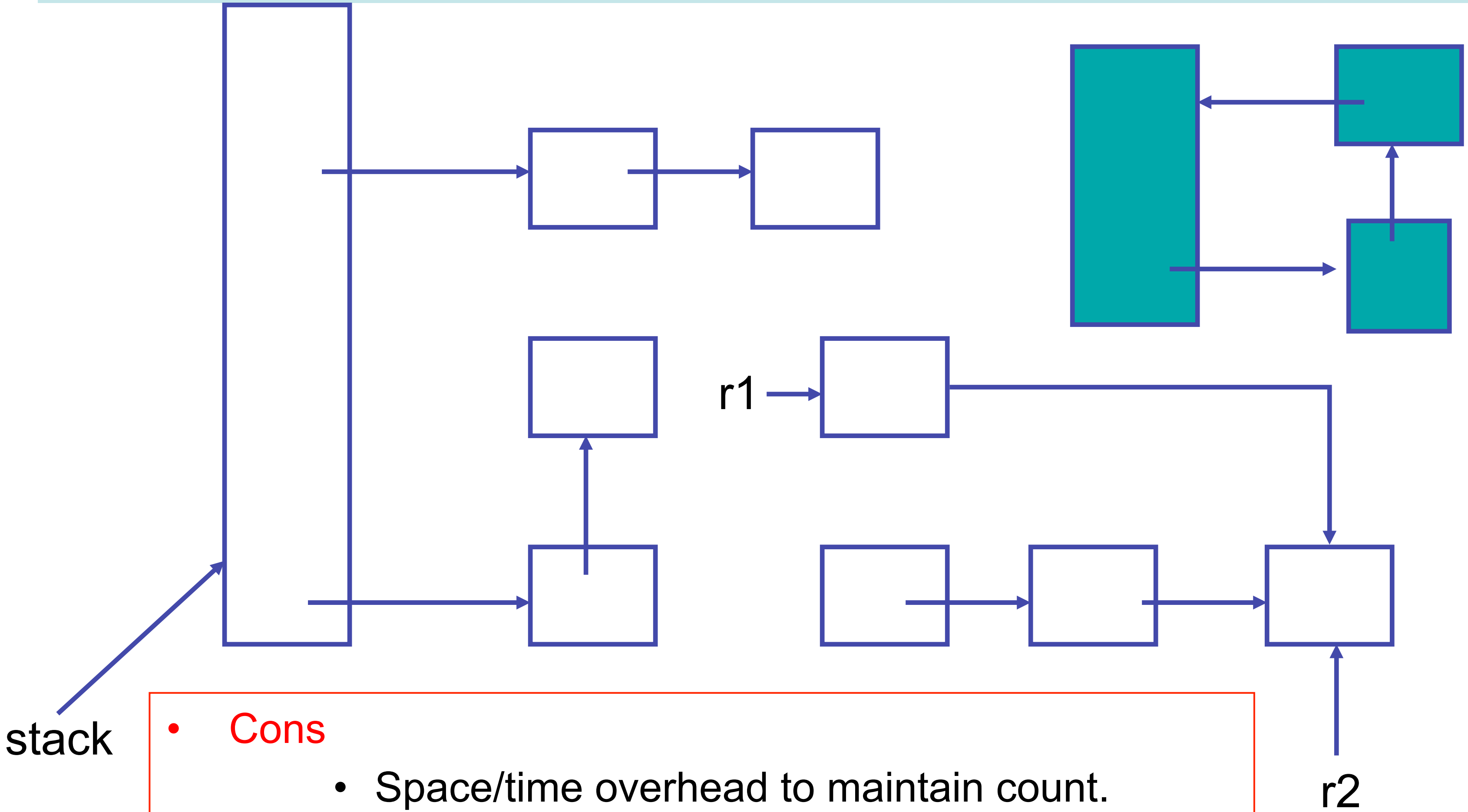
A Unified Theory of Garbage Collection.
David F. Bacon, Perry Cheng, V.T. Rajan.
OOPSLA 2004.

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…)

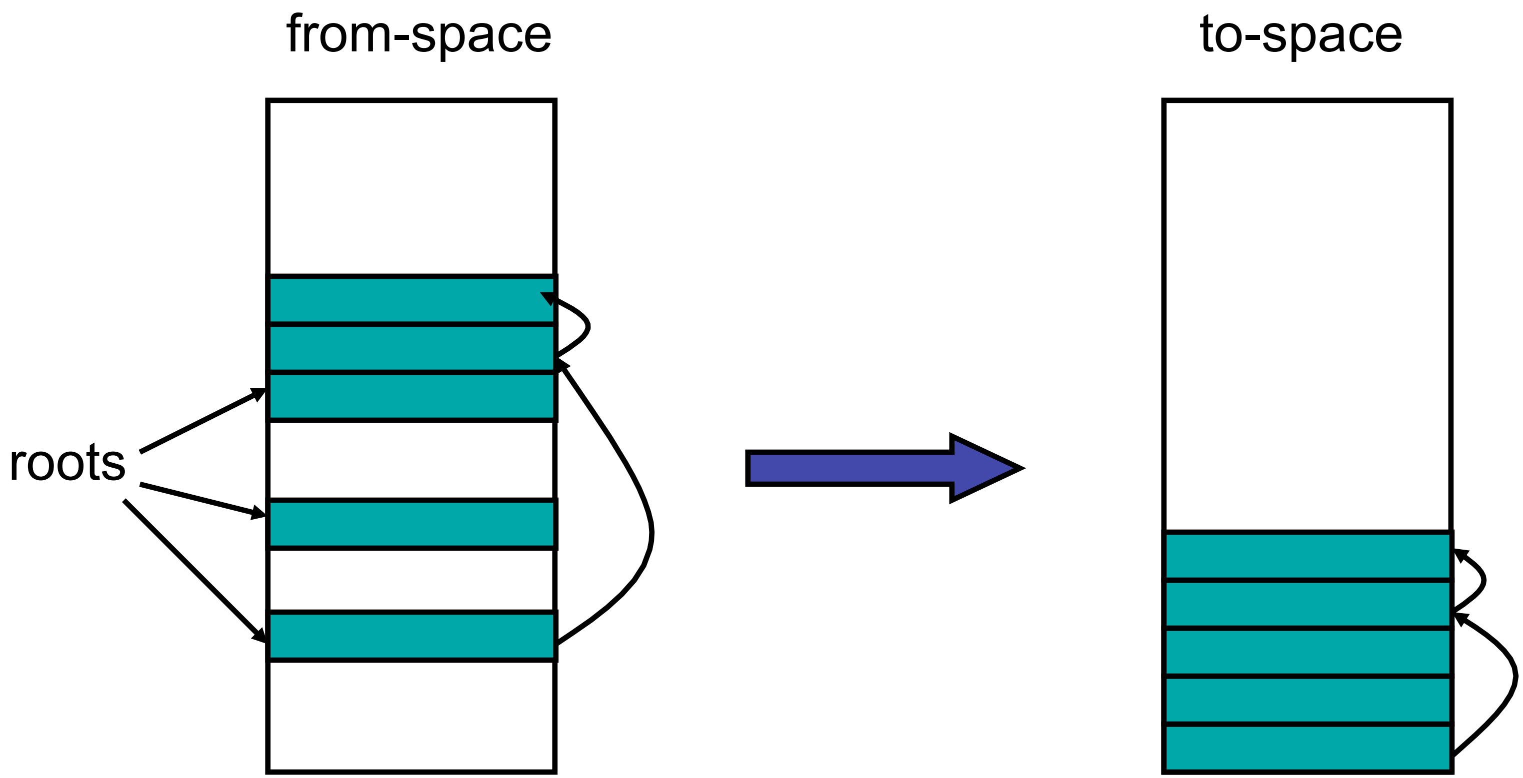
Mark and Sweep

- 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.

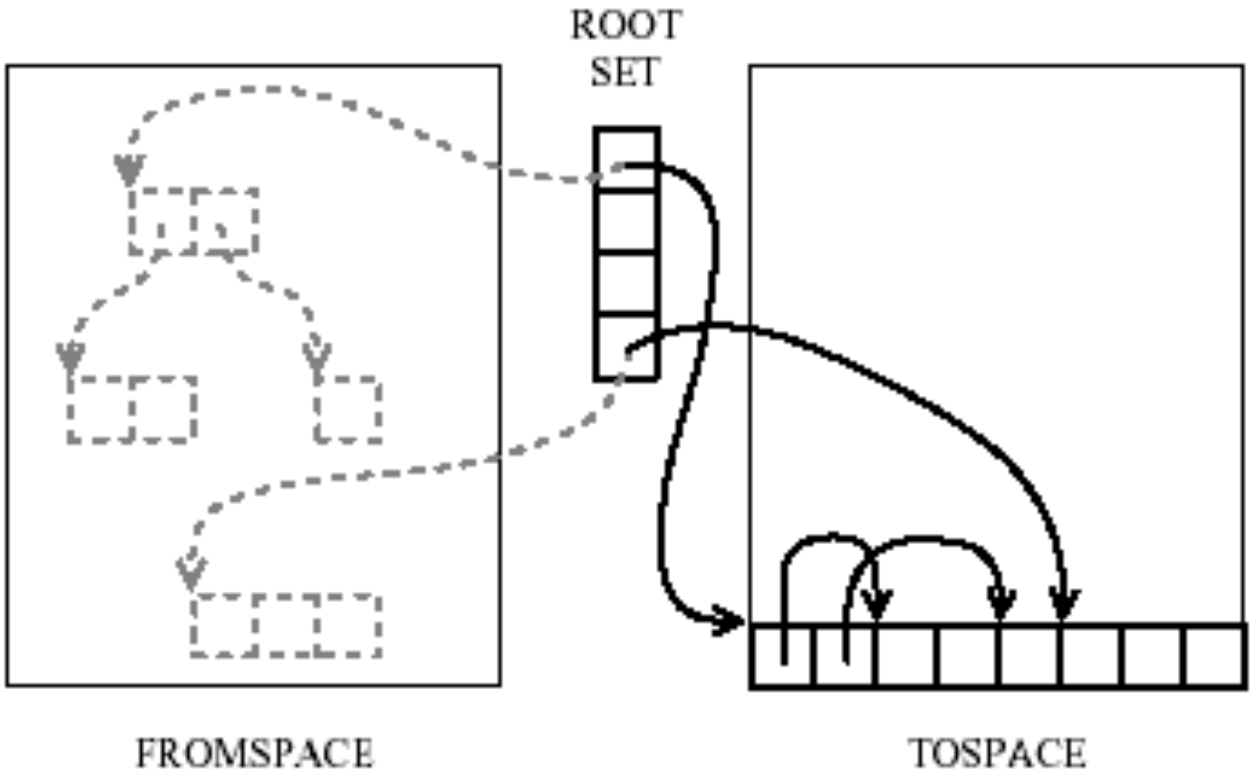


Diagram from Andrew Appel’s **Modern Compiler Implementation**

IDEA: Generational garbage collection

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

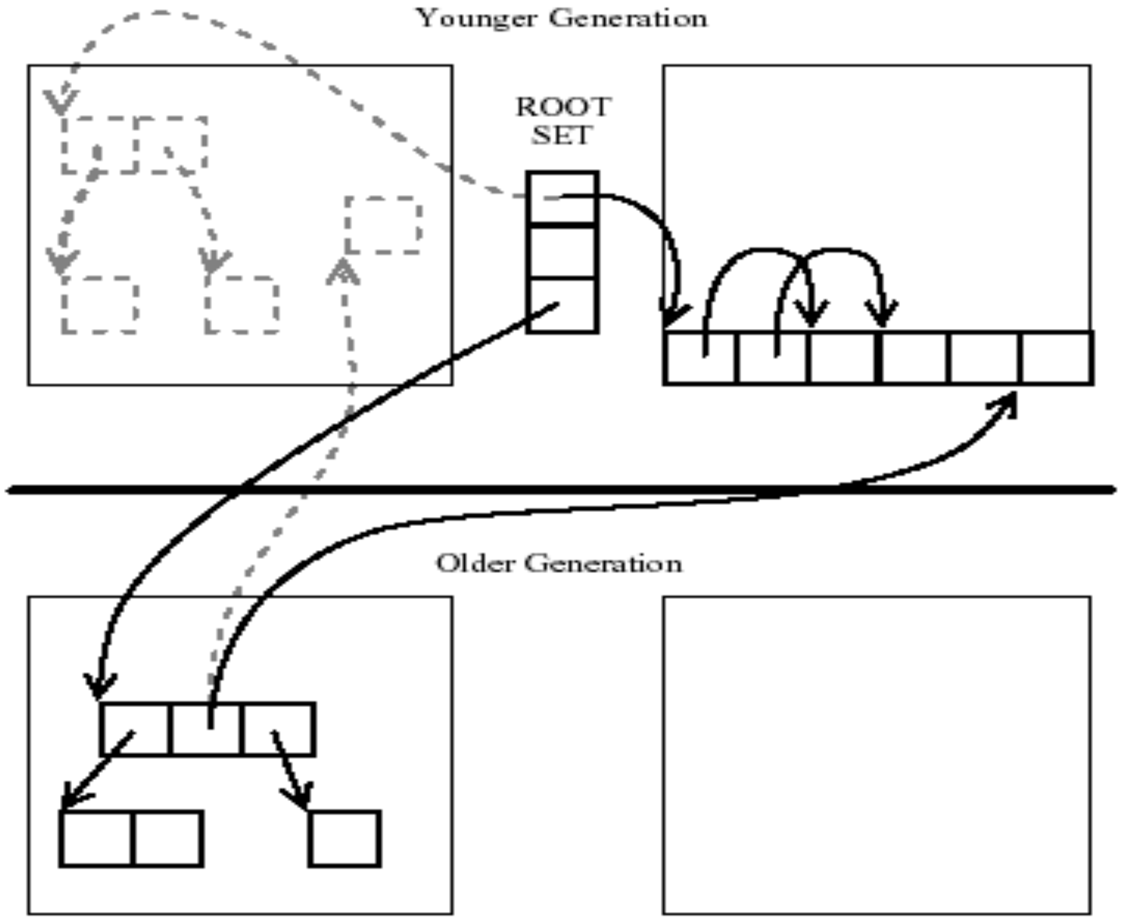


Diagram from Andrew Appel's **Modern Compiler Implementation**

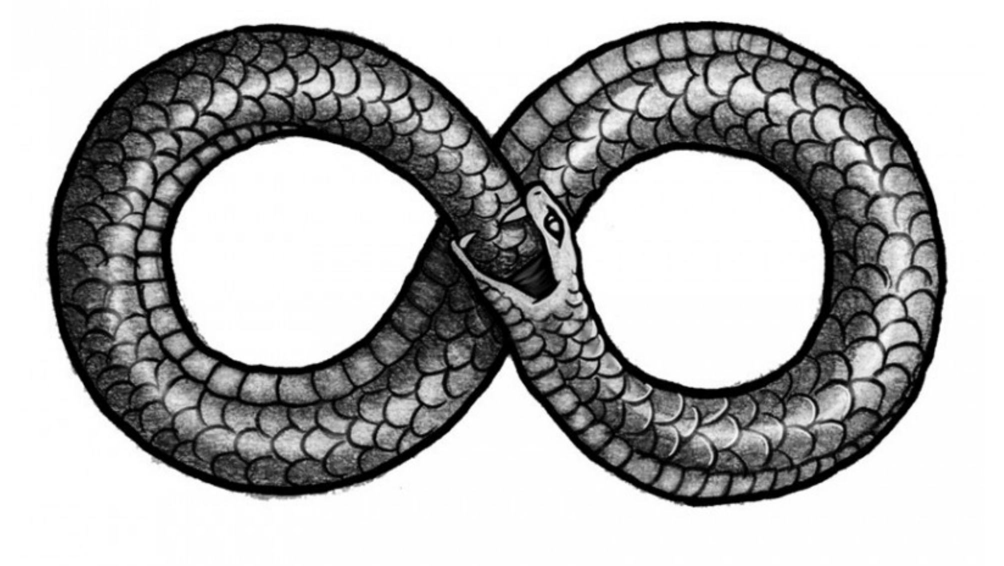
Other issues...

- 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

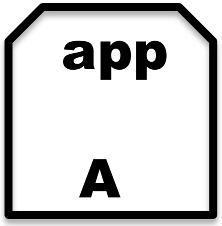
- Compilers compiling themselves!
- Read Chapter 13 Of
  - Basics of Compiler Design
  - by Torben Mogensen

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

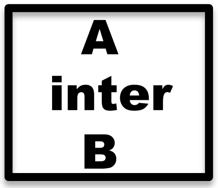


http://mythologian.net/ouroboros-symbol-of-infinity/

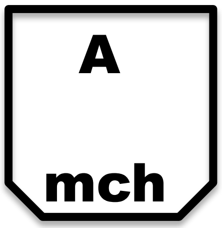
Bootstrapping. We need some notation . . .



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

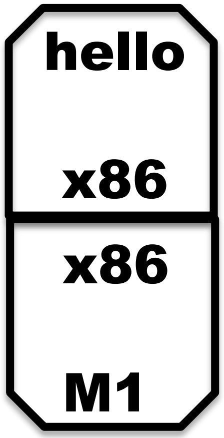


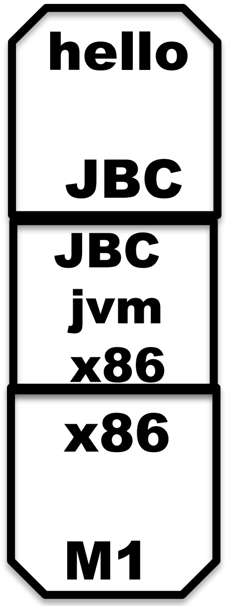
An interpreter or VM for language **A** Written in language **B**



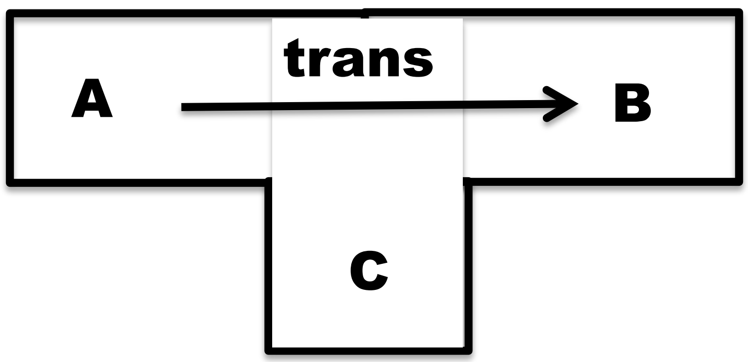
A machine called **mch** running language **A** natively.

Simple Examples



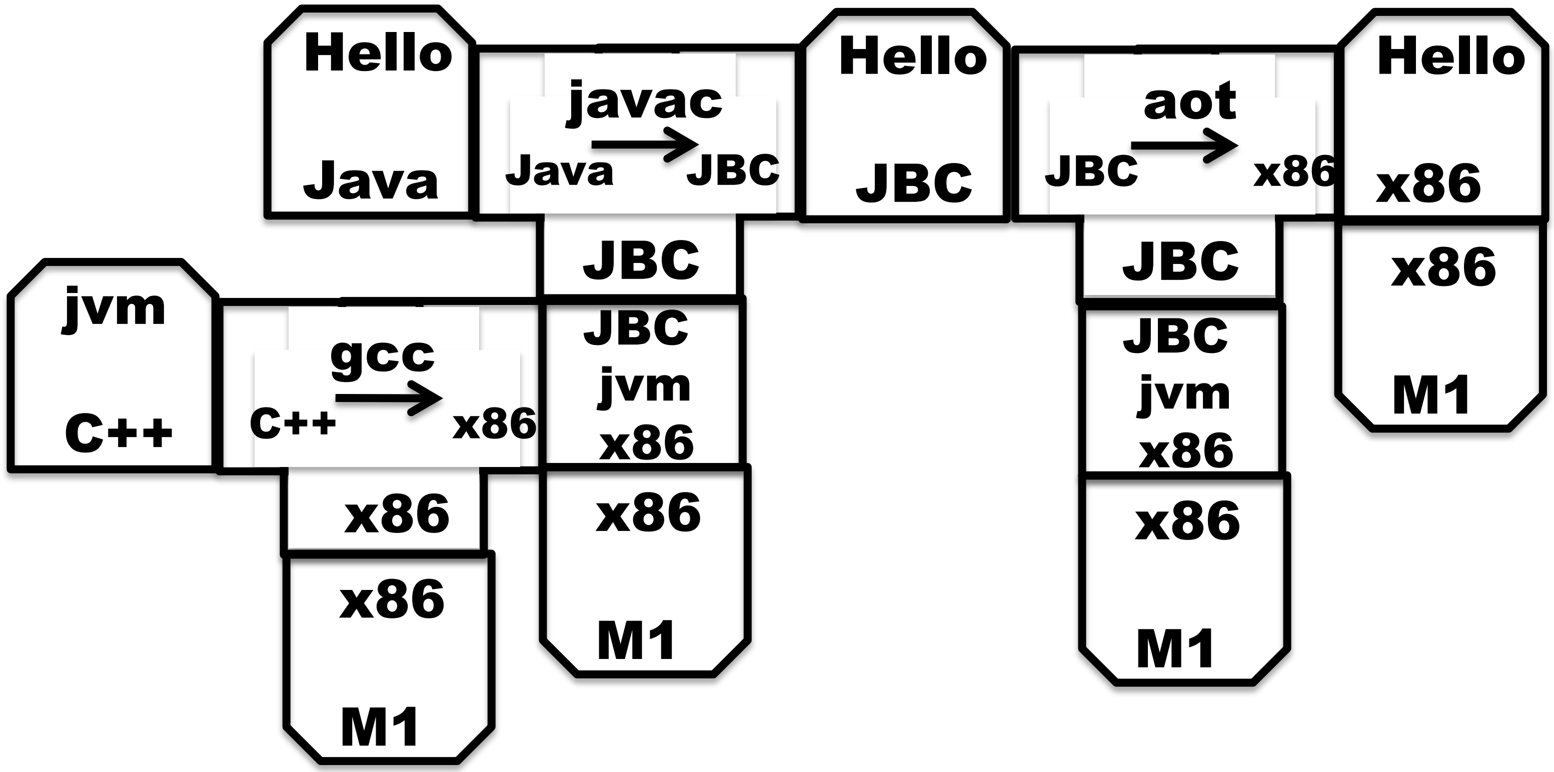


Tombstones



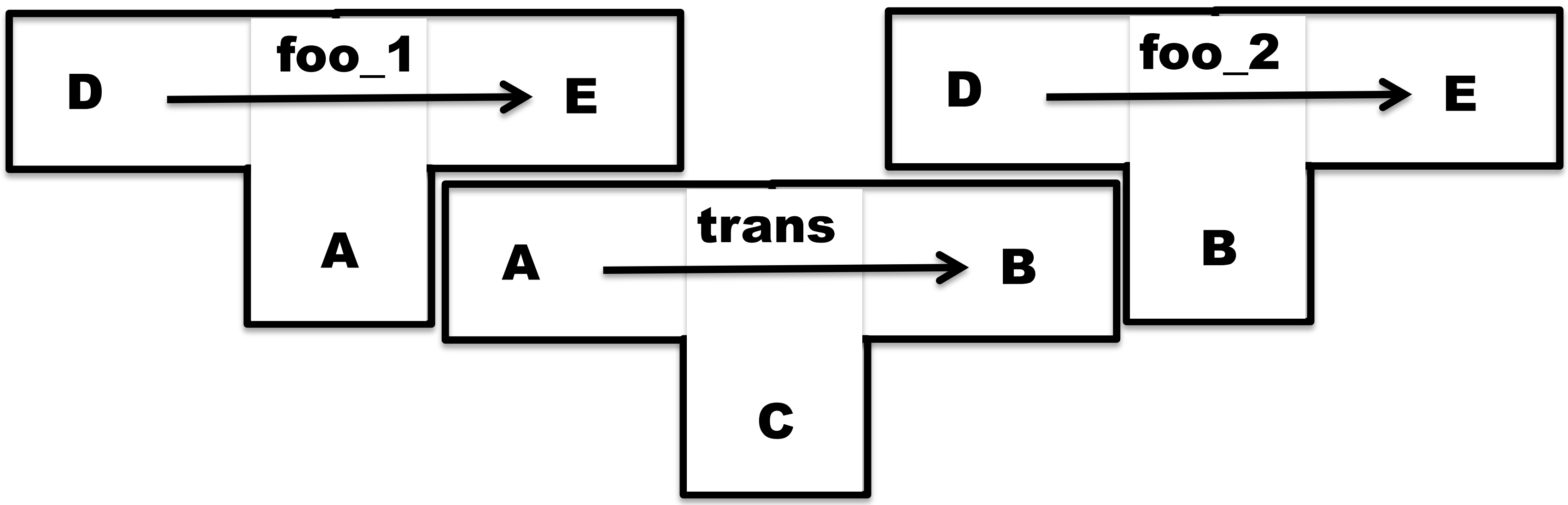
This is an application called **trans** that translates programs in language **A** into programs in language **B**, and it is written in language **C**.

Ahead-of-time compilation



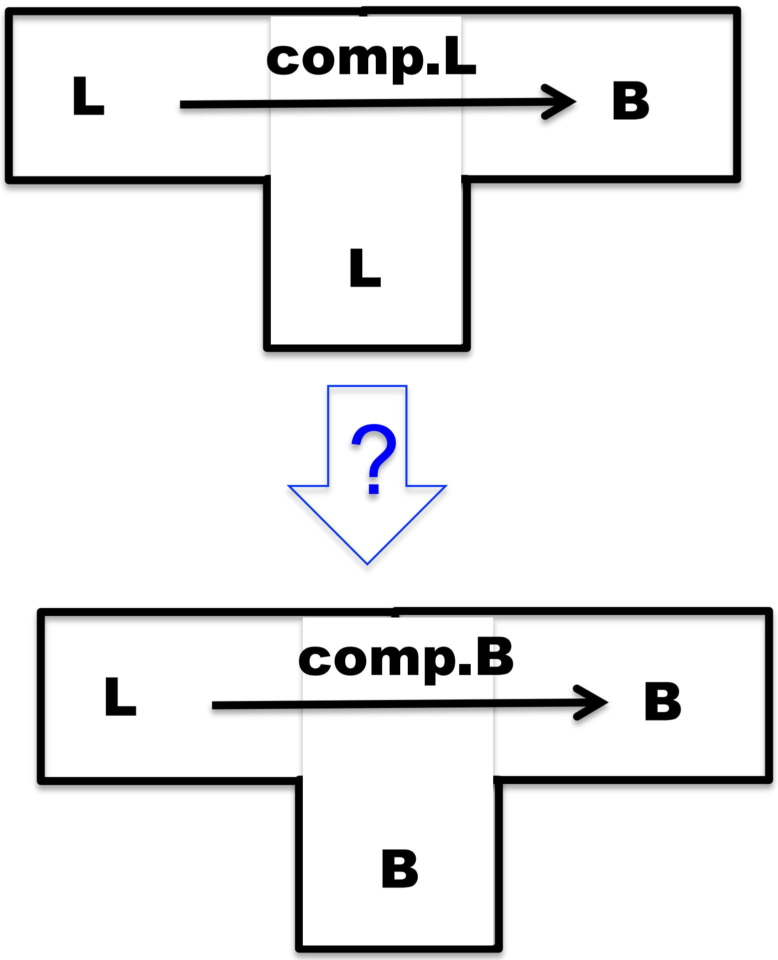
Thanks to David Greaves
for the example.

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.**

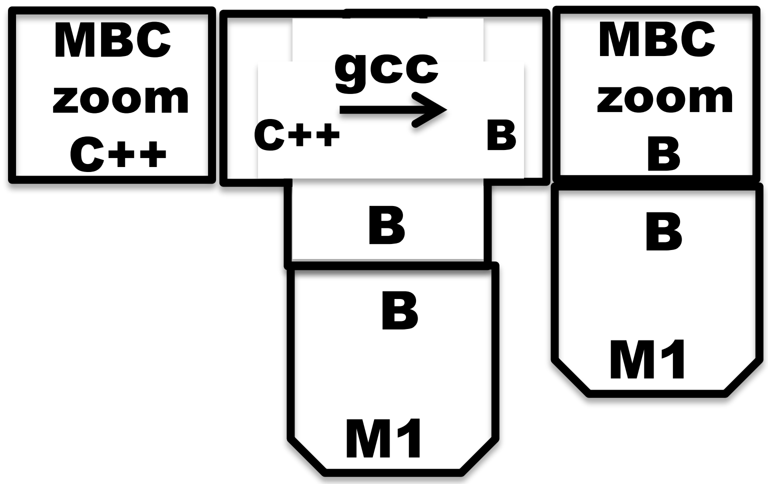
Our compiler is written in L!

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.

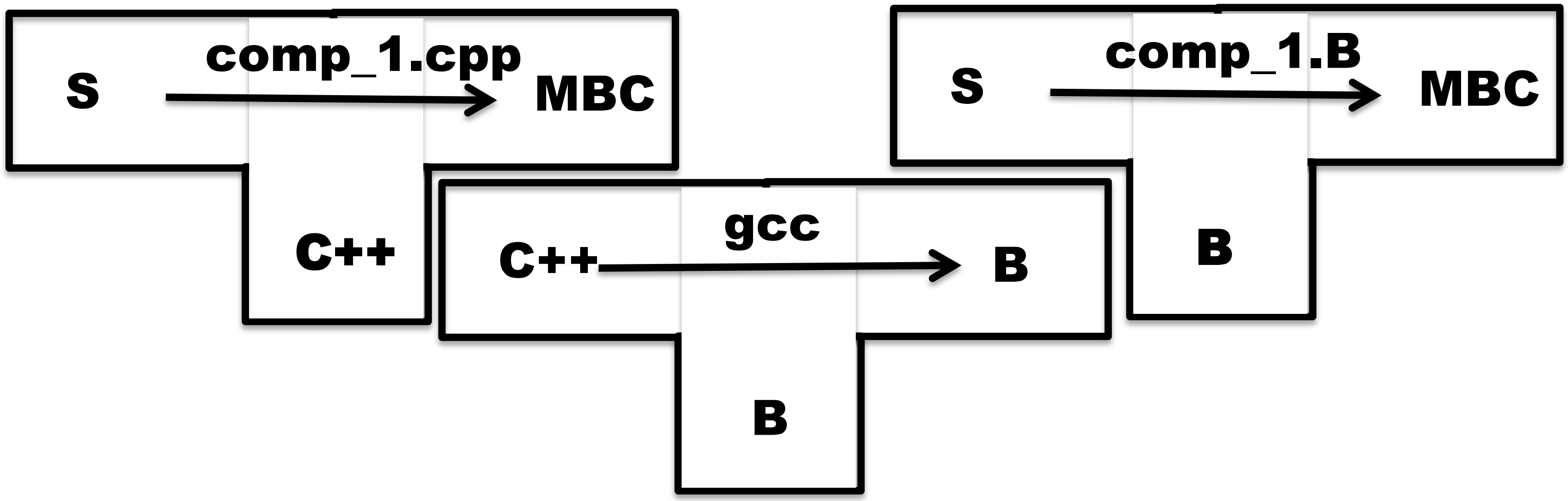
Step 1
Write a small interpreter (VM) for a small language of byte codes

MBC = My Byte Codes



The **zoom** machine!

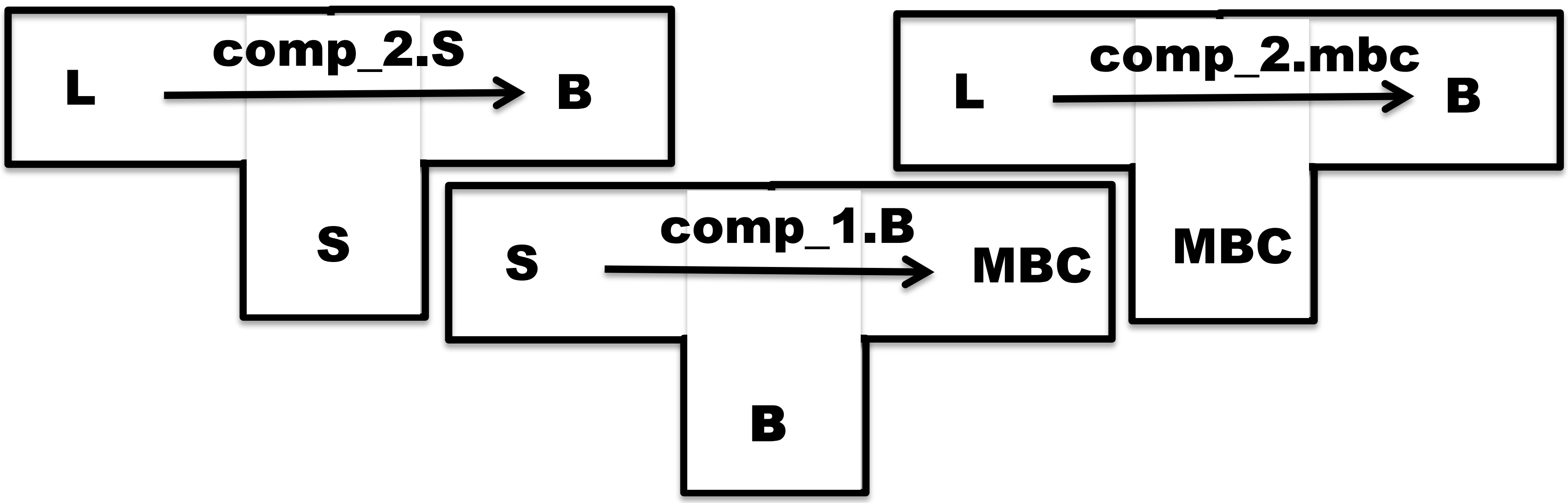
Step 2
Pick a small subset S of L and write a translator from S to MBC



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.

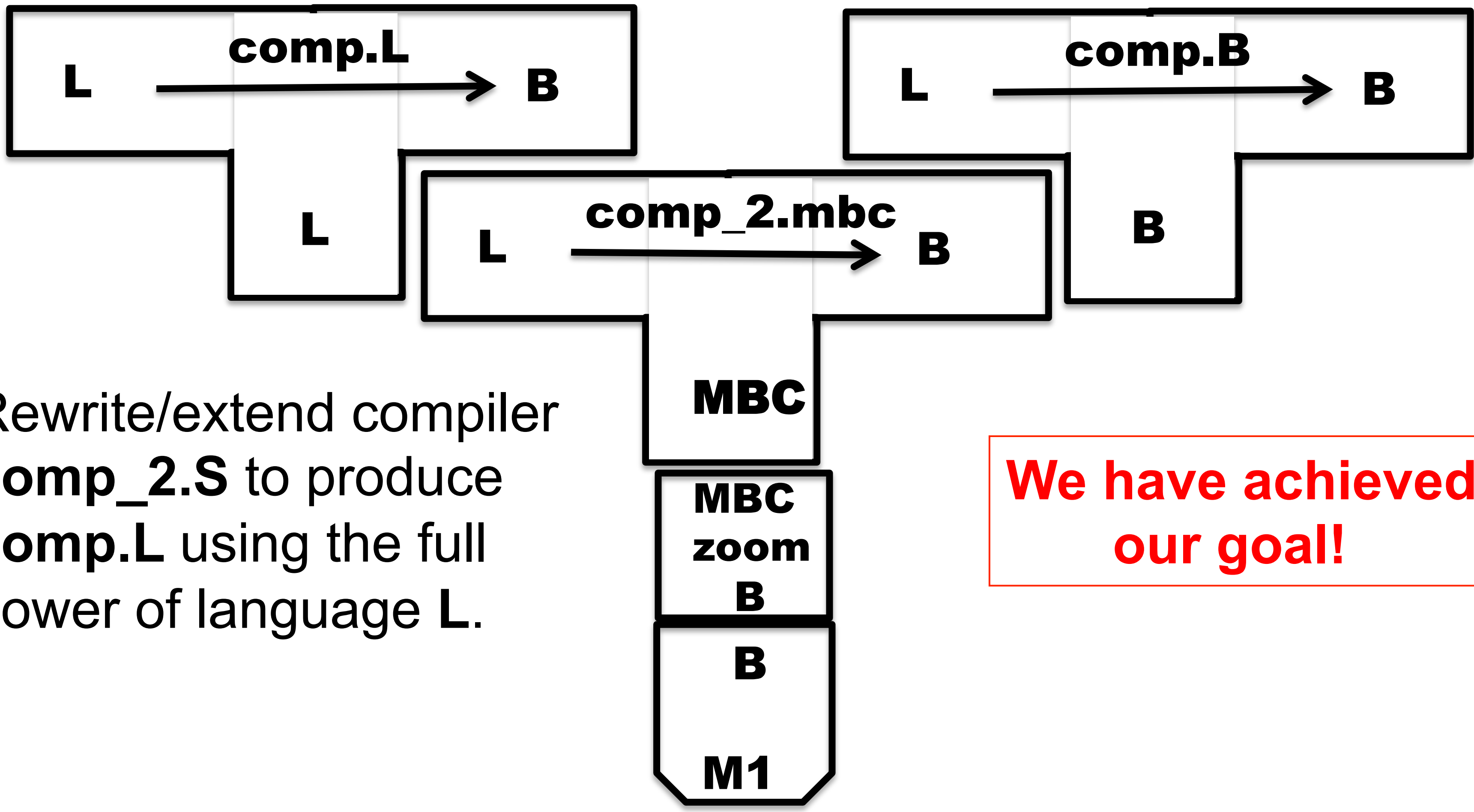
Step 3
Write a compiler for L in S



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!

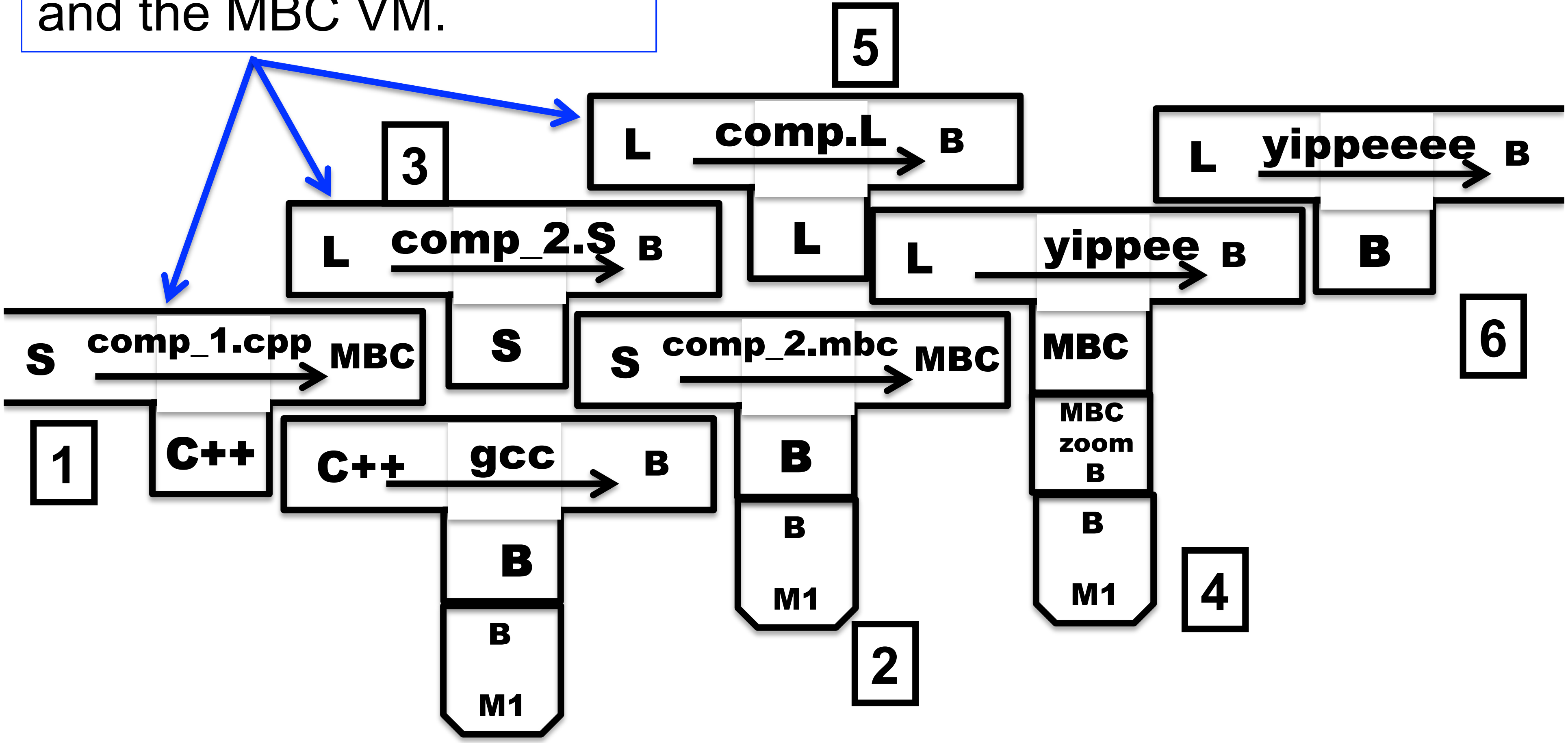


Rewrite/extend compiler **comp_2.S** to produce **comp.L** using the full power of language **L**.

We have achieved our goal!

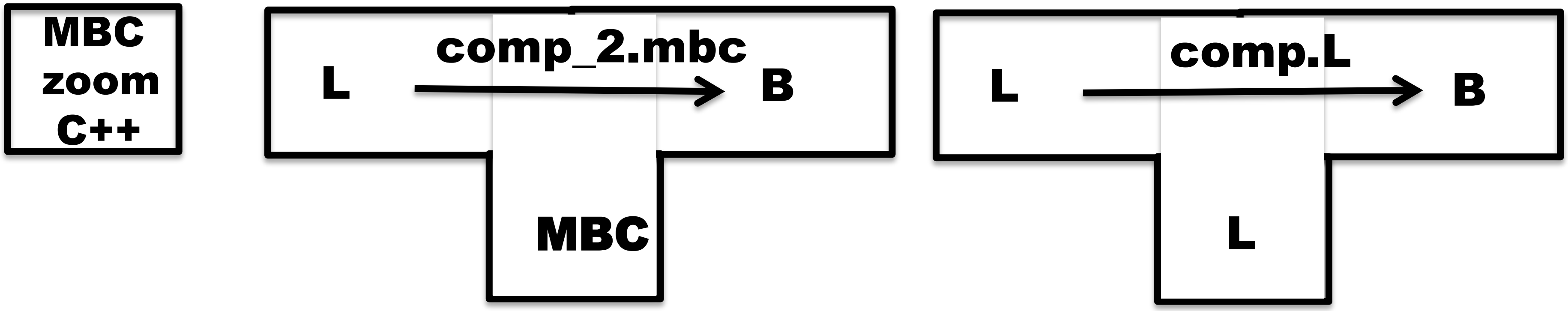
Putting it all together

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.