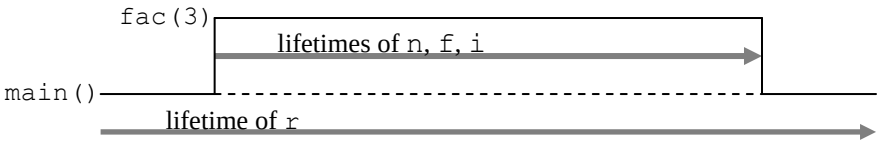
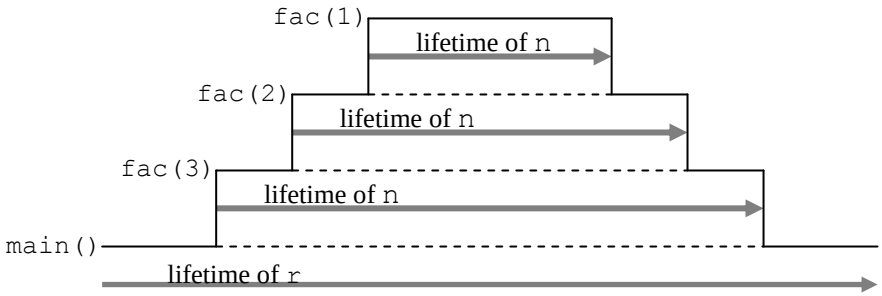
Exercises 9 (Variables and storage) – Solutions

9A. *(Lifetimes of global and heap variables)*

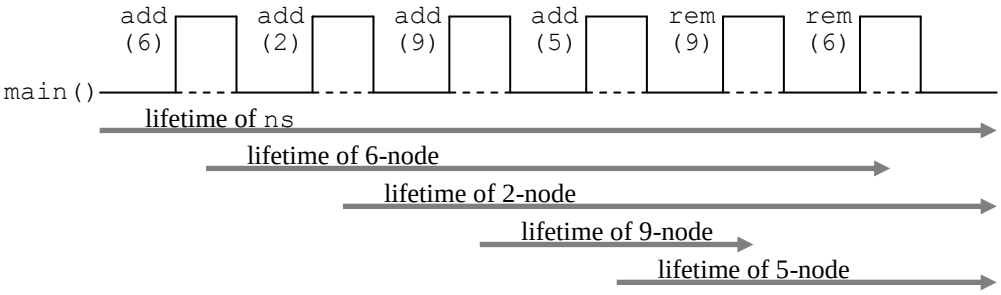
(a) Non-recursive version:



(b) Recursive version:



9B. *(Lifetimes of global and heap variables)*



9C. *(Equivalent commands)*

In Java:

```
C ;                          ≡  C
; C                          ≡  C
if (E) C                     ≡  if (E) C else ;
if (true) C1 else C2         ≡  C1
if (false) C1 else C2        ≡  C2
while (E) C                  ≡  if (E) {
                                  C
                                  while (E) C
                                }
do C while (E);              ≡  C
                                if (E) {
                                  do C while (E);
                                }
                             ≡  C
                                while (E) C
```

```
switch (E) C {          ≡  { int v = E;
  case l1 : C1                 if (v == l1) { C1 C2 ... Cn}
  case l2 : C2                 else if (v == l2) { C2 ... Cn}
  ...                          ...
  case ln : Cn                 else if (v == ln) { Cn}
}                           }
```

9D. *(Expressions with side effects)*

(a) Advantages and disadvantages of side effects:

+ concise coding
– obscure coding.

(b) To prevent a C non-void function from having side effects, we must ensure that the function does not assign to a non-local variable. In particular, the function should contain (i) no assignment that updates a non-local variable, and (ii) no call to a void function.

A C compiler could not enforce restriction (i), since in an assignment like "`p->f` = …" it could not tell whether `p` is currently pointing to a local, global, or heap variable.

These restrictions would reduce the language's expressive power. Restriction (i) would prevent harmless updates to heap variables that are destroyed before the function returns. Restriction (ii) would prevent calls to harmless void functions.

Exercises 10 (Bindings and scope) – Solutions

10A. *(Environments)*

Environments at numbered points in the C program:

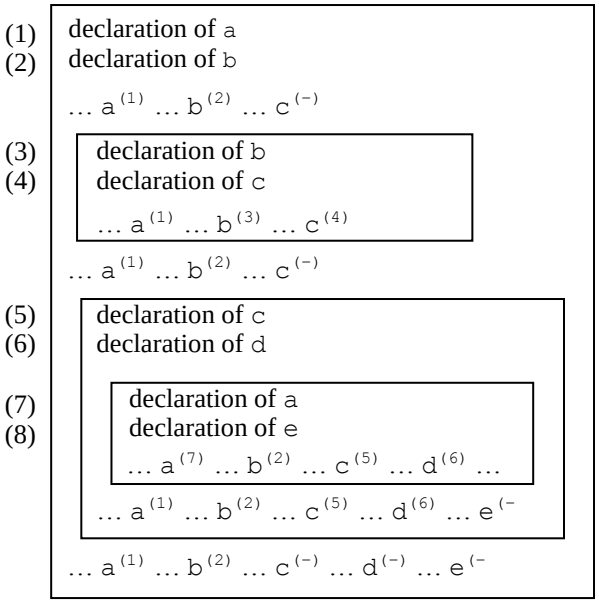
(1) { `n` → an INT global variable,
`zero` → a (VOID → VOID) function }

(2) { `d` → an INT parameter,
`inc` → an (INT → VOID) function,
`n` → an Integer global variable,
`zero` → a (VOID → VOID) function }

(3) { `argc` → an INT parameter,
`argv` → a POINTER parameter,
`inc` → an (INT → Void) function,
`main` → an (INT × Pointer → Void) function,
`n` → an INT global variable,
`zero` → a (VOID → VOID) function }

10B. *(Block structure)*

In the following diagram, each applied occurrence is superscripted to indicate the corresponding binding occurrence, or marked "(–)" if there is no corresponding binding occurrence.



10C. *(Static vs dynamic scoping)*

(a) If the language is *dynamically* scoped, 21 will be printed at point (1), whilst 22 will be printed at point (2).

(b) If the language is *statically* scoped, the program will fail to compile, since the applied occurrence of `d` on line 2 has no corresponding declaration.

10D. *(Initializing variable declarations)*

Advantages and disadvantages of compulsory initialization of variables:

\+ Variables never have undefined values.

– Initialization in a variable declaration is a waste of time if the program never uses the initial value.

10E. *(C type declarations)*

(a) C has **`typedef`**, **`enum`**, **`struct`**, and **`union`** type declarations.

(b) Possible redesigned syntax:

```
decl  =  …
      |  type id ('=' expr )? ';'          (variable declaration)
      |  'type' id '=' type ';'            (type definition)
type  =  'char'
      |  'int'
      |  'float'
      |  …
      |  id
      |  type '[' ']'
      |  'enum' '{' id (',' id)* '}'
      |  'struct' '{' type id (',' type id)* '}'
      |  'union' '{' type id (',' type id)* '}'
```