Solutions to Exercises 3

3.3.1 Composite variables in ADA:

Variables of record and array types are composite. Record and array variables can be selectively updated by assignments of the form:

R`.`I `:=` E_1`;` -- selectively updates record variable R

A`(`E_2`)` `:=` E_3`;` -- selectively updates array variable A

They can also be totally updated:

R `:=` E_4`;` -- totally updates record variable R

A `:=` E_5`;` -- totally updates array variable A

Both static and dynamic arrays are supported: all arrays of type **`array`** `(`S`)` **`of`** T have the same index range, whilst different arrays of type **`array`** `(`S **`range`** `<>)` **`of`** T may have different index ranges.

* **3.3.2** Consequences of treating strings as arrays of characters:

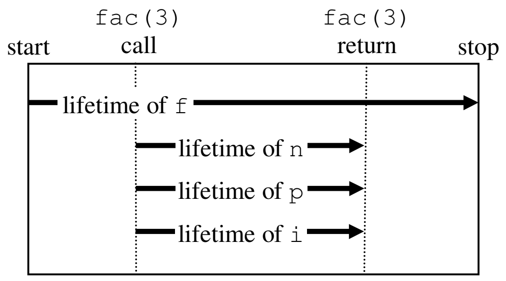
(a) When string variables are static arrays, we have fixed-length strings whose lengths are literals. In practice we need a space-padding convention to achieve the effect of bounded-length strings. Concatenation is awkward to program: what is the type of the result?

(b) When string variables are dynamic arrays, we have fixed-length strings whose lengths are variables. This is only slightly better than (a).

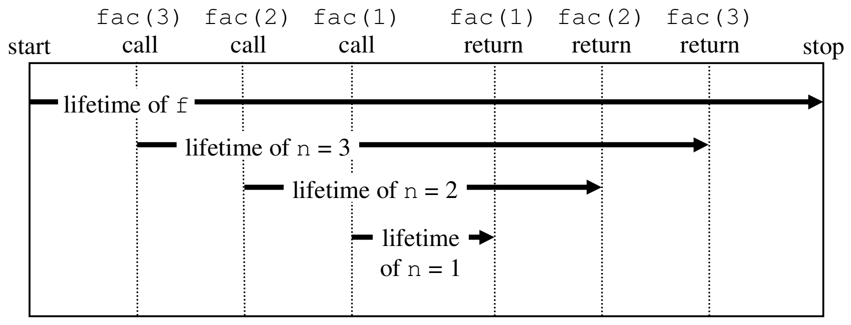
(c) When string variables are flexible arrays, we have variable-length strings. This is much more convenient. Operations such as concatenation can be programmed easily.

3.5.2 Lifetimes of the local variables:

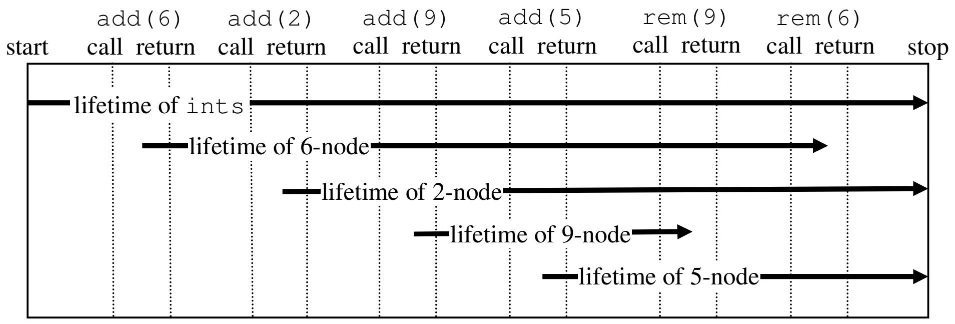
(a) non-recursive version:



(b) recursive version:



3.5.3 Lifetimes of the heap variables and global variable:



** **3.6.2** Possible redesign of ADA to enable programmers to define recursive types directly:

Introduce a recursive type definition, e.g.:

```
type IntList is
     recursive record
       head: Integer;
       tail: IntList;
     end record;
```

Treat **null** as a value of every recursive type. E.g.:

```
qs: IntList := null;
ps: IntList := (2, (3, (5, (7, null))));
…
put(ps.head);         -- writes 2
put(ps.tail.head);    -- writes 3
```

Recursive types could be represented by linked structures. E.g., an `IntList` value would be a pointer to a node containing a `head` field and a `tail` field. (As in HASKELL and JAVA, however, programmers would be unable to manipulate pointers directly.)

Copy semantics for assignment would entail copying an entire list (or tree), which would be complicated and expensive.

Reference semantics would entail copying only a pointer. However, this would lead to anomalies if selective updating were allowed. E.g.:

```
qs:= ps;
qs.head := 1;   -- updates ps as well as qs
```

To avoid this, it would be reasonable to prohibit selective updating of variables of recursive type.

3.7.1 Forms of commands in ADA:

ADA has a skip command, assignment commands, procedure calls, sequential commands, if-commands, case commands, while-commands, for-commands, basic-loop commands, return commands, exit commands, and goto-commands. Return commands are needed for procedures and functions. Exit commands enable single-entry-multi-exit control structures to be programmed. Goto-commands enable arbitrary control structures to be programmed, but they are superfluous in practice. The only form of command covered in Section 3.7 but missing in ADA is the collateral command.

** **3.7.2** A possible extension to ADA to support list types is illustrated below:

```
type IntList is list of Integer;
qs: IntList := [];
ps: IntList := [2, 3, 5, 7];
...
for i in ps loop
   put(i);
end loop;
for i in [1 .. n] loop
   put(i*i);
end loop;
```

There should also be operators or functions to return the head, tail, length, etc., of a given list.

* **3.7.3** Some equivalences between commands (using C/C++/JAVA syntax):

```
C ;              ≡   C

; C              ≡   C

if (E)           ≡   if (E)
   C                    C
                     else
                        ;

if (true)        ≡   C1
   C1
else
   C2

if (false)       ≡   C2
   C1
else
   C2

while (E)        ≡   if (E)  {
   C                    C
                        while (E)
                           C
                     }
```

* **3.7.5** In an ADA for-command:

(a) After termination of the loop, the control variable has been destroyed.

(b) After a jump or escape out of the loop, the control variable has been destroyed.

(c) The loop body cannot assign to the control variable, since the latter is actually treated as a constant inside the loop body.

* **3.8.1** Advantages and disadvantages of allowing side effects in expressions:

+ A very concise programming style becomes possible.

+ The useful properties of expressions (composition) and commands (updating) are combined.

+ There is no need to distinguish between proper procedures and function procedures.

– The distinction between expressions and commands is blurred.

– Programs become harder to understand.

* **3.8.2** To prohibit side effects in ADA functions, ensure that no function body contains:

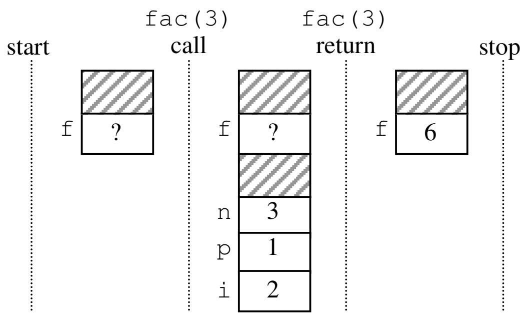
(i) an assignment command that updates a global variable or heap variable;

(ii) a procedure call.

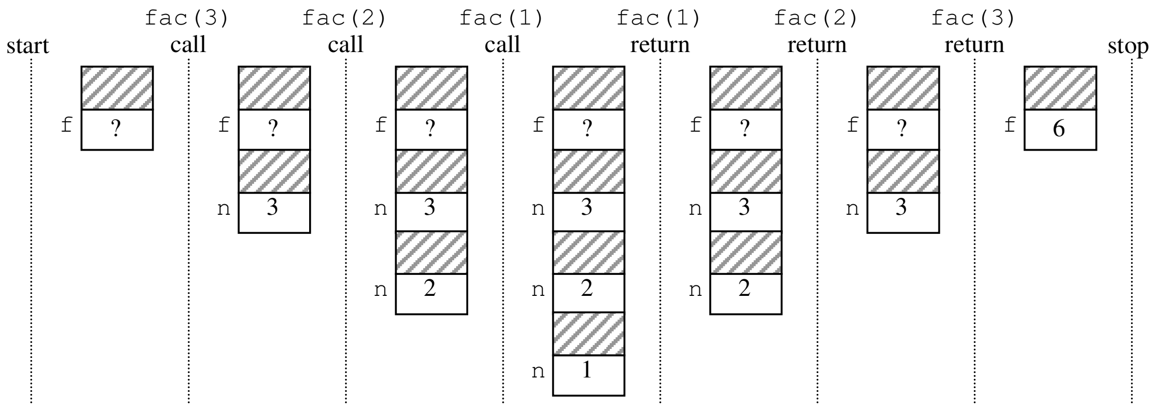
These restrictions are enforceable by the compiler. They do reduce the language's expressive power. In particular, (ii) is a bit heavy-handed, even forbidding calls to procedures that update only variables passed as (in-)out-parameters.

To prohibit side effects in C non-void functions is much more difficult, since the compiler cannot tell whether a given pointer is currently pointing to a global, local, or heap variable, so what would be the analogue of (i) above?

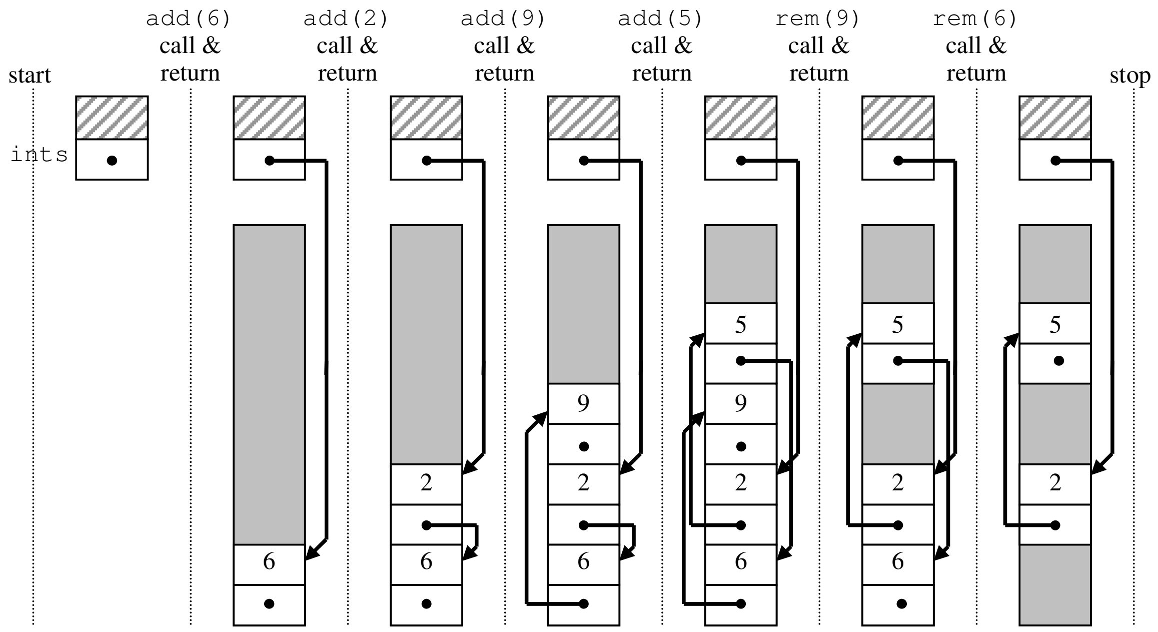
3.9.1 Allocation of storage in the program of Exercise 3.5.2(a):



Allocation of storage in the program of Exercise 3.5.2(b):



3.9.2 Allocation of heap storage in the program in Exercise 3.5.3:


add(6)
call &
return
add(2)
call &
return
add(9)
call &
return
add(5)
call &
return
rem(9)
call &
return
rem(6)
call &
return
start
stop
ints
6
2
9
5