Solutions to Exercises 4

4.1.1 Environment at each numbered point in the C program:

(1) { `n` → an integer variable }

(2) { `n` → an integer variable,
`zero` → a procedure with no parameter }

(3) { `n` → an integer variable,
`zero` → a procedure with no parameter }

(4) { `n` → an integer variable,
`zero` → a procedure with no parameter,
`inc` → a procedure with an integer parameter,
`d` → an integer parameter }

(5) { `n` → an integer variable,
`zero` → a procedure with no parameter,
`inc` → a procedure with an integer parameter }

(6) { `n` → an integer variable,
`zero` → a procedure with no parameter,
`inc` → a procedure with an integer parameter,
`main` → a procedure with an integer parameter and a pointer-to-pointer-to-character parameter,
`argc` → an integer parameter,
`argv` → a pointer-to-pointer-to-character parameter }

4.1.2 Environment at each numbered point in the ADA program:

(1) { `max` → integer 9999 }

(2) { `max` → integer 9999, `Nat` → type {0, …, 999} }

(3) { `max` → integer 9999, `Nat` → type {0, …, 999},
`m` → a `Nat` variable, `n` → a `Nat` variable }

(4) { `max` → integer 9999, `Nat` → type {0, …, 999},
`m` → a `Nat` variable, `n` → a `Nat` in-parameter,
`func` → a function with a `Nat` in-parameter }

(5) { `max` → integer 9999, `Nat` → type {0, …, 999},
`m` → a `Nat` variable, `n` → a `Nat` variable,
`func` → a function with a `Nat` in-parameter }

(6) { `max` → integer 9999, `Nat` → type {0, …, 999},
`m` → a `Nat` in-parameter, `n` → a `Nat` variable,
`func` → a function with a `Nat` in-parameter,
`proc` → a procedure with a `Nat` in-parameter }

(7) { `max` → integer 9999, `Nat` → type {0, …, 999},
`m` → a `Nat` in-parameter, `n` → integer 6,
`func` → a function with a `Nat` in-parameter,
`proc` → a procedure with a `Nat` in-parameter }

(8) { `max` → integer 9999, `Nat` → type {0, …, 999},
`m` → a `Nat` variable, `n` → a `Nat` variable,
`func` → a function with a `Nat` in-parameter,
`proc` → a procedure with a `Nat` in-parameter }

4.2.2 Static *vs* dynamic scoping:

(a) If the language is dynamically scoped, 21 would be printed at (1), and 22 would be printed at (2).

(b) If the language is statically scoped, the program will fail to compile: the applied occurrence of `d` in `add` has no corresponding binding occurrence.

4.3.1 Advantages and disadvantages of compulsory initialization of variables:

\+ No variable ever contains *undefined*. This eliminates a common source of error, without the overhead of run-time checks.

– If a variable's default initialization is immediately followed by an explicit assignment to that variable, the default initialization is a waste of time.

4.3.2 C has four distinct forms of type definition: **`typedef`**, **`enum`**..., **`struct`**, **`union`**.

A possible redesign would be to replace all these by a single form of type definition:

`type` *I* **`=`** *T***`;`**

and to introduce a single form of variable declaration:

T *I*$_1$ **`=`** *E*$_1$**`, ..., `** *I*$_n$ **`=`** *E*$_n$**`;`**

(where the initialisers are optional). The following forms of type denoter would be supported:

Type denoter (T)	*Explanation*
`int`, **`float`**, etc.	denote primitive types
`enum` {*I*$_1$**`, ...,`** *I*$_n$}	denotes an enumeration type with n distinct values
`struct` {*T*$_1$ *I*$_1$**`, ...,`** *T*$_n$ *I*$_n$}	denotes a structure type with fields of types T_1, ..., T_n
`union` {*T*$_1$ *I*$_1$**`, ...,`** *T*$_n$ *I*$_n$}	denotes a union type with variants of types T_1, ..., T_n
I	denotes the type bound to identifier *I*