變數與繫結

資科系
林偉川

變數的定義

- 名稱(name)：
  - 變數的名字，但是指標變數的名稱僅是引用該變數的方式並非真正名稱
- 屬性(attribute)：
  - 變數的性質，或稱為型態如整數、實數、字元、字串
- 引用(reference)：
  - 變數的貯存位址，或稱為位址
- 值(value)：
  - 變數的值

2

儲存區配置問題

- 儲存區配置的方式有二種:
  - 靜態記憶體配置(static storage allocation):

    程式執行前(compile time)，配置記憶體空間給變數。

    如 Fortran、Cobol、Pascal 的全域變數與 PL/1、C 的 static 記憶指令。
  - 動態記憶體配置(dynamic storage allocation):

    程式執行時(run time)，配置記憶體空間給變數。

    如 Algol 60、APL、LISP、Snobol 與 Pascal 的區域變數。

3

參考的透明性 (referential transparency)

- 意義:
  - 程式執行的過程中計算運算式後，應只能得到一個值，不能更改或破壞原來執行的環境，這種現象便是參考的透明性。

```
void sub(int *y) { *y=*y+3; }
void main() { int x=1; sub(&x); }
```

  - 傳址呼叫來傳遞參數，x與*y佔相同的記憶體位址，執行完畢後也會將x之值加以改變，故此程式不具參考的透明性

4

捷徑計算 (short circuit evaluation)

- 對運算式作求值動作時，無需做完整個運算式即可得出最後的結果，如此的計算方式便稱為捷徑計算。
- 例如：
  - A1*A2*…*A100中，若A1為0則其他數值不需再計算，其結果為0
  - A and B and C中，若A為false，則其他運算式不需再運算，其結果為false
  - A or B or C中，若A為true，則其他運算式不需再運算，其結果為true

5

完全計算(complete circuit evaluation)

- 對運算式作求值動作時, 必需做完整個運算式即可得出最後的結果, 如此的計算方式便稱為完全計算。

6

各種語言對捷徑計算的處理方式

- PASCAL：以內定方式來決定採用捷徑計算或完全計算，若在程式中加入{$B-}，則在該敘述後之所有布林運算是將採用捷徑計算，若在程式中加入{$B+}，則在該敘述後之所有布林運算是將恢復完全計算
- C與C++及JAVA則內定採用捷徑計算
- ADA：and / or(完全計算), and then / or else (捷徑計算)
- VB.NET：and / or(完全計算), AndAlso、OrElse (捷徑計算)

7

捷徑計算範例

- 底下的 Pascal 程式片段企圖從一陣列中搜尋某 key值的所在位置。請說明為何這程式片段可能有誤。其中 n 為陣列 a 中的元素個數。

```
{$B-}
i:=1;
while (i<=n) and (a[i]<>key) do i:=i+1;
```

- Pascal 程式採用完全計算，當值i為n+1，則會有out of range錯誤
- 對運算式作求值動作時,無需做完整個運算式即可得出最後的結果,如此的計算方式便稱為捷徑計算。

8

避免完全計算錯誤範例

- 請改寫上述程式片段以避免所提之問題。

```
{$B+}
 i:=1;  flag:=false;
 while (i<=n) and (flag=false) do
 begin
   If(a[i]< >key) then i:=i+1 else flag:=true
 end;
```

9

捷徑計算範例

- 以下的C程式段執行的結果變數 b 與 c 的值各別為何?

(A)b=11 c=0 (B) b=10 c=1 (C) b=11 c=1 (D) b=10 c=0

```
void main() {
  int a=1,b=10
  c=((a>1) && (b++));
}
```

答：D，因為a>1已為不成立，故b++不會執行

10

捷徑計算範例

- ADA利用不同的運算子來代表採用捷徑計算及完全計算，and then 及 or else代表採用捷徑計算，但是and 及or 則代表採用完全計算
  a or else b or else c及a and then b and then c
- JAVA利用|| 及 &&來代表採用捷徑計算，而 | 及 &則代表採用完全計算

11

繫結(binding)

- 意義:

  繫結是指變數的名稱與其位址,屬性或值相結合的動作。

12

四種不同的繫結時間

- 程式語言設計時的繫結
  - 對象為程式語言所提供的所有可能資料型態及符號的意義。如敘述，資料結構的可能型式。
- 程式語言製作時的繫結:
  - 對象為所有跟機器相關的特性及變數的可能值。程式語言所提供在不同機器上，即使相同的數學運算式，亦可能得到不同的結果。

13

四種不同的繫結時間

- 翻譯時的繫結
  - 對象為變數的位址(採用靜態儲存(static)配置法)及變數的真正的型態(採用外顯式(explicit)型態法)
- 執行時的繫結
  - 對象為變數的位址(採用動態儲存(dynamic)配置法)及變數的真正的型態(採用內隱式(implicit)型態法)，及變數真正的值

14

繫結時間範例

- 以C程式語言指令y=x+100;
  a. “=“為指定運算子
  b. “+“為加法運算子
  c. “+“為整數加法
  d. “100”表示整數100
  e. 整數100 的內部表示法
  f. 100為整數
  g. x, y的資料型態
  h. x, y的值
  i. x, y的位址
  j. x, y可以宣告的型態

15

繫結時間說明

- a. “=“為指定運算子，在程式語言設計時決定
  b. “+“為加法運算子，在程式語言設計時決定c. “+“為整數加法須等到運算元的型態決定後才能決定，而運算元型態需在編譯時才能決定，故為翻譯時繫結
  d. “100”表示整數100 ，在程式語言設計時決定
  e. 整數100 的內部表示法，涉及機器相關的實作，故為語言製作時繫結

16

繫結時間說明

- f. 100為整數為符號代表的意義，在程式語言設計時決定，故為程式語言設計的繫結
  g. x, y的資料型態，須在編譯時才能決定，故為翻譯時繫結
  h. x, y的值通常在執行時決定，故為執行時繫結
  i. x, y的位址通常在執行時決定，故為執行時繫結
  j. x, y可以宣告的型態，在程式語言設計時決定，故為程式語言設計時繫結

17

其它應注意

- **變數名稱之設計考量**
  - **Maximum length? Java : 255**
  - **Are connector characters allowed? _ 或 - 或 $**
  - **Are names case sensitive?**
  - **Are special words reserved words or keywords?**
- **變數名稱之長度**
  - **FORTRAN I: maximum 6**
  - **COBOL: maximum 30**
  - **FORTRAN 90 and ANSI C: maximum 31**
  - **Ada: no limit, and all are significant**
  - **C++: no limit, but implementors often impose one**

18

其它應注意

- 變數名稱之大小寫敏感之缺點:
  - readability (names that look alike are different)
  - worse in Modula-2 because predefined names are mixed case (e.g. WriteCard)
  - C, C++, Java, and Modula-2 names are case sensitive
  - The names in other languages are not
- 變數名稱之*Special words*
- 定義: A *keyword* is a word that is special only in certain contexts
- 定義: A *reserved word* is a special word that cannot be used as a user-defined name
- 如果沒有保留字、關鍵字、特殊字的缺點: poor readability

19

其它應注意

- 變數名稱為an abstraction of a memory cell
- 變數名稱可有下列六個特徵：
  - 變數名稱, 位址, 值, 資料型別, 生命週期, and 範圍
- 位址 - memory address with which it is associated
  - A variable may have different addresses at different times during execution
  - A variable may have different addresses at different places in a program
  - If two variable names can be used to access the same memory location, they are called *aliases*
  - Aliases are harmful to readability

20

其它應注意

- ***How aliases can be created?***
  - **Pointers, reference variables, Pascal variant records, C and C++ unions, and FORTRAN EQUIVALENCE (and through parameters)**
  - **Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN**
  - **replace them with dynamic allocation**

21

其它應注意

- **資料型別-** determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
- **值 -** the contents of the location with which the variable is associated
  - ***Abstract memory cell* - the physical cell or collection of cells associated with a variable**
- **The *l-value* of a variable is its address**
- **The *r-value* of a variable is its value**

22

其它應注意

- ***Def*: A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol**
- ***Def*: *Binding time* is the time at which a binding takes place.**

23

其它應注意

- **Possible binding times:**
  - **Language design time - bind operator symbols to operations (+ for add and string concatenation)**
  - **Language implementation time - bind floating point type to a representation**
  - **Compile time -- bind a variable to a type in C or Java**
  - **Load time -- bind a FORTRAN 77 variable to a memory cell (or a C static variable)**
  - **Runtime -- bind a non-static local variable to a memory cell**

24

其它應注意

- ***Def*: A binding is *static* if it occurs before run time and remains unchanged throughout program execution.**
- ***Def*: A binding is *dynamic* if it occurs during execution or can change during execution of the program.**

25

其它應注意

- **Type Bindings**
  - **How is a type specified?**
  - **When does the binding take place?**
- **Static binding, type may be specified by either an explicit or an implicit declaration**
- **Def: An *explicit declaration* is a program statement used for declaring the types of variables**
- **Def: An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)**

26

其它應注意

- **FORTRAN, PL/I, BASIC, and Perl provide implicit declarations**
  - *Advantage:* **writability**
  - *Disadvantage:* **reliability (less trouble with Perl)**

27

其它應注意

- **Dynamic Type Binding - type specified through an assignment statement**
  - **APL & SNOBOL4**
    - **LIST <- 2 4 6 8**
    - **LIST <- 17.3**
  - *Advantage:* **flexibility (generic program units)**
  - *Disadvantages:*
    - **High cost (dynamic type checking and interpretation)**
      - **i (int) := x (int)**
      - **i (int) := y (float)**
    - **Type error detection by the compiler is difficult**

28

其它應注意

- **Type Inferencing (ML, Miranda, and Haskell)**
  - **Rather than by assignment statement, types are determined from the context of the reference**
    - **Fun cir(r)=3.14159*r*r ➔ real number assign**
    - **Fun tim10(r)=10*r ➔ integer number assign**
    - **Fun squ(r)=r*r ➔ error assign**
    - **Fun squ(r: int)=r*r ➔ error assign**

29

其它應注意

- **Storage Bindings**
  - ***Allocation* - getting a cell from some pool of available cells**
  - ***Deallocation* - putting a cell back into the pool**
- **Def: The *lifetime* of a variable is the time during which it is bound to a particular memory cell**

30

其它應注意

- **Categories of variables by lifetimes**
  1. ***Static*** **- bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.**
     - **All FORTRAN I、II、IV variables, C static variables**
     - ***Advantage:*** **efficiency (direct addressing), history-sensitive subprogram support**
     - ***Disadvantage:*** **lack of flexibility (no recursion)**

31

其它應注意

2. **Stack-dynamic - Storage bindings are created for variables when their declaration statements are elaborated whose types are statically bound.**
   - **If scalar, all attributes except address are statically bound**
   - **Local variables in Pascal and C/C++ subprograms**
   - ***Fortran 77 and 90 can use static-dynamic variables for locals ➔ SAVE list (variable list is static)***
   - ***Advantage:*** **allows recursion; conserves storage**
   - ***Disadvantages:***
     - **Overhead of allocation and de-allocation**
     - **Subprograms cannot be history sensitive**
     - **Inefficient references (indirect addressing)**

32

其它應注意

3. **Explicit heap-dynamic – Nameless memory cells allocation and deallocated by explicit run-time instructions specified by the programmer, allocate from and deallocated to heap which take effect during execution ()**
   - **Referenced only through pointers or references**
   - **Dynamic objects in C++ (via new and delete) all objects in Java**
     int *node; … node=new int; … delete node;
   - **In java, all data except the primitive scalars are objects. Java objects are explicit heap dynamic and are accessed through reference variables**

33

其它應注意

- ***Advantage:*** **provides for dynamic storage management**
- ***Disadvantage:*** **difficulty of using pointer and reference variables correctly, cost of references to variables (inefficient and unreliable)**

34

其它應注意

4. **Implicit heap-dynamic - Allocation and deallocation caused by assignment statements**
   - All variables in APL and ALGOL 68 (flex a;)
   - *Advantage:* flexibility and allowing highly generic code to be written
   - *Disadvantages:*
     - Runtime overhead of maintaining all the dynamic variables
     - Inefficient, because all attributes are dynamic
     - Loss of error detection

35

其它應注意

- Type Checking - Generalize the concept of operands and operators to include subprograms and assignments
- Def: *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- Def: A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type. This automatic conversion is called a *coercion*.

36

其它應注意

- **Def: A *type error* is the application of an operator to an operand of an inappropriate type**
  - **If all type bindings are static, nearly all type checking can be static**
  - **If type bindings are dynamic, type checking must be dynamic**
- **Def: A programming language is *strongly typed* if type errors are always detected**
- ***Advantage of strong typing:* allows the detection of the misuses of variables that result in type errors**

37

其它應注意

- **It is much better to detect errors at compile time than at run time because the earlier correction is usually less costly**
- **Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution**
- **Languages characteristics of type checking:**
  - **FORTRAN 77 is not: parameters, EQUIVALENCE**

38

其它應注意

- Pascal is not: variant records
- Modula-3 is not: has a predefined procedure named LOOPHOLE that is used to type checking
- C and C++ are not strongly typed checking: parameter type checking can be avoided; unions are not type checked
- Ada is, almost strongly typed (UNCHECKED_CONVERSION library function is used not to do the type checking) (Java is similar)

39

其它應注意

- Coercion rules strongly affect strong typing – they can weaken it considerably (C++ versus Ada)
- Language with great deal of coercion, like Fortran, C, and C++, are significantly less reliable than those with little coercion, such as Ada
- Java has half as many kinds of type coercions as C++
- Not all type errors can be detected by static type checking

40

其它應注意

- **Type Compatibility**
- ***Name type compatibility* and *Structure type compatibility***
- **Most languages use combinations of these two different techniques**
- **Def: *Name type compatibility* means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name**

41

其它應注意

- **Easy to implement but highly restrictive:**
  - **Subranges of integer types are not compatible with integer types in PASCAL**
    type ind = 1..100;
    var
    cc : integer;
    id : ind;
    cc & id would not be compatible, so cc:=ind; (error!!)
  - **Formal parameters must be the same type as their corresponding actual parameters (Pascal)**

42

其它應注意

- **Def: *Structure type compatibility* means that two variables have compatible types if their types have identical structures**
  - **More flexible, but harder to implement**
- **Name type compatibility, only two type names must be compared to determine compatibility**
- **Structure type compatibility, the entire structures of the two types must be compared**

43

其它應注意

- ***Consider the problem of two structured types:***
  - **Are two record types compatible if they are structurally the same but use different field names?**
  - **Are two array types compatible if they are the same except that the subscripts are different? (e.g. [1..10] and [-5..4])**
  - **Are two enumeration types compatible if their components are spelled differently?**
  - **With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)**

44

其它應注意

- ***Language examples*:**
  - **Pascal: usually structure type, but in some cases name is used (formal parameters)**
  - **C: structure type, except for struct and union typedef just define a new name for existing type**
  - **Java bring another object compatibility and its relationship to be inheritance hierarchy**
  - **Ada: restricted form of name**
    - **Derived types allow types with the same structure to be different**
    - **Anonymous types are all unique, even in: A, B : array (1..10) of INTEGER; A and B Would be of anonymous but distinct and incompatible types**

45

其它應注意

- **Scope**
- **Def: The *scope* of a variable is the range of statements in which it is visible**
- **Def: The *nonlocal* variables of a program unit or block are those that are visible within the program unit or block but not declared there**
- **The scope rules of a language determine how a particular occurrence of names are associated with variables**

46

其它應注意

- **Static scope**
  - **Binding names to nonlocal variables**
  - **The scope of a variable can be statically determined prior to execute**
  - **To connect a name reference to a variable, you (or the compiler) must find the declaration**
  - ***Search process:* search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name**
  - **Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent***

47

其它應注意

Procedure big; var x : integer;
procedure sub1; begin ... x … end;
procedure sub2; var x: integer; begin … end;
begin … end;

- **Under static scoping, the reference to variable x in sub1 is to the x declared in the procedure big;**

48

其它應注意

- **Variables can be hidden from a unit by having a "closer" variable with the same name**
  Program main; var x : integer;
  procedure sub1; var x : integer;
  begin … x ... end;
  begin … end;
- **Under static scoping, the reference to variable x in sub1 is to sub1's declared x. The x declared in the main program is hidden from the code of sub1**

49

其它應注意

- **C++ and Ada allow access to these "hidden" variables**
- **In Ada hidden variables from ancestor scopes can be accessed with selective references, including the ancestor scope's name ➔ the x declared in the main program can be accessed in sub1 by the reference main.x**

50

其它應注意

- **Blocks - a method of creating static scopes inside program units – first introduced in ALGOL 60**
- **Allow a section of code to have its own local variables whose scope is minimized**
- **Typically such variables are stack dynamic i.e. their storage allocated when the section is entered and deallocated when the section is exited**

51

其它應注意

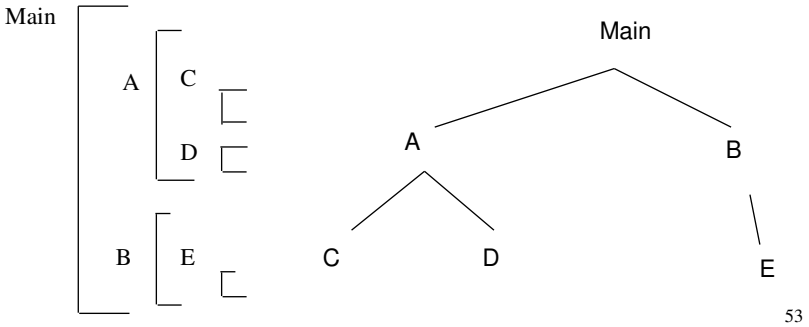
- **Examples:**
- **C and C++: for (...) { int index; ... }**
- **Ada: …**
  declare t1,t2,tt : FLOAT;
  begin tt=t1; t1=t2; t2=tt; ... end;

52

其它應注意

- **Evaluation of Static Scoping**
- **Consider the example:**

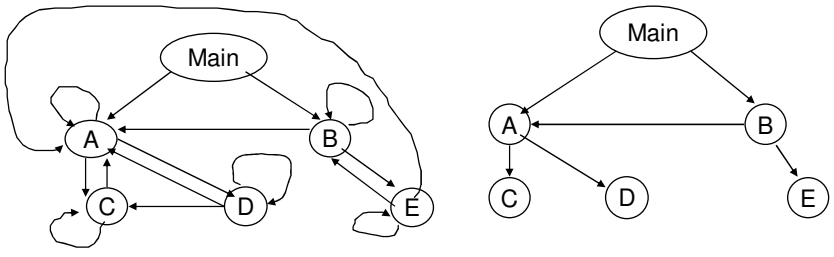
  MAIN calls A and B; A calls C and D;B calls A and E; program & tree structure is shown



53

其它應注意

- Overall scope inside main is A and B. Inside A are scope for the procedures C and D. Inside B are scope for the procedures E.
- The potential call graph is shown



54

其它應注意

- Suppose the spec is changed so that D must now access some data in B
- *Solutions:*
  - Put D in B (but then C can no longer call it and D cannot access A's variables)
  - Move the data from B that D needs to MAIN (but then all procedures can access them)
  - Same problem for procedure access!
- *Overall:* static scoping often encourages many global variables

55

其它應注意

- Dynamic Scope
  - Based on calling sequences of program units, not on their textual layout or spatial relationship to each other and can be determined only at runtime (temporal versus spatial)
  - References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

56

其它應注意

Procedure big; var x : integer;
procedure sub1; begin ... x … end;
procedure sub2; var x: integer; begin … end;
begin … end;

- **The identifier x referenced in sub1 is dynamic – cannot be determined at compile time**
- **Suppose the calling sequence is changed into ig calls sub2 which calls sub 1. The search from the local sub1 to its caller sub2 where a declaration for x is found!!**

57

其它應注意

```
MAIN
 - declaration of x
 SUB1
   - declaration of x -
   ...
   call SUB2
   ...
 SUB2
   ...
   - reference to x -
   ...
 ...
 call SUB1
 ...
```

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Static scoping - reference to x is to MAIN's x

Dynamic scoping - reference to x is to SUB1's x

58

其它應注意

- **Evaluation of Dynamic Scoping:**
  - ***Advantage:*** **convenience**
  - ***Disadvantage:*** **poor readability**
- **Scope and lifetime are sometimes closely related, but are different concepts!!**
- **Consider a static variable in a C or C++ function is statically bound to storage. So its scope is static and local to the function, but its lifetime extends over the entire execution of the program of which it is a part**

59

其它應注意

- **void print() { … }**
  void comp() { int ss; … print(); }
- **The scope of the variable ss is contained within the comp(). It does not extend to the body of print(), although print() executes in the midst of the execution comp()**
- **The lifetime of ss extends over the time during which print() executes**

60

其它應注意

- **Referencing Environments**
- **Def: The *referencing environment* of a statement is the collection of all names that are visible in the statement**
- **In a static scoped language, that is the local variables plus all of the visible variables in all of the enclosing scopes**

61

其它應注意

```
Program ex;   var a,b : integer;  …
procedure sub1; var x, y: integer; begin …① end;
procedure sub2; var x: integer; …
procedure sub3; var x: integer; begin …② end;
begin …③ end;  begin …④ end;
```

- The reference environment of ① is x and y of sub1, a and b of ex
- The reference environment of ② is x of sub3, a and b of ex (x of sub2 is hidden)

62

其它應注意

- The reference environment of ③ is x of sub2, a and b of ex
- The reference environment of ④ is a and b of ex
- Though the scope of sub1 is at a higher level than sub3, the scope of sub1 is not a static ancestor of sub3, so sub3 does not have access to the variables declared in sub1

63

其它應注意

- **A subprogram is *active* if its execution has begun but has not yet terminated**
- **In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms**
- main calls sub2, which calls sub1

```
void sub1() { int a,b;  … ① }
void sub2() { int b,c;  … ② sub1(); }
void main() { int c,d;  … ③ sub2(); }
```

64

其它應注意

- The reference environment of ① is a and b of sub1, c of sub2, d of main (c of main and b of sub2 are hidden)
- The reference environment of ② is b and c of sub2, d of main (c of main is hidden)
- The reference environment of ③ is c and d of main

65

其它應注意

- **Def: A *named constant* is a variable that is bound to a value only when it is bound to storage**
  - ***Advantages:* readability and program reliability use the name pi instead of the constant 3.14159**
- ***Languages characteristics:***
- **Ada, C++ and Java allow dynamic binding of values to name constants**
  MAX: constant integer := 2*WIDTH+1;
  declare MAX to be an integer type named constant whose value is set to the value of the expression 2*WIDTH+1

66

其它應注意

- *Pascal named constant declarations require simple values on the right side of the = operator*
- *Modula-2 and FORTRAN 90 allow constant expressions (contain previously declared name constants, constant values, and operators) to be used*
- *Pascal and Modula-2 are both used static binding of values to named constants*
- **The name constants use static binding of values are manifest constants**

67

其它應注意

- **Variable Initialization**
- **Def: The binding of a variable to a value at the time it is bound to storage is called *initialization***
- **If the variable is statically bound to storage, binding and initialization occur before run time**
- **If the storage binding is dynamic, initialization is also dynamic**

68

其它應注意

- **Initialization is often done on the declaration statement, the actual initializations take place at compile time**
- **FORTRAN initial values of variables can be specified in a DATA statement**
  REAL PI
  INTEGER SUM
  DATA SUM /0/, PI /3.14159/
- **Ada initial value is shown as followed:**
  SUM : FLOAT := 0.0;

69

其它應注意

- **In general, initialization occurs only once for static variables, but it occurs with every allocation for dynamically allocated variables, such as the local variables in an Ada procedure**

70

Exercise 1

```
Program main;   var x : integer;
procedure sub1; begin writeln('x=',x); end;
procedure sub2; var x: integer;
begin x:=10; sub1; end;
begin x:=5; sub2; end;
```

- Assume the program was compiled and executed using static scoping rule what value of x is printed? Under dynamic scoping rules, what value of x is printed?

71

Exercise 2

```
Program main;   var x, y, z : integer;
procedure sub1; var a, y, z: integer;
procedure sub2; var a, b, z : integer; begin …end;
begin … end;
procedure sub3; var a, x, w: integer; begin … end;
begin … end;
```

- List all the variables, along with the program units where they are declared, that are visible in the bodies of sub1, sub2, and sub3, assuming static scoping is used?

72

Exercise 3

```
Program main;   var x, y, z : integer;
 procedure sub1; var a, y, z: integer; begin … end;
 procedure sub2; var a, x, w : integer;
 procedure sub3; var a, b, z: integer; begin … end;
 begin … end; begin … end;
```

- List all the variables, along with the program units where they are declared, that are visible in the bodies of sub1, sub2, and sub3, assuming static scoping is used?

73

Exercise 4

```
void fun1(void); void fun2(void); void fun3(void);
 void main() { int a,b,c; … }
 void fun1(void) { int b,c,d; … }
 void fun2(void) { int c,d,e; … }
 void fun3(void) { int d,e,f; … }
```

74

Exercise 4

- Consider the skeletal C program, given the following calling sequence and assuming dynamic scoping is used, what variables are visible during execution of the last function called?
  a. main calls fun1, fun1 calls fun2, fun2 calls fun3
  b. main calls fun1, fun1 calls fun3
  c. main calls fun2, fun2 calls fun3, fun3 calls fun1
  d. main calls fun3, fun3 calls fun1
  e. main calls fun1, fun1 calls fun3, fun3 calls fun2
  f. main calls fun3, fun3 calls fun2, fun2 calls fun1

75

Exercise 5

```
void fun(void) { int a,b,c; /* def 1 */  …
  while (…) { int b,c,d; /* def 2 */ … ①
    while (…) { int c,d,e; /* def 3 */ … ② }
     …③ }
  … ④ }
```

- Consider the skeletal C program as above, for each of the four marked points in this function, list each visible, along with the number of the def statement that defines it.

76

Exercise 6

```
Program main;   var x, y, z : integer;
 procedure sub1; var a, y, z: integer; begin … end;
 procedure sub2; var a, b, z : integer; begin … end;
 procedure sub3; var a, x, w: integer; begin … end;
 begin … end;
```

77

Exercise 6

- Given the following calling sequence and assuming dynamic scoping is used, what variables are visible during execution of the last subprogram activated?
  a. main calls sub1, sub1 calls sub2,sub2 calls sub3
  b. main calls sub1, sub1 calls sub3
  c. main calls sub2, sub2 calls sub3,sub3 calls sub1
  d. main calls sub3, sub3 calls sub1
  e. main calls sub1, sub1 calls sub3,sub3 calls sub2
  f. main calls sub3, sub3 calls sub2, sub2 calls sub1

78