Chapter 5

Names, Bindings, Type Checking, and Scopes

"A variable can be characterized by a collection of properties, or attributes, the most important of which is type. The design of the data types of a language requires that a variety of issues be considered : scope, lifetime of variables, type checking, and initialization".

5.1 Introduction

- ***Imperative Languages*** 폰 노이만 아키텍쳐를 추상화 시켜논 거다
  - ***are abstraction of the underlying von Neumann computer architecture***
    - ⇔ ***memory : stores both instruction and data***
    - ⇔ ***CPU : provides the operations for modifying the contents of the memory***
  - ***Variable*** **in an imperative language**
    - **⇔ an abstraction of memory cell** 변하는 속성(dynamic)과 변하지 않는 속성(static)
    - **⇔ can be characterized by a collection of properties (or *attributes*)**
      - **⇒ data types, scope, life time, type checking, initialization,…**

C언어에서는 타입은 static이고
값은 dynamic

- **How well the data types match the real-world problem space ?**

reliable하게 데이터를 처리하려고 타입을 도입
잘못된 계산 등을 못하게 하기 위해

5.2 Names

– a name is a string of characters used to identify some entities (variables, labels, subprograms, and formal parameters) in a program

- Design Issues
  – What is the maximum length of a name ?
  – Can connector (underscore) characters be used in names ?
  – Are names case sensitive ?
  – Are the special words, reserved words or keywords ? 프로그램에서 사용하는 단어들을 변수 이름으로 사용할 수 있는가
- Name Forms
  – *Name length*
    - ⇔ Earliest programming languages : used single-character names (name for unknown)
    - ⇔ FORTRAN I : allowing upto 6 characters
    - ⇔ COBOL : allowing upto 30 characters
    - ⇔ C#, Ada, and Java: no limit, and all are significant
  – *Case sensitive*
    - ⇔ C (C++) and Modular recognize the difference between the cases of letters in names
    - ⇔ *Is it a good feature ?*
- Special Words : an aid to readability; used to delimit or separate statement clauses
  – *Keyword* : a word of a programming language that is special only in certain contexts
    - ⇔ In FORTRAN
      - ⇒ REAL APPLE
      - ⇒ REAL = 3.4

REAL 타입의 APPLE 변수
REAL이란 변수 값이 3.14
FORTRAN에는 reserved word도 변수 가능

  – *Reserved word* : a special word of a programming language that can not used as a name

(Reserved words are better than keywords for readability) (COBOL has 300 reserved words!)

5.3 Variables

- an abstraction of a computer memory cell, or a collection of cells
- can be characterized as a sextuple of attributes
  (name, address, value, type, lifetime, scope)

- ***Name***
  - Variable name is the most common names in programs
  - often referred to as identifiers
- ***Address***
  - The address of a variable is the memory address with which it associated
  - In many languages, it is possible for the same name to be associated with different addresses at different places in the program (*local variable*)
  - When more than one variable name can be used to access a single memory location, the names are called *alias*es

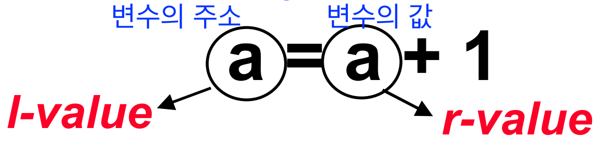
    그 메모리를 접근할 수 있는 다른 방법을 alias 라고 함

    ex C의 포인터

    alias를 사용하면 값이 바뀌는 걸 직관적으로 볼 수 없음
    폰 노이만 구조에서 변수 오른쪽에 값을 할당해서 변수 값을 바꾸는데
    alias를 허용하면 변수를 다른 방식으로 접근해서 값을 바꿀 수 있기에
    직접적으로 변수를 사용해서 값을 바꾸지 않았어도 alias로 값을 바꿨으면
    값이 바뀌니까
    ex int a = 1
    *a = 2
    직접적으로 a 변수를 바꾼건 아니지만 포인터로 바꿔서 a는 2로 됨

    - ⇔ In FORTRAN, through *EQUIVALENCE* statement
    - ⇔ In Pascal, through *variant* record structures
    - ⇔ through *subprogram parameters*
    - ⇔ through usage of *pointers*
- ***Type***
  - The type of variable determines the *range of values* the variable can have and *the set of operations* that are defined for values of the type
    - ⇔ 예) Integer in FORTRAN : -32,768~32,767, arithmetic operations

    word size - cpu가 한 머신사이클에 처리할 수 있는 크기
    32bit cpu - 한 머신 사이클에 4byte를 처리
    int는 word size다
- ***Value***
  - The value of variable is the *contents of the abstract memory cell* associated with the variable
    - ⇔ *l-value* : the address of a variable
    - ⇔ *r-value* : the value of variable



5.4 The Concept of Binding

결정되는 시간

- a **binding** is an **association**, such as between an ***attribute and an entity*** or between ***an operation and a symbol***

바인딩이 여러 시점에서 일어남

- **Binding Time** : ***the time at which a binding takes place***

한번 결정되고 변하지 않는 거 - static binding ex. C의 타입
변하는 거 - dynamic binding C의 변수값

  - **at language design time**

    *가 곱하기를 의미한다는 걸 언어를 만들 때 결정함

    ⇔ **'*' is usually bound to the multiplication operator**

C언어의 타입 바인딩 - 컴파일 타임
전역 변수의 스토리지 바인딩(메모리 위치 결정) - 로딩타임
지역 변수의 스토리지 바인딩 - 런타임(스택에 저장하기에)
변수 값 바인딩 - 런타임(해당 변수에 값을 넣을 때)

  - **at language implementation time**

    ⇔ **a data type (such as Integer) is bound to a range of possible values**

  - **at compile time**

    C언어는 컴파일 할 때 타입 유효성을 체크하고, 런타임일 때는 안함

    ⇔ **a variable in a Pascal program is bound to a particular data type**

  - **at link time**

    서브루틴이 어느 obj파일을 가리키는지 이때 결정됨

    ⇔ **a call to a library subprogram is bound to the subprogram code**

  - **at load time**

    ⇔ **a variable may be bound to a storage cell when the program is loaded into memory**

  - **at run time**

    ⇔ **a variable in a procedure may be bound to a storage cell when the procedure is actually called**

컴파일의 단위는 파일단위
한 파일에 코드를 다 쓰면 그걸 다 다시 컴파일 해야함
그래서 기능을 나눠서 다른 파일에 작성해서 파일마다 컴파일
각 컴파일된 파일을 연결해서 최종 프로그램이 됨(링킹)

예제

```
int count ;
….
count = count * 5;
sub1(count);
….
sub1(int aa) { …. }
```

main.c app1.c app2.c 파일이 어셈블러, 컴파일러를 통해
main.obj app1.obj app2.obj 기계어 파일이 되고
이 파일들을 연결해서 exe 파일이 됨
만약 app1.c이랑 app2.c 둘다에 func라는 함수가 있다면,
main - app1 으로 링크한 거랑 main - app2 으로 링크한 거랑 func의 내용이 다르다
즉 서브루틴(func)의 바인딩 타임은 링크 타임이다

(1) Binding of Attributes to Variables

- a binding is
  - *static* if it occurs *before run time* and *remain unchanged* throughout program execution
  - *dynamic* if it occurs during *run time* or *can change* in the course of program execution

(2) Type Binding

- *Static Type Binding (Variable Declaration)* :
  - *How the type is specified ?*
    - ⇔ an *explicit* declaration is a statement in a program that lists variable names and declare them to be of particular type 타입을 컴파일 때 체크하기 때문에 런타임 때는 타입 체킹 안함
      - ⇒ most programming languages designed since the mid-1960 requires explicit declaration of all variables
    - ⇔ an *implicit* declaration is a means of associating variables with types through default convention
      - ⇒ FORTRAN (`I,J,K,L,M,N`), BASIC, PL/I
- *Dynamic Type Binding* (JavaScript, Python, Ruby, PHP, and C# (limited))
  - the variable is bound to a type when it is assigned a value in an assignment statement.
  - usually implemented using interpreters because it is difficult to dynamically change the type of variables in machine code
  - Advantages 하나의 언어가 여러가지를 처리할 수 있는 언어
    - ⇔ Flexibility (Generic program)
  - Disadvantages
    - ⇔ Hard to detect errors at compile time
    - ⇔ Run time cost for type checking, and space cost for tag

타입 정보를 추가적으로 저장할 공간이 필요

```
count = 1;
…
count = 3.0;
…
count = [1,2,3] ;
```

JavaScript

변수값이 저장되는 런타임 때 타입 바인딩
그리고 실행하는 도중에 타입이 변경 될 수 있기 때문에 계속 타입 체킹을 해야 함 -> 속도가 느림,
바뀔 때마다 컴파일하기 어려움 -> 컴파일러 말고 인터프리터로 언어를 구현

- *Type Inference*
  - *in ML,*

```
fun circum(r) = 3.14*r*r;
```

(3) Storage Binding and Lifetime

- the *lifetime* of a program variable is the time during which the variable is bound to a specific memory location

address binding - 메모리가 할당되는 때

변수에 대한 메모리 셀이 언제 결정이 되냐
변수가 메모리에 머물고 있는 시간 - lifetime
ex. global 변수의 lifetime in C - 끝날 때까지
subroutine 변수의 lifetime - return 때까지

- ***Static Variables*** lifetime - 프로그램 끝까지
  - static variables are those that are bound to memory cells before execution begins and remain bound to those same memory cells until execution terminated
    - 로딩 타임 때
    - ⇔ global variables, history sensitive variable
  - advantages : efficiency
  - disadvantages : reduced flexibility (can not support recursion)

변수를 런타임 때 스택에 저장

재귀를 구현하기 위해서는 변수를 런타임 때 메모리에 저장해야 한다(stack dynamic).
fortan같은 미리 메모리를 잡는 언어는 재귀가 안됨

- ***Stack Dynamic Variable*** lifetime - subroutine 끝까지
  - stack dynamic variables are those whose storage bindings are created when their declaration statement are elaborated, but whose types are statically bound
    - ⇔ the local variables declared in a procedure
  - advantages: 제일 나중에 생성된 변수가 가장 빨리 사라지게 하기 위해 스택으로 구현된 메모리에 저장
    - ⇔ allowing recursion
    - ⇔ sharing of the same memory cells between different procedures
  - disadvantages
    - ⇔ run time overhead to allocate and deallocate the memory cells for local variables

- ***Explicit Heap Dynamic Variables (by programmer)*** lifetime - free 할 때까지
  - **Explicit dynamic variables are nameless objects whose storage is allocated and deallocated by explicit run-time instructions specified by the programmers (referenced via pointer variables)**
  - **It is bound to a type at compile time, but is bound to storage at the time it is created**
  - **Example**

```
type intnode = ^integer ;
var anode : intnode ;
. . .
new(anode) ;
. . .
dispose(anode) ;
. . . .
```

int *a,*b
a = malloc()
b = a
free(b)
a와 b는 주소를 가리켰기에
free(b)한 순간 a 주소의 메모리가 free돼서
그 다음부터 a로 접근하면 쓰레기 값이 나옴

Pascal : procedure
Ada : operator
C : function (`malloc()`)
C++ : `new` and `delete`

  - **advantages**
    - ⇔ **used for dynamic structures**

런타임 때 heap에 할당됨

heap은 stack과 다르게 알아서 변수가 없어지지 않음
free해야 함
heap에 저장된 메모리를 가리키는 포인터를 지우면 heap 영역에는 뭐가 저장되어 있지만
그걸 가리키는 포인터가 없어서 가비지가 됨

  - **disadvantages**
    - ⇔ **difficulty of using them correctly and the cost of references, allocations, and deallocations ➔ *inefficient and unreliable***

- ***Implicit Heap Dynamic Variables (by system) (JavaScript, and PHP)***
  - **Implicit dynamic variables are bound to storage only when they are assigned values**
  - **Example: (in APL)**

```
LIST = 10.2 5.1 0.0
LIST = 47
```

$\rightarrow$ ***memory allocation for list***
$\rightarrow$ ***memory allocation for integer***

  - **advantages**
    - ⇔ **high flexibility**
  - **disadvantages**
    - ⇔ **run time overhead of maintaining all the dynamic attributes, loss of error detection**

5.5 Type Checking

int i
float j
i = j 변환을 해주는가

- *Type checking* is the activity of ensuring that the operands of operator are of compatible types
- A compatible type is one that is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code to a legal type (coercion) 컴파일러에 의한 타입변환 - coercion
- If all binding of variables to types are static in a language, then type checking can nearly always be done statically (static type checking)
- Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution (Pascal variant records)

5.6 Strong Typing

- Strongly typed language 컴파일 타임 때 프로그램 내의 타입 에러를 모두 찾을 수 있는 언어
  - each name in a program in the language has a single type associated with it, and that type is known at compile time (all types are statically bound)
  - type errors are always detected
- Example
  - FORTRAN 77 : not strongly typed because the relationship between actual and formal parameters is not type checked
  - Pascal : nearly strongly typed, but fails in its design of variant record
  - C, ANSI C, C++ : not strongly typed languages because all allow function for which parameters are not type checked
  - Ada : nearly strongly typed language

printf는 서브루틴 - 여기에 매개변수를 넘겨줄 때 개수가 제한이 없음
즉 C 언어는 함수 호출 때 매개변수 개수랑 타입을 체크를 안함
함수 프로토타입잉을 하는 이유
컴파일이 파일 단위로 일어나는데, 한 파일에서 int 매개변수를 다른 파일의 서브루틴으로 넘기는데 그 서브루틴은 float 매개변수를 받도록 되있는
경우, 컴파일이 파일 단위로 일어나기 때문에 문제가 없다
위에 프로토타입을 적어줘야 다른 파일에서 이런 매개변수를 받는구나를 알게됨

5.7 Type Equivalence

- Two variables are type compatible *if either one can have its value assigned to the other*
  - Simple and rigid for predefined scalar type
  - In the case of structured type (array, record,..), the rules are more complex
- Two types are equivalence *if an operand of one type in an expression is substitute for one of the other, without coercion*
  - A restricted form of type compatibility (type compatibility without coercion)
  - There are two approaches to defining type equivalence
    - ⇔ Name Type Equivalence : two variables have equivalence types if they are defined either the same declaration or in declarations that use the same type name (Pascal)
      - ⇒ Easy to implement, but highly restrictive

```
type indextype is 1..100
count : Integer;
index : indextype;

count := index; /* ➔ Error !! */
```

    - ⇔ Structure Type Equivalence : two variables are compatible type if their types have identical structures (Ada)
      - ⇒ More flexible, but difficult to implement

```
subtype Small_type is Integer range 0..99;

/* Small_type is equivalent to the type Integer */
```

5.8 Scope

– The *scope* of a program variable is the range of statement in which the variable is visible
– A variable is *visible* in a statement if it can be referenced in that statement
– The scope rules of a language determine how references to names are associated with variables

⇔ static scoping, dynamic scoping

(1) Static Scoping

– the scope of variable can be statically determined (ALGOL60, ...)
– Suppose a reference is made to a variable X in procedure A. The correct declaration is found for the variable there, the search continues in the declaration of the procedure that declared procedure A (static parent) until a declaration for X is found

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

big -> sub2 -> sub1

dynamic scoping의 장점 - 매개 변수로 넘겨주지 않아도 됨

nested procedure definition (ex, Pascal)

- The local variables of a program unit are those that are declared in that unit
- The nonlocal variables of a program unit are those that are visible in the unit but not declared there
- global variables are a special category of nonlocal variables

parent를 쫓아가서 변수가 있는지 없는지 확인

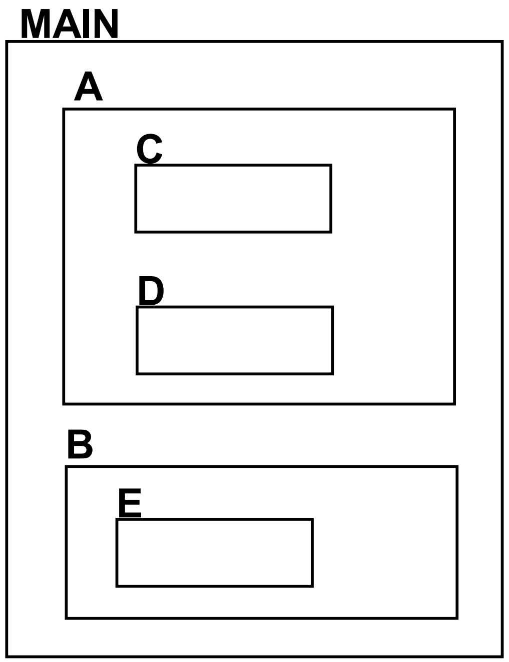
- Some languages allow new static scopes (called blocks) to be defined in the midst of executable code
  - allows a section of code to have its own (semidynamic) local variables
  - compound statement in C

```
if (a < b) { int temp ;
             temp = a ; a = b ; b = temp; }
```

- Ada
- JavaScript
- Fortran 2003+
- F#
- Python)

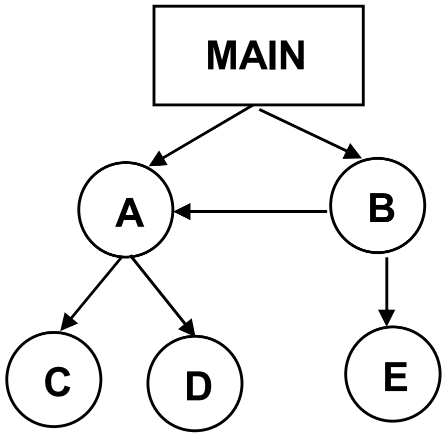
nested subprogram

- Problems in static scoping
  - an errorneous call to a procedure that should not have been callable will not be detected as an error by the compiler
  - too much data access
  - hard to modify the structures of program safely

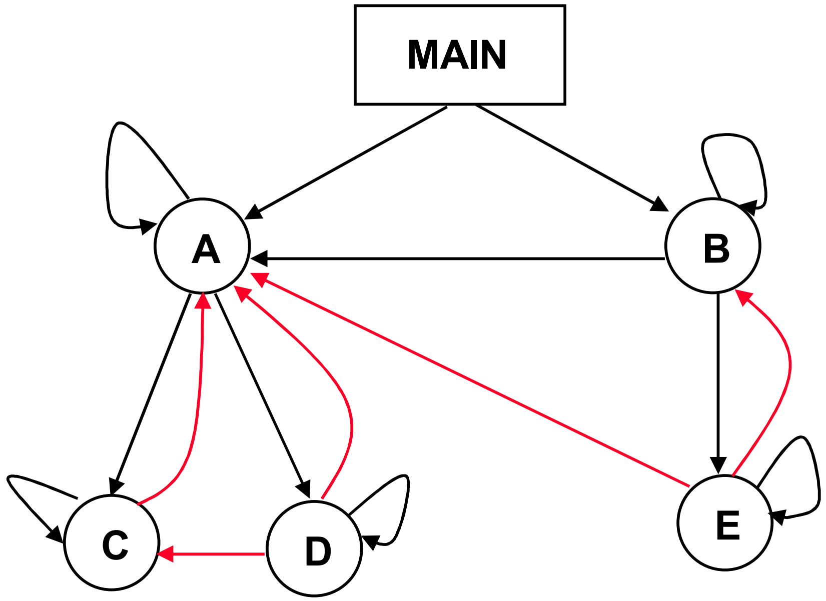


structure of program

b -> a는 되지만
a -> b는 안되는 이유
원패스 컴파일러를 쓰면



desirable call graph



potential call graph

(2) Dynamic Scope

- *Dynamic scoping* is based on the calling sequence of subprograms (dynamic parent), not on their spatial relationship(static parent) to each other. Thus, the scope is determined at run time
- a convenient method of communication (parameter passing) between program units
- APL, SNOBOL4, some dialect of LISP
- Problems in dynamic scoping
  - the local variables of the subprogram are all visible to any other executing subprogram, regardless of its textual proximity

    ⇔ results less reliable program
  - inability to statically type check referenced to nonlocals

5.9 Scope and Lifetime

- a variable declared in a Pascal procedure
  - scope : from its declaration to the end reserved word of the procedure (textual or spatial concept)
  - lifetime : the period of time beginning when the procedure is entered and ending when execution of the procedure reaches the end
- Example

  - scope of sum ?
  - lifetime of sum ?

```
procedure example ;
     procedure printerheader ;
        begin
          ....
        end ;
     procedure compute ;
        var sum : integer ;
         begin
            printerhead ;
         end
       begin
         compute;
       end ;
```

5.10 Referencing Environments

- The **referencing environment** of a statement is **the collection of all names that are visible in the statement**
  - static scoping languages

    ⇔ the variable declared in its local scope + the collection of all variables of its ancestor scopes
  - dynamic scoping languages : ?
- Example

  Point 1 : x and y of sub1, a and b of example
  Point 2 : x of sub3, a and b of example
  Point 3 : x of sub2, a and b of example
  Point 4 : a and b of example

- **active procedure** : its execution has begun but has not yet terminated

- **hidden variable** : ?

```
procedure example ;
     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 ;
```

static scoping에 의해 우선 여기서 x를 찾고
없으면 parent에 x가 있는지 확인

5.11 Named Constant

- a named constant is a variable that is bound to a value only at the time it is bound to storage
- readability can be improved
  - Ex) `const listlen = 10` (Pascal)

1. #define a 100 이랑
2. const int a 의 차이
1. k = a 를 하면 a를 다 찾아서 100으로 바꿈 즉 a 변수는 메모리에 안 올라옴

5.12 Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization
  - In FORTRAN,

```
REAL PI
INTEGER SUM
DATA SUM /0/, PI /3.14159/
```

  - In Ada,
    - ⇔ `SUM : INTEGER := 0 ;`
  - In ALGOL68
    - ⇔ initializing variable : `int first := 10 ;`
    - ⇔ initializing named constant : `int second = 10 ;`

<Homework>

1. Some programming languages are typeless. What are the obvious advantages and disadvantages of having no types in a language?

2. Consider the following JavaScript program: 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.

```
var x, y, z;
function sub1() {
  var a, y, z;
  function sub2() {
    var a, b, z;
      . . .
  }
  . . .
}
function sub3() {
var a, x, w;
. . .
}
```

3. Consider the following Python program. 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.

```
x = 1;
y = 3;
z = 5;
def sub1():
  a = 7;
  y = 9;
  z = 11;
. . .
def sub2():
  global x;
  a = 13;
  x = 15;
  w = 17;
  . . .
  def sub3():
    nonlocal a;
    a = 19;
    b = 21;
    z = 23;
    . . .
    . . .
```

indentation

4. Consider the following skeletal C program. Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last function called? Include with each visible variable the name of the function in which it was defined.
   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.

```
void fun1(void); /* prototype */
void fun2(void); /* prototype */
void fun3(void); /* prototype */
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;
  . . .
}
```

5. Consider the following program, written in JavaScript-like syntax. Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last subprogram activated? Include with each visible variable the name of the unit where it is declared.
   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.

```
// main program
var x, y, z;
function sub1() {
  var a, y, z;
  . . .
}
function sub2() {
  var a, b, z;
  . . .
}
function sub3() {
  var a, x, w;
  . . .
}
```