E : Non-terminal

Names, Bindings, Type Checking and Scopes

BBM 301 – Programming Languages

Today

- Introduction
- Names
- Variables
- The Concept of Binding
- Type Inference
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- Imperative programming languages are abstractions of the underlying von Neumann computer architecture.
- Architecture's two main components are:
  - **Memory** – stores both instructions and data
  - **Processor** – provides operations for modifying the contents of the memory

Abstraction

- Abstractions for memory are **variables**
- Sometimes abstraction is very close to characteristics of cells.
  - e.g. Integer – represented directly in one or more bytes of a memory
- In other cases, abstraction is far from the organization of memory.
  - e.g. Three dimensional array.
  - requires software mapping function to support the abstraction

Names

- Variables, subprograms, labels, user defined types, formal parameters all have names.
- Design issues for names:
  - What is the maximum length of a name?
  - Are names case sensitive or not?
  - Are special words reserved words or keywords?

Names (continued)

- **Length**
  - If too short, they cannot be connotative
  - Language examples:
    - Earliest languages : single character
    - FORTRAN 95: maximum of 31 characters
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31 characters
    - C#, Ada, and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Name Forms

- Names in most PL have the same form:
  - A letter followed by a string consisting of letters, digits, and underscore characters
  - In some, they use special characters before a variable's name
- Today "camel" notation is more popular for C-based languages (e.g. `myStack`)

- In early versions of Fortran – embedded spaces were ignored. *e.g.* following two names are equivalent

```
Sum Of Salaries
SumOfSalaries
```

Names (continued)

- **Special characters**
  - PHP: all variable names must begin with dollar signs
  - Perl: all variable names begin with special characters (`$`, `@`, `%`), which specify the variable's type
  - Ruby: variable names that begin with `@` are instance variables; those that begin with `@@` are class variables

Names (continued)

- **Case sensitivity**
  - In many languages (e.g. C-based languages) uppercase and lowercase letters in names are distinct
    - e.g. rose, ROSE, Rose
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)
  - Also bad for writability since programmer has to remember the correct cases

Names (continued)

- **Special words**
  - An aid to readability; used to delimit or separate statement clauses
    - A ***keyword*** is a word that is special only in certain contexts, e.g., in Fortran

      `Real VarName` (`Real` *is a data type followed with a name, therefore* `Real` *is a keyword)*

      `Real = 3.4` (*`Real` is a variable)*

```
INTEGER REAL
REAL INTEGER
```

This is allowed but not readable.

Names (continued)

- **Special words**
  – A ***reserved word*** is a special word that cannot be used as a user-defined name
    - Can't define `for` or `while` as function or variable names.
    - Good design choice
    - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Special Words

- **Predefined names:** have predefined meanings, but can be redefined by the user

- Between special words and user-defined names.

- For example, built-in data type names in Pascal, such as `INTEGER`, normal input/output subprogram names, such as `readln`, `writeln`, are predefined.

- In Ada, `Integer` and `Float` are predefined, and they can be redefined by any Ada program.

Variables

- A **variable** is an abstraction of a memory cell
- It is not just a name for a memory location
- A variable is characterized by a collection of attributes
  - Name
  - Address
  - Value
  - Type
  - Scope
  - Lifetime

Variable Attributes – Name

- Most variables are named (often referred as identifiers).
- Although nameless variables do exist (e.g. pointed variables).

Variable Attributes – Address

- **Address** - the memory address with which it is associated
- It is possible that the same name refer to different locations
- in different parts of a program:
  - A program can have two subprograms `sub1` and `sub2` each of defines a local variable that use the same name, e.g. `sum`
- in different times:
  - For a variable declared in a recursive procedure, in different steps of recursion it refers to different locations.
- Address of a variable is sometimes called l-value, because address is required when a variable appears on the left side of an assignment.

Aliases

- Multiple identifiers reference the same address – more than one variable are used to access the same memory location
- Such identifier names are called **aliases**.
- Aliases are created via pointers, reference variables, C and C++ unions
- Aliases are harmful to readability (program readers must remember all of them)

Variable Attributes – Type

- **Type** – determines
  - the range of values the variable can take, and
  - the set of operators that are defined for values of this type.
  - in the case of floating point, type also determines the precision

- For example `int` type in Java specifies a range of

  -2147483648 to 2147483647

Variable Attributes – Value

- The contents of the location with which the variable is associated
- e.g. *l_value* ← *r_value* (assignment operation)
  – The l-value of a variable is its address
  – The r-value of a variable is its value

X = 5

Abstract memory cell

- **Abstract memory cell** – the physical cell or collection of cells associated with a variable
  - Physical cells are 8 bits
  - This is too small for most program variables

The concept of Binding

- A **binding** is association between
  - entity ↔ attribute (such as between a variable and its type or value), or
  - operation ↔ symbol

- **Binding time** is the time at which a binding takes place.
  - important in the semantics of PLs

Possible Binding Times

- **Language design time** – bind operator symbols to operations
  - * is bound to the multiplication operation,
  - pi=3.14159 in most PL's.
- **Language implementation time**
  - bind floating point type to a representation
  - `int` in C is bound to a range of possible values
- **Compile time** -- bind a variable to a type in C or Java

Possible Binding Times (continued)

- **Link time**
  - A call to the library subprogram is bound to the subprogram code.
- **Load time**
  - A variable is bound to a specific memory location.
  - e.g. bind a C or C++ `static` variable to a memory cell
- **Runtime**
  - A variable is bound to a value through an assignment statement.
  - A local variable of a Pascal procedure is bound to a memory location.

Binding Times

- **Example:**
  - `count = count + 5`

- The type of `count` is bound at compile time
- The set of possible values of `count` is bound at compiler design time
- The meaning of the operator symbol `+` is bound at compile time, when the types of its operands have been determined
- The internal representation of the literal `5` is bound at compiler design time
- The value of `count` is bound at execution times with this statement

Static and Dynamic Binding

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

Type Bindings

- Before a variable can be referenced in a program, it must be bound to a data type.

- Two important aspects
  - How is a type specified?
  - When does the binding takes place?

- If static, the type may be specified by either an explicit or an implicit declaration

Static Type Binding – Explicit/Implicit Declarations

- explicit declaration (by statement)
  - A statement in a program that lists variable names and specifies that they are a particular type
- implicit declaration (by first appearance)
  - Means of associating variables with types through default conventions, rather than declaration statements. First appearance of a variable name in a program constitutes its implicit declaration
- Both creates static binding to types

Static Type Binding

- Most current PLs require explicit declarations of all variables
  - Exceptions: Perl, Javascript, ML
- Early languages (Fortran, BASIC) have implicit declarations
  - e.g. In Fortran, if not explicitly declared, an identifier starting with `I`,`J`,`K`,`L`,`M`,`N` are implicitly declared to integer, otherwise to real type
- Implicit declarations are not good for reliability and writability because misspelled identifier names cannot be detected by the compiler
  - e.g. In Fortran variables that are accidentally left undeclared are given default types, and leads to errors that are difficult to diagnose

Static Type Binding

- Some problems of implicit declarations can be avoided by requiring names for specific types to begin with a particular special characters

- Example: In Perl
  - `$apple` : scalar
  - `@apple` : array
  - `%apple` : hash

Dynamic Type Binding

- Type of a variable is not specified by a declaration statement, nor it can be determined by the spelling of its name (JavaScript, Python, Ruby, PHP, and C# (limited))
- Type is bound when it is assigned a value by an assignment statement.
- Advantage: Allows programming flexibility. example languages: Javascript and PHP
- e.g. In JavaScript
  - `list = [10.2 5.1 0.0]`
    - `list` is a single dimensioned array of length 3.
  - `list = 73`
    - `list` is a simple integer.

Dynamic Type Binding – Disadvantages

1. Less reliable: compiler cannot check and enforce types.

- Example: Suppose `I` and `X` are integer variables, and `Y` is a floating-point.
- The correct statement is

```
I := X
```

- But by a typing error

```
I := Y
```

- Is typed. In a dynamic type binding language, this error cannot be detected by the compiler.
  `I` is changed to float during execution.

- The value of `I` becomes erroneous.

Dynamic Type Binding – Disadvantages

2. Cost:

- Type checking must be done at run-time.
- Every variable must have a descriptor to maintain current type.
- The correct code for evaluating an expression must be determined during execution.
- Languages that use dynamic type bindings are usually implemented as interpreters (LISP is such a language).

Type Inference

- ML is a PL that supports both functional and imperative programming
- In ML, the types of most expressions can be determined without requiring the programmer to specify the types of the variables

- General syntax of ML

```
fun function_name(formal parameters) =
expression;
```

- The type of an expression and a variable *can be determined by the type of a constant* in the expression
- Examples

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

  `fun times10 (x) = 10*x;` (`times10` is integer)

[Note: fun is for function declaration.]

Type Inference

```
fun square (x) = x*x;
```

- Determines the type by the definition of * operator
- Default is `int`. if called with `square(2.75)` it would cause an error
- ML does not coerce real to int

- It could be rewritten as:

```
fun square (x: real) = x*x;
fun square (x):real = x*x;
fun square (x) = (x:real)*x;
fun square (x) = x*(x:real);
```

  - In ML, there is no overloading, so only one of the above can coexist

- Purely functional languages Miranda and Haskell uses Type Inference.

Storage Bindings and Lifetime

- **Allocation:** process of taking the memory cell to which a variable is bound from a pool of available memory

- **Deallocation:** process of placing the memory cell that has been unbound from a variable back into the pool of available memory

- **Lifetime of a variable:** Time during the variable is bound to a specific memory location

- According to their lifetimes, variables can be separated into four categories:
  - static,
  - stack-dynamic,
  - explicit heap-dynamic,
  - implicit dynamic.

Static Variables

- Static variables are bound to memory cells before execution begins, and remains bound to the same memory cells until execution terminates.
- **Applications:** globally accessible variables, to make some variables of subprograms to retain values between separate execution of the subprogram
- Such variables are history sensitive.
- **Advantage:** Efficiency. Direct addressing (no run-time overhead for allocation and deallocation).
- **Disadvantage:** Reduced flexibility (no recursion).
- If a PL has only static variables, it cannot support recursion.
- Examples:
  - All variables in FORTRAN I, II, and IV
  - Static variables in C, C++ and Java

Stack-Dynamic Variables

- Storage binding: when declaration statement is elaborated (in run-time).
- Type binding: static.

- The local variables get their type binding statically at compile time, but their storage binding takes place when that procedure is called. Storage is deallocated when the procedure returns.
- Local variables in C functions.

Stack-Dynamic Variables

- Advantages:
  - Dynamic storage allocation is needed for recursion. Each subprogram can have its own copy of the variables
  - Same memory cells can be used for different variables (efficiency)
- Disadvantages: Runtime overhead for allocation and deallocation

- In C and C++, local variables are, by default, stack-dynamic, but can be made static through static qualifier.

```
foo ()
{
static int x;
…
}
```

All attributes other than storage is statically bound to this type of variables

Explicit Heap-Dynamic Variables

- Nameless variables
- storage allocated/deallocated by explicit run-time instructions
- can be referenced only through pointer variables
- e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java
- types can be determined at run-time
- storage is allocated when created explicitly

Explicit Heap-Dynamic Variables

- Example:
  - In C++

```
int *intnode;                    // Create a pointer
intnode = new int;               // Create the heap-dynamic variable
….
delete intnode;                  // Deallocate the heap-dynamic variable
```

- Advantages:
  - Required for dynamic structures (e.g., linked lists, trees)
- Disadvantages:
  - Difficult to use correctly, costly to refer, allocate, deallocate.

Implicit Heap-Dynamic Variables

- Storage and type bindings are done when they are assigned values.
- Advantages:
  - Highest degree of flexibility (generic code)
- Disadvantages:
  - Runtime overhead for allocation/deallocation and maintaining all the attributes which can include array subscript types and ranges.
  - Loss of error detection by compiler
- Examples: All variables in APL; all strings and arrays in Perl, JavaScript, and PHP.

Variable Attributes – Scope

- **Scope** of a variable is the range of statements 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 variables declared outside the currently executing subprogram or block are associated with variables

Variable Attributes – Scope

- 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

Static Scope

- Scope of variables can be determined statically
  - by looking at the program
  - prior to execution
- First defined in ALGOL 60.

- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration

Static Scope

- **Search process**:
  - search declarations,
    - first locally,
    - then in increasingly larger enclosing scopes,
    - until one is found for the given name

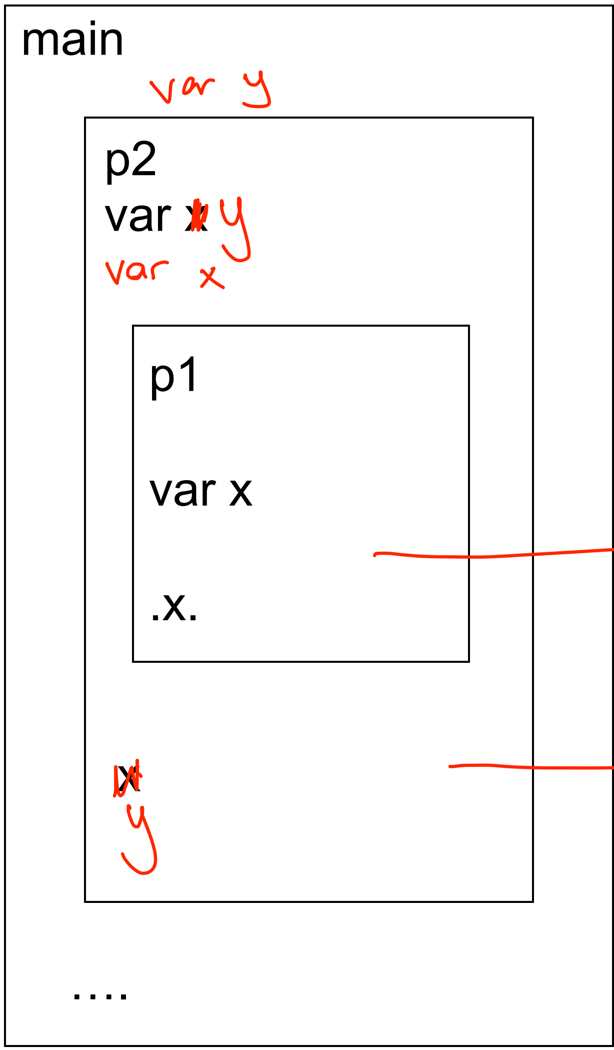
Static Scope

- In all static-scoped languages (except C), procedures are nested inside the main program.
- Some languages also allow nested subprograms, which create nested static scopes
  – Ada, JavaScript, Common LISP, Scheme, Fortran 2003+, F#, and Python - do
  – C based languages – do not
- In this case all procedures and the main unit create their scopes.

Static Scope

- Enclosing static scopes (to a specific scope) are called its static ancestors
- the nearest static ancestor is called a static parent

Static Scope



- Kendilerine en yakın değişkene erişirler
- Parent'ına bakar. (ilk)

- main is the static parent of p2 and p1.
- p2 is the static parent of p1

y'ye erişir, önce p2'deki

p2'deki x'i değil, kendi x'ine erişir.

p1'deki x'e erişemez.

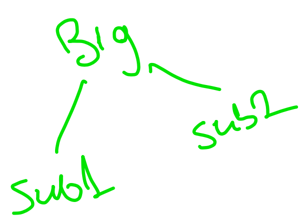
Static Scope

```
Procedure Big is
   x : integer
   procedure sub1 is
      begin    -    of
        sub1
      .... x ....
      end - of sub1
   procedure sub2 is
      x: integer;
      begin    -    of
        sub2
      .... X...
      end - of sub2
   begin - of big
   ...
   end - of big
```

kendi x'i visible
Big'deki hidden

- The reference to variable `x` in sub1 is to the `x` declared in procedure `Big`
- `x` in `Big` is hidden from `sub2` because there is another `x` in `sub2`

```
function big() {
        function sub1() {
        var x = 7;        1
        sub2();
        }
        function sub2() {
        var y = x;        2
        }
var x = 3;        3
sub1();
}
```



	Visible	Hidden
1	x in sub1	x in Big
2	y in sub2 x in Big	—
3	x in Big	

Static Scope

- In some languages that use static scoping, regardless of whether nested subprograms are allowed, some variable declarations can be hidden from some other code segments
- e.g. In C++

```
void sub1() {
int count;
...
while (...) {
int count;
...
}
...
}
```

- The reference to `count` in while loop is local
- `count` of `sub1()` is hidden from the code inside the while loop

Static Scope

- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
  - In Ada: `unit.name`
  - In C++: `class_name::name`

Blocks

- Some languages allow new static scopes to be defined without a name.
- It allows a section of code its own local variables whose scope is minimized.
- Such a section of code is called a block
- The variables are typically stack dynamic so they have their storage allocated when the section is entered and deallocated when the section is exited
- Blocks are first introduced in Algol 60

Blocks

- In Ada

```
...
declare TEMP: integer;
begin
TEMP := FIRST;
FIRST := SECOND;
SECOND := TEMP;
end;
...
```

Block

Blocks

C and C++ allow blocks.

```
int first, second;
...
first = 3; second = 5;
{ int temp;
     temp = first;
     first = second;
     second = temp;
}
...
```

`temp` is undefined here.

Blocks

- C++ allows variable definitions to appear anywhere in functions. The scope is from the definition statement to the end of the function

- In C, all data declarations (except the ones for blocks) must appear at the beginning of the function

- `for` statements in C++, Java and C# allow variable definitions in their initialization expression. The scope is restricted to the `for` construct

Dynamic Scope

- APL, SNOBOL4, early dialects of LISP use dynamic scoping.
- COMMON LISP and Perl also allows dynamic scope but also uses static scoping
- In dynamic scoping
  - scope is based on the calling sequence of subprograms
  - not on the spatial relationships
  - scope is determined at run-time.

Dynamic Scope

- Draw your Sequence
- Go back

Big → sub2 → sub1

```
Procedure Big is
    x : integer
    procedure sub1 is
        begin - of sub1
        .... x ....            (1)
        end - of sub1
    procedure sub2 is
        x: integer;
        begin - of sub2
        .... sub1()            (2)
        end - of sub2
    begin - of big
    ... sub2()
    end - of big
```

- When the search of a local declaration fails, the declarations of the dynamic parent is searched
- Dynamic parent is the calling procedure

Big -> sub2 -> sub1

	V	H
Dyn ①	x in sub2	x in Big
②	x in sub2	x in Big
St ①	x in Big	—
②	x in sub2	x in Big

- **Big calls sub2**
- **sub2 calls sub1**

- **Dynamic parent of sub1 is sub2, sub2 is Big**

	Visible	Hidden
1	x (sub2)	x (Big)
2	x (sub2)	x (Big)

Dynamic Scope

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

?

P1

dynamic scoping (called form big)

static scoping

dynamic scoping (called form **sub1**)

To determine the correct meaning of a variable, first look at the local declarations.

For **static** or **dynamic scoping**, the **local variables are the same**.

In dynamic scoping, look at the dynamic parent (calling unit).

In static scoping, look at the static parent (unit that declares, encloses).

Case1 (call of sub2 in big)
big->sub2->sub1 P1- x of sub2

Case2: (call of sub1 in big)
big -> sub1 P1- x of big

From H.A. Güvenir's notes

```
function big() {
        function sub1() {
        var x = 7;   (1)
        }
        function sub2() {
        var y = x;
        var z = 3;    (2)
        }
var x = 3;              (3)
sub1()
}
```

big-> sub1 -> sub2

First, big calls sub1, which calls sub2. Big → S1 → S2

Next, sub2 is called directly from big

big -> sub2

Static Scoping

Big
S1   S2

Point in code	Visible	Hidden
1	x (sub1)	x (big)
2	y,z (sub2), x(big)	
3	x (big)	

Dynamic Scoping

Point in code	Visible	Hidden
1	x (sub1)	x (big)
2	y,z (sub2), x(sub1)	x (big)
3	x (big)	

Dynamic Scoping

Point in code	Visible	Hidden
2	y,z (sub2), x(big)	
3	x (big)	

Referencing Environments

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

```
void sub1() {
int a, b;
. . . 1
} /* end of sub1 */
void sub2() {
int b, c;
.. . . 2
sub1();
} /* end of sub2 */
void main() {
int c, d;
. . . 3
sub2();
} /* end of main */
```

Point	*Referencing Environment*
1	a and b of sub1, c of sub2, d of main, (c of main and b of sub2 are hidden)
2	b and c of sub2, d of main, (c of main is hidden)
3	c and d of main

main() -> sub2() -> sub1()

	Visible	Hidden
1	a,b(sub1), c(sub2), d(main)	b (sub2), c(main)
2	b,c(sub2),d(main)	c(main)
3	c,d(main)	

Further Examples

Assume the following JavaScript program was interpreted using

A- **static-scoping rules**. What value of x is displayed in function sub1?
B- Under **dynamic-scoping rules**, what value of x is displayed in function sub1?

```
var x;
function sub1() {
document.write("x = " + x + "<br />");
}
function sub2() {
var x;
x = 10;
sub1();
}
x = 5;
sub2();
```

Static Scoping

in sub1 x(main) is visible
x = 5

Dynamic Scoping

main()-> sub2() -> sub1()

in sub1 x(sub2) is visible, x(main) is hidden

x = 10

Consider the following JavaScript program:

```
var x, y, z;
function sub1() {
var a, y, z;
function sub2() {
var a, b, z;
. . .          (1)
}
. . .  (2)
}
function sub3() {
var a, x, w;
. . .  (3)
}
```

	Visible	Hidden
1	a,b,z(sub2), y(sub1), x(main)	a,z(sub1), y,z(main)
2	a,y,z(sub1), x(main)	y,z(main)
3	a,x,w(sub3), y,z (main)	x(main)

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

Consider the following skeletal C program:

```
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;
. . . (2)
}
void fun2(void) {
int c, d, e;
. . .
}
void fun3(void) {
int d, e, f;
. . . (1)
}
```

Dynamic scoping

a) main->fun1->fun2->fun3

	Visible	Hidden
(1)	d,e,f(fun3), c(fun2), b(fun1) a(main)	d,e(fun2) c,d(fun1) b,c(main)

Dynamic scoping

c) main->fun2->fun3->fun1

	Visible	Hidden
(2)	b,c,d(fun1), e,f(fun3), a(main)	d(fun3), c,d,e(fun2), b,c(main)

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 sub3; sub3 calls sub1.~~

~~e. main calls sub1; sub1 calls sub3; sub3 calls sub2.~~

~~f. main calls sub3; sub3 calls sub2; sub2 calls sub1.~~

Blocks

```
void main() {
int x, y, z;
while ( . . . ) {
int a, b, c;
. . .
while ( . . . ) {
int d, e;
. . .
}
}
while ( . . . ) {
int f, g;
. . .
}
. . .
}
```

while1

while2

while3

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Scope of a variable is the range of statements in which the variable is visible and can be static, or dynamic.