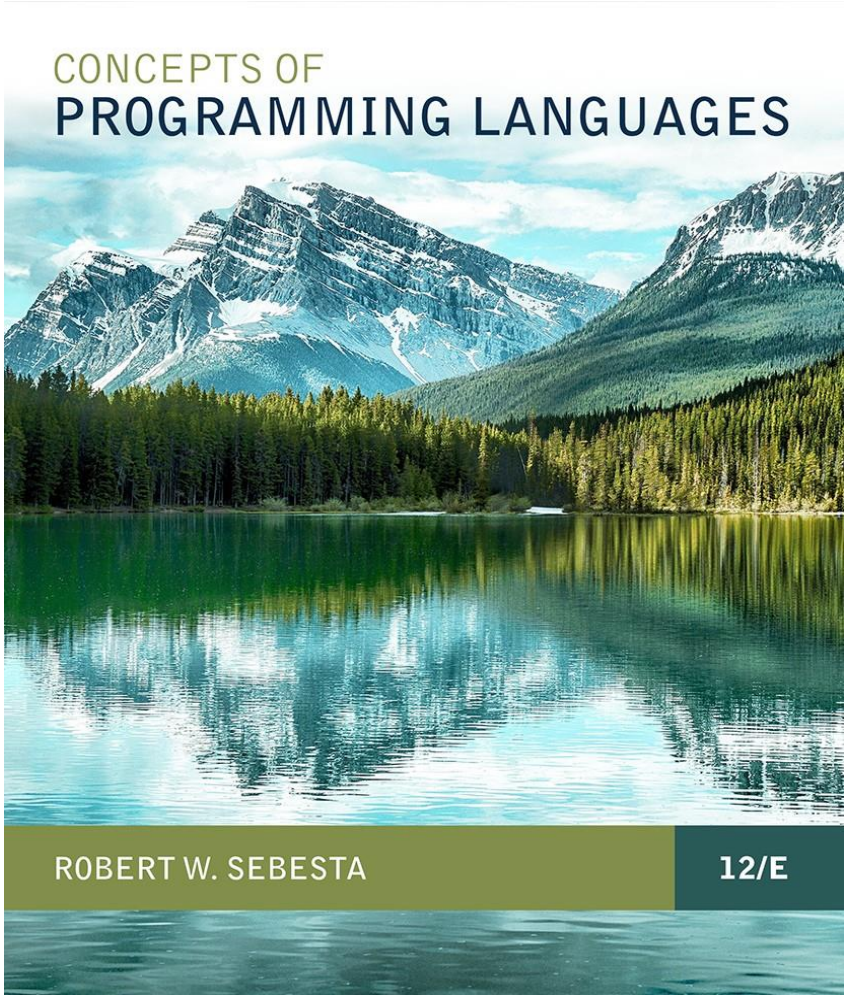
CAPÍTULO 5
NAMES, BINDINGS, TYPE CHECKING, AND SCOPES
NOMBRES, LIGACIONES, VERIFICACIÓN DE TIPOS Y ALCANCE

Estructura de los lenguajes

Dr. Christian von Lücken

Ajustado al libro “Concepts of Programming Languages”, Robert Sebesta, 12/E. Pearson. 2019. ISBN 0-321-49362-1



Tópicos

- Introducción
- Nombres
- Variables
- El concepto de asociación (Binding)
- Verificación de tipos (Type Checking)
- Tipeo Estricto (Strong Typing)
- Compatibilidad de tipos (Type Compatibility)
- Alcance y tiempo de vida (Scope and Lifetime)
- Entornos de referencia (Referencing Environments)
- Constantes con nombre (Named Constants)

Introducción

- Los lenguajes imperativos son abstracciones de la arquitectura de von Neumann
  - Memoria
  - Procesador
- Cuestiones de diseño de los tipos de datos de un lenguaje:
  - el ámbito (alcance) y el tiempo de vida de las variables,
  - el chequeo de tipos,
  - la inicialización, y
  - la compatibilidad de tipos

Nombres / identificadores

Un nombre es una cadena de caracteres utilizada para nombrar a una entidad de un programa:

- Variables
- Etiquetas
- Subprogramas
- Parámetros formales

Nombres: Cuestiones principales de diseño

1. ¿Cuál es la longitud máxima del nombre?
2. ¿Pueden utilizarse caracteres conectores con los nombres?
3. ¿Son los nombres sensibles a mayúscula/minúscula (case-sensitive)?
4. ¿Las palabras especiales son palabras reservadas o palabras claves (keywords)?

5.2.2 Forma de los Nombres: longitud

- Longitud
  - Si son demasiado cortos pueden no ser connotativos
  - Ejemplos:
    - FORTRAN I: máximo 6
    - COBOL: máximo 30
    - FORTRAN 90 y ANSI C: máximo 31
    - Ada y Java: no limitan,
    - C++: no limita, pero los implementadores usualmente imponen un límite

Nombres: uso de conectores

- Pascal, Modula-2 y FORTRAN 77 no permiten caracteres conectores en sus nombres.
- La mayoría de los lenguajes lo hacen.
- Notaciones camel case / pascal case

Nombres: case-sensitive

- Desventaja: facilidad de lectura/escritura
  - Nombres que lucen similares pero son distintos
  - Peor en C++ y Java debido a que nombres predefinidos tienen nombres con mezclas de mayúsculas y minúsculas (ej. parseInt != ParseInt)

- Los nombres en C, C++, y Java son case sensitive, en algunos lenguajes esto no es así

5.2.3 Palabras especiales

- Objetivos:
  - Mejorar la legibilidad
  - Separar o delimitar sentencias

- Un *keyword* tiene un significado especial sólo en ciertos contextos.

  Ej. en Fortran

  `Real VarName` (`Real` *es un tipo de dato seguido por un nombre, por lo que* `Real` *es un keyword)*

  `Real = 3.4` (*`Real` es una variable)*

- Una *reserved word* no puede ser utilizado como un nombre definido por el usuario

Variables

- Una variable es una abstracción de una celda de memoria o un conjunto de celdas
- Las variables pueden caracterizarse por una séxtupla de atributos:
  - Nombre
  - Dirección
  - Valor
  - Tipo
  - Tiempo de vida
  - Alcance (*Scope*)

Atributos de Variables

Nombres - no todas las variables lo tienen

Dirección (l-value)

- la dirección de memoria a la cual está asociada
- un mismo nombre puede tener diferentes direcciones en diferentes momentos durante la ejecución y diferentes lugares en un programa

  Alias

  – cuando dos nombres pueden utilizarse para acceder a la misma posición de memoria
  – se crean utilizando punteros, variables de referencia, uniones de C y C++
  – peligrosos para la legibilidad

Atributos de Variables (cont.)

- ***Tipo*** –el rango de valores de las variables y el conjunto de operaciones definidos para los valores del tipo, en el caso del punto flotante también determina la precisión
- ***Valor –*** el contenido de la posición con la cual la variable esta asociada

  Celda de memoria abstracta – la celda física o la colección de celdas asociadas con una variable

Atributos de Variables (cont.)

- El r-value de una variable es su valor, es lo que se requiere cuando la variable es utilizada en el lado derecho de una sentencia de asignación.

- El l-value de una variable es su dirección

- Para acceder a su r-value el l-value debe ser determinado primero.

5.4 El concepto de enlace (*binding*)

- Un *binding* es una asociación tal como la existente entre un atributo y una entidad, o entre una operación y un símbolo

- El *Binding time* es el momento en el cual ocurre el binding

Tiempos de binding posibles

- Tiempo de diseño de lenguaje – asociar símbolos de operadores a operaciones
- Tiempo de implementación de lenguaje – asocia un tipo floating point a una representación
- Tiempo de compilación – asocia una variable a un tipo en C o Java
- Tiempo de vinculación – asocia una llamada a un subprograma biblioteca al código del subprograma
- Tiempo de ejecución – asocia una variable no estatica local a una celda de memoria

Enlaces y sus tiempos de enlace

```
int count;
count = count + 5;
```

- *Conjunto de posibles tipos para count*: enlazado en el momento del diseño del lenguaje.
- *Tipo de count:* enlazado en el momento de la compilación.
- *Conjunto de posibles valores de count:* durante el diseño del compilador.
- *Valor de count:* al momento de la ejecución con esta sentencia.
- *Conjunto de posibles significados para el símbolo operador +:* enlazado en el momento de definición del lenguaje.
- *Significado del símbolo operador +:* enlazado en el momento de la compilación.
- *Representación interna del literal 5:* enlazado durante el diseño del compilador.

Asociación de atributos a vars. Binding Estático y Dinámico

- Un binding es *estático* si este ocurre antes del tiempo de ejecución y permanece sin cambio durante toda la ejecución del programa.

- Un binding es *dinámico* si ocurre durante la ejecución o puede cambiar durante la ejecución del programa.

Hardware bindings

- La asociación física de una variable a una celda de memoria en un entorno de memoria virtual es complejo: la página o el segmento del espacio de direcciones en el cual la celda reside puede moverse desde y hacia la memoria muchas veces durante la ejecución del programa.

- En cierto sentido, las variables son enlazadas y desenlazadas repetidamente.

- Estos enlaces se mantienen gracias al hardware de la computadora y son invisibles para el programa y el usuario.

Type Binding

- ¿Cómo se especifica un tipo?
- ¿Cuando ocurre el binding?

- Si es estático, el tipo podría ser especificado con una declaración explícita o implícita

Declaración de variables

Declaración Explícita/Implícita

- Una declaración explícita es una sentencia de programa utilizada para declarar los tipos de las variables
- Una declaración implícita es un mecanismo por defecto para especificar los tipos de las variables (la primera aparición de a variable en el programa)

Declaraciones implícitas

- FORTRAN, PL/I, BASIC, y Perl proveen declaraciones implícitas:
  - Ventaja: facilidad de escritura
  - Desventaja: legibilidad

Ejemplo declaración implícita: FORTRAN

Regla en FORTRAN

- Un identificador que aparece en un programa que no esta declarado de manera explícita está declarado de manera implícita de acuerdo a la siguiente convención:
  - si el identificador comienza con una de las letras I, J, K, L, M, N sea en mayúscula o minúsculas se declara de manera explícita como entero de otra forma se declara como de tipo real
- Implicit none

Ejemplo declaración implícita: PERL

Nombre que empieza con:

- $ es un escalar que puede almacenar una cadena o un valor numérico
- @ es un array
- % es una estructura hash
  - @apple es distinto a %apple
- Diferentes espacios de nombres, el lector siempre sabe a que tipo de dato corresponde la variable

Declaraciones implícitas

- Pueden producir un detrimento a la confiabilidad debido a que el compilador puede ser incapaz de detectar errores tipográficos y de programación
- Errores difíciles de diagnosticar
- Posibilidad de determinar opcional la declaración implícita/explícita
- Reglas para los nombres de tipos específicos

Declaraciones y definiciones

En C y C++ usualmente se hace una distinción:

- Las declaraciones especifican los tipos y otros atributos, pero no causan asignación de almacenamiento.
- Las definiciones especifican atributos y causan la asignación de almacenamiento.
- Para un nombre específico un programa en C puede tener cualquier número de declaraciones compatibles pero una única definición (extern)

Enlace dinámico de tipos
Dynamic Type Binding

- El tipo de una variable no se especifica con una declaración ni se determina por la escritura del nombre.

- El tipo se asocia al nombre mediante una sentencia de asignación.

- JavaScript y PHP

Ejemplo JavaScript:

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

Enlace dinámico de tipos
Dynamic Type Binding

- Ventaja: flexibilidad (unidades genéricas de programa, capaces de lidiar con datos de cualquier tipo numérico)

- Desventajas:
  - Alto costo (verificación dinámica de tipos e interpretación)
  - Difícil detección de errores de tipo por el compilador (problemas de confiabilidad)

JavaScript:

list = [10.2, 3.5]; (array unidimensional de tam. 2)

list = 47; (variable escalar)

Dynamic Type Binding

- Los lenguajes que utilizan dynamic type binding deben ser implementados utilizando interpretes puros en vez de compiladores
- Las computadoras no tienen instrucciones cuyos tipos de operandos no son conocidos en tiempo de compilación
- La interpretación pura típicamente lleva al menos 10 veces más de tiempo de ejecución que su contraparte en código máquina

Atributos de Variables (cont.)

- Inferencia de tipos (ML, Miranda, y Haskell)
  - En vez de sentencias de asignación, los tipos son determinados a partir del contexto de la referencias sin necesidad que el programador especifique los tipos de las variables

Asociación de almacenamiento y tiempo de vida

- Storage Bindings y Lifetime
  - Asignación – obtener una celda de un conjunto de celdas disponibles
  - Desasignación – liberar la celda que ha sido desasociada poniendola de nuevo en el conjunto de celdas disponibles
- El tiempo de vida de una variable es el tiempo durante el cual esta asociada a una celda particular de memoria

Categorías de las variables por sus tiempos de vida

- Static
- Stack-dynamic
- Explicit heap-dynamic
- Implicit heap-dynamic

Categorías de lenguajes con scope estático

- Asociación a las celdas de memoria antes de iniciar la ejecución y se mantiene ligado a la misma celda de memoria durante toda la ejecución, ejemplo, todas las variables de FORTRAN 77, las variables estáticas de C
  - Ventajas: eficiencia (direccionamiento directo), los subprogramas pueden ser "history sensitive" es decir mantener valores entre ejecuciones separadas
  - Desventaja: flexibilidad (sin recursión)

Categorías de las variables por sus tiempos de vida

- Stack-dynamic – las asociaciones de almacenamiento se establecen cuando se elaboran las sentencias de declaración
- Si es un escalar, todos los atributos con excepción de sus direcciones son asociadas de manera estática
  - las variables locales en los subprogramas en C y los métodos Java
- Ventajas: permite la recursión; mantiene el almacenamiento
- Desventajas:
  - Sobrecosto de asignación y desasignación
  - Los subprogramas no pueden ser "history sensitive"
  - Referencias ineficientes (direccionamiento indirecto)

Categorías de las variables por sus tiempos de vida

- *Explicit heap-dynamic* – asignados y desasignados por directivas explícitas, especificadas por el programador, las cuales tienen efecto durante la ejecución
- Referenciado solo a través de punteros o referencias, ej. objetos dinámicos en C++ (via new y delete), todos los objetos en Java
- Ventaja: manejo dinámico del almacenamiento
- Desventaja: ineficiente y poco confiable

Categorías de las variables por sus tiempos de vida

- *Implicit heap-dynamic* – asignación y desasignación causada por sentencias de asignación
  - Todas las variables en APL; todas las cadenas y arrays en Perl y JavaScript
- Ventaja: flexibilidad
- Desventajas:
  - Ineficiente, debido a que todos los atributos son dinámicos
  - Perdida de detección de errores

Verificación de tipos

- Generalización del concepto de operandos y operadores para incluir subprogramas y asignaciones
- La verificación de tipos es la actividad de asegurar que los operandos son de tipos compatibles
- Un tipo compatible es uno que es legal para el operador o se permite que sea convertido de manera implícita bajo reglas del lenguaje por código generado por el compilador a un tipo válido
  - La conversión automática se llama *coercion*.
- Un error de tipo es una aplicación de un operador a un operando de tipo inapropiado

Verificación de tipos (cont.)

- Si todas las asociaciones son estáticas, prácticamente todas la verificaciones de tipo pueden ser estáticas
- Si las asociaciones de tipo son dinámicas, las verificaciones de tipo deben ser dinámicas
- Un lenguaje es fuertemente tipado si los errores de tipo siempre son detectados

Tipeo fuerte - *Strong Typing*

- Ventaja: permite la detección de usos incorrectos de las variables que resultan en errores de tipo
- Ejemplos de lenguajes:
  - FORTRAN 77 no es: parámetros, `EQUIVALENCE`
  - Pascal no es: *variant records*
  - C y C++ no son: la verificación del tipo de los parámetros puede evitarse, las uniones no son verificadas
  - Ada es (`UNCHECKED CONVERSION`) (Java is similar)

Strong Typing (cont.)

- Las reglas de coercion afectan el tipeo estricto y pueden afectarlo de manera considerable (C++ vs. Ada)
- A pesar que Java tiene sólo la mitad de las reglas de coerción de C++, su strong typing es aún menos efectivo que para Ada

Compatibilidad de nombre de tipo

- *Compatibilidad de nombre de tipo* significa de dos variables tienen tipos compatibles si están en la misma declaración o en declaraciones que utilizan el mismo nombre de tipo
- Fácil de implementar pero altamente restrictivo:
  - Subrangos de tipos enteros no son compatibles con tipos enteros
  - Los parámetros formales deben ser del mismo tipo que el correspondiente parámetro actual

Compatibilidad de estructura de tipo

- *Structure type compatibility* significa que dos variables tienen tipos compatibles si sus tipos tienen estructuras identicas
- Más flexible, pero difícil de implementar

Compatibilidad de tipos (cont.)

- Considere el problema de dos tipos estructurados:
  - Son los dos tipos record compatibles si es que tienen la misma estructura pero diferente nombre de campos?
  - Son dos tipos array compatibles si son iguales excepto que los subindíces son diferentes?

    (ej. [1..10] y [0..9])
  - Son dos tipos enumerados compatibles si sus componentes se escriben diferentes?
  - Con compatibilidad estructural de tipos, no se puede diferenciar entre tipos de la misma estructura (ej. Unidades diferentes de velocidad, ambos float)

Continuará….

- Louden (cap. 7 parte 1) entornos de ejecución.

5.8 Scope

Alcance

Atributos de variables: Scope

- El alcance de las variables es el rango de las sentencias sobre la cual es visible
- Las variables *no locales* de una unidad de programa son aquellas que son visibles pero que no están declaradas en la unidad
- Las reglas de alcance de una variable determina cómo las referencias a los nombres están asociadas con variables

Static Scope
Alcance Estático

- Basado en el texto del programa
- Para conecta una referencia de nombre a una variable, el compilador debe encontrar la declaración
- El proceso de búsqueda: buscar las declaraciones, primero localmente, luego en alcances cada vez mas grande hasta que se encuentre uno para el nombre dado
- Los alcances estáticos que rodean un alcance específico son llamados ancestros estáticos y el más cercano de estos es su padre estático

Scope (cont.)

- Las variables pueden estar escondidas de una unidad teniendo una variable más cercana con el mismo nombre
- C++ y Ada permiten acceder e estas varibales escondidas ("hidden“)
  - En Ada: **`unit.name`**
  - En C++: **`class_name::name`**

Bloques

- Un método para crear alcances estáticos dentro de unidades de programa – (viene de ALGOL 60)
- Ejemplos:

C y C++:
```
for (...) {
    int index;
    ...
}
```

Ada:
```
declare LCL : FLOAT;
begin
    ...
end
```

Evaluación de alcance estático (Static Scoping)

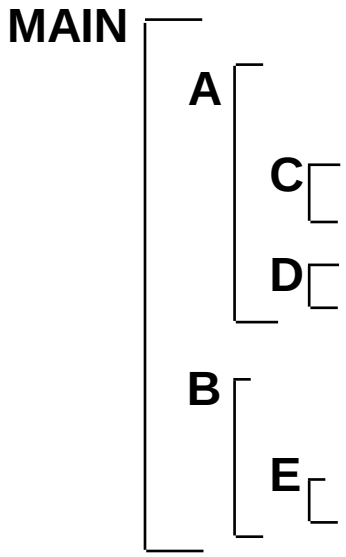
- El alcance estático provee un mecanismo para acceder a no-locales que trabaja bien en muchas situaciones.

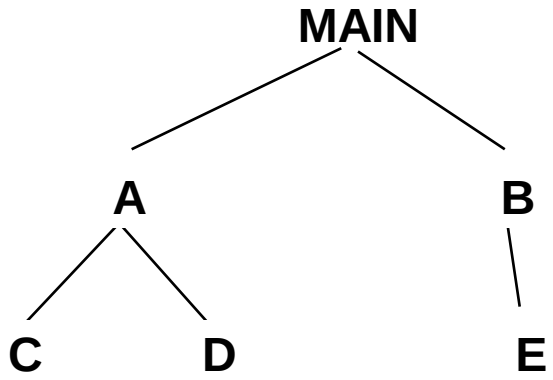
Evaluación de alcance estático (Static Scoping)

- Suponga que MAIN llama A y B

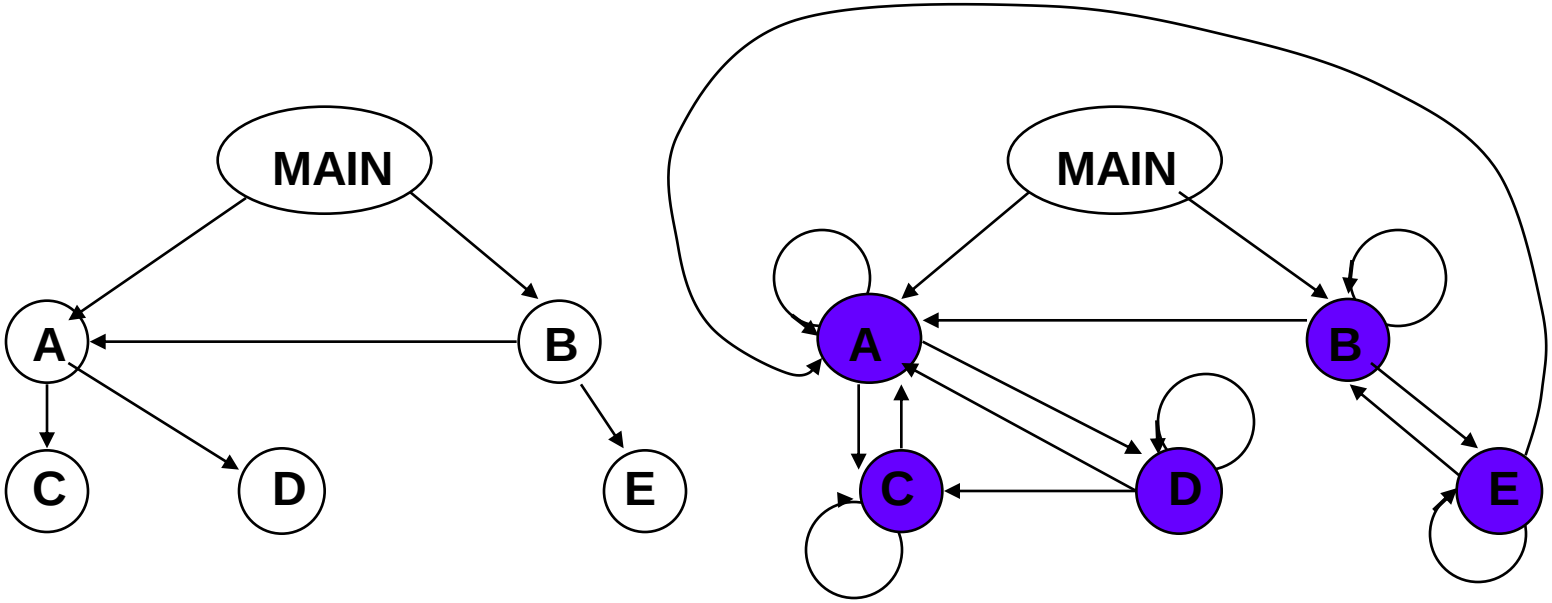
  A llama C y D

  B llama A y E





Ejemplo de Static Scope



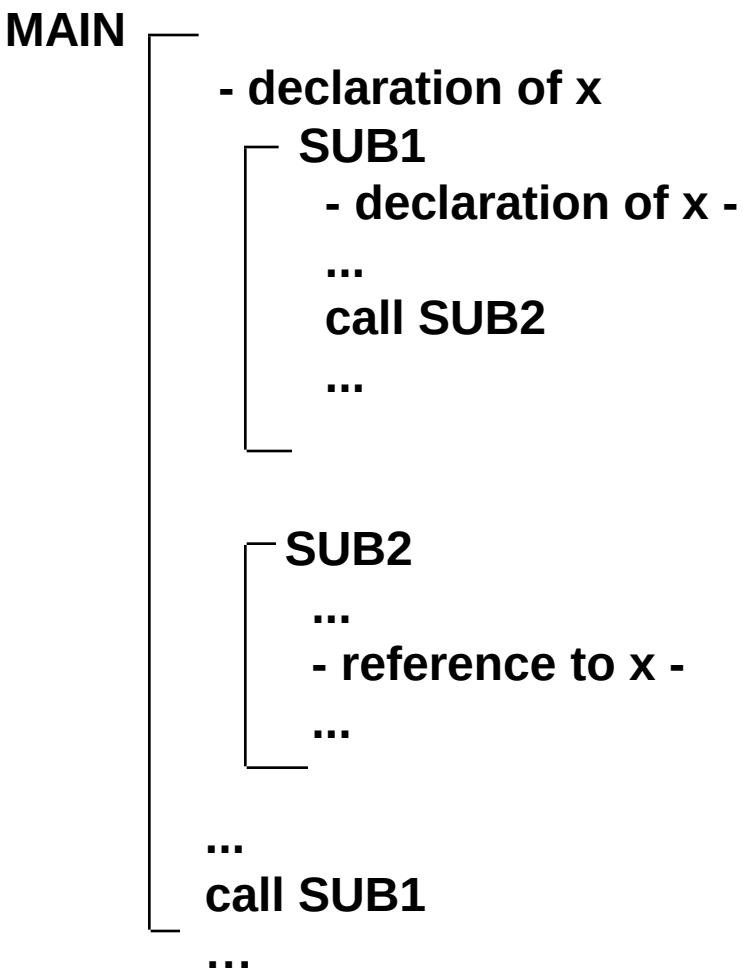
Static Scope (cont.)

- Suponga que la especificación cambio tal que D debe acceder a algún dato en B
- Soluciones:
  - Poner D en B (pero entonces C y no podra llamarlo y D no podrá acceder a las variables de A)
  - Mover el dato desde B tal que D necesita a MAIN (pero todos los procedimientos pueden acceder a él)
- El mismo problema para el acceso a los procedimientos
- En general, el alcance estático conduce a muchas variables globales

Dynamic Scope

- *Basado en secuencias de llamadas a unidades de programa, no su posición textual (temporal versus espacial)*
- Las referencias a las variables están conectadas a declaraciones buscando hacia atrás por una *cadena de llamadas* a subprograms que lleva la ejecución a ese punto

Ejemplo de Scope



MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Ejemplo de Scope

- Static scoping
  - La referencia a x es al x de MAIN
- Dynamic scoping
  - La referencia a x es a la x de SUB1
- Evaluación del alcance Dinámico:
  - Ventaja: conveniencia
  - Desventaja: pobre legibilidad

Scope y Lifetime

- El Scope y el tiempo de vida están generalmente muy relacionadas, pero existen diferentes conceptos
- Considere una variable `static` en una función de C o C++

Entornos de Referencia

- El entorno de referencia de una sentencia es la colección de todos los nombres que son visibles en la sentencia
- En un lenguaje de alcance estático, son las variables locales más todas las variables visibles en todos los alcances que lo engloban
- Un subprograma esta activo si su ejecución a comenzado pero aún no ha terminado
- En un lenguaje de alcance dinámico, el entorno de referencia son las variables locales más todas las variables visibles en todos los subprogramas activos

Constantes con nombre (Named Constants)

- Una constante con nombre es una variable que está ligada a un valor solo cuando esté esta ligado al almacenamiento
- Ventajas: legibilidad y modificabilidad
- Utilizado para parametrizar programas
- La asociación de valores a nombres de constantes puede ser estático (*manifest constants*) o dinámico

Inicialización de Variables

- La asociación de una variable a un valor al momento en que se asocial a un almacenamiento se llama inicialización
- La inicialización se realiza usualmente en la sentencia de declaración
- La inicialización se realiza usualmente en la sentencia de declaración

```
int sum = 0;
```

Resumen

- La diferenciación entre mayúsculas y minúsculas y las relaciones entre nombres y palabras especiales representan cuestiones de diseño para los nombres
- Las variables se caracterizan por una sextupla: nombre, dirección, valor, tipo, tiempo de vida, alcance
- Binding es la asociación de atributos a entidades de programa
- Las variables escalares se caracterizan como: estáticos, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing significa detectar todos los errores de tipo

Figure 5.1

The structure of a program

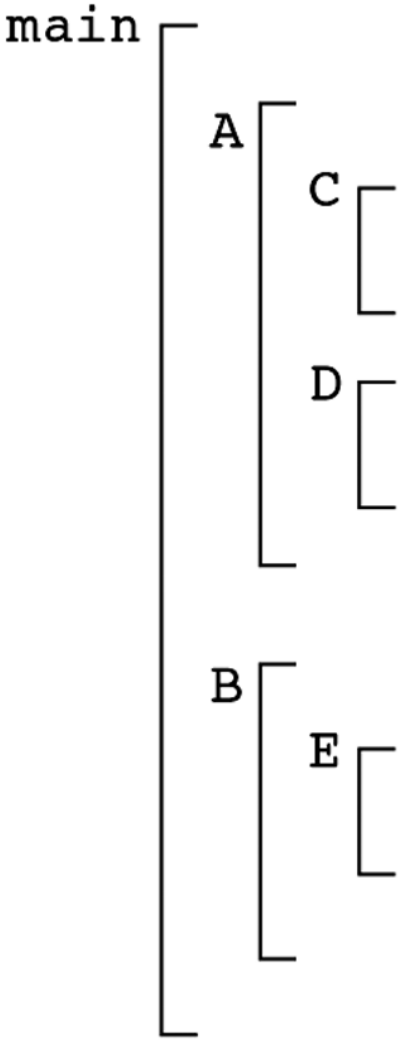


Figure 5.2

The tree structure of the program in Figure 5.1

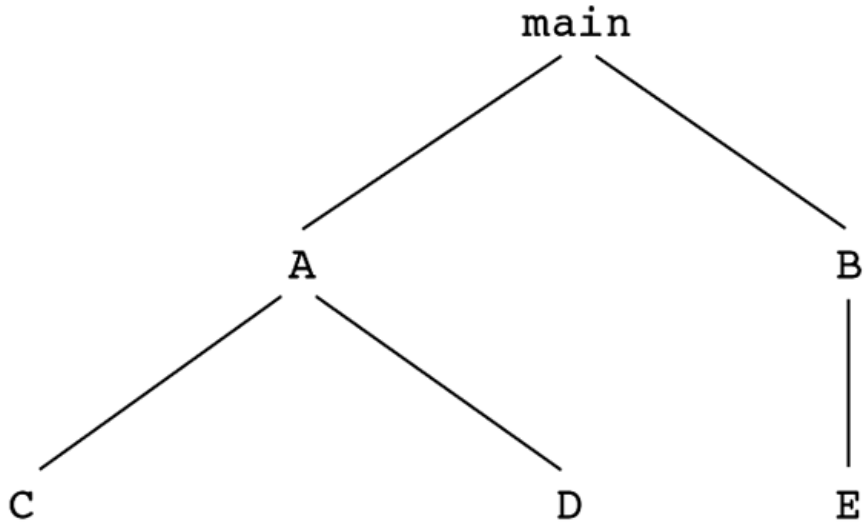


Figure 5.3

The potential call graph of the program in Figure 5.1

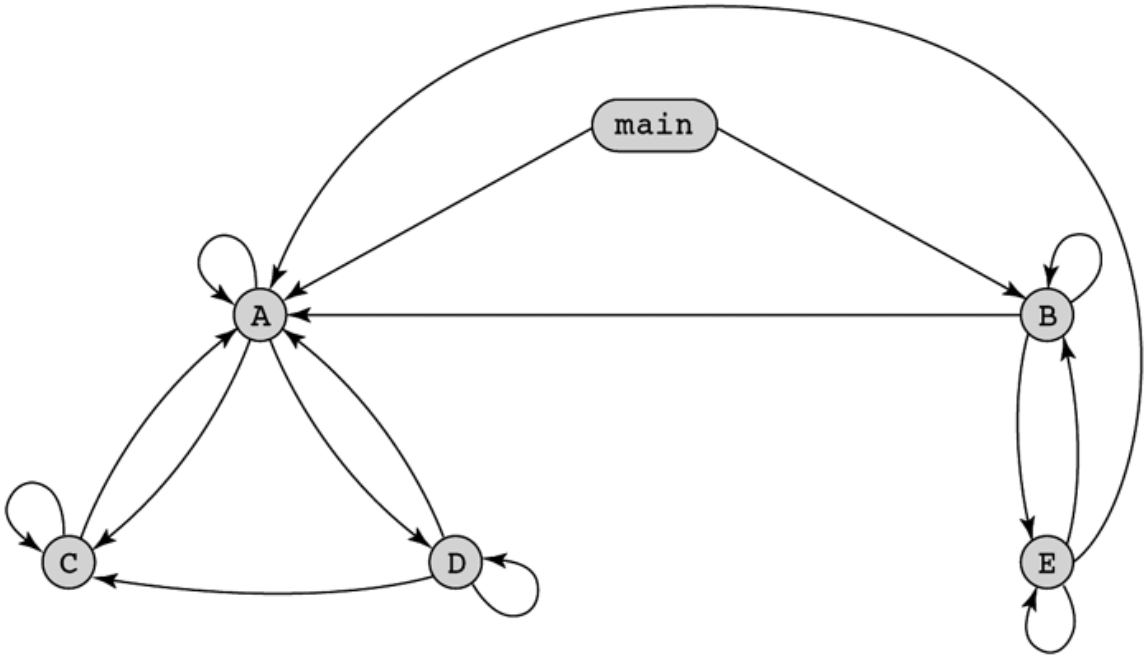


Figure 5.4

The graph of the desirable calls in the program in Figure 5.1