Names, Scopes, and Bindings II

Hwansoo Han
Scope Rules

- A scope is textual region where bindings are active
  - A program section of maximal size
  - Bindings become active at the entry

- No bindings change in the middle, or
- No re-declarations are permitted in the middle at least
Scope Rules in Subroutines

- A subroutine opens a new scope on its entry
  - Create bindings for new local variables,
  - Deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)

- On subroutine exit
  - Destroy bindings for local variables
  - Reactivate bindings for global variables that were deactivated

- Term “elaboration” first used in Algol 68 and Ada
  - Process of creating bindings when entering a scope
  - Allocate space at stack, assign initial values, ...
Static Scoping

- Static scoping = lexical scoping

- A scope is defined in terms of the physical (lexical) structure of the program
  - Scopes can be determined at compile time
  - All bindings for IDs can be resolved by examining program
  - Typically, most recent, active binding made at compile time
    - Current binding is the declaration of most closely surrounding block

- Most compiled languages employ static scope rules
  - C, Pascal, ...
Static Scoping – nested blocks

- A classical example of static scope rules
  - The most closely nested rule
  - Used in block structured languages (Algol 60 and Pascal)

- Most closely nested rule
  - An identifier is known in local scope (where it is declared)
  - Also known in each enclosing scope from the closest, unless re-declared in an enclosed scope – “hidden”

- Resolving a reference to an identifier
  - Examine the local scope and statically enclosing scopes until a binding is found
Static Scoping - modules

- Object-oriented languages
  - More sophisticated, but static scope rules among classes

- Binding is not destroyed (different from subroutine)
  - Modules in OOL (Modula, Ada, etc.) give you closed scopes without the limited lifetime
  - Bindings to variables declared in a module are inactive outside the module, not destroyed

- Similar effect can be achieved in many languages
  - static (C term) variables
  - own (Algol term)
Static Links

- Access to non-local variables through **Static Links**
  - Each frame contains a static link to point to the *parent* frame
  - Parent frame means the most recent invocation of the lexically surrounding subroutine

- You access a variable in a scope $k$ levels outside
  - by following $k$ static links and then using the known offset within that frame
Static Chains

A() {
    B() {
        C() { … }
        D() { call C(); }
        call D();
    }
    E() {
        call B();
    }
    call E();
}

Scopes

Static link
Dynamic Scoping

- With **dynamic scope rules**, bindings depend on the current state of program execution
  - Cannot always be resolved by examining the program because they are dependent on calling sequences
  - To resolve a reference, we use the most recent, active binding made at run time
Dynamic Scoping

- Dynamic scoping often used in interpreted languages
  - APL, Snobol, Tex, early dialects of LIPS, Perl

- No type checking at compile time
  - Type determination is not always possible at compile time, when dynamic scope rules are in effect
Example: Static vs. Dynamic

n : integer; ← global

procedure main {

    procedure first {
    n := 1;
    }
    procedure second {
    n : integer; ← local
        first();
    }

    n := 2;
    second();
    write(n);

    Output

    Static scoping: 1
    Dynamic scoping: 2
How dynamic scoping works for the prev. example

- Create a binding for `global n` when we enter `main()`
- Create another binding for `local n` when we enter `second()`
  - This is the most recent, active binding when `first()` is executed
- In `first()`, modify `n local` to `second()`, not `global n`
- In `main()`, `write()` uses `global n`,
  - `n local` to `second()` is no longer active in `main()`
Aliases

- **Aliasing**
  - Same address but multiple names
  - *Variant Record* in Pascal and *Union* in C
  - *Common* and *Equivalence* in FORTRAN
  - Parameter passing by reference to a subroutine

- **What are aliases good for?**
  - Space saving
  - Multiple representations
  - Pointer-based data structures
Overloading

- Overloading
  - The same name performs different things
    - functions, operators, enumeration constants, etc.

- Overloading happens in almost all languages
  - “Integer +” vs. “real +”
  - Enumeration constants in Ada

```
type autumn is (sep, oct, nov);
type base is (dec, bin, oct, hex);
mo : autumn;
pb : base;
mo := nov;
pb := oct;
print(oct);  -- error!
```

```markdown
-- cannot decide type
```
Overloaded Functions

- Two different things with the same name (in C++)

```c
int norm (int a){return a>0 ? a : -a;}
complex norm (complex c ) { ... }

int i;
complex c;

norm(i);    // integer norm function
norm(c);    // complex norm function
```
Polymorphic Functions

- One thing that works in more than one way
  - Polymorphism ≠ Overloading – they are slightly different

- Parametric polymorphism
  - Code takes types as parameters explicitly or implicitly
  - Generic in Java (or templates in C++) explicitly takes types
  - Lisp, ML, Scheme, Haskell implicitly take types

- Subtype polymorphism
  - Code takes subtypes as well as original type in OOL
  - Inheritance in OOL provides this with virtual methods
  - Involves dynamic binding of overriding function
    - Overriding – a name in base class is redefined in a sub class, with the exactly same number/types of parameters
Parametric Polymorphism

- Explicit parametric polymorphism

  ```java
  // Java generic with interface java.lang.Comparable<T>
  public static <T extends Comparable<T>> T max(T a, T b) {
    if (a.compareTo(b) >= 0) return a;
    else return b;
  }
  max(1, 5); // T is Integer (autoboxing int)
  max(1.4, 5.6); // T is Double (autoboxing double)
  ```

- Implicit parametric polymorphism
  - Interpreted languages determine operators at run time

  ```scheme
  (define min (lambda (a b) (if (< a b) a b))) // Scheme
  min a b = if a < b then a else b // Haskell
  ```
Generic Functions

- A syntactic template that can be instantiated in more than one way at compile time
  - Via macro processors in C/C++
  - Built-in in C++ and Ada

```cpp
// C++ template
template<class X> X max(X a, X b) {
  return a>b ? a : b;
}

void g(int a, int b, char c, char d) {
  int m1 = max(a,b);
  char m2 = max(c,d);
}
```
Subtype Polymorphism

// Java subtype with virtual method
public class Car {
    public void brake() {}
    public void stop() { brake(); }

    public static void main(String args[]) {
        ManualCar m = new ManualCar();
        AutoCar a = new AutoCar();
        m.stop();
        a.stop();
    }
}

class ManualCar extends Car {
    public void clutch() { ... }
    public void brake() { clutch(); ... }
}

class AutoCar extends Car {
    public void brake() { ... }
}
Coercion

- Coercion allows implicit type conversion
  - Compiler automatically converts a value of one type into a value of another type, when the context requires it
  - Could cause performance problem

```c
double min(double x, double y) { ... }

double f, g, h;
int i, j, k;

f = min(g, h);
i = min(j, k);
```

Type conversion operations are inserted for parameters and return value.
Accessing Variables in Dynamic Scope

- Two methods
  - Stack
  - Central table

- Stack (*association list*) of all active variables
  - To find a variable, hunt down from top of stack
  - Equivalent to searching the activation records on the dynamic chain
  - Slow accesses but fast calls
Accessing Variables in Dynamic Scope

- Central table with one slot for every variable name
  - If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time
  - Otherwise, you'll need a hash function to do lookup
  - Every subroutine changes the table entries for its locals at entry/exit
  - Slow calls but fast accesses
Language Features

- Language features can be surprisingly subtle

- A language that is easy to compile often leads to
  - A language that is easy to understand
  - More good compilers on more machines
    (compare Pascal and Ada!)
  - Better (faster) code
  - Fewer compiler bugs
  - Smaller, cheaper, faster compilers
  - Better diagnostics
Summary

- Scope rules
  - Static vs. dynamic scope

- Meaning of names within a scope changes
  - Aliases
  - Overloading
  - Polymorphism

- Type conversion
  - Coercion – implicit type conversion