

# Scope

## Programming Languages CS 214



# Scope

The portion of a program in which an identifier is bound to a particular meaning is called the \_\_\_\_\_.

Example:

If scope is \_\_\_\_\_:

*i* refers to the integer declared in *p1*.

*Ada* and most languages use \_\_\_\_\_ scope.

```
procedure p1 is
  i: integer := 1;
  procedure p2 is
  begin
    put_line(i);
  end p2;
  procedure p3 is
    i: float := 3.0;
  begin
    p2;
  end p3;
begin
  p2;
  p3;
end p1;
```

If scope is \_\_\_\_\_:

When called by *p1*, *i* refers to the integer declared in *p1*.

When called by *p3*, *i* refers to the float declared in *p3*.



# Dynamic Scope

Lisp uses *dynamic scope*, in which an identifier's binding is

---

The meaning of a symbol depends on the *run-time context* in which it is accessed.

Example: If we write:

and then call *p2* by itself:

but if we call *p2* in different contexts (*p1* vs. *p3*):

```
(setq i 0)
```

```
(defun p2 ()  
  (princ i)  
  (terpri))
```

```
(defun p3 ()  
  (let ((i 3.0))  
    (p2)))
```

```
(defun p1 ()  
  (setq i 1)  
  (p2)  
  (p3))
```

(p2)	;; call p2
_____	;; by itself

(p1)	;; call p2
_____	;; within p1
_____	;; within p3

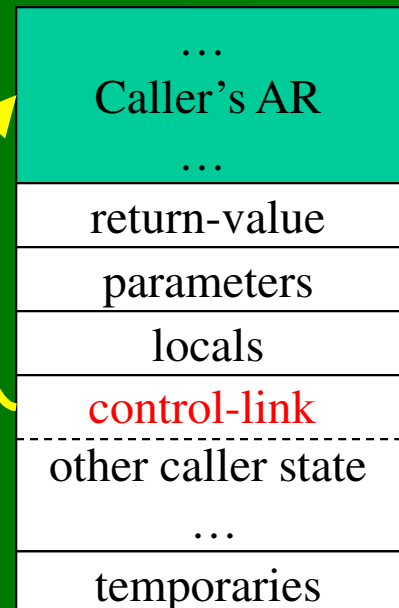


# Dynamic Scope: What is happening?

Each activation record contains a \_\_\_\_\_,  
pointing to the activation record of its caller:

When an identifier *id* is accessed, the system  
follows this algorithm:

- a. *temp* = *stack pointer*;
  - b. do (in the AR pointed to by *temp*):
    - 1) *found* = search *temp*'s parameters/locals for *id*;
    - 2) if not *found*: *temp* = *temp*->*control\_link*;
- while *temp* != NULL && not *found*.



On a non-local access, dynamic scope \_\_\_\_\_  
“down” through the run-time stack until a definition of *id* is found.



# Dynamic Scope: Trace

The initial *setq* defines *i* globally...

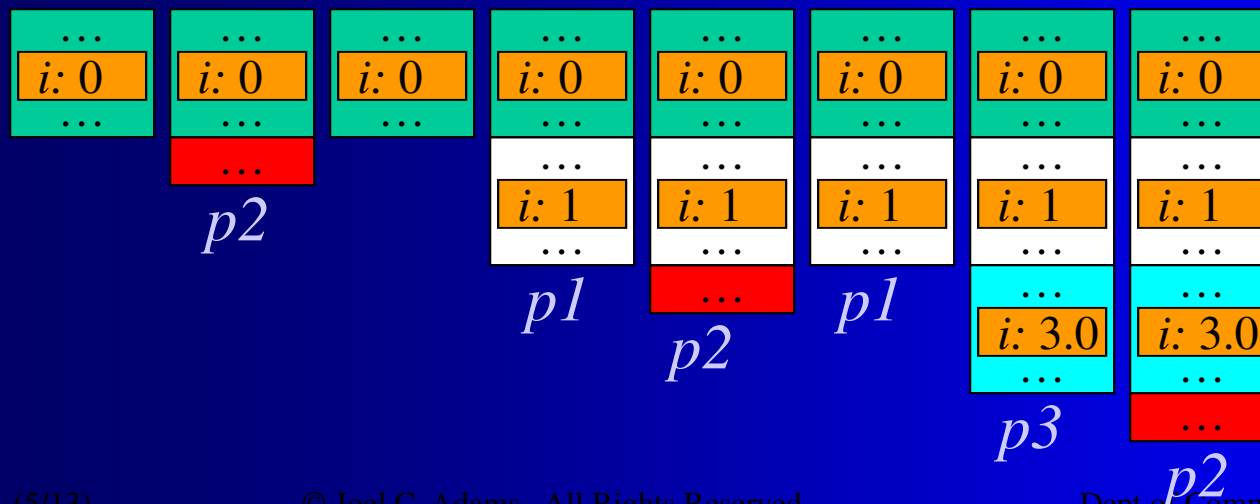
We call *p2*, which searches down, finds *i* == 0  
*p2* terminates...

We call *p1* that defines *i* as the integer 1...

*p1* calls *p2*, which searches down, finds *i* == 1  
*p2* terminates...

*p1* calls *p3*, which defines *i* as the real 3.0

*p3* calls *p2*, which searches down, finds *i* == 3.0



```
(setq i 0)
```

```
(defun p2 ()  
  (princ i)  
  (terpri))
```

```
(defun p3 ()  
  (let ((i 3.0))  
    (p2)))
```

```
(defun p1 ()  
  (setq i 1)  
  (p2)  
  (p3))
```

```
(p2)
```

```
(p1)
```



# Static Scope

\_\_\_\_\_ languages bind identifiers at \_\_\_\_\_.

The basic static scope rule:

The scope of a binding begins at  
an identifier's \_\_\_\_\_; and  
ends at \_\_\_\_\_.

So *p2* always accesses integer *i*,  
because *p2* lies within its scope,  
and not within the scope of float *i*.

A *scope context* is the construct  
used by a particular language to  
delimit a binding's scope.

```
procedure p1 is
  i: integer := 1;
  procedure p2 is
    begin
      put_line(i);
    end p2;
  procedure p3 is
    i: float := 3.0;
    begin
      p2;
    end p3;
  begin
    p2;
    p3;
  end p1;
```



# Static Scope: Holes

A nested declaration of the same identifier creates a \_\_\_\_\_ in the scope of the outer binding:

- Within *p3*, references to *i* access float *i*, not integer *i*.
- Ada provides a workaround:  
within *p3*, \_\_\_\_\_ will access integer *i*.

Holes are not limited to Ada...

```
char i = 'J';
{
  int i = 1;    // char i now
  if ( ... )   // hidden
  {
    double i = 3.0;
  } // char, int i both hidden
}
```

```
procedure p1 is
  i: integer := 1;
  procedure p2 is
  begin
    put_line(i);
  end p2;
  procedure p3 is
    i: float := 3.0;
  begin
    p2;
  end p3;
begin
  p2;
  p3;
end p1;
```

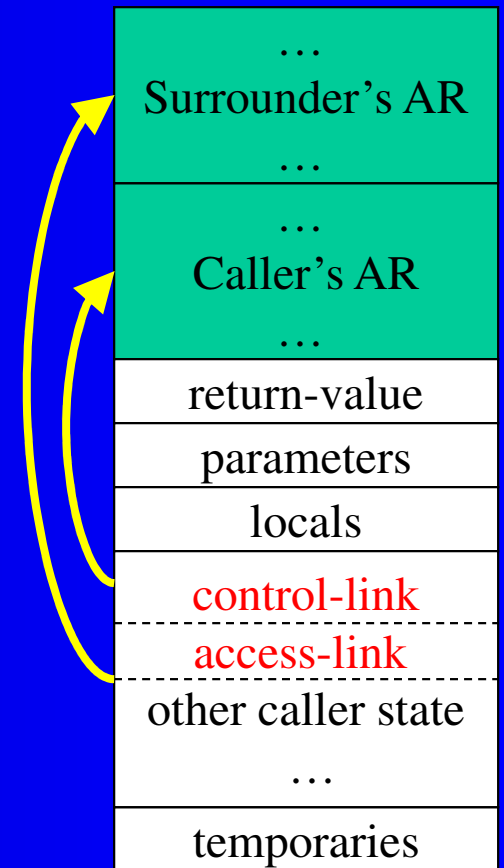


# Static Scope: How it works

In addition to the control-link, static-scoped languages add an \_\_\_\_\_ that points to the AR of the *textually-surrounding scope context* (which may or may not be the caller).

- The textually-surrounding scope context is guaranteed to be already on the stack; otherwise this subprogram could not be called.
- *Non-local accesses* \_\_\_\_\_ to find the definition, instead of control-links.

Access-link traversal occurs at run-time, so \_\_\_\_\_ than accesses to locals.





# Static Scope: Trace

We run our program (*p1*)...

*p1* calls *p2*, which traverses its access-links  
to find integer  $i == 1$

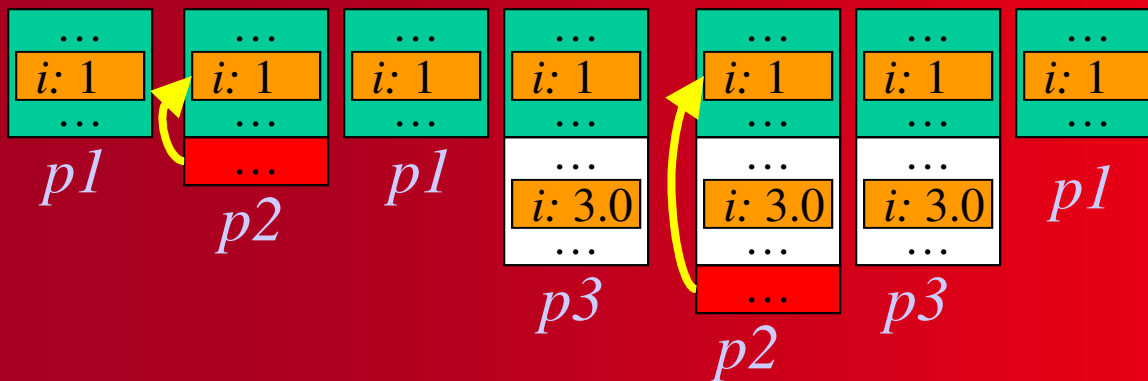
*p2* terminates...

*p1* calls *p3*, which defines *i* as the real 3.0

*p3* calls *p2*, which traverses its access-links  
to find integer  $i == 1$

*p2* terminates...

*p1* terminates...



```
procedure p1 is
```

```
i: integer := 1;
procedure p2 is
begin
```

```
put_line(i);
```

```
end p2;
```

```
procedure p3 is
```

```
i: float := 3.0;
```

begin

p2;

```
end p3;
```

**begin**

p2;

p3;

```
end p1;
```

Note:  $i$ 's physical address is \_\_\_\_\_...



# Non-local Accesses

Non-locals should be avoided where possible because:

- Non-local accesses take 2+ times as long as local accesses
  - Traversal of access-links at run-time increases access time.
- Non-local accesses separate the *use* of an identifier from its *declaration*, \_\_\_\_\_ *and maintainability*.
- Subprograms that access non-locals are *not self-contained*, *reducing their modularity*, as they \_\_\_\_\_ *the non-local*.
- If a subprogram accesses no non-locals, then *it doesn't matter whether the language uses static or dynamic scope*
  - Static- and dynamic-scoped languages \_\_\_\_\_



# Non-local Accesses (ii)

Non-local accesses cannot be completely avoided though...

```
#include <iostream> using namespace std;

class Point {
public:
    Point(int x, int y) { myX = x; myY = y; }
    int getX() const { return myX; }
    int getY() const { return myY; }
    void print(ostream & out)
        { out << '(' << myX << ', ' << myY << ')'; }
private:
    int myX, myY;
};

int main() {
    Point aPoint(0, 0);
    aPoint.print(cout);
}
```

How many  
non-local  
accesses  
can you  
find in  
this  
example  
?



# Non-local Accesses (ii)

Non-local accesses cannot be completely avoided though...

```
#include <iostream> using namespace std;

class Point {
public:
    Point(int x, int y) { myX = x; myY = y; }
    int getX() const { return myX; }
    int getY() const { return myY; }
    void print(ostream & out)
        { out << '(' << myX << ',' << myY << ')'; }
private:
    int myX, myY;
};

int main() {
    Point aPoint(0, 0);
    aPoint.print(cout);
}
```

How many  
non-local  
accesses  
can you  
find in  
this  
example  
?



# Summary

Dynamic-scoped languages bind identifiers at \_\_\_\_\_

- Traversing \_\_\_\_\_ down the run-time stack.

Static-scoped languages bind locals at \_\_\_\_\_

- But binding for non-locals still occurs at run-time via \_\_\_\_\_

Static- and dynamic-scoping behave exactly the same,  
except when resolving \_\_\_\_\_.

