

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:

```
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;
```

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

_____ and most
languages use
_____ scope.

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

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

Dynamic Scope

Lisp uses _____, in which an identifier's binding is determined at _____.

The meaning of a symbol depends on the _____ 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
0       ;; by itself
```

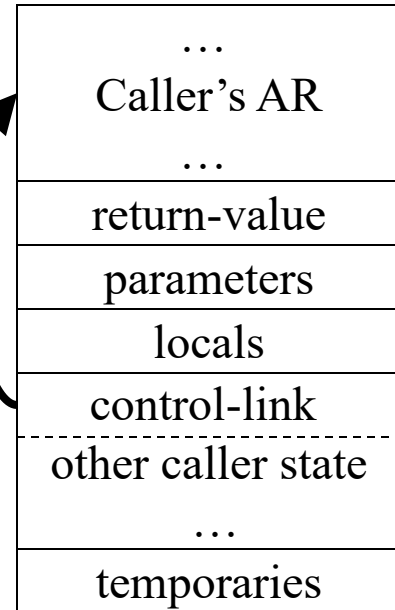
```
(p1)    ;; call p2
1       ;; within p1
3.0     ;; within p3
```

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...

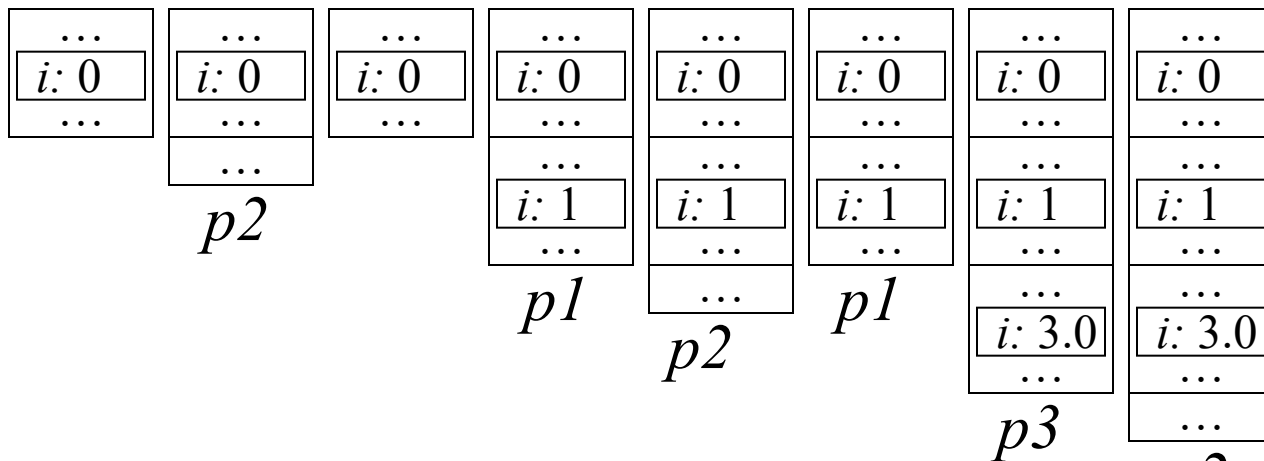
We call *p2*, which searches down, finds *i* == 0
p2 terminates...

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)

0

(p1)

1

3.0

Static Scope

_____ bind identifiers at _____.

The basic static scope rule:

The scope of a binding begins at an identifier's *declaration*; and ends at the *end of the structure* containing that declaration.

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

A _____ 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 _____ *i*, not _____ *i*.
- Ada provides a workaround:
within *p3*, *p1.i* will access _____ *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
}
```

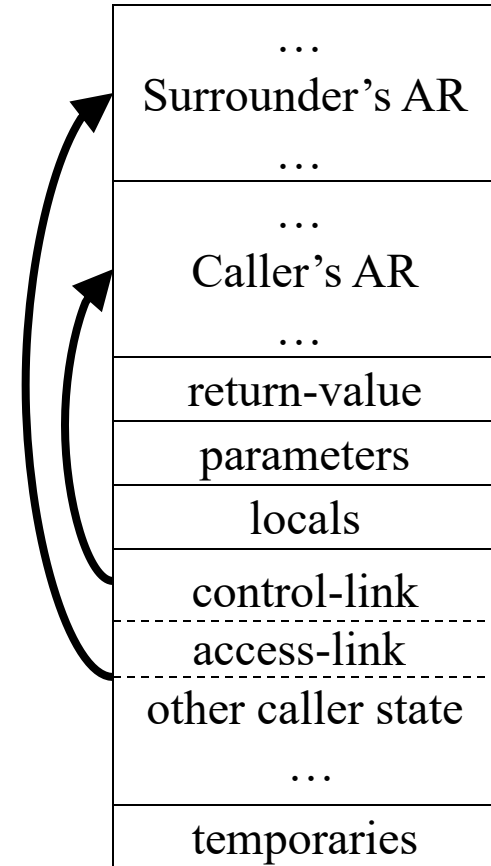
```
procedure p1 is
  i: integer := 1;
  procedure p2 is
  begin
    put_line(i);
  end p2;
  procedure p3 is
    i: float := 3.0;
  begin
    p2;
  end p3;
begin
  p2;
  p3;
end p1;
```

Static Scope: How it works

In addition to the control-link, static-scoped languages add an _____ that points to the AR of the

_____ (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.
- _____. traverse _____ to find the definition, instead of control-links.



Access-link traversal occurs at run-time, so

Static Scope: Trace

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

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

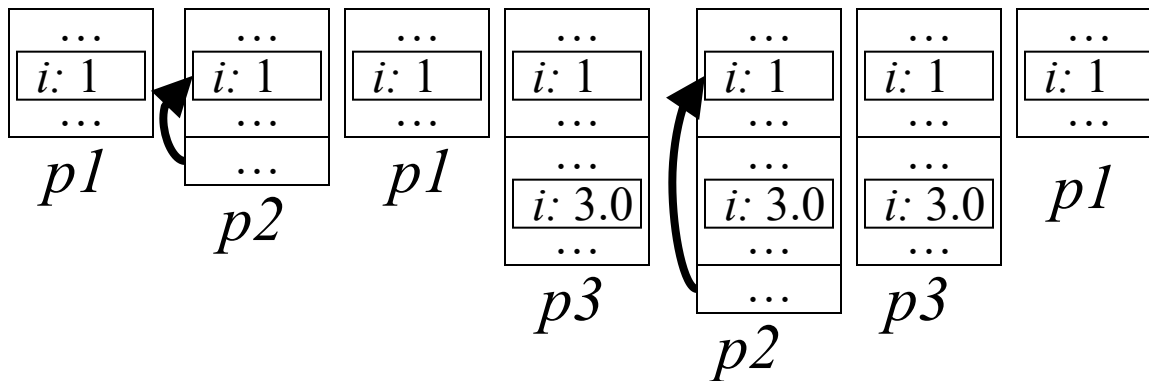
p2 terminates...

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...

p3 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 _____ is _____ ...

Non-local Accesses

Non-locals should be avoided where possible because:

- Non-local accesses take ___ times as long as local accesses
 - Traversal of access-links at run-time increases access time.
- Non-local accesses separate the ___ of an identifier from its _____, *reducing readability and maintainability.*
- Subprograms that access non-locals are _____, *reducing their _____*, as they *depend on the non-local.*
- If a subprogram accesses no non-locals, then *it doesn't matter whether the language uses static or dynamic scope*
 - _____.

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 accesses to non-local variables.

- If you never use non-locals, you'll never have scope problems, regardless of which kind of scope a language uses!