Static vs. Dynamic Scope

- **Pascal is lexically scoped.** We can look (textually, or at compile-time) at a procedure and determine to which object an identifier refers.

- **Some languages (Snobol, APL, Perl, some dialects of LISP) are dynamically scoped.** The binding between an identifier and the object it refers to is not decided until run-time.

---

### Dynamic Scope

The current binding for an identifier is the one last seen during execution and whose scope has yet to be destroyed.

Consider the example on the next slide.

**Static scope:** the program prints 1.
**Dynamic scope:** the program prints 2.

Static scope rules match the use of an identifier with the closest lexically enclosing declaration.

Dynamic scope rules choose the most recent active declaration at runtime.

---

```plaintext
var a: integer;

procedure first();
a := 1;

procedure second();
var a: integer;
first();

begin
a := 2;
second();
write(a);
end
```
Dynamic Scope — Problems

```pascal
var max : integer;
procedure scale(x : integer) : real;
  return x/max;
procedure compute(y : integer);
  var max : integer;
  write(scale(y));
```

- Dynamic scope makes it easy to accidentally redefine a variable.

Dynamic Scope — Advantages

```pascal
procedure A(base : integer)
  printInt(base, 245);
procedure B(base : integer)
  A();
procedure C(base : integer)
  B();
begin C(16); end
```

- We often have to pass around state so that deeply nested procedures can make use of it. DEBUG-flags is a common example.

Dynamic Scope — Advantages...

```pascal
var base : integer := 10;
procedure A()
  printInt(base, 245);
procedure B()
  A();
procedure C()
  B();
begin
  var last_base := base;
  base := 16; C();
  base := last_base;
end
```

- We can, of course, use global variables.

Dynamic Scope — Advantages...

```pascal
procedure A()
  printInt(base, 245);
procedure B()
  A();
procedure C()
  B();
begin
  var base : integer := 16;
  C();
end
```

- Dynamic scope makes it easy customize the behavior of procedures.
Subroutine Closures

- A closure is a structure $(\text{procedure_addr}, \text{environment})$.
- To pass $C()$ to $A$ we construct a closure consisting of $C$'s address and the static link that would have been used if $C$ would have been called directly:

  ```
  procedure A(procedure P)
  P();
  end
  procedure C(); begin end;
  begin
  A(C);
  end
  ```

- There are two $I$s when $B$ is called.

  ```
  procedure A(I:integer; procedure P)
  procedure B(); begin write(I); end;
  begin
  if I > 1 then P() else A(2,B);
  end
  procedure C(); begin end;
  begin
  A(1,C);
  end
  ```

First-Class Subroutines

- A language construct is first-class if it can be passed as a parameter, returned from a subroutine, or assigned to a variable.
- A language construct is second-class if it can be passed as a parameter but not be returned from a subroutine, or assigned to a variable.
- A language construct is third-class if it can’t even be passed as a parameter.
- Procedures are second-class in most imperative languages.
First-Class Subroutines...

If a procedure can be returned as the result of a function we could reference an environment that has gone out of scope:

```pascal
procedure A() : procedure;
  var x : integer := 5;
  procedure B();
    write(x);
  end;
begin
  return B;
end;

begin
  var X : procedure := A();
  X();
end
```

First-Class Subroutines...

In functional languages functions are first-class.

- Functional languages specify that local variables have unlimited extent — they exist for as long as someone references them.
- Algol-like languages specify that local variables have limited extent — they exist until the scope in which they are declared is exited.
- Objects with limited extent can be stored on a stack. Objects with unlimited extent must be stored on the heap.

First-Class Subroutines...

C and C++ do not have nested scope — no problem.

- Modula-2 — global procedures are first-class (can be stored), local procedures are third-class.
- Modula-3 — global procedures are first-class, local procedures are second-class (can be passed as parameters).
- Ada 83 — procedures are third class.
- Ada 95 — nested procedures can be returned if the scope in which it was declared is at least as wide as that of the declared return type. I.e. a procedure can only be propagated to an area of the program where the referencing environment is active.

First-Class Subroutines...

- Call-With-Current-Continuation

The Scheme built-in function call-with-current-continuation (also called call/cc) takes a function as argument:

```scheme
call-with-current-continuation (foo) (foo cont)
```

- foo takes a continuation as argument.

- (call/cc foo) calls foo, passing it the current continuation.

- A continuation is a closure that holds the current program counter and environment.
Call-With-Current-Continuation...

- foo can invoke the continuation and immediately return to the situation as it was when the call was made.
- Any intermediate stack frames are popped off.
- Continuations are first-class: you can store them in variables, return them from functions, etc.
- call/cc can be used as a general building-block to construct a variety of control structures, such as iterators and coroutines.
- Continuations can, for example, be used to quickly exit a tree-search procedure once the node we’re looking for has been found.

  The function throws the continuation the value 99 which makes it pop out of the current evaluation and return 99:

  ```lisp
  > (call/cc (lambda (c) (c 99)))
  99
  ```

  The expression `(* [] 76)` is never executed. Rather, the function pops out and returns 99:

  ```lisp
  > (call/cc (lambda (c) (* (c 99) 76)))
  99
  ```

Call-With-Current-Continuation...

- Continuations can be stored in variables and invoked later:

  ```lisp
  > (let ((cont #f))
      (call/cc (lambda (k) (set! cont k)))
      (cont #f))
  99
  ```

  Or, like this:

  ```lisp
  > (define cont #f)
  > (+ 5 (call/cc (lambda (e) (set! cont e) (* 4 3))))
  17
  > (cont 10)
  15
  ```

setjmp/longjmp

- In C, setjmp/longjmp can be used to implement exceptional control flow:

  ```c
  if (!setjmp(buffer)) {
      /* setjmp returned 0. Protected code. */
      ...
      longjmp(buffer);
      ...
  } else {
      /* setjmp returned 1. Handler code. */
  }
  ```
**set jmp/long jmp...**

- The first time `set jmp` returns 0 and execution continues as normal. When `long jmp` is called it appears as if `set jmp` has returned for the second time, this time returning 1. The state is now the same as it was when `set jmp` was first called.
- `set jmp`'s buffer argument stores the program's current state, in particular register values.
- Unlike a “real” exception handler, the stack is not rewound nicely. Rather, all stack frames are thrown away. This can lead to problems if not all register values have been saved back in memory. Variables that may be thus affected should be declared `volatile`, i.e. they will always be returned to memory after operated on.

---

**Tail Recursion**

- A function is **tail-recursive** if there is no more work to be done after the recursive call.
- Tail-recursive functions are important because they can be easily be made iterative — no stack space needs to be allocated dynamically.
- For tail-recursive functions the compiler can **reuse** the space of the current stack frame instead of allocating a new one for the recursive call.

---

**Tail Recursion...**

```c
int gcd(int a, int b) {
  if (a == b) return a;
  else if (a > b) return gcd(a-b, b);
  else return gcd(a, b-a);
}
```

```c
int gcd(int a, int b) {
  start:
  if (a == b) return a;
  else if (a > b) {a=a-b; goto start; }
  else {b=b-a; goto start; }
}
```

---

**Tail Recursion...**

- You can often transform a non-tail-recursive function into a tail-recursive one.
- The idea is to pass a **continuation** of the work that is to be done after the call as a parameter to the call.
- This is called **continuation-passing style** (CPS).
- The next slide shows how the factorial function has been made tail-recursive using the CPS transformation.
Tail Recursion...

(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1))))))

(define (fact-cps n C)
  (if (= n 1)
      (C 1)
      (fact-cps (- n 1) (
        lambda(v) (C (* n v)))))))

(fact-cps 5 (lambda(v) (display v)))

Coroutines...

Coroutines are supported by Simula and Modula-2. They are similar to Java’s threads, except the programmer has to explicitly transfer control from one execution context to another.

Thus, like threads several coroutines can exist simultaneously but unlike threads there is no central scheduler that decides which coroutine should run next.

A coroutine is represented by a closure.

A special operation transfer(C) shifts control to the coroutine C, at the location where C last left off.

The next slide shows an example from Scott where two coroutines execute “concurrently”, by explicitly transferring control between each other.

In the example one coroutine displays a moving screen-saver, the other walks the file-system to check for corrupt files.

Coroutines...

var us, cfs: coroutine;
coroutine update_screen() {
  ...
detach
  loop {
    ...
      transfer(cfs) ...
  }
}
coroutine check_file_system() { ...
main () { ...

Coroutines...
Coroutines...  

```plaintext
coroutine check_file_system() {
  ...
  detach
  for all files do {
    ... transfer(cfs)
    ... transfer(cfs)
    ... transfer(cfs) ...
  }
}
```

main () {
  us := new update_screen();
  cfs := new check_file_system();
  transfer(us);
}

Coroutines in Modula-2

- Modula-2's system module provides two functions to create and transfer between coroutines:

```plaintext
PROCEDURE NEWPROCESS(   
  proc: PROC;           (* The procedure *)  
  addr: ADDRESS;        (* The stack    *)  
  size: CARDINAL;       (* The stack size *)  
  VAR new: ADDRESS);    (* The coroutine *)

PROCEDURE TRANSFER(   
  VAR source: ADDRESS;  (* Current coroutine *)  
  VAR destination: ADDRESS); (* New coroutine *)
```

- The first time `TRANSFER` is called `source` will be instantiated to the main (outermost) coroutine.

Coroutines in Modula-2...

```plaintext
VAR crparams: CoroutineParameters;
  source: ADDRESS; (* current coroutine is called by this *)
  newcr: ADDRESS; (* coroutine just created by NEWPROCESS *)

PROCEDURE Coroutine;
  VAR myparams: CoroutineParameters;
  BEGIN
    myparams := crparams;
    TRANSFER(newcr, source); (* return to calling coroutine *)
    (* rest of coroutine *)
  END Coroutine;

PROCEDURE Setup(params: CoroutineParameters; proc: PROC);
  BEGIN
    NEWPROCESS(proc, addr, size, newcr);
    crparams := params; TRANSFER(source, newcr);
  END Setup;
```

Readings and References

- Read Scott, pp. 115–116, 129–132, 139–144, 298, 471–479


  - [http://www.mathematik.uni-ulm.de/oberon/0.5/articles/coroutines.html](http://www.mathematik.uni-ulm.de/oberon/0.5/articles/coroutines.html)