1 Subroutine Closures

- A closure is a structure
  
  \[(\text{procedure.addr, environment}).\]

- 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:

\[
\text{program M;}
\text{procedure A(procedure P)}
\text{P();}
\text{end}
\text{procedure C(); begin end;}
\text{begin}
\text{A(C);}
\text{end}
\]

2 Deep Binding

- When a reference to a procedure is created (for example by passing it as a reference to another procedure), when are scope rules applied?

  1. When the reference is first created?
  2. When the routine is first called?

- Early binding of a referencing environment (what Pascal uses) is called deep binding.
3 Subroutine Closures... 

```plaintext
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
```

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

4 Subroutine Closures... 

```
B()  
A(2,B)  
A(1,C)  
main
```

- A closure was created for B when A(2,B) was closed, hence B will print 1.

5 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.

6 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:
procedure A() : procedure;
var x : integer := 5;
procedure B();
    write(x);
end
begin
    return B;
end;
begin
    var X : procedure := A();
    X();
end

7 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.

8 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.

9 Call-With-Current-Continuation

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

  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.
10 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.

11 Call-With-Current-Continuation... 

- 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
  ```

12 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
  ```

13 Readings and References 

- Read Scott, pp. 141–143