Stateful vs. Non-Stateful Programming

Outline

- ◇ Procedure implementation
  - Activation records
  - Recursion
  - Tail-call optimization
  - Parameter passing

- ◇ Nested Procedures
  - HOP + assignment
  - Scoping

- ◇ The language Oz
  - Combines stateful and non-stateful programming

Procedure Implementation

- Parameter passing
  - Actual arguments must be bound to formal arguments
- Return mechanism
  - Upon return from invocation, control must be transferred back to callee
- Local variable allocation
  - Each invocation must have a separate set of local variables, to allow for recursion
- Solution: *Activation Record*

VAL Code Structure for Procedures

Caller

⋮

caller prologue

- save caller registers
- bind actual arguments to formal arguments
- set up return address

perform call

`return_address:`

caller epilogue

- de-allocate arguments
- restore caller registers

⋮

Callee

callee prologue

- save callee registers
- allocate local variables

body of procedure

callee epilogue

- deallocate local vars
- restore callee registers

return

Stack

caller registers
arguments
return address
callee registers
local variables

EBP

Activation Record

Activation Records for C

```
:
r = f(1,2,3,4) ;
:
int f(int a, int b,
      int c, int d) {
  int x, y, z ;
:
}
```

return address

current EBP

green entries: callee saved registers

Stack

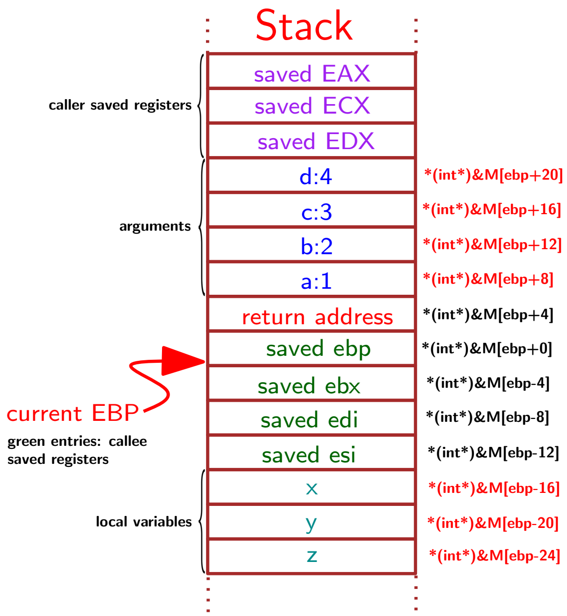
	Stack	
caller saved registers	saved EAX	
	saved ECX	
	saved EDX	
arguments	d:4	*(int*)&M[ebp+20]
	c:3	*(int*)&M[ebp+16]
	b:2	*(int*)&M[ebp+12]
	a:1	*(int*)&M[ebp+8]
	return address	*(int*)&M[ebp+4]
	saved ebp	*(int*)&M[ebp+0]
	saved ebx	*(int*)&M[ebp-4]
	saved edi	*(int*)&M[ebp-8]
	saved esi	*(int*)&M[ebp-12]
local variables	x	*(int*)&M[ebp-16]
	y	*(int*)&M[ebp-20]
	z	*(int*)&M[ebp-24]

Concrete Example

```
:
r = f(1,2,3,4) ;
:
```

return address

```
int f(int a, int b,
      int c, int d) {
  int x, y, z ;
  x = a + b + c ;
  y = b + c + d ;
  z = x / y ;
  return 2 * z ;
}
```



Caller

```
    esp -= 4 ; *(int*)&M[esp] = eax ; // push eax
    esp -= 4 ; *(int*)&M[esp] = ecx ; // push ecx
    esp -= 4 ; *(int*)&M[esp] = edx ; // push edx
    esp -= 4 ; *(int*)&M[esp] = 4 ; // push 4
    esp -= 4 ; *(int*)&M[esp] = 3 ; // push 3
    esp -= 4 ; *(int*)&M[esp] = 2 ; // push 2
    esp -= 4 ; *(int*)&M[esp] = 1 ; // push 1
    eax = (int) && return_address ;
    esp -= 4 ; *(int*)&M[esp] = eax ; // push return addr
    goto f ;
return_address:
    esp += 16 ; // clear arguments
    edx = *(int*)&M[esp] ; esp += 4 ; // pop edx
    ecx = *(int*)&M[esp] ; esp += 4 ; // pop edx
    *(int*)&M[ebp-20] = eax ; // assume r is at ebp-20
    eax = *(int*)&M[esp] ; esp += 4 ; // pop eax
```

prologue

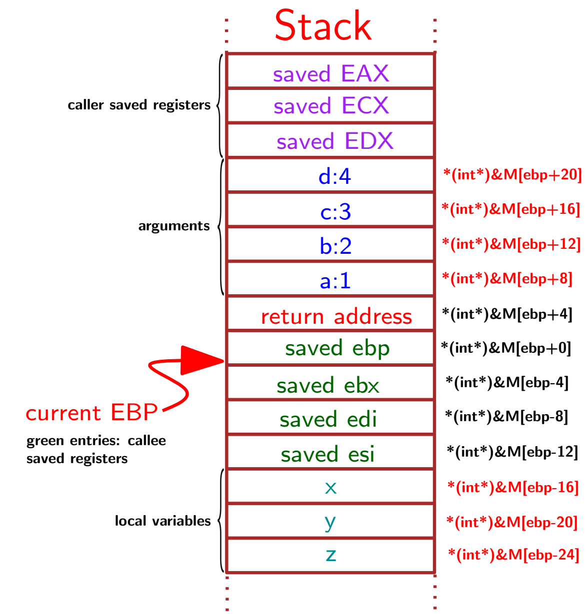
epilogue

Concrete Example

```
:
r = f(1,2,3,4) ;
:
```

return address

```
int f(int a, int b,
      int c, int d) {
  int x, y, z ;
  x = a + b + c ;
  y = b + c + d ;
  z = x / y ;
  return 2 * z ;
}
```



prologue

Callee

```
f:  esp -= 4 ; *(int*)&M[esp] = ebp ; // push ebp
    ebp = esp ;
    esp -= 4 ; *(int*)&M[esp] = ebx ; // push ebx
    esp -= 4 ; *(int*)&M[esp] = edi ; // push edi
    esp -= 4 ; *(int*)&M[esp] = esi ; // push esi
    esp -= 12 ; // allocate space for local vars

    eax = *(int*)&M[ebp+8] ;
    eax += *(int*)&M[ebp+12] ;
    eax += *(int*)&M[ebp+16] ;
    *(int*)&M[ebp-16] = eax ;

    eax = *(int*)&M[ebp+12] ;
    eax += *(int*)&M[ebp+16] ;
    eax += *(int*)&M[ebp+20] ;
    *(int*)&M[ebp-20] = eax ;
```

```
eax = *(int*)&M[ebp-16] ;
eax /= *(int*)&M[ebp-20] ;
*(int*)&M[ebp-24] = eax ;

eax = 2 ;
eax *= *(int*)&M[ebp-24] ;

esp += 12 ; // clear local vars
esi = *(int*)&M[esp] ; esp += 4 ; // restore
edi = *(int*)&M[esp] ; esp += 4 ; // callee
ebx = *(int*)&M[esp] ; esp += 4 ; // registers
ebp = *(int*)&M[esp]; esp += 4 ;
esp += 4 ; goto * *(void*)&M[esp-4] ; // return
```

epilogue

Important Points

- ◇ Each function invocation creates a new AR, which is discarded when the function returns.
- ◇ ARs are allocated on the stack; this is natural because of the way function calls are sequenced.
- ◇ ARs are pointed to by a *frame pointer*, which is typically implemented as a CPU register; for the Pentium, it is the EBP register.
- ◇ In the case of C, the frame pointer points to the "middle" of the AR, so that it allows for variable number of arguments in a function call.
- ◇ In spite of the AR not being placed at a fixed address, inside the body of the callee, each variable and argument **can be translated into a fixed VAL expression**.

ARs and Recursion

```
int fib(int n) {
    int x, y ;
    if ( n <= 1 )
        return n ;
    x = fib(n-1) ;
    y = fib(n-2) ;
    return x+y ;
}
```

$n \rightarrow$ ebp+8

$x \rightarrow$ ebp-16

$y \rightarrow$ ebp-20

```
fib:  esp -= 4 ; *(int*)&M[esp] = ebp ; // push ebp
      ebp = esp ;
      esp -= 4 ; *(int*)&M[esp] = ebx ; // push ebx
      esp -= 4 ; *(int*)&M[esp] = edi ; // push edi
      esp -= 4 ; *(int*)&M[esp] = esi ; // push esi
      esp -= 8 ; // allocate space for local vars

      if (*(int*)&M[ebp+8] > 1) goto skip ;
      eax = *(int*)&M[ebp+8] ;
      goto exit_fib ; // jump to epilogue

skip:
      esp -= 4 ; *(int*)&M[esp] = eax ; // push eax
      esp -= 4 ; *(int*)&M[esp] = ecx ; // push ecx
      esp -= 4 ; *(int*)&M[esp] = edx ; // push edx
      eax = *(int*)&M[ebp+8] ;
      eax -= 1 ;
      esp -= 4 ; *(int*)&M[esp] = eax ; // push n-1
      eax = (int) && return_address1 ;
      esp -= 4 ; *(int*)&M[esp] = eax ; // push return addr
      goto fib ;
return_address1:
      esp += 4 ; // clear arguments
      edx = *(int*)&M[esp] ; esp += 4 ; // restore edx
      ecx = *(int*)&M[esp] ; esp += 4 ; // restore edx
      *(int*)&M[ebp-16] = eax ; // x = return value
      eax = *(int*)&M[esp] ; esp += 4 ; // restore eax
```

ARs and Recursion

```
int fib(int n) {
    int x, y ;
    if ( n <= 1 )
        return n ;
    x = fib(n-1) ;
    y = fib(n-2) ;
    return x+y ;
}
```

$n \rightarrow$ ebp+8

$x \rightarrow$ ebp-16

$y \rightarrow$ ebp-20

```
        esp -= 4 ; *(int*)&M[esp] = eax ; // push eax
        esp -= 4 ; *(int*)&M[esp] = ecx ; // push ecx
        esp -= 4 ; *(int*)&M[esp] = edx ; // push edx
        eax = *(int*)&M[ebp+8] ;
        eax -= 2 ;
        esp -= 4 ; *(int*)&M[esp] = eax ; // push n-2
        eax = (int) && return_address2 ;
        esp -= 4 ; *(int*)&M[esp] = eax ; // push return addr
        goto fib ;
return_address2:
        esp += 4 ; // clear arguments
        edx = *(int*)&M[esp] ; esp += 4 ; // restore edx
        ecx = *(int*)&M[esp] ; esp += 4 ; // restore edx
        *(int*)&M[ebp-20] = eax ; // y = return value
        eax = *(int*)&M[esp] ; esp += 4 ; // restore eax

        eax = *(int*)&M[ebp-16] ; // load x
        eax += *(int*)&M[ebp-20] ; // add y, return value set now

exit_fib: // callee epilogue
        esp += 8 ; // clear local vars
        esi = *(int*)&M[esp] ; esp += 4 ; // restore
        edi = *(int*)&M[esp] ; esp += 4 ; // callee
        ebx = *(int*)&M[esp] ; esp += 4 ; // registers
        ebp = *(int*)&M[esp]; esp += 4 ;
        esp += 4 ; goto * *(void*)&M[esp-4] ; // return
```

ARs and Recursion: Step-by-Step

fib(3)

Stack

$n = 3$

$x = ?$
$y = ?$

ARs and Recursion: Step-by-Step

	Stack
fib(3)	n = 3 x = ? y = ?
fib(2)	n = 2 x = ? y = ?

ARs and Recursion: Step-by-Step

Stack

fib(3)

$n = 3$

$x = ?$
$y = ?$

fib(2)

$n = 2$

$x = ?$
$y = ?$

fib(1)

$n = 1$

$x = ?$
$y = ?$

ARs and Recursion: Step-by-Step

Stack

fib(3)

n = 3

x = ?
y = ?

fib(2)

n = 2

x = 1
y = ?

ARs and Recursion: Step-by-Step

Stack

fib(3)	n = 3 x = ? y = ?
fib(2)	n = 2 x = 1 y = ?
fib(0)	n=0 x = ? y = ?

ARs and Recursion: Step-by-Step

Stack

fib(3)

$n = 3$

$x = ?$
$y = ?$

fib(2)

$n = 2$

$x = 1$
$y = 0$

ARs and Recursion: Step-by-Step

fib(3)

Stack

n = 3
x = 1
y = ?

ARs and Recursion: Step-by-Step

Stack

fib(3)

n = 3

x = 1
y = ?

fib(1)

n = 1

x = ?
y = ?

ARs and Recursion: Step-by-Step

fib(3)

Stack

n = 3
x = 1
y = 1

ARs and Recursion: Step-by-Step

Stack

returns 2 $\longleftarrow$ fib(3)

Tail Call Optimization

```
int factorial(int n, int k) {
    if ( n == 0 ) return 1 ;
    return factorial(n-1,k*n) ;
}
```

Without Tail Call Optimization

```
factorial:
      esp -= 4 ; *(int*)&M[esp] = ebp ;
      ebp = esp ;

      eax = *(int*)&M[ebp+8] ;
      if ( eax != 0 )   goto skip ;
      eax = 1 ;
      goto factorial_exit ;
skip:
      eax -- ;
      esp -= 4 ; *(int*)&M[esp] = eax ;
      eax ++ ;
      eax += *(int*)&M[ebp+12] ;
      esp -= 4 ; *(int*)&M[esp] = eax ;
      eax = (int) && return_address ;
      esp -= 4 ; *(int*)&M[esp] = eax ;
      goto factorial ;
      esp += 8 ;
factorial_exit:
      ebp = *(int*)&M[ebp] ;
      esp += 8 ;
      goto * *(int*)&M[esp-4] ;
```

Create new AR upon recursive call

With Tail Call Optimization

```
factorial:
      esp -= 4 ; *(int*)&M[esp] = ebp ;
      ebp = esp ;
factorial_tco:
      eax = *(int*)&M[ebp+8] ;
      if ( eax != 0 )   goto skip ;
      eax = 1 ;
      goto factorial_exit ;
skip:
      eax -- ;
      *(int*)&M[ebp+8] = eax ;
      eax ++ ;
      eax += *(int*)&M[ebp+12] ;
      *(int*)&M[ebp+12] = eax ;


      goto factorial_tco ;

factorial_exit:
      ebp = *(int*)&M[ebp] ;
      esp += 8 ;
      goto * *(int*)&M[esp-4] ;
```

Reuse existing AR upon recursive call

Parameter Passing

- Call-by-Value : The values of arguments are placed in the AR.
  - Changes made to formal arguments in the procedure are not propagated back into the calling environment.
  - Used in the previous example.

- Call-by-Reference : The addresses of the arguments are placed in the AR.
  - Restriction may be placed that each actual argument be a standalone variable.
  - Each accesss to the argument inside the procedure must dereference the address.
  - Changes to arguments inside the procedure will be propagated to the calling environemnt.
  - Concurrent threads may observe intermediate values of argument before return from procedure (security concern).

- Call-by-Valiue-Return : Argument values are placed in the AR. The values are copied back to their sources upon return.
  - Similar effect to call-by-reference.
  - Concurrent thread cannot observe intermediate values (security concern addressed).
  - Faster procedure execution, since there's no need to dereference.
  - Potentially significant overhead for large arguments.

- Call-by-Name : used in macro expanison (the `#define` directive of the C preprocessor).
  - Textually replaces the formal argument with the actual argument in the body of the procedure.
  - Potentially inefficient, since argument expressions may be evaluated multiple times (addressed with memoization, which is used in *lazy evaluation*)

Nested Procedures

```python
def f(x):
    def g(y):
        return x+y
    return g

>>> h = f(3)
>>> h(2)
5
>>> h(4)
7
```

```scheme
(define (f x)
    (define (g y) (+ x y))
    g)

(define h (f 3)
(h 2)
5
(h 4)
7
```

```ocaml
let f x = let g y = x+y in g ;;

let h = f 3 ;;

h 2 ;; -- evaluates to 5
h 4 ;; -- evaluates to 7
```

```c
int (*f(int x))(int) {
    int g(int y) {
        return x+y ;
    }
    return &g ;
}

int main() {
  int (*h)(int) = f(3) ;
  printf("%d %d\n",h(2),h(4));
}
```

Nested Procedures

```python
def f(x):
    def g(y):
        return x+y
    return g

>>> h = f(3)
>>> h(2)
5
>>> h(4)
7
```

```scheme
(define (f x)
    (define (g y) (+ x y))
    g)

(define h (f 3))
(h 2)
5
(h 4)
7
```

```ocaml
let f x = let g y = x+y in g ;;

let h = f 3 ;;

h 2 ;; -- evaluates to 5
h 4 ;; -- evaluates to 7
```

```c
int (*f(int x))(int) {
    int g(int y) {
        return x+y ;
    }
    return &g ;
}

int main() {
  int (*h)(int) = f(3) ;
  printf("%d %d\n",h(2),h(4));
}
```

What if we assign to `x` ? (we can't do that in Ocaml, unless we declare `x` as a reference)

Nested Procedures with Assignment

```python
def f(x):
    def g(y):
        nonlocal x
        x = x + y
        return x
    return g

>>> h = f(1)
>>> h(5)
6
>>> h(5)
11
>>> h(5)
16
```

```scheme
(define (f x)
  (define (g y)
    (set! x (+ x y))
    x)
  g)

> (define h (f 1))
> (h 5)
6
> (h 5)
11
> (h 5)
16
>
```

`x` is stored in the AR of `f`, which is not discarded upon return. Variable `x` is now *hidden*.

Scoping

```
def f(x):
    def g(y):
        nonlocal x
        x = x + y
        return x
    return g

>>> x = 100
>>> h = f(1)
>>> h(5)
6
>>> h(5)
11
>>> h(5)
16
```

If this `x` is changed: *static scoping*

If this `x` is changed: *dynamic scoping*

- *Static scoping* : non-local variable access refers to the variable that is closest to the place where the procedure is *defined*.
- *Dynamic scoping* : non-local variable access refers to the variable that is closest to the place where the procedure is *used*.

The Language Oz

- Combines Prolog-style unification with higher-order programming techniques specific of functional languages.
- Allows stateful programming in an elegant manner.
- Fine-grain concurrency model that we'll study later.
- No implicit backtracking.
- Terms are still allowed as data
- Arithmetic expressions get evaluated and have value.
- No op declarations.

Oz vs. Prolog

- Variables are single assignment, and must have capitalized names.
  - Variables must be declared, and they are initially *unbound*.
- There are no predicates. Procedures can be declared as abstractions, and assigned to variables (consequently, procedure names are capitalized).
- Equality denotes unification.
  - When unification fails, an exception is thrown, rather than trigger backtracking.
- In a procedure, any variable can be used for input or output, like in Prolog.

Example

```
declare
proc {App X Y R}
   case X
   of nil then R = Y
   [] H|T then R1 {App T Y R1} in R = H|R1
   end
end
R Z
{App [1 2 3] [4 5 6] R}
{Browse R}
{App [1 2 3] Z [1 2 3 4 5 6]}
{Browse Z}
W
{App W [4 5 6] [1 2 3 4 5 6]}
{Browse W}
```

Example

All top-level variables will be global

Procedure definition

Procedure name

Procedure arguments

```
declare
proc {App X Y R}
   case X
   of nil then R = Y
   [] H|T then R1 {App T Y R1} in R = H|R1
   end
end
R Z
{App [1 2 3] [4 5 6] R}
{Browse R}
{App [1 2 3] Z [1 2 3 4 5 6]}
{Browse Z}
W
{App W [4 5 6] [1 2 3 4 5 6]}
{Browse W}
```

Example

```
declare
proc {App X Y R}
   case X
   of nil then R = Y
   [] H|T then R1 {App T Y R1} in R = H|R1
   end
end
R Z
{App [1 2 3] [4 5 6] R}
{Browse R}
{App [1 2 3] Z [1 2 3 4 5 6]}
{Browse Z}
W
{App W [4 5 6] [1 2 3 4 5 6]}
{Browse W}
```

Pattern matching. Suspends if X unbound.

Scope definition

Recursive call

Local variable

Example

```
declare
proc {App X Y R}
   case X
   of nil then R = Y
   [] H|T then R1 {App T Y R1} in R = H|R1
   end
end
R Z
{App [1 2 3] [4 5 6] R}
{Browse R}
{App [1 2 3] Z [1 2 3 4 5 6]}
{Browse Z}
W
{App W [4 5 6] [1 2 3 4 5 6]}
{Browse W}
```

terminator

global variables

binds R to [1 2 3 4 5 6]

prints R in separate window

binds Z to [4 5 6]

blocks (can be unblocked by another thread)

(will be explained in another lecture)

executed sequentially

Syntactic Sugar

```
R = {App X Y}
```
≡
```
{App X Y R}
```

```
proc {App X Y R} ... end
```
≡
```
App = proc {$ X Y R} ... end
```

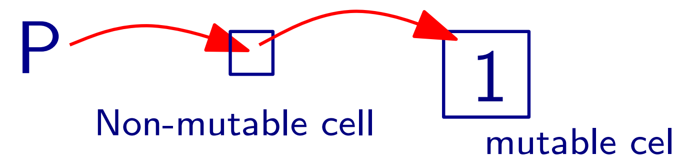
Alternative functional notation

```
fun {App X Y}
   case X
   of nil then Y
   [] H|T then H | {App T Y}
   end
end
```
≡
```
App = fun {$ X Y}
   case X
   of nil then Y
   [] H|T then H | {App T Y}
   end
end
```

Stateful Programming in Oz

```
declare
fun {Factorial N}
   P = {NewCell 1}
   proc {Helper N}
      if ( N == 0 )
      then skip
      else
            P := @P * N
            {Helper N-1}
      end
   end
   in
   {Helper N}
   @P
end
{Browse {Factorial 5}}
```

Stateful Programming in Oz




```
declare
fun {Factorial N}
   P = {NewCell 1}
   proc {Helper N}
      if ( N == 0 )
      then skip
      else
            P := @P * N
            {Helper N-1}
      end
   end
   in
   {Helper N}
   @P
end
{Browse {Factorial 5}}
```

create new mutable cell

local function

dereference and assignment to mutable cell

access value of mutable cell

operator evaluated right away

sequential execution

body of function

return value (cell dereference)