□ Continuations

The rest of the computation.

```
1 + (\x -> x * 3) (1 + 1)
```

```
(+) 1 ((\x -> (*) x 3) ((+) 1 1))
```

Call-by-value breakdown:

```
let v1 = (+); v2 = 1
    v3 = (\v -> let v1 = (*); v2 = 3 in v1 v v3)
    v4 = (+); v5 = 1; v6 = 1
    v7 = v4 v5 v6
    v8 = v3 v7
in v1 v2 v8
```

□ Continuation Passing Style

Making continuations explicit.

```
plus e1 e2 k = e1 $ \v1 ->
               e2 $ \v2 ->
               k $ v1 + v2
(^) x k = k x
```

`plus (plus (1 ^) (2 ^)) (plus (3 ^) (4 ^)) id` $\hookrightarrow$ 10

CPS expressions require an initial continuation to know what to do with the answer.

Any term can be transformed into CPS.

□ CPS

- Continuations and CPS studied since mid 60's.
- CPS makes control flow explicit.
- CPS used to structure compilers.
- CPS is very similar to Monadic style.

□ CPS Terms & Values

```
type K v = (v -> Answer) -> Answer

data Term = Var String
          | String :\ Term
          | Term :$: Term

data Value = Err String
           | Fun (Value -> K Value)
```

□ CPS Interpreter

```
type Answer = Value

getVar :: Env -> String -> K Value
getVar ((x,v):xs) y k = if x == y then k v else getVar xs y k
getVar [] y k = k $ Err $ "unbound variable "++y

apply :: Value -> Value -> K Value
apply (Fun v1) v2 k = v1 v2 k
apply f _ k = k $ Err $ "expected function, found "++(show f)

interp :: Term -> Env -> K Value
interp (Var x) env k = getVar env x k
interp (x :\ e) env k = k $ Fun $ \v -> interp e ((x,v):env)
interp (e1 :$: e2) env k =
    interp e1 env $ \v1 ->
    interp e2 env $ \v2 ->
    apply v1 v2 k

test e = interp e [] id
```

□ The Current Continuation

```
data Term = ... | CallCC String Term

callcc :: ((v -> K w) -> K v) -> K v
callcc f k = f (\a k' -> k a) k

interp (CallCC x e) env k =
    callcc (\v -> interp e ((x,Fun v):env)) k
```

Note:

```
callcc :: ((v -> (w -> Answer) -> Answer) -> (v -> Answer) -> Answer) ->
          (v -> Answer)
          -> Answer
```

□ Using CallCC

Access to the current continuation opens up many possibilities.

```
Con 1 :+: CallCC "k" (Con 2 :+: (Var "k" :$: Con 3))

Con 1 :+: CallCC "k" (Con 2 :+: (Var "k" :$: (Var "k" :$: Con 3)))

Con 1 :+: CallCC "k" (Con 2 :+: Con 3)

CallCC "k" $ "x" :\ (Var "k") :$: ("y" :\ Var "x" :+: Var "y")
```

Can use `CallCC` to get static exceptions:

```
handle e e' = CallCC "k" (e :$: ("_" :\ Var "k" :$: e'))
raise e = e :$: Skip

"x" :\ handle ("ex" :\ If (Var "x" :==: Con 10)
                          (raise (Var "ex"))
                          (Var "x" :+: Con 2)
              )
              (Con (-1))
```

Many other strange and complex possibilities.

□ Extending CPS Interpreter

```
data Term = ... | Skip | Out
data Value = ... | Unit

type Answer = (String, Value)

out :: Show a => a -> K Value
out v k = let (o, v1) = k Unit in ((show v)++"; "++o, v1)

interp Skip env k = k Unit
interp (Out e) env k = interp e env $ \v -> out v k
```

Other extensions similarly done.

□ Monads via CPS

```
type Answer = M Value

type M a = Int -> (String, (a, Int))

tick :: K ()
tick k = bind (\s -> ("",((),s+1))) k

out :: Show a => a -> K Value
out v k = bind (\s -> ((show v)++"; ",(Unit, s))) k

apply :: Value -> Value -> K Value
apply (Fun v1) v2 k = tick $ \_ -> v1 v2 k
apply f _ k = k $ Err $ "expected function, found "++(show f)
```

□ CPS via Monads

```
type K a = (a -> Answer) -> Answer

unit :: a -> K a
unit a = \k -> k a

bind :: K a -> (a -> K b) -> K b
bind m f = \k -> m (\v -> f v k)


bind unit x == x

\x -> bind f (unit x) == f

\x -> bind (\y -> bind g (f y)) x == \x -> bind g (bind f x)
```

□ Conclusion

- Can write monads as continuations.

- Can write continuations as monads.

- Slightly easier to abstract monadic types—— have to include special application functions to use abstract CPS types.