

## Midterm Review

John Mitchell

## Topics

- ◆ JavaScript
- ◆ Block structure and activation records
- ◆ Exceptions, continuations
- ◆ Types and type inference
- ◆ Haskell
  - Functional programming
  - Type classes
  - Monads
    - IO Monad
    - General monads
- ◆ Operational semantics
- ◆ Modularity
- ◆ OO concepts
  - encapsulation
  - dynamic lookup
  - subtyping
  - inheritance
- ◆ Simula and Smalltalk
- ◆ C++
- ◆ Java
- ◆ Concurrency

## JavaScript Functions

- ◆ Anonymous functions make great callbacks
 

```
setTimeout(function() { alert("done"); }, 10000)
```
- ◆ Curried function
 

```
function CurriedAdd(x) { return function(y) { return x+y }; }
g = CurriedAdd(2);
g(3)
```
- ◆ Blocks w/scope can be expressed using *function*

```
var y=0;
(function () { // begin block
  var x=2; // local variable x
  y = y+x;
}) (); // end block
```

## Lambda Calculus

- ◆ Given function  $f$ , return function  $f \circ f$ 

$$\lambda f. \lambda x. f (f x)$$
- ◆ How does this work?
 
$$(\lambda f. \lambda x. f (f x)) (\lambda y. y+1)$$

$$= \lambda x. (\lambda y. y+1) ((\lambda y. y+1) x)$$

$$= \lambda x. (\lambda y. y+1) (x+1)$$

$$= \lambda x. (x+1)+1$$

In pure lambda calculus, same result if step 2 is altered.

## Basic object features

- ◆ Use a function to construct an object
 

```
function car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```
- ◆ Objects have prototypes, can be changed
 

```
var c = new car("Ford", "Taurus", 1988);
car.prototype.print = function () {
  return this.year + " " + this.make + " " + this.model;
}
c.print();
```

## Unusual features of JavaScript

- ◆ Some built-in functions
  - Eval, Run-time type checking functions, ...
- ◆ Regular expressions
  - Useful support of pattern matching
- ◆ Add, delete methods of an object dynamically
  - Seen examples adding methods. Do you like this? Disadvantages?
  - myobj.a = 5; myobj.b = 12; delete myobj.a;
- ◆ Redefine native functions and objects (incl undefined)
- ◆ Iterate over methods of an object
  - for (variable in object) { statements }
- ◆ With statement ("considered harmful" – why??)
  - with (object) { statements }

## Garbage collection

### ◆ Automatic reclamation of unused memory

- Navigator 2: per page memory management
  - Reclaim memory when browser changes page
- Navigator 3: reference counting
  - Each memory region has associated count
  - Count modified when pointers are changed
  - Reclaim memory when count reaches zero
- Navigator 4: mark-and-sweep, or equivalent
  - Garbage collector marks reachable memory
  - Sweep and reclaim unreachable memory

Reference [http://www.unix.org.ua/oreilly/web/js/script/ch11\\_07.html](http://www.unix.org.ua/oreilly/web/js/script/ch11_07.html)  
 Discuss garbage collection in connection with Lisp

## Language features in CS242

### ◆ Stack memory management

- Parameters, local variables in activation records

### ◆ Garbage collection

- Automatic reclamation of inaccessible memory

### ◆ Closures

- Function together with environment (global variables)

### ◆ Exceptions

- Jump to previously declared location, passing values

### ◆ Object features

- Dynamic lookup, Encapsulation, Subtyping, Inheritance

### ◆ Concurrency

- Do more than one task at a time (JavaScript is single-threaded)

## Block structure and storage mgmt

### ◆ Block-structured languages and stack storage

#### ◆ In-line Blocks

- activation records
- storage for local, global variables

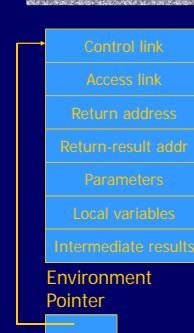
#### ◆ First-order functions

- parameter passing
- tail recursion and iteration

#### ◆ Higher-order functions

- deviations from stack discipline
- language expressiveness => implementation complexity

## Activation record for static scope



#### ◆ Control link

- Link to previous (calling) block

#### ◆ Access link

- Link to record of enclosing block

#### ◆ Return address

- Next instruction after return

#### ◆ Return-result address

- Place to put return value

#### ◆ Parameters

- Set when function called

#### ◆ Local variables

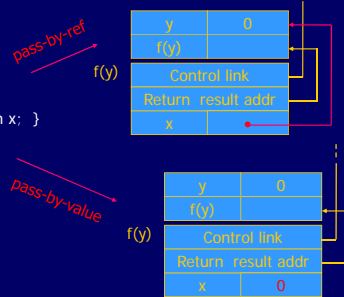
#### ◆ Intermediate results

## Example

pseudo-code

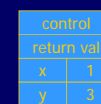
```
function f(x) =
  { x = x+1; return x; }
var y = 0;
print (f(y)+y);
```

activation records



## Tail recursion elimination

f(1,3)



f(2,3)



f(4,3)



```
fun f(x,y) = if x>y
  then x
  else f(2*x, y);
f(1,3);
```

### Optimization

- pop followed by push = reuse activation record in place

### Conclusion

- Tail recursive function equiv to iterative loop

## Complex nesting structure

```
function m(...) {
  var x=1;
  ...
  function n(...){
    function g(z) { return x+z; }
    ...
    { ...
      function f(y) {
        var x = y+1;
        return g(y*x); }
      ...
      f(3); ... }
    ... n( ... ) ... }
  ... m(...)
```

**Simplify to**

```
var x=1;
function g(z) { return x+z; }
function f(y) {
  { var x = y+1;
    return g(y*x); }
  f(3);
```

Simplified code has same block nesting, if we follow convention that each declaration begins a new block.

## Function Argument and Closures

Run-time stack with access links

```
{ var x = 4;
  { function f(y)(return x*y);
    { function g(h) {
      int x=7;
      return h(3)+x;
    };
    g(f);
  }})
```

access link set from closure

## Function Results and Closures

```
function mk_counter (init) {
  var count = init;
  function counter(inc) {count=count+inc; return count; }
  return counter; }
var c = mk_counter(1);
c(2) + c(2);
```

Code for mk\_counter

Code for counter

## Summary of scope issues

- ◆ Block-structured lang uses stack of activ records
  - Activation records contain parameters, local vars, ...
  - Also pointers to enclosing scope
- ◆ Several different parameter passing mechanisms
- ◆ Tail calls may be optimized
- ◆ Function parameters/results require closures
  - Closure environment pointer used on function call
  - Stack deallocation may fail if function returned from call
  - Closures *not* needed if functions not in nested blocks

## Exceptions: Structured Exit

- ◆ Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- ◆ Two main language constructs
  - Declaration to establish exception *handler*
  - Statement or expression to *raise* or *throw* exception

Often used for unusual or exceptional condition: other uses too

## JavaScript Exceptions

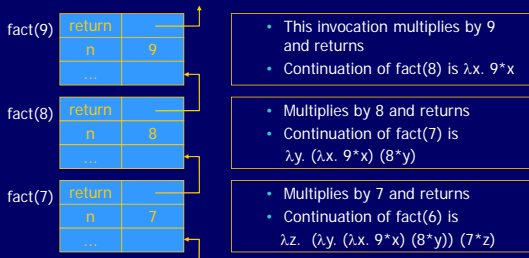
```
throw e //jump to catch, passing exception object as parameter
```

```
try { ... //code to try
} catch (e if e == ...) { ... //catch if first condition true
} catch (e if e == ...) { ... //catch if second condition true
} catch (e if e == ...) { ... //catch if third condition true
} catch (e) { ... // catch any exception
} finally { ... //code to execute after everything else
}
```

[http://developer.mozilla.org/En/Core\\_JavaScript\\_1.5\\_Guide/Exception\\_Handling\\_Statements](http://developer.mozilla.org/En/Core_JavaScript_1.5_Guide/Exception_Handling_Statements)

## Continuation view of factorial

fact(n) = if n=0 then 1 else n\*fact(n-1)



## Types and Type Inference

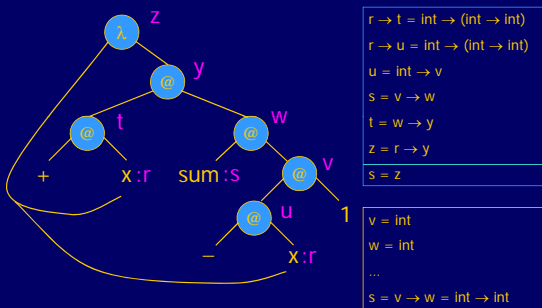
### ◆ Type safety

- Can use compile-time or run-time typing
- Cannot have dangling pointers

### ◆ Type inference

- Compute types from the way symbols are used
- Know how algorithm works (for simple examples)
- Know what an ML polymorphic type means
  - fun f(g,x) = g(g(x));
  - > val it = fn : (t -> t) \* t -> t
- Polymorphism different from overloading

fun sum(x) = x + sum(x-1);



## Haskell

### ◆ Haskell is a programming language that is

- Similar to ML: general-purpose, strongly typed, higher-order, supports type inference
- Different from ML: lazy evaluation, purely functional, evolving type system.

### ◆ Designed by committee in 80's and 90's

- Haskell 1.0 in 1990, Haskell '98, still ongoing
- "A history of Haskell: Being lazy with class" HOPL 3



Paul Hudak



John Hughes

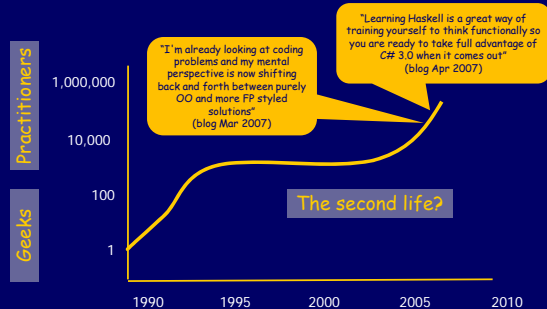


Simon Peyton Jones



Phil Wadler

## Haskell

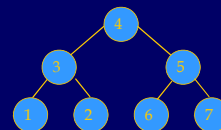


## Datatypes and Pattern Matching

### ◆ Recursively defined data structure

data Tree = Leaf Int | Node (Int, Tree, Tree)

Node(4, Node(3, Leaf 1, Leaf 2), Node(5, Leaf 6, Leaf 7))



### ◆ Recursive function

sum (Leaf n) = n  
 sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)



## Backus' Turing Award

- ◆ John Backus was designer of Fortran, BNF, etc.
- ◆ Turing Award in 1977
- ◆ Turing Award Lecture
  - Functional prog better than imperative programming
  - Easier to reason about functional programs
  - More efficient due to parallelism
  - Algebraic laws
    - Reason about programs
    - Optimizing compilers

## No-side-effects language test

Within the scope of specific declarations of  $x_1, x_2, \dots, x_n$ , all occurrences of an expression  $e$  containing only variables  $x_1, x_2, \dots, x_n$ , must have the same value.

### ◆ Example

```
begin
  integer x=3; integer y=4;
  5*(x+y)-3
  ... //? // no new declaration of x or y //
  4*(x+y)+1
end
```

## Polymorphism vs Overloading

### ◆ Parametric polymorphism

- Single algorithm: many types
- Type variable may be replaced by any type
  - if  $f:t \rightarrow t$  then  $f:\text{int} \rightarrow \text{int}$ ,  $f:\text{bool} \rightarrow \text{bool}$ , ...

### ◆ Overloading

- Single symbol: more than one algorithm
- Choice of algorithm determined by type context
- Types may be arbitrarily different
  - + has types  $\text{int} * \text{int} \rightarrow \text{int}$ ,  $\text{real} * \text{real} \rightarrow \text{real}$

## Type Classes

Works for any type 'n' that supports the Num operations

```
square :: Num n => n -> n
square x = x*x
```

The class declaration says what the Num operations are

```
class Num a where
  (+)  :: a -> a -> a
  (*)  :: a -> a -> a
  negate :: a -> a
  ..etc...
```

An instance declaration for a type T says how the Num operations are implemented on T's

```
instance Num Int where
  a + b = plusInt a b
  a * b = mulInt a b
  negate a = negInt a
  ..etc...
```

plusInt :: Int -> Int -> Int  
mulInt :: Int -> Int -> Int  
etc, defined as primitives

## Compiling Type Classes

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
class Num a where
  (+)  :: a -> a -> a
  (*)  :: a -> a -> a
  negate :: a -> a
  ..etc...
```

```
data Num a
  = MkNum (a->a->a)
  (a->a->a)
  (a->a)
  ..etc...
...
(*) :: Num a -> a -> a -> a
(*) (MkNum _ m _ ...) = m
```

The class decl translates to:

- A data type decl for Num
- A selector function for each class operation

A value of type (Num n) is a dictionary of the Num operations for type n

## Compiling Instance Declarations

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
instance Num Int where
  a + b = plusInt a b
  a * b = mulInt a b
  negate a = negInt a
  ..etc..
```

```
dNumInt :: Num Int
dNumInt = MkNum plusInt
          mulInt
          negInt
          ...
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

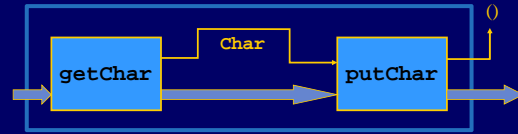
A value of type (Num n) is a dictionary of the Num operations for type n

## Before Monads

- ◆ Streams
  - Program issues a stream of requests to OS, which responds with a stream of inputs.
- ◆ Continuations
  - User supplies continuations to I/O routines to specify how to process results.
- ◆ World-Passing
  - The "World" is passed around and updated, like a normal data structure.
  - Not a serious contender because designers didn't know how to guarantee *single-threaded* access to the world.
- ◆ Stream and Continuation models inter-definable.
  - Haskell 1.0 Report adopted Stream model.

## The (>=>) Combinator

```
(>=>) :: IO a -> (a -> IO b) -> IO b
```



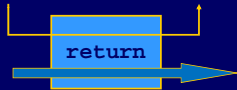
- ◆ Connect two actions to make another action

```
echo :: IO ()  
echo = getChar >=> putChar
```

## The **return** combinator

- ◆ The action (**return v**) does no IO and immediately returns **v**:

```
return :: a -> IO a
```



```
getTwoChars :: IO (Char,Char)  
getTwoChars = getChar >=> \c1 ->  
  getChar >=> \c2 ->  
  return (c1,c2)
```

## The "do" Notation

- ◆ The "do" notation adds syntactic sugar to make monadic code easier to read.

```
-- Plain Syntax  
getTwoChars :: IO (Char,Char)  
getTwoChars = getChar >=> \c1 ->  
  getChar >=> \c2 ->  
  return (c1,c2)
```

```
-- Do Notation  
getTwoCharsDo :: IO(Char,Char)  
getTwoCharsDo = do { c1 <- getChar ;  
  c2 <- getChar ;  
  return (c1,c2) }
```

- ◆ Do syntax designed to look imperative.

## Making Modifications...

- ◆ To add error checking
  - Purely: modify each recursive call to check for and handle errors.
  - Impurely: throw an exception, wrap with a handler.
- ◆ To add logging
  - Purely: modify each recursive call to thread logs.
  - Impurely: write to a file.
- ◆ To add a count of the number of operations
  - Purely: modify each recursive call to thread count.
  - Impurely: increment a global variable.

## The Monad Constructor Class

- ◆ The Prelude defines a type constructor class for monadic behavior:

```
class Monad m where  
  return :: a -> m a  
  (>=>) :: m a -> (a -> m b) -> m b
```

- ◆ The Prelude defines an instance of this class for the IO type constructor.
- ◆ The "do" notation works over any instance of class `Monad`.

## Hope monad defined in lecture

- ◆ We can make `Hope` an instance of `Monad`:

```
instance Monad Hope where
  return = Ok
  (>>=) = ifOKthen
```

- ◆ And then rewrite the evaluator to be monadic

```
eval3 :: Exp -> Hope Int
-- Cases for Plus and Minus omitted but similar
eval3 (Times e1 e2) = do {
  v1 <- eval3 e1;
  v2 <- eval3 e2;
  return (v1 * v2)
}
eval3 (Div e1 e2) = do {
  v1 <- eval3 e1;
  v2 <- eval3 e2;
  if v2 == 0 then Error "divby0" else return (v1 `div` v2)
}
eval3 (Const i) = return i
```

## Monad Menagerie

- ◆ We have seen many example monads
  - IO, Hope (aka Maybe), Trace, ST, Non-determinism
- ◆ There are many more...
  - Continuation monad
  - STM: software transactional memory
  - Reader: for reading values from an environment
  - Writer: for recording values (like Trace)
  - Parsers
  - Random data generators (e.g. in Quickcheck)
- ◆ Haskell provides many monads in its standard libraries, and users can write more.

## PL Fundamentals

- ◆ Grammars, parsing (skipped this year)
- ◆ Lambda calculus (quick overview only)
- ◆ Denotational semantics (skipped this year)
- ◆ Functional vs. Imperative Programming
  - Why don't we use functional programming?
  - Is implicit parallelism a good idea?
  - Is implicit *anything* a good idea?
- ◆ Operational semantics
  - One lecture overview, illustrating applications to JavaScript and web security

## Operational semantics

- ◆ Define meaning of a program
  - Sequence of actions of abstract machine
  - Aim for clarity and precision, not efficient implementation
- ◆ States generally corresponds to
  - Term (statement or expression) being evaluated
  - Abstract description of memory and other data structures involved in computation

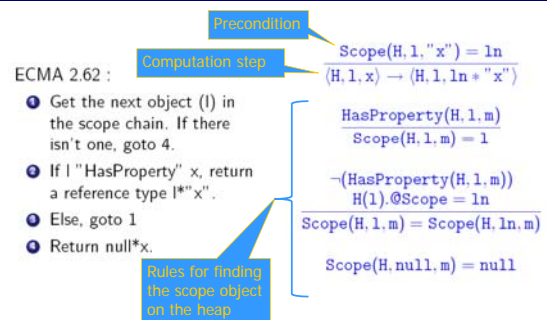
## JavaScript Ambiguities

- ◆ Example 1
 

```
var b = 10;
var f = function(){ var b = 5;
  function g(){ var b = 8; return this.b;};
  g();
  var result = f();
```
- ◆ Example 2
 

```
var f = function(){ var a = g();
  function g() { return 1;};
  function g() { return 2;};
  var g = function() { return 3;};
  return a;
}
var result = f();
```

## Small part of operational semantics



## Example application: BeamAuth

- Two factor authentication mechanism, proposed by Ben Adida.
- Consider visiting the login page of your bank (might be a phishing page!).
- A piece of *JavaScript* sits in one of your bookmarks which contains yours login information.
- When the bookmarklet is clicked, it verifies whether the source of the page is `www.bank.com` and then fills in the login info on your behalf.

### Problem

- Write a JavaScript program which can check whether a page belongs to the hostname `bank.com`.

## Attacks and defenses

### Attempt 1 - Seems Easy

```
If (window.location.host === "bank.com") return "good page" else return "bad page"
```

Attack : `var window = {location:{host: "bank.com"}}`  
Mitigation : Use `this` instead of `window`.

### Attempt 2 - I can fix it

```
If (this.location.host === "bank.com") return "good page" else return "bad page"
```

Attack : `window._defineGetter_("location", function () {return {host : "bank.com"}})`  
Mitigation: Use `...lookupGetter...`

and there are more ...

## Summary

- ◆ JavaScript
- ◆ Block structure and activation records
- ◆ Exceptions, continuations
- ◆ Types and type inference
- ◆ Haskell
  - Functional programming
  - Type classes
  - Monads
    - IO Monad
    - General monads
- ◆ Operational semantics
- ◆ Modularity
- ◆ OO concepts
  - encapsulation
  - dynamic lookup
  - subtyping
  - inheritance
- ◆ Simula and Smalltalk
- ◆ C++
- ◆ Java
- ◆ Concurrency