Functional Programming

I thought four rules would be enough, provided that I made a firm and constant resolution not to fail even once in the observance of them. -- René Descartes

Conjunction junction, what's your function? -- The Electric Company

- Declarative programming X.X
- Functions 14.2, 14.5
  - lambda expressions
  - functional forms
- Functional programming 14.3, 14.5
  - data structures
  - control structures
  - function definition
- Advanced topics 14.5, 14.8
  - side effects, function evaluation, code generation, list comprehensions

345

Declarative Programming

Let's do some calesthenics (stretching some muscles we don't normally use):

- *algorithm*
  - alogrithm = logic + control
    - the logic is the definition of the algorithm
    - the control is the execution of steps in computing the algorithm
- *procedural programming*
  - specify the logic *and* the control
- *declarative programming*
  - specify the logic, leave the control up to the language itself
    - *logic programming*: control is the resolution prover
    - *functional programming*: control is the expression evaluator

346

Functions

A *function* maps members of one set (the *domain*) to members of another set (the *range*).

- the *definition* of a function specifies the mappings from all elements of the domain to their maps in the range
- the *application* of a function to a single member of the domain returns the value of its map in the range
  - the *evaluation* of a function and a domain member is the value of the range member mapped to by the function on the domain member

- *definition* of square: $f(x) = x^2$
- *application* of f to 5 returns 25
- *evaluation* of $f(5)$ is 25

$\}$ *different ways to say the same thing*

347

Lambda Expressions

There is no particular reason why a function needs a name:

- *square* is the function that maps a number x to the number x^2
- "the function that maps a number x to the number x^2"

We usually use the symbol λ to write functions with no name:

- $\lambda x.x^2$
- $\lambda x.3x^2+1$
- $\lambda xy.x \times y$

We can apply *lambda expressions* to domain members:

- $\lambda x.x^2(5) = 25$
- $\lambda x.3x^2+1(3) = 28$
- $\lambda xy.x \times y(4,2) = 8$

348

Functional Forms

In functional programming (as in mathematics), functions are *first class objects*:

- a function is an element of the set of functions
- a function can be in the domain of another function
- a function can be in the range of another function

What does this mean?

- we can *compose* functions
  - apply a function to another function
- we can *return* functions
  - a function may evaluate to another function

Examples of Functional Forms

A function that can be applied to other functions or that returns a function is sometimes called a *functional form* or *second-order function*.

Composition:

- $f(x) = 3x$
  $g(x) = x^2$
  $h(x) = f \circ g = f(g(x)) = 3x^2$

Returning functions:

- $d(f) = g$
  $f(x) = x^3$
  $g(x) = 3x^2$

Functional Programming

- *functional programming*
  - a style of programming emphasizing the evaluation of expressions rather than the execution of commands
- *functional programming language*
  - a language that promotes functional-style programming

The textbook (§ 15.4) seems to suggest that *functional* programming is more *procedural* than *declarative*; but usually *functional* programming is considered *declarative.*

-
  ```
  var b, i, s: byte
      a: array[1..10] of byte;
  begin
  ...
  s := 0;
  for i := 1 to 10 do
    s := s + a[i];
  ...
  ```

-
  ```
  (define (addl L)
    (if (null? L)
      0
      (+ (car L)
         (addl (cdr L)))
    )
  )
  ```

Functional Programming Style

We defined functional programming as a style of programming. It is important to note that it is possible to program in a functional style in procedural languages, and often to program in a procedural style in functional languages:

-
  ```
  function sum(lo, hi: byte): byte;
  begin
    if lo = hi then
      sum := hi
    else
      sum := lo + sum(lo+1, hi)
  end;
  ```

-
  ```
  (define (sum lo hi)
    (do ( (x lo (+ x 1))
          (s 0 (+ s x)) )
        ((> x hi) s)
    )
  )
  ```

bad Pascal (inefficient)

bad Scheme (side-effecty)

Data Structures In Functional Programming

- atoms
  - symbolic constants
  - numeric constants
  - sometimes character constants and string constants
  - sometimes truth values (`false` and `true`)
- lists
  - heterogeneous dynamic length sequences of data
  - sometimes the empty list might be considered an atom
- functions
  - functions?
  - you heard me... functions!
  - remember, functions are first-class objects
    - they can be passed as arguments to other functions
    - they can be returned from functions

353

Data Structures *Not* in Functional Programming

- records
  - don’t need them, we’ve got lists
- arrays
  - don’t need them, we’ve got lists
- variables
  - no variables?

354

No Variables

Pure functional programming languages do not have variables in the sense of imperative language variables. There are variables like you’d find in Math (or Prolog):

- a variable is an abstraction of elements of a set
- values are not set by the programmer
- values are set as a result of applying a function to a domain element
  - sort of like read-only pass-by-value formal parameters
- once a variable is bound to a value, it is fixed to that value for its entire lifetime and everywhere in its scope

Control Structures In Functional Programming

- function application
  - apply a function to some data
- function composition
  - apply a function to a function
- expression evaluation
  - if the expression is an atom or a list constant, get its value
  - if the expression is a function application, get its return value
- conditional evaluation
  - evaluate an expression only if some other expression evaluates to true

Control Structures Not in Functional Programming

- sequence
  - don’t need it, we’ve got function composition
- selection
  - don’t need it, we’ve got conditional evaluation
- iteration
  - don’t need it, we’ve got function application

Why don’t we need iteration if we’ve got function application?

That’s right! function application gives us recursion!

357

Functional Programming Semantics (an Aside)

We said that in purely functional languages the programmer cannot set the value of a variable:

- variables are bound to values as a result of function application

Because of this nice little fact, function side-effects are not possible, guaranteeing *referential transparency*:

- function f maps domain element x to range element y always!

This greatly simplifies the semantics of programs. The simple semantics along with the notion that programs can be manipulated as data allow us to make demonstrably provable statements about the meaning of functional programs.

358

Lisp

Lisp was the first functional language (and one of the first programming languages, invented around the same time as Fortran and Cobol).

- a program is a collection of function definitions
- programs are run (as in Prolog) by a user submitting expressions (function applications) to the Lisp interpreter
- data (lists):
  - `(a b c)`
  - a list of three atoms
- functions:
  - `(f a b)`
  - the function f applied to two arguments

Using the same notation for both data and programs seems confusing at first, but it is a vital part of the power of Lisp.

Lisp Quirks

- Lisp's top-level expression evaluator (the "interpreter") assumes that `(a b c)` is a function application; it tries to evaluate a as a function
  - to force Lisp to consider `(a b c)` a list, it must be quoted: `'(a b c)`
- Lisp has variables that can be set by the programmer
- Lisp variables have a strange kind of *dynamic scoping*
  - indefinite scope
    - local bindings (variable names defined locally) are visible everywhere
  - dynamic lifetime
    - local bindings exist as long as the construct in which they are defined is active
  - shadowing
    - a local binding within a construct called later in a call chain will shadow (hide) a local binding earlier in the call chain

Lisp Quirks in Action

- to define a function:
  - `(defun` *functionname* `(`*parameters...*`)` *functionbody*`)`

```
(defun foo (x)
  (bar 5)
)
(defun bar (y)
  (printit)
)
(defun printit ()
  (print x)
)
```

```
(defun foo (x)
  (bar 5)
)
(defun bar (x)
  (printit)
)
(defun printit ()
  (print x)
)
```

```
lisp> (foo 13)

13

lisp>
```

```
lisp> (foo 13)

5

lisp>
```

It Gets Worse

- to define a variable and initialize it:
  - `(setq` *variablename* *expression*`)`
- to set up a construct with local bindings explicitly:
  - `(let (`*variabledeclarations...*`)` *expressions...*`)`

```
(setq x 5)

(let ((x 13) (y x))
  (print x)
  (print y)
)
```

```
lisp>

13 5

lisp>
```

- the local bindings set up in *variabledeclarations* don't become active until the body of the `let`
  - the local x binding is not active when y is initialized, so it uses the x binding established by the `setq`

Scheme: A Lisp Dialect

Scheme is a small dialect of Lisp:

- uses essentially the same syntax as Lisp
- programs are interpreted by a top-level expression evaluator as in Lisp
- explicit variable bindings are allowed, making Scheme an *impure* functional language (like Lisp)

But Scheme has some important differences from Lisp:

- lexical scoping
- first-class functions
- proper tail recursion

We'll come back to these soon, but first let's look at Scheme!

Defining Bindings

To bind a value to a name in Scheme:

- `(define` *name dataobject*`)`
- `(define x 1)`
  `(define e 2.71828)`

Since functions are first-class objects in Scheme we can bind *them* to names as well:

- `(define square (lambda (x) (* x x)))`
- `(define (square x) (* x x))`

Lists in Scheme

A list in Scheme is a fully dynamic length collection of expressions:

- (*expr*$_1$ *expr*$_2$ ... *expr*$_N$)

As in Lisp, the top-level expression evaluator attempts to evaluate a list as a function application:

- evaluate *expr*$_N$ giving *val*$_N$ (a data object)
  evaluate *expr*$_{N-1}$ giving *val*$_{N-1}$ (a data object)
  ...
  evaluate *expr*$_2$ giving *val*$_2$ (a data object)
  evaluate *expr*$_1$ giving *val*$_1$ (a function)
  apply function *val*$_1$ to parameters *val*$_2$..*val*$_N$

As in Lisp, evaluation may be suppressed by quoting:

- `(quote (a b c))` or `’(a b c)`

365

List Construction and Access

A list is constructed of a head (*h*) and a tail (*t*):

- `(cons` *h* *t*`)`
- *h* is an expression
- *t* is a list
- `()` is the empty list

There are built-in functions for accessing elements of lists:

- `(car (cons` *h* *t*`))` ≡ *h*
- `(cdr (cons` *h* *t*`))` ≡ *t*

And others too:

- `(caar L) ≡ a`
- `(cadr L) ≡ d`
- `(cdar L) ≡ (b c)`
- `(cadar L) ≡` b
- `(caddr L) ≡` e
- `(cddaddddr L) ≡` (i j)

this one may not be built-in

Let’s say `L` is `((a b c) d e f (g h i j) k l)`

366

Conditional Evaluation

Normally all expressions everywhere are evaluated. Conditional evaluation allows some choice in which expressions are evaluated. Consider the mathematical function *factorial*:

- $factorial(n) = \begin{cases} 1 \text{ if } n = 0 \\ n \times factorial(n-1) \text{ if } n > 0 \end{cases}$

Scheme has two conditional evaluation structures:

- `(if` *condexpr* *expr*$_t$ *expr*$_f$`)`
- `(cond (`*exprc*$_1$ *expr...*`)`
  `(`*exprc*$_2$ *expr...*`)`
  ...
  `(else` *expr...*`)`
  `)`

Some Scheme Examples

```
(define (member? e L)
  (cond
    ((null? L) #f)
    ((equal? e (car L)) #t)
    (else (member? e (cdr L)))
  )
)
```

```
]=> (member? 1 '(3 1 2 5))
;Value: #t
]=> (member? 4 '(3 1 2 5))
;Value: ()
]=> (member? '(c d) '(b (c d)))
;Value: #t
]=>
```

```
(define (repeats? L)
  (cond
    ((null? L) #f)
    ((null? (cdr L)) #f)
    ((equal? (car L) (cadr L)) #t)
    (else (repeats? (cdr L)))
  )
)
```

```
]=> (repeats? '(3 1 2 5))
;Value: #f
]=> (repeats? '(2 1 1 5))
;Value: ()
]=> (repeats? '((c d)(c d)))
;Value: #t
]=>
```

More Scheme Examples

```
(define (empty? S)
  (null? S)
)

(define (push i S)
  (cons i S)
)

(define (top S)
  (if (empty? S)
    '()
    (car S)
  )
)

(define (pop S)
  (if (empty? S)
    '()
    (cdr S)
  )
)
```

```
]=> (push 1 '())

;Value: (1)

]=> (push 2 (push 1 '()))

;Value: (2 1)

]=> (push 3 (push 2 (push 1 '())))

;Value: (3 2 1)

]=> (top (pop (push 3 (push 2 (push 1 '())))))

;Value: 2

]=> (empty? (pop (pop (push 2 (push 1 '())))))

;Value: #t

]=>
```

Global Data and Local Data

Although scoping in Scheme is static, global names are still possible. And like Lisp, local data can be declared using a `let` form:

-
  ```
  ]=> (define x 5)
  ;Value: x
  ]=> x
  ;Value: 5
  ]=> (let ((x 13) (y x)) y)
  ;Value: 5
  ]=> (let ((x 13) (y x)) x)
  ;Value: 13
  ]=>
  ```

Local Functions...

Since functions are first-class objects, they can be local data as well. First, let’s check out a standard version of *reverse*:

```
(define (toend e L)
  (cond
    ((null? L) (cons e ’()))
    (else (cons (car L) (toend e (cdr L))))
  )
)

(define (rever L)
  (cond
    ((null? L) L)
    (else (toend (car L) (rever (cdr L))))
  )
)
```

The function `rever` reverses a list as we’d expect.

Local Functions Continued

If `rever` is the only place we need access to the `toend` function, why not define it locally in `rever`:

```
(define (rever2 L)
  (letrec ((toend (lambda (e L)
                (cond
                  ((null? L) (cons e ’()))
                  (else (cons (car L) (toend e (cdr L))))
                )
              )
            ))
    (cond
      ((null? L) L)
      (else (toend (car L) (rever2 (cdr L))))
    )
  )
)
```

Why in the name of Sam Hill would we want to do this?

Lexical (Static) Scoping

We said we were going to look at those three elements of Scheme that are different from Lisp:

- lexical scoping
- first-class functions
- proper tail recursion

The following example shows Scheme's lexical scoping:

```
(define (foo x)
  (bar 5)
)

(define (bar x)
  (whichx)
)

(define (whichx)
  x
)
```

```
]=> (define x 13)

;Value: x

]=> x

;Value: 13

]=> (foo 10)

;Value: 13
```

The "Fun" in "Functional Programming"

In Scheme, functions are first-class objects (*i.e.*, they can be manipulated as data:

- functions can be arguments to functions
- functions can be returned by functions

```
(define (square x)
  (* x x)
)

(define (threef f)
  (lambda (x) (* 3 (f x)))
)
```

```
]=> (square 4)
;Value: 16

]=> ((threef square) 4)
;Value: 48

]=>
```

Now come on! You've got to admit that's pretty cool.

Tail Recursion

tail recursion:

- if the last thing done in the body of a recursive function is the recursive call, then there is no need to return from the call

Here's and example of an obviously tail recursive function:

```
(define (fiboseq n)
  (tailfibo 2 n ’(1 1))
)
(define (tailfibo x n f)
  (if (= x n)
    f
    (tailfibo (+ x 1) n (cons (+ (car f) (cadr f)) f)
    )
  )
)
```

```
]=> (fiboseq 10)
;Value: (55 34 21 13 8 5 3 2 1 1)

]=>
```

375

Proper Tail Recursion

Proper tail recursion:

- repetition can be accomplished in constant space, even if the repetition is specified using recursion
- this is accomplished by converting recursive definitions to tail recursive definitions

Let's have another look at the standard factorial program:

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))
  )
)
```

Is this function tail recursive and why not?

After the recursive call we must multiply the return value by n!

376

Converting Non-Tail Recursive Recursion...

Here's one way to convert factorial to be tail recursive:

```
(define (factorial n)
  (tailfact n 1)
)
(define (tailfact n sofar)
  (if (= n 0)
      sofar
      (tailfact (- n 1) (* sofar n))
  )
)
```

The real trick, though is to figure out how to do this to *any* non-tail recursive function...

377

Still Converting Non-Tail Recursive Recursion

Consider the general recursive function:

```
(define (recfun n)
  (if (base? n)
      (basefun n)
      (fun n (recfun (simplify n)))
  )
)
```

```
(define (fac n)
  (if (eqzero? n)
    (plus1 n)
    (* n (fac (minus1 n)))
  )
)
```

The tail recursive version looks like this:

```
(define (recfun n)
  (tailfun n (lambda (x) x)))
(define (tailfun n do_it)
  (if (base? n)
      (do_it (basefun n))
      (tailfun (simplify n) (lambda (y) (fun (do_it y))))
  )
)
```

This one is not on the exam, ok!

378

Function Evaluation

Running a program in a functional programming language consists of submitting expressions to a top-level expression evaluator

- when you sit at the Scheme or Lisp prompt, you submit expressions to this evaluator
- the evaluator is also responsible for evaluating all the expressions within functions called as a result of evaluating your expression
- the evaluator is always running, recursively evaluating
- the evaluator is good
- you can use the evaluator, if you want
  - that's right! Scheme and Lisp allow you to call the evaluator explicitly from within your functions

379

Calling the Top-Level Expression Evaluator

But why on Earth would you want to call the evaluator explicitly? Have a look at this pretty useless Scheme function:

```
(define (useless f x y)
  (eval (f x y))
)
```

```
]=> (useless + 5 7)
;Value: 12

]=>
```

Here is a slightly less useless application:

```
(define (f2L f L)
  (if (null? L)
      L
      (eval (cons (f L)))
  )
)
```

```
]=> (f2L factorial '(6))
;Value: 720

]=> (f2L + '(5 7))
;Value: 12

]=> (f2L + '(5 7 9 11))
;Value: 32

]=>
```

380

Still not convinced?

```
(define (genodds)
  (gosub 1)
)
(define (gosub n)
  (cons n (cons gosub (list (+ n 2))))
)

(define (infmemb? e L)
  (cond
    ((= e (car L)) #t)
    ((< e (car L)) #f)
    (else (infmemb? e (eval (cdr L))))
  )
)

(define (genfibo)
  (gfsub 1 1)
)
(define (gfsub n1 n2)
  (cons n1
       (cons gfsub
             (cons n2
                   (list (+ n1 n2)))))
)
```

```
]=> (genodds)
;Value: (1 #[proc gosub] 3)

]=> ((cadr (genodds)) (caddr (genodds)))
;Value: (3 #[proc gosub] 5)

]=> (infmemb? 1 (genodds))
;Value: #t

]=> (infmemb? 5 (genodds))
;Value: #t

]=> (infmemb? 3125 (genodds))
;Value: #t

]=> (infmemb? 3124 (genodds))
;Value: ()

]=> (genfibo)
;Value: (1 #[proc gfsub] 1 2)

]=> (infmemb? 13 (genfibo))
;Value: #t

]=> (infmemb? 6765 (genfibo))
;Value: #t
]=>
```

381

Lazy Evaluation

In the previous example, we were able to simulate infinite sets. Don’t generate all the elements in the infinite set (impossible!). Instead, define the set as

- a minimal number of initial elements
- a function that knows how to calculate the rest of the elements

Crucially, we do not evaluate the function until necessary. When we do evaluate the function, it gives us:

- the next element in the set
- another function for calculating the rest of the elements

The technique of delaying evaluation of arguments as long as possible is called *lazy evaluation*.

382

Haskell

Scheme and Lisp have *greedy evaluation*:

- for a function application ($f\ arg_1\ arg_2 \ldots arg_n$)
  - evaluate all arguments $arg_1 \ldots arg_n$ to $val_1 \ldots val_n$
  - evaluate f to val_f
  - apply val_f to $val_1 \ldots val_n$

Haskell is a functional language with *lazy evaluation*:

- for a function application ($f\ arg_1\ arg_2 \ldots arg_n$)
  - apply f to $arg_1 \ldots arg_n$ without evaluating them
  - arguments will be evaluated inside f if necessary

383

Basic Haskell

Haskell is a statically scoped functional language like Scheme with some major differences

- lazy evaluation
- no side effects (*pure* functional language)
- strong typing
- type inferencing

We'll look at Haskell's strong typing and type inferencing in just a moment. But first, some examples...

384

Haskell Examples

-
  ```
  factorial 0 = 1
  factorial n = n * factorial (n - 1)
  ```

-
  ```
  factorial n  |  n == 0   = 1
               |  n > 0    = n * factorial (n - 1)
  ```

-
  ```
  addem []     = 0
  addem (x:xs) = x + addem xs
  ```

-
  ```
  oddem n = addem [1, 3..n]
  ```

-
  ```
  fibo n  |  n == 0   = 1
          |  n == 1   = 1
          |  n > 1    = fibo (n - 1) + fibo (n - 2)
  ```

-
  ```
  fiboseq n = [fibo m  |  m <- [0..n]]
  ```

-
  ```
  qsort []     = []
  qsort (x:xs) = qsort smalls ++ [x] ++ qsort bigs
                   where smalls = [y  |  y <- xs, y < x]
                         bigs   = [y  |  y <- xs, y >= x]
  ```

Where Are All the Types?

We claimed that Haskell was strongly typed, yet those examples have no types in them at all. What gives?

- Haskell has the normal types
  - `Integer` (an infinite precision integer)
  - `Char` (a character)
  - `Integer -> Integer` (a function mapping integers to integers)
  - `[Integer]` (a homogenous list of integers)
  - `(Integer, Char)` (a pair)
  - (also, rationals, floating points, booleans...)
  - (also user-defined types, type synonyms)
- all type errors are detected at compile time
- the type system is powerful enough that the programmer (almost) never has to declare types

What’s Your Type?

Look at this little Pascal program fragment:

-
  ```
  program p
    var x, y: integer;
  begin
    ...
    z := chr(x + y);
    ...
  ```
- in fact, given the single visible statement, we know that x and y must be integers (and Pascal knows it too, since it would give us a compile time type error if x and y were declared as any other type!)

Why does Pascal force us to declare `x` *and* `y`*?*

That’s right! Because Pascal is a drag!

387

Type Inferencing

There is no reason why Pascal couldn’t just *infer* the type of those variables from their use in the program.

- *type inferencing*
  - determining automatically the type of an expression from the context in which it is used

Haskell has

- strong typing
  - all type errors can be detected at compile time
- polymorphic types
  - expressions that can consist of any type

  -
    ```
    length         :: [a] -> Integer
    length []      =  0
    length (x:xs)  =  1 + length xs
    ```
- powerful type inferencing

388

Haskell and Lazy Evaluation

We saw how to simulate infinite sets in Scheme by preventing the top-level evaluator from evaluating expressions until we wanted it to. Since Haskell already has lazy evaluation, infinite sets are natural:

-
  ```
  tenodds  = [1, 3..19]
  allodds  = [1, 3..]
  allfibos = [fibo n | n <- [0..]]

  infmember e h:t  |  e < h       = False
                   |  e == h      = True
                   |  otherwise   = infmember e t
  ```

389

ML

ML is a functional language a lot like Haskell in many ways:

- static scoping
- strong typing
- polymorphic types
- type inferencing

But there are some important differences as well:

- “eager evaluation”
- side effects

390

ML Types

- atomic types
  - `int`, `real`, `bool`, `string`, `unit`
- complex types
  - tuples

`(1, 2, 3)`	`int*int*int`
`(1, ("foo", 0), true, 2)`	`int*(string*int)*bool*int`

  - lists

`1::2::3::[]`	`int list`
`[1, 2, 3]`	`int list`
`["yeah", "baby"]`	`string list`
`[]`	`'a list`

  - records

`{name="Gern", age=42, fertile=false}`	`{name:string, age:int, fertile:bool}`

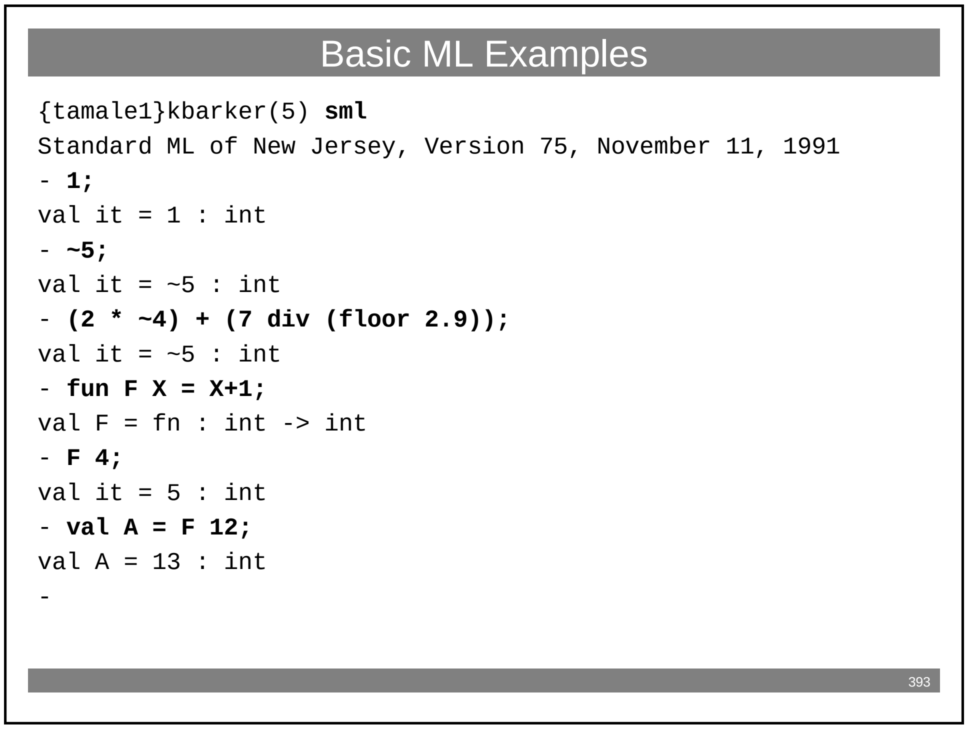
391

ML Functions

Functions in ML are first-class objects

- all functions take *one* argument and return a value
  - don't panic!
  - the single argument could be a tuple
- the type of a function:
  - *argument_type* `->` *return_type*
  - `(int * real) -> real`
  - `('a * 'a list) -> 'a list` (cons!)
  - `(('a -> 'b) * ('c -> 'a)) -> ('c -> 'b)`
- lambda expressions
  - functions with no name
  - `fn` *x* `=>` *expr*

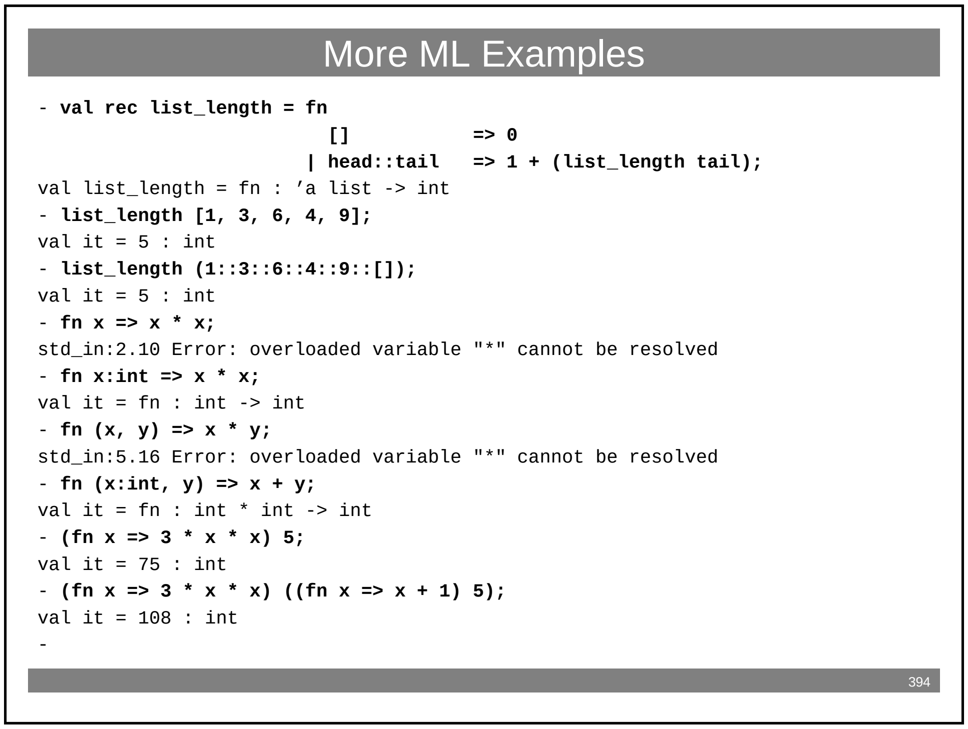
392



Basic ML Examples

```
{tamale1}kbarker(5) sml
Standard ML of New Jersey, Version 75, November 11, 1991
- 1;
val it = 1 : int
- ~5;
val it = ~5 : int
- (2 * ~4) + (7 div (floor 2.9));
val it = ~5 : int
- fun F X = X+1;
val F = fn : int -> int
- F 4;
val it = 5 : int
- val A = F 12;
val A = 13 : int
-
```

393



More ML Examples

```
- val rec list_length = fn
                          []           => 0
                        | head::tail   => 1 + (list_length tail);
val list_length = fn : 'a list -> int
- list_length [1, 3, 6, 4, 9];
val it = 5 : int
- list_length (1::3::6::4::9::[]);
val it = 5 : int
- fn x => x * x;
std_in:2.10 Error: overloaded variable "*" cannot be resolved
- fn x:int => x * x;
val it = fn : int -> int
- fn (x, y) => x * y;
std_in:5.16 Error: overloaded variable "*" cannot be resolved
- fn (x:int, y) => x + y;
val it = fn : int * int -> int
- (fn x => 3 * x * x) 5;
val it = 75 : int
- (fn x => 3 * x * x) ((fn x => x + 1) 5);
val it = 108 : int
-
```

394

Type Inferencing in ML

As you can see from the examples, ML can *infer* most of the types in a program.

- for each expression submitted, ML reports the types involved

  -
    ```
    - val rec list_length =
                fn
                    []           => 0
                  | head::tail   => 1 + (list_length tail);
    val list_length = fn : 'a list -> int
    ```

    list_length's type

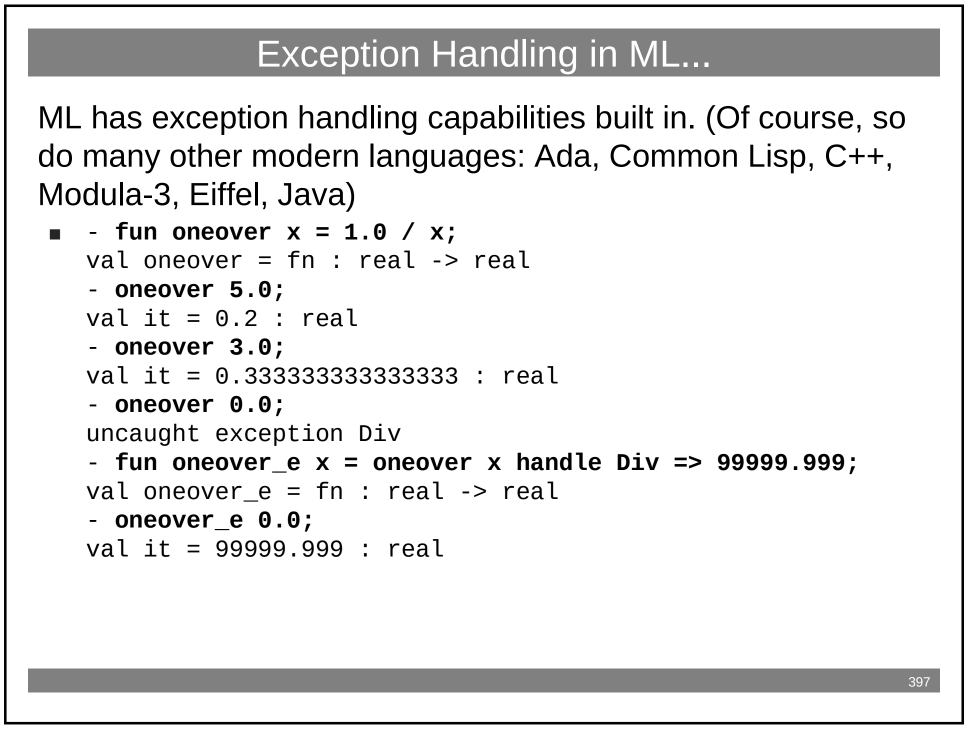
395

Exception Handling

- *exception*
  - an unusual event
  - possibly an error, possibly just some unlikely event
  - requires special processing
- *exception handling*
  - the special processing to be done when an exception is raised
- *exception handler*
  - the program unit that does the special processing when an exception is raised

- examples
  - divide by zero
  - dereferincing the null pointer
  - end of file

396

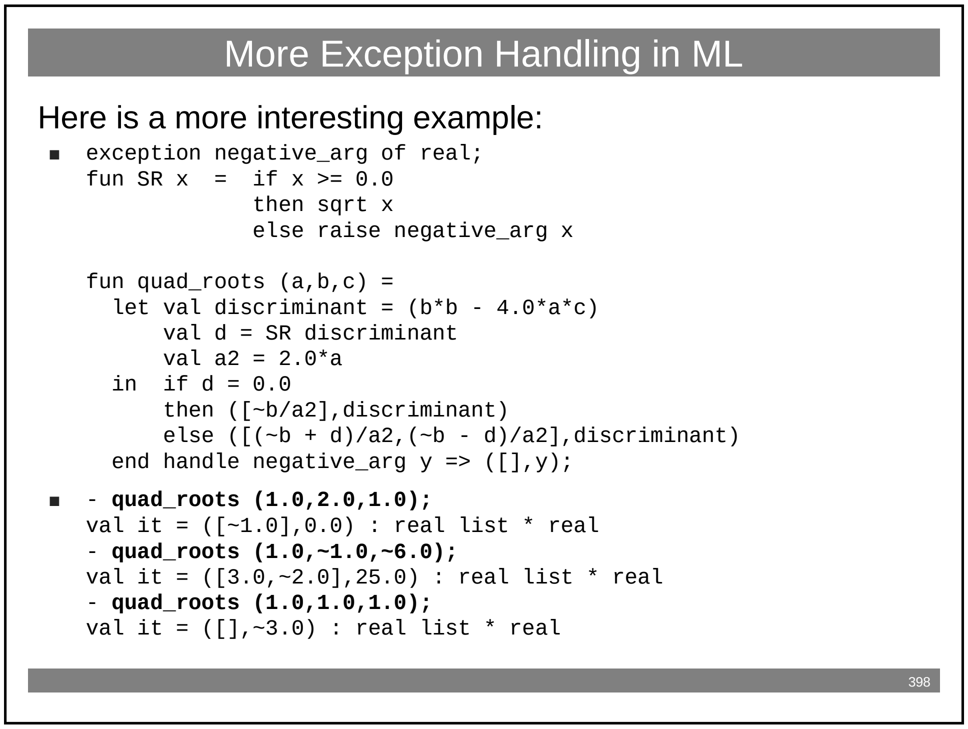


Exception Handling in ML...

ML has exception handling capabilities built in. (Of course, so do many other modern languages: Ada, Common Lisp, C++, Modula-3, Eiffel, Java)

```
- fun oneover x = 1.0 / x;
val oneover = fn : real -> real
- oneover 5.0;
val it = 0.2 : real
- oneover 3.0;
val it = 0.333333333333333 : real
- oneover 0.0;
uncaught exception Div
- fun oneover_e x = oneover x handle Div => 99999.999;
val oneover_e = fn : real -> real
- oneover_e 0.0;
val it = 99999.999 : real
```

397



More Exception Handling in ML

Here is a more interesting example:

```
exception negative_arg of real;
fun SR x  =  if x >= 0.0
             then sqrt x
             else raise negative_arg x

fun quad_roots (a,b,c) =
  let val discriminant = (b*b - 4.0*a*c)
      val d = SR discriminant
      val a2 = 2.0*a
  in  if d = 0.0
      then ([~b/a2],discriminant)
      else ([(~b + d)/a2,(~b - d)/a2],discriminant)
  end handle negative_arg y => ([],y);
```

```
- quad_roots (1.0,2.0,1.0);
val it = ([~1.0],0.0) : real list * real
- quad_roots (1.0,~1.0,~6.0);
val it = ([3.0,~2.0],25.0) : real list * real
- quad_roots (1.0,1.0,1.0);
val it = ([],~3.0) : real list * real
```

398

Abstraction. Again.

The main theme of this course has been abstraction:

4 An entity's name is an *abstraction*:
- ▲ it allows us to use the entity without regard for internal complexities

5 A data type is an *abstraction*:
- ▲ of a set of values and the allowable operations on those values

6 An expression is an *abstraction*:
- ▲ of the operations required to *evaluate* the expression

7 Control structures are *abstractions*:
- ▲ of the branching constructions required to control program flow

8 … 11 Data abstraction, procedural abstraction, encapsulation, polymorphism, first-class functions
- ▲ abstraction, abstraction, abstraction!

399

Mercury

Mercury is a logic-based/functional language (mostly logic)

- invented to address certain weaknesses of Prolog:
  - side effects
  - efficiency
- solutions:
  - no side effects
  - compiles to C code
- strong typing
  - + type inferencing like ML and Haskell
- strong mode system
  - arguments to predicates specified as `in` or `out`
- strong determinism
- real encapsulation
- first class functions, closure, currying, lambda expressions

400

Basic Mercury

Mercury looks like what Prolog would look like if it had types and modes specified:

- Prolog

```
append([], L2, L2).
append([H | T], L2, [H | L3]) :-
   append(T, L2, L3).
```

- Mercury

```
:- type list(Typ) ---> [];  [Typ | list(Typ)].
:- pred append(list(Typ)::in, list(Typ)::in, list(Typ)::out).

append([], L2, L2).
append([H | T], L2, [H | L3]) :-
   append(T, L2, L3).
```

Modes in Mercury

Programmer must declare what the instantiation state of parameters will be when a predicate is called. For the *factorial* predicate, the first parameter will be instantiated when the predicate is called, but the second one won't be

- `:- pred factorial(int::in, int::out).`

One of the great things about Prolog, though, is that predicates can have several modes. If a Mercury predicate has more than one pattern of modes, they must all be specified.

```
:- pred append(list(Typ), list(Typ), list(Typ)).
:- mode append(in, in, out).
:- mode append(out, out, in).
```

Mercury infers which one is appropriate at compile time (mode inference).

Determinism

Another strength of Logic programming is that predicates can be *nondeterministic*. Recall the classic Prolog example:

-
  ```
  ?- append([a,b,c], [d,e], L3).
  L3 = [a,b,c,d,e]
  ?- append(L1, L2, [a,b,c,d,e]).
  L1 = []
  L2 = [a,b,c,d,e] ;
  L1 = [a]
  L2 = [b,c,d,e] ;
  L1 = [a,b]
  L2 = [c,d,e] ;
  L1 = [a,b,c]
  L2 = [d,e] ;
  L1 = [a,b,c,d]
  L2 = [e] ;
  L1 = [a,b,c,d,e]
  L2 = [] ;
  no
  ?-
  ```

403

Mercury’s Strong Determinism System

A predicate is *deterministic* if it always succeeds exactly once. Mercury allows the programmer to specify determinism characteristics of predicates

- `det`
  - predicate will always succeed exactly once
- `semidet`
  - predicate will succeed at most once
- `multi`
  - predicate will succeed at least once
- `nondet`
  - predicate will succeed 0 or more times

404

Explicit Determinism Specifiers

Here is `append` in Mercury with determinism specifiers.

-
  ```
  :- pred append(list(Typ), list(Typ), list(Typ)).
  :- mode append(in, in, out) is det.
  :- mode append(out, out, in) is multi.
  ```

An example of a completely nondeterministic predicate would be the `member` predicate.

-
  ```
  :- pred member(Typ, list(Typ)).
  :- mode member(in, in) is nondet.

  member(E, [E | _T]).
  member(E, [_H | T]) :-
    member(E, T).
  ```

405

What's the Point of Strong Determinism?

- One of the biggest sources of bugs in Prolog programs is nondeterministic rules that the programmer thinks are deterministic

  -
    ```
    fac(0, 1).
    fac(N, F) :-
      N1 is N - 1,
      fac(N1, F1),
      F is N * F1.
    ```

Where is the bug in this program?

We want it to be deterministic, but it's not!

  -
    ```
    ?- fac(5, F).
    F = 120 ;

    ...
    ```

406

What Strong Determinism Allows

The Mercury compiler takes the programmer's determinism specifications and attempts to *prove* them.

- if the compiler is unable to prove the programmer's specifications, it rejects the program
- the Mercury compiler would reject the following definition of factorial (it would generate compile errors).
  - ▲
    ```
    :- pred fac(int::in, int::out) is det.
    fac(0, 1).
    fac(N, F) :-
      N1 is N - 1,
      fac(N1, F1),
      F is N * F1.
    ```
- in the general case, the proof is undecidable, but Mercury has some pretty good tricks

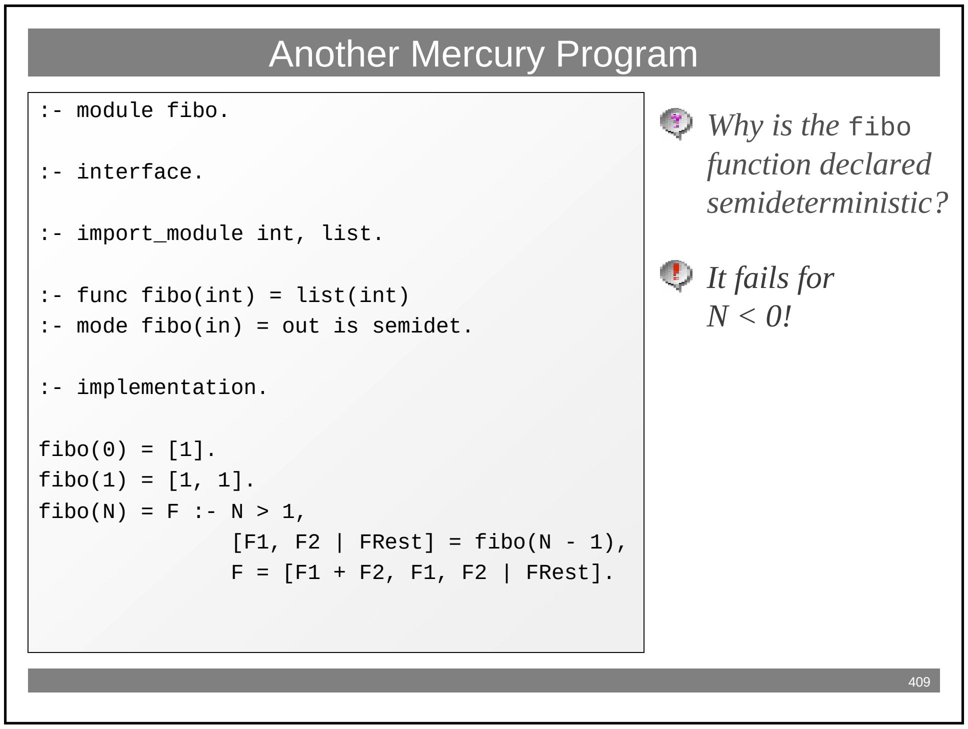
A Mercury Program

```
:- module stack.                           :- implementation.

:- interface.                              :- import_module list.
                                           :- type stack == list(Typ).
:- type stack.
                                           newstack([]).
:- pred newstack(stack::in) is semidet.
                                           pop([_H | T], T).
:- pred pop(stack, stack).
:- mode pop(in, out) is semidet.           push(E, S, [E | S]).

:- pred push(Typ, stack, stack).
:- mode push(in, in, out) is det.
:- mode push(out, out, in) is semidet.
```

What would this program look like in Prolog?



Another Mercury Program

```
:- module fibo.

:- interface.

:- import_module int, list.

:- func fibo(int) = list(int)
:- mode fibo(in) = out is semidet.

:- implementation.

fibo(0) = [1].
fibo(1) = [1, 1].
fibo(N) = F :- N > 1,
               [F1, F2 | FRest] = fibo(N - 1),
               F = [F1 + F2, F1, F2 | FRest].
```

Why is the `fibo` *function declared semideterministic?*

It fails for N < 0!

409