Lambda Calculus

see notes on Lambda Calculus

Shakil M. Khan
adapted from Gunnar Gotshalks

recap

- so far:
  - Lisp data structures
  - basic Lisp programming
  - bound/free variables, scope of variables
  - Lisp symbols, property lists
  - control over evaluation and function application
- today:
  - Lambda calculus
  - pure functional programming

λ-calculus history

- developed by Alonzo Church during 1930's-40's
- one fundamental goal was to describe what can be computed
- full definition of λ-calculus is equivalent in power to a Turing machine
  - Turing machines and λ-calculus are alternate descriptions of our understanding of what is computable

λ-calculus history (cont'd)

- in the mid to late 1950's, John McCarthy developed Lisp
  - a programming language based on λ-calculus
  - implementation includes syntactic sugar functions and forms that do not add to the power of what we can compute but make programs simpler and easier to understand

λ-calculus basis

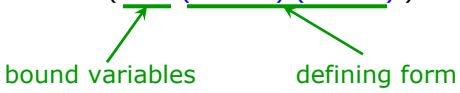
- mathematical theory for anonymous functions
  - functions that have not been bound to names
- present a subset of full definition to get the flavor
- notation and interpretation scheme identifies
  - functions and their application to operands
    argument-parameter binding
  - clearly indicates which variables are free and which are bound

bound and free variables

- bound variables are similar to local variables in Java function (or any procedural language)
  - changing the name of a bound variable (consistently) does not change the semantics (meaning) of a function
- free variables are similar to global variables in Java function (or any procedural language)
  - changing the name of a free variable normally changes the semantics (meaning) of a function

λ-functions

- consider following expression
  - (u + 1) (u – 1)
  - is u bound or free?
- disambiguate the expression with the following λ-function
  - (λ u . (u + 1) (u – 1))



  - clearly indicates that u is a bound variable
- note the parallel with programming language functions
  - functionName (arguments) { function definition }
  - it seems obvious now but that is because programming languages developed out of these mathematical notions

λ-functions (cont'd)

- consider the following expression
  - (u + a) (u + b)
- can have any of the following functions, depending on what you mean
  - (λ u . (u + a) (u + b))
    u is bound, a and b are free (defined in the enclosing context)
  - (λ u, b . (u + a) (u + b))
    u and b are bound, a is free
  - (λ u, a, b . (u + a) (u + b))
    u, a and b are all bound, no free variables in the expression

function application

- functions are applied to arguments in a list immediately following the λ-function
  - { λ u . (u + 1) (u + 2) } [3]
    3 → u then ==> (3 + 1) (3 + 2) → 20
  - { λ u . (u + a) (u + b) } [7 – 1]
    7–1 → u then ==> (6 + a) (6 + b)
    and no further in this context
  - {λ u, v . (u – v) (u + v) } [2p + q , 2p - q]
    ==> ((2p+q) – (2p - q)) ((2p + q) + (2p – q))
    can pass expressions to a variable
- can use different bracketing symbols for visual clarity; they all mean the same thing.

using auxiliary definitions

- build up longer definitions with auxiliary definitions
  - define u / (u + 5)
    where u = a (a + 1)
    where a = 7 – 3

{ λ u . u / (u + 5) } [{ λ a . a (a + 1) } [7 – 3]]

note the nested function definition and argument application

==> { λ u . u / (u + 5) } [4 (4 + 1)]
→ { 20 / (20 + 5) }
→ 0.8

functions are variables

- define f (3) + f (5)
  where f (x) = a x (a + x)
  where a = 4

  { λ f . f (3) + f (5) } [{ λ a . { λ x . a x (a + x) } } [4]]

- arguments must be evaluated first
  ==> { λ f . f (3) + f (5) } [{ λ x . 4 x (4 + x) }]
  → { λ x . 4 x (4 + x) } (3) + { λ x . 4 x (4 + x) } (5)
  → 4 * 3 (4 + 3) + 4 * 5 (4 + 5) ==> 264

Lambda notation in Lisp

- Lambda expressions are a direct analogue of λ-calculus expressions
  - they are the basis of Lisp functions – a modified syntax to simplify the interpreter
- e.g.
  - (defun double (x) (+ x x))
    is the named version of the following unnamed lambda expression
    (lambda (x) (+ x x)) --- { λ x . (x + x) }
    note the similar syntax with λ-calculus and the change to prefix, from infix, to permit a uniform syntax for functions of any number of arguments

anonymous functions

- recall in the abstraction for sumint we defined support functions to handle each case

```
(defun double (int) (+ int int))
(defun square (int) (* int int))
(defun identity (int) int)
```

- this adds additional symbols we may not want, especially if the function is to be used only once
- using lambda we get the same effect without adding symbols

```
(sumint #'(lambda (int) (+ int int)) 10)
(sumint #'(lambda (int) (* int int)) 10)
(sumint #'(lambda (int) int) 10)
```


the function 'function'

- what is the meaning of #' in the following

```
(sumint #'(lambda (int) (+ int int)) 10)
```

- it is a short hand

```
#'(...) → (function (...))
```

- one of its attributes is it works like quote, in that its argument is not evaluated, thus, in this simple context the following will also work

```
(sumint '(lambda (int) (+ int int)) 10)
```

- later we will see another attribute of function that makes it different from quote
- whenever a function is to be quoted use #' in place of '

recursion

- recursion with lambda functions uses labels to temporarily name a function
- the following is a general λ-calculus template
  - the name is in scope within the entire body but is out of scope outside of the lambda expression

```
{ label name ( lambda arguments .
                    body_references_name ) }
```

- in Lisp can use labels to define a mutually recursive set of functions

```
( labels (list of named lambda expressions)
   sequence of forms using the temporarily named
   functions
 )
```


example of recursion

- a recursive multiply that uses only addition
  - the temporary function is called mult
  - use quote and eval not function

```
(eval '(labels
            ((mult (k n)
                   (cond ((zerop n) 0)
                         (t (+ k (mult k (1- n))))
                   )
            ))
            (mult 2 3)
       )
)
```

example of recursion (cont'd)

- recTimes computes k * n by supplying the paramters to a unary function that is a variation of previous example

```
(defun recTimes (k n)
    (labels (( temp (n)
                (cond ((zerop n) 0)
                      ( t (+ k (temp (1- n))))
                ))
            (temp n)
    ))
```


Functional Programming

see notes on functional programming

Shakil M. Khan
adapted from Gunnar Gotshalks

functional programming: history

- 1977 Turing[1] Lecture John Backus described functional programming
  - "the problem with 'current languages' is that they are word-at-a-time"[2]

    notable exceptions then were Lisp and APL now ML, Haskell and others

1 Turing award is the Nobel prize of computer science

2 "Word-at-a-time" translates to "byte-at-a-time" in modern jargon. A word typically held 2 to 8 bytes depending upon the type of computer

meaningful units of work

- work with operations meaningful to the application, not to the underlying hardware & software
  - analogy with word processing is not to work with characters and arrays or lists of characters
  - but work with words, paragraphs, sections, chapters and even books at a time, as appropriate

requires abstraction

- abstract out the control flow patterns
- give them names to easily reuse the control pattern
  - for example in most languages we explicitly write a loop every time we want to process an array of data
  - if we abstract out the control pattern, we can think of processing the entire array as a single operation

example 1

- consider the inner product of two vectors
  < a1, a2, ... , an > < b1, b2, ... , bn >
  → (a1*b1 + a2*b2 + ... + an*bn)
- in Java or C/C++, the following is an algorithm

```
result = 0;
for (i = 1 , i <= n , i++) {
        result = result + a[i]*b[i];
}
```

- note the explicit loop (or recursion) and introduction of variables result, i and n (have to explicitly know the length of the vectors

example 1 – FP form

- innerProduct ::= (/ +) o (α x) o trans
- note the following properties of functional programs
  - NO explicit loops (or recursion)
  - NO sequencing at a low level
  - NO local variables
- in addition, functional programs have the following properties
  - functions as input – in the above
    + (plus), x (times), trans
  - functions as output – not shown in the above
    in FP frequently write functions that produce a new function using other functions as input

evaluating (/ +) o (α x) o trans

- apply the function to a single argument consisting of a list of the actual arguments.
  - innerProduct : < < a1, ... , an > < b1, ... bn > >
- work from right to left – o is function composition
  - f o g : x → f (g (x))
  - thus we execute trans first – which means the transpose of a matrix – swap rows and columns

    trans : < < a1, ... , an > < b1, ... bn > >
    → < < a1, b1> < a2, b2 > ... < an, bn > >

evaluating ... (cont'd)

- now execute (α ×)
  - (α ×) – read as apply times to all – means apply the function × (times) to all items in the argument list
  - (α ×) : < < a1, b1> < a2, b2 > ... < an, bn > >
    → < a1 × b1, a2 × b2, ... , an × bn >
- now execute (/ +)
  - (/ +) – read as reduce using + – means put the function + (plus) between the arguments and apply from left to right
  - (/ +) : < a1 × b1, a2 × b2, ... , an × bn >
    → a1 × b1 + a2 × b2 + ... + an × bn
- and we have the inner product

Backus notation (BN) and Lisp

- data structures – the list
  - Lisp – (a b c d)
  - BN – < a, b, c, d >

    the list is a fundamental structure we will see it again in Prolog
- selector functions
  - Lisp – car / first, cdr / rest
  - BN – tail (equivalent to rest), 1, 2, 3, ... as needed or implemented, select item from the list
- constructor functions
  - Lisp – cons
  - BN – [f-1 , f-2 , ... , f-n] – each f-i operates on the input to produce a list as output

BN and Lisp (cont'd)

- choice – if … then … else …
  - Lisp – (cond (p.1 s.1-1 s.1-2 … s.1-p)
    (p.2 s.2-1 s.2-2 … s.2-q)
    …
    (p.n s.n-1 s.n-2 … s.n-r)
)
  - BN –p.1 --> function.1 ; if p.1 then function.1 else…
    p.2 --> function.2 ;
    … ;
    p.n --> function.n

BN and Lisp (cont'd)

- function application
  - Lisp – (f x1 … xn), (apply f (x1 … xn)), (funcall f x1 … xn)
  - BN – f : < x1, … xn >
- mapping functions
  - Lisp – (mapcar f …), (maplist f …), …
  - BN – (α f)
- other functions

	reduction	composition	binding	constant
– Lisp –	(reduce f x)	(comp f g)	(bu f k)	literal
– BN –	(/ f)	f o g	(bu f k)	literal

inner product – one argument versions

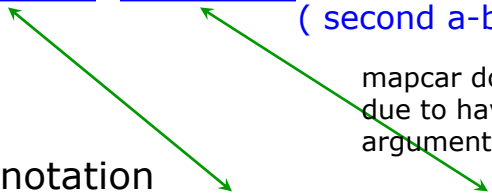
- lisp recursive version

```
(defun innerProduct (a-b-pair)
  ( cond (( null (car a-b-pair)) 0)
         ( t ( + ( * (caar a-b-pair) (caadr a-b-pair))
                 (innerProduct (list (cdar a-b-pair)
                                     (cdadr a-b-pair)))))
))
```


inner product – one argument versions (cont'd)

- lisp functional version

```
( defun innerProduct ( a-b-pair )
    ( reduce `+ ( mapcar `* ( first a-b-pair )
                            ( second a-b-pair ))))
```



mapcar does transpose due to having multiple arguments

- Backus notation

innerProduct ::= (/ +) o (α x) o trans

library of functions

- depending upon the application area other functions are created.
  - e.g. trans – transpose a matrix
- some are created using existing functionals
  - e.g. innerProduct

library of functions (cont'd)

- others are created "outside" of the system for efficiency reasons
- e.g. trans may be more efficient to implement outside of Lisp
  - although as compiler knowledge grows compilers produce more efficient code than "coding by hand"
  - machine speeds increase so many functions execute fast enough
- the file www.cse.yorku.ca/course/3401/functionals.lsp contains additional library functions

binding function – bu

- given a binary function it is often useful to bind the first parameter to a constant – creating a unary function
  - also called currying after the mathematician Curry who developed the idea
  - (bu '+ 3) – creates a unary "add 3" from the binary function "+"
    (mapcar (bu '+ 3) '(1 2 3)) → (4 5 6)
  - cons x before every item in a list
    (mapcar (bu 'cons 'x) '(1 2 3)) → ((x.1) (x.2) (x.3))
- note that mapcar expects a function definition as the second argument, so we use bu to help construct the function

bu (cont'd)

- we could define the function 3+

  (define 3+ (x) (+ 3 x))

  and use

  (mapcar '3+ ' (1 2 3)) → (4 5 6)

  but this adds to our name space
- for use-once functions we can use lambda expressions

  (mapcar #'(lambda (x) (+ 3 x)) '(1 2 3)) → (4 5 6)

  (mapcar (function(lambda (x) (+ 3 x))) '(1 2 3))

  → (4 5 6)

bu (cont'd)

- the previous slide solutions are seen as being clumsy and more difficult to read compared to the following – bu has a clear meaning – with the above you have to reverse engineer to understand

  (mapcar (bu `+ 3) `(1 2 3)) ==> (4 5 6)
- can define functions using bu

  (defun 3+ (y) (funcall (bu `+ 3) y))

  in such cases we would rather write

  (defun 3+ (y) (+ 3 y))

  we do not normally use bu to define named functions

bu (cont'd)

- BU is defined as follows

```
(defun bu (f x)
        #`(lambda (y) (funcall f x y))
)
```

- the long form

```
(defun bu (f x)
        (function (lambda (y) (funcall f x y)))
)
```

- BU uses a function as input and produces a function as output

bu (cont'd)

- how does Lisp represent the output of bu?
- in GCL (Gnu Common Lisp) you can see what takes place
  - (bu '+ 3)

```
(LAMBDA-CLOSURE ( ( X 3) ( F + )) ()
( (BU BLOCK #<@001E8D10>) )
  (Y)
  (FUNCALL F X Y)
)
```

- we see the parameter and body from the definition of bu together with the bindings ((X 3) (F +))
- the closure adds the bindings to the environment so the body uses those bindings when it executes

the functional rev

- rev – reverse the order of the arguments of a binary function

```
(defun rev (f) #' (lambda (x y)(funcall f y x)))
```

- earlier we wrote

```
(mapcar (bu 'cons 'a) '(1 2 3))
        → ((a.1) (a.2) (a.3))
```

- suppose we want ((1.a) (2.a) (3.a)) then we write

```
(mapcar (bu (rev 'cons) 'a) '(1 2 3))
        → ((1.a) (2.a) (3.a))
```

other useful functionals

- on course page (functionals.lsp) and the notes on functionals, the following functionals are described
  - (comp unaryFunction1 unaryFunction2)
    compose two unary functions
  - (compl unaryFunction1 unaryFunction2 ... unaryFunctionN)
    compose a list of unary functions
  - (trans matrix)
- see slides on developing functional programs

other useful functionals (cont'd)

- (distl anItem theList)
  distribute anItem to the left of items in theList
  (distl 'a '(1 2 3)) → ((a 1) (a 2) (a 3))
- (distr anItem theList)
  distribute anItem to the right of items in theList
  (distr 'a '(1 2 3)) → ((1 a) (2 a) (3 a))

Examples of how a Functional Program can be Developed

from an existing recursive program analysis of input and output diagrams

Shakil M. Khan
adapted from Gunnar Gotshalks

transpose a 2d matrix – 1

- 2-d matrix is represented as a list of rows all of the same length
- e.g.

  1 2 3
  4 5 6 → ((1 2 3) (4 5 6) (7 8 9))
  7 8 9

- the transpose (swap rows and columns) of the above is

  1 4 7
  2 5 8 → ((1 4 7) (2 5 8) (3 6 9))
  3 6 9

transpose a 2d matrix – 2

```
(defun trans ( theMatrix )
      (cond ( ( null ( car theMatrix ) ) nil )
            ( t ( cons ( firstOfEach theMatrix )
                  ( trans (restOfEach theMatrix ))))
))
(defun firstOfEach ( theMatrix ) ; extract first of each row
      (cond ( ( null theMatrix ) nil )
            ( t (cons ( caar theMatrix )
                  ( firstOfEach ( cdr theMatrix ))))
))
(defun restOfEach ( theMatrix ) ; remove first of each row
      (cond ( ( null theMatrix ) nil )
            ( t ( cons ( cdar theMatrix )
                  ( restOfEach ( cdr theMatrix ))))
))
```


transpose a 2d matrix – 3

- analysis of the transpose program shows that trans invokes firstOfEach to every decreasing rows (restOfEach)
- this is what maplist does
- so a first pass of trans becomes

```
(defun trans (theMatrix)
   (maplist 'firstOfEach theMatrix)
)
(trans '( (1 2 3) (4 5 6) (7 8 9) ) ) → ((1 4 7) (4 7) (7))
```

- what went wrong?

transpose a 2d matrix – 4

- put a print statement in firstOfEach

```
(defun firstOfEach (theMatrix) ; extract first of each row
  (print theMatrix)
  (cond ((null theMatrix) nil)
        (t (cons (caar theMatrix)
                 (firstOfEach (cdr theMatrix))))
))
```


transpose a 2d matrix – 5

- output

```
((1 2 3) (4 5 6) (7 8 9)) ; first call from maplist
((4 5 6) (7 8 9)) ; recursion
((7 8 9))
NIL
((4 5 6) (7 8 9)) ; second call from maplist
((7 8 9)) ; recursion
NIL
((7 8 9)) ; third call from maplist
NIL ; recursion
((1 4 7) (4 7) (7)) ; the answer
```

transpose a 2d matrix – 6

- maplist is removing the rows not the first of each row because maplist is working on the matrix a row at a time
  - input is ((1 2 3) (4 5 6) (7 8 9)) -- one list of rows
- we want maplist to work on each row
  - input should be (1 2 3) (4 5 6) (7 8 9) -- three lists
  - this is a common problem we want to remove the outer parenthesis
  - recall that apply removes the outer level of parenthesis when invoking a function on arguments
- thus trans becomes

```
(defun trans (theMatrix)
   (apply `maplist `firstOfEach theMatrix)
)
```


transpose a 2d matrix – 7

- we try trans and get an error message such as

```
Error: Expected 1 args but received 3 args
Fast links are on: do (si::use-fast-links nil) for
debugging
Error signalled by MAPLIST
Broken at FIRSTOFEACH
```

transpose a 2d matrix – 8

- ah! now we have one argument for each row as input to firstOfEach, but it expects a single argument – a list of rows
  - use the keyword &rest to collect all the arguments into one

```
(defun firstOfEach ( &rest theMatrix )
  (print theMatrix)
  ( cond ( ( null theMatrix ) nil )
         ( t ( cons ( caar theMatrix )
                    ( firstOfEach ( cdr theMatrix ))))
))
```


transpose a 2d matrix – 9

- we try trans and get infinite recursion – the print statement shows the following for the first few lines

```
((1 2 3) (4 5 6) (7 8 9))
(((4 5 6) (7 8 9))) ; list nested one deeper
(NIL)
(NIL)
(NIL) goes on forever
```

transpose a 2d matrix – 10

- each recursive call to firstOfEach adds a layer of parenthesis
  - again a common error – we need to remove the parenthesis before the recursive call – use apply

```
(defun firstOfEach ( &rest theMatrix )
  ( cond ( ( null theMatrix ) nil )
         ( t ( cons ( caar theMatrix )
                    ( apply 'firstOfEach ( cdr theMatrix ))))
))
```


transpose a 2d matrix – 11

- trans now works with the upper level being a functional, but firstOfEach is still recursive

```
(defun trans ( theMatrix )
   ( apply `maplist `firstOfEach theMatrix )
)
(defun firstOfEach ( &rest theMatrix )
   ( cond ( ( null theMatrix ) nil )
          ( t ( cons ( caar theMatrix )
                     ( apply `firstOfEach ( cdr theMatrix ))))
))
```

transpose a 2d matrix – 12

- notice that firstOfEach takes the first item from each sublist

```
(defun firstOfEach (&rest theMatrix)
   (cond ((null theMatrix) nil)
            (t (cons (caar theMatrix)
                     (apply 'firstOfEach (cdr theMatrix))))
   ))
```

- car gives the first of a list and mapcar will apply it to every sublist in a list and collect the results in a list so we have

```
(defun firstOfEach ( &rest theMatrix )
    ( mapcar 'car theMatrix )
)
```


transpose a 2d matrix – 13

- we now have two functionals for the solution

```
(defun trans ( theMatrix )
          ( apply 'maplist 'firstOfEach theMatrix )
)
(defun firstOfEach ( &rest theMatrix )
          ( mapcar 'car theMatrix )
)
```

- using lambda we can eliminate firstOfEach

```
(defun trans ( theMatrix )
   (apply 'maplist #'( lambda ( &rest theMatrix )
                             ( mapcar 'car theMatrix ))
          theMatrix
   )
)
```

transpose a 2d matrix – 14

- but nothing beats creative insight and knowledge of available operations!!!
- the following gives the transpose

```
(defun trans ( theMatrix )
        (apply 'mapcar 'list theMatrix )
)
```


all pairs functional – 1

- we want the following functional

allPairs : < <a, b, c> <1, 2, 3, 4> > input

→

< <a,1> <a,2> <a,3> <a,4>
<b,1> <b,2> <b,3> <b,4> output
<c,1> <c,2> <c,3> <c,4> >

- we make use of the 'picture' of the input and output to infer a functional solution

all pairs functional – 2

- looking at the functionals in the library it seems that distribution may be useful
- lets try it

  distl : < <a, b, c> <1, 2, 3, 4> >

  →

  < << a, b, c >, 1 > << a, b, c >, 2 > << a, b, c >, 3 > ... >
- looks good but we want to distribute second argument over the first
- rev could be used but we have distr

  distr : < <a, b, c> <1, 2, 3, 4> >

  →

  < < 1, < a, b, c > > < 2, < a, b, c > > < 3, < a, b, c > ... >

all pairs functional – 3

- we have

  distr : < <a, b, c> <1, 2, 3, 4> >

  →

  < < 1, < a, b, c > > < 2, < a, b, c > > < 3, < a, b, c > ... >
- if we distribute 'right' the numbers over each list we have

  < < <a , 1>, < b, 1>, <c, 1 > > ... >
- but examining the output we see that 'a' is repeated first not the "1"

  < <a,1> <a,2> <a,3> <a,4>
  <b,1> <b,2> <b,3> <b,4>
  <c,1> <c,2> <c,3> <c,4> >

  output

all pairs functional – 4

- what we need to do is to reverse the order of the arguments so the letters are distributed first

  distr o [2 , 1] : < <a, b, c> <1, 2, 3, 4> >

  → distr : < <1, 2, 3, 4> <a, b, c> >

  → < < a, < 1, 2, 3, 4 > > < b, < 1, 2, 3, 4 > > ... >
- now if we apply distribute left to each sublist we have

  (α distl) : < < a, < 1, 2, 3, 4 > >
  < b, < 1, 2, 3, 4 > > ... >

  →

  < < < a, 1 > < a, 2 > < a, 3 > < a, 4 > >
  < < b, 1 > ... >

all pairs functional – 5

- so far we have

  (α distl) o distr o [2 , 1]

  →

  < < < a, 1 > < a, 2 > < a, 3 > < a, 4 > > < < b, 1 > ... >
- but we have the pairs nested within an extra pair of lists
- what we need to do is to reduce the lists into one using append

  (/ append) :

  < < < a, 1 > < a, 2 > < a, 3 > < a, 4 > > < < b, 1 > ... >

  →

  < < a, 1 > < a, 2 > < a, 3 > < a, 4 > < b, 1 > ... >

all pairs functional – 6

- so the final function definition is

  allPairs ::= (/ append) o (α distl) o distr o [2 , 1]
- other orderings are possible using other combinations of swapping or not swapping the initial lists and using left or right distribution for the second distribution

  allPairs ::= (/ append) o (α distr) o distr o [2 , 1]

  allPairs ::= (/ append) o (α distl) o distr

  allPairs ::= (/ append) o (α distr) o distr

info

- assignment/report 1 is out
  - due June 4
  - hand in hard copy + submit soft copy (programs only: submit 3401 r1 <files>)
- class test 1 (7:00~8:30) on June ???
  - June 4 (2 weeks from now)

    June 11 (3 weeks from now)
  - included everything covered up to June ???