Functional Languages

Scheme

Running Scheme on CIMS Cluster

- Use putty to login to CIMS:
  access.cims.nyu.edu
- Start the Scheme interpreter, type:
  $mzscheme
- Exit the Scheme interpreter, type:
  >(exit)

  NOTE: must include the parentheses

Scheme: Basics

- Basic Operators:
  - Read-Evaluate-Print:
    - In Scheme command line, user types expression
    - Expression is read in, then evaluated
    - Result of expression is printed
  - Example: Arithmetic:
    - All are prefix notation
    - Sample Addition:

Sample: >(+ 1/3 1/3 1/3)

Note: '1/3' is treated as a number and not as 'one divided by three' because arithmetic operations are prefix.

Scheme: Definitions

- Definitions:
  - (define id expression)
  - Name is bound to result of expression
  - Sample: >(define x (+ 1 2))

    >x

    3

    Note: (define x (+ 1 2)) is the same as (define x 3)
  - (define (id+) expression+)
  - The first id is the name of the function, and any remaining ids are its arguments. The function performs the expressions.
  - Sample: >(define (add a b) (+ a b))

    > (add 2 3)

    5
  - The function, called “add”, has arguments a and b, and returns their sum.

Scheme: Lists

- Create a list:

  >(define x '((1)(2 4)3(4 5)))
  >(length x)
  4

- Basic List functions:

  car = first item (left half)
  cdr = second and all subsequent items in list (right half)
  cadr = second item
  cons = cons x onto y means to construct a new object with (cons x y).
  >(car x)
  (1)
  >(cdr x)
  ((2 4) 3 (4 5))
  (cadr x)
  (2 4)

Scheme: Lists (cont'd)

- Basic list functions (continued):

```
>(cdr '(6))
()
>(cdar x) ;; same as (cdr (car x))
()
>(cadr x) ;; same as (car (cdr x))
(2 4)
>(cons x '(6)) ;; append '6' to the end of the list
(((1) (2 4) 3 (4 5)) 6)
>(cons 7 x) ;; insert '7' into the beginning of the list
(7 (1) (2 4) 3 (4 5))
>(list x 6 8)
(((1) (2 4) 3 (4 5)) 6 8)
>(append '(1 2) '(3 4))
(1 2 3 4)
```

Scheme: Operations on Lists

- Quote doesn't get evaluated:

```
>(list (+ 1 2))
(3)
>'(+ 1 2)
(+ 1 2)
```

- Map: applies the function to each element of the list

```
>(map abs '(1 -1 2 3))
(1 1 2 3)
```

- Apply: applies the function to the entire list:

```
>(apply + '(1 2 3))
6
```

Scheme: Recursion

- Computing nth fibonacci number:

  Fibo: 0, 1, 1, 2, 3, 5, 8,…

  Nth fibo = n-2+n-1

```
>(define (fibo n)
(if (< n 2)
;; if n=0, 0 if n=1, 1
n
(+ (fibo (- n 1)) (fibo (- n 2)))))
(fibo 0)
0
(fibo 1)
1
…
(fibo 6)
8
```

ML: Online Resources

- Online introductory book on programming in SML (Standard ML):

  http://www.cs.cmu.edu/~rwh/smlbook/online.pdf

ML: Running SML

SML is a strongly typed language - all (well-formed) expressions have a type that can be determined by examening the expression.

- Connect to access.cims.nyu.edu using putty
- At the prompt, type **sml** (all lower case) to start the SML program.
  - SML will type "-" as its prompt.
  - To exit sml, type **ctrl-D**
- Type an expression, ending it with a semicolon and <enter>.
  - SML will interpret it, name the result "it", and print out its value and type as:

    val it = <value> : <type>

    For example, type:

    -2+2;

    val it = 4 : int (this is the response)

    The expression evaluated to '4', and this value is bound to the name 'it', and is of type 'int'

ML: Compiling and Running SML

- Create an SML program file:
  hello.sml
- At the prompt, compile the program using mlton:

  mlton hello.sml

  An executable file will be generated.

- Run the executable similar to a C program:

  ./hello

ML: Basic Expressions

- Mathematical expressions are infix compared to Scheme, which are prefix.
- Basic Arithmetic operators: '+', '-', '*', '/'.

  NOTE: the operator, '/' is applied to reals (an error will occur if you don't use reals). For integers, for example, you should use 'div':

  - 2 div 3;
  val it = 0 : int

- Negative numbers are represented using '~':

  - 2 + (~4);
  val it = ~2 : int

ML: Define and Call Functions

- To define a function:

```
- fun add1 x = x + 1;
val add1 = fn : int -> int
```

- To call a function:

```
- add1 4;
val it = 5 : int
```

ML: Hello World

- Function: print(string):

- print("Hello World from ML!\n");

Hello World from

ML: Types

- Basic Types:
  - Integers, reals: 1, 0.1
  - Strings: “Hello”
- Lists:
  - Must be homogenous (i.e., all of one type).
  - Length is not limited.

    val a = [1,2,3]; or val b = [“a”,”b”,”c”];

    Cannot have: [1,”a”,3];
    Empty lists are denoted by: [] or nil
- Tuples:
  - Can combine types.
  - Length is limited.

    val x = (1,2,3); // cannot add to the end of x

    val y = (1,0.1,”a”); // int*real*string

ML: Types

- Boolean Types
  - Boolean values: true, false
  - Logical operators:
    - not(negation) : not(true);
    - andalso(conjunction) : true andalso false
    - orelse(disjunction) : true orelse false;

ML: Operations on Lists

- hd (similar to car in Scheme):

```
- val x = [1,2,3,4,5,6];
val x = [1,2,3,4,5,6] : int list

- hd x;
val it = 1 : int
```

- tl (similar to cdr in Scheme):

```
- tl x;
val it = [2,3,4,5,6] : int list
```

- @ adds elements to the front of the list:

```
- [7,8]@x;
val it = [7,8,1,2,3,4,5,6] : int list
```

ML: Operations on Lists (cont'd)

- The "cons" operator (for "construct"):
  - combines an element and a list to make a larger list.
  - The element becomes the head of the resulting list.
  - The list which is given as the second argument to cons becomes the tail of the new list.
  - The cons operator is written with two colons, as "::".

```
- 4::[3,2,6,7];
val it = [4,3,2,6,7] : int list
```

ML: More Operations on Lists

- Explode:
  - ‘Explodes’ a string into a list of characters

    - explode("hello");
    val it = [#"h",#"e",#"l",#"l",#"o"] : char list

- Implode:
  - Form a string from a list of characters

    - implode [#"h",#"e",#"l",#"l",#"o"];
    val it = "hello" : string

ML: Functions

- General Form:

  fun <identifier> (<parameters>) = <expression>;
- Function Type:
  - Type of parameters → type of expression
- Example:

- fun double(x) = 2*x;

val double = fn : int → int

- double(4);

val it = 8 : int

*** Declares double as a function from integers to integers (of type int → int).**

*** If apply the wrong argument, for example, a real, then will get an error.**

ML: Functions

- Can explicitly specify types

```
- fun max(x:int,y:int,z:int) =
if((x>y) andalso (x>z)) then x
else (if (y>z) then y else z);
val max = fn : int * int * int -> int

- max(3,2,2);
val it = 3 : int

- max(3,6,5);
val it = 6 : int

- max(3,4,6);
val it = 6 : int
```

ML: Functions with List

- Computes the length of a list

```
- fun length L = if null(L) then 0
else length (tl(L))+1;
val length = fn : 'a list -> int

- val z = [1,2,3,4,5,6,7,8,9,0];
val z = [1,2,3,4,5,6,7,8,9,0] : int list

- length z;
val it = 10 : int
```

ML: Recursion

- Language Feature:
  - The use of recursion is one of the main characteristics of functional languages.
  - Recursion is used as a structuring mechanism in preference to iterative constructs (e.g., while-loops).
  - Recursion is useful for defining list processing functions (next slide)

ML: Recursion: Concatenation of Two Lists

- Example: concat
  - Arguments: two lists of same type
  - Returns: concatenated list
- Recall, a list is:
  - An empty list: [], nil
  - Of the form x::y

```
- [1,2,3] = 1::[2,3];
val it = true : bool
```

  - If use :: on two lists, get an error:

```
- [1,2]::[3,4];
stdIn:39.1-39.13 Error: operator and operand don't agree [literal]
operator domain: int list * int list list
operand: int list * int list
in expression:
(1 :: 2 :: nil) :: 3 :: 4 :: nil
```

ML: Recursion: Concatenation of Two Lists (cont'd)

- Use recursion to implement concat:
  - Concats two lists

```
- fun concat(x,y) = if x=[] then y
else hd(x)::concat(tl(x),y);
stdIn:59.23 Warning: calling polyEqual
val concat = fn : ''a list * ''a list -> ''a list
- concat([],[2,3]);
val it = [2,3] : int list
- concat([4,5],[]);
val it = [4,5] : int list
- concat([2,3],[4,5,6]);
val it = [2,3,4,5,6] : int list
```

  - Concatenation of lists in ML: '@', which is a built-in operator.

```
- [2,3,4]@[5,6,7];
val it = [2,3,4,5,6,7] : int list
```

ML: Recursion: Reverse a List

- Reversing a List – using recursion and the built-in concat operator (‘@’) in the recursive definition:

```
- fun reverse(L) =
if L = nil then nil
else reverse(tl(L)) @ [hd(L)];
stdIn:84.6 Warning: calling polyEqual
val reverse = fn : ''a list -> ''a list

- reverse [1,2,3];
val it = [3,2,1] : int list
```

ML: Recursion: Using 'Remove' as an Auxiliary Function

- The funcion 'remove' can be used as an auxiliary function in the definition of another function that removes all duplicate occurrences of elements from a list:

```
- fun removedupl(L) =
if (L=[]) then []
else hd(L)::remove(hd(L),removedupl(tl(L)));
stdIn:118.6 Warning: calling polyEqual
val removedupl = fn : ''a list -> ''a list

- removedupl([2,3,4,2,2,6,5,2,4]);
val it = [2,3,4,6,5] : int list
```

ML: Recursion: Remove Elements from a List

- Recursive: removes all occurrences of the first element from the list (second element):

```
- fun remove(x,L) =
if (L=[]) then []
else (if (x=hd(L))
then remove(x,tl(L))
else hd(L)::remove(x,tl(L)));
stdIn:105.6 Warning: calling polyEqual
stdIn:106.12 Warning: calling polyEqual
val remove = fn : ''a * ''a list -> ''a list

- remove(1,[5,3,1]);
val it = [5,3] : int list

- remove(2,[4,2,4,2,4,2,2]);
val it = [4,4,4] : int list
```

ML: Recursion

- Fibonacci numbers
  - Definition: Fn = Fn-1 + Fn-2
  - Seed Values: F0 = 0, F1 = 1
  - Sequence: 0,1,1,2,3,5,8,13,21,34,55,89,144,…
  - Program:

```
- fun fib x = if (x=1) then x
else if (x=0) then x
else
fib(x-1)+fib(x-2);
- fib 0;
val it = 0 : int
- fib 1;
val it = 1 : int
- fib 2;
val it = 1 : int
- fib 3;
val it = 2 : int
- fib 4;
val it = 3 : int
- fib 5;
val it = 5 : int
- fib 6;
val it = 8 : int
```

ML: Recursion (cont'd)

- Factorial
  - Example: 5! = 5*4*3*2*1 = 120
  - Program:

```
- fun fac n = if (n=1) then 1
else n*fac(n-1);
val fac = fn : int -> int
-fac 5;
val it = 120 : int
```

ML: Mutual Recursion

- Mutual recursion: when two functions are defined in terms of each other.
- Example:

```
- fun even(0) = true
| even(n) = odd(n-1)
and
odd(0) = false
| odd(n) = even(n-1);
val even = fn : int -> bool
val odd = fn : int -> bool
- even(2);
val it = true : bool
- even(3);
val it = false : bool
- odd(3);
val it = true : bool
- odd(4);
val it = false : bool
```

You have to use the keyword and to combine mutually recursive functions like this. Otherwise the compiler would flag an error when you refer to oddbefore it has been defined (as normally functions in SML need to be defined before they can be used).

ML: Merge Sort

- To sort the list:
  - First split the list, L, into two disjoint sublists.
  - Then, recursively sort the sublists.
  - Finally, merge the sorted sublists.
- Needed functions:
  - Split a list into two sublists.
  - Merge the two sublists into one sorted list.

ML: Merge Sort (cont'd)

- Implementing firstk : returns the list of the first k elements of L by recursively calling firstk and using the con function to construct the first element with up to k elements:

```
- fun firstk(L, k) =
if k = 0 then []
else if L = nil then nil
else hd(L)::firstk(tl(L),k-1);
```

Testing firstk

```
- val L = [4,2,6,4,7,1,8];
val Left = [4,2] : int list
```

ML: Merge Sort (cont'd)

- Implementing dropk : returns the list of all but the first k elements of L by recursively calling dropk and using the tl function to return all but the first element up to k element:

```
- fun dropk(L,k) =
if k = 0 then L
else if L = nil then nil
else dropk(tl(L),k-1);
```

Testing dropk

```
- val Right = dropk(L,3);
val Right = [4,7,1,8] : int list
```

ML: Merge Sort (cont'd)

- Implementing merge :

```
- fun merge ([], M) = M
| merge (L, []) = L
| merge (x::x1, y::y1) =
if (x:int) < y then x::merge(x1, y::y1)
else y::merge(x::x1, y1);
```

Testing merge:

```
- merge([1,5,7,9],[2,3,5,5,10]);
val it = [1,2,3,5,5,5,7,9,10] : int list
```

Notes:

"|" is used for case statements

The merge function is doing the actual sorting.

ML: Merge Sort (cont'd)

- Implementing newsort :

```
- fun newsort(L) =
if L = nil then nil
else if tl(L) = nil then L
else merge(newsort(firstk(L,length(L) div 2)),
newsort(dropk(L,length(L) div 2)));
```

- Let k = length(L) div 2, then we obtain the left and right halves of L.
- Calling merge passing a sorted sublist.
- The sublist is obtained by calling firstk and dropk passing the list and the length of the list divided by 2 as k.

ML: Merge Sort (cont'd)

- Running the Merge Sort code:

```
- val L1 = [5,2,6,8,2,9,8,4,0];
val L1 = [5,2,6,8,2,9,8,4,0] : int list
- val Left = firstk(L1,2);
val Left = [5,2] : int list
- val Right = dropk(L1,2);
val Right = [6,8,2,9,8,4,0] : int list
- val Left_s = newsort(Left);
val Left_s = [2,5] : int list
- val Right_s = newsort(Right);
val Right_s = [0,2,4,6,8,8,9] : int list
- merge(Left_s, Right_s);
val it = [0,2,2,4,5,6,8,8,9] : int list
- newsort(L1);
val it = [0,2,2,4,5,6,8,8,9] : int list
```

ML: Higher-Order Functions

- Language Feature:
  - Higher-order functions either:
    - Take one or more functions as arguments
    - Return a function

  - Example:

```
- fun apply(f,L) =
if (L=[]) then []
else f(hd(L))::(apply(f,tl(L)));
stdIn:129.6 Warning: calling polyEqual
val apply = fn : (''a -> 'b) * ''a list -> 'b list

- fun square(x) = (x:int)*x;
val square = fn : int -> int

- apply(square,[2,3,4]);
val it = [4,9,16] : int list
```