CSC/MAT-220: Discrete Structures

Lab 2: ML and LaTeX

Due September 22, 2017

Part 1: ML

Overview. At its core, ML consists of expressions to be evaluated. Each expression has three characteristics: type, value, effect. Some examples of types are int, real, char, string, bool, and the functional type. It is important to note that ML does not perform implicit conversion, like Python does. So, for example, *3.5+4* is ill-typed and will result in an error. All well-typed expression is evaluated to determine its value. For example, the expression *2+3* returns *val it = 5 : int* while the expression *print "Hello"* returns *val it = () : unit*. Note that () unit is an empty tuple that acts as a trivial placeholder for expressions that are side-effect based. As noted earlier, we will discuss effects later on in this course.

Bindings. Just as in other languages, in ML we can store values in variables and then use these variables in proceeding expressions. However, in contrast to other languages, variables in ML do not vary. It is for this reason that we adjust our terminology and say that a value is bound to a variable. For example, the expression below.

val x = 2

will bind the value 2 to the variable x. This type of binding is called a *value binding*. In addition, we may bind types, see the expression below.

type count = int

The power in *type bindings* will become more clear when we create our own types.

At this point, we note that the value binding above forces the compiler to implicitly determine the variable's type. We can explicitly declare this for the compiler, see the example below.

val pi : real = 3.14 and e : real = 2.71

Note that, *and* is a logical operator for evaluating two expressions. Whereas, *andalso* is an operator for combining two boolean variables.

The purpose of a binding is to make a variable or type available for use within a particular scope. In ML, scope is static meaning, the scope of a binding is determined by text, and not the order of evaluation. In the previous examples we assumed global scope, and in what follows we discuss how to limit scope.

Limiting Scope. The scope of a binding may be limited using *let* and *local* expressions. The form of these expressions are as follows.

let *dec* in *exp*	local *dec* in *dec'*
The scope of the declaration *dec* is limited to the expression *exp*.	The scope of the declaration *dec* is limited to the declaration *dec'*.

To clarify when you should use *let* vs. *local*, note that the former results in a expression, whereas the latter results in a declaration. The *local* expression is especially useful when declaring temporary helper functions. To get a slight appreciation for the power in limiting scope, consider the following.

$$
\begin{array}{l}
val\ m\ :\ int\ =\ 2 \\
val\ r\ :\ int\ = \\
let \\
\quad val\ m\ :\ int\ =\ 3 \\
\quad val\ n\ :\ int\ =\ m * m \\
in \\
\quad m * n \\
end\ *m
\end{array}
$$

This expression returns a value of 54 for the variable r. This happens because the binding of m is temporarily overridden during the evaluation of the *let* expression, and restored upon completion of this evaluation.

Functions, Bindings, and Scope Revisited. The function allows us to abstract the calculation of the value of an expression. The functional data type in ML is of the form $type \rightarrow type'$. In Lab 1, we gave several syntax examples for working with functions. By far, the most concise was as follows

fun fourthroot (x:real):real = Math.sqrt (Math.sqrt x)

It is important to note that functions in ML are *first-class*. Meaning, they may be computed as the value of an expression, bounded to a variable, and passed as an argument of a function. In the above example, we are in fact binding the function value *fn x : real => Math.sqrt (Math.sqrt x)* to the variable *fourthroot*. Also, we are explicitly declaring the type of the function *fourthroot* as $real \rightarrow real$. Now, consider the following lines of code

$$
\begin{array}{l}
val\ x : real\ = 1.0 \\
val\ y : real\ =\ 2.0 \\
fun\ f(x : real) : real\ =\ x + y \\
fun\ g(y : real) : real\ =\ x + y \\
fun\ h(x : real, y : real, f : real -> real, g : real -> real) : real\ = \\
\quad let\ val\ x : real\ =\ 3.0\ in\ f(x)\ end * g(y) * f(x)
\end{array}
$$

Note that the occurrence of x in the body of the function f refer to the parameter of f, whereas the occurrence of x in the body of the function g refer to the previous *val* binding. To make things more interesting, the function h has a local *val* binding which is shadowing the parameter x. Before proceeding to the next paragraph, determine the value of the following expression.[1]

$$val\ test{:}real = h(\tilde{}1.0{,}3.0{,}f{,}g)$$

Tuples. A distinguishing feature of ML is that aggregate data structures, such as tuples, may be created and manipulated with ease. It is not necessary to concern yourself with the allocation or deallocation of data structures, such as in Fortran, nor with any particular representation strategy involving pointers, such as in C.

An *n-tuple* is a finite ordered sequence of values of the form

$$(val_1,\ val_2, \ldots, val_n)\,,$$

and is of a *product type* of the form

$$type_1 * type_2 * \cdots * type_n.$$

Note that each type can differ, even within the same tuple.

The *0-tuple*, also known as the *null tuple*, is the empty sequence of values (). In ML, it is a value of type *unit*. The null tuple type is surprisingly useful, especially in the presence of effects. On the other hand, there seems to be no use for the *1-tuple*, so they are absent from ML. A *2-tuple*, or ordered pair, may be defined as follows

$$val\ pair : int{*}int = (2{,}3)$$

Similarly, the following are well-formed bindings

$$val\ triple\ :\ int * real * string\ =\ (2, 2.0, \text{“2”})$$
$$val\ pair\_of\_pairs\ :\ (int * int) * (real * real)\ =\ ((2,3),(2.0,3.0))$$

Note that the nesting of the parentheses matters. A pair of pairs is not the same as a quadruple. A very interesting and powerful feature of ML is the use of *pattern matching* to access specific elements of an aggregate data structure. For example,

$$val\ (x : int,\ y : real,\ z : string)\ =\ triple$$
$$val\ ((\_,\ \_),\ (r : real,\ \_)) = pair\_of\_pairs$$

In the first binding, we are assigning the corresponding values in *triple* to the variables x, y, and z, respectively. In the second binding, the underscores indicate positions in the pattern that we are ignoring, and the result of the expression is the value 2.0 is bounded to the variable r.

[1] The answer is 20.0, here is why: during the evaluation of the let expression the val binding of $x = 3.0$ shadows the parameter x, $f(3.0) = 5.0$ once the *let* expression is completed, the parameter values are $x = \tilde{}1.0$ and $y = 3.0$, and $g(3.0) = 4.0$ and $f(\tilde{}1.0) = 1.0$.

Intermission: ML Assignment

Put the following problems in a file named *yourname_lab2.sml*. In addition, include the following commented line at the top of the file

(Name, Lab #, Date *)*

Problem 1

Include every example from Part 1: ML, along with a short comment regarding the context of the example.

Problem 2

Provide detailed comments regarding what is wrong with the following lines of code, keeping in mind the intent: to define real data types vec (ordered pair) and mat (pair of pairs), and several variables for global use.

local

$type\ vec\ = real * real\ andalso\ mat\ =\ (real * real) * (real * real)$

in

$x : vec\ =\ (1.0, 2.0)\ and\ y : vec\ =\ (-2.0, 3)\ and\ a : mat\ =\ (1.0, 2.0, 2.0, 1.0)$

end

Now, provide a working version of the above code.

Problem 3

Use the *local* expression to define two helper functions and two global functions as follows. Be sure to use *pattern matching* and to test your functions.

I. fun help1 (x:vec,c:real):vec should result in a new vector that is cx.

II. fun help2 (x:vec,y:vec):vec should result in a new vector that is $x + y$.

III. fun mat_vec_mul (a:mat,x:vec):vec should result in the matrix-vector product ax. Be sure to use both help1 and help2, and note that if the columns of a are a_1 and a_2 and the elements of x are x_1 and x_2, then

$$ax = x_1a_1 + x_2a_2.$$

IV. fun rot(x:vec,t:real):vec should result in a vector that is the counter-clockwise rotation of x through the angle t. Be sure to use mat_vec_mul in the process, and note that the rotation matrix is given by

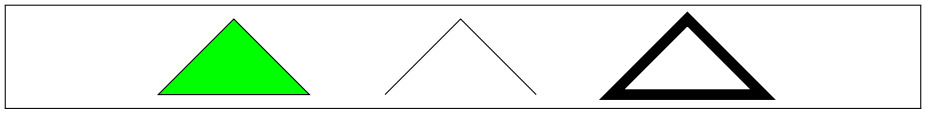
$$\begin{bmatrix} \cos(t) & -\sin(t) \\ \sin(t) & cos(t) \end{bmatrix}$$

Part 2: LaTeX

TikZ is a recursive acronym for "TikZ ist kein Zeichenprogramm." Now that you know what TikZ is not, know that TikZ is a set of higher level macros that use a lower level language PGF to create vector graphics. The designer of these languages, Professor Till Tantau, is also the creator of the TeX interpreter, which allows us to integrate these graphics within our LaTeX file. You can gain access to the TikZ macros by including the *usepackage{tikz}* line at the top of your LaTeX file. There are two basic elements of TikZ that we will make extensive use of: *paths* and *nodes*.

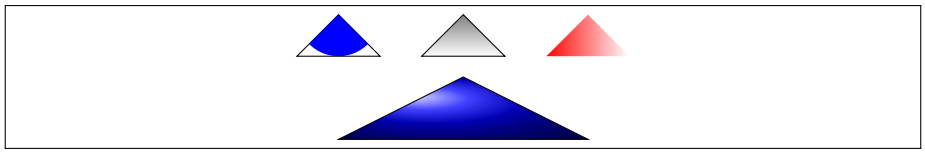
Paths. A path is a series of straight and curved line segments, connecting points within a coordinate system. See the examples below.

```
\begin{tikzpicture}
\path[draw] (0,0) -- (1,1) -- (2,0);
\path[draw,line width=4pt] (3,0) -- (4,1) -- (5,0) --cycle;
\path[draw, fill=green] (-1,0) -- (-2,1) -- (-3,0) -- cycle;
\end{tikzpicture}
```



Note how the positioning of each figure is referenced by the origin $(0, 0)$ of the coordinate system, and the use of semicolons to end each command. This picture can be made larger or smaller using the scaling option. In the following example, we highlight the use of several shading and clipping options, while using the scaling option to make the graphics fit within our desired window.

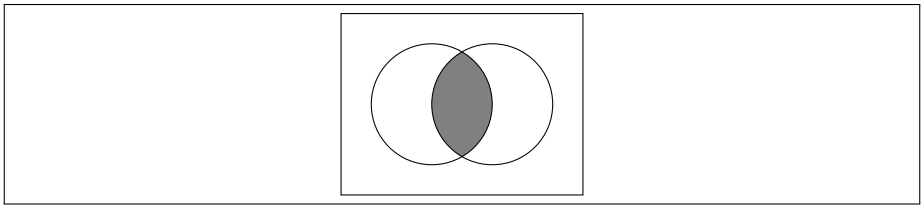
```
\begin{tikzpicture}[scale=0.55]
%Top Row
\path[shade,draw] (0,0) -- (1,1) -- (2,0) -- cycle;
\shade[left color=red] (3,0) -- (4,1) -- (5,0) -- cycle;
\begin{scope}
       \path[clip, draw] (-1,0) -- (-2,1) -- (-3,0) -- cycle;
       \path[fill=blue] (-2,1) circle (1);
\end{scope}
%Bottom Row
\path[draw, shading=ball, ball color=blue] (-2,-2) -- (1,-0.5) -- (4,-2) -- cycle;
\end{tikzpicture}
```



Note the scope control on the clipping in the top row, this is done so that the clipping does not effect the bottom row command. Second, note the shading within the path on the bottom row, in contrast to the shade in the top row, which does not outline a path. Lastly, note the circle command places a circle centered at (-2,1) with radius (1) within the predefined path clipping.

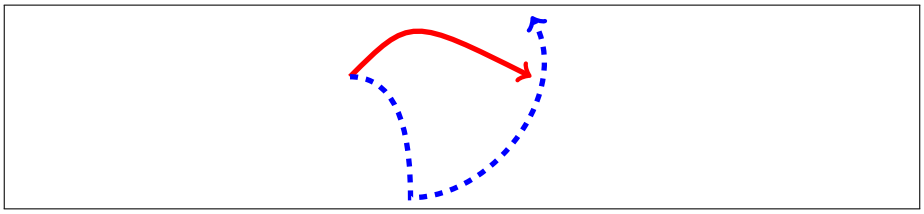
To show the power in clipping and scope, we use these commands along with some basic shapes to create a Venn Diagram of the intersection of two sets.

```
\begin{tikzpicture}[scale=0.8]
\draw (-2,1.5) rectangle (2, -1.5);
\begin{scope}
       \clip ( -0.5,0) circle (1);
       \clip (0.5,0) circle (1);
       \fill[color=gray] (-2,1.5) rectangle (2,-1.5);
\end{scope}
\draw (-0.5,0) circle (1);
\draw (0.5,0) circle (1);
\end{tikzpicture}
```



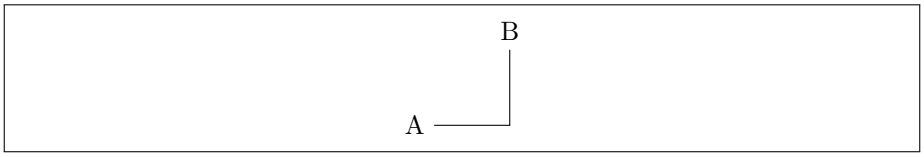
Lastly, we can also do curved paths connecting points by specifying control points, or specifying directions in and out of points. The in and out parameters are given in terms of angles relative to the start and end points, we specify each path by its color and style.

```
\begin{tikzpicture}[scale=0.8]
\draw[line width=2pt, color=red, ->] (0,0) .. controls(1,1) .. (3,0);
\draw[line width=2pt, color=blue, dashed, ->] (0,0) to [in=90, out=0] (1,-2) to [in=300,
      out=0] (3,1);
\end{tikzpicture}
```



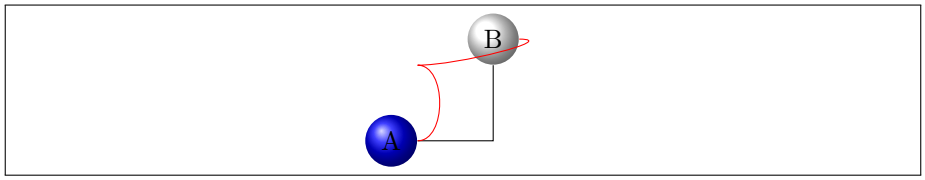
Nodes. Take a moment to appreciate the beauty of the figures previously drawn. There is just one problem, we have no labels. This is the purpose of the node. Consider the example below, and note the use of the left and above options, which force the node labels to not overlap with the path.

```
\begin{tikzpicture}
\path[draw] (0,0) node[left] {A} -- (1,0) -- (1,1) node[above] {B};
\end{tikzpicture}
```



From an organizational standpoint, it is often better to define the nodes first and then use them later.

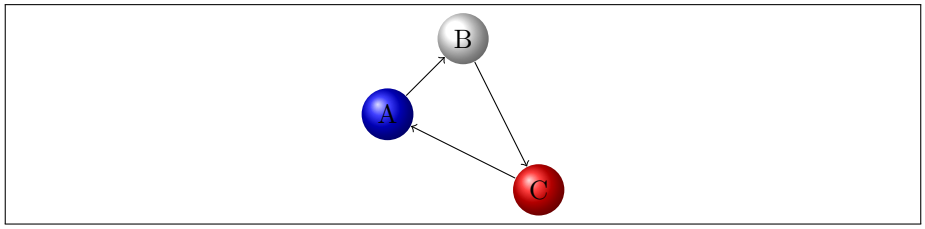
```
\begin{tikzpicture}
\node[left, circle, shading=ball, ball color=blue] (a) at (0,0) {A};
\node[above,circle, shading=ball, ball color=white] (b) at (1,1) {B};
\path[draw] (a) -- (1,0) -- (b);
\path[draw,color=red] (a) to [in=0, out=0] (0,1) to [in=0, out=0] (b);
\end{tikzpicture}
```



Note that in the previous example, we were able to use the same nodes on multiple occasions; therefore, defining the nodes first also has efficiency benefits. This can be done in a powerful iterative form, consider the following example.

```
\begin{tikzpicture}
\node[circle, shading=ball, ball color=blue] (a) at (0,0) {A};
\node[circle, shading=ball, ball color=white] (b) at (1,1) {B};
\node[circle, shading=ball, ball color=red] (c) at (2,-1) {C};

\foreach \from/\to in {a/b, b/c, c/a}
        \draw[->] (\from) -- (\to);
\end{tikzpicture}
```



There is so much more to learn about TikZ (perhaps another Lab, or two), including: calculations, loops, global references, shapes, trees, and mind maps. But, for now, you have all you need to make beautiful relational graphs. On the next page is an example of the relation 14.7 a, note the use of *tikzstyle* and *looseness*, and also the coordinate scale setting of x=5em, y=5em.

```latex
\begin{tikzpicture}[x=5em,y=5em]
%set style options the same for each node
\tikzstyle{every node}=[circle, shading=ball, ball color=white];
\node (a) at (0,0) {$1$};
\node (b) at (1,0) {$2$};
\node (c) at (1,1) {$5$};
\node (d) at (0,1) {$10$};
%draw (a,b), (a,c), (a,d), (b,d), and (c,d)
\foreach \from/\to in {a/b, a/c, a/d, b/d, c/d}
       \draw[->] (\from) -- (\to);
%draw (a,a), (b,b), (c,c), and (d,d)
\foreach \from/\to in {a/a, b/b, c/c, d/d}
       \draw[->] (\from) to [in=15, out=90, looseness=5] (\to);
\end{tikzpicture}
```