Image goes here

Introduction to Postscript

CptS 355 - Programming Language Design

Washington State University

PostScript

PostScript is a *stack-based* language using *postfix* notation. There are several stacks, but only four are of interest to us

- the operand stack - the primary stack used for most computation. Sometimes just called "the stack".
- graphics stack - a stack used to save and restore *graphics environments*
- dictionary stack - a stack used to save and restore *dictionaries*. All names in the program are *defined* and their definition placed into a dictionary
- execution stack - used to save and restore program execution location across function calls

Basic (Operand) Stack Manipulations

- constants - dynamic typing, most primitive types, array composite type

```
3               % push the integer 3 onto a stack
(hello there)   % push the string "hello there" onto a stack
3.3             % push the real number 3.3
false           % push the boolean value false onto the stack
{3 2 +}         % push code onto the stack, don't evaluate
/x              % push the name x onto the stack
x               % evaluate the name x
```

- operations on data - take values from stack, do something, push the result

```
3 4 add         % op1 op2 add --> op1+op2, e.g.,  3 + 4
5 2 sub         % op1 op2 sub --> op1-op2, e.g.,  5 - 2
4 3 mul         % op1 op2 mul --> op1*op2, e.g.,  4 * 3
```

- stack manipulations

```
dup             % duplicate the top value on the stack
pop             % pop the top value from the stack
=               % pop the top value from the stack and print it
stack           % display the contents of the stack
count           % push a count of how many values are on the stack
exch            % exchange the top two stack values
4 2 roll        % move the top 2 values on the stack into the 4th
                %  stack position from the top
count -1 roll   % move the top stack value to the bottom
4 index         % copy the 4th stack value (from top) onto the top
3 index         % copy the 3rd stack value (from top) onto the top
3 copy          % copy the top three stack values onto the stack
2 copy          % copy the top two stack values onto the stack
count copy      % copy the entire stack on itself!
```

> Please refer to your PostScript cheatsheets, found from the Resouces link on the class web page, for more operations.

Let's do a few examples to get a feel for how it works. Compute 3 + 4 and leave the result on the stack

```
   3 4 add
```

Compute (3 + 4) * 2

```
   3 4 add 2 mul
```

Compute 3 + (4 * 2)

```
   3 4 2 mul add
```

Compute 3x^2 + y^4, assuming x and y are defined names.

```
   3 x x mul mul y y mul dup mul add
```

Next let's do some stack manipulations. For each of the following we will push 1 2 3 onto the stack first. What will leave 1 3 2 on the stack?

```
1 2 3 exch
```

What will leave `1 2` on the stack?

```
1 2 3 pop
```

What will leave `1 2 3 3` on the stack?

```
1 2 3 dup
```

What will leave `1 2 2 3` on the stack?

```
1 2 3 exch dup 3 2 roll
```

What will leave `1 2 3 1 2 3` on the stack?

```
1 2 3 count copy
```

What will leave `1 2 3 3 2 1` on the stack?

```
1 2 3
  dup     % duplicate the 3
  2 index % copy the 2 from the 2nd stack position
  4 index % copy the 1 from the 4th stack position
```

Basic Control Operators

PostScript has operators that perform like *control structures* in other languages. The first operator is `if`. We first have to be able to evaluate conditions.

```
  0 1 eq                 % Does 0 == 1?  Evaluates to false.
  x 1 eq                 % Does x == 1? push true on stack else push false
  x 1 lt                 % Is x < 1?
  x 4 4 mul lt           % Is x < (4 * 4) ?
  x 1 eq x 2 eq or       % Does (x == 1) or (x == 2)?
  x 2 gt not x 1 gt or   % Does (x > 2) => (x > 1)?
```

The `if` operator pops the top two values from the stack. If the 2nd to top value is `true` then it executes the top value (which has to be code). For example, let's push 3 on the stack if x == 0.

```
     x 1 eq    % first evaluate the condition
     {3}       % next push the code for the "true" branch
  if           % finally, evaluate the if
```

By convention, let's indent the branches. Let's clear the stack if x > 0.

```
  x 1 gt {clear} if
```

`ifelse` just adds a false branch. For example, let's push 3 on the stack if x == 0, else push 4.

```
     x 1 eq    % first evaluate the condition
     {3}       % next push the code for the "true" branch
     {4}       % next push the code for the "false" branch
  ifelse       % finally, evaluate the ifelse
```

Let's clear the stack if x > 0, else let's copy the stack

```
  x 1 gt {clear} {count copy} ifelse
```

Nested `ifelse`'s can be tricky. Let's assume we want to do the following.

```
 if x == 1 then push 5
 else { if x == 2 then push 6
        else push 7 }
```

In PostScript we could use the following.

```
    x 1 eq
    {5}
    {
       x 2 eq
       {6}
       {7}
    ifelse
    }
 ifelse
```

There are also looping constructs. The `repeat` operator is used for a repeat loop.

```
 % push four copies of 3 onto the stack
     4
     {3}
  repeat
 % push five copies of the top of the stack, 1
  1
     5
     {dup}
  repeat
 % Let's compute until the top of the stack exceeds 5
  1
     100
     {1 add      % add one to the top of the stack
         dup 5 gt   % is it bigger than 5
         {exit}
      if
      }
  repeat
```

The `loop` operator just keeps looping. You have to add an explicit `exit` condition.

```
 % Let's compute until the top of the stack exceeds 5
  1
     {1 add        % add one to the top of the stack
         dup 5 gt   % is it bigger than 5
         {exit}
      if
      }
  loop
```

The for operator takes an initial value, an increment, a limit, and a code array on the top the stack. It executes the code array multiple times each time with pushing a different value of the control value on the top of the stack. **Warning, the for loop is a bit tricky because it puts the loop control value on the top of the stack for each iteration; You have to remove it explicitly if that's what you want.**

```
 % Let's push 1 through 5 on the stack
      1
      1
      5
      {dup}
   for
 % Oops, stack has 1 1 2 2 3 3 4 4 5 5
 % Instead,
      1
      1
      5
      {}
   for
 % Let's push five copies of 1 onto the stack
      1
      1
      5
      {1}
   for
 % Oops, stack has 1 1 2 1 3 1 4 1 5 1
```

So "for" loops are confusing if you manipulate the stack inside the loop, unless the manipulation does not add to the stack.

The `run` operator takes a filename (string) from the top of the stack, finds the file in the filesystem and executes the contents just as if you had typed them at the command line. Be aware that the search rules for finding the file involve the current working directory. Sometimes, especially on MSWindows machines, the working directory may not be what you expect, so using a full pathname to the file may work better.

The Current Page and Path

Drawing occurs on an imaginary *page* with unlimited coordinates. The origin is the lower-left of the page when printed/displayed via a `showpage`. To draw

1. construct a path

```
newpath         % this starts a path, wiping out the previous path
                % it is optional
```

1. establish current point, draw from point to point

```
0 100 moveto    % set the current point to 0 100
100 100 lineto  % put a line from the current point to 100 100
                % current point is now 100 100
100 100 rlineto % put a line relative to the current point
```

2. complete the path to draw a path that connects at each end

```
closepath       % this completes the path by filling in the line
                % it is optional
```

3. you have control over linewidth, kind of line, color of line, fill and fill pattern

```
10  setlinewidth   % set the line width
fillfactor setgray % make it a gray pattern fill
fill               % Fill the path on the current page
```

2. stroke the path -- write it to the current page

```
stroke          % this actually puts it into the current page
```

The drawing operations usually take an x-coordinate and y-coordinate, e.g.,

```
  x-coordinate y-coordinate moveto
```

The page origin, point (0, 0), is the lower left corner of the page. Each point is 1/72 of an inch. So

```
  72 72 moveto
```

will move to the point one inch up and to the right of the origin. If you want to work in "inches" rather than points, you can do the following.

```
/inch {72 mul} def
/inches {72 mul} def
  1 inch 1 inch moveto
  7 inches 1 inch moveto
```

If you want to work in centimeters

```
/centimeter {28.3464567 mul} def
/centimeters {28.3464567 mul} def
  7 centimeters 1 centimeter moveto
```

There are three basic transformations that can be applied to the entire coordinate system, they change coordinates for future drawing operations.

- scale - shrink or enlarge the whole coordinate system

```
.5 .5  scale       % Make everything half as big
.5 1.0 scale       % Squish half as big in the x-axis only
```

- rotate - rotate entire coordinate system in a clockwise direction

```
45 rotate          % rotate by 45 degrees, clockwise
-90 rotate         % rotate by 90 degrees, counter clockwise
```

- translate - move entire coordinate system

```
200 300 translate  % move origin to 200 300
```

- transformations are cumulative!

```
200 300 translate  % move origin to 200 300
45 rotate          % rotate the coordinate system by 45 degrees
.5 .5  scale       % scale the translated, rotated coordinates
```

- can save and restore current graphics state

```
gsave
200 300 translate  % move origin to 200 300
grestore
.5 .5  scale       % Make everything half as big
```

As an example, let's draw a smiley face.

```
200 200 translate         % move origin to near center of page
% First draw the circle for the head.
newpath
  0 0 100 0 360 arc       % draw a circle of radius 100
closepath stroke
% Next let's draw the left eye
newpath
  -35 25 10 0 360 arc fill
closepath stroke
% Draw the right eye
newpath
  35 25 10 0 360 arc fill
closepath stroke
% Draw the mouth
  0 30 100 220 320 arc
stroke
```

Another useful command is `showpage`. It "shows" the current page, creating a new, blank page to draw on. Drawing text is very easy (if the PostScript interpreter can find the font). Just move to a desired location.

```
100 100 moveto
(hello there) show
```

You can change/set the font and color (refer to the on-line resources).

```
/Times-Roman findfont    % load times-roman font
12 scalefont             % Scale to a 12 point font
setfont
100 100 moveto
(hello there) show
```

You can shrink or make text large (example from U of Indiana guide to PostScript).

```
gsave
  100 100 moveto
  1 2 scale              % Make the text tall
  (Tall Text) show       % Draw it
grestore
```

Finally, you'll probably have to use the `setgray` operator.

```
0 setgray      % set the fill color to black
1 setgray      % set the fill color to white
.5 setgray     % set the fill color to gray, half black/half white
```

Dictionaries

Everything that is not a constant is a name, names are bound to meanings in a *dictionary*

```
/joe               % push the name joe onto the stack
joe                % fetch the meaning of joe from the dictionary
                   % and evaluate that meaning
```

The `def` operator defines a name. The syntax of the operator is the following.

```
name code def
```

It defines the `name` to have the meaning of the `code`. For example.

```
/joe {3 4 add} def % define joe to be {3 4 add}
                   % braces mean defer evaluation until used
joe                % evaluate 3 4 add
```

Names have dynamic scope. You can redefine names with impunity.

```
/sub {3 4 add} def % define sub to be {3 4 add}
sub                % evaluate 3 4 add
```

See if you can guess what the following examples do; but don't do these in your programs!

```
/x {1} def
/x {x} def

/x {1} def
/x {/x {1} def} def

/x {1} def
/y {x} def
/x {y} def
```

Observe that the meaning of a constant is the constant, so

```
/x {1} def
```

is really the same as

```
/x 1 def
```

We can use this fact to define a name to be whatever the top value on the stack is.

```
/x exch def
```

So if we assume the stack contains `5 6 2`, then we push the name `/x`, giving us the stack `5 6 2 /x`. The `exch` operator then exchanges the top two values yielding `5 6 /x 2`. Finally `def` pops two values and defines `/x` to have the meaning `2`, leaving the stack as `5 6`. We can explicitly establish *local* scope by creating a local dictionary and using it. The following factorial program is incorrect because it doesn't establish the correct scope for the local name.

```
% This factorial is incorrect because it doesn't do
% a local name properly
```

```
/factorial {
  /x exch def      % try to establish local name x, no good!
  0 x eq
    {1}
    {x 1 sub factorial x mul}
    ifelse
  } def
```

A solution is to establish a local dictionary on the dictionary stack. Names are resolved by looking down the dictonary stack until the name is found.

```
/factorial {
  1 dict           % create a local dict. that can have 1 entry
  begin            % push it onto the dictonary stack
    /x exch def      % establish local name x
    0 x eq
      {1}
      {x 1 sub factorial x mul}
      ifelse
  end              % pop the dictionary stack
  } def
```

A word on PostScript style

Readibility of code is important. Use indentation, names, and comments to communicate to others. Generally it is nice to not pack too much (or too little) into a single line. The rule of thumb is one "logical operation" per line. Indentation can also improve readability; indent

- inside a begin/end
- the branches of an ifelse
- inside a loop
- inside a def

Finally, comment all defined "names", and "parameters", and important steps inside a definition as you see fit. Try not to describe what the operation does (my comments in the PostScript code in these notes are different because I'm writing for a specific purpose, that is different from most programming). For example, try to avoid doing

```
pop    % pop the top of the stack
```

Consider the following factorial function

```
/factorial {
  1 dict begin /x exch def
  0 x eq {1} {x 1 sub factorial x mul} ifelse
  end
  } def
```

It lacks comments and is "squashed". Below is a more appropriate definition, stylistically.

```
% factorial computes the factorial of the top of stack,
% leaving the result on the stack.
/factorial {
   1 dict begin
      /x exch def

         0 x eq
```

```
            {1} % Base case, factorial(0) = 1
            % Recursive case, factorial(x) = factorial(x-1)*x
            {x 1 sub factorial x mul}
         ifelse

      end
   } def
```