Postscript Control Flow

CSE 413, Autumn 2005
Programming Languages

http://www.cs.washington.edu/education/courses/413/05au/

Variables

- Postscript uses dictionaries to associate a name with an object value
  » /x 3 def
    • associate value 3 with key x
  » /inch {72 mul} def
    • define function to convert from inches to points
- Postscript dictionaries are similar to Java HashMaps
  » key / value pairs

Several Dictionaries

- When the interpreter encounters a name, it searches the current dictionaries for that key
- At least three dictionaries are always present
  » user dictionary
    • writeable dictionary in local virtual memory associates names with procedures and variables for the program
  » global dictionary
    • writeable dictionary in global virtual memory
  » system dictionary
    • read-only dictionary associates keywords with built-in actions

Dictionary Stack

- References to the dictionaries are kept on the dictionary stack
- Interpreter looks up a key by searching the dictionaries from the top of stack down
  » search starts with current dictionary on top of stack
  » initially, user dictionary is top of stack
  » system dictionary is bottom of stack
  » can define and push additional user dictionaries on top
Virtual Memory

- Postscript environment includes stacks and virtual memory
- Operand stack contains simple objects (eg, integers) and references to composite objects (eg, strings, arrays)
- Virtual memory (VM) is a storage pool for the values of all composite objects

save and restore

- Simple user programs define their objects in local VM
- The save operator makes a snapshot of local VM
- The restore operator throws away the current local VM and restores the state from the last save
- Local VM with save/restore pairs is used to encapsulate information whose lifetime conforms to a hierarchical structure like a page

Defining and using a variable

- Define a variable ppi and give it a value
  » /ppi 72 def
  » push the name ppi on the operand stack as a literal
  » push the number 72 on the operand stack
  » pop both items and store in the current dictionary using ppi as the key and 72 as the value
- Use the variable's value
  » ppi 2 mul
  » find the value of ppi (72) and push it
  » push the number 2
  » pop both operands, multiply, push the result

Defining and using a procedure

- Define a procedure name and give it a value
  » /inch {72 mul} def
  » push the name inch on the operand stack as a literal
  » push mark, 72, mul on the operand stack
  » pop to the mark, create an executable array, and store in the current dictionary using inch as the key and the executable array as the value
- Use the procedure's value
  » 2 inch
  » push the number 2
  » look up the name inch, find the procedure, execute
  » push 72, pop both numbers, multiply, push the result
fm constructors are procedures

% Circle constructor.

% FM call format => Circle(radius)
% PS call format => radius Circle.Circle
% Result: Reference to a fields array with
% values set by arguments or defaults.

/Circle.Circle { 
Circle.fields.SIZEOF array % Create the array
dup Circle.fields.radius % radius field
4 -1 roll put % store radius

dup Circle.fields.grayfill 0.5 put
dup Circle.fields.graystroke 0.0 put
dup Circle.fields.linewidth 1.0 put
} def

Conditionals and loops

• There are several operators for specifying the
  flow of control in a Postscript program
• Executable arrays are a basic element for the
  control flow operators
  » the code block (executable array) is defined in-line
  » \{add 2 div\} - calculate 2-value average
  » the curly brackets defer interpretation of the code and
    force the creation of a new executable array
    (procedure) object

if, ifelse operators

• Take a boolean object and one or two executable
  arrays on the stack.
• Select and execute one of the executable arrays
  depending on the boolean value
• leaves nothing on the stack
  » the code that executes may leave something ...
  » \textit{bool proc} \texttt{if}
  » \textit{bool proc}_1 \texttt{proc}_2 \texttt{ifelse}

Boolean operators

• Comparison operators
  » eq, ne, gt, lt, ge, le

• logical operators
  » not, and, or, xor
  » true, false

\begin{verbatim}
GS>2 3 ge
false
GS<1>==
false
GS>2 2.0 eq ==
true
GS>(abc) (acc) lt ==
true
GS>[1 2 3] dup eq ==
true
GS>[1 2 3] [1 2 3] eq ==
false
\end{verbatim}
an if example

% if current point beyond right margin, do LF CR.

/chkforendofline
{currentpoint pop % discard y position
RM gt % current x > right margin?
0 lineheight neg translate % "linefeed"
LM 0 moveto % "carriage return"
} if
def

conditional.ps

repeat operator

• Repeat a procedure body n times
• \textit{n proc} repeat

\begin{verbatim}
GS>1 2 3 4 3 {pop} repeat
GS<1>==
1
GS>
\end{verbatim}

for operator

• Controls the standard indexed counting loop

\textit{initial increment limit proc for}
  » the control value is calculated
  » if greater than limit, the loop exits
  » otherwise the control value is pushed and the procedure is executed

\begin{verbatim}
GS>1 1 4 {} for
GS<4>pstack
4
3
2
1
GS<4>clear
GS>0 1 1 4 {add} for
GS<1>==
10
GS>
\end{verbatim}

loop and exit operators

• Repeat a procedure an indefinite number of times, usually until some condition is met
• The \textit{loop} operator takes a procedure and executes it until an \textit{exit} command is encountered within the procedure
• \textit{proc} loop
  » there must be an \textit{exit} encountered within the body of the procedure, or the code will loop forever
loop example

% call: radius y lineofcircles
/lineofcircles { /ypos exch def /radius exch def /xpos 0 def
{xpos pagewidth le
 { doCircle increase-x }
{ exit }
ifelse
} loop
} def

Recursion

• A loop can be set up in a program by having a procedure call itself
  » recursion must always:
    • have a base case (an exit condition)
    • make progress towards the base case during recursion

Recursion example

/fractArrow {
 gsave
 kXScale kYScale scale
 kLineWidth setlinewidth
down
doLine
depth maxdepth le
{135 rotate fractArrow -270 rotate fractArrow
} if
up
grestore
} def