CptS 355- Programming Language Design

Postscript Programming Language

Instructor: Sakire Arslan Ay

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Postscript Programming Language

- Background:
  - It is a barebones programming language
    - constants, variables, functions, loops, conditionals, etc.
  - It is a computer language for creating vector graphics.
  - It was created by Adobe Systems in 1980's.
  - It is created as a language to communicate to printers

Postscript Programming Language

- Post-fix arithmetic notation
  - Notation for writing arithmetic expressions in which the operands appear before their operators
  - Example: 1 2 +
  - There are no precedence rules to learn and parentheses are never needed
  - Also called, Reverse Polish Notation

- Other arithmetic notations:
  - Ordinary arithmetic notation. Ex: 1+2
  - Pre-fix arithmetic notation. Ex: + 1 2

Postscript Programming Language

- PostScript is based around a stack
- To evaluate (rule is):
  - For values: push them on a stack
  - For operators: combine the top stack elements and push the result onto the stack.
- Example: 1 2 + 3 * 6 -

Postscript Programming Language

- Postscript uses several stacks, but only two are of interest to us:
  - the operand stack - the primary stack used for most computation. Sometimes just called "the stack".
  - dictionary stack - a stack used to save and store dictionaries. All names in the program are defined and their definition placed into a dictionary
  - graphics stack - a stack used to save and restore graphics environments
  - execution stack - used to save and restore program execution location across function calls

Basic Elements of Postscript Programs

- Constants — numbers, strings, arrays, booleans, code, names
  - `3` `% integer 3`
  - `(This is a string)` `% the string " This is a string "`
  - `3.4` `% real number 3.4`
  - `false, true` `% boolean values false, true`
  - `[1 2 3 add]` `% array value`
  - `/x` `% name constant x`
  - `{3 2 +}` `% code block (don't evaluate )`

⇒ Constants are directly pushed onto the stack

- Variables
  - `X` `% evaluated name x`

Basic Elements of Postscript Programs (cont.)

- Operations on data
  - Arithmetic operators: pops the top 2 values from the stack, computes the result of the arithmetic operation, pushes the result onto the stack
    - `3 4 add  % op1 op2 add → op1+op2`
    - `5 2 sub  % op1 op2 sub → op1-op2`
    - `4 3 mul  % op1 op2 mul → op1*op2`
    - `4 2 div  % op1 op2 mul → op1/op2`
    - `5 2 mod  % op1 op2 mod → op1%op2`
    - `4 neg  % op1 neg → -op1`
  - Comparison operators: pops the top 2 values from the stack , compares those values, and pushes the result (which is true or false) onto the stack
    - `4 3 lt`
    - `4 3 gt`
    - `4 4 eq`

Basic Elements of Postscript Programs (cont.)

- Stack manipulation operators
  - `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 index` % copy the 4th stack value (from top) onto the top %(staring from index 0)
  - `2 copy` % copy the top 2 stack values onto the stack
  - `count copy` % copy the entire stack on itself!
  - `4 2 roll` % move the top 2 values on the stack into % the 4th stack position from the top
  - `4 -2 roll` % move the last 2 of the top 4 values to % the top of the stack
  - `count -1 roll` % move the bottom stack value to the top

Basic Elements of Postscript Programs

Names and Variables:

- In postscript we are going to differentiate between variables and names.
  - Variable
    - Example: x xy
  - Name
    - Example /x /xy

- `1 2 /x` (the name itself is pushed on the stack)

vs.

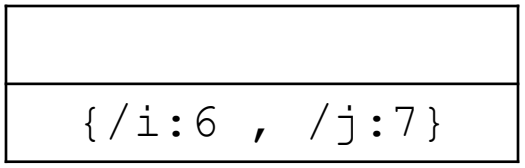
- `1 2 x` (look up the value of variable x and push that on the stack)

Basic Elements of Postscript Programs

Names and Variables (cont.):

- To give a value to a variable use the `def` operator.
- `1 2 /x 3 def` → def takes a value and name from the stack and defines a variable with the name having that value.
- Variable definitions are stored in the dictionary stack
- Example:
  - `/i 6 def 3 i add`
    `/j 7 def 3 j add`

Dictionary stack

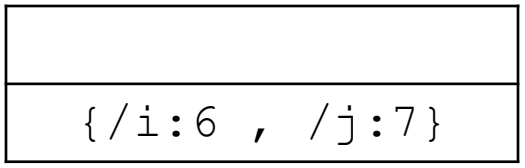


Basic Elements of Postscript Programs

Dictionary stack:

- It is a stack of dictionaries. Each time we pop and push onto/from stack we get/put a dictionary.
- **`def`** always creates or modifies a dictionary entry in the top most dictionary on the dictionary stack.
- To look up for variables: start in the topmost dictionary and keep looking until you find the variable.
- Example:
  - `/i  6  def`
    `/j  7  def`
    `i  j`

Dictionary stack



Basic Elements of Postscript Programs

Operators for dictionary stack:

1. **`dict`** operator takes the initial size of the dictionary from the stack, and puts a brand new empty dictionary on the operand stack.
2. **`begin`** operator takes a dictionary from the top of the operand stack and pushes it on the dictionary stack.
3. **`end`** operator - pop the top dictionary from the dictionary stack and throw it away.

Example:

```
/x 111 def
x
5 dict begin
/x  222 def
x
end
x
stack
```

Basic Elements of Postscript Programs

- Dictionaries can also be specified explicitly.
- You can define dictionary values and store them in variables (inside dictionary stack dictionaries)
  - use `put` and `get` commands to put and get variables to/from a dictionary.
  - **Example:**

```
/mydict 10 dict def
mydict /x 123 put
mydict /x get
pstack
```

Postscript Functions

- Define a function name and give it a value
- Example : `/toInch {12 mul} def`
  - push the name "`toInch` " on the operand stack as a literal
  - push code array `{12 mul}` on the operand stack
  - `def` will pop to the code array and the function name from the operand stack and store in the current dictionary using `toInch` as the key and the code array as the value

- Make a function call
- Example: `2 toInch`
  - push the number 2
  - look up the name `toInch`, find the code array
  - Execute code array: push 12, pop both numbers, multiply, push the result

Postscript Functions

- **How can we define local variables inside a function?**
  - We can explicitly establish a *local* scope by creating a new dictionary and using it to define local variables of our functions.
- Example :

```
/x 3 def
/f1 {/x  9 def x } def
f1
x mul
```

vs.

```
/x 3 def
/f1  { 1  dict
       begin /x  9 def  x end } def
f1
x mul
```

Boolean Operators and Conditionals (if and elseif)

- Comparison operators
  - `eq, ne, gt, lt, ge, le`
- Logical operators
  - `not, and, or, xor`
  - `true, false`
- Conditionals:
  - *`bool code-array`* `if`
  - *`bool code-array1 code-array2`* `ifelse`
  - Take a boolean object and one or two “code arrays” on the stack.
  - Select and execute one of the “code arrays” depending on the boolean value
  - leaves nothing on the stack
    - the code that executes may leave something ...

Boolean Operators and Conditionals (if and elseif)

- Example:

if:

```
/x 3 def
 x 3 eq
   {x   1    add}
if
```

ifelse:

```
/x 3 def
 x 3 eq
    {x   1    add}
    {x   1    sub}
ifelse
```

Boolean Operators and Conditionals (if and elseif)

- Example:

```
ifelse:
/x 4 def
 x 3 eq
    {x 1 add /result exch def}
    {x 4 eq
        {x 2 add /result exch def}
        {x 3 add /result exch def}
     ifelse }
ifelse
result
```

Loops – for operator

- `<init> <incr> <final> <code array>` **`for`**
  - Pops four items: `init` and `incr` and `final` and `code array`.
  - Then, for each integer starting at `init` and going by steps of `incr` until passing `final`, it pushes the current integer index onto the stack and then executes the array.
  - Examples:

```
1  1  3  {10  mul}   for

1  1  5  {dup}  for

1  1  5  {dup =}  for
```

- **Warning!** The `for` 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.

Loops – repeat operator

- `<N> <code array> repeat`
  - Pops two items: "`N`" and "`code array`".
    - `N` is an integer value, representing the repeat count of the loop.
  - **`repeat`** executes the code in "`code array`" "`N`" times.
  - Examples:

```
5 { 1 } repeat

0 5 {10 add} repeat

/n 5 def 0 n {n add} repeat
```

- **Warning!** Unlike `for` loop, repeat doesn't push the loop counter onto the stack.

Postscript – `mark` operators

- **`mark`**
  - put a mark on the stack
- **`counttomark`**
  - counts elements down to the topmost mark;
  - pushes the count onto the stack.
- **`cleartomark`**
  - Pops the elements down through the mark. Pops the mark as well.

Postscript – mark operators- example

```
GS> 1 2 3 mark 10 20
GS<6> counttomark
GS<7> pstack
2
20
10
-mark-
3
2
1
GS<2> cleartomark
GS<2> pstack
3
2
1
```

String Operators

- `<string>` **`length`**
  - pops the `<string>` object from the stack
  - and pushes the length of the string onto the stack
  - Examples: `(abcdefg) length` **`% pushes 7 onto the stack`**
    `(12345)` **`% pushes 5 onto the stack`**

String Operators

- `<string> <index>` **`get`**
  - pops (in order) the `<index>` and the `<string>` from the stack
  - gets the element at location `<index>` in `<string>`
    - `<index>` is 0-based
    - '(' and ')' characters are delimiters and are not part of the string value. The character at index "0" is the first character after '('.
  - pushes the ASCII value of the character at position index onto the stack
  - Example:

```
(CptS355) 0 get
% The operand stack will be:  67  (the ASCII value 67 corresponds to
character C)
```

String Operators

- `<string> <index> <count>` **`getinterval`**
  - pops (in order) the `<count>`, `<index>`, and `<string>` from the stack
  - gets the slice of the string from `<index>` to `<index>+<count>`
  - pushes that substring on the top of the stack
  - Examples:

```
(CptS355) 4 3 getinterval
% pushes (355) onto the stack
(CptS355) 0 4 getinterval
% pushes (CptS) onto the stack
```

String Operators

- `<string> <index> <value>` **`put`**
  - pops (in order) the `<value>`, `<index>` and `<string>` from the stack
  - puts `<value>` at `<index>` of `<string>` (`<index>` is 0-based)
  - **`put`** won't push the updated string back onto the stack. In order to save the string reference, you need to either duplicate the reference with `dup` operator or bind the string reference to a variable (before you call `put`) .
  - Example:

```
(CptS355) dup 6 48 put
  %The operand stack will be: (CptS350)
```

String Operators

- `<string1> <index> <string2>` **`putinterval`**
  - removes (in order) the `<string2>`, `<index>`, and `<string1>` from the stack
  - replaces the section of `<string1>` with `<string2>` starting at `<index>`
  - **`putinterval`** won't push the updated string back onto the stack. In order to save the string reference, you need to either duplicate the reference with `dup` operator or bind the string reference to a variable (before you call `putinterval`). See examples in the following slides.

String Operators

- `<string> <seek>` **`search`**
  - removes (in order) the `<seek>` and `<string>` from the stack
  - If `<seek>` appears in `<string>`, **`search`** will:
    - split the `<string>` at `<seek>` once; and push split strings onto the opstack.
    - push `True` onto the opstack
  - If `<seek>` doesn't appears in `<string>`, **`search`** will:
    - push back `<string>` onto the opstack.
    - push `False` onto the opstack
  - **`search`** won't push the original string onto the stack if `<seek>` appears in `<string>`.

Example: `(355-322-451) (-) search`

```
%The operand stack will be: (322-451) (-) (355) true
```

Postscript Strings – examples

```
%define an string variable
GS> /myStr (abcdefg) def
GS> myStr
GS<1> dup
GS<2> pstack
(abcdefg)
(abcdefg)
%put string BCD at index 1
GS<2> myStr 1 (BCD) putinterval
GS<2> pstack
(aBCDefg)
(aBCDefg)
%get element at index 2
GS<2> 2 get
GS<2> pstack
67
(aBCDefg)
GS<2>
```

```
GS<2> pop dup pstack
(aBCDefg)
(aBCDefg)
GS<2> 0 (A) putinterval
GS<2> pstack
(ABCDefg)
GS<2> myStr
GS<2> pstack
(ABCDefg)
(ABCDefg)
```

Postscript Arrays

- Postscript arrays are 1-dimensional collections of objects
  - indexed from 0
  - objects can be of any type - integers, strings, other arrays, dictionaries, etc.
- They are similar to lists in Python and object arrays in Java
  - Postscript arrays can hold primitive elements as well as reference elements

Postscript Arrays

Array bracket operators:

- `[`
  - put a mark on the stack. The following elements are going to be scooped up in an array
- `]`
  - create an array containing all elements back to the topmost mark. The array is created on the heap in virtual memory, and an array reference is left on the stack
  - examples: `[ 1 1 1 add]` → 2-element array : [1 2]

```
/x 3 def
[ 1 2 x ] → 3-element array : [1 2 3]
```

Array Operators

- `<array>` **`length`**
  - pops the `<array>` object from the stack
  - and pushes the length of the array onto the stack
  - Examples:

```
[0 1 2 3 4] length      % pushes 5 onto the stack
[0 1 2 add 4] length    % pushes 3 onto the stack
```

Array Operators

- `<array> <index> get`
  - pops (in order) the `<index>` and the `<array>` from the stack
  - gets the element at location `<index>` in `<array>` (`<index>` is 0-based)
  - pushes that element on the top of the stack
  - Examples:

```
[0 1 2 30 4] 3 get     % pushes 30 onto the stack
[0 1 10 add 2 sub 4] 1 get    % pushes 9 onto the stack
```

Array Operators

- `<array> <index> <count>` **`getinterval`**
  - pops (in order) the `<count>`, `<index>`, and `<array>` from the stack
  - gets the slice of the array from `<index>` to `<index>+<count>`
  - pushes that subarray on the top of the stack
  - Examples:

```
[0 1 20 30 40 5] 2 3 getinterval
% pushes [20 30 40] onto the stack
[0 1 10 add 2 sub 4 5 6 7] 1  4 getinterval
% pushes [9 4 5 6] onto the stack
```

Array Operators

- `<array> <index> <value>` **`put`**
  - pops (in order) the `<value>`, `<index>` and `<array>` from the stack
  - puts `<value>` at `<index>` of `<array>` (`<index>` is 0-based)
  - **`put`** won't push the updated array back onto the stack. In order to save the array reference, you need to either duplicate the reference with `dup` operator or bind the array reference to a variable (before you call `put`) . See examples in the following slides.

Array Operators

- `<array1> <index> <array2>` **`putinterval`**
  - removes (in order) the `<array2>`, `<index>`, and `<array1>` from the stack
  - replaces the section of `<array1>` with `<array2>` starting at `<index>`
  - **`putinterval`** won't push the updated array back onto the stack. In order to save the array reference, you need to either duplicate the reference with `dup` operator or bind the array reference to a variable (before you call `putinterval`). See examples in the following slides.

Postscript Arrays – example – 1

```
GS> [1 2 3 4 5]        %create an array of length 5
GS<1> dup
GS<2> length           %calculate the length of the array
GS<2> pstack
5
[1 2 3 4 5]
GS<2> 1 sub            %subtract 1 from top value
GS<2> pstack
4
[1 2 3 4 5]
GS<2> get              %get the element at index 4
GS<1> pstack
5
```

Postscript Arrays – example – 2

```
GS> [0 1 2]                          %create an array of length 3
GS<1> 3 array                        %create an empty array of
length 3
GS<2> pstack
[null null null]
[0 1 2]
GS<2> dup 1 10 put      %set element at index 1 to 10
GS<2> dup 2 20 put      %set element at index 2 to 20
GS<2> pstack
[null 10 20]
[0 1 2]
GS<2> exch dup 2 3 put      %set element at index 2 to 3
GS<2> pstack
[0 1 3]
[null 10 20]
GS<2>
```

Note that put operator doesn't push the modified array back onto the stack. We need to first call the "dup" operator to duplicate the reference of the array and then call "put".

Postscript Arrays – example – 3

```
%define an array variable
GS> /myArray [0 1 2] def
GS> myArray
GS<1> dup
GS<2> pstack
[0 1 2]
[0 1 2]
%put 10 at index 1
GS<2> myArray 1 10 put
GS<2> pstack
[0 10 2]
[0 10 2]
%get element at index 2
GS<2> 2 get
GS<2> pstack
2
[0 10 2]
GS<2>
```

```
GS<2> pop dup pstack
[0 10 2]
[0 10 2]
GS<2> 2 20 put
GS<2> pstack
[0 10 20]
GS<2> myArray
GS<2> pstack
[0 10 20]
[0 10 20]
```

Postscript Arrays – example – 4

```
GS> /myArray [0 1 2 3 4 5] def
GS> myArray 2 3 getinterval
GS<1> pstack
[2 3 4]
GS<1> clear
GS> myarray 1 [10 20 30 40] putinterval
GS<2> myarray
GS<2> pstack
[0 10 20 30 40 5]
```

Array Operators

- `<array> <codearray> forall`
- Postscript defines a `forall` operator that takes an array and a procedure as operands. The procedure is performed on each member of the array.
- For example:

```
GS<> [1 2 3 4] {2 mul} forall
GS<2> pstack
8
6
4
2
GS<2> clear
GS<2> 0 [5 7 3 10] {add} forall
GS<2> pstack
25
```

Array Operators

- `<array>` **`aload`**
  - removes the array object `<array>` from the stack
  - pushes every value of `<array>` onto the `opstack`
  - pushes the `<array>` back onto the opstack

- `<array>` **`astore`**
  - removes the array object `<array>` from the stack
  - pops as many values from the opstack as the length of the `<array>`
  - stores popped values in the `<array>` (overwrites the existing values)
  - pushes the updated `<array> object` back onto the opstack

Postscript Arrays

Array example-4:

```
GS> /x 2 def
GS> [0 1 x 3]  %create an array of length 4
GS<1> pstack
[0 1 2 3]
GS<1> aload
GS<5> pstack
[0 1 2 3]
3
2
1
0
```

Postscript Arrays

Array example-5:

```
GS> 1 2 3 4 5 6
GS><6> [0 0 0 0]      %create an array of length 4
GS<7> astore
GS<3> pstack
[3 4 5 6]
2
1
GS<3> pop
GS<2> 2 array astore
GS<1> pstack
[1 2]
```