String **C**omputation Progr**am**

Project Proposal

Scott Pender
scp2135@columbia.edu

COMS4115 Fall 2012

9/26/2012

Project proposal for the COMS4115 term project: Explain what problem the language solves & how it is used; describe an interesting representative program in your language; provide examples of its syntax

Mission

Design and implement a custom language whose compiler is built using O'Caml. The language should be able to solve a problem by accepting a combination of commands, algorithms, and data, perform appropriate computation, and then return meaningful output. Additionally, this goal should be achieved using a minimalistic base language.

Language Description

String **C**omputation Progr**am** (**scam**) is a small, yet powerful compiled language created for the sole purpose of text processing. Using a minimal character set, scam allows a program to interpret and process string tokens. Adapting a hand-chosen blend of features from PERL, Python, and awk scripting languages, scam uses shorthand notation to denote relationships between strings. Key components of the scam language include variables, arrays, functions, control statements, strings, and integers.

Simply put, scam may be used to compute algorithms on string data. The language itself does not include any algorithms to process this data beyond Boolean comparison operators. Rather than include processing rules in the language itself, scam provides the ability to access included data structures and their respective attributes so that you as the developer can create your own rules.

Syntax

Among the most compelling features of scam is the ability to call functions using postfix notation. For example, the user-defined function to test for equality, '=' might accept two arguments X and Y and return the strings "yes" or "no" To call this function, a program would include a line similar to that listed below.

```
@: "Does X=Y? " X Y =   →   #Does X=Y? no
```

This line of code combines three main tasks into one command. First, the compiler parses from the right and sees the '=' function label. Since the function takes two values, the next two space-delimited variables to the left are accepted to the function and a "yes" or "no" string is returned. As the parser continues to the left it concatenates the two string values. When it reaches the colon, it reads the token to the left to see what to do with the data. The lone '@' to the left of the colon tells the compiler to send the value to the right of the colon to the system console. The concatenated values are printed to the console without quotes.

For a more detailed description of language syntax and behavior, an alpha version of the scam reference manual can be found in ***APPENDIX A***.

Sample Program

A sample of a small program in scam is provided below.

```
? This is a program to compare files called "diff"
? The program is called using the following command:
? "scam diff filename1 filename2"  (without quotes)

MAIN:
      data1() : @1
      data2() : @2
      data1() data2() DIFF
      @: "Diff test complete"

DIFF: file1() file2()
      lineNum: (0)
      file1Unfinished: (1)
      file2Unfinished: (1)
      WHILE: file1Unfinished file2Unfinished OR
            IF: lineNum (file1()) >
                  file1Unfinished: (0)
                  @: "DIFF: Line " lineNum "of " @2 " does not exist in " @1
            IF: lineNum (file2()) >
                  file2Unfinished: (0)
                  @: "DIFF: Line " lineNum " of " @1 " does not exist in " @2
            IF: file1Unfinished file2Unfinished AND
                  IF: file1(lineNum) file2(lineNum) !=
                        COMPARE_LINES: file1(lineNum) file2(lineNum) lineNum
            lineNum: lineNum (1) +

COMPARE_LINES: line1 line2 lineNum
      charNum = (0)
      testFinished: (0)
      WHILE: testFinished !
            IF: [line1(charNum) line2(charNum) =] !
                  @: "DIFF: Character " charNum " of line " lineNum " does
                  not match"
            IF: [charNum (line1) >=] [charNum (line2) >=] OR
                  testFinished: (1)
            charNum: charNum (1) +
```

APPENDIX A: Language Reference Manual (alpha)

Parsing rules

- Right-first traversal

Token properties

- @ external source/destination
    - @./filename.txt
    - @ no following symbols means console
    - @1 @2 @3 etc. represent command-line arguments
- : define something or set left to value of right
- (integers)
    - Integer numbers are always read as strings except when surrounded by ()
    - Mathematical functions + and – are defined only for use on integers, using postfix notation.
- " strings
    - No need to escape characters - everything is valid until the closing " except for string formatting characters
    - String formatting characters
        - ~N for new line
        - ~T for tab character
    - Implicit concatenation of everything but entire arrays and numbers (ignore these)
    - All null values or missing parameters default to ""
    - A string follows the same rules as an array
        - Sub-array
            - "myString"(0:2) references the first 3 characters in a string
        - Array cell
            - "myString"(0) references the first char in a string
        - Length
            - ("mystring") returns (8)
    - Single character string ,"A"
        - ("A") if string length = 1, return the ascii decimal value of the character
- [] Group expressions together
- ? Single-line comments

Input/Output

- Output
    - Output X to console (concatenates all string values)
        - @: X
    - Output to file
        - @filepath: "text to print to file"
        - @"C:/my folder/textfile": "text to print to file"
- Input
    - From console
        - X(): @
    - From file
        - X(): @filepath

Functions, arrays, & variables

- Arrays
  - X() array of undefined size. Used both to define and reference the array.
  - X(3) references cell 4 of an array - 0 based
  - X(0:2) represents the sub-array containing first 3 cells
  - M(): X Y Z … create an array where M(0)=X, M(1)=Y, & M(2)=Z
  - Index order must be sequential and starts at 0
  - X(indexVal) gets cell of index value , previously defined as a variable. If undefined use return “”.
  - (X(12)) gets the length -> (13)
  - Arrays may not store numbers. Strings only!
  - No space between array name and ()
- Functions
  - myLabel: X Y Z
    Impl of myLabel
    myLabel: "returnValue"
  - Declaration of a function 'myLabel' that accepts parameters X,Y, and Z. Note that tab is used as the delimiter for the inside of a function. If there is no tab-delimited line following the method signature, the definition will be parsed as a variable definition.
  - Within function def, the function label may be set like a variable to indicate a return value. Default return value is "". Since this is set like a variable, it can be used to make recursive calls
  - X Y Z myLabel ...call the function
    - Y Z myLabel ...Y is read as X, Z is read as Y
    - Counts left looking for the exact number of args it accepts. Will stop early if there are extras. If less than expected it will stop once a non-quote symbol or eol is hit and use only tokens before (to the right of) the symbol.
  - Implicit return from function (can return string or integer)
    - @: X Y Z myLabel
- Variables
  - X: 2
    - Define variable X as string value of 2
  - X: (2)
    - Define variable X as integer value of 2
  - XY:"strinng"
    - Define variable XY as string "strinng"
  - (X)
    - Length of value in X. default is 0.
  - All global (outside of functions) vars always accessible (except if a local var within a procedure hides it)

Defined Functions

- Boolean evaluation
  - =, <, >, <=, >= will return (0) or (1)
  - OR & AND
  - ! Not

Control Statements

- Pre-defined labels using the same notation as a standard function.
- The right-hand side of the evaluation will accept (1) as true (unless ! is used). All other values are false.
- IF: X
  Do a bunch of stuff
- WHILE: X
  Do a bunch of stuff
- FOREACH: cellX myArr()
  Do a bunch of stuff with cellX
  - Only applies to arrays or strings