B E C Y

Tabular – data manipulation language

Reference Manual

Authors:

Bong Koh *bdk2109@columbia.edu*
Eunchul Bae *eb2263@columbia.edu*
Cesar Vichdo *cv2139@columbia.edu*
Yongju Bang *yb2149@columbia.edu*

1. Introduction

The language Becy is a scripting language to manipulate and present tabular data. The ultimate goal of such a language is to provide a simple and flexible scripting language tailored for numerical manipulations of lists of data. Since the goal of Becy is efficiency, it includes many shortcut constructs including implicitly typed variables, abbreviated conditional statement syntax, shortened loop manipulation syntax, and included basic mathematical functions. Data files formatted in tab-separated values will have inherent access included within the language. Upon processing of a data file, a program in the language will also be able to generate a formatted HTML file for use with a typical browser through included presentation functions.

This manual describes the syntax and the semantics of Becy. The language constructs will be explained by the usual extended BNF(EBNF) using special notation. The "?" and "[]" means an optional item. The '+' and '*' means repeat 1 or more times, and 0 or more times, respectively. Non-terminals are shown in italics, keywords are shown in bold, and other terminal symbols are enclosed in single quotes like '.' and ';'.

2 - Lexical Conventions

2.1 comments

Comments starts with the characters /* and ends with the characters */.

2.2 Identifiers (Names)

Identifiers in Becy can be any string of letters, digits, and underscores, not beginning with a digit. This coincides with the definition of identifiers in most languages. Becy is case-sensitive. Upper and lower case letters are considered differently.

2.3 Keywords

The following keywords are reserved and cannot be used as identifiers:

false	function	local
return	true	table

2.4 Constants

2.4.1 Numerical constants

Numerical constant is a sequence of digits. It may be written with an optional fraction part. Examples of valid numerical constants are

3 3.0 3.1416

2.4.2 Character constants

Character constants are 1 or 2 characters enclosed in single quotes " ' ". It may contain the number and single symbol. Like C, 2 characters with backslash "\" within matched single quotes are considered special flag.

\n --- newline

\r --- carriage return
\t --- horizontal tab
\v --- vertical tab

2.5 Literal Strings

String will be defined as a sequence of characters confined within matched double quotes. Literals may contain nested " "" " pairs. It may run for several lines and don't interpret any special flag.

2.6 Other tokens

```
+      -      *      /      ^        =    ||
!=     <=     >=     <      >       ==
(      )      {      }      &&       %
;      :      ,      .
```

3. Values and types.

There are four basic types in Becy: boolean, number, string, and table. Variables will not have explicit type declaration since an indicator such as "@"and "$" will be used preceding any variable. (see. 4.1)

3.1 Boolean

Boolean is the type of the values false and true. In Becy, both null and false make a condition false.

3.2 Number

Number basically represents real numbers. There is no explicit type declaration. (see. 4.1)

3.4 String

String represents a sequence of characters. There is no explicit type declaration. (see. 4.1)

3.5 Table

The type table implements associative lists. The lists can be indexed by numbers with special symbol as well as any evaluated values (see. 4.2). Like indices, the table can contain values of all type. The value of a table field can be of any type. Thus tables may also carry not only built-in methods but also user-defined methods.

Tables are the sole data structuring mechanism in Becy; they may be used to represent ordinary arrays, symbol tables, sets, and some sort of records, etc. To represent records, Becy uses the field name as well as special symbols for an index. There is a convenient ways to create tables in Becy.

#tablename_(startrow TO endrow. startcolumn TO endcolumn)

For example :

#table1_(x1TOx100.y1TOy50)
This will create a table with 50 columns and 100 rows.

4. Variables

4.1 Basic types of variables

Variables are places that stores values. Becy will utilize two basic types of variables, numbers and strings. However, it will not have explicit type declaration since it will use an indicator character preceding any variable. For numerical variables, the character will be the '$' character and '@' character will be used for string variables. For example, to declare and assign a new string variable and a new numerical variable the values "hello" and 1234, one would write:

@string_var1 = "hello";
$num_var1 = 1234;

Also for table type, '#' character will be used. For example :

#table1_x1.y2 = 10

it means the value of the first column and the 2nd row is 10.

4.2 User-defined variables

All user-defined variables' identifiers must begin with a letter though as variables with numerical identifiers have a special purpose in Becy. These variables will be reserved for built-in "shortcut" variables for columns and rows of data. A text file with tab-separated data will naturally be organized into columns and rows lending easily to a matrix structure. The entire column can be represented through the appropriate indicator character and the number of the column, indexed from the leftmost column starting at 1.
For example, if the dataset contained:

	Column 1	***Column 2***	***Column 3***
Row1	Jimbob	44	123
Row2	Jane	32	117

$y1 refers to all the data in "Column 1" => "Jimbob", "Jane"
$x1 refers to all the data in "Row 1" => 44, 123

Becy would also provide immediate access to specific entries in the table through a dot notation:
$y1.x1 refers to "Jimbob".

5. Statements

Becy supports almost conventional set of statements but in a special form. Basically, the statements are executed in sequence. This set includes assignment, control structures, looping structure, function calls, and table constructors.

5.1 Statement block

Statements can be enclosed by curly brackets. Nested statements are also allowed with matched curly brackets. Any statements within curly brackets are grouped and run together in a sequential manner.

5.2 Expression statement

Most statements are expression statements, which have the form

expression ;

Usually, expression statements are assignments or function calls

5.2.1 Assignment

Assignment can take only one form using the symbol "=". This assignment is a direct assignment in the form of:

left-value expression = *right-value express*;

Left-value must always be a variable whereas the right side can be a constant, variable, the result of an operation or any combination of them. It is necessary to emphasize that the assignation operation always takes place from right to left and never at the inverse.

5.2.2 Function call

Functions in Becy require explicitly identified invoking list objects. (see 6)

5.3 Compound statement

The compound statement is provided in Becy as following form:

compound-statement: {statement-list}
statement-list: statement statement-list

5.4 Conditional statement

Becy will also contain support for conditionals only in this manner;

(*expression*) ? *statement1* : *statement2* ;

If true, then first statement is evaluated, otherwise second statement.

5.5 Loop statement

Like any other languages, Becy will provide a looping structure. However, since lists are a vital component of Becy's design, its looping syntax reflects the necessity for a convenient method to use lists;

listname.func_name(*startrow* TO *endrow*);

5.6 Return statement

Return statement can be used within function body. It returns either a variable's value or a number including null value to the caller. It is mandatory that the statement is ended with a semicolon. A basic return statement as follows:

return ;
return (*expression*) ;

6. Functions

Functions in Becy require explicitly identified invoking list objects. Thus, the general syntax for using a function follows this format:

list_name.func_name (parameter_list);

6.1 Included mathematical functions

Basic mathematical functions on groups of data will be supported by Becy;

list_name.average($column_number);
list_name.sd($column_number);
list_name.median($column_number);
list_name.var($column_number);
list_name.sum($column_number);
list_name.max($column_number);
list_name.min($column_number);

list_name.round($column_number, $row_number);
list_name.sqrt($column_number, $row_number);
list_name.trunc($column_number, $row_number);
list_name.fact($column_number, $row_number);

6.2 Included display functions

list_name.print();
list_name.table(); …

6.3 Included data definition, and import functions

Basic data definition and import functions on groups of data such as defining data sources loading a data source will be supported by Becy. All such functions follow the typical function call syntax but will use curly braces '{' and '}' instead of parenthesis:

Definition:
list_name{
data_col1_row1 data_col2_row1,

```
                data_col1_row2          data_col2_row2
};
```

Import:

```
    list_name{include “file_name.txt”};
```

6.4 Manipulation functions.

Adding and removing from the source will provided as follows:

Addition:

```
    list_name.push{bob 10 3, kim 6 2};
    list_name.push{include “file_name.txt”};
```

Deletion:

```
    list_name.remove_row{row1, row2 . . . rown};
    list_name.remove_col{ col1, col2 . . . coln };
```

6.5 User-defined Functions

Declaration:

```
function func_name {

    statement1;
 …
    statementn;
    [return value];
}
```

Invocation:

```
listname.func1(); listname.func2(); …
```

7. Expressions

7.1 Arithmetic Operators

Becy supports the usual arithmetic operators: Unary arithmetic operators “+” and “-” can be prefixed to an expression. The binary “+” (addition), “-” (subtraction), “*” (multiplication), “/” (division), and “^” (exponentiation); If the operands are numbers, operations except exponentiation have the usual meaning.

7.2 Relational Operators

The relational operators in Becy are

```
    ==    !=    <    >    <=    >=
```

These operators always result in false or true. Equality “==” first compares the type of its operands. If the types are different, then the result is false. Otherwise, the values of the operands are compared. Numbers and strings are compared in the

usual way. These operations can be applied to tables only if both tables have same numbers of fields, and 'corresponding' fields have the same type.

The operator "!=" is exactly the negation of equality "==".
The order operators work as follows. If both arguments are numbers, then they are compared as such. Otherwise, if both arguments are strings, then their values are compared according to the current locale.

7.3 Logical Operators

The logical operators in Becy are

```
&&        ||
```

Like the control structures all logical operators consider both false and null as false and anything else as true. The conjunction operator "&&"returns its first argument if this value is false or null; otherwise, returns its second argument. The disjunction operator "||" returns its first argument if this value is different from null and false; otherwise, "||" returns its second argument. Both "&&" and "||"use short-cut evaluation, that is, the second operand is evaluated only if necessary.

7.4 Precedence and Associative.

Operator precedence in Becy follows the below, from lower to higher priority:

```
||
&&
<          >          <=          >=          !=          ==
+          -
*          /
^
```

You can use parentheses to change the precedence in an expression.
Exponentiation "^" operators are right associative. All other binary operators are left associative.

7.5 Table constructor

#tablename_(startrow TO endrow. startcolumn TO endcolumn)

8. Scope rules

8.1 Lexical scope

Becy is a lexically scoped language. The scope of variables begins at the first statement after their declaration and lasts until the end of the innermost block that includes the declaration.

It is an error to re-declare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

8.2 Example of lexical scope

```
$number = 10;
@name = 'lucas';

    function age {
                $number=5;
                return $number;
      }

 $number2=$numer+10;
```

9. Examples

NAME	GRADE1	GRADE2
jean	**10**	**2**
peter	**8**	**1**
josh	**9**	**3**
amber	**9**	**6**

```
/*
This is an example of code
*/

/* Adding values to the list*/
list{jean 10 2, peter 8 1, josh 9 3,
    amber 9 6}

/* Sum of grade1 */
$sum = list.sum($2.2TO5);

/* Average of grade 1 */
$avg1 = list.average($2.2TO5)

/* Average of grade 2 */
$avg2 = list.average($3.2TO5)

/* conditional expresion*/
```

```
$max_avg = ($avg1>$avg2)?$avg1:$avg2

/* function example*/
function max($n1,$n2){
  $max=$n1;
  ($avg1<$avg2)?$max=$n2;
 return $max;
}

$max_avg = list.max($avg1,$avg2);
```

10. Grammar

```
/*
  grammar.g : the lexer and the parser, in ANTLR grammar.
*/

class BecyAntlrLexer extends Lexer;

options{
     testLiterals = false;
     k = 2;
}

protected
ALPHA    : 'a'..'z' | 'A'..'Z' | '_' ;

protected
DIGIT     :     '0'..'9';

WS        : (' ' | '\t')+              { $setType(Token.SKIP); }
          ;

NL        : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
              { $setType(Token.SKIP); newline(); }
          ;

COMMENT : ( "/*" (
                          options {greedy=false;} :
                            (NL)
                          | ~( '\n' | '\r' )
                       )* "*/"
              | "//" (~( '\n' | '\r' ))* (NL)
            )                                 { $setType(Token.SKIP); }
          ;

LPAREN   : '(';
```

```
RPAREN   : ')';
MULT     : '*';
PLUS     : '+';
MINUS    : '-';
DIV      : '/';
MOD      : '%';
SEMI     : ';';
LBRACE   : '{';
RBRACE   : '}';
ASGN     : '=';
COMMA    : ',';
GE       : ">=";
LE       : "<=";
GT       : '>';
LT       : '<';
EQ       : "==";
AND      : "&&";
OR       : "||";
NEQ      : "!=";
COLON    : ':';
AT       : '@';
DOLLAR   : '$';
DOT      : '.';
QUES     : '?';

ID  options { testLiterals = true; }
          : ALPHA (ALPHA|DIGIT)*
          ;

/* NUMBER example:
  1, 1., 1.1
*/

NUMBER  : (DIGIT)+ ('.' (DIGIT)*)? ;

STRING  : ""!
                  (   ~(""  | '\n')
                    | (""!"")
                   )*
              ""!
          ;


class BecyAntlrParser extends Parser;

options{
    k = 2;
    buildAST = true;
}
```

```
program
        : ( statement | func_def | data_def )* EOF!
          ;

statement
          : assignment
          | condition
          | func_call
          | return
          ;

assignment
          : variable  ASGN^ arith_expr SEMI!
          | variable ASGN^ variable SEMI!
          | variable ASGN^ NUMBER SEMI!
          | variable ASGN^ STRING SEMI!
          ;

variable
          : DOLLAR! ID
          | AT! ID
          ;

arith_expr
          : LPAREN! arith_term PLUS^ arith_term RPAREN!
          | LPAREN! arith_term MINUS^ arith_term RPAREN!
          | arith_term
          ;

arith_term
          : LPAREN! factor MULT^ factor RPAREN!
          | LPAREN! factor DIV^ factor RPAREN!
          | factor
          ;

factor
          : (PLUS^ | MINUS^) NUMBER
          | func_call
          | variable
          ;

condition
        : LPAREN! bool_expr RPAREN! QUES^ statement COLON! statement SEMI!
          ;

bool_expr
          : LPAREN! logic_term OR^ logic_term RPAREN!
          | logic_term
```

```
            ;

logic_term
            : LPAREN! logic_factor AND^ logic_factor RPAREN!
            | logic_factor
            ;

logic_factor
            : arith_expr (GE^ | LE^ | GT^ | LT^ | EQ^ | NEQ^) arith_expr
            | 'true'
            | 'false'
            ;

func_call
          : ID DOT! ID LPAREN! expr_list RPAREN! SEMI!
            | ID DOT! ID LBRACE! data RBRACE! SEMI!
            ;

expr_list
            : ( arith_expr ( COMMA! arith_expr )* )?
            ;

data
            : ( ID^ | NUMBER^ )+
            | "include"^ STRING
            ;

data_dec
            : ID^ LBRACE! data RBRACE! SEMI!
            ;

func_dec
            : "function"^ ID LBRACE! statement* return? RBRACE! SEMI!
            ;

return
           : "return"^ (expression)? SEMI!
            ;
```