Multi-style form data postprocessing language architecture

Definition document

I Motivation

In some cases there are several instances of documents that contain similar information, but record this information in extremely various ways. It is therefore necessary to process the resultant scan data differently for each type of document, in order to retrieve the pertinent information from all forms. It would be best if there were a layer of abstraction between the exact stored data and the required information from the form, thus allowing data to be captured from any form using the same engine. To this end we will discuss a data processing language which will be used to define a protocol to perform data extraction from form data files and present it in a homogenous way.

II Generalized Process Architecture

Because this system will be used for any number of different form families (each with different data needs) these macros cannot themselves be predefined; thus we will need a way of describing the set of macros for a form family. We will need a way of describing the implementation of these macros in a standard way, and we will also need a way to describe how the retrieved data is stored. This will be accomplished by creating a language with several segments, each corresponding to a type of configuration file which performs the tasks listed above.

III Specifics

1-Relational Operators

The forms to be processed store information in several different styles on a given page. The bulk of the data follows a very regular format, appearing as rows and columns; column-wise data is of the same type but applies to different instances, row-wise the data is heterogeneous but refers to a single instance or grouping.

This sort of grid-like grouping gives rise to the interpretation of the mesh of rows and columns as a matrix which can be traversed by relational movement commands. Thus we define the operators:

```
        Prev
Left            Right
        Next
```

which can be used to move through the matrix. These operators will require a sense of current position in order to be effective, so a cursor that keeps track of the current position (column-wise, row-wise) in the matrix is also defined.

These atomic movement operations will form the foundation for the capture of data off of a form because they can be used to represent a series of steps performed in order to reach a particular matrix position. To further facilitate a the use of these operators parameters should be defined which will control specifically how movement is performed.

Given any one of the relational operators, we will overload it with the following parameter types:

Next(*n*) where *n* is an integer; indicating the operation should be repeated *n* times.

Next(*"textstring"*, *p, e*) where *textstring* is a literal string; the Next operation is repeated until the cursor is over a data element matching (with up to *e* percent error) *textstring*. *e* is an optional parameter. **For Next and Prev only:** In the case where no matching string is found in the current cluster, search continues into the next cluster (skipping over clusters of types different than the initial cluster) and at the same column position

(determined by offset from the left, in the case of different numbers of columns) if *p* is set to 1. If *p* is set to 0 or omitted, the search is constrained to the current block.
For all operands, if no match is found, the cursor is returned to the starting position.

Next(BLOCKEND) will repeat the Next operation until the end of the block is reached, leaving the cursor in the current column and at the last row of the block. This operand type applies only to Next and Prev.

Next(BLOCKSTART) will repeat the Next operation until the end of the block is reached, leaving the cursor in the current column and at the first row of the next (or previous) block. This operand type applies only to Next and Prev.

Left(MAX) and Right(MAX) will seek to the left and right extents of a matrix, in the current row.

The atomic operators will allow movement between data clusters. The atomic operators should also return boolean style Int values representing positional information – 0 if the action caused (or would cause) the cursor to move beyond the bounds of the current block, 1 if the no boundaries were crossed..

The input data will be formatted with headers which will define the start of blocks. These headers will provide information about the number of columns and will also serve as a means to identify different blocks. These headers should be used to determine the action of the atomic operators. Next and Prev especially, when moving to another block, will continue in the same column (measured by offset from the left) as it was in before the command was executed.

A note about string matching error: the form scanning process is an imperfect one and data is often captured incompletely or incorrectly. Thus it will be necessary to take this fact into consideration when processing recognizable fields. Since the amount of error is a tradeoff (field accuracy vs expected scan quality) it needs to be (in general) globally modifiable without having to change the parameters of a lot of Next() commands. Thus the command:

```
SetDfltFldErrPercent(n);
```

will set the default number of allowed error characters. Realize it is still possible to override this global default by providing the character error parameter to the operator.

We will need to be able to detect what data cluster we are in; to this end the operators:

```
GetBlockName(varName);
```

will be defined. GetBlockName will retrieve the current cluster name (from the header) and store it in *varName*.

2-Form layout and Macro Constructs

The forms that are to be processed with the relational operators can be logically viewed not only as a homogenous matrix, but also as several groupings of related information. To use the example of an insurance form which contains a header of the patients described in the document, and a body containing groups of records for each patient, delineated by some marker, we can see that the header forms one group, each patient entry in the body forms another, and the body itself forms a third group.

The application for this form processing system will be to glean required information out of different areas of the form, and a lot of the data will be taken from the same place in each group. Thus it would be useful to combine several relational operations into a macro which could seek not just between matrix cells, but between groups. In fact, because the information in a group is so regular, and because for any set of forms which serve the same purpose there is likely to be a great deal of similarity in structure, we could define standard macro names, and fill their procedures with form-specific steps to get the data from its raw format.

First, the language format for defining a macro construct should be described. This is perhaps best done by example:

```
Macro SeekToHeader
{
       Right(3);
       Next("patient detail");
       Next(5);
       Left(3);
       Done;
}
```

The format of this macro is fairly clear:

```
Macro macroName
{
       atomic operators;
}
```

A semicolon follows all operators. Whitespace is acceptable to break up long lines. Comments can be included with an // pair; any text on the same line and following the // pair is ignored.

```
       Next("Subtotal");          // This text is commented out
```

At this point it may also be useful to define some other language terms to aid in the construction of macros:

Repeat(*n*) {*operations*}	: repeats the operations within the curly brackets *n* times

If ("*textstring*")
{ *operations* }
Else { *operations* }
	: executes the statements following Then if the value at the cursor
	: position is equal to *textstring*, or those following Else if not.

Until("*textstring*")
{ *operations* }
	: executes the statements in the brackets while the value of the cell at the
	: current cursor position is not equal to *textstring*

Done;	: exits the macro
Home;	: Moves the cursor back to the home position (by default the beginning : of the file.
SetHome;	: Sets the home position to the current cursor position.
PushHome;	: Saves the current Home position on a stack
PopHome;	: Sets the Home position to the top value on the stack (if the stack is : empty, does nothing)

Permutations of the If and Until loops should also be implemented, where instead of a literal string constant a variable can be substituted, or as a third case, where an equality expression is used in place of the string constant, and the cursor value is not compared. A special identifier representing the end of file should also be able to be used; its state is false while there is more data to read, and takes the form:

```
Until (EOF)
{
       instructions…
}
```

And of course, variable definitions and logical operators:

String *stringName*	: holds a string

Int *intName*	: holds an integer value
==	: set left hand variable to the value of the right hand side (either a variable or constant (String)
=, !=	: Equal and not Equal logical string comparison operators.
!, &&, ||	: logical not, and, or for If and Until expressions
+, -, *, /	: arithmetic instructions (for use on Ints)
+	: concatenation operator for use on Strings.

Constants can be defined with quotes for Strings, as:

```
        "some string constant"
```

Substrings can be referred to with the following syntax:

StringName.Left(*n*);	: The first *n* characters, starting from the left hand side
StringName.Right(*n*);	: The first *n* characters, starting from the right hand side

It will be necessary to be able to recognize if a string has a certain "type" of data. Thus we will define the string operators:

IsText	(alphabetic characters and standalone numerals)
IsNumeric	(only numeric characters)
IsAlpha	(only alphabetic characters and ".")
IsDate	04/22/2xxx, 04/22/xx
IsDollar	124.45
IsSSN	123-45-8594
IsBlank	(contains no characters)

These operators will return true if the string matches the listed format and will be called with the syntax:

StringName. IsAlpha

and will be most useful when used within an If or Until statement. Variables defined in a macro are local to that macro and any macro called by the macro containing the definition. No variable name may collide with any other variable name in the namespace; however when a macro exits the variables defined by it are removed from the namespace.
The language needs some mechanism for retrieving data, so we define the operator:

```
        Get(varName);
```

which will load the value at the current cursor position into *varName.*

3-Data loading and Output formatting protocols

The system architecture needs to be able to store information in order to process it. To this end a set of hierarchical predefined data items will be defined. Thus language for data constructs must also be included. Considering overall system architecture for a moment: there will already be a macro definition file for each type of form to be processed. We will create yet another definition file that will determine the data constructs which will be available to the macros, and which will be able to contain whatever data we require from the forms. The data structure definition syntax will be as follows:

```
Define Struct StructTypeName
        {
                Structs, Strings, Numbers, pointers, Lists
        }
```

will create a type of structure, which can be instantiated by the statement:

```
        StructTypeName       StructName;
```

Lists will be useful to store sets of data:

```
        List   ListName;
```

A list will be implemented as a linked list with at least Next, Prev, First, and Last operations, and a new item (equal to a copy of *varName*) will be added into the list at the current list pointer position by using the statement:

```
        ListName = varName;
```

`The Get operator will obtain the value of the current member of the list:`

```
        String a;

        ListName.Get(a);
```

Variables within other variables will be referred to with the usual dotted object-oriented conventions, such as:

```
        Header.RecordList.First.PatientName
```

When defining these data structure terms, we need to also consider the format in which they are output. To this end the following operators are defined:

```
        Print("constant string");
        Print(varName);
```

Additionally, we define some special system variables which have special meanings to the system and can be used for, among other things, data output:

%Date%	: Date in mm/dd/yyyy format (All dates are relative to the local system time)
%Day%	: Day in Mon, Tue, Wed format
%Time%	: Time in 24hr, 16:24:02 format
%FormFamily%	: The name of the form family being processed
%FormName%	: The name of the form type being processed
%JobIdentifier%	: The Job identifier of the current job

4-System Architecture

We will use a set of files to define the operation of the data-processing engine. Their specifications are as follows:

formName.FRM files will contain the macros. The file will contain a header, a Main macro, and 0 or more other macros; the Main macro will be the function called by the data-processing engine. The header information will be made up of an entry as follows:

FORM-FAMILY *formFamily*
FORM-NAME *formName*

One of these files will be written for each type of form in the family.

*formFamily_formName_JobIdentifier.*DAT files will contain the output from processing the form data.

5-Sample .FRM file

```
FORM-FAMILY   ExplanOfBen
FORM-NAME     EmpireEOB

Define Struct SubtotalStruct
       {
              List Subtotals
       }

Macro Main
{
       SubtotalStruct SubtotalStorage;   // Used to store subtotals

       Print("%s,%s %s/nFormFamily: %s/n",%Day%,%Date%,%Time%,%FormFamily%);
       Print("FormName: %s/n"JobIdentifier: %s/n/n",%FormName%,%JobIdentifier%);

       Print("Subtotal Listing/n/n");
       SetDfltFldErr(3);
       SeekToHeader;
       SeekToBody;
       Until("Detach Check")
       {
              GetSubTotal;
              Print("Subtotal: $%s/n",SubtotalStorage.Subtotals);
       }
}
Macro  SeekToHeader
{
       Right(3);
       Next("Patient Detail");
       Next(5);
       Left(3);
       Done;
}

Macro  SeekToBody
{
       Right(3);
       Next("Service Detail");
       Next(5);
       Left(3);
       Done;
}

Macro  GetSubTotal
{
       String subtotal;
       Right(3);
       Next("Subtotal");
       Get(subtotal);
       SubtotalStorage.Subtotals = subtotal;
       Next(1);
       Left(3);
       Done;
}
```

8-Sample .DAT file

```
Mon, 01/24/2xxx
FormFamily: ExplanOfBen
FormName: EmpireEOB
JobIdentifier: 901247097234

Subtotal Listing

Subtotal: $1045.96
Subtotal: $348.00
Subtotal: $719.03
Subtotal: $34.74
Subtotal: $1988.44
```

IV Language Reference

1-.FRM file language

Header:
FORM-FAMILY *FamilyName*
FORM-NAME *FormName*

Format:
Header, followed by implementations of all macros defined in the associated .MAC file.

Main macro format:
```
Macro   Main
{
        Operations;
}
```

Macro Format:
```
Macro macroName
{
        atomic operators;
}
```

Language terms:

Prev(*param*)	:Atomic operators; *param* may be a number (repeat factor)
Next(*param*)	:or a String constant or variable (repeat until cursor value is
Left(*param*)	:equal to *param*)
Right(*param*)	

Repeat(*n*) {*operations*} : repeats the operations within the curly brackets *n* times

If ("*textstring*")	: executes the statements if the value at the cursor
{ *operations* }	: position is equal to *textstring*, or those in the next brackets if not.
{ *operations* }	

Until("*textstring*")	: executes the statements in the brackets while the value of the cell at the
{ *operations* }	: current cursor position is not equal to *textstring*

Permutations of the If and Until loops should also be implemented, where instead of a literal string constant a variable can be substituted, or as a third case, where an equality expression is used in place of the string constant, and the cursor value is not compared.

Done;	: exits the macro
Get(*varName*);	: Sets the value of *varName* to the value at the current cursor position

Print("*constant string*");
Print(*varName*);

%Date%	: Date in mm/dd/yyyy format (All dates are relative to the local system time)
%Day%	: Day in Mon, Tue, Wed format
%Time%	: Time in 24hr, 16:24:02 format
%FormFamily%	: The name of the form faily being processed
%FormName%	: The name of the form type being processed

%JobIdentifier%	: The Job identifier of the current job
==	: set left hand variable to the value of the right hand side (either a variable or constant (String or Number)
=, **!=**	: Equal and not Equal logical operators.
!, &&, ||	: logical not, and, or for If and Until expressions
+	: concatenation operator for use on Strings.
Home;	: Moves the cursor back to the home position (by default the beginning : of the file.
SetHome;	:Sets the home position to the current cursor position.
PushHome;	: Saves the curent Home position on a stack
PopHome;	: Sets the Home position to the top value on the stack (if the stack is : empty, does nothing)

Variable Instantiation:
String *StringName;*

String Type Operators:

IsText	(alphabetic characters and standalone numerals)
IsNumeric	(only numeric characters)
IsAlpha	(only alphabetic characters and ".")
IsDate	04/22/2000, 04/22/00
IsDollar	124.45
IsSSN	123-45-8594
IsBlank	(contains no characters)

These operators will return true if the string matches the listed format and will be called with the syntax:

*StringName.*IsPercent

Struct definition:
Define Struct *StructTypeName*
{
Structs, Strings, Numbers, pointers, Lists
}

Struct Instantiation:
StructTypeName StructName;

List instantiation:
List *ListName*

List assignment:
ListName = varName;

List Operators:

*ListName.***First;**	// Set the current position in the list to the first element
*ListName.****Last;***	// Set the current position to the last element
*ListName***Prev***;*	// Set the current position to the last element

```
ListName.Next;                  // Set the current position to the last element
ListName.Get(varName);          // Put the value of the current element in the variable
```