HTML Forms Data Processing Tool

Dale Clark and *Jayashree Harikumar*, Arctic Region Supercomputing Center, University Of Alaska, Fairbanks

ABSTRACT: *Online forms have extensively increased the degree of interaction between users on the World Wide Web, through HTTP. However the tools for creating forms and interpreting the resultant data are relatively primitive. At the Arctic Region Supercomputing Center (ARSC) we have developed an HTML Forms Data Processing Tool to help us process our online applications and other survey data. The interface consists of two parts, a general purpose form data preprocessor to decode embedded constraints to the submitted form data and an administrative tool to process the resultant data. The data preprocessor is designed to return to the user an annotated form with error advice in the event of an error, and the administrative tool is designed to store relevant user data to the database.*

Introduction

The ARSC Web server is established to serve as a single source of information to the users. One particular service that ARSC provides through its Web server is our online account application form. The ease of use of these online application forms has made the forms popular with ARSC account applicants but introduced two problems in its wake: (1) the need to enter the submitted data into ARSC's database and (2) interpret erroneous or incomplete form data and give the user a chance to correct the data at the time of submission.

CGI applications handle submitted form data with some preprocessing capabilities but do not inform the applicant of errors at the time of submission. To address the above issues, the HTML Forms Data Processing Tool or **FPT** (Form Processing Tool) was created. FPT can (1) encode contraints for the form creator, (2) decode constraints applied to designated form fields to inform users and form designers of missing or erroneous data, and (3) enter user data into the ARSC database.

Design Considerations

Requirements

Special purpose applications can be written to target special fields in individual forms but this type of ad hoc approach involves duplication of effort and requires reworking of code whenever the targeted form is modified meaning the duplication of substantial amounts of code. To avoid duplication of effort FPT is designed to meet the following requirements:

- preprocess the output from any arbitrary form
- check forms for "errors of form'
- check forms for 'errors of content"
- inform users of errors via the same form with errors flagged below the actual data submitted
- provide a mechanism to resubmit data forcibly (essential if the user considers the data to be correct and the error message erroneous)
- return the data submitted by the user to the form owner in a machine readable format (email) and human readable reconstructed form
- inform the form owner of possible errors in user data
- inform the form owner of possible errors from preprocessing the data
- enter the data into a database

Additionally it was required that the tool do no harm to the owner's environment, either from malice or misadventure. To extend the tool's practical utility, it was suggested that the tool be implemented in a highly portable language, use a Graphical User Interface (GUI) and provide a facility for creating new forms thus allowing the user of this tool to specify, at the time of creation of each new form field, the constraints to be placed upon it.

Design

FPT is a three part tool. Tool 1 creates HTML forms and embeds within them constraints against each form input field. Tool 2 collects user data, applies the constraints embedded by the tool 1 to the data, and communicates the results to the form user and the form owner. Tool 3 transfers accepted form data into a database. Tool 1 by definition is a HTML editor and tool 3 is a site-dependent tool designed to meet the needs of the database employed to store the data. At ARSC tool 3 is written using AppleScript.

The primary design decision to error-check on arbitrary form data involved a detailed design of tool 2. FPT needs to receive both its data and its constraints externally, constructing a virtual special purpose preprocessor based upon the expressions contained within the constraints supplied to it. To do so data and constraints need to be coordinated. Data is entered into a form by a person sitting at a terminal and the data has no existence outside of the brower's memory. Constraints, on the other hand, are carefully created and stored in a non-volatile manner. No assumption is made about the form user with respect to these constraints. As only name=value pairs are returned to FPT, the link to the constraints appear as such a pair, in a hidden form field. To permit the user creating the form and FPT have access to the constraints, the constraints are embedded in the form. This reliably associates the data with the constraints and guarantees their mutual arrival. Constraints were appended to the name attribute of input field. This provided tight binding of the constraint to its intended field, and made simpler the job of form reconstruction, as the number of fields involved was just the original number of fields; none needed to be added. For security reasons FPT does not write files. All data generated is returned as a stream. With respect to the form user, the natural means for communication was through HTTP. The user had initiated this communication through HTTP, and it was easy to extend this to a dialog. With respect to the form owner, it was decided to use SMTP, or email. Communication with a remote form owner was easy, and at ARSC it was a simple matter to design an application to read emailed data directly into a database, or to wherever destined. Facility to import an existing form, and add to it from a set of predefined constraints is provided.

Implementation Language

FPT performs the following major tasks: pattern matching to decide whether data submitted by a form user satisfies constraints against its form specified by the form owner, and expression evaluation to decide whether a given data input item submitted by the user satisfies an expressed relationship with other input items as specified by the form owner. These expressions are read in and executed at run time. Document fetching: FPT receives no input other than URL encoded name=value pairs. To return as output a variety of reconstructed forms. FPT fetches form blanks through the HTTP protocol, and involves tasks common to every Web client such as setting up a socket, binding and connecting to it and document rebuilding. FPT takes data submitted by a form user and reinserts this data, along with possible error messages, into a blank form for use in reconstructing the user's form. The first two tasks each require that FPT be able to read in and execute at run time arbitrary regular and general expressions. Hence FPT is written in Perl. Perl was selected for the pattern matching and document rebuilding tasks. Its capacity to read in at run time arbitrary expressions and trap otherwise fatal errors when evaluating them made it ideal for the expression evaluation task, and finally, the provision by Perl of its own socket-handling procedures makes easy the implementation of document fetching.

Targeted Platforms

FPT interfaces both with Web clients and with Web servers. With respect to Web clients, FPT was required to generate documents capable of being displayed correctly and attractively across a wide range of clients ranging from simple character-based browsers to sophisticated multimedia browsers. To promote portability only HTML tags that are a part of the HTML 2.0 standard are used for documents generated by FPT. FPT requires web servers to be CGI 1.0 compliant so files such as FPT are executed, rather than simply returned. FPT consults these seven environment variables: 1. Http_referer, 2. Remote_host, 3. Request_method, 4. Query_string, 5. Script_name, 6. Server_name, 7. Server_port. Five of these seven are checked by FPT only in the event that certain name=value pairs are missing in the form data it receives. After error-checking submitted form data FPT attempts to reconstruct the form as it appeared to the user at the time of submission. To do so, FPT fetches a blank copy of the form, since only the data has been submitted to it. The URL of the form is usually embedded in a hidden form field named 'FormURL'. If this value has not been set by the form owner, FPT looks to see if HTTP_REFERER has been set. The HTTP_ prefix of this environment variable indicates that this is a value set by the client, not the server, and yields the URL of the document on display when the submit button was pressed; i.e., the form URL. Not all clients set this value. Mosaic, for instance, does not set it for forms. If the form owner has not supplied the form URL, and if the HTTP_REFERER value has not been set, FPT proceeds without the blank form, working just with the form data. Likewise, the other four of these five non-essential environment variables are consulted for assistance only when needed, and their absence dies not cause FPT will fail. As for the other two environment variables, Request_Method And Query_String, it is necessary that at least one of these has been set. For method POST, Request_Method needs to be set to POST, so that FPT knows to look to STDIN for its input. For method GET, Query_String needs to be set, as the input will then be found there.

Security Considerations

Web servers commonly allow users to execute CGI applications, thereby opening the possibility that weaknesses in their design can cause them to be exploited for purposes other than those for which they were designed. CGI applications derive their usefulness by returning to the client a virtual document tailored to the client's request. This tailoring is generally guided by data supplied by the client. Vulnerability arises when the application naively executes a command supplying this data as arguments. For example: consider the string argument system ("/bin/echo $ENV{'QUERY_STRING'}"); While this command is intended to echo submitted data back to the client, it can cause damage if the value of QUERY_STRING is:

Gotcha! \n rm * First "Gotcha!" is returned, followed by an attempt at file deletion. FPT is expressly designed to accept external snippets of Perl code. It is this facility, for accepting and applying at run time arbitrary external regular and general expressions for use in generating customized constraints and error messages, that makes FPT of general utility, rather than a special purpose handler for only certain known forms. By design, FPT writes no files; data is returned to the form user through HTTP, and to the form owner through SMTP (email). With respect to the form user, an FPT failure means time wasted preparing, entering, submitting the data, and loss of data itself. To the form owner, an FPT failure means being deprived of receiving data intended for him. However, an errant or malicious snippet of imported code, when executed by FPT, has the potential for inflicting upon the form owner all the damage that the owner could inflict upon himself. In light of potential damages, FPT is designed to permit only the importation and execution of code read in from a companion file, named fpt.aux, and only when that file resides in the same directory as FPT and has in common the same owner as FPT. To deal with code supplied by FPT's owner, and therefore assumed to be of benign intent but which may possess syntax errors to cause a run time error, the eval function in Perl is used. Code which is eval'd is executed as a little Perl program within the context of the main Perl program. Any run time errors are trapped, in which case control is simply returned to the main program. Use of the fpt.aux file thus provides secure access to auxiliary code and is also a convenient means for supplying this code. Regular expressions can be relatively short but general expressions may be of arbitrary length and complexity. To embed such expressions within a form field's name attribute is awkward and use of fpt.aux file helps. The fpt.aux file provides a convenient place to gather together reserved characters in HTML, such as "<", ">", and "&". For users unable or unwilling to create such a file, FPT supplies sixteen built-in constraints consisting of regular expressions for use in specifying certain commonly occurring data types. Upon failure by FPT, the server returns an error message to the user. The user in such a case would usually be able to return to the form, but the data within it exists only in the browser's memory and display. None of the browsers in the FPT test suite offer any facility for saving this data to disk, or even for printing it.

Constraint Types and Formats

The fpt.aux file, is designed to function as a repository of expressions to be used in applying constraints and generating error messages. fpt.aux is a list each item of which has the following format: LABEL: CONSTRAINT EXPR: ERROR MS EXPR: The 'LABEL:' tag supplies a name to be associated with the constraint and error expressions following it. The label name must be unique with respect to other listed labels, and must also be a single valid Perl word (letters, digits, and the underscore). The 'CONSTRAINT EXPR:' tag supplies a Perl expression for use in evaluating submitted form data associated with a label. One of two types of expressions may be used: a regular expression against which the associated data input item is matched; or arbitrary Perl code to be eval'd as an expression. A return value of zero indicates failure of the form data to satisfy the constraint placed upon it. The 'ERROR MS EXPR:' tag supplies a Perl expression for use in generating an error message in the event an error in the data submitted by the user failed to satisfy the constraint placed upon it. This Perl expression may be arbitrary Perl code to be eval'd, or may be just plain text. The 'ERROR MS EXPR:' tag is optional; if not present, FPT will supply a default error message.

Constraint Encoding

FPT is primarily designed to check form data for errors. This error-checking is based upon constraints specified by the form owner. Certain characters within HTML are reserved and cannot be used within a regular expression without escapement. In order to avoid having to encode and decode these reserved characters, and in order to employ a uniform procedure for specifying both types of expressions, labels are used for regular expressions also. Labeled constraints are attached directly to the name attribute of form fields. To distinguish labels from separators two short groups of characters whose purpose would be clearly identifiable from their appearance are used. Two groups were required to distinguish labels for regular expressions from labels for general expressions, as both may appear attached to a name. The following input field example, taken from the "Using fpt.aux" help document, shows how both may be specified: <input name = "SalesTax[RE]_Dollar?[GE]SalesTaxCode" size = 5> The name attribute to this input field is 'SalesTax', and has appended to it labels for both a regular expression ('_Dollar') and a general expression ('SalesTaxCode'), separated from the name, and from each other, by '[RE]' and '[GE]' respectively. The labels themselves are required to be unique, and to consist of a single valid Perl word. The '_Dollar' label is one of the built-in labels recognized by FPT, and as with all such built-in labels, it begins with an underscore in order to reduce the chances of colliding with a form owner-generated label. For labels corresponding to regular expressions, a trailing question mark signals to FPT that it is to apply the constraint only when data is present. For labels corresponding to general expressions, question marks are not used, as general expressions presume some value has been entered and are not evaluated otherwise. FPT offers two modes of operation: a 'standalone' mode and an 'expert' mode.

Standalone mode

Standalone mode, functions without the fpt.aux file, and is therefore capable of being used remotely. A form owner could have FPT import his pre-existing form into FPT, and choose for each input item of his form a constraint drawn from sixteen built-in data types. Standalone mode requires nothing more of form owners other than that they have a form. Users do not need to install any software, nor know any programming. Access to standalone mode is through FPT's welcome page. The 'UsingFptPrl.html' link on the welcome page returns a document explaining how to use FPT to customize a form. The document requests seven pieces of information for use in customizing a form and FPT's actions upon it. The information requested is:

- Form source URL (required): URL of the form to which the owner wishes to add constraints required)
- Form owner's email address (required): email address to which user data should be sent
- Form destination URL: new name for the transformed form if desired
- Acknowledgement document URL
- Subject line for emailed form data: If left blank, data pairs are returned to the owner, using default subject lines
- Subject line for emailed reconstructed form: If left blank, the reconstructed form is returned to the owner, using default subject lines
- Override button label: label to use for the forcible submit button

If the submitted items are in good form, FPT fetches a copy of the specified form, and goes through it replacing the action URL to its own URL, embedding in hidden fields those items submitted, supplying a mode code, and replacing all input fields within the form with a menu widget listing sixteen predefined data types: Anything, Integer ,Phone, URL Date, Letters, SSAN, Word, Dollar, Name, State, Year, Email, Number, Text, Zip. Each type is presented twice, once with a question mark and once without. Choosing a data type followed by a question mark indicates that input for that field is optional, but if present should conform to that type. The data types chosen represent commonly occurring data types in forms and are limited to sixteen (or thirty two, with the optional types) simply because it was found that specifying more than sixteen results in a widget larger than the default browser window. Each data type is a label corresponding to a regular expression for that type built into FPT. The transformed form is returned to the form owner with an explanation of how to proceed. Once the choices have been made, the owner presses the submit button, which now reads 'transform'. FPT reads in the choices made, fetches the blank form, and this time goes through the form appending any chosen data type labels to the name attribute of the associated input field. The transformed form is then returned to the form owner. At this point, as the form exists only in the browser's window and in the browser's memory, it is necessary for the form owner to save the transformed form to disk if he/she wishes to reuse it. Once this done, the process is complete. Data entered by any user of this form will now be sent to FPT for preprocessing, according to the constraints specified by the owner. Once accepted, this data will then be emailed to the form owner in the formats specified.

Expert mode

Standalone mode was designed to be easy to use, and the simple data types it provides are sufficient for catching many common errors of commission, and all errors of omission. But it has three significant limitations: (1) only data types built into FPT may be used. If the form owner has input fields for which none of the supplied types are appropriate, there is no help for it, (2) regular expressions are the only expressions used in evaluating the form data. This means that only errors of form can be caught, not errors of content, and (3) only input fields of type 'text' may be error-checked. Other input types, such as checkbox, radio, and select are unaffected. This is because with these types choices are presented to the form user, and the resultant value is binary - on or off - according to whether the choice was selected or not. Imposition of data types has no meaning with these items. Expert mode overcomes these limitations, and allows for the construction and evaluation of arbitrary expressions incorporating any or all of the submitted data. Expert mode uses a auxiliary code file, fpt.aux. Security concerns require that when used in expert mode both fpt.prl and fpt.aux be downloaded and installed in the form owner's server cgi directory. This allows for sophisticated error-checking and for the generation of meaningful and narrowly targeted error messages. However, use of expert mode imposes two principal requirements upon the form owner: (1) the file supplying the auxiliary expressions to fpt.prl should be located in the same directory as fpt.prl, and be owned by the same owner as fpt.prl, and (2) the form owner should have a working knowledge of Perl. FPT is a Perl script, and constraint expressions are eval'd as snippets of Perl code. The ability to embed arbitrary Perl expressions is only useful to the degree that one can generate them. Once fpt.prl has been installed, and fpt.aux either downloaded or created, actual usage of FPT in expert mode differs from standalone mode in two principal ways. The chief difference is the use of fpt.aux in òexpertó mode. fpt.aux supplies to fpt.prl the expressions which the form owner has created. fpt.aux contains the list of labeled constraints and list items that must conform to a certain format in order to be read in by fpt.prl. The second difference is that in standalone mode FPT performs the actual insertion of labels into the form, whereas in expert mode if the form owner wishes to supply labels other than those built into FPT then it must be done manually, with an editor. Instructions for creating constraints, listing them in fpt.aux, and appending their labels to form field name attributes, are given in the document UsingFptAux.html, along with detailed examples.

Constraint Decoding

FPT is a general purpose forms preprocessor and is used by specifying its URL for the action attribute in the opening form tag, as in this example:

<form action = "http://www.arsc.edu/cgi/fpt.prl" method = "POST"> When a form user presses the submit button, all data entered into the form field containing that submit button is collected by the browser, formed into a list of name=value pairs, URL-encoded into a single whitespace-free string, and forwarded to the action URL according to the specified method. Two methods are currently available for returning form data, 'GET' and 'POST'. With method GET, the string is simply appended to the requested URL. With method POST, data is

returned through STDIN. There are length limitations imposed when appending data to a URL and method POST provides a clean and general mechanism for returning data. FPT accepts data submitted by either method. FPT passes the data to a simple routine which decodes the data, and then to another routine for reconstructing the name=value pairs, at which time any appended constraint labels are identified and loaded into arrays separate from the values and unencumbered names. Despite the fact that FPT writes no files, expires between invocations, and is totally stateless, it does operate in three modes, triggered by the presence or absence of key name=value pairs. Two of these modes are associated with its constraint-embedding function - preparing a form for transformation, and then performing the transformation - while the third is its default error-checking mode, entered into in the absence of these key name=value pairs. In error-checking mode, FPT first looks to see if there are any constraint labels other than the built-in constraints. If so, and if deemed secure, the contents of fpt.aux are read in. In either event, the actual expressions corresponding to whatever labels have been encountered are loaded into associative arrays. FPT then steps through the values array. If a regular expression is associated with a value, then the value is matched against it, within the context of an eval, in order to trap any run time errors. If a general expression is found, it is likewise eval'd. In either case, a return from eval with value 0 causes the associated error message expression to be eval'd. If there is an evaluation error, or if no error expression has been supplied, FPT then generates a default error expression, which identifies the item found in error and supplies the expression which failed to evaluate. Once the values have been exhausted, errors tallied, and generated error messages loaded into arrays, FPT fetches a blank form for reconstruction. The name=value pairs also generated by FPT are designed primarily to be read by scripts, which may then load this data into a database, or otherwise manipulate the data without human intervention. When errors in form data are encountered, FPT returns to the user a reconstructed form in which the data has been kept intact and errors flagged where they occurred. Also present (unless the form owner has specifically disabled this feature) is a forcible submit button, whereby the user may resubmit his data and bypass error-checking. The provision of this button spares the user the frustration of constantly having his data bounced back to him in situations where the error may lie with the constraint, rather than the data. For example unanticipated data formats, as with a foreign postal code for a field expecting a ZIP code. Communication with the form user is exclusively through HTTP. When data is finally accepted, either because it was found to be free of errors or was forcibly submitted, an acknowledgment document is returned to his browser. In the absence of an acknowledgment document specified by the form owner, FPT generates and returns a default acknowledgment. Communication with the form owner is exclusively through email. This provides a general mechanism for returning data in a machine-readable form, and allows for remote operation of FPT. The form owner can receive either the user's reconstructed form or the decoded name=value pairs, or both. A number of errors other than those found in user data may be encountered during this processing. These include fetch failures, as with problems encountered trying to fetch a blank form, or an acknowledgment document: constraint failures, as with ill-formed expressions, or labels for which no corresponding expressions can be found; I/O errors; and others. A bit vector 32-wide is used to signal the presence or absence of all errors, and error messages generated are stored in arrays created for that error type. In the event of errors other than user data errors, FPT attempts to return what data it can while shielding the user from error messages generated by conditions beyond his control. The form owner, however, receives full reports on all error conditions and messages encountered.

Future Work

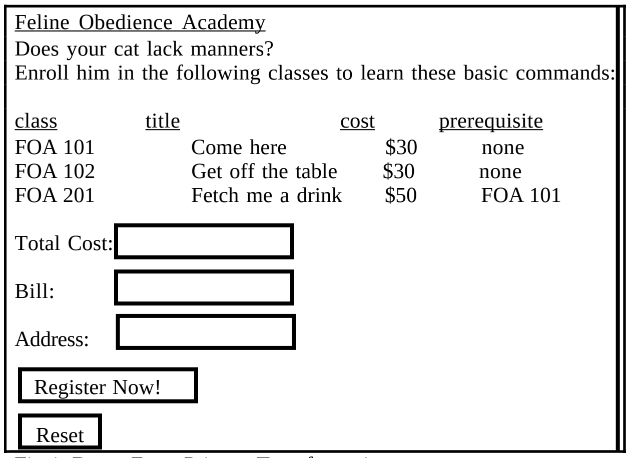
The implementation of FPT as a CGI application was the only choice available at the time FPT was designed. Today, Java-capable browsers can download snippets of Java code, 'applets', for immediate execution. This means that FPT could be rewritten as an applet to provide client-side preprocessing, thus saving valuable bandwidth and improving response time for the form user.

References

[1] *Programming Perl*, Larry Wall and Randall Schwartz. O'Reilly and Associates, 1991.

[2] *The Tao Of AppleScript*, Derrick Schneider, Hans Hansen, and Tim Holmes,1994

Appendix



Feline Obedience Academy
Does your cat lack manners?
Enroll him in the following classes to learn these basic commands:

class	title	cost	prerequisite
FOA 101	Come here	$30	none
FOA 102	Get off the table	$30	none
FOA 201	Fetch me a drink	$50	FOA 101

Total Cost:

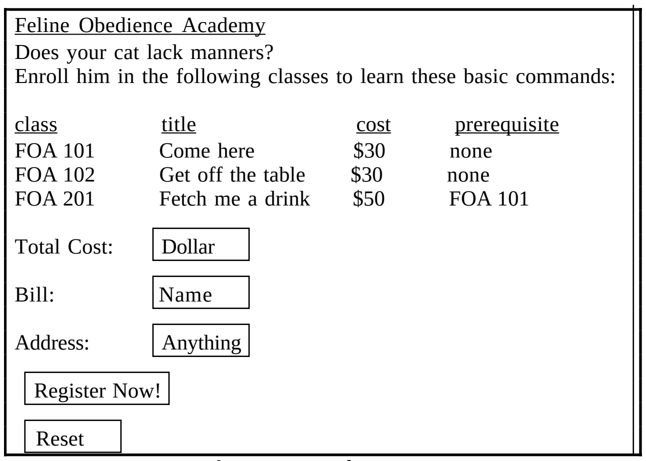
Bill:

Address:

Register Now!

Reset

Fig 1. Demo Form Prior to Transformation



Feline Obedience Academy

Does your cat lack manners?

Enroll him in the following classes to learn these basic commands:

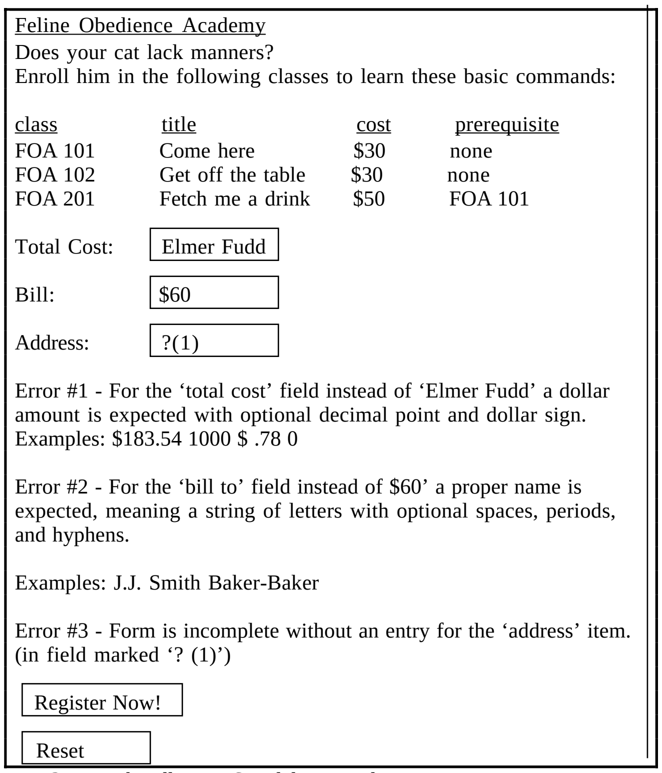
class	title	cost	prerequisite
FOA 101	Come here	$30	none
FOA 102	Get off the table	$30	none
FOA 201	Fetch me a drink	$50	FOA 101

Total Cost:	Dollar
Bill:	Name
Address:	Anything

Register Now!

Reset

Fig 2. Demo Form Undergoing Transformation



Feline Obedience Academy

Does your cat lack manners?

Enroll him in the following classes to learn these basic commands:

class	title	cost	prerequisite
FOA 101	Come here	$30	none
FOA 102	Get off the table	$30	none
FOA 201	Fetch me a drink	$50	FOA 101

Total Cost:	Elmer Fudd
Bill:	$60
Address:	?(1)

Error #1 - For the 'total cost' field instead of 'Elmer Fudd' a dollar amount is expected with optional decimal point and dollar sign. Examples: $183.54 1000 $ .78 0

Error #2 - For the 'bill to' field instead of $60' a proper name is expected, meaning a string of letters with optional spaces, periods, and hyphens.

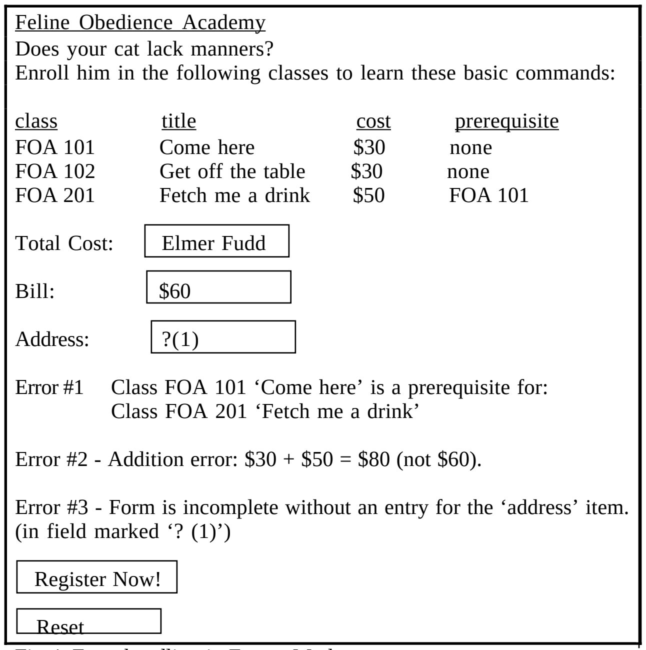
Examples: J.J. Smith Baker-Baker

Error #3 - Form is incomplete without an entry for the 'address' item. (in field marked '? (1)')

Register Now!

Reset

Fig 3. Error handling in Standalone Mode



Feline Obedience Academy

Does your cat lack manners?

Enroll him in the following classes to learn these basic commands:

class	title	cost	prerequisite
FOA 101	Come here	$30	none
FOA 102	Get off the table	$30	none
FOA 201	Fetch me a drink	$50	FOA 101

Total Cost:	Elmer Fudd
Bill:	$60
Address:	?(1)

Error #1 Class FOA 101 'Come here' is a prerequisite for:
Class FOA 201 'Fetch me a drink'

Error #2 - Addition error: $30 + $50 = $80 (not $60).

Error #3 - Form is incomplete without an entry for the 'address' item. (in field marked '? (1)')

Register Now!

Reset

Fig 4. Error handling in Expert Mode

```
LABEL: _Anything
  CONSTRAINT EXPR: \S+
  "Form is incomplete without an entry for the
  \'$Names[$i]\' item. (in field marked '$Values[$i]')"

  LABEL: _Dollar
  CONSTRAINT EXPR: ^\s*\$?\s*(\d+\.?\d*|\.\d+)\s*$
  ERROR MS EXPR:
  $Str = ($Values[$i] =~ m|(\? \(\d+\))|) ? "(marked with '$1')" :
                                    "instead of '$Values[$i]'";
  "For the '$Names[$i]' field $Str
  a dollar amount is expected, meaning a string of digits with
  optional decimal point and dollar sign.
  Examples: \$183.54  1000  \$ .78  0"

  LABEL:_Name
  CONSTRAINT EXPR: ^\s*[-a-z.][-a-z.\s]*$
  ERROR MS EXPR:
  $Str = ($Values[$i] =~ m|(\? \(\d+\))|) ? "(marked with '$1')" :
                                    "instead of '$Values[$i]'";
  "For the '$Names[$i]' field $Str
  a proper name is expected, meaning a string of letters with
  optional spaces, periods, and hyphens.
  Examples: J.J.Smith  Baker-Baker  von Neumann"
```

Fig 5. Sample of Regular Expressions

```
LABEL: DemoPrerequisite
CONSTRAINT EXPR: defined($FormData{'come'})
ERROR MS EXPR: local($NineSpaces) = '&nbsp;' x 9;
          "Class NOA 101 ‘Come here!' is a prerequisite for:\n
$NineSpaces Class NOA 201 ‘Fetch me a drink!'"

LABEL: DemoSum
CONSTRAINT EXPR: local($Total);
$Total =  $FormData{'total cost'};
$Total =~ s/\s*\$?([\d|.]*)$/sprintf("%d",int($1))/e;
$Total == 30 * defined($FormData{'come'}) +
          30 * defined($FormData{'down'}) +
          50 * defined($FormData{'beer'})
ERROR MS EXPR: local($Class,$Cost,@Stack,$Total);
if (defined($FormData{'come'})) { $Total += 30; push(@Stack,'$30');
}
if (defined($FormData{'down'})) { $Total += 30; push(@Stack,'$30');
}
if (defined($FormData{'beer'})) { $Total += 50; push(@Stack,'$50');
}
if    ($#Stack < 0)
{  "No classes selected."; }
elsif ($#Stack > 0)
{
  $Expr = join(' + ',@Stack);
  "Addition error: $Expr = \$$Total (not $FormData{'total cost'})";
}
else{
  defined($FormData{'come'}) && ($Class = "NOA 101") ||
  defined($FormData{'down'}) && ($Class = "NOA 102") ||
  defined($FormData{'drink'}) && ($Class = "NOA 201");
  $Cost = pop(@Stack);
  "Tuition for $Class is $Cost (not $FormData{'total cost'})";}
```
Fig 6. Sample of General Expressions