

# Knitting Patterns: for interview and analysis

Stephen Jenkins

## Abstract

This paper describes an implementation of lexical patterns, based on Regular Expressions and developed for use with Snap survey software. The intention is to make electronic interviews more amenable to respondents whilst providing researchers with in-depth response interrogation tools.

The paper is of interest to all those conducting online and offline research using direct respondent input into electronic questionnaires, and those writing software for such applications.

## Keywords

Survey Metadata, Regular Expressions, Data Validation

## 1. Motivation

The past five or more years has seen a rapid growth in the use of electronic questionnaires such as those intended for web, kiosk and laptop-based interviewing. Key benefits cited by proponents of the technology are that interviews are faster and cheaper because the manual data entry step involved in paper questionnaires is removed.

That simple observation ignores the fact that the process of keying data from paper questionnaires often does not just involve the direct transcription of respondent replies. With open-ended questions in particular there is often a requirement to normalise the replies during keying. The process involves application of one or more transformations of the respondent data in order to perform operations such as:

- remove currency symbols (or other symbols defining units)
- type conversions (e.g. convert a height in feet and inches into metres)
- change worded forms of numbers into numerals (e.g. convert a response of two into 2)
- interpret formatting within numbers (e.g. 1k becomes 1000 and 1,234 becomes 1234)

It is our assertion that if alternate methods of expression such as these are disallowed to respondents of electronic questionnaires by overly restrictive formatting requirements (as they frequently are), then respondents get put off completing. Use of this technology could thus contribute to increased response rates.

## 2. Looking for solutions

As has been alluded to above, the problem of accepting non-conforming respondent replies to questions exists even in the world of paper questionnaires. Aside from the data-entry-time transformation of responses into a standard form mentioned previously, other techniques are used to try to guide the respondent towards giving a response in an acceptable format. These standard solutions generally involve either printing constant parts of the response in the hope (not always realised) that the respondent will not repeat them, or splitting the single input into parts, often separated by constant elements.

For example, when requesting currency amounts, one might see:

£

or:

£  .

Sometimes the elements are labelled, so for date inputs one might see:

/  /   
day month year

None of these is foolproof of course. It is not unusual to see:

£

given as a response on a paper questionnaire.

For electronic questionnaires we are working on the premise that a questionnaire designer would want to be as friendly as they can be towards respondents. They would thus want to be in a position where 1.5k is an acceptable answer to an appropriate quantity question. In general, we can consider that a questionnaire designer has three choices when dealing with such a situation:

1. Constrain the input to the exact format anticipated and perform calculations and range checks etc. on the fly during the interview. In our view these formatting restrictions are unacceptable as a blanket rule, for the reasons stated.
2. Allow any input at interview time and clean the problem up when interview data is collated. This is fine but the opportunity to get the respondent to correct inadvertent typographical or magnitude errors will usually be lost. Furthermore, the response to such a question (whether immediately valid or not) cannot be used in routing or any later calculations.
3. Impose some flexible interpretation mechanism, specialised to the question, that allows appropriate flexible input forms but is still able to expose the intended value to the underlying validation, routing and calculation processes in order to continue driving the interview.

Our solution to this problem is based on Patterns as exemplified in Regular Expressions. Basing the system on a ubiquitous technology means that formatting constraints are more

easily represented in different systems thus allowing, for example, rewriting as JavaScript Regular Expressions for use in web/HTML surveys.

### 3. Regular Expressions

Regular Expressions provide a way to match text with patterns. In general they provide a powerful way to find and replace strings. They are typically used in two ways:

- To provide text-file editing facilities (implementing the common find-and-replace operation in text editors)
- To validate and interpret user input in windows and web-forms.

It is this latter use that we are most interested in for validating respondent input into electronic questionnaires.

#### An example

Consider the problem of validating the response to a date question such as *When did you enrol?*. We want to validate respondent replies such as the following:

1/9/07            01/09/07            01.9.07            1.09.07

That is, we assume that the date will be of the form dd/mm/yy where the dd part is one or two digits, the mm part is one or two digits and the yy part is exactly 2 digits. For flexibility, we want to recognise either a dot or a slash as a separator.

The (or more accurately, a) regular expression for such a pattern is:

$$\backslash d\{1,2\}(\backslash|\.)\backslash d\{1,2\}(\backslash|\.)\backslash d\backslash d$$

The construction to  $\backslash d\{1,2\}$  specifies "between 1 and 2 digits", the  $\backslash|\.$  part specifies "either a slash or a dot", and the construction  $\backslash d\backslash d$  specifies "two digits". So the whole pattern reads as "1 or 2 digits followed by a slash or a dot, followed by 1 or 2 digits, followed by a slash or a dot, followed by 2 digits."

As complex as it is, the regular expression cited is only an approximate specification for a date. Whilst it allows inputs such as 19/09/07, which could later be interpreted as 19<sup>th</sup> September 2007, it would also allow 58/93/07 (the 58<sup>th</sup> day of the 93<sup>rd</sup> month?) and 19/09.07 (with its inconsistent use of field separators.) Although it would be possible to extend beyond the simple pattern shown to reduce some of these issues, it would not sensibly be feasible to extend it to the level of validating the day, month and year values to ensure that only valid dates were passed (for example to reject 29/02/07). Those value constraints normally require the use of external code: the pattern identifies the individual parts and the outer code validates the parts in conjunction with one another.

So, whilst regular expression patterns are very expressive and powerful in the right hands, and given the right application, they are very complex to learn and to apply successfully for any requirements beyond the most basic in our field of interest.

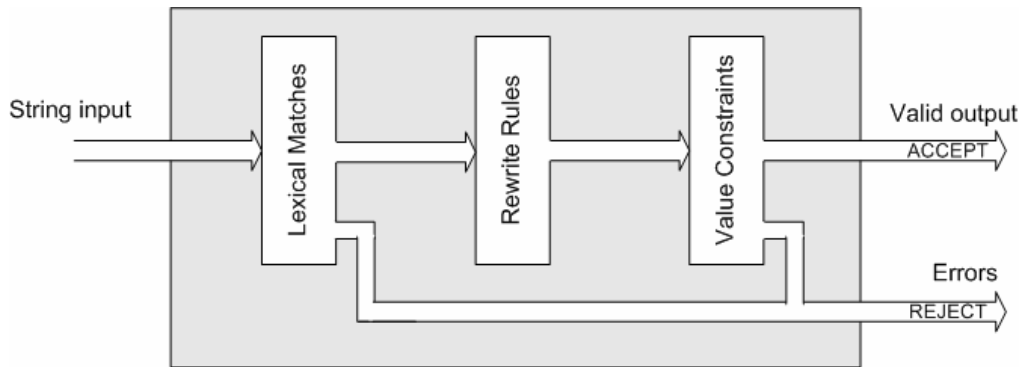
Furthermore, if provided as the sole means of allowable format specification, they require a user (in our case the user of survey questionnaire design or analysis software) to learn another language which, because of its level of abstraction, removes them from the immediate world of survey data manipulation.

## 4. The essence of a pattern

We can consider the validation process required of a pattern to be comprised of three steps:

1. Attempt one or more lexical matches
2. If successful, possibly rewrite the input in a different form
3. Check that the result is an acceptable value

The steps of the validation process are illustrated in the following figure and described below.



**Figure 1 – The steps in pattern validation**

The **Lexical Matches** specify the allowable syntax of the input. Note the lexical match candidates can either be literal values or more patterns (and thus in that way a pattern may be defined recursively.)

If one of the lexical matches finds the input acceptable, then the **Rewrite Rules** are invoked as the next step. These can either:

- Take components of the input and rewrite them in a standard way. For example, to accept 20.3.07 or 20/3/07 as a date but to output them in a standard form as 20/03/07.
- Convert all letters to upper- or lower-case (this is actually a variant of the above but identified separately as it is such a common requirement).
- Write something lexically different to, but semantically the same as, the validated input. For example, a pattern that converts worded numbers into digits would include a pattern that recognised `two` as input but which writes out `2` as the result.

The final step is to apply **Value Constraints**. Typically these are comprised of a single range of allowable constant values. Value constraints may have been applied already as a side-effect of the available Lexical Matches, for example suppose a pattern was to be set up to allow the respondent to enter integer values of 1 to 3 only: if the Lexical Matches section allowed 1 or 2 or 3 then the Value Constraints could be empty. If the Lexical Matches allowed 0 through 9 as valid input then a Value Constraint of 1 to 3 should be introduced to provide a final check on the value.

## 5. Patterns as implemented

An overall goal was to provide the user with a way of creating patterns to validate input that are at least approaching the power of regular expressions, and which may be translated into regular expressions for publication to electronic forms such as HTML.

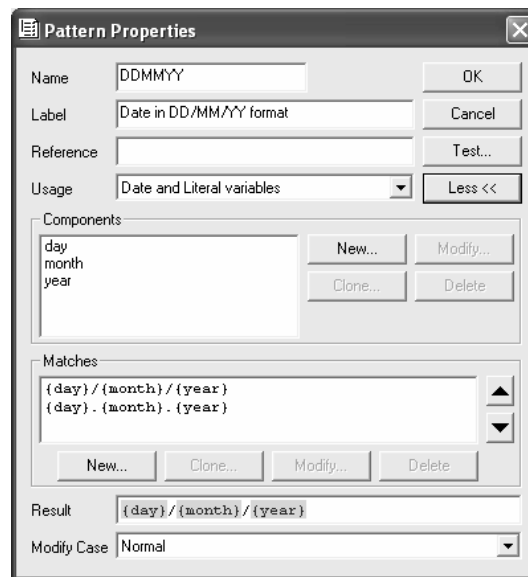
The basic idea introduced was to base the description of the allowable lexical matches on parameterised strings. Snap users are familiar with the use of these to provide text substitution (sometimes known as response-piping) at interview time. For example, they might include a question which includes the following sequence:

Q1 Which of these fruits is your favourite?: Apples / Pears / Bananas

Q2 You said your favourite was {Q1}, why is that? \_\_\_\_\_

Then at interview time, the text of Q2 has the response from Q1 inserted to specialise the question to the individual respondent.

We used a similar technique to enable the user to specify lexical matches. Consider the example pattern below which validates dates (it does essentially the same job as the sample regular expression pattern for parsing dates described in section 3.)



**Figure 2 – Pattern properties dialog**

Amongst other things it contains three **Components**: `day`, `month`, and `year` which are themselves patterns. The lexical matches (shown in the **Matches** section of the dialog) are then described in terms of the components and literal values. Any (respondent) input passing one of the lexical matches is considered valid and is then processed by the **Result** section. In the example shown, the components are again used to standardise the result (and in this case no case conversion is applied.)

It is now quite easy to allow other lexical match formats. For example, to allow dashes as a separator, the user simply needs to create a new match of `{day}-{month}-{year}`. As an alternative, it would be possible to specify the separators as a list of alternatives and associated with a `separator` component. If that component is set to `consistent` then occurrences of it found in one string must be identical (and hence only slashes, or only dots, or only dashes would be allowed as the separators in a valid date response.) However, working in the way shown helps to keep the pattern easier to compose, understand and test.

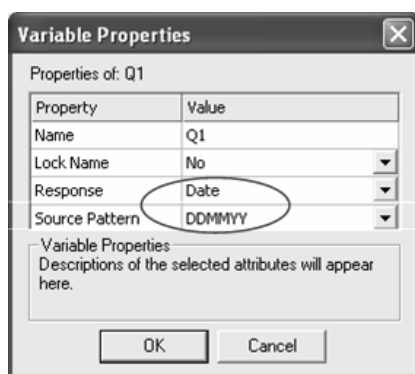
## 6. Patterns and in-built response types

When a user creates an open-ended question in Snap, they select a response type from the list of in-built types (shown in Table 1, below.) For example, the in-built Quantity type allows positive or negative decimal numbers with or without a decimal point. The in-built Date type allows the usual dd/mm/yy forms with 2 or 4 digit years and a variety of separators. It also allows for dates to be expressed in worded form, for example 1st September 2007 would be considered an acceptable response.

Type	Description
Quantity	A numeric value, positive or negative, decimal or integer
Date	A date based on the Gregorian calendar
Time	A time of day (from 0:00 to 23:59:59)
Literal	Any other type of open response

**Table 1 – In-built open-ended response types**

When the user is entering questions and assigning each a response type (Figure 3), they are in fact associating in-built, patterns (which the user doesn't see) that implement the basic validation for the type. Patterns created by the user are associated with one of the in-built types<sup>1</sup> (see Figure 4.) The program uses this association to present a list of possible patterns available for the question (variable) being edited.



**Figure 3 – Variable properties dialog**



**Figure 4 – Pattern properties dialog**

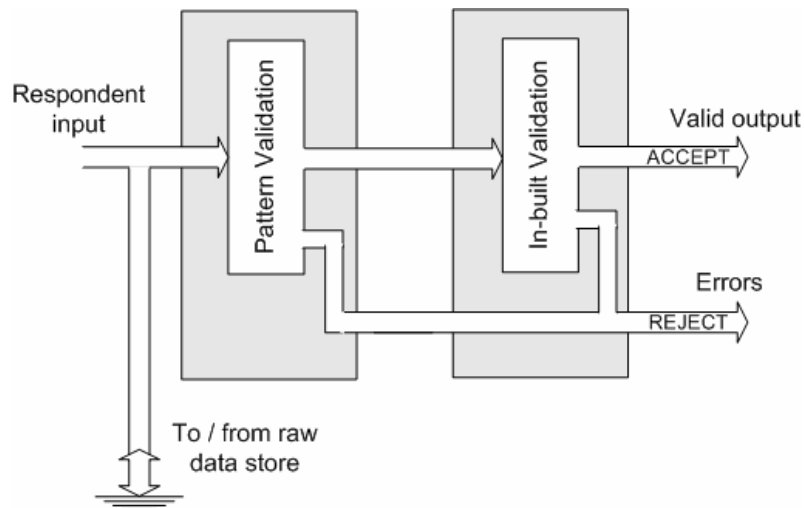
### Processing data through patterns

As has been discussed, the fundamental purpose of a pattern is to validate respondent input. That input could be coming from one of two places:

<sup>1</sup> Note that since any respondent data can be represented as a text string, any pattern intended for Quantity, Date or Time response data may also be interpreted as Literal response data.

- direct from the respondent – as would be the case where the pattern is validating data during an interview.
- from a data store – as would be the case for data already keyed from paper questionnaires or stored from interviews conducted before.

In both cases, the pattern works in the same way: it takes input (respondent) data, passes it through its own lexical matches, rewrite rules and value constraints then, if acceptable, passes it on to the appropriate in-built validation associated with the response type of the variable. The process is illustrated in Figure 5.



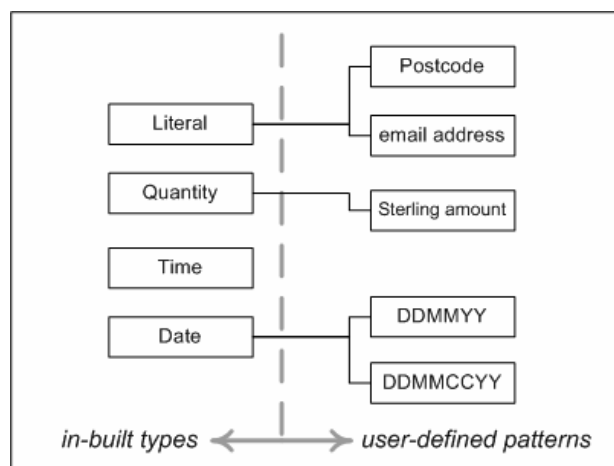
**Figure 5 – Processing respondent input through pattern and in-built validation**

Since the processing done by patterns and the in-built validation can be expensive in terms of CPU time, the results are cached by the program and held in an inverted datastore (that is, processed results are held by question whereas raw data is held by respondent.). Any changes to the raw data results in the automatic re-processing of the data. The result is that analysis tables and charts are always accurate and fast to build.

An important point to note is that the raw data records the exact respondent input, not transformations of it. Thus there is always a direct and documented audit trail from respondent input to analysis results. In addition, the original data is still available to be processed by other variable / pattern combinations to provide different interpretations of individual data entries if deemed appropriate and useful. This contrasts with the traditional method when keying data from paper questionnaires using data entry operators to "clean" the data, where the audit trail between the paper copy and the electronic copy of respondent input is lost.

## 7. Patterns as types

By connecting a pattern to an in-built response type, the user is essentially *specialising* those types.



**Figure 6 – Specialising in-built response types**

Those specialisations are of three forms: those that apply format restrictions; those that apply format extensions; and those that apply value restrictions. Sometimes these are applied singly, sometimes in combination. For examples, consider the following:

- Format restrictions – for example, an **integer** pattern based on the in-built **quantity** pattern would restrict recognition to values with no decimal point allowed.
- Format extensions – for example, a **worded number** pattern based on the in-built **quantity** pattern (allowing respondents to enter `two` to mean 2, for example.)
- Value restrictions – for example, a **human age** pattern based on quantity

As we have seen (Figure 5), we can view the pattern as a small processing machine which, when given an acceptable input string, emits a value of the connected in-built response type at the back end. Thus, an acceptable input to the pattern becomes an acceptable input to an in-built response type. The user has created an IS\_A relationship between the pattern and the associated in-built type. That is, for example, a **worded number** (as outlined above) IS\_A **quantity**.

Now, the in-built types have in-built operations available. For example one quantity can be added to another to derive a quantity representing the sum of the other two. The nature of IS\_A relationship is that all operations available on the in-built type are available for the specialised type represented by the pattern. Thus the in-built operators, operating on the in-built types can be considered to be inherited by the patterns. So, for example, the response to a question built using the worded integer pattern can be added to, or subtracted from, the response to another suitable question.

So, by composing patterns in the way described, the user is extending the type-space of the underlying program.

### New operators for free

One useful side-effect of the use of named components to construct a pattern is that those components become available for interrogation. For example, a UK postcode pattern might be created from four components each representing the standard Royal Mail terminology (Table 2.)

The final match pattern based on these components would then be:

```
{area}{district} {sector}{unit}
```

Component	Lexical match	Meaning
area	one or two letters	Sorting office
district	one or two digits, or one digit and one letter	Delivery office
sector	one digit	Local area
unit	two letters	Postal route

**Table 2 – UK Postcode components**

Now, if a question, say Q5, was assigned this pattern, it would be possible to interrogate the *components* of a response. So the expression Q5 `area` gives up the area part of a complete postcode response, Q5 `district` gives up the district part, and so on.

The components can be considered to be unary operators, each of which divulges a part of a complete response. Thus patterns composed in this way can be considered to be types in their own right describing, as they do, allowable data values *and operations on them*.

### Response box formatters for free

A second useful side-effect where named components are used is the possibility of drafting complex formatted input boxes from the pattern specification. For example, given the earlier DDMMYY pattern, but allowing `{day}/{month}/{year}` as the only possible match, it becomes feasible to publish response boxes questionnaires which look like this for a paper questionnaire:

or like this for a web questionnaire:

This formatting possibility comes absolutely for free in that the user is not required to participate at all – all of the required information is already present in the definition of the pattern.

### Further work

There are a number of extensions we are currently exploring:

- Patterns are currently specialisations of in-built types only. A simple extension would be to allow users to create patterns as specialisations of other patterns (which themselves would ultimately be specialisations of the in-built types). This essentially enables the user to build unbounded trees of types.
- Patterns are currently implemented for open questions but it would be appropriate to consider their use in closed (single- and multiple-response) questions as well.

- It should be possible to allow the user to select *one* of the possibly many lexical matches to use as a default format for reporting response data. This would allow, for example, a pattern to be built that accepts dates in a variety of formats but always reports them in dd/mm/ccyy format.

## 8. Conclusions

The original motivation for this work was an observation that responses to paper questionnaires give an insight into the ways that respondents perceive questions and the way that they think they are expected to respond (in terms of formatting their responses). That was allied to a personal view that many forms-based web applications were (and are) appallingly written in the sense that they apply onerous restrictions to formatting with little guidance as to the expected format of particular data fields.

The solution uses the well-researched and established technique of regular expression patterns for parsing input. An established (albeit a canonical rather than formal) standard as the basis means that translations for external platforms are possible. The main drawback for end-user applications is the arcane syntax of regular expressions. This problem has been overcome in two ways: firstly by building in patterns for common requirements thus removing the necessity for users to get involved in the low-level at all; and secondly by representing patterns as rich objects with identified properties which when assigned appropriate values behave in the same way as their tersely-specified relatives.

The resultant solution has produced some serendipitous side-products that enable patterns to be perceived as genuine response-type builders complete with values constraints and appropriate operations such that they are useful not only at data collection time but equally so at analysis time as well.

## References

"Regular Expressions", wikipedia article, [http://en.wikipedia.org/wiki/Regular\\_expression](http://en.wikipedia.org/wiki/Regular_expression)

"Postcodes and addresses explained", Royal Mail website,  
<http://www.royalmail.com/portal/rm/content1?catId=400120&mediaId=9200078>

"Mastering Regular Expressions", Friedl, J, O'Reilly and Associates, 2006

## About the Author

Dr Stephen Jenkins is Technical Director of Snap Surveys Ltd. and is responsible for the direction and development of Snap survey software. He has over 30 years' experience in the design, development and use of survey software. He is a founding member of The Triple-S Group and an author of the Triple-S survey interchange standard. He can be contacted at Snap Surveys Ltd., Mead Court, Thornbury, Bristol BS35 3UW, UK, or by email [sjenkins@snapsurveys.com](mailto:sjenkins@snapsurveys.com).