The Orkney Dictionary: Creating an Online Dictionary Efficiently from a Printed Book

Thomas Widmann, Phyllis Buchanan

Complexli Limited, 27 Kinloch Road, Newton Mearns, Glasgow G77 6LY, Scotland
E-mail: thomas@complexli.com, phyllis@complexli.com

Abstract

A great number of older dictionaries were compiled before the world of lexicography moved into the digital era. The result is that many older texts exist only in book format even though they contain a wealth of information that could still be extremely relevant today. A great deal of work went into these historical texts and some smaller languages and dialects are represented only in this format. Losing this information simply because the cost involved in digitising such resources is prohibitive would represent a wasted opportunity. In this paper we will demonstrate an efficient and cost-effective solution for converting these paper products into online resources. We will base the paper on the conversion of The Orkney Dictionary which we undertook in 2016.
In our approach, the book goes through the following phases: we began with the paper book, moved onto visual markup (HTML), this was converted to a simple tagging structure which formed the basis for the XML and then HTML. Finally the text was put into WordPress. Despite the numerous steps involved, many of them are standard components that can be reused, which is why it constitutes an efficient, low-cost way forward for retrodigitisation.

Keywords: XML conversion; online dictionaries; retrodigitisation

1. Introduction

A lot of dictionaries were compiled before the advent of digital lexicography, and today all that exists is a printed book. Many of these dictionaries are still of interest, for instance because they describe small or historical languages or dialects for which no digital resources exist.

However, the money available for digitising such books is often scarce, and thus many of these dictionaries never get the chance to move into the digital world.

In this paper, we shall demonstrate an efficient approach for converting a paper dictionary into an online product, based on the conversion of the Orkney Dictionary from a book (Flaws & Lamb, 1996) to a website (www.orkneydictionary.scot) which we undertook in 2016. In our approach, the book goes through the following phases:

1. Paper book
2. Visual markup (HTML)
3. Simple tagging
4. Fully nested XML tagging
5. HTML
6. WordPress

Although this process seems to contain many steps, many of them are standard components that can be reused, which is why this is an efficient, low-cost way to digitise an old dictionary.

2. The project

We were contacted in early 2016 by Simon W. Hall, who at the time was Education Scotland's Scots Language Coordinator in Orkney, with a view to turning Margaret Flaws and Gregor Lamb's seminal *Orkney Dictionary* from 1996 into an online dictionary.

This dictionary describes the Orkney (or Orcadian) dialect of Scots (the language derived from Old English that is spoken in the Scottish Lowlands and the Northern Isles, as well as parts of Ulster). The Orkney dialect is still widely spoken, for instance in local radio.

The project was funded by the Orkney Heritage Society, and the budget was quite limited compared to similar projects that we have undertaken for commercial dictionary publishers.

The Orkney Dictionary is not a large dictionary by any means, containing just over 2000 headwords on the Orkney-English side, and just shy of 1500 headwords on the other, as well as a few chapters describing the grammar and general orthographical principles.[1]

That said, it is the main dictionary describing the Orkney dialect, and copies of it are found in homes and schools everywhere on the islands. Making it available online for free was therefore of immense benefit to many people in Orkney.

We could have converted the dictionary to HTML directly from PDF without the detour via XML, but that would have made it harder to implement a decent search interface and live cross-references. We also thought that it would be a good opportunity to create a modern look, making good use of whitespace and colour. We therefore decided to undertake a proper conversion from PDF to XML instead, although this was going to make it challenging to stay within the budget.

The way we squared the circle was by reusing different components that we had created over a number of years.

3. The steps

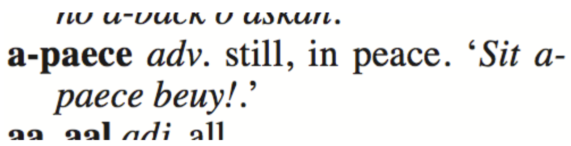
In the following we will look at the individual steps making up the conversion process.

3.1 Paper book to HTML

This first step was to convert the PDF file to a simple HTML file containing only visual mark-up.

In our case we were lucky enough to receive a PDF of the published book, but the process would have been similar if we had worked from a paper book, in which case we would have had to get it scanned or double-keyed instead.

A typical entry in *The Orkney Dictionary* looks like this:



no a-back o askan.
a-paece *adv.* still, in peace. '*Sit a-paece beuy!*.'
aa, aal *adj.* all

It should be clear from a quick glance that this is not an straightforward format to identify the structure from. For instance, italics are used for both part-of-speech labels and examples, and full stops are everywhere.

[1] These chapters were converted separately to WordPress pages, but this is not of any interest here, given the lack of lexicographic content.

In this case, we used an on-line service to convert the PDF to Word format, and then an OpenOffice plug-in to create XHTML. This was based on trial and error, and different PDF files might have been easier to convert using different tools.

We now had a HTML file containing entries such as this:

```
<p class="Textbody">
  <span style="font-family:Times;font-weight:bold;font-size:14.666667px"
    >a-paece </span>
  <span style="font-family:Times;font-style:italic;font-size:14.666667px"
    >adv. </span>
  <span style="font-family:Times;font-size:14.666667px"
    >still, in peace. ‘</span>
  <span style="font-family:Times;font-style:italic;font-size:14.666667px"
    >Sit a- paece beuy!</span>
  <span style="font-family:Times;font-size:14.666667px">.’ </span>
</p>
```

This is a rather neat example. Many entries were interrupted by `<p>` tags, and in a few cases the text was not even sequential, but skipped back and forwards between the two columns. In general it was a decent file to base the next step on, though.

3.2 HTML to simple tagging

We now needed to convert the HTML to our own simple tagging format. This is a way to identify logical elements such as headwords, translations and examples without worrying about creating any explicit structure yet. Of course there might be tags indicating the beginning of grammatical categories or of new senses, but nothing is nested at this stage.

This mark-up is optimised for manual editing in Emacs so that a lexicographer is able to correct any conversion errors efficiently. We also developed a few Emacs macros to make it easy to undertake common editing operations, such as splitting up a translation containing a comma, or upgrading a phrase (together with its translation and other associated information) to a full entry.

Apart from inline tags (such as `<b>...</b>`, which are left as-is, every tag starts on a new line, and a TROFF-like notation (i.e., `.tag` without an end tag) is used instead of XML syntax (`<tag>...</tag>`) to minimise typing.

In this case, the program would convert the above example to this:

```
.hw a-paece
.ps adv
.tr still, in peace
.qu Sit a-paece beuy!
```

The advantage of this tagging system is that any conversion errors will jump out immediately. For instance, if the beginning quotation mark after “peace” had not been correctly identified, we might have ended up with something like this:

```
.hw a-paece
```

```
.ps adv
.tr still, in peace. ‘
.ph Sit a-paece beuy!.’
```

It is very easy to see that something is wrong here – much easier than spotting an error in HTML, and much easier to fix than doing it in XML (where a change to the structure might be necessary).

In this case, the conversion program did a very good job – the only correction needed was to split up the two translations that had been separated by a comma, resulting in this:

```
.hw a-paece
.ps adv
.tr still
.tr in peace
.qu Sit a-paece beuy!
```

Another common issue was that many headwords had become phrases (especially when they were derivations), e.g.:

```
.hw blether
.ps v
.tr talk nonsense
.ps n
.tr chatterbox
.ph bletherskate
.ps n
.tr someone who talks nonsense
```

All we needed to do in order to change the structure here was to replace `.ph` with `.hw`:

```
.hw blether
.ps v
.tr talk nonsense
.ps n
.tr chatterbox

.hw bletherskate
.ps n
.tr someone who talks nonsense
```

The equivalent change in XML would have required much more typing (or complex macros).

To convert the HTML file to this simple tagging format, we wrote a Perl program making heavy use of regular expressions. It would first replace the `<span style="...">` tags with more intuitive tags (such as `<b>` and `<i>`), and then replace them with our simple tagging tags based on the context.

For instance, a bold chunk of text at the beginning of a paragraph would become a headword (`.hw`), anything following this (separated by a comma) would become an alternative form of the headword (.ha), and any other chunk of bold text would become a phrase (.ph). Any phrase enclosed in quotation marks would finally be turned into a quotation (.qu).

The program got many things right, and when it made an error, it was often very obvious and easy to fix, as described above.

3.3 Simple tagging to rich XML

At this point, we needed to design a DTD describing the resulting XML.

We considered using a standard TEI Dictionary structure (Text Encoding Initiative, 2016), but we thought that it was too verbose in places, made unnecessary distinctions for our purposes and yet conflated distinctions made in our source, so we created a custom DTD that mimicked the implicit structure adopted by the authors of *The Orkney Dictionary.*

This decision was aided by the fact that it seemed unlikely the data from the project would be reused elsewhere, given the low number of Orkney dialect speakers. If data sharing becomes important at a later date, it should be eminently possible to convert the data to a standard TEI structure.

It is important to bear in mind at this point that this conversion was being done on a very small project, so a decision had to be made quickly and pragmatically. In an ideal world, we would have spent some time exploring the XML structures used by similar projects and liaising with other experts in the field, but that would have left us with practically no time to undertake the actual conversion.

The relevant subset of our DTD relating to the entry we have been examining above looks as follows:

```
<!ELEMENT entry (hw, gram+)>
<!ELEMENT hw (#PCDATA)>
<!ELEMENT gram (pos, sense+)>
<!ELEMENT pos (#PCDATA)>
<!ELEMENT sense (tran+, quotes*)>
<!ELEMENT tran (#PCDATA)>
<!ELEMENT quotes (quote)>
<!ELEMENT quote (#PCDATA)>
```

In order to convert our simple tagging format to XML conforming to this DTD, we used our own conversion program (written in Perl, C, Flex and Bison) to convert it to highly structured XML (see Section 4 for more details on this program).

This program reads a description of the resulting XML file that is similar to the DTD, so if this has been done correctly, it should in theory always produce valid XML (apart from attribute values and inline tags and a few other things that are external to the program[2]).

[2] It would be relatively simple to extend the program with functionality to check that inline tags only get used in the correct locations, and that attribute values always are taken from a closed set, but we have found it is just as easy simply to validate the resulting files against the DTD afterwards.

It is completely reusable, and we have used it for many XML conversion projects over the years.

The relevant subset of the grammar that the conversion program reads looks as follows:

```
entry;hw gram+
hw;.hw
gram;pos sense+
pos;.ps
sense;tran+ quotes*
tran;.tr
quotes;quote
quote;.qu
```

It should be clear that this corresponds very closely to the DTD. Apart from the syntax, the biggest difference is that the `#PCDATA` bits have been replaced with the relevant tags used in our simple tagging system.

If the simple tagging input does not conform to this structure, the conversion program will produce an error message when it encounters the first offending line. For instance, if the `.ps` had been omitted, it would complain when it saw the `.tr` tag. Because of this, the conversion from our simple tagging format to XML is very safe.[3] However, running it can be quite an iterative process, requiring the lexicographer to correct the simple tags in the text when it cannot be converted.

As an example, imagine that the following entry were encountered:

```
.hw a-paece
.tr still
.tr in peace
.qu Sit a-paece beuy!
```

One would have three alternatives here: (1) To insert the missing `.ps` tag; (2) to amend the `gram` rule from `gram;pos sense+` to `gram;pos? sense+`; or (3) to convert "missing" `.ps` tags to a `<pos>` with a specific value that can then be corrected later. In this case, option (1) would clearly be best, but there are other cases where the other options might be preferable, for instance if the client wants to make any corrections themselves after the conversion has been completed.

In this case, our entry was transformed to the following bit of XML:

```
  <entry>
    <hw>a-paece</hw>
    <gram>
      <pos>adverb</pos>
      <sense>
        <tran>still</tran>
        <tran>in peace</tran>
```

[3] It will always create XML that validates against the DTD if the grammar rules have been written correctly. However, there is no guarantee that the XML tags will have been used in a semantically correct fashion, for instance if the simple tags were wrong to start with.

```
        <quotes>
          <quote>Sit a-paece beuy!</quote>
        </quotes>
      </sense>
    </gram>
  </entry>
```

At the end of this stage, the entire dictionary had been converted to XML, which could be edited using any XML editor or a proper dictionary editing system such as IDM's DPS. In the case of The Orkney Dictionary, we preferred to implement all corrections in the simple tagging format and then reconvert it, though, simply because our simple tagging system is easier to work with than the resulting XML structure.

3.4 XML to HTML

We now needed to convert the XML to HTML. For this purpose, we wrote a simple XSLT program.[4] The resulting HTML consisted mainly of `<div>`s and `<span>`s:

```
<div class="entry">
  <span class="hw">a-paece </span>
  <span class="pos">adverb </span>
  <span class="tran">still</span>
  <span class="punct"> &#8226; </span>
  <span class="tran">in peace</span>
  <span class="punct">  &#9758; </span>
  <span class="quote">‘‘Sit a-paece beuy!’’</span>
</div>
```

We did not have a specific design brief, but basically tried to find a modern dictionary design that the client would be happy with. However, given the simplicity of the XML structure used here, we do not find it likely that any design proposals would have been too hard to implement had the client so desired.

We also developed a previewer based on the formatting program. This was optimised to highlight any conversion errors by ensuring that all tags would output in a distinctive fashion. This was not designed to be pretty, but it was a great way to find the last remaining conversion errors.

We also created a CSS file to display the HTML, and the result was the same that can be seen on the website today.

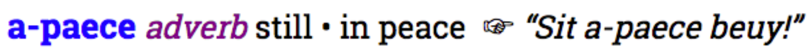
3.5 HTML to WordPress

The resulting HTML was then stored in a few MySQL tables and uploaded to a standard WordPress installation with some added search functionality. We also quickly converted the chapters describing the grammar and orthography of the Orkney dialect of Scots to HTML and made them available as WordPress pages.

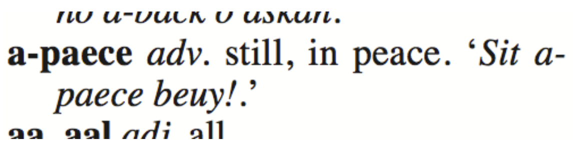
[4] There is probably no point in describing the XSLT program in any detail, given that it exhibits no features of great interest.

The WordPress theme is a child theme[5] of the *Twenty Sixteen* theme (WordPress.org, 2016), which implements the search functionality (including fuzzy matching) in PHP and incorporates the CSS code necessary to make the entries display correctly. Most of it could be reused with very few changes to create other online dictionaries.

At this point, the entry we have been looking at above now looks like this:


a-paece *adverb* still • in peace ☞ *"Sit a-paece beuy!"*

This is quite a difference from our starting point:


no a-back o askan.
a-paece *adv*. still, in peace. '*Sit a-paece beuy!*.'
aa, aal *adj.* all

The XML can of course also be used for other purposes. It would for instance be relatively easy to create ICML from it in order to typeset the dictionary in InDesign (which is something we have done for another client), and the HTML could also be used to create a smartphone app. The cost of developing a dictionary app for The Orkney Dictionary would probably be prohibitive, but there is no reason why the costs could not be shared by a number of similar projects, and this is something we are currently looking into.

4. Our conversion program

In this section we shall describe our own conversion program (written in Perl, plain C, Flex and Bison) that we used to convert our simple tagging format to fully nested XML.

This program is extremely flexible and allows the conversion of many types of non-nested data to different XML structures. We have only used it for dictionary conversions, but there is no reason why it could not be used for other purposes as well.

It consists of two parts: A parser written in C, Flex and Bison, and a grammar preprocessor written in Perl that transforms the grammar (corresponding to the DTD) into Bison code.

Flex and Bison (open-source alternatives to Lex and Yacc) are standard programming tools used for writing parsers, e.g., for programming languages. Flex tokenises the input, and Bison takes these tokens and uses them to build a syntax tree. (Bison can only write parsers for context-free grammars.[6]) Variants for these tools exist for several different programming languages, but we used the one that produces C code, for the simple reason

[5] A WordPress theme is a collection of files that work together to produce a graphical interface with an underlying unifying design for a blog. A child theme is a theme that inherits the functionality and styling of another theme, called the parent theme. Child themes are the recommended way of modifying an existing theme.

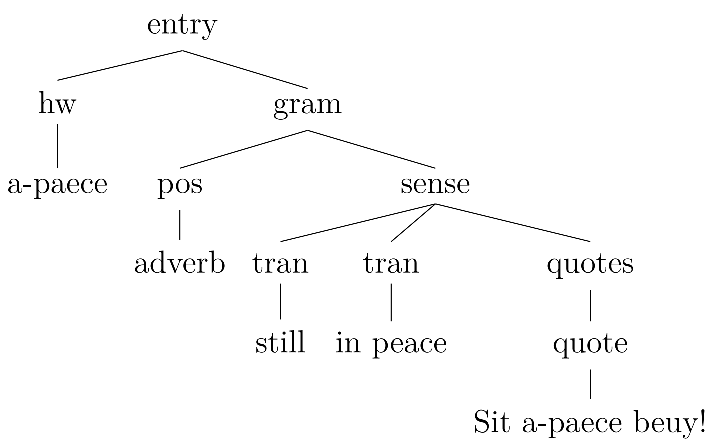
[6] A context-free grammar is a set of recursive rewriting rules that all have a single non-terminal on their left hand side.

that this was the one we were already familiar with.[7] The fact that the parser is written in plain C means that it is lightning fast. If no preprocessors or postprocessors are used, our parser can convert a large dictionary to XML in just a few seconds.

To understand how these tools can be used for producing XML from a flat structure, we need to realise that an arbitrary chunk of XML can be visualised as a tree. For instance, let us have a look at this entry:

```
<entry>
  <hw>a-paece</hw>
  <gram>
    <pos>adverb</pos>
    <sense>
      <tran>still</tran>
      <tran>in peace</tran>
      <quotes>
        <quote>Sit a-paece beuy!</quote>
      </quotes>
    </sense>
  </gram>
</entry>
```

The equivalent tree notation would look like this:



Bison is great at building trees like this. The syntax looks like this (`Dhw` and `CONTENTS` are two of the tokens produced by Flex, corresponding to `.hw` and the following text):

```
entry:
  hw gram_plus      { /* C code to add this to the tree */ }
| hw pron gram_plus { /* C code to add this to the tree */ }
/* ... */
;

hw:
  Dhw CONTENTS      { /* C code to add this to the tree */ }
```

[7] In fact, when we wrote the parser, we had to dust off our old copy of Aho et al. (1986), which was a very nostalgic experience.

```
;

gram_plus:
  gram              { /* C code to add this to the tree */ }
| gram_plus gram    { /* C code to add this to the tree */ }
;

/* many more rules here */
```

Bison will then write code that will select the correct rule based on the input.

It would be quite possible to write these Bison rules manually, but it would get rather tedious. Because of this, we have written a Perl program that makes it possible to specify the syntax in a much more compact way that almost mimics a DTD.

As a very simple example, let us imagine the input format only contains the tags `.hw` and `.tr`:

```
.hw a-paece
.tr still
```

Let us assume that this minimal entry should end up looking like this:

```
<entry>
  <hw>a-paece</hw>
  <tr>still</tr>
</entry>
```

This can be achieved with the following grammar rules:

```
entry;hw tr
hw;.hw
tr;.tr
```

If we want to allow sequences of headwords and translations, we can achieve this by adding `+` after the relevant tag, just like one would do in a DTD:

```
entry;hw+ tr+
hw;.hw
tr;.tr
```

In the same way, we can use `?` and `*` (again with the same semantics as in DTDs), e.g.:

```
entry;hw+ pos? tr+ quotation*
hw;.hw
tr;.tr
pos;.ps
quotation;.qu
```

It is also possible to add extra grouping tags. For instance, if every `.ps` tag starts a new grammatical category, we could write it like this:

```
entry;hw+ gram+
gram;pos tr* quotation*
hw;.hw
tr;.tr
pos;.ps
quotation;.qu
```

Although brackets cannot be used (yet), they can be emulated by using non-outputting grouping tags (starting with an underscore). For instance, if a `<gram>` can consist of a `<pos>` and a sequence of either `<tr>`s or `<def>`s, it could be expressed as follows:

```
gram;pos _trs_or_defs
_trs_or_defs
 ;tr+
 ;def+
```

We might add bracket notation in a future version of the parser to make such expressions follow the DTD syntax more closely.

Another feature is what we call named rules: These contain an underscore after the first word, such as `sense_first`. The tag they generate contains only the first word (`<sense>` in this case), but it increases readability and makes it possible to have many rules generating the same tag.

The tree that is built can be highly nested. For instance, consider the following grammar:

```
tag0;tag1
tag1;tag2
tag2;tag3
tag3;tag4
tag4;.tag
```

Based on this, the parser would turn the single line `.tag Hello world` into the following chunk of XML:

```
<tag0>
  <tag1>
    <tag2>
      <tag3>
        <tag4>Hello world!</tag4>
      </tag3>
    </tag2>
  </tag1>
</tag0>
```

The only real shortcoming of our parser is that it cannot look ahead in order to choose between two options – it knows the preceding lines and the current one, but it has no idea about what it will encounter later. For instance, imagine a situation where `.tr` tags have to be incapsulated in `<sense>` tags if and only if any `.tr` tag within the same `<gram>` structure is preceded by an `<lb>` tag. It would be logical to write rules such as these:

```
gram
```

```
 ;pos tr+
 ;pos _senses
_senses;sense_first? sense_extra+
sense_first;tr
sense_extra;lb+ tr
```

However, this would not work on the following structure:

```
.hw a-paece
.ps adverb
.tr still
.lb lit
.tr in peace
```

The parser would enter the first `gram` rule and then get stuck when it found the first label.

To deal with such situations, it is necessary to write preprocessors (we use Perl) to make the structure easier to parse by inserting extra tags. To resolve the conflict described above, the preprocessor might insert `.sense` tags like this:

```
.hw a-paece
.ps adverb
.sense
.tr still
.sense
.lb lit
.tr in peace
```

In this way, it is possible to generate XML for even highly complex and ambiguous structures.

In theory, it should also be possible to use Bison's *GLR* mode (this is an extension to handle nondeterministic and ambiguous grammars) instead of writing these preprocessors. GLR parsers handle Bison grammars that contain no unresolved conflicts in the same way as deterministic parsers. However, when there are unresolved shift/reduce and reduce/reduce conflicts, GLR parsers use the simple expedient of doing both, effectively cloning the parser to follow both possibilities. Each of the resulting parsers can again split, so that at any given time, there can be any number of possible parses being explored. Each of the cloned parsers eventually meets one of two possible fates: either it runs into a parsing error, in which case it simply vanishes, or it merges with another parser, because the two of them have reduced the input to an identical set of symbols (Free Software Foundation, 2015).

We have not explored this, but it would be worthwhile looking into it in the future.

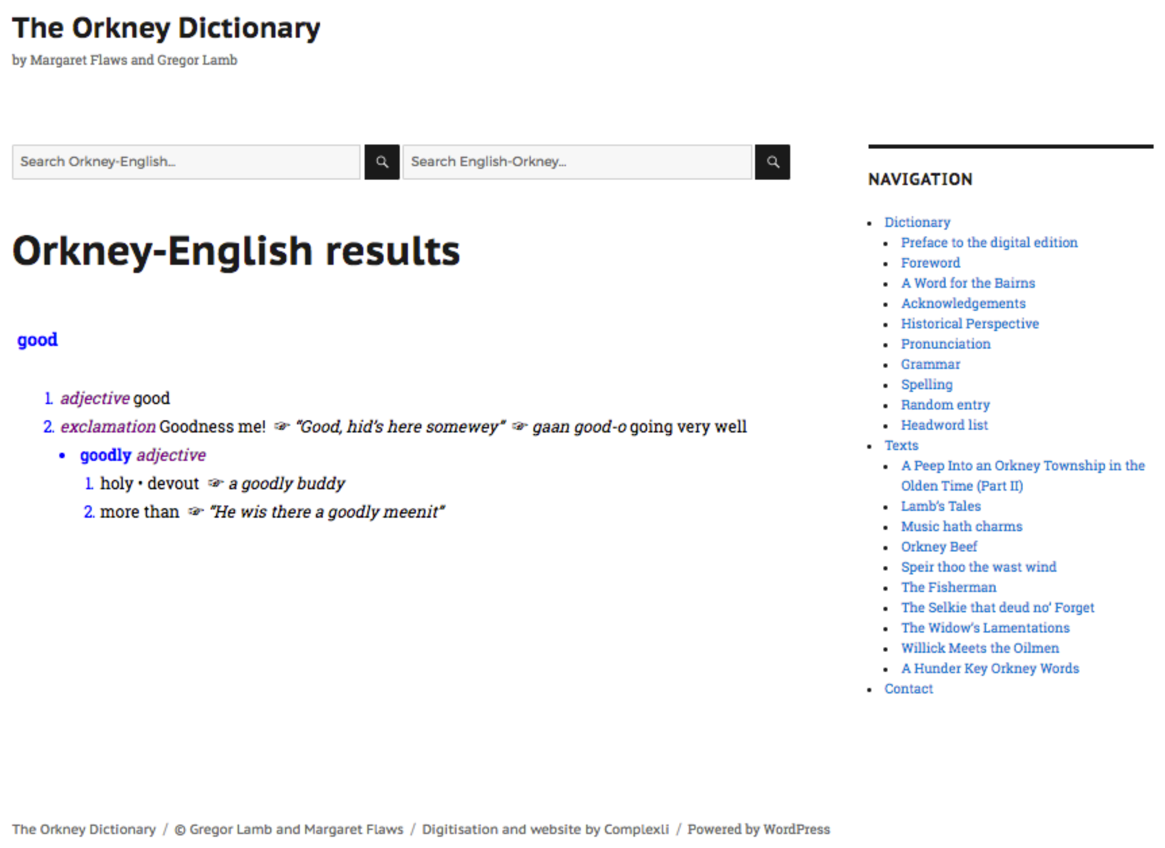
We believe that our context-free parser is a great tool for converting a flat structure to XML. It is sadly not possible to make it freely available at the moment, as it would require a great deal of work to extend and document it before this could happen. It would not be particularly hard for other programmers familiar with Lex/Flex and Yacc/Bison to write something similar.

5. Discussion

It would have been much easier to take the HTML and put it on the web directly. However, going through the steps we did conferred many advantages:

1. It allowed us to present the dictionary in a modern way rather than being tied down to the old format.
2. Converting the data enabled us to catch many typos and inconsistencies that had been overlooked in the book.
3. Having proper XML made it easy to implement the search functionality (because it was clear what needed to be indexed), and it also made it simple to create live cross-references on the website.
4. If the authors decide to make any additions or alterations to the data in the future, they can do so using a modern dictionary editing system instead of a word processor.
5. It becomes possible to typeset the book in InDesign – this is unlikely to happen soon, however, as the book has recently been reprinted.
6. It becomes possible to create a smartphone app. This is likely to be the next step in this project.

It was only possible to deliver this project on budget because we already had our conversion program, which we had developed for other XML conversion projects. Most of the time was spent on converting the HTML into our simple tagging format, and on correcting remaining errors manually in this format.



The Orkney Dictionary

by Margaret Flaws and Gregor Lamb

Orkney-English results

good

1. *adjective* good
2. *exclamation* Goodness me! ☞ *"Good, hid's here somewey"* ☞ *gaan good-o* going very well

- **goodly** *adjective*
  1. holy • devout ☞ *a goodly buddy*
  2. more than ☞ *"He wis there a goodly meenit"*

NAVIGATION

- Dictionary
  - Preface to the digital edition
  - Foreword
  - A Word for the Bairns
  - Acknowledgements
  - Historical Perspective
  - Pronunciation
  - Grammar
  - Spelling
  - Random entry
  - Headword list
- Texts
  - A Peep Into an Orkney Township in the Olden Time (Part II)
  - Lamb's Tales
  - Music hath charms
  - Orkney Beef
  - Speir thoo the wast wind
  - The Fisherman
  - The Selkie that deud no' Forget
  - The Widow's Lamentations
  - Willick Meets the Oilmen
  - A Hunder Key Orkney Words
- Contact

The Orkney Dictionary / © Gregor Lamb and Margaret Flaws / Digitisation and website by Complexli / Powered by WordPress

Figure 1: A screenshot from orkneydictionary.scot showing a sample entry

The online version of the dictionary (see the screenshot in Figure 1) has been very well received in Orkney. The past year has seen slightly fewer than 4,000 unique visitors, which might seem like a low figure, but it should be remembered that Orkney's total population is not much more than 20,000.

6. Conclusion

We have demonstrated that it is possible to digitise a paper dictionary and to create a website for it on a small budget without sacrificing quality.

We believe this is good way to convert legacy dictionaries into XML and onto the web. The key is to use standard components that can be reused in other projects, and to have simple data formats that are easy to edit using free tools.

7. References

Aho, A.V., Sethi, R. & Ullman, J.D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

Flaws, M. & Lamb, G. (1996). *The Orkney Dictionary.* Orkney Language and Culture Group.

Free Software Foundation (2015). GNU Bison – The Yacc-compatible Parser Generator. http://www.gnu.org/software/bison/manual/.

Text Encoding Initiative (2016). P5: Guidelines for Electronic Text Encoding and Interchange: Dictionaries. http://www.tei-c.org/release/doc/tei-p5-doc/en/html/DI.html.

WordPress.org (2016). Twenty Sixteen. https://en-gb.wordpress.org/themes/twentysixteen/.