Report on the Bellows Language

Rick Hanson

Project Report For CS4115 PLT, Fall 2007.

Contents

CHAPTER 1

Introduction

Bellows is a file format description language that is designed to be utilized by calling programs which want to access the contents of some object file described by a Bellows program. Such a program resembles, and can be thought of as, a description of the object file's format. The Bellows compiler will convert the source into an executable which will be able to read any object file that conforms to the format, and which will output the object file's contents as tagged XML.

The object files are flat (line-oriented, ASCII text) and typically contain collections of multi-line records. Usually one object file contains more than one collection of different types of records in the object file. Hence, the object file can be thought of as a heterogeneous collection of data records, and the Bellows programming constructs handle such descriptions quite naturally.

1. An Example

The following is an example Bellows program, paired with an example object file.

Bellows Source File

```
 1 format mytimefile
 2   'Version 42'
 3   title
 4   numitems
 5   each item num=numitems
 6     begintime endtime moreflag
 7     if moreflag == 1
 8       more1 more2 more3 more4
 9     fi
10   chae
11   lastupdatedby
```

Object File

```
1 Version 42
2 Table_of_items
3 3
4 6:00 9:00 0
5 2:03 12:45 1
6 0.000 42.000 1.000 2.000
7 9:33 10:12 0
8 Rick
```

The Bellows source can be read as follows. The object file in question is called `mytimefile`. The first line of the object file contains the literal string `Version 42`. The second line of the object file contains a string which we call `title`, i.e. we assign this string to the variable `title`; in the case of the object file given, the string `Table_of_items` will be assigned to `title`. The third line of the object file contains an integer which we call `numitems`.

Subsequent lines in the object file, immediately after the third line, are groups of lines which correspond to records described by the `each`/`chae` block in the Bellows source. The identifier after the `each` keyword, in this case `item`, is the name of the record; the curious descriptor `num=numitems` says that there are `numitems` of such records which follow. In this case, there should be 3 records in the object file which should match the record description.

The record description, in the `each/chae` block, says that an `item` record can have one or two lines. If the third field of the record's first line (denoted by the field name `moreflag`) is equal to (==) `1`, then the record has a second line; otherwise it's a one-liner.

Finally, the last line of the Bellows source says that we assign the string on the line immediately after the three `item` records, namely `Rick`, to the variable name `lastupdatedby`.

It turns out that the example object file indeed conforms to the format described by the example Bellows source file, and so the Bellows executable will return the following XML output (corresponding to the object file's contents).

```
<mytimefile>
  <literal>Version 42</literal>
  <title>Table_of_items</title>
  <numitems>3</numitems>
  <items>
    <item>
      <begintime>6:00</begintime>
      <endtime>9:00</endtime>
      <moreflag>0</moreflag>
    </item>
    <item>
      <begintime>2:03</begintime>
      <endtime>12:45</endtime>
      <moreflag>1</moreflag>
      <more1>0.000</more1>
      <more2>42.000</more2>
      <more3>1.000</more3>
      <more4>2.000</more4>
    </item>
    <item>
      <begintime>9:33</begintime>
      <endtime>10:12</endtime>
      <moreflag>0</moreflag>
    </item>
  </items>
  <lastupdatedby>Rick</lastupdatedby>
</mytimefile>
```

2. Exception Handling

When the contents of the object file fail to conform to the format description of the Bellows source file associated to it, the Bellows program should either return the last legal match of identifier (tag) to object file data item, or return the first item in the object file (by line and column number) which fails to conform to the Bellows format, and return the associated tag name and type. Alternatively, the implementation can output legal matches (in the XML file) up to the point of a non-match and halt, flushing the output buffer—the user should then be able to conclude precisely where the data mismatch error occurs.

3. Data Type Declarations

Bellows programs may exhibit an optional section which contains data type declarations for the identifiers which occur in the format section of the program. For instance, the following is our previous example program, now furnished with data types.

```
format mytimefile
  'Version 42'
  title
  numitems
  each item num=numitems
    begintime endtime moreflag
    if moreflag == 1
      more1 more2 more3 more4
    fi
  chae
  lastupdatedby
tamrof

type
  title         String
  numitems      Integer
  begintime     String
  endtime       String
  moreflag      NumBool
  more1         Float
  more2         Float
  more3         Float
  more4         Float
  lastupdatedby String
epyt
```

Any item declared in the `type` section informs the executable program that that particular item in the object file should go through data type validation during the reading process.

CHAPTER 2

Language Reference

The source code of Bellows language program, viewed from a high level, is a collection of, possibly nested, blocks of code. Each block corresponds to a substructure in the overall heirarchical structure of the object file (raw data) to which the program corresponds. Another way to think of a Bellows program is that it has the structure of the object data which one would see in the resultant XML output, but with the data itself elided.

To put everything more precisely, we next discuss the language's lexical conventions and grammar.

1. Lexical Conventions

1.1. The End of the Line. Bellows source files themselves are line-oriented to reflect the line-orientation of the object files which Bellows programs read. This makes easier the human consumption of the Bellows source file. For these reasons, we define a special token which represents the end of a line of source code.

$$\textbf{eol} \;\rightarrow\; (\boxed{\texttt{\textbackslash r}}?\ \boxed{\texttt{\textbackslash n}})+$$

1.2. Whitespace. Whitespace is defined to be either a space character or a tab character, except for such characters delimited between matched pairs of single quotes (as those characters would be part of a string literal). Whitespace will be discarded by the scanner.

$$\textit{ws} \;\rightarrow\; (\text{␣} \mid \boxed{\texttt{\textbackslash t}})+$$

1.3. Intrinsic Operators. Bellows has binary predicate operators which are designed to be used in the test portion of the conditional construct. They are >, >=, <, <=, ==, != , which stand for "greater than", "greater than or equal to", "less than", "less than or equal to", "equal to" and "not equal to" respectively.

1.4. Keywords. Keywords are a special subset of the set of finite strings of alphanumeric characters and are reserved by the compiler for special purposes.

$$
\begin{array}{rcl}
\textbf{format} & \longrightarrow & \boxed{\texttt{format}} \\
\textbf{tamrof} & \longrightarrow & \boxed{\texttt{tamrof}} \\
\textbf{type} & \longrightarrow & \boxed{\texttt{type}} \\
\textbf{epyt} & \longrightarrow & \boxed{\texttt{epyt}} \\
\textbf{if} & \longrightarrow & \boxed{\texttt{if}} \\
\textbf{fi} & \longrightarrow & \boxed{\texttt{fi}} \\
\textbf{each} & \longrightarrow & \boxed{\texttt{each}} \\
\textbf{chae} & \longrightarrow & \boxed{\texttt{chae}} \\
\textbf{string} & \longrightarrow & \boxed{\texttt{String}} \\
\textbf{integer} & \longrightarrow & \boxed{\texttt{Integer}} \\
\textbf{float} & \longrightarrow & \boxed{\texttt{Float}} \\
\textbf{numbool} & \longrightarrow & \boxed{\texttt{NumBool}}
\end{array}
$$

1.5. Identifiers. An identifier is a variable name which, in the Bellows source, is denoted by a string of alphanumeric characters, starting with a letter.

$$
\begin{array}{rcl}
\textit{letter} & \longrightarrow & \boxed{\texttt{a}} - \boxed{\texttt{z}} \mid \boxed{\texttt{A}} - \boxed{\texttt{Z}} \\
\textit{digit} & \longrightarrow & \boxed{\texttt{0}} - \boxed{\texttt{9}} \\
\textit{nzdigit} & \longrightarrow & \boxed{\texttt{1}} - \boxed{\texttt{9}} \\
\textbf{id} & \longrightarrow & \textit{letter}\ (\textit{letter} \mid \textit{digit})*
\end{array}
$$

1.6. Literals. String literals are denoted by a sequence of characters between a pair of single quotes, e.g. `'Version 42'`. Natural numbers (literals) are a sequence of number characters (i.e. `0`, `1`, `2`, . . . , `9`), except for the one-length sequence `0` (since obviously 0 is not a natural number). This reference also calls natural numbers *positive integers*. Numeric literals are a sequence of number characters, optionally preceded by a minus sign, then followed by an optional sequence of number characters preceded by a period.[1]

$$
\begin{array}{rcl}
\textbf{strlit} & \longrightarrow & \boxed{\texttt{'}}\ \neg(\textbf{eol} \mid \boxed{\texttt{'}})*\ \boxed{\texttt{'}} \\
\textbf{natlit} & \longrightarrow & \textit{nzdigit}\ \textit{digit}* \\
\textbf{numlit} & \longrightarrow & \boxed{\texttt{-}}?\ \textit{digit}+\ (\boxed{\texttt{.}}\ \textit{digit}+)?
\end{array}
$$

2. Grammar and Semantics

A Bellows program has a format section and an optional type section. The optional type section can occur before or after the format section.

$$
\begin{array}{rcl}
\textit{program} & \longrightarrow & \textit{formatsection typesection}? \\
& \mid & \textit{typesection formatsection}
\end{array}
$$

[1]While natural numbers are numbers, according to the definitions, we assume that the scanner (or parser) we implement will be able to resolve the potential ambiguity introduced here, by way of classifying natural numbers as **natlit**s and all other numbers which are not natural numbers as **numlit**s. Hence, we should view **numlit**s as "numbers other than natural numbers."

A type section is delimited by pairs of lines which begin with the keywords `type` and `epyt` respectively. Between these lines are type specifications. A type specification is a line of code starting with an identifier (which ostensibly occurs in the format section) followed by some whitespace and then a type.

$$\begin{aligned} \textit{typesection} &\rightarrow \textbf{type eol } \textit{typespec}+ \textbf{ epyt eol} \\ \textit{typespec} &\rightarrow \textbf{id } \textit{type} \textbf{ eol} \\ \textit{type} &\rightarrow \textbf{string} \mid \textbf{integer} \mid \textbf{float} \mid \textbf{numbool} \end{aligned}$$

A format section is delimited by pairs of lines which begin with the keywords `format` and `tamrof` respectively. Between these lines are format specifications. The identifier after the `format` keyword designates the name of the root element of the XML file which corresponds to the object file.

$$\begin{aligned} \textit{formatsection} &\rightarrow \textbf{format } \textit{root} \textbf{ eol } \textit{formatspec}+ \textbf{ tamrof eol} \\ \textit{root} &\rightarrow \textbf{id} \end{aligned}$$

A format specification is a piece of code which describes the format of a piece of corresponding data in an object file. The format specification is either a simple data line, which contains a space-separated list of identifiers or literals, or a block of such data lines.

$$\begin{aligned} \textit{formatspec} &\rightarrow \textit{dataline} \mid \textit{block} \\ \textit{dataline} &\rightarrow (\textbf{id} \mid \textit{literal})+ \textbf{ eol} \\ \textit{literal} &\rightarrow \textbf{strlit} \mid \textbf{natlit} \mid \textbf{numlit} \end{aligned}$$

A block of data lines is either an `if` block or an `each` block.

$$\textit{block} \rightarrow \textit{ifblock} \mid \textit{eachblock}$$

The `if` block is delimited by a pair lines which start with the keywords `if` and `fi`, respectively. The `if` block contains one or more format specifications. There is a conditional test written after the keyword `if` which when it holds, the format specifications in the `if` block must match the object file data; otherwise the `if` block format specifications do not apply to the object file data.

$$\textit{ifblock} \rightarrow \textbf{if } \textit{test} \textbf{ eol } \textit{formatspec}+ \textbf{ fi eol}$$

Conditional tests look like the infix relational operation xRy, where R is any one of the relational operators discussed in the previous section: >, >=, <, <=, ==, `!=`. The operands x and y can be either an identifier or a literal, but they cannot both be a literal. The ordering relations for numbers have the same semantics as in the algebra of real numbers. The ordering relations on strings are based on the lexicographical ordering of the constituent ASCII characters. The equality relations for numbers are such that the two numbers should have the same type and internal value. Equality for strings is such that the two strings must have the same length and their respective ASCII character values must be equal.

$$\begin{aligned} \textit{test} &\rightarrow \textbf{id } \textit{relop} \ (\textbf{id} \mid \textit{literal}) \\ &\mid \ \ \textit{literal relop } \textbf{id} \\ \textit{relop} &\rightarrow \boxed{>} \mid \boxed{>=} \mid \boxed{<} \mid \boxed{<=} \mid \boxed{==} \mid \boxed{!=} \end{aligned}$$

The `each` block is delimited by lines which start with the keywords `each` and `chae`, respectively. The `each` block contains one or more format specifications which define a record, and we call this the record specification. There is a record name, i.e. an identifier, after the keyword `each`. After the record name, one or more qualifiers should follow. If a qualifier is `num=N`, then it means that precisely `N` such records should be matched in the object file. (Of course, `N` should be an existing identifier bound to a positive integer.) If a qualifier is `end='endofrecords'`, then the executable Bellows program continually reads (matches) records in the object file, until it encounters a line which matches `endofrecords`. The end qualifier is used when the Bellows program cannot know statically (beforehand) how many such records are to be read. If a qualifier is `sep='endofthisrecord'`, then when the executable is ready to read the first line of the next record, but instead encounters a line which matches `endofthisrecord`, it will ignore this line and assume that the next line of input is to match the first line of the next record.

An `each` block must have one of either a `num`-qualifier or an `end`-qualifier. The RHS of a `num`-qualifier should be a positive integer literal or an identifier bound to a positive integer. The RHS of a `sep`-qualifier or a `end`-qualifier should be a string literal or an identifier bound to a string. The reader should also note that the lexemes occurring in the qualifier, namely `num`, `sep` and `end`, are not reserved words per se. One should be able to use them as identifiers in other contexts.

Since, the record specification is designed to match a certain positive number of data records in the object file, it should be clear that the `each` block is a looping construct.

$$\begin{array}{rcl}
\textit{eachblock} & \rightarrow & \textbf{each}\ \textit{recordname}\ \textit{qualifier}+\ \ \textbf{eol}\ \textit{formatspec}+\ \ \textbf{chae eol} \\
\textit{recordname} & \rightarrow & \textbf{id} \\
\textit{qualifier} & \rightarrow & \textit{numqualifier} \mid \textit{strqualifier} \\
\textit{numqualifier} & \rightarrow & \boxed{\texttt{num}}\ \boxed{=}\ (\textbf{id} \mid \textbf{natlit}) \\
\textit{strqualifier} & \rightarrow & (\ \boxed{\texttt{sep}} \mid \boxed{\texttt{end}}\ )\ \boxed{=}\ (\textbf{id} \mid \textbf{strlit})
\end{array}$$

CHAPTER 3

Compiler Interface

Supposing that the file name of the Bellows source is `myprog.bel`, the command line incantation

```
bellowsc myproj.bel
```

will produce the executable script `myprog`. This compiled script can then be called by any program or user to produce an XML file `output.xml` from an object file `objfile`, as follows.

```
myprog objfile > output.xml
```

CHAPTER 4

Lessons Learned

Other than the obvious, and hackneyed, "I should have started earlier" observation, here are some observations I note for this project.

Designing a language is a good way to architect a system.

We didn't implement a large system in this project. However, even though this project's scope was relatively small, one can easily extrapolate from it how implementing a domain language for a system can, in many cases, be a big design win. In the case of our little system, we could have implemented it without implementing a specialized language first: we could have just encoded the object file's format into a piece of monolithic code. Indeed, this author did this on many occasions, only to be pained by having to change the code every time the input file's format changed (via another party). Designing a special language which specifically handles the changing file formats is a huge advantage when some file format changes, or when a novel file is introduced to the database.

In the general case, designing a system implementation language as a fundamental abstraction layer in one's system, provides a great deal of flexibility to handle future system interface changes. This flexibility amounts to *development-time scalability* (for as much of the system as is written in the special language) and can be seen as analogous to the *run-time scalability* about which software engineers like to discuss.

Also, it is much easier to write high-level code. You write fewer lines of code to achieve the same result in the lower-level language. But perhaps, the bigger gain is that you are writing programs in a vocabulary suited to the task at hand. This makes the expression more natural. Writing in the lower-level language without resorting to high-level constructs is akin to having to explain oneself over and over again vice reaching for handy, commonly-known nomenclature. (Being mentally stuck in the low-level programming paradigm is also known as the "Gee I seem to be writing a lot of for loops" phenomenon.)

To be sure, if the lower-level language had a powerful enough abstraction facility, one could write the abstractions in the lower-level language and then use the abstractions to implement the rest of the system. Fair enough, but this is just designing a special language again—it's just that, in this case, the language is embedded in the lower-level, or host, language by way of creating and using the appropriate abstractions. The difference is the same: a powerful software development technique based on "little languages" (hat tip, Jon Bentley).

Python-style block-wise indentation is not as easy to implement as it would seem.

I like Python-style blocks and I naively thought that they wouldn't be so hard to implement. I had this as a requirement in my language proposal and before finalizing the draft language reference manual, in which I had to implement a lexer and parser, I quickly dropped the requirement.

It was much harder than I thought. I would have had to have the lexer pass back a token with the whitespace prefix on each input line, so that the parser could then compute the extent of the block. Then I thought what do you do if you encounter a group of lines indented by i spaces followed by a group of lines indented by k spaces, followed by a group of lines indented by j spaces, such that $i < j < k$. Then, I would have to implement a policy whereby the parser would determine which block the group of lines indented by j spaces belonged to. Another issue is how to handle tab characters in the indentation scheme.

I didn't relish any of these thoughts; so I chickened out and scrapped the idea for now.

Run-time speed was not issue in this project.

I wrote the compiler to compile to Scheme code first. Then, if speed were an issue, I could always have the PLT compiler compile the Scheme code to a native executable. It turns out that, even though I ran out of time to possibly implement such a "back end", I didn't need the speed for all practical purposes, after all.

CHAPTER 5

Source Code

1. bellowsc

```
#!/bin/bash

mzscheme="/usr/local/bin/mzscheme"

if (( $# != 1 )); then
  echo "" >&2
  exit 42
fi

objname=${1}obj
exename=${1%.bel}

rm -f debug-bellows ${objname}
$mzscheme -f bellows-compile.scm $1 >/dev/null
rm -f debug-bellows

gawk '
BEGIN { print "#!'${mzscheme}' -qr"; }
{
 if ( /;;;;serials;;;;/ ) while ( getline < "'${objname}'" > 0 ) print;
 else                     print;
}' bellows-reader.scm > ${exename}

chmod +x ${exename}

exit
```

2. bellows-compile.scm

```
(require "bellows-parser.ss"
         (lib "serialize.ss"))

(define srcfile (car (vector->list (current-command-line-arguments))))
```

```
(define (slurp-file filepath)
  (with-input-from-file filepath
    (lambda ()
      (let loop ((iline (read-line)) (acc '()))
        (if (eof-object? iline)
            (apply string-append (reverse! acc))
            (loop (read-line) (cons (string-append iline "\n") acc)))))))

(define source-string (slurp-file srcfile))

(define ast (bellows-parser (make-lexer source-string)))

(define objname (string-append srcfile "obj"))

(with-output-to-file objname
  (lambda ()
    (write `(define serial-ast ',(serialize ast)))
    (write `(define serial-vars ',(serialize vars)))))

(exit)
```

3. bellows-parser.ss

```
(module bellows-parser mzscheme
  (require (lib "lex.ss" "parser-tools")
           (lib "yacc.ss" "parser-tools")
           (prefix : (lib "lex-sre.ss" "parser-tools"))
           (lib "list.ss" "srfi" "1"))

  (provide vars get-variable-type get-variable-value
           set-variable-type! set-variable-value!
           bellows-lexer bellows-parser make-lexer)

  (define-tokens regular (ID STRLIT NATLIT NUMLIT))
  (define-empty-tokens keywords
    (EOF EOL GT GTE LT LTE EQ NEQ
         BIND
         ;;NUM SEP END
         FORMAT TAMROF TYPE EPYT IF FI EACH CHAE
         STRING INTEGER FLOAT NUMBOOL))

  (define-lex-abbrevs
    (eol (:+ (:: (:? #\return) #\newline)))
    (letter (:or (:/ "a" "z") (:/ "A" "Z")))
    (digit (:/ "0" "9"))
    (nzdigit (:/ "1" "9")))

  (define bellows-lexer
    (lexer ((eof) (token-EOF))
           (eol (token-EOL))
           (blank (bellows-lexer input-port))
           ;; The following three are punted to the parser.
           ;;("num" (token-NUM))
           ;;("sep" (token-SEP))
           ;;("end" (token-END))
           (">" (token-GT))
           (">=" (token-GTE))
           ("<" (token-LT))
           ("<=" (token-LTE))
           ("==" (token-EQ))
           ("!=" (token-NEQ))
           ("=" (token-BIND))
           ((:: letter (:* (:or letter digit)))
            (keyword-filter lexeme))
           ((:: nzdigit (:* digit)) (token-NATLIT lexeme))
           ((:: (:? "-")
                (:+ digit)
                (:? (:: "." (:+ digit))))
            (token-NUMLIT lexeme))
           ((:: #\' (:* (:~ #\newline #\')) #\')
            (token-STRLIT (substring lexeme 1
                                     (- (string-length lexeme) 1))))))

  ;; keyword-filter : string -> token
  (define (keyword-filter str)
    (let ([maybe-kwd (assoc str keyword-list)])
      (if maybe-kwd
          ((cadr maybe-kwd))
          (token-ID (string->symbol str)))))

  (define keyword-list
    `(("format" ,token-FORMAT)
      ("tamrof" ,token-TAMROF)
      ("type" ,token-TYPE)
      ("epyt" ,token-EPYT)
      ("if" ,token-IF)
      ("fi" ,token-FI)
      ("each" ,token-EACH)
      ("chae" ,token-CHAE)
      ("String" ,token-STRING)
      ("Integer" ,token-INTEGER)
      ("Float" ,token-FLOAT)
      ("NumBool" ,token-NUMBOOL)))

  (define vars (make-hash-table))

  (define (get-variable-type id)
    (let ((sval (hash-table-get vars id #f)))
      (and sval (car sval))))

  (define (get-variable-value id)
    (let ((sval (hash-table-get vars id #f)))
      (and sval (cdr sval))))

  (define (set-variable-type! id typ)
    (hash-table-put! vars id (cons typ (get-variable-value id)))
    id)

  (define (set-variable-value! id val)
    (hash-table-put! vars id (cons (get-variable-type id) val))
    id)

  (define bellows-parser
    (parser (start program)
            (debug "debug-bellows")
            (tokens regular keywords)
            (grammar (program ((formatsection maybe-typesection) $1)
                              ((typesection formatsection) $2))
                     (maybe-typesection ((typesection) null) (() null))
                     (typesection ((TYPE EOL typespecs EPYT EOL)
                                   null))
                     (typespecs ((typespec typespecs) null)
                                (() null))
                     (typespec ((ID type EOL) (set-variable-type! $1 $2)))
                     (type ((STRING) 'string) ((INTEGER) 'integer)
                           ((FLOAT) 'float) ((NUMBOOL) 'numbool))
                     (formatsection ((FORMAT root EOL formatspecs TAMROF EOL)
                                     (list $2 $4)))
                     (root ((ID) $1))
                     (formatspecs ((formatspec formatspecs) (cons $1 $2))
                                  (() null))
                     (formatspec ((dataline) $1) ((block) $1))
                     (dataline ((defids-or-literals EOL) $1))
                     (defids-or-literals
                       ((defid-or-literal defids-or-literals) (cons $1 $2))
                       (() null))
                     (defid-or-literal
                       ((ID) (set-variable-value! $1 'NO-VALUE-YET))
                       ((literal) $1))
                     (block ((ifblock) $1) ((eachblock) $1))
                     (ifblock ((IF test EOL formatspecs FI EOL)
                               (cons 'if (cons $2 $4))))
                     (test ((ID relop id-or-literal) (list $2 $1 $3))
                           ((literal relop ID) (list $2 $1 $3)))
                     (id-or-literal ((ID) $1) ((literal) $1))
                     (literal ((STRLIT) $1)
                              ((NATLIT) (string->number $1))
                              ((NUMLIT) (string->number $1)))
                     (relop ((GT) '>) ((GTE) '>=) ((LT) '<) ((LTE) '<=)
                            ((EQ) '=) ((NEQ) '(lambda (x y) (not (= x y)))))
                     (eachblock ((EACH recordname qualifiers EOL formatspecs
                                       CHAE EOL)
                                 `(each ,$2 ,$3 ,$5)))
                     (recordname ((ID) $1))
                     (qualifiers ((qualifier qualifiers) (cons $1 $2))
                                 (() null))
                     (qualifier ((qualifier-nat) $1)
                                ((qualifier-str) $1)
                                ((qualifier-id) $1))
                     (qualifier-nat ((ID BIND NATLIT)
                                     (check-qualifier-nat $1 $3)))
                     (qualifier-str ((ID BIND STRLIT)
                                     (check-qualifier-str $1 $3)))
                     (qualifier-id ((ID BIND ID)
                                     (check-qualifier-id $1 $3))))
            (end EOF)
            (error (lambda (a b c)
                     (error 'bellows-parser
                            "error occurred, ~v ~v ~v" a b c)))))

  (define (check-qualifier-nat maybe-key rhs)
    (if (= 'num maybe-key)
        (list maybe-key rhs)
      (error 'bellows-check-qualifier-nat
             "expecting keyword 'num' in qualifer; found ~v" maybe-key)))

  (define (check-qualifier-str maybe-key rhs)
    (if (member maybe-key '(sep end))
        (list maybe-key rhs)
      (error 'bellows-check-qualifier-str
             "expecting keywords 'sep' or 'end' in qualifer; found ~v"
             maybe-key)))
  (define (check-qualifier-id maybe-key id)
    (cond
      ((not (hash-table-get vars id #f))
       (error 'bellows-check-qualifier-id
              "variable '~v' in this qualifier is not defined previously."
              id))
      ((not (member maybe-key '(num sep end)))
       (error
         'bellows-check-qualifier-id
         "keyword '~v' in this qualifier must be one of 'num', 'sep' or 'end'."
         maybe-key))
      (else (list maybe-key id))))

  (print-struct #t)
  (define (make-lexer str)
    (let ([ip (open-input-string str)])
      (lambda ()
        (let ((token (bellows-lexer ip)))
          (display "token = ")
          (display token)
          (newline)
          token))))
)
```

4. bellows-reader.scm

```
(require "bellows-parser.ss"
         (lib "serialize.ss")
         (lib "string.ss"))

(define (display+ . stuff) (for-each display stuff))
(define (getline-list)
  (regex-split "[\t\r\n ]+" (getline)))

(let ((putback-line #f))
  (define (getline)
    (if putback-line
        (let ((res putback-line))
          (set! putback-line #f)
          res)
        (read-line)))
  (define (putline pline)
    (set! putback-line pline))
  putback-line)
```

```
;;;;serials;;;;

(define ast (deserialize serial-ast))
(set! vars (deserialize serial-vars))

(define root-name (car ast))
(define format-list (cadr ast))

(define source-file-name "src1.txt")

(define (display-element tag-name contents)
  (display+ "<" tag-name ">" contents "</" tag-name ">\n"))

(display+ "<" root-name ">\n")

(with-input-from-file source-file-name
  (let loop ((fmt format-list))
    (cond
     ((empty? fmt) null)
     ((equal? 'each (car fmt))
      (let* ((tag-name (nth 1 fmt))
             (quals (nth 2 fmt))
             (efmt (nth 3 fmt))
             (num (lookup 'num quals))
             (sep (lookup 'sep quals))
             (end (lookup 'end quals)))
        (cond
         ((and num end)
          (error 'found-mutually-exclusive-each-qualifiers
                 "The each construct cannot have both num and end qualifier."))
         (num
          (let ((num
                 (cond ((symbol? num)
                        (let ((nval (get-variable-value num)))
                          (or nval
                              (error
                               'num-qualifier-RHS-DNE
                               "RHS of num qualifier has no value.")))))))
            (if (not (and (integer? num) (> num 0)))
                (error 'num-qualifier-should-be-a-natural
                       "RHS of num qualifier needs to be a natural."))
            (let num-loop ((num num))
              (cond ((> num 0)
                     (loop efmt)
                     (num-loop (- num 1)))))))
         (end
          (let ((end
                 (cond ((symbol? end)
                        (let ((end-val (get-variable-value end)))
                          (or end-val
                              (error
                               'end-qualifier-RHS-DNE
                               "RHS of end qualifier has no value.")))))))
            (if (not (string? end))
                (error 'end-qualifier-should-be-a-string
                       "RHS of num qualifier needs to be a string."))
            (let end-loop ()
              (let ((iline (getline)))
                (cond ((string= end iline) null)
                      (else (putline iline)
                            (loop efmt)))))))
         (else
          (error 'malformed-each-qualifier
                 "An each needs either a num qualifier or an end qualifier.")))))
     ((equal? 'if (car fmt))
      (let ((condition (cadr fmt))
            (ifmt (cdr fmt)))
        (if condition (loop ifmt))))
     (else (check-format-line (car fmt) in-line)
           (if (empty? (cdr fmt)) null
               (loop (cdr fmt) (getline)))))))

(define (lookup key alist)
  (let ((apair (assoc key alist)))
    (and apair (cdr apair))))

(define (check-format-line fline)
  (for-each
   (lambda (fitem iitem)
     (cond ((string? fitem)
            (if (string= fitem iitem)
                (display-element "literal" iitem)
                (error 'literal-no-match
                       "Literal \"~v\" doesn't match format."
                       fitem)))
           ((symbol? fitem)
            (set-variable-value! fitem iitem)
            (display-element fitem iitem))
           (else (error 'impossible-condition
                        "Parser didn't yield a symbol here"))))
   fline (getline-list)))
```

```
(display+ "</" root-name ">\n")
```