BELLOWS LANGUAGE REFERENCE MANUAL

RICK HANSON

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 a multi-line (vertical) list of multi-line records, usually preceded by the number of such records. There are also conditional elements: those records or data items which are present in the object file under a condition described in the file format. This condition holds according to the value of some previous data item in the object file.

1.1. **An Example.** The following is an example Bellows program, paired with an example object file.

Bellows Source File	Object File

`mytimefile::Name`
`'Version 42'`
`title::String`
`#item::Integer`
`each item`
`  begintime::String endtime::String moreflag::NumBool`
`  if moreflag == 1`
`  more1::Float more2::Float more3::Float more4::Float`
`  fi`
`chae`
`lastupdatedby::String`
	`Version 42`
	`Table_of_items`
	`3`
	`6:00 9:00 0`
	`2:03 12:45 1`
	`0.000 42.000 1.000 2.000`
	`9:33 10:12 0`
	`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 `#item`.

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

Date: Fri 28 Sep 2007.
For CS4115 PLT, Fall 2007.

`each` keyword, in this case `item`, is the name of the record; the number of such records can be obtained by inspecting the value of the variable which has the name of the record prepended by the # character, in this case `#item` which has the value of `3`.

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>
  <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>
```

1.2. **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[1] 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

[1]We might want to define here what *legal match* means—just arm-waving for now.

(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.

2. Lexical Conventions

2.1. **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.

```
EOL : (('\r')? '\n' {newline();})+ ;
```

2.2. **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.

```
WS : (' ' | '\t')+ {$setType(Token.SKIP);} ;
```

2.3. **Intrinsic Operators.** Bellows has binary predicate operators which are designed to be used in the test portion of the conditional construct. Bellows also has a binary type declarator `::`, which stands for "has type". For instance, a usage such as `varname::Integer` means that `varname` has type `Integer`.

```
GT  : '>' ;
GTE : ">=" ;
LT  : '<' ;
LTE : "<=" ;
EQ  :  "==" ;
NEQ : "!=" ;
HAS_TYPE : "::" ;
```

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

```
IF      : "if" ;
FI      : "fi" ;
EACH    : "each" ;
CHAE    : "chae" ;
NAME    : "Name" ;
STRING  : "String" ;
INTEGER : "Integer";
FLOAT   : "Float" ;
NUMBOOL : "NumBool" ;
```

2.5. **Identifiers.** An identifier is a variable name which, in the Bellows source, is denoted by a string of alphanumeric characters (starting with a letter), and optionally preceded by the # character.

```
protected HASH : '#' ;
protected LETTER : ( 'a'..'z' | 'A'..'Z' ) ;
protected DIGIT  : '0'..'9' ;
ID
options { testLiterals=true; }
    : (HASH)? LETTER (LETTER | DIGIT)* ;
```

2.6. **Literals.** String literals are denoted by a sequence of characters between a pair of single quotes, e.g. `'Version 42'`. Numeric literals are a sequence of number characters (i.e. `0`, `1`, `2`, ..., `9`), followed by an optional sequence of number characters preceded by a period.

```
NUMBER_LITERAL : (DIGIT)+ ('.' (DIGIT)+)? ;
STRING_LITERAL : '\'' (~('\r' | '\n' | '\''))* '\'' ;
EOL : (('\r')? '\n' {newline();})+ ;
```

3. Grammar and Semantics

A Bellows program starts with a name line, which names the object file and has the format *objectfilename*`::Name`, followed by the object file's format specifications.

```
program   : name_line (format_spec)+ ;
name_line : ID HAS_TYPE NAME EOL ;
```

A format specification is either a simple data line, which contains a space-separated list of identifiers with optional type specifiers, or a block of such data lines.

```
format_spec : data_line | block ;
data_line   : (ID (HAS_TYPE typespec)?)+ EOL ;
typespec    : STRING | INTEGER | FLOAT | NUMBOOL ;
```

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

```
block : if_block | each_block ;
```

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 specifiers. There is a conditional test written after the keyword `if` which when it holds, the format specifiers in the `if` block must match the object file data; otherwise the `if` block format specifiers do not apply to the object file data.

```
if_block : IF test EOL (format_spec)+ FI EOL;
```

Conditional tests look like the infix relational operation xRy, where R is any one of the relational operators >, >=, <, <=, ==, !=, which stand for "greater than", "greater than or equal to", "less

than", "less than or equal to", "equal to" and "not equal to" respectively. 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.

```
test    : (ID relop (ID | literal))
        | (literal relop ID) ;
relop   : GT | GTE | LT | LTE | EQ | NEQ ;
literal : STRING_LITERAL | NUMBER_LITERAL ;
```

The each block is delimited by lines which start with the keywords each and chae, respectively. The each block contains one or more format specifiers which define a record, and we call this the record specification. There is a record name, i.e. an identifier, after the keyword each. The record specification will match a certain number of data records in the object file; hence, the each block is a looping construct. The number of such records can be obtained by inspecting the value of the variable which has the name of the record prepended by the # character.

```
each_block  : EACH record_name EOL (format_spec)+ CHAE EOL ;
record_name : ID ;
```