

# RegReg 1.1

A parser generator based on cascades of tagged regular expressions

Mario Latendresse, June 2003, Last revision October 2006



# Table of Contents

<b>1</b>	<b>Overview .....</b>	<b>1</b>
1.1	An example of a two level parser using TREs .....	1
<b>2</b>	<b>Describing a parser: syntax and semantics ...</b>	<b>5</b>
2.1	General structure .....	5
2.2	Declaring the name of the parser .....	6
2.3	Macros .....	6
2.4	Tagged Regular Expressions (TREs) .....	6
2.5	The parse tree structure .....	9
2.6	The special <error> token .....	10
2.7	Parsing ambiguities .....	11
2.8	The shortest matching rule vs the longest matching rule .....	11
2.8.1	Examples of the short option .....	11
2.8.2	Contradictory specifications using <code>short</code> .....	12
2.9	The <code>prefer</code> option .....	12
2.10	The <code>discard</code> option .....	13
2.11	Empty string generated twice at level 1 .....	14
<b>3</b>	<b>Generating a parser from a description .....</b>	<b>16</b>
3.1	Running the compiled RegReg .....	16
3.2	Running RegReg under a Scheme interpreter .....	16
3.3	Instantiating a parser from a DFAs vector .....	16
<b>4</b>	<b>Processing the tagged parse tree .....</b>	<b>18</b>
4.1	The pre-defined parse tree extraction functions .....	18
4.2	Examples of tagged parse tree manipulations .....	19
<b>5</b>	<b>Debugging a parser .....</b>	<b>20</b>
<b>6</b>	<b>Elaborate RegReg examples: decoding meteorological bulletins .....</b>	<b>22</b>
6.1	Decoding Terminal Aeorodrome Forecast (TAF) bulletins .....	22
6.2	Decoding Pilot REPorts (PIREPs) .....	25
6.3	Decoding AMDARs and ACARs .....	30
<b>7</b>	<b>Acknowledgements .....</b>	<b>33</b>

# 1 Overview

RegReg is a lexer/parser—henceforth simply called a parser—generator based on the Scheme programming language. The description of the parser is a cascade of lexers, each lexer being described by tagged regular expressions (TREs). A TRE is a regular expression with bindings allowing easy and robust extraction of substrings. We call them *tagged* regular expressions, since tags may identify sub-expressions. RegReg is particularly well suited to create parsers where lexical analysis is complex and irregular.

The particularity to tag a part of a regular expression is an essential feature of RegReg. It sets it apart from other tools or languages such as Lex, SILEx, Perl, Awk, etc. After parsing, these tags appear in the parse tree, facilitating extraction of substrings from the matched string.

RegReg generates deterministic finite automata (DFA) as one vector. This vector is combined with a driver (contained in file `'regregDrvr.scm'`), and probably some other Scheme user functions, to form a complete Scheme program. When this program is run, let say under a Scheme interpreter, it scans the input and creates a tagged parse tree as a result—similar but simpler than an XML document. The user specifies a Scheme function to receive this parse tree that can easily be manipulated to extract substrings.

The tool RegReg is written in the Scheme programming language. It also generates Scheme code. You should program the semantics analysis in Scheme.

The parser is described solely by TREs which are essentially regular expressions. It is therefore not as general as a LALR(1) parser generator, such as Yacc or Bison. On the other hand, it is well suited to generate parsers for which the lexical analysis is complex and irregular.

In this document, we assume some familiarity with Scheme. In the next section we present a complete example of a parser described by a cascade of lexers.

## 1.1 An example of a two level parser using TREs

Here is an example of a complete RegReg description. This parser breaks a text into lines formed by words, some special characters, and numbers.

```

(define (process-text r)
  (display "The resulting parse tree is: ")
  (write r)
  r)

(RegReg

  (declare (name example1))

  (macros
    (blank "[ \\010\\013]"))

  (level 1
    (w "[A-Za-z]*{l=[A-Za-z]}")
    (n "{f=[1-9]}[0-9]*")
    (s "[-,=]")
    (e "[.;:!]")
    (b "{blank}+"))

  (level 2
    (line ((+ (? b) (: s w n)) e)))

  (level 3
    (text (* line) process-text)))

```

Note: this example is available in the file ‘`example1.reg`’ in directory `doc`.

First, let us focus on one particular feature which makes RegReg more valuable than other lexer generators: the binding operator `=`. This is one of the main difference from regular expressions commonly used in tools such as Lex, Flex, Perl, Awk, Emacs, etc.

The binding operator `=` associates a symbol, that is a tag, with a tagged regular expression (TRE). In the resulting parse tree, this tag is associated with a substring—which is represented as a subtree in general—that matches this TRE. For example, in the definition of token `w` in level 1, the tag `l` identifies the last letters of every word. In the definition of token `n`, the tag `f` is bound to the first digit of every number. They are explicit tags. The tokens `w`, `n`, `s`, `e`, `b`, `line`, and `text` define implicit tags, so that they identify the corresponding tokens. (Note: In the resulting tagged parse tree, there are no differences between explicit and implicit tags. They are all Scheme symbols associated to a subtree.)

A parser description always begin with the symbol `RegReg`.

The first sub-list declares the name of the parser, namely `example1`. It names the vector representing the complete parser.

The second sub-list defines the macros used at level 1. Each macro is defined by specifying an identifier (a Scheme symbol) and a tagged regular expression (TRE). The macro section may contain several definitions that may only be referenced by level 1.

In this example, there are three levels. A TRE may be described by a mix of s-expressions and strings. The level 1 defines TREs based on characters; whereas levels 2 and 3 contain TREs based on tokens. Level 2 may only refer to tokens defined in level 1; similarly, level 3

may only refer to tokens defined in level 2. In general, TREs in level  $n > 1$  may only refer to tokens defined in level  $n-1$ . TREs in level 1 may refer to macros.

A level contains a series of rules. A rule is identified by a Scheme symbol, or tag, a TRE and possibly some optional parameters. Possible optional parameters are: (**short**), and the name of user defined function to call when the rule has been selected due to a match. The order of those rules is important: in a case of multiple matchings, an early rule has precedence over a late one.

The general structure of the parser is at level 3. It defines only one token, namely **text**. It is formed by a series of **line**. The last level, in this case 3, drives the whole parsing process. The parser is described by the last level, and it relies on the previous level for its tokens; similarly, the next to last level relies on its previous level, etc.

Level 2 gives the structure of a **line**. A line is a series of words, numbers or special characters, probably separated by **blanks**, and terminated by an end mark.

In this example, level 2 and 3 use only s-expressions to describe the TREs. Level 1 uses only the string form. Actually any level can specify TREs as strings, s-expressions or a mix of both. This two different syntactic forms are provided for convenience. (Note: in level 2 and up it is not possible to refer to individual characters. This is only possible at level 1.)

For this particular example, the function **process-text** is called when all lines have been read. Such a function may appear for each token in any level (but not in the macros section). These users functions may be defined in the parser description file before the **RegReg** s-expression like this example. If no user defined function is provided, the resulting parse tree is returned.

Parsing the string "There are 6 words in this sentence. A second line." result in the tagged parse tree: (this tree is passed to **process-text** which is specified in the description; it would be the resulting parse tree if no function is called)

```
(text
  ((line
    (((()      (w ("Ther") (l "e"))))
      ((b " ") (w ("ar") (l "e"))))
      ((b " ") (n ((f "6") ()))
      ((b " ") (w ("word") (l "s"))))
      ((b " ") (w ("i") (l "n"))))
      ((b " ") (w ("thi") (l "s"))))
      ((b " ") (w ("sentenc") (l "e")))) (e "."))
    (line
      (((b " ") (w (() (l "A"))))
        ((b " ") (w ("secon") (l "d"))))
        ((b " ") (w ("lin") (l "e"))))
        (e ".")))))
```

The tag **text** is associated to two lines as described by the list of two elements next to the tag itself. The lines are formed by a list of several tokens. Each word is tagged by **w**, the number 6 by **n**, the blanks by **b** and the special characters . by **e**. The tags **l** always identify the first letters and the tag **f** the first digit. The first empty list **()** is due to the

absence of a blank token at the beginning of the first line. The empty list for the first word in the second line is due to the short word 'A'; it only has a last letter. Similarly, the number has only a first digit.

The function `process-text` is called with this parse tree as an argument. It is then up to the user defined function to process the parse tree appropriately.

The user function can do whatever it pleases. However, if this function is used at an intermediate level, it should return a list of two elements: a tag (a Scheme symbol) and the value of that tag. (Note: The identity function does just that.) This is necessary since the parser depends on this value to continue processing. Otherwise the parser may fail aborting parsing. If the user function is used at the last level, the returned value can be anything.

Some general functions are provided to easily extract the subtrees and substrings of the tagged parse tree. Consult the section 'Processing the tagged parse tree'.

## 2 Describing a parser: syntax and semantics

### 2.1 General structure

The general structure of a file describing a RegReg parser is a series of Scheme expressions (may be empty), an s-expression whose head is the symbol `RegReg`, and more Scheme expressions (may be empty). This should be contained in a file with extension `‘.reg’`.

The description of a RegReg parser is a Scheme s-expression made of multiple sub-expressions. It must begin with the symbol `RegReg` and the declaration of the name of the parser with `(declare (name s))` where `s` is a Scheme symbol.

```
(define (proc-first r)
  (display "proc-first received tagged parse tree: ")
  (write r) (newline)
  r)

(define (proc-file r)
  (display "proc-file received tagged parse tree: ")
  (write r) (newline)
  r)

(RegReg
 (declare (name e1))
 (macros
  (integer "[0-9]+")
  (word "[A-Za-z]+"))
 (level 1
  (firstline "BEGIN\\010" (short) proc-first)
  (end "END" (short))
  (line "({integer}|{word}|[ ])+\\010"))
 (level 2
  (file "firstline line+ end" proc-file)))
```

Note: this description is in file `‘e1.reg’`, in directory `‘doc’`. There are spaces before `integer` and `word` tokens in the definition of token `line`; these are significant spaces to be matched during parsing.

The main part is formed by the macros section, if any, and the level sections (aka rules sections). At least one level section must be present. Each level section is an s-expression identifying its level number and its series of rules. The level sections must be numbered from 1 to `n`, where `n` is the number of such sections. They can be specified in any order which has no significance.

The last level defines the whole parser; in a sense it drives the whole parsing process. It relies on the previous level for its tokens; and this previous level relies on its previous level, etc. In this example, level 2 drives the parsing process by using level 1.



The file containing the `RegReg` s-expression may contain other s-expressions before and after the `RegReg` s-expression. These s-expressions are reproduced verbatim in the generated file of the parser before and after the vector that describes it.

The s-expressions appearing before the `RegReg` one can be the user defined functions specified in the parser description. The names `proc-first` and `proc-file` are user defined functions. Therefore, these may appear before the `RegReg` description. If the generated parser file is loaded in a Scheme interpreter, the vector defining the parser would correctly refer to those functions.

The s-expressions after the `RegReg` one can be expressions to execute when the parser is loaded in a Scheme interpreter. The test files in directory `test` use this approach to verify the generated parsers.

## 2.2 Declaring the name of the parser

Any `RegReg` parser description must have a declaration of its name. This is independent of the name of the file containing the description.

It has the following syntax: `(declare (name id))` where the identifier *id* must be a valid Scheme symbol.

The declaration may appear anywhere in the `RegReg` s-expression. It is used by the `regreg` generator to name the vector containing the DFAs. The resulting name is *id-dfas*. It is essential to know this name as you must use it to instantiate your parser.

It is your responsibility to choose an *id* such as to avoid any name clashes in your resulting Scheme code.

## 2.3 Macros

The `macros` section is a series of definition of macros. It is optional. A macro definition is a list formed by an identifier, that is its name, and a TRE. A macro may be referred, using braces `{}` around its name, by other TREs defined only in the macros section and level 1. They cannot be used in level 2 and up, since macros describe strings of characters, not tokens. The purpose of macros is to reduce the length of TREs defined at level 1.

A macro must be defined before it is referenced—recursive definitions are not permitted. They must also be unique, no two macros may be named the same.

The name of a macro must be a valid Scheme symbol using only letters, digits, dash `'-'` and underscore `'_'`.

## 2.4 Tagged Regular Expressions (TREs)

A tagged regular expression (TRE) is specified as a string, an s-expression, or a mix of both. If it is specified using an s-expression, only the leaf can be strings. This dual syntax helps for readability and brevity.

A TRE is essentially a regular expression; a new binding operator `=` is available to identify parts of the regular expression; which results in a tagged regular expressions.

In TREs specified in level 2 and up, it is not possible to directly refer to characters. This is only allowed in level 1 and macros. In level 2 and up, TREs are built by referring to tokens from the previous level.

Here is an informal description of the syntax and semantics of TREs.

1. In a string, a single character matches itself to the exception of the special characters used as operators and described in this section. The special characters are: `. () {} [] ? + * \`. To specify a special character as itself you must escape it using a backslash `\`. The binding operator `=` is not special in general, except right after an identifier between braces. So, the string `"A"` matches `'A'`. In most cases a series of characters, as in `"ABC"`, are specified: this is actually using the concatenation operation on individual characters. Characters can also be specified, in s-expressions, using the Scheme syntax `#\` or other forms available in the Scheme implementation used.
2. Concatenation is specified either by juxtaposition in a string or as a list in an s-expression. For example, `"ABC"` matches `"ABC"`; `("A" "B" "C")` and `(#\A #\B "C")` matches same. The empty list `()` or empty string `""` represent a TRE matching the empty string. For brevity's sake, operators `+`, `?`, `=`, and `*` provide an implicit concatenation, in s-expressions, when more than one sub-TREs are specified. See their description below.
3. In level 2 and up, the basic element is the token. For example, the TRE `(blank word blank)` describes a TRE matching tokens `blank`, `word`, and `blank`, in that order. The string `"blank word blank"` specifies the same TRE. After level 1, all TREs are based on tokens; so a series of characters at these level do not refer to characters but token names.
4. The backslash `\` is an escape character and can be used to represent an operator as itself; for example `"\\?\\+"` matches the string `"?+"`. (Note: in a Scheme string, the backslash `\` is already an escape character. It must be doubled to specify one backslash.) In particular `"\\."` matches a dot, not any character. To specify a backslash, two should be specified, which means that four must be written in a Scheme string; hence, the TRE `"\\\\"` matches a single backslash. The backslash must be used to specify non-printable characters: it is followed by at most three digits specifying, in decimal, the 8 bits ASCII code of that character. For example, `"\\013"` describes the return character. (Note: it is assumed that your Scheme implementation uses 8 bits ASCII code for the primitive `integer->char` and `char->integer`. This is not part of the R5RS standard.). A non-special character that follows a backslash simply stands for itself. You can therefore escaped any character even though it does need to. The backslash notation is strictly for level 1 and the macros section.
5. Square brackets `[]` describe a set of characters; it matches one character of that set. Inside the square parentheses a series of single characters or range of characters can be specified. The only special characters are `-`, `^`, and `]`. Other characters (e.g. `.`, `?`, etc.) are not special. They can be escaped with a backslash, but that is not necessary. A range of characters is specified by using a dash; for example, `"[A-Za-z]"` is the set of all letters. The complement of a set is specified by `^` as the first character; for example, `"[^0-9]"` is the set of non-digit characters; `"Begin[ A-Z]+End"` matches the string `"Begin Hello End"`. The backslash can still be used to specify non-printable characters. For example, `"[01\\010\\\\"]` matches either `"0"`, `"1"`, a newline, or a backslash. Range and set of characters can be specified in s-expressions using the operators `::` (in set), `!::` (not in set), `..` (in range), and `!..` (not in range). For example `(.. #\a #\z)` is equivalent to `"[a-z]"`; `(!.. #\0 #\9)` is equivalent to `"[^0-`

- 9"]; (`:: #\a #\e #\i #\o #\u`) is equivalent to `"[aeiou]"`. The brackets notation is strictly for level 1 and the macros section.
6. The dot character `.` is equivalent to `"[^\\010]"`; that is any character except a newline. For example, `"..."` matches any string of length three without a newline; `".*"` matches any string without a newline. If a dot appears between square brackets, it specifies itself, not any character; so the TRE `"[.]"` is equivalent to `"\\."` which matches only a single dot. The dot notation is strictly for level 1 and the macros section.
  7. In a string specification, pair of parentheses `()` modify operators precedence and form only one TRE. They allow TRE sub-expressions grouping. (Note: do not confuse parentheses being used in s-expressions and the parentheses to group TRE sub-expressions in a string. Precedence is always explicitly stated in s-expressions. )
  8. Curly braces `{}`, specified in a string, refer to a macro or specify a binding. The form `"{s}"` refers to the macro `s`. The form `"{s=e}"` specifies a binding between the tag (aka symbol) `s` and the matching string for TRE `e`. For example, `"{blank}"` refers to a macro named `blank` (defined in the macros section); `"{i=[0-9]+}{f=[0-9]+}"` matches some decimal numbers and binds the integral part to `i` and fractional part to `f`. The braces notation can be used in any level. Note that tags do not have to be unique – it is up to the semantics analysis to make sense of non-unique tags.
  9. Operator `=` in an s-expression specifies a binding. It has the same function as used in curly braces. The first argument is an identifier. The second argument is a TRE; or there is a list of TREs, in which case an implicit concatenation is assumed between them to form only one TRE. For example, the TRE `(= f "[0-9]")` binds `f` to the substring matching `"[0-9]"`; the TRE `(= r first second third)` binds `r` to the subtree matching `(first second third)` where `first`, `second` and `third` are assumed to be macros or tokens.
  10. Operator `?` applied to a TRE matches the empty string or that TRE; for example `"CLE?A?R"` matches the strings `"CLR"`, `"CLER"`, `"CLAR"`, and `"CLEAR"`. It can be applied to a more complex TRE as in `"(KT)?"` which matches `"KT"` or the empty string. It can be specified using an s-expression as in `(? "KT")`. For an s-expression, the number of arguments can be greater than one; in such a case an implicit concatenation is applied to the list of arguments. For example, `(? one two three)` is equivalent to `(? (one two three))`. This operator can be used in any level.
  11. Operator `+` applied to a TRE matches one or several times that TRE. For example, `"BA+B"` matches `"BAAAB"` but not `"BB"`. It can be applied as an s-expression as in `(+ "OH!")` which, for instance, matches `"OH!OH!OH!"`. If more than one sub-TREs are specified, an implicit concatenation is assumed. For example, `(+ (? "A") "B")` is equivalent to `(+ ((? "A") "B"))` which matches `"AB"`, `"BBBB"`, `"ABABB"`, etc. This operator can be used in any level.
  12. Operator `*` applied to a TRE matches zero or several times that TRE. For example, `"A*"` matches the empty string, `"A"`, `"AA"`, etc. It can be specified using an s-expression as in `(* "A")` which is equivalent to `"A*"`. If more than one sub-TREs are specified, an implicit concatenation is assumed. For example, `(* (: "A" "B") "C")` is equivalent to `(* ((: "A" "B") "C"))` and `"((A|B)C)*"`. This operator can be used in any level.
  13. The operator `|`, specified in a string, between TREs, matches if one of its TRE matches (this is also known as the 'or' operator). It has a lower precedence than concatenation.

tion. Therefore, the TRE `"abc|def|hfg"` matches `"abc"`, `"def"` or `"hfg"`; nothing else. It can be specified as an s-expression by using `:` as in `(: "abc" "def" "hfg")`. (Note: of course, in the case of `:`, more than one sub-TREs does not imply an implicit concatenation.). This operator can be used in any level.

Note: specified as a string, each operator `?`, `*` and `+` applies to the immediate preceding TRE; for example `"a|b+"` matches the string `"bbb"` but not `"ababa"`. Parentheses change this behavior as in `"(a|b)+"` which matches, for example, `"ababa"`.

Note that tagged regular expressions can be entirely specified as a string or as an s-expression, or a combination of both.

For example the TRE `"{i=[0-9]+}\.\{f=[0-9]*\}[]*((E|e){e=[-+]?[0-9]+})?"` is equivalent to

```
((= i "[0-9]+" ) #\.  
(= f "[0-9]*") "[ ]*"   
(? (: "E" "e") (= e "[-+]?[0-9]+")))
```

The dot character `'.'` is specified as a character, i.e. `#\.`, whereas the exponent characters `"E"` and `"e"` are specified as strings. There is no difference, although the dot when specified as a string must be escaped as in `"\."`. You can use file `'test4.reg'` in directory `'test'` to try out this example.

## 2.5 The parse tree structure

This section presents further details on the structure of the parse tree constructed by RegReg when a string matches a regular expression.

Essentially, the structure of the parse tree corresponds to the structure of the regular expression. Also, all defined tags using the equal sign operator (e.g., `{l=[0-9]+}`) that appears in the regular expression matched appears also in the parse tree. These tags may repeat in the parsed tree even though they do not repeat in the regular expression. These repetitions are due to the operators `+` and `*`. (Note: you may also repeat the tags in the regular expression. This is up to your semantics analysis to make sense of these repetitions in the parse tree.)

In general the parse tree is a list of lists for which the leaves are strings and symbols. The strings can only be formed by strings coming from the input. The symbols can only be tags.

At level 1, the generated tree is either a string of length 0 or 1 (e.g. `"a"`), a singleton list of a string of length greater than 1 (e.g. `("xyz")`), a list of two elements the first one being a symbol (a tag) and the second one a string of any length (e.g. `(1 "98")`); or a list of these. The first case occurs for a matched regular expression without a tag which matched exactly 0 or 1 character; the second case occurs for a matched regular expression without a tag which matched two or more characters. The third case occurs when the matched regular expression is tagged by exactly one tag. The fourth case occurs when the matched regular expression has two or more tags, or part of the matched regular expression has at least one tag whilst other parts are untagged.

Level 2 constructs a parse tree from trees generated at level 1; similarly, level 3 constructs a parse tree from trees generated at level 2 – and so on. No further new character strings are

generated from level 2 and up. The concatenation and repeat operators `*` and `+` generate lists of trees.

Notice the peculiar result for operator `?` with a tag embedded in a matching regular expression: the empty list will be given if the sub-expression under `?` did not contribute to the match. For example, for the TRE `{a=[A-Z]}{d=[0-9]}?{b=[A-Z]}` matching the string "AB", the resulting parse tree is `((a "A") (d ())) (b "B"))`. The tag `d` is associated to an empty list, i.e. an empty tree, since no digit is in "AB".

## 2.6 The special `<error>` token

Any rule section may define the special token `<error>`. Such a definition specifies a function to call when no rule of the section matches the input. The function has two arguments, the first one being the list of tokens active when the last character or token was read, the second one is the list of tokens or the string of characters read so far.

Let us assume the following RegReg description.

```
(define (out-rules1-error rule-ids s)
  ;; Caught a parsing error; fake a time token.
  '(time ,s))

(define (out-rules2-error rule-ids r)
  (display "Error: ")
  (display (list rule-ids r)) (newline)
  r)

(define (out-line r)
  (display "Tagged parse tree is: ")(write r) (newline)
  r))

(RegReg
 (declare (name out))
 (level 1
  (time      "{h=[0-9] [0-9]}{m=[0-9] [0-9]}Z")
  (blank     "[ ]+")
  (dot       "\\.")
  (<error> out-rules1-error))
 (level 2
  (line      ((+ time blank) dot) out-line)
  (<error> out-rules2-error)))
```

Note: this description is available in file `'out.reg'` in directory `'doc'`.

The special token `<error>` in level 1 specifies the user defined function `out-rules1-error`. It will be called by the RegReg driver when no basic token can be recognized. Similarly, in level 2, the function `out-rules2-error` must be called when no rule is recognized at that level. The function `out-line` is called when parsing is successful.

Note that `out-rules1-error` receives a string for argument `s`; whereas `out-rules2-error` receives a list of tokens for `r`. The function `out-rules1-error` returns a list with the tag `time` as head. In other words it fakes the recognition of a `time` token. An error function, from an intermediate level, may return whatever it sees fit to correct the parsing, but it should return a list for which the head is a symbol identifying a rule. Such a function returns to the parser; whereas an error function for the last level does not.

If the string "1020 2034Z" is parsed, an error occurs at level 1 as there is no 'Z' following '1020'. The current active list of rules is `(time)` and the string of characters is "1020 ". Therefore, the function `out-rules1-error` receives these two values. The error function may analyze the list of active rules and the string to take corrective action. In this example, it simply fakes the token `time` to continue processing. This is acceptable for level 2. The next call to level 1 returns a correctly parsed `time` token. But, this fails at level 2, since a blank is expected. The function `out-rules2-error` receives the list `((time "1020 ") (time ((h ("20")) (m ("34")) "Z"))))` and the rule-ids is `(line)`. This error function simply displays an error message.

If the string "1020Z 2034Z" is parsed, an error occurs at level 2 as there is no blank and dot following '2034Z'. In that case, the function `out-rules2-error` receives the list of rule-ids `(line)` for parameter `rule-ids` and the list `((time ((h ("10")) (m ("20")) "Z")) (blank (" ")) (time ((h ("20")) (m ("34")) "Z"))))` for parameter `r`.

## 2.7 Parsing ambiguities

Parsing ambiguities may exist in some TRE specifications. For example, the TRE `"{x=[0-9]}?{y=[0-9]}?"` is ambiguous when parsing a single digit: should it be bound to `x` or `y`? RegReg neither refuses this TRE nor does it warn of its ambiguity. Yet, this ambiguity may not cause any problem. The user code extracting `x` and `y` should take care of this ambiguity, e.g., search for the tags `x` and `y` and act accordingly.

## 2.8 The shortest matching rule vs the longest matching rule

A TRE can be qualified as *short* by specifying `(short)` after the specification of the regular expression – this is a short TRE. By default the driver matches the longest string but a short TRE matches the shortest one, that is as soon as an accepting state is reached.

Although this is not certain, the specification of short TREs can reduce the size of the corresponding DFA. It definitely does not slow down the parser. Essentially, the `(short)` option forces the driver to stop and accept the string as soon as a DFA final state (aka accepting state) is reached. The implementation of that feature is simple: any out transition of a final DFA node corresponding to a regular expression qualified as short is eliminated.

If two regular expressions qualified as short match a string, the first declared one is the preferred one.

### 2.8.1 Examples of the short option

Assume the following parser description

```
(define (store-text r)
  (display "Received the tagged parse tree: ")
  (write r)
  (newline)
  r)

(RegReg
  (declare (name exampleShort))
  (level 1
    (begin "BEGIN" (short))
    (end "END" (short))
    (any "[a-zA-Z0-9_]+" )
    (blank "[ \\010\\013]+" ))
  (level 2
    (text (begin (? blank) (+ any blank) end) store-text)))
```

Note: this description is available in file ‘exampleShort.reg’ in directory ‘doc’.

Two TREs use the short qualification, namely "BEGIN" and "END". They do so to recognize substrings prefixed by 'BEGIN' and 'END' as tokens **begin** and **end**, respectively, and not as token **any**.

For example, parsing "BEGINHELLO ENDEND" does produce the parse tree

```
(text ((begin ("BEGIN"))
      ()
      (((any ("HELLO")) (blank (" "))))
      (end ("END"))))
```

If the two tokens **begin** and **end** were not qualified with (**short**), that string would not have matched; the token **any** would have been the longest match for 'BEGINHELLO' resulting in a parsing error at level 2 where the token **begin** is expected at the beginning.

Note: the string "BEGINHELLOENDEND" would not be recognized as the substring 'HELLOENDEND' would be perceived as a token **any**.

### 2.8.2 Contradictory specifications using short

It is possible that the use of **short** contradicts a TRE specification. For example, if a short option is applied to the TRE "[0-9]+", it no longer matches a series of digits but only one. In that case, the + operator is redundant: the TRE should have been specified as "[0-9]".

The TRE "[0-9]+A" does not have this problem since the ending is not of variable length. In general, a TRE that has a variable length ending should not be qualified as **short**. Either, the TRE is wrongly specified or the longest matching rule should be used without specifying the (**short**) option.

## 2.9 The prefer option

Another way to control the matching mechanism is to use the option **prefer**. It cannot be used with **short**, but can replace it.

Here is a complete example.



```

(define (fId r)
  (display "This is an Id parse tree:")
  (display r)
  (newline)
  r)

(RegReg
 (declare (name examplePrefer))
 (level 1
  (Id "[a-zA-Z][a-zA-Z]*" (prefer) fId)
  (tk "[^ \\010\\013]+")
  (_ "[ \\010\\013]+"))
 (level 2
  (file (* (: tk _ Id))))))

(parse-string "One, 12No two34 ;;; three." examplePrefer-dfas)

```

If this code is compiled by `RegReg` and executed, there will be three output lines, one for each `Id`: ‘`One`’, ‘`two`’, and ‘`three`’. The `prefer` qualifier forces the identification of sequences of letters as `Id`; without this qualifier no `Id` would be found but only `tk`. Note, though, that ‘`12No`’ is still recognized as `tk`, not `Id`.

The `short` qualifier does not work for this case, since it would be a contradictory specification: only the first letter of an identifier would be recognized as an `Id`.

The `prefer` qualifier has the following semantics. When scanning, if at least one accepting state of a `prefer` TRE is reached, all non-`prefer` TREs are disregarded until the match is completed; and all `prefer` TREs which are lower than the highest accepting `prefer` TREs are also disregarded. (Note: TREs are ordered according to the definition order: the first ones are higher.)

The use of `prefer` does not slow down the parser. It may increase or decrease the parser automata.

## 2.10 The discard option

The `discard` option applied to a regular expression forces the driver not to build the parse tree once a match has been found.

For example, the `discard` option for `blank` avoids the construction of parse tree for all recognized sequences of spaces, line feeds, returns and tabs:



```
(RegReg
  (declare (name exampleDiscard))

  (level 1
    (blank "[ \\010\\013\\009]+" (discard))
    (word "[^ \\010\\013\\009]+"))

  (level 2
    (file (+ (: word (? blank))) (discard))))
```

A function can still be specified to process the result of a match for a regular expression qualified with `discard`, but the parse tree passed to that function is empty.

This option is mainly used to increase performance. Indeed, in some cases, the parse tree may be very large, and useless. Consider again the last example: level 2 reads all the words, if the file is very large this will create a large parse tree. But it is probably better to process each word by calling a function at level 1 on the regular expression `word` and avoid building the parse tree at level 2 by specifying `(discard)` for `file`.

The `discard` option may be combined with `short` or `prefer` as in `(prefer discard)`.

## 2.11 Empty string generated twice at level 1

The RegReg driver verifies that no two empty strings are recognized in sequence. If it happens, the entire parser is stopped using the function `error`. (Assumed defined in your Scheme interpreter or by the user.)

This behavior avoids an infinite loop in the parser; otherwise it would loop indefinitely by recognizing the empty string.

For example, consider the following description

```
(RegReg
  (declare (name twice))

  (level 1
    (word "[A-Za-z]+" )
    (b "[ ]+" )
    (empty ()))

  (level 2
    (line (: (+ empty) (+ word (? b))))))
```

Note: this code is available in file `'twice.reg'` in directory `'doc'`.

Note the `empty` token being forced to be recognized more than once at level 2 if no word is recognized. (This is a contrived example to emphasize the mechanism of infinite loop.)

In most cases no problem occurs; for instance "HELLO" is completely parsed, "HELLO 9" is partly parsed into `(line (((word ("HELLO")) (b (" ")))))` stopping at '9'. (Note: in this second case it recognized the `empty` token but it is not included in the final result since it cannot follow a `b`.)

But a major problem occurs with "9 HELLO": two **empty** tokens are recognized, which corresponds to two empty strings. This stops the parser. It happens since the '9' is not part of any token at level 1; the empty string is recognized; and level 2 asks for another token from level 1 due to the + operator. In essence the parser is indefinitely stuck on the character '9'.

## 3 Generating a parser from a description

Once a parser description has been built, the `RegReg` parser generator must translate it into some Scheme code. More precisely, this translation generates DFAs in the form of a single vector. This vector can be used to create a parser using the function `make-parser`.

Let suppose that your parser description is in file `'example1.reg'`. The DFAs vector is generated by feeding `'example1'` to `regreg`.

There are two ways to generate the parser: by running a compiled version of `RegReg`, or in a Scheme interpreter. ( We provide the module file `'regreg-m1.scm'` to compile `'regreg.scm'` under Bigloo.) The compiled `RegReg` is much faster, but there is no other benefit.

It creates a single file `'example1.scm'` which contains a vector named `id-dfas`, where `id` is the name declared in `'example1.reg'`. It can be loaded or copied verbatim into another Scheme file to instantiate a parser; more on this in the following section.

### 3.1 Running the compiled RegReg

If you have compiled `'regreg.scm'` into an executable as given by `'regreg-m1.scm'` under Bigloo; you generate the parser by the command:

```
prompt> regreg example1
```

Do not specify the file extension `'reg'` as it is appended to the given file name. The file `'example1.scm'` will be generated. Therefore, if you describe the parser with `'f.reg'` do not use the file name `'f.scm'` for some other code as this file name is used by `RegReg` to store the compilation of `'f.reg'`.

### 3.2 Running RegReg under a Scheme interpreter

You may use your favorite Scheme interpreter to generate your parser. (`RegReg` has been tested under Gambit v3.0 and Bigloo v2.5b.)

You must load the `'regreg.scm'` code into the interpreter.

You simply call the function `regreg-file->scm` with input and output file names as in

```
Scheme> (regreg-file->scm "example1.reg" "example1.scm" #f)
```

It generates the file `'example1.scm'`.

The finite automata, in some serialized form, are displayed on the current output port if `#t` is specified instead of `#f` as the third argument.

### 3.3 Instantiating a parser from a DFAs vector

The file `'regregDrvr.scm'`, the driver, contains the function `make-parser`. It takes three arguments: a function of zero argument to read characters from the input, a vector generated by `regreg` and a string. It returns a function of zero argument which embodies the complete parser. This function reads the input text, parses it, and returns the parse tree. The string

is used to initialize the internal buffer of the driver. In effect, it is as if the string was the prefix of the input. For example, it could be used as the sole input by providing a read function that returns the eof object.

For example, the following code instantiates a parser from a DFAs vector using the Scheme primitive `read-char` as the function to read characters from a string `s`; and call that function to parse the string `s`. (Note: the function `with-input-from-string` is assumed to exist in your Scheme implementation.)

```
;; I: s, a string
;;    na, a vector of DFAs generated by regreg.
;;
(define (parse-string s na)
  (with-input-from-string s
    (lambda ()
      (let* ((lex (make-parser read-char na "")))
        (lex))))))
```

Note: This function `parse-string` is provided in ‘`regregDrvr.scm`’.

The parsing of the two sentences from the overview chapter can be done by calling `parse-string`:

```
(parse-string "There are 6 words in this sentence. A second line."
              example1-dfas)
```

## 4 Processing the tagged parse tree

The parser description should specify, at the highest level, at least one user defined function to process the resulting tagged parse tree. When a user function is called after recognizing a rule, the resulting parse tree is passed as an argument. The parse tree contains all characters parsed as well as the tags that identified the parts. These tags may be implicit as token symbols, or explicit as specified by the binding = operator.

These functions can process the tagged parse tree as they please, but we highly recommend to use the functions presented in this section which are defined in `'regregDrvr.scm'`.

Extraction of parts of the parsed tree could be done without the tags, but that is not a robust approach. The tags eliminate the needs to keep track of the exact position of a part in a tree. In particular, you should not rely on the exact locations of sub-parts in the parse tree, for example by using `list-ref`, `caddr` etc. You should use the mechanism of tags. In any case, tags should be the major reason to use RegReg.

In a parse tree, each implicit or explicit tag forms, with their value, a list of two elements. This value is the result of a sub-parse. Hereafter, the term *value of a tag* refers to the second element of that list.

### 4.1 The pre-defined parse tree extraction functions

In most cases, tag values are extracted from the resulting parse tree. To that end, some functions are already provided by the RegReg package to extract single tag value, multiple tag values, and flattening a subtree into a string. They are defined in the Scheme file `'regregDrvr.scm'`.

It is highly recommended to use these functions to manipulate the parse tree. Here is a brief description of these functions:

1. The function `gstree` extracts the value of a single tag from a tagged parse tree. It takes two arguments: a symbol—the tag—and the tagged parse tree. It returns the value of the first tag by traversing the tree from the root to the leafs and from left to right; in other words, by searching in-order. In general, the value of a tag is a tagged subtree.
2. The function `gstree-all` returns a list of values formed by the values of all instances of one tag. It takes two arguments, a symbol—the tag—and a tagged parse tree. Once the value of a tag is found, which is a subtree, it does not search in that subtree for any instances of the tag. In other words, all values returned are independent subtrees.
3. The function `tree->string` converts a tree into a string by concatenating all strings of the tree, in-order. In other words, it returns the complete string parsed into that subtree.
4. The function `gstks` is simply the composition of `gstree` and `tree->string`; it returns the string value of a tag. In other words, it is the string parsed and associated to that tag.

The following section presents some examples.

## 4.2 Examples of tagged parse tree manipulations

For the following examples, assume that `r` is bound to the tagged parse tree: (note: this is usually done when the parse tree is passed as an argument to a user defined function specified in a rule.)

```
(text
  ((line
    (((()      (w ("Ther") (l "e"))))
     ((b " " ) (w ("ar")  (l "e"))))
     ((b " " ) (n ((f "6") ()))
     ((b " " ) (w ("word") (l "s"))))
     ((b " " ) (w ("i")   (l "n"))))
     ((b " " ) (w ("thi")  (l "s"))))
     ((b " " ) (w ("sentenc") (l "e"))))) (e ".")))
  (line
    (((b " " ) (w () (l "A"))))
     ((b " " ) (w ("secon") (l "d"))))
     ((b " " ) (w ("lin")  (l "e"))))
     (e "."))))
```

This is actually the result of parsing the two sentences "There are 6 words in this sentence. A second line." using the parser description `example1`.

We present a few examples using the functions defined in 'regregDrvr.scm'.

Example 1) The Scheme expression `(map tree->string (gstree-all 'w r))` returns  
 ("There" "are" "words" "in" "this" "sentence" "A" "second" "line")

Indeed, the subcall `(gstree-all 'w r)` has the value

```
((("Ther") (l "e"))
 ("ar") (l "e"))
 ("word") (l "s"))
 ("i") (l "n"))
 ("thi") (l "s"))
 ("sentenc") (l "e"))
 () (l "A"))
 ("secon") (l "d"))
 ("lin") (l "e"))
```

and the application of `tree->string` on each element converts the broken words to strings.

Example 2) The following Scheme expression gives all the last letters of all the words: `(gstree-all 'l r)` the result being `("e" "e" "s" "n" "s" "e" "A" "d" "e")`.

Example 3) The call `(gstree 's r)` returns `#f` as there are no special characters in these two lines.

Example 4) The expression `(tree->string r)` returns the complete string parsed, that is "There are 6 words in this sentence. A second line."

## 5 Debugging a parser

The file ‘`regregDrvr.scm`’ defines a macro named `drvr:debug-trace` that can be slightly modified to turn on a detailed trace of the running parser. It suffices to change `#f` to `#t` right after the `if` to turn on tracing.

The trace includes the following information:

1. The entry of the parser at each level.
2. The exit of the parser with its result at each level.
3. Each character being read and the list of active rules (level 1) before reading that character.
4. Each token being recognized at level  $n-1 > 0$  and the list of active rules at level  $n$  before it is recognized.
5. The possible transitions at level  $n > 1$  for each recognized token at level  $n-1$ .

The first two pieces of information are simple: They give a nested trace of the parser through its levels.

The list of active tokens is very instructive as you can observe when a rule is no longer recognized for a specific character or token. We recommend having the trace active when you design your parser. It provides a direct feedback and underscores misunderstood features.

The possible transitions at level  $n > 1$  are necessarily tokens from level  $n-1$ . These are possible transitions before the current recognized token is applied. If this last recognized token is not in that list, a parsing error will immediately occur.

Note: the possible transitions for level 1 are not traced as there could be a large number of those.

As an example, assume the parser description `out` from the section on the `<error>` token. If the trace is active, the parsing of ‘`2034Z .`’ gives:

```
Parser entering level 2
Parser entering level 1
Level 1, active rules: (<level-1> dot blank time), read char: #\2
Level 1, active rules: (time), read char: #\0
Level 1, active rules: (time), read char: #\3
Level 1, active rules: (time), read char: #\4
Level 1, active rules: (time), read char: #\Z
Level 1 result: (time ((h ("20")) (m ("34")) "Z"))
Level 2, active rules: (<level-2> line), token recognized: time
Possible transitions (time)
Parser entering level 1
Level 1, active rules: (<level-1> dot blank time), read char: #\space
Level 1, active rules: (blank), read char: #\.
Level 1 result: (blank (" "))
Level 2, active rules: (line), token recognized: blank
Possible transitions (blank)
Parser entering level 1
```

```
Level 1, active rules: (<level-1> dot blank time), read char: #\  
Level 1 result: (dot ".")  
Level 2, active rules: (line), token recognized: dot  
Possible transitions (dot time)  
Level 2 result: (line (((time ((h ("20")) (m ("34")) "Z"))  
(blank (" ")))) (dot ".")))
```



## 6 Elaborate RegReg examples: decoding meteorological bulletins

Meteorological bulletins are produced and disseminated around the world by thousands of centers. They are encoded for succinctness although they are read by human on a daily basis. For example, a Terminal Aerodrome Forecast (TAF) is produced by a meteorologist (a human, not a computer!), then either entirely typed by a human or via some computer software – in either case some TAFs contain typos and other errors which we are trying to recover from. This can be seen by simply scanning the thousands of TAFs provided by some agencies (e.g., the U.S. National Weather Service). Pilots decoding erroneous TAFs can in many cases make sense of them. An automatic decoders should try to do the same. For this reason, decoding TAFs is difficult: it should follow the formal definition of the lexical syntax of TAFs according to the standards, as well as attempt to interpret errors occurring in them.

PIREPs are pilot meteorological reports of direct observations from an aircraft. They are relayed by land stations.

On the other hand, AMDARs are produced automatically onboard aircrafts. They contain observations useful for other pilots, or as data points to forecast the weather. Even though they are produced by computers, many contain errors. The software written to produce them has sometimes been incorrectly programmed. In this case, hundreds of AMDARs are incorrectly produced each day. Decoding them is easier than the TAFs; yet, an AMDAR decoder should be easily and quickly modifiable to adapt it to observed errors. RegReg provides such a facility.

In the next subsections we present RegReg parsers to parse and decode these formats.

### 6.1 Decoding Terminal Aerodrome Forecast (TAF) bulletins

Terminal Aerodrome Forecast (TAF) bulletins are produced and disseminated by thousands of forecasting centers around the world on a daily basis – in the US and Canada they are produced four times a day. A TAF is a forecast of the local weather of an aerodrome for the next 24 hours. Despite the advent of sophisticated coding schemes (e.g., XML), the TAFs are still encoded using a scheme dating from the 1950s. Moreover, most of them are entered manually: they may contain human typing errors. They are difficult to parse for these two reasons.

We give a brief description of the TAF format and its meaning. A RegReg decoder is then presented for TAFs. For further information on TAF code you can consult the international standard for the TAF code, FM 51-X Ext. TAF, in the WMO Manual on Codes, WMO No. 306, volume I.1, part A. Some small amendments may have been done by some countries, in particular by various U.S. agencies (e.g., the U.S. Navy).

The structure of a TAF report is composed of a first global forecast followed by a series of sub-forecasts which may be further subdivided. This initial forecast must contain an interval of valid time and a station-id; it may contain an issue time. A sub-forecast, or period, starts with a token of the form FMhhmmZ?, where hh is a two digits hour and mm a two digits minute (the Z may be missing). This is a time from which the following tokens describe a modification of the initial forecast.

A subdivision of a FM period is started by BECMG, PROBxx or TEMPO token (the form 'PROBxx TEMPO' is also possible); followed by an interval of time of the form h1h2, where h1 and h2 are two digits hour, from h1 until h2. If h2 is smaller than h1, h2 refers to the next day.

The first period specifies an interval of time, which we refer to as the report-interval. But the FM periods do not explicitly specifies an interval, only the from time, although the until time may be inferred from the following subdivision.

Here are two examples of TAF bulletins:

A) A TAF from station-id 'KCTB'

```
TAF KCTB 072330Z 080024 26019G26KT P6SM SCT100 BKN150
FM0200 29014KT P6SM SC T100 BKN150 FM1000 35013KT P6SM SCT050
BKN100 PROB30 1216 3SM -SN BR BKN040 FM1600 34017G26KT P6SM
SCT070 BKN120
```

It has the following periods.

1. TAF KCTB 072330Z 080024 26019G26KT P6SM SCT100 BKN150
2. FM0200 29014KT P6SM SC T100 BKN150
3. FM1000 35013KT P6SM SCT050 BKN100
4. PROB30 1216 3SM -SN BR BKN040
5. FM1600 34017G26KT P6SM SCT070 BKN120

B) The following example has been divided in sub-periods on separate lines to ease human reading. These are not necessarily so produced by the forecasting centers.

```
KTEB 212340Z 220024 15008KT P6SM BKN060 OVC090
TEMPO 0002 BKN035 BKN070
FM0200 12004KT P6SM SCT025 BKN070
FM0700 12006KT 5SM -RA SCT015 OVC025
TEMPO 0710 2SM SHRA BR BKN010 OVC025
FM1200 VRB03KT 1 1/2SM -RA BR BKN009 OVC025
FM1500 19005KT P6SM SCT025 BKN070
FM2000 22010KT P6SM BKN045 OVC070=
```

Here is a RegReg TAF decoder:

```

(RegReg
  (declare (name taf))

  (macros
    (day_interval "{day=[0-9] [0-9]}{h1=[0-9] [0-9]}{h2=[0-9] [0-9]}"
      (interval "{h1=[0-9] [0-9]}{h2=[0-9] [0-9]}")
      (issue_time "[0-9] [0-9] [0-9] [0-9] [0-9] [0-9]Z?")
      (valid_time "{day=[0-9] [0-9]}?{hh=[0-9] [0-9]}{mm=[0-9] [0-9]}")
      (station_id "[A-Z] [A-Z] [A-Z] [A-Z]")
      (_ "[ \\013\\010]+"))

    ;; The issue_time can be missing. In that case, the valid_time
    ;; has the day of the forecast and the interval in hours.

    ;; Level 1 always succeed, assuming the message to parse does not
    ;; contain the EOT character. That is, no parse error can occur at this
    ;; level, since any non-empty sequence of characters is recognized by
    ;; it (except EOT).

    (level 1
      (main_period ((? _) (? "TAF" _) (? (: "COR" "AMD") _)
        station_id _
        (? issue_time _ (? (: "COR" "AMD") _) valid_time)
        (? (? (: "COR" "AMD") _) (? day_interval))))
        (from ("FM{hh=[0-9] [0-9]}{mm=[0-9] [0-9]}Z?")
          (tempo ("TEMPO" (? _) (? interval)))
          (prob ("PROB[0-9] [0-9]?" _ (? "TEMPO") (? interval)))
          (becoming ("BECMG" (? _ "TEMPO") (? _) (? interval)))
          (_ _)
          (token "[^=\\013\\010\\003 ]+")
          (endMessage "=")
        )

      (level 2
        (endMessage endMessage)
        (main_forecast ((? _) main_period (? _) (* token (? _))))
        (modification ((? _) (: tempo becoming prob from) (? _)
          (* token (? _))))
      )

      (level 3
        (taf (main_forecast (* modification) (? endMessage)) process-taf)
      )
    )
  )

```

The general structure of a TAF can be seen at level 3: it is a main forecast that may be followed by modifications. It usually end with the `endMessage` token, but this is not always the case as some centers do not produce it.

At level 2, we see that the main forecast is composed of the `main_period` and the forecast itself which is a list of tokens. A modification is one of the sub-forecast `tempo`, `becoming`, `prob`, or `tempo` header followed by the forecast itself in a series of tokens (which may be empty). These should be separated by spaces or newlines, but sometimes they are not due to human errors. Therefore we use a conditional space (`? _`) to catch these errors. Other simple errors are also detected, and silently ignored, for example a missing Z for some of the time-stamps.

At level 1, the details of each basic token are defined using the macros defined in the macro section.

There is only one function called after a single TAF has been parsed and a parse tree constructed, it is `process-taf`. This function is not presented in this document.

## 6.2 Decoding Pilot REPorts (PIREPs)

Pilot REPorts are direct meteorological observations reported from aircrafts by pilots. For example, they report observed turbulences.

The parser description was built according to the formal description found on pages 21-31 of AFMAN 15-124. We have also included variations of this formal description by analyzing PIREP bulletins from the U.S. National Weather Service.

The description of the parser is long since it decodes almost every piece of data in a PIREP. Here is a (long) list of PIREP examples:

```

SUA UA /OV SUA/TM 1116/FL190/TP SW4/SK OVC150-TOP165 CA/TB
LGT-MOD RA/IC LGT RIME 150-165=

HFF UA /OV SDZ220023/TM 2036/FL080/TP BE35/SK SCT AND BUILDING TO
THE SOUTH/WX FVO5SM HZ/TA 11C/RM SMOOTH RIDE=

FTW UA /OV FTW180010/TM 0018/FL028/TP M020/IC LGT RIME=

OWB UA /OV OWB/TM 2325/FLDURGD/TP JS32/SK OVC006-TOPUNKN/WX
FZRA/IC LGT RIME/RM A/C ON ILS RY36 BA POOR=

IND UA /OV VHP350015/TM 2330/FLDURGC/TP C441/SK OVC044 TOPS 090
HYR LYR ABV/IC NEG=

JBR UA /OV JBR/TM 0025/FLUNKN/TP PA34/SK OVC015/TA FL060 +4/FL015
-4/IC TRACE=

CYVC UA /OV YVC280020/TM 2323/FLVFR/TP DHC6/IC LGT MX 022/RM
PTCHY -FZDZ AT BASE OF CLOUD FROM 20W CYVC TO CYVC=

HRO UA /OV HRO150025/TM 0112/FL090/TP AC50/SK IMC/TB LGT/IC LGT
RIME/RM 145NW MEM OTP AT FL090=

DSM UA /OV DSM/TM 1147/FLUNKN/TP C56X/SK OVCUNKN-TOP038/CLR ABV=

LIT UA /OV LIT/TM 1144/FLUNKN/TP B757/SK OVC-TOP046/IC TRACE/RM
DURGD=

CYYB UA /OV YYB187015/TM 1140/FL080/TP DH8A/SK XXX OVC 068 CLR
ABV/IC TR-LGT RIME ICGIC 058-068=

RDG UA /OV RDG/TM 1445/FLUNKN/TP LJ60/SK OVC004/TB NEG BLO 100/IC
NEG/RM /FM A00 LISTENING/=

RDU UUA /OV RDU010020/TM 1023/FL080/TP PA31/SK IMC/WX +RW/TA
SFC-035/-1C 040-060/+6/TB LGT MDT CHOP 080/IC MOD CLR SFC-035/RM
DURGC=

SFB UA /OV SFB/TM 1935/FLUNKN/TP C172/SK 006 OVC/WX 1 1/2SM/RM
ILS 27R=

```

The RegReg decoder of PIREPs follows.

```

;; Notes:
;;
;; - The flight height is in mean sea level (MSL) unless the remarks
;;   section contains AGL. The remarks section may also contain DURC or
;;   DURD to specify if it was during Climbing or Descending.

```

```

;;
;;   - For sky cover: the remarks section may contain IMC; it means
;;     the aircraft is in clouds.

(RegReg

  (declare (name pirep))

  (macros
    (_      "[ \\010\\013]+" )
    (cloud   "{shyClear=SKC}|{few=FEW}|{scattered=SCT}|
              {broken=BKN}|{overcast=OVC}")
    (hhh     "[0-9][0-9][0-9]")
    (intensity "{trace=TRA?C?E?}|{light=LGT}|
               {moderate=MOD}|{moderate=MDTR?}|
               {severe=SEV}|{extreme=EXTRM}|{heavy=HVY}")
    (typeIce  "{rime=RIME?}|{clear=CLE?A?R}|{mixed=MI?XE?D?}")
  )

  (level 1
    (_      _)
    (OV      "/OV") (TA      "/TA") (WX      "/WX") (TM      "/TM")
    (FL      "/FL") (RM      "/RM")
    (SK      "/SK") (TP      "/TP") (TB      "/TB") (IC      "/IC")
    (header   ("[" ]*{cccc=[A-Z][A-Z][A-Z]}[ ]+"
              (: (= urgent "UUA") (= routine "UA" ))))
    ;; The location can be a line.
    (location
      ("/OV"
        (:
          ;; First is the most common case, something like: RDU010020
          ;; The airport/NavaId identifier should come first, but we accept
          ;; it after the direction and distance specification. Station should
          ;; usually be three or four letters. Three letters is an airport;
          ;; whereas four letters is a Navaid.
          ((: ( _ ({station1=[A-Z]+}
                  ({_}?{dir1=[0-9][0-9][0-9]}{dis1=[0-9]+})?))
              ( _ (({dir1=[0-9][0-9][0-9]}{dis1=[0-9]+})?
                    {_}?{station1=[A-Z]+})))
            ;; It could be a line with two stations, separated by "-" or dots.
            (? (? _) "(-|\\.+)"
              (: ((? _) ({station2=[A-Z]+}({_}?
                  {dir2=[0-9][0-9][0-9]}{dis2=[0-9]+})?))
                ((? _) (({dir2=[0-9][0-9][0-9]}{dis2=[0-9]+})?
                          {_}?{station2=[A-Z]+}))))))
        ))
      ;; Second case may come from AWW: lat and lon

```

```

({_}{lat=[0-9]+}({latN=N}|{latS=S})
  {_}{lon=[0-9]+}({lonW=W}|{lonE=E}))

;; Something rare like RDU APCH RWY 001 100FL
"_{_}{station1=[A-Z]+}_{_}[A-Z]+({_}?[A-Z0-9]+)*"
))
(time   "/TM{_-}?{TMhh=[0-9][0-9]}{TMmm=[0-9][0-9]}") ;; Hours minutes

;; /FL should not be followed by a space, but we still accept it.
(flightLevel   ("/FL" (? _ ) ( "{FLfrom={hhh}}(-{FLto={hhh}})?"
                                "{unknown=UNKN?}" "{FLascent=DURG?C}"
                                "{FLdescent=DURG?D}"))))

;; Type of Aircraft.
(typeAC   "/TP{_-}?({unknown=UN?KN}|{aircraft=[A-Z][A-Z0-9]+})")

;; A sky cover may contain several layers, each separated by a slash '/'.
(skyCover  ("/SK" (? _ )
            ((= from cloud) (? ( "-" _ ) (= to cloud))
             (? (? _ ) (= elev ( hhh "UNKN?" )))
             (? (? _ ) ( "-" ?TOP" "-" ) (? _ ) ( ( = top hhh ) "UNKN?" )))
            (* (? _ ) "/" (? _ )
              (cloud (? "-" cloud) (? ( hhh "UNKN" )
                (? "-"TOP" ( : hhh "UNKN" ) )))))

;; The weather section may contain slashes. The visibility may appear
;; anywhere in the section.

(weather   ("/WX" (? _ )
            (* ( ( = visibility "FV{_-}?{Vdis=[0-9]+}(SM)?" )
                "F[^V/\003=]+"
                "[^F/\003=]+"
                "/[^RTWI\003=]"
                "T[^AB\003=]"
                "I[^C\003=]"
                "W[^V\003=]"
                "R[^M\003=]"
                ))
            ))
(temp     ("/TA{_-}?"
            ( : "UN?KN?"
              ({Tminus=M|-}?{Tplus=\|+}?{_-}?
               {Tval=[0-9]+}_{_-}?{Tcelsius=C}?) ) ) )
(wind     "/WV{_-}?{WVdir=[0-9][0-9][0-9]}{WVspeed=[0-9][0-9][0-9]?}(KT)?" )

;; There may be several layers of turbulences.
(turb     ( : ( = nothing "/TB{_-}{noTurbulence=NEG|NIL}" )
              ("/TB"

```

```

(? _ (= occasional "OCNL"))
(? _ "{TBinten1={intensity}}" (? _ (= occasional "OCNL")))
  (? "(-|{_}){TBinten2={intensity}}")
(? _ (= turbType "{smooth=SMO?O?T?H}|{cat=CA?T}|{chop=CHO?P}"))
(? _
  (: ((THRU{_}|BL(O|W){_}|SFC({_}TO{_}-|{_})?)
    {TBto={hhh}}F?L?)
    "ABV{_}{TBfrom={hhh}}F?L?"
    "{TBfrom={hhh}}F?L?((-|{_}){TBto={hhh}}F?L?)?")
  )))
;; Similar to turbulences layers, but for icing.
(icing (: ("/IC{_}?{noIcing=(NEG|NIL)}")
  ("/IC" _
    (? (= occasional "OCNL{_}"))
    "{ICinten1={intensity}}((-|{_}){ICinten2={intensity}})?"
    (? _ typeIce) (? _ (: "ICE?" "ICG"))
    (? _
      (: ((THRU{_}|BL(O|W){_}|SFC({_}TO{_}-|{_})?)
        {ICto={hhh}}F?L?)
        "ABV{_}{ICfrom={hhh}}F?L?"
        "{ICfrom={hhh}}F?L?((-|{_}){ICto={hhh}}F?L?)?")
      )
    ))
  ))
;; The remarks section may contain slashes. A '=' ends that section.
(remarks ("/RM" (* "\\010|\\013") (= line "[^\\010\\013\\003=]+")
  (* (+ "\\010|\\013") (= line "[^\\010\\013\\003=]+"))
  ))
;; If it fails at any point:
(any "[^ \\010\\013\\003/=]+|/")
(endPirep "=" (short))
(endBulletin "\\003" (short))
)

(level 2
  (pirep ((* (: _ any))
    (*
      (:
        (= unrecognized
          (= tag (: OV TA WX TM FL RM SK TP TB IC))
          (* (: _ any)))
        (= header header (* (: _ any)))
        (= location location (* (: _ any)))
        (= time time (* (: _ any)))
        (= flightLevel flightLevel (* (: _ any)))
        (= typeAC typeAC (* (: _ any)))
        (= skyCover skyCover (* (: _ any)))
        (= weather weather (* (: _ any)))
      )
    )
  )
)

```



```

      (= temp      temp      (* (: _ any)))
      (= wind      wind      (* (: _ any)))
      (= turb      turb      (* (: _ any)))
      (= icing     icing     (* (: _ any)))
      (= remarks   remarks   (* (: _ any))))
    endPirep
    (? _))
  process-pirep
)
(endBulletin endBulletin)
(<error>      process-pirep-error))

(level 3
  (pireps ((+ pirep) (? endBulletin)))))

```

The level 3 of the parser simply decodes several PIREPs – it should end with a control-code 003, but it may be missing. The level 2 is written in such a way that the various small sections of a PIREP can occur in any order. This is what the long ‘:’ expression does. Note that the **any** lexem of level 1 is listed after all possible meaningful observations. It is therefore recognized only if all others are not – which is an almost ‘catch all’ lexem. This allows many typing errors (i.e., PIREPs incorrectly produced) that could not be recognized by the other lexems to be parsed by level 1. The lexem **any** is simply passed over by level 2 either as full lists (i.e., by `(* (: _ any))`) or at the end of every possible observations, again as a list of **any** lexems.

There is essentially only one Scheme function called for every PIREP: it is function `process-pirep`. If a PIREP could not be recognized at all, the `<error>` of level two would catch this case and call the function `process-pirep-error`. These functions are not presented in this documentation.

## 6.3 Decoding AMDARs and ACARs

An Aircraft Meteorological Data Relay (AMDAR) is a weather report automatically generated from an aircraft and relayed to a ground station. (ASDARs are similar but are transferred via satellites.) AMDARs are short reports containing position (i.e., latitude, longitude, and height) and observations from onboard instruments, typically wind speed, temperature, and turbulence. They are used as source of data points for weather modelling. The type of the system reporting the information is also provided. AMDARs, ACARs, and AIREPs have very similar encodings. The following decoders actually covers some ACARs encoding as it decode “section 3”.

Here are a few examples of AMDAR reports:

```

LVR EU5191 0435N 07318W 062322 F350 MS414 224/025 TB0 S011=
DES EU6287 4143N 01235E 041700 F044 PS042 /// 014/007 TB0 S031
    333 F044 VG008=
DES AU0040 3220S 15058E 070021 F336 MS468 248/032 TB0 S031
    333 F336 VG015=
LVR AU0032 4957N 01739E 070016 F330 MS584 243/087 TB0 S031
    333 F330 VG007=

```

The following decoder can process several AMDARs and ACARs.

```

(RegReg
  (declare (name amdar))

  (macros
    (_      "[ \\013\\010\\009]+")
    (dd     "({val=[0-9] [0-9] [0-9]?} | /// ?)")
    (nl     "[ \\013]*\\010[ \\013]*")
    (GGgg   "((= hh "[0-9] [0-9]") (= mm "[0-9] [0-9]")))
    (YYGG   "((= dd "[0-9] [0-9]" (= hh "[0-9] [0-9]")))
    (YYGGgg "((= dd "[0-9] [0-9]" (= hh "[0-9] [0-9]"
      (= mm "[0-9] [0-9]")))
    ;; Aircraft identifier
    (aircraft-id "{aircraft-id=[0-9A-Z]+}")
    (phase       "(= phase (: \"LVR\" \"ASC\" \"DES\" \"LVW\" \"UNS\" \"///\")))
    (lat         "{latV=[0-9] [0-9] [0-9] [0-9]} ({north=N} | {south=S})")
    (lon         "{lonV=[0-9] [0-9] [0-9] [0-9] [0-9]} ({west=W} | {east=E})")
    ;; height in hundreds of feet
    (FL          "({pos=F} | {neg=A}) {Fval=[0-9]+}")
    ;; temp dew point in tenths of a degree Celsius
    (tempDewPoint "({TDpos=PS?} | {TDneg=MS?}) {TDval=[0-9]+}")
    (temp        "({Tpos=PS?} | {Tneg=MS?}) {Tval=[0-9]+}")
    (relHumidity "{relH=[0-9]+} | ///") ;; relative humidity
    (wind        "((({Wdir=[0-9] [0-9] [0-9]} | ///) / ({Wspeed=[0-9] [0-9]+} | /// ?))")
    (TBBA        "(: \"TB{turbVal=0|1|2|3|}\" \"///\")") ;; Turbulence
    ;; Type navigation, Type system, Temperature precision
    (Ss1s2s3     "S{typeNav=[0-1]} {typeSys=[0-5]} {tempPrec=[0-1]}")
    (Fhdhdhd     "F({presFL=[0-9]+} | / +)") ;; Acars, Pressure
    (VGfgfgfg    "VG({vertGust=[0-9]+} | / +)") ;; Acars, vertical gust
  )

  (level 1
    (section1 ((? _) "AMDAR" (? _ YYGG) _))
    (section2 ((= id ((? phase _) (? aircraft-id _))
      lat (? _ ) lon _
      (= time (: YYGGgg GGgg)) (? _ (: FL "///+"))))
      (? _ )
      (= syg
        (? temp (? _ (: "///" tempDewPoint relHumidity)))

```

```

                (? _ (: wind "////+"))
            (? _ TBBA)    (? _ Ss1s2s3)))
(section3    (= acars-section ("333" _ Fhdhdhd _ VGfgfgfg)))
(amdar_nil  ((? _) "NIL" (? _)))
(endBulletin "\\003")
(endMessage "=")
(_          _)
)

(level 2
  (amdar      ((? _) section2 (? _ section3) (? _) endMessage (? _)))■
    process-amdar)
  (section1    section1)
  (amdar_nil  (amdar_nil (? endMessage)))
  (_          _)
  (endBulletin endBulletin)
  (<error> process-amdar-error)
  )

(level 3
  (amdars ((? section1) (* (: amdar amdar_nil)) (? _) (? endBulletin))))■
)

```

## 7 Acknowledgements

The implementation is based on Dube and Feeley’s paper “Efficiently building a parse tree from a regular expression”, *Acta Informatica*, 37(2):121–144, December 2000. The technique has been modified to insert tags in the parse tree.

This work was done at FNMOC (Fleet Numerical Meteorology and Oceanography Center) for the US Navy. FNMOC uses RegReg to decode meteorological bulletins.

This documentation was last modified in October 2006.