RegReg: a Lightweight Generator of Robust Parsers for Irregular Languages

Mario Latendresse

Northrop Grumman IT
Technology Advancement Group/FNMOC/U.S. Navy
E-mail: mario.latendresse.ca@metnet.navy.mil

Abstract

In reverse engineering, parsing may be partially done to extract lightweight source models. Parsing code containing preprocessing directives, syntactical errors and embedded languages is a difficult task using context-free grammars. Several researchers have proposed some form of lexical analyzer to parse such code. We present a lightweight tool, called RegReg, based on a hierarchy of lexers described by tagged regular expressions. By using tags, the automatically generated parse tree can be easily manipulated. The ability to control the matching rule mechanism for each regular expression increases efficiency and disambiguation choices. RegReg is lightweight as it uses a minimal number of features and its implementation uses only deterministic automaton. It has been implemented in Scheme which allows extending the tool in a functional programming style. We demonstrate how RegReg can be used to implement island and fuzzy parsing. RegReg is publicly available under a BSD-like license.

1. Introduction

Source code containing syntactically incorrect code, macro expansion, and embedded languages is a challenge for parsing. A formal description (i.e. context-free grammar) of the underlying language may be very complex to do; if not impossible in some cases. We qualify such languages as *irregular*.

Extracting specific source models for irregular languages is a challenge. An approximate, or partial, parsing strategy may suit the task at hand since using a full fledged solution may be too time consuming. Parsing using regular expressions is such an approximation. It cannot replace the more powerful context-free grammar parsers, but it can be much simpler to build to extract partial models of a source code.

There are several works addressing the approximate parsing [6, 5, 13, 20, 22, 18, 19, 24, 12, 4] in the context of source code analysis.

Very interestingly, all of them, despite first appearances, use, in practice, regular languages. One group uses a form of *hierarchical lexers* [6, 5] and for [18, 19, 24, 12, 4] such a hierarchy can be used to solve the presented cases.

We advocate, for partial and robust parsers, the use of deterministic automaton built from a cascade (hierarchy) of lexers, where the input of the first lexer is a stream of characters, the second lexer input is the stream of tokens generated from the first level, etc. This approach still does not require the specification of a context-free grammar, offers simple mechanisms for disambiguation, is more efficient than a single lexer, and allows the user to decompose the parsers in stages with intermediary processing.

This supports the design of our tool, called RegReg, a parser generator based on a cascade of lexers and the automatic construction of parse trees. It also offers *tagged regular expressions* to generate tagged parse trees: they are easy to process. We also present simple and efficient ways to control the matching rule mechanism such that lexical ambiguities can be controlled by the user to enforce the desired choice.

Tools to generate parsers based on a cascade of lexers have also been presented in [5, 3]. In [8], a cascade of parsers is used; and techniques and tools have been published [11, 10, 16, 17, 7] to extend the regular expression compilation to DFA (Deterministic Finite Automaton) or NFA (Non-deterministic Finite Automaton) such that parse trees can automatically be built. These tools are not as simple and extendible as RegReg—and none of them are publicly available.

Efficiency, in particular determinism, is an important feature advocated throughout the design of RegReg. No mechanisms are ever introduced to slow down parsing—in particular, introducing backtracking.

In Section 2 we present previous work to motivate the design decisions of RegReg. Section 3 briefly presents RegReg's parser description language. Section 4 delves more deeply into disambiguation through the matching rule mechanisms. The essential implementation detail of RegReg is presented in Section 5. Section 6 presents parsing examples using RegReg, and in particular compare it to SDF capabilities through a similar paradigm as the island parsing approach. Section 7 addresses macro expansion and conditional preprocessing. We present a summary and future work in Section 8.

2. Motivation and Related Work

Building partial and robust parsers using regular expressions is not new. The analysis of the previous work motivates the approach taken for RegReg.

Cox and Clarke present two studies [6, 5] on parsing code using regular expressions. Their goal is not to do partial parsing, but to completely parse as, say, a LR parser. This is not the intended goal of RegReg, but some of their results are instructive about the capabilities and limits of parsing using regular expressions.

In their first work [5], a cascade of lexical scanners—similar to RegReg—was used. They name their generator MULTILEX—unfortunately it is not publicly available. They show good recognition capabilities, except in the case of nested constructs like arithmetic expressions.

In the more recent work [6], parsing is based on iterative applications of lexers. Once a match has been found at one level, the resulting matched string is rewritten and rescanned by the next level. By appropriate rewriting it is possible to recognize some language constructs that are context-free. Their goal is to remove the shortcomings of recognizing nested constructs as done in [5]. The main problem of their approach is the execution time: it took over six minutes to parse a 8464 lines C program. Although, this may be due to several reasons: their implementation was not time efficient, the rewrite steps were too small, or re-scanning is based on the lexicographic content of tokens—'constReal' and 'constInt' could both be recognized by the regular expression `const.*`. They show that this approach is better than the cascade of lexers of their previous work. Yet, their experimental approach is disputable: the code was preprocessed before parsing. This partly nullifies the lexical approach to robustly parse code with preprocessing elements, since they had to fix a certain number of free preprocessing variables, probably eliminating some parts of the code, and expanding all macros to some specific values[1]. Interestingly, they do recognize the usefulness for a shortest-match strategy as in RegReg.

Bickmore and Filman [3] has designed MultiLex—not to be confused with MULTILEX of Cox and Clarke—a parser generator based on cascade of lexers, as in RegReg. But unlike RegReg, it uses backtracking, that is each level is implemented by a non deterministic automaton; and it has no facility to bind parts of matched strings with identifiers, which RegReg provides. It has been written in Common Lisp, but is not publicly available.

Dyadkin [8] proposes Multibox, a generator of syntactical analyzers based on a cascade of parsers. Each level, or box, is described by a LL(1) grammar. Unfortunately, the generator is not publicly available and it is unclear how easy it can be used to build robust parsers.

TLex [11, 10] is an advanced scanner generator. It has a similar approach to RegReg in the identification of sub-parts of the matched string: bindings may be specified in the regular expressions. It does not use a cascade of lexers, though. The API to retrieve the parse tree is rather complex. Unfortunately, the TLex software is no longer available.

The ASF+SDF Meta-Environment [23] is designed to build parsers and transformers. To describe a parser, it uses the conventional approach of two stage descriptions—regular and context-free grammars. The user specifiable control over the lexer is the use of sorts, rejection, and ordering. It has no short matching rule as offered by RegReg. The ASF+SDF meta-environment might be used for the construction of robust parsers, but: it is not lightweight requiring a good investment in learning; and the implementation is based on a deterministic simulation of nondeterminism, scannerless Generalized LR parsing (SGLR), which may turn out to be very inefficient on some ambiguous grammars. In Section 6.2 we compare the parsing capabilities of ASF+SDF with RegReg.

Revealer [20] parsing technique is entirely based on regular expressions. The syntax to specify parsers is based on XML. This is an unfortunate choice since XML verbosity makes it user-unfriendly. The implementation is based on the Perl language which uses non-deterministic automaton for regular expression compilation. Consequently, Revealer does not provide any new implementation technique as a tool but

1 A macro may have different values in the code depending on the free preprocessing variables.

rather a framework of components to parse and extract source models.

Van den Brand *et al.* [22] give an analysis of parsing technologies in the context of irregular languages. Their general conclusion is that building context-free grammars for such languages is not an easy task. They promote GLR parsing, even scannerless GLR parsing where the boundary between the lexer and the syntactical parser is eliminated by using non-regular grammar to specify the lexer.

Cascade of lexers is also used in natural language parsing [1]. Although not truly natural languages, meteorological bulletins—as defined by the World Meteorological Organization (WMO)—have complex lexical definitions: we have used RegReg to generate parsers for them.

These works have a clear recurring element: robust and partial parsers can be built using regular grammars or a hierarchy of such grammars—supporting the basic design approach of RegReg.

3. RegReg Parser Description

In this section we briefly present RegReg's description language and how to instantiate parsers. Complete details can be found in the RegReg documentation [14].

Figure 1 presents a basic example of a complete parser description of three levels. At each level a list of definitions is given. A definition is at least a symbol and a regular expression. The definitions order is relevant: the top ones are qualified as higher than the lower ones; more on the use of order in Section 4.

Level 1 deals only with characters; level 2 is based on tokens produced by level 1; and level 3 is based on tokens from level 2. In general, no limit is set on the number of levels, although at least one level 1 is required.

The syntax of regular expressions can be specified with s-expressions using prefix operators '`:`', '`*`', '`+`', '`?`', etc., as token `line` at level 2. A Lex-like syntax can also be used using binary infix operator '`|`' and unary postfix operators '`?`', '`*`', '`+`', as token `w` at level 1. In that case it is a string. Both ways can be used simultaneously as for token `n` at level 1. Non-printable characters are coded using '\' and their encoding values; but since it is an escape character in Scheme strings, it must be doubled. The dot ('`.`') represents all possible characters of the underlying Scheme implementation.

In some cases it is preferable to use the infix notation for succinctness (i.e. sets of characters as in `"[A-Za-z]"`) but in some cases it is preferable to use s-expressions as in the second and third levels which refer to tokens. An s-expression is more amenable to indentation, so it should be used for complex and long regular expressions.

```
1.  (RegReg
2.    (declare (name example1))
3.    (macros
4.      (blank "[ \\010\\013]"))
5.    (level 1
6.       (w  "{p=[A-Za-z]*}{l=[A-Za-z]}")
7.       (n  ((= f "[1-9]") "{r=[0-9]*}"))
8.       (s  "[-,=]")
9.       (e  "[.;:!]")
10.      (b  "{blank}+"))
11.   (level 2
12.      (line ((+ (? b) (: s w n)) e)))
13.   (level 3
14.      (text (* line)  process-text)))
```

Figure 1. A parser description in RegReg

The s-expression 'macros' is a section to describe frequently occurring regular sub-expressions. They can be referred to by any level by enclosing its name with curly braces when using the string syntax, as in token `b` at level 1, or simply as an identifier in a s-expression.

Line 2 names the resulting structure describing the parser with which we can instantiate a real parser at run-time. Each level is an autonomous lexer, implemented as a deterministic finite automaton (DFA). The input of level 1 is the character stream available via the standard input functions in Scheme. The input of a level $i > 1$ is the stream of tokens generated from the level $i - 1$; it has also access to the parse trees of level $i - 1$.

RegReg regular expressions are actually tagged regular expressions (TRE). Tagging is done using the binding equal operator '`=`'. For example, for token `w` at level 1, in '`{l=[A-Za-z]}`', the '`=`' binds the matching substring with the identifier `l`. In that case it can only be one letter. The other binding operation in `w` is '`{p=[A-Za-z]*}`' between `p` and the substring matching '`[A-Za-z]*`'. For example, the string `"HELLO"` would create the bindings $p \rightarrow$ "HELL" and $l \rightarrow$ "O". For token `n` at level 1, there is a binding for `f`, using the s-expression syntax, and a binding for `r`.

For every TRE definition, there is an implicit binding between the identifier for that definition and the resulting tagged parse tree.

When a match is found against a string s, a parse tree is built over s according to the structure of the regular expression. The operators '`*`', '`+`' and sequencing generates lists; and list of characters are converted to strings. A sub-expression in a TRE generates a subtree. Note that the choice operator '`|`' (':' for s-expression) disappears since only one choice is assumed after the

```
(text
  ((line
     ((((()      (w ((p "Her") (l "e"))))
       ((b " ") (w ((p "ar") (l "e"))))
       ((b " ") (n ((f "3") ())))
       ((b " ") (w ((p "word") (l "s")))))
       (e ".")))
   (line
     ((((b " ") (w (() (l "A"))))
       ((b " ") (w ((p "secon") (l "d"))))
       ((b " ") (w ((p "lin") (l "e")))))
       (e ".")))))
```

Figure 2. A tagged parse tree

match is done—ambiguities are always resolved to one case. Bindings are represented as 2-tuples, with the identifier first and the subtree second.

For example, the string `"Here are 3 words. A second line."` gives the parse tree of Figure 2.

Note that for the substring '3', the binding for `r` disappears since there is no substring matching the sub-TRE '`[0-9]*`'.

For each definition, the name of a Scheme function may be specified. It is called with the resulting tagged parse tree when the parser finds a match. The function must return a pair whose head is a symbol and tail is a tagged parse tree. This can simply be the input of the function or a modified version of it. Therefore, user functions can manipulate the intermediate results of the parse and pass modified intermediate results to higher levels. This approach is still in the functional programming style since the other levels receiving the results are unaware of the existence of these functions.

Adding intermediate processing functions becomes necessary if the analyzed files are large. Otherwise, huge parse trees are built requiring large amount of heap space before any processing is done. And in most cases, it is simpler to process intermediate well-defined results. This is the case for lightweight source models extraction.

The user functions can manipulate the parse trees without any API; but a simple one is provided by the scanner, in the form of four functions, which fulfills most needs. Function `gstree(`k, t`)`, where k is a symbol and t a tagged parse tree, returns the subtree which is bound to k (it does a pre-order search); function `gstree-all(`k, t`)` returns all subtrees in t which are bound to k; function `tree->string(`t`)` converts a parse tree t to the string used to build it; and `gstks(`k, t`)` is the composition of `gstree` and `tree->string`.

```
1. ;; I: s, a string
2. ;;    na, a vector of DFAs generated
3. ;;        by RegReg.
4. ;; O: parse tree of s.
5. ;;
6. (define (parse-string s na)
7.  (with-input-from-string s
8.   (lambda ()
9.    ((make-parser read-char na "")))))
```

Figure 3. Instantiating a parser

By default the longest matching rule is used. But qualifiers `short` or `prefer` can be specified for each definition to modify that behavior. Parse tree construction can be deactivated with qualifier `discard`. Section 4 covers these.

Once a parser description is compiled by RegReg into a structure describing it, a functional parser can be instantiated via the driver; this is covered in the next subsection.

3.1. Instantiating a parser

Figure 3 presents a simple example of a function instantiating a parser from a cascade of lexers bound to the variable `na`—the result of compiling a parser description. The function `make-parser` returns a function of arity zero; it allocates all necessary buffer spaces which can grow as much as the heap space can allow. Identifier `read-char` is the usual Scheme function to read one character from the current input—in this case from the string bound to `s`. The empty string could be a non-null prefix string for the parser to read before using `read-char`. On line 9, the parser is immediately called after its creation. Consequently, the function `parse-string` returns the tagged parse tree of `s`.

This example demonstrates that parsers are dynamically created at run-time. This operation is efficient as minimal amounts of buffer spaces are allocated. More complex situations may arise where different parsers are created to serve other needs found at execution time.

4. Lexical Disambiguation by Controlling the Matching Rule Mechanism

This section presents the control, offered by RegReg, over the matching rule mechanism to disambiguate lexical matching.

Indeed, ambiguity abounds in a lexical analyzer unless some mechanisms are provided to enforce a choice. The most common disambiguation mechanisms are based on what to do next when an accepting state

of a regular expression is met. This can be a shortest matching rule, the longest matching rule or variations of these.

Most scanner generators assume the longest matching rule, but this is not always useful. In terms of the automaton, the longest matching rule is: as long as the scanner matches the input, it continues processing; if a failing state is reached it returns to the last accepting state (if any). If two accepting states are met, the one associated with the highest definition is chosen. This is the default rule in RegReg among the rules of one level.

We provide two qualifiers to override this longest matching rule: `short` and `prefer`. The `short` qualifier forces the scanner to stop as soon as an accepting state is reached; whereas `prefer` forces the scanner to consider preferred tokens only once an accepting preferred state is reached—the scanner continues to search for the longest match for the preferred tokens.

We will see in Section 6 that these two options combined are more appropriate, for robust parsers, than the ordering of sorts as in SDF; this is mainly due to the `short` qualifier, unavailable in SDF.

The following two subsections present more precisely the `short` and `prefer` qualifiers. The third subsection presents the `discard` qualifier to deactivate the parse tree construction.

4.1. The `short` qualifier

The qualifier `short`, applied to a TRE, enforces a shortest matching rule for that TRE: as soon as the automaton finds an accepting state for the qualified TRE, it stops. It would override any `prefer` qualification. If two TREs are qualified with `short`, the highest defined TRE is chosen.

Figure 4 shows a common example for the use of a shortest matching rule. The token `start` describes a string starting with '`BEGIN`' and ending with '`END`', with any characters in between. If the longest match is used, the scanner would read the entire input before making its decision; but the user wants to stop as soon as the word '`END`' is scanned after seeing '`BEGIN`'. The qualifier `short` just does that. Therefore, for the string '`BEGIN hello END Allo END`' only '`BEGIN hello END`' matches.

Note that this behavior could be described by disallowing '`END`' in the sub-regular expression '`.*`'. If for example, the operator minus (`-`) were provided, we could have written '`(.*)-(.*END.*)`' to specify all strings but not the ones containing '`END`'. But such constructions—and in general the minus and complement operators—are quite costly in space (DFA) whereas the short qualifier reduces space and has no running time cost.

```
1. (RegReg
2.   (declare (name example_short))
3.   (level 1
4.     (start  "BEGIN.*END" (short))))
```

Figure 4. Shortest rule applied to a TRE

```
1. (RegReg
2.  (declare (name example_prefer))
3.  (macros (L "[^;{}]*"))
4.  (level 1
5.   (for "for\\({L};?{L};?{L}\\)?{L}(;|\\{)"
6.        (prefer) process-for)
7.   (blank "[ \\010]+")
8.   (token "[^ \\010]+")))
9.  (level 2
10.  ((file (* (: blank token for))))))
```

Figure 5. Qualifier `prefer`

Some uses of `short` contradict the TRE description. For example, applying `short` to '`[a-z]+`' does not make sense, since the scanner would stop as soon as a lower case letter is scanned. Thus, the TRE should have been described as '`[a-z]`'.

Its implementation is quite simple: the final nodes of a regular expression qualified as `short` are stripped of any outgoing transitions; see Subsection 5.1 for more detail.

4.2. The `prefer` qualifier

Figure 5 defines a regular expression for a `for` statement: it has several optional parts to catch several possible syntactic errors—it is indeed very liberal. For example, it matches the syntactically incorrect '`for(){`' and '`for(i=0;;;`'. It is qualified as `prefer` to avoid recognizing a `for` statement followed by non spaces as a `token`. For example, without a `prefer` qualifier, the string '`for(;;);i=0`' would be recognized as a `token`, skipping the syntactically correct `for` statement, since the longest matching rule would be applied.

A TRE qualified with `prefer` is called a preferred TRE.

The `short` qualifier does not offer the correct behavior in that case. For example, it would stop the scanner as soon as '`for(;`' is scanned in '`for(;i<10;);`', but the whole string should be scanned to catch the entire `for` statement.

The meaning of `prefer`, derived from the implementation as presented in Subsection 5.2, is as follows.

While scanning, if at least one preferred accepting TRE is reached all non preferred TREs are no longer considered and all the preferred TREs that are lower than the highest accepting TRE reached are also no longer considered. Note that this does not imply any additional work on the part of the scanner, as the automaton is built in such a way to automatically offer this semantics.

No contradictory statements, as for the `short` qualifier, can be made with `prefer`. For example, qualifying `[a-z]+` as `prefer` is not contradictory since the longest match is sought.

5. An Efficient Implementation

RegReg, the generator as well as the driver, is currently implemented in Scheme—a minimalist language in programming concepts with a functional subset. Consequently, we believe it can easily be ported to many other functional languages like OCaml, Haskell, Common Lisp, etc. The parser generator has only around 1500 lines of code, and the scanner driver has around 800 lines of code. Scheme with its advanced data structures and high order functions provides relevant programming features for succinct programming.

The approach taken for RegReg is to provide a lightweight tool with few bells and whistles, but enough to build robust and efficient parsers. The added features to control the matching rules were chosen to maintain an efficient tool and a clear semantics—this is demonstrated in this section by presenting some details of its implementation.

We assume the reader is familiar with deterministic finite automaton construction (DFA) from a regular expression as described in [2]. Such an automaton can recognize a string, but it does not build a parse tree for it.

The automatic generation of the tagged parse trees, by the lexer, is based on approach described in [7]. We have modified it to avoid the use of a matrix for chains of operations, eliminated the use of the operation `sel` since we generate only one parse tree for each match, and introduced a new operation `set` to bind subtrees to identifiers.

The technique is essentially as follows. While translating a TRE to a NFA, every transition is annotated with one of the following stack operations:

pushE Pushes an empty list on top of the stack;

push Pushes an input character, or a token subtree pair from a lower level automaton, on top of the stack;

snoc Conses the top stack element at the end of the list of the second top stack element; removes the top stack element;

nop Does nothing;

set Binds an identifier to the top stack element.

Furthermore, the transformation from NFA to DFA generates chains of operations on every DFA transition.

At parse-time, these chains are used to construct the parse tree once an accepting state is reached. The right sequence of chains is found by going from the accepting state back to the start state. This chain is applied to the input using a stack; which is empty at the beginning of the parse. The result, on top of the stack, is a tagged parse tree; it is removed and passed to the user function or the higher level.

Other efficient approaches has been described, in detail, in the literature: Kearns [10, 11] and Laurikari [16, 17] describe different approaches to automatically build parse trees from which subtrees can be efficiently extracted by some forms of addressing. Dubé and Feeley's approach appears the simplest to implement as it integrates well with the conventional Thompson-Glushkov technique [9, 21, 2] of automaton construction from a regular expression. Yet, it is unclear which technique is the most efficient.

We describe how the qualifiers `prefer`, `short` and `discard` are implemented in the following subsections.

5.1. Implementation of qualifier `short`

The qualifier `short` has a straightforward implementation.

As usual, during the transformation from NFA to DFA, a DFA node represents a set of NFA nodes; and if a DFA node contains a NFA accepting node, it becomes a DFA accepting node. But if the NFA accepting node is associated with a TRE qualified as `short`, then all outgoing transitions are removed from the DFA node containing it. Moreover, the DFA node is marked by that NFA node: its corresponding token is associated with the DFA node. If a DFA node contains more than one NFA `short` node, the one associated with the highest TRE definition is chosen.

Note that during scanning, this removal of all transitions makes it irrelevant to verify if a node is marked as `short`. And this increases speed since no looking ahead is done—which is the usual behavior for the longest matching rule—when reaching that accepting node.

A clear and precise semantics can be derived from this implementation: for one level, when an accepting state is reached for at least one TRE qualified with

`short`, the scanned string is immediately attributed to the highest one of those TREs.

5.2. Implementation of qualifier `prefer`

The `prefer` qualifier implementation is similar to `short` but slightly more complicated. In the following, a preferred NFA node is from a preferred TRE; and NFA nodes are ordered according to their corresponding TREs.

A preferred TRE is transformed to a NFA with preferred nodes only. As usual, for the NFA to DFA transformation, the set of NFA nodes making up a DFA node depends on the set of NFA nodes outgoing transitions; but this set is curtailed as follows.

If the set contains no NFA accepting node, the set is not reduced; but if there is at least one accepting preferred NFA node: 1) the DFA node is accepting and its representative is the highest preferred accepting NFA node; 2) the non preferred NFA nodes and the preferred NFA nodes which are lower than the representative accepting node are removed from the NFA nodes forming this DFA node. This reduction in the set of NFA nodes also reduces the number of outgoing transitions, and forces the scanner to only consider the right preferred TRE once an accepting node of preferred TRE has been reached.

This implementation may increase the size of the automaton since more different sets of NFA nodes may be generated which would represent more DFA nodes. But it could also reduce it. It definitely does not slow down the scanner.

A precise and clear semantics for `prefer` can be derived from this implementation as explained in Subsection 4.2.

5.3. The `discard` qualifier

In Figure 6, the second level describes the entire file as a sequence of strings, tokens and blanks. The strings may not span several lines. Only strings are of interest, as only one function, namely `process-string`, is called for each string recognized. The entire parse tree describing the file is of no interest: constructing it could be an large task if the file is made of millions of lines; and the parser could run out of heap space. Therefore, it is efficient and more robust to discard the parse tree of `file` and simply return an empty parse tree as a result. The qualifier `discard` just does that. Note that `short` or `prefer` can be specified with `discard`, as in `(short discard)`; but `short` and `prefer` are incompatible.

```
1. (RegReg
2.  (declare (name example_discard))
3.  (level 1
4.   (blank "[ \\010\\013]+")
5.   (string "\"[^\"\\010\\013]*\"?"
6.           process-string)
7.   (token "[^ \"\\010\\013]+"))
8.  (level 2
9.    (file (* (: string token blank))
10.         (discard))))
```

Figure 6. Qualifier `discard` deactivates parse tree construction for a specific TRE

6. Robust Parsing Examples

Two well known approaches to partial and robust parsing are *fuzzy* [12, 4] and *island* parsing [18, 19, 24]. In this section we present some of the work based on these approaches and analyze how RegReg can solve the case studies addressed by these approaches. In the last subsection we present a general approach to building robust parsers in RegReg.

6.1. Fuzzy parsing

Fuzzy parsing [12, 4] is semi-formally defined in [12]. Its definition is based on context-free grammars and is not, *a priori*, a lexical approach. A *fuzzy parser* for a context-free grammar $G = (N, \Sigma, R, S)$, based on a set of anchors $A \subseteq \Sigma$—where N is the set of non-terminals, Σ the set of terminals, R the set of production rules and S the start symbol—is a set of sub-parsers P_a, one for each anchor symbol $a \in A$. Each sub-parser grammar is a subset of G based on one non-terminal. A partial parser is formed by scanning the input, and when an anchor a is found, the sub-parser P_a is called.

Even though its formal definition is based on context-free parsers, the fuzzy parser examples in [12] can be described by regular grammars. Moreover, the case studies of [12], done on the C++ language, preprocessed the input before parsing.

For example, the class tree generator ctg tool presented in [12], as well as CodeAnalyzer and Sniff, extract class definitions of C++ programs by using the anchors `class`, `struct`, `private`, `protected`, and `public`. They only need to scan the input, look for the anchors, and when one is recognized, they extract a minimal amount of information like the base class name. All the rest of the input is skipped. Ctg is based on the *flex* generator. Sniff has a hand-coded parser.

Such a partial parser can easily be specified using a regular grammar. In RegReg, each recognition of a class

declaration can be processed by one function maintaining a graph relation. All the rest is discarded using the `discard` qualifier. The parser is very efficient since only the parse trees of the class headers are built.

On a 40 MHZ SPARCstation IPX, the rate of parsing of ctg and Sniff is around 100 KB per second, and CodeAnalyzer is around 40 KB per second. Ctg and Sniff parsing speeds are difficult to beat since they are based on deterministic automaton.

Sniff has 1500 lines of C++ code. The overall CodeAnalyzer program has 10000 lines of C++ code. And these are only *instances* of fuzzy parsers, to extract the hierarchy of C++ classes, and not tools to generate such parsers, like RegReg.

The overall complexities of these implementations show the need of a custom designed tool for partial and robust parsing.

6.2. Island parsing

The goal of island parsing [18, 19, 24] is to create partial and robust parsers. An island grammar definition is based on a context-free grammar G. The definition given in [18] is based on a set of *constructs of interest* $I \subseteq \Sigma^*$, which are substrings of $L(G)$. An island parser recognize the substrings of I (island) in the strings Σ^*.

There are no specific tools to build these parsers, as any tool could potentially be used. Moonen and Verhoeven use the ASF+SDF Meta-Environment to build island parsers.

Interestingly, all examples in [18, 19, 24] of island parsers, based on the SDF tool, used regular languages. They use the context-free description of SDF, but these turn out to describe regular languages. (Although, the case study of [19] has a grammar of 148 productions which is not completely presented.)

For example, in Figure 7 is a simple COBOL island parser, from [24] (pp. 21–22, we have combined two descriptions to make the example self-contained), described in SDF to extract copy statements. Although it uses the context-free SDF section, this grammar is regular. This is the case for all parser descriptions presented in [24]. RegReg can handle all these cases; but with greater simplicity as demonstrated below.

Figure 8 presents the equivalent description in RegReg. We can readily prove that this parser is complete: at level 1 the three tokens `_`, `dot` and `token` cover all possible input characters and level 2 refers to all possible combination. The token `Island` uses the `prefer` qualifier to force its selection over `token` when such an input sequence occurs. Note that even though we have to specify all lexical details in RegReg, like the layout token `_` which is pre-defined in SDF, the description remains succinct.

```
1. lexical syntax
2.  [A-Z][A-Za-z0-9\-\_]* -> Id
3. lexical restrictions
4.  Id -/- [A-Za-z0-9\-\_]
5. context-free syntax
6.  "COPY" Id  -> Copy
7.  Copy       -> Island
8.  Island     -> Token
9.  Drop       -> Token
10.  Token* "." -> Sentence
11.  Sentence*  -> Program
12. context-free priorities
13.  Island -> Token >
14.  Drop   -> Token
```

Figure 7. A COBOL island parser in SDF[24]

```
1. (RegReg
2.   (declare (name COBOL_copy))
3.   (macros
4.     (_       "[ \\010]+")
5.     (Id      "[A-Z][A-Za-z0-9\-\_]*"))
6.   (level 1
7.     (Island  ("COPY" _ Id) (prefer))
8.     (Token   "[^. \\010]+")
9.     (Dot     ".")
10.    (_       _))
11.  (level 2
12.    (Program
13.      (* (= Sentence
14.            (* (: Token _ Island)) Dot)))))
```

Figure 8. A RegReg equivalent of Fig. 7

In [24] the running time comparisons of SDF and Perl are not in favor of SDF: Perl can be an order of magnitude faster than SDF. This is not SDF's fault, but points to the fact that island parsers tend to be based on regular expressions which is Perl's strength.

RegReg can be faster than Perl as demonstrated in Table 1. The C code of 102KB is GC-Boehm `os_dep.c` file; the other two files are duplicated concatenations. They were scanned to find calling statements of one variable argument, as described by the regular expression '`.*[a-zA-Z]+[]*\\([a-zA-Z]+\\)`'. The island parser Scheme code was compiled using Bigloo 2.5c with option `-O4`. Perl v5.6.0 was used with a script looking at every line for that pattern. All executions were done on a 500MHz Pentium III. RegReg is around 60% faster for that search. This is not a thorough benchmarking as Perl has chaotic execution speed due to non-determinism. But this shows the sound implementation approach of RegReg. It also shows that since Perl is faster than SDF, RegReg may be expected to

File size	Perl (sec.)	RegReg (sec.)
102KB	1.94	1.18
204KB	3.85	2.37
816KB	7.73	4.65

Table 1. Perl vs. RegReg timings, searching calling statements

```
1. (RegReg
2.   (declare (name robust1))
3.   (level 1
4.     (case1 ... (prefer))
5.      ...
6.     (tk    "[^ \\010]+")
7.     (_     "[ \\010]+"))
8.   (level 2
9.     (file (* (: tk _ case1 ...)
10.             (discard)))))
```

Figure 9. The general structure for robust parsers in RegReg

be an order of magnitude faster than SDF. But more benchmarks should be used to compare them.

6.3. Partial and Robust parsing in general

We believe that the general approach of island parsing, where a very general parser is first specified, and then specific constructs (islands) are defined, is the most flexible paradigm for partial and robust parsers. Moreover, we advocate that the simplest way to do so is through regular grammars only, since they allow simpler control of ambiguities.

Figure 9 presents the general structure of such robust parsers in RegReg. The `tk`, `_`, and `file` tokens are enough to parse any file. To extract some specific source models, new cases should be added with the `prefer` qualifier. In some cases, the qualifier `short` should be used to recognize prefixes as mentioned in Subsection 4.1. Each additional token case should also be added to the `file` token and should provide its own function to extract the relevant information. The `discard` qualifier should only be removed if the overall parse must be analyzed—which is quite rare for the extraction of lightweight source models.

When adding a new construct, if it already exists as a prefix of another preferred token, it should be added first. More levels may be needed if the added cases are complex.

```
1. #if defined(T)
2. #  define X  <
3. #else
4. #  define X  ==
5. #endif
6. if (a X y) y++; else a++;
```

Figure 10. Parsing with macro expansion

7. Macros and Preprocessing

Textual preprocessing with conditional compilation and macro expansion may greatly hinder parsing. A simple table facility with an emulation of expansion can provide an approximate solution to it, but it is far from being a complete solution: due to conditional compilation a macro may have several values at expansion time. A simple table mechanism cannot solve such a problem.

For example, in the code segment of Figure 10 the parameterless macro `X` has two possible values at line 6. A form of "conditional binding" is needed at line 6 to handle all possible cases.

As far as we know, and despite numerous research projects done on this subject, no tool has ever been designed to handle such parsing. For example, Sniff [4] does not expand such macros. We intend to integrate the approach of [15] in RegReg to do such parsing.

8. Summary and Future Work

RegReg is a tool to generate robust parsers based on a cascade of lexers, automatically building parse trees, and allowing succinct specification through *tagged* regular expressions. It is available under a BSD-like license at [14]. As far as we know, there are no other publicly available tools to generate parsers based on a hierarchy of lexers.

RegReg, unlike Perl, strives for efficiency by using only deterministic automata adding no features that would require backtracking. Moreover, the added features to control the matching rule mechanism have a tendency to increase speed and reduce automaton size. The user may even selectively deactivate the construction of parse trees for each regular expression to further increase efficiency.

We have shown that by using a cascade of lexers, automatically built tagged parse trees and a minimalist number of features to control matching, we can succinctly describe robust parsers to extract lightweight source models over irregular languages. We have more specifically shown that such approaches as island and fuzzy parsings, as used in practice, can easily be implemented in RegReg.

RegReg is a lightweight tool as it provides the right features for partial parsing; and at the same time, it is implemented to allow extensions through the underlying Scheme language.

The use of tagged parse trees, built automatically from the tagged regular expressions, meshes well with the purely functional programming style available in Scheme. The parser driver is designed to avoid any uncomfortable limitations, i.e., limited buffer size or fixed number of lexer levels. Parsers are created dynamically by a function call allowing well defined integration of RegReg into other tools written in Scheme.

As described in Section 7, to completely parse in the presence of macros and conditional compilation, macros should be expanded and a control flow analysis of preprocessing directives should be done. This should be provided by the tool. To do so in RegReg, we intend to incorporate the approach described in [15].

Several different approaches are described in the literature to automatically build parse trees from regular expressions. We have used the approach of [7], yet other approaches [16, 10] might turn out to be more efficient in practice.

References

[1] S. Abney. Partial parsing via finite-state cascades. In *Proc. of Workshop on Robust Parsing, 8th European Summer School in Logic, Language and Information, Prague, Czech Republic*, pages 8–15, 1996.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers; principles, techniques and tools.* Addison-Wesley, 1986.

[3] T. Bickmore and R. E. Filman. MultiLex, a pipelined lexical analyzer. *Software Practice and Experience*, 27(1):25–32, Jan. 1997.

[4] W. R. Bischofberger. Sniff: A pragmatic approach to a C++ programming environment. In *Proc. USENIX C++ Conference*, pages 67–82, 1992.

[5] A. Cox and C. Clarke. A comparative evaluation of techniques for syntactic level source code analysis, 7th Asia-Pacific Software Engineering Conference, pages 282–289, Singapore, December 2000.

[6] A. Cox and C. Clarke. Syntactic approximation using iterative lexical analysis. In *International Workshop on Program Comprehension (IWPC), Portland, Oregon*, May 2003.

[7] D. Dubé and M. Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.

[8] L. J. Dyadkin. Multibox parsers. *ACM SIGSOFT Software Engineering Notes*, 19(3):23–25, 1994.

[9] V.-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[10] S. M. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, August 1991.

[11] S. M. Kearns. TLex. *Software - Practice and Experience*, 21(8):805–821, August 1991.

[12] R. Koppler. A systematic approach to fuzzy parsing. *Software - Practice and Experience*, 26(6):637–649, June 1997.

[13] D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, Nov. 1995.

[14] M. Latendresse. The RegReg package, version 1.1, and online documentation. http://www.metnet.navy.mil/~latendre/.

[15] M. Latendresse. Fast symbolic evaluation of C/C++ preprocessing using conditional values. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 170–179, March 2003.

[16] V. Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, pages 181–187. IEEE, Sept. 2000.

[17] V. Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2001.

[18] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society Press, 2001.

[19] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society Press, June 2002.

[20] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proc. of Working Conference on Reverse Engineering (WCRE)*, pages 170–178, 2002.

[21] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[22] M. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proc. Sixth International Workshop on Program Comprehension*, pages 108–117, 1998.

[23] M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In *Computational Complexity*, pages 365–370, 2001.

[24] E. Verhoeven. COBOL island grammars in SDF. Master's thesis, Informatics Institute, University of Amsterdam, 2000.