Intelligent Control and Automation, **2013, 4, 309-312**
http://dx.doi.org/10.4236/ica.2013.43036 Published Online August 2013 (http://www.scirp.org/journal/ica)

An Automata-Based Approach to Pattern Matching

Ali Sever
Pfeiffer University, Misenheimer, USA
Email: ali.sever@pfeiffer.edu

Received January 21, 2013; revised March 4, 2013; accepted March 11, 2013

ABSTRACT

Due to its importance in security, syntax analysis has found usage in many high-level programming languages. The Lisp language has its share of operations for evaluating regular expressions, but native parsing of Lisp code in this way is unsupported. Matching on lists requires a significantly more complicated model, with a different programmatic approach than that of string matching. This work presents a new automata-based approach centered on a set of functions and macros for identifying sequences of Lisp S-expressions using finite tree automata. The objective is to test that a given list is an element of a given tree language. We use a macro that takes a grammar and generates a function that reads off the leaves of a tree and tries to parse them as a string in a context-free language. The experimental results indicate that this approach is a viable tool for parsing Lisp lists and expressions in the abstract interpretation framework.

Keywords: Computation and Automata Theory; Pattern Matching; Regular Languages

1. Introduction

There have been many studies on how to generate a tree parser over Lisp lists, so that the existence of a list as an element of a regular language can be determined through equality checks [1]. Matching on lists using Automate-based approach requires a significantly more complicated model, with a different programmatic approach than that of string matching which is studied in [2]. We present that one must abandon the use of regular expressions, for list pattern matching, in favor of tree parsers. Regular expressions are convenient in string matching because of the relative simplicity and predictability of string structures and their straightforward representation [3,4]. In the realm of trees, the closest we come to this is in comparisons at tree nodes. Rather, to conclude that a tree is a member of a regular tree language, we must represent it in a useful format for our purposes or else require each "cons" cell in the tree to have its own production.

Another requirement of this approach is to provide a certain malleability in the interface—the ability for a programmer to define his or her own transition rules (or rewrite rules) on-the-fly—in the make-tree-matcher **Figure 1** function call, for example. We made sure the code would run on Common Lisp as well as on different Lisp implementations. This required some bending of the constraints of Lisp, like redefinition of Boolean symbols out of utility, and working around the organization of nested lists.

The inability to use regular expressions in a simple format was perhaps the biggest hurdle. We will use standard definitions, but we give some of these for convenience of the reader.

- **Analytic grammar—**A set of rules for parsing and returning of truth values confirming a string as either consistent or inconsistent with the rules of a formal language.
- **Context-free grammar—**A set of rules over an alphabet, defined by a quad-tuple G = (V_t, V_n, P, S), where
  P is a set of **production rules**;
  V_n is a set of **non-terminals**;
  V_t is a set of **terminals**; and
  S is a starting non-terminal and an element of V_n.
- **Context-free language—**All strings which can be generated by a context-free grammar.
- **Finite automaton—**A set of states and transitions, commonly expressed in a flow diagram. Also called a **finite state machine**.
- **Finite state machine:** a model of computation composed of states, a transition function, and an input alphabet.
- **Transition function:** describes a condition that would need to be fulfilled to enable the transition.
- **input alphabet:** input recognized by the finite state machine
- **Formal grammar—**A description of a set of rules for a given alphabet over which a set of finite strings can

```
(defparameter *falsehood* '*falsehood*)

(defun nonterminal? (x)
   (and (symbolp x)
         (let* ((str (symbol-name x))
                (fr (char str 0))
                (ls (char str (- (length str) 1))))
           (and
            (char= fr #\<)
            (char= ls #\>)))))


(defun yield (tree)
   (cond
    ((null tree) nil)
    ((atom tree) (list tree))
    (t (append (yield (car tree)) (yield (cdr tree))))))

;; Use a special symbol rather than nil for false.
;(defmacro n-and (&rest conjuncts)
; `(and ,@(mapcar #'(lambda (x) `(if (eq ,x *falsehood*) nil ,x))
conjuncts) t))

(defun n-and (&rest conjuncts)
  (if (eq (car conjuncts) *falsehood*)
    *falsehood*
    (if (eq (cdr conjuncts) nil)
      (car conjuncts)
      (apply #'n-and (cdr conjuncts)))))


;(defmacro n-or (&rest disjuncts)
; `(or ,@(mapcar #'(lambda (x) `(if (eq ,x *falsehood*) nil ,x)) dis-
juncts) nil))

(defmacro n-not-null (v)
  (if (null v)
    *falsehood*
    v))


(defun n-or (&rest disjuncts)
  (if (null disjuncts) *falsehood*
    (if (eq (car disjuncts) *falsehood*)
      (apply #'n-or (cdr disjuncts))
      (car disjuncts))))

(defmacro make-tree-matcher (name start &body rules)
   (labels
        ((gencall (right)
              (format t "generating expansion for ~A~%" right)
              `(if (or (null y) (eq *falsehood* y)) *falsehood*
                 (let ((l (copy-list y)))
                    (if (not (eq *falsehood* (n-and ,@(mapcar
#'(lambda (x)
                                                          (if (nonter-
minal? x)
                                                              `(if
(null l) *falsehood*
                                                                   (setf l (,x
l)))
                                                              `(if
(null l) *falsehood*
                                                                   (if (eql
(quote ,x) (pop l))
                                                                       l

*falsehood*))))
                                                          right))))
                        l
                        *falsehood*))))

         (genproduction (p)
                        (let ((name (car p))
                              (rules (cdr p)))
                          (format t "Genproduction:~%  name is:
~A~%  rules is/are: ~A~%~%" name rules)
                          `(,name (y)
                                  (n-or ,@(mapcar #'gencall rules))))))
          (format t "Defining function named ~a~%" name)
         `(defun ,name (y)
         (labels ,(mapcar #'genproduction (car rules))
           (not (eq (,start (yield y)) *falsehood*))))))

(defmacro tst ()
  (make-tree-matcher plusone <plus>
   ((<plus> (+ <num>))
    (<num> (1 <num>) (1)))))

(defun ld () (load "tyield.lisp"))

(make-tree-matcher booleval <true>
   ((<true> (and <true> <true>)
            (not <false>)
            (or <false> <true>)
            (or <true> <false>)
            (or <true> <true>)
            (1))
    (<false> (and <false> <false>)
             (and <true> <false>)
             (and <false> <true>)
             (or <false> <false>)
             (not <true>)
             (0))))

(defun add-match-tracers ()
   (trace n-and)
   (trace n-or))
```

Figure 1. The make-tree-matcher function.

be defined.

- **Kleene closure**/**Kleene star—**The set of all possible combinations of non-terminals of a regular language. Specifically, the superset of a set of strings containing the empty string ε and closed on the string concatenation function. Every string that is part of a regular language can be found in its Kleene expansion.
- **Parse tree—**A description of the syntax of a string within a formal grammar.
- **Parser—**A method or algorithm or its implementation which examines the application of a given string within an **analytic grammar**.
- **Regular tree language—**The set of trees accepted by a finite tree automaton.
- **Terminal—**A constant, indivisible value that cannot be further reduced to a more simplified form within its own grammar.
- **Tree automaton—**While finite automata typically act on strings, tree automata are used for tree expressions.
- **Yield—**The string pattern formed from a tree's leaves as encountered in an ordered traversal.

2. Analysis

Finite tree automata are much more difficult to implement than finite automata, and regular tree languages do not have a nice compact notation like regular (string) languages do. Instead of reading only one next symbol, finite state machines that are used to recognize regular expressions can read any finite number of next symbols. Each of these next symbols can have any finite number of next states. Since parse trees are not generally unique, we do not know anything about the structure of the tree when we decide it's a member of the regular tree language by parsing its yield. We know it's a parse tree for a string in the grammar, but there can be more than one, and we don't know which it is.

Although our initial vision involved the usage of regular expressions, which would be appropriate for normal strings [5,6], the nested form of S-expressions required a more iterative and comprehensive method. Traversing a Lisp list involves the usage of a parse tree. Through this application, one can back trace the sequence of an expression through the parent nodes that generated each step [7].

Generalized nondeterministic finite automata can be defined as the 5-tuple $(S, \Sigma, \delta, s, a)$, where:

S = A complete set of states;

Σ = A finite alphabet;

δ = A transition function (δ: $(S - \{a\}) \times (S - \{s\}) \rightarrow R$);

s = A start state; and

a = A accept state;

where R is the set of all regular expressions over Σ ("Automata").

This can be similarly migrated to tree parsing. Recognizing trees as elements of a regular language only allows us to say that the input is part of the language, or set of all possible elements of the regular grammar. This is done through Boolean comparisons in a top-down tree automaton context. Such automata can be represented with the four-tuple (Q, F, Qf, Δ), where:

Q = A set of states;

F = A ranked alphabet;

Qf = A subset of terminal states in Q; and

Δ = A set of transition rules [8,9].

As our implementation uses a nondeterministic pushdown approach, we can only test for the presence of a list in a regular tree language. As the language can have any number of similarly organized tree structures, we cannot tell with our algorithm which one of them has been found—only that the pattern in question is present in the language.

Using a finite tree automaton to match a Lisp list would require every cons cell in the pattern to have its own production, all labeled "cons".

Theorem 1. A regular tree language is the set of parse trees for a context-free grammar [10].

We parse the yield of the tree to show that it's a member of the context-free language, and use the theorem 1 to conclude that it's a member of the regular tree language. This does not provide us any information about the structure of the original tree, but we do know whether it is a member of the tree language we defined. If it is a member, then we know that it matches the pattern we are looking for.

3. Experiments

Parsing the string allowed us to test whether it was a member of a given string language. A language is a set of sentences (strings and trees are "sentences" in the sense that we use them in formal language theory). A context-free language is the set of all strings that can be generated by its grammar. We also know because of the theorem that there is a regular tree language composed of all its parse trees.

We test whether that tree is in the regular tree language by testing whether the string is a parse tree of the context-free language.

Our tree pattern matching implementation allows for parsing of a tree structure as a string for subsequent comparison. By seeing that a given list is a parse tree accepted by a finite tree automaton, we know, by the general theorem which states "A regular tree language is the set of parse trees for a context-free grammar". That means the list in question is a member of the regular language tested by the automaton. The use of tree yield functions simplified this step. Conceptually, trees consist of parent nodes, where child nodes extend from the right leaf of each node, with the outermost levels on the left side of the graph. Our custom yield function generates usable strings from tree inputs. With the make-tree-matcher function, the yield of a given tree is produced, and recursive-descent parsing is performed on the value by the same function **Figure 1**.

We experimented and provided the results with the make-tree-matcher function in **Figures 2** and **3**. Notice that the BOOLEVALUATION function is the language of all true Boolean expressions without variables using efficient implementations of automata operations.

From the definition of a context-free language, we know that a regular tree language is the set of all parse trees of this language's grammar. Therefore, a tree is in the regular tree language if its yield is in the context-free language.

4. Conclusion

We proposed a symbolic approach for pattern matching on LISP programs. We use a symbolic automata representation and implement set of functions and macros for identifying sequences of Lisp S-expressions using finite


```
(make-tree-matcher boolevaluation <true>
  ((<true> (and <true> <true>)
           (not <false>)
           (or <false> <true>)
           (or <true> <false>)
           (or <true> <true>)
           (1))
   (<false> (and <false> <false>)
            (and <true> <false>)
            (and <false> <true>)
            (or <false> <false>)
            (not <true>)
            (0))))
```

Figure 2. The make-tree-matcher implementation of automata.

```
Defining function named BOOLEVALUATION
Genproduction:
    name is: <TRUE>
    rules is/are: ((AND <TRUE> <TRUE>) (NOT <FALSE>) (OR <FALSE> <TRUE>) (OR <TRUE> <FALSE>) (OR <TRUE> <TRUE>) (1))
generating expansion for (AND <TRUE> <TRUE>)
generating expansion for (NOT <FALSE>)
generating expansion for (OR <FALSE> <TRUE>)
generating expansion for (OR <TRUE> <FALSE>)
generating expansion for (OR <TRUE> <TRUE>)
generating expansion for (1)
Genproduction:
    name is: <FALSE>
    rules is/are: ((AND <FALSE> <FALSE>) (AND <TRUE> <FALSE>) (AND <FALSE> <TRUE>) (OR <FALSE> <FALSE>) (NOT <TRUE>) (0))
generating expansion for (AND <FALSE> <FALSE>)
generating expansion for (AND <TRUE> <FALSE>)
generating expansion for (AND <FALSE> <TRUE>)
generating expansion for (OR <FALSE> <FALSE>)
generating expansion for (NOT <TRUE>)
generating expansion for (0)
BOOLEVALUATION

This is the language of all true Boolean expressions (without variables).
      > (boolevaluation '(or (not 1) (or 1 0)))
      T
      > (boolevaluation '(0))
      NIL
      > (boolevaluation '(1))
      T
      > (boolevaluation '(or 0 1))
      T
      > (boolevaluation '(not 0))
      T
      > (boolevaluation '(and 1 (not 1)))
      NIL
```

Figure 3. Sample run of "boolevaluation" function.

tree automata. Our experiments demonstrate that the study has produced a viable tool for parsing Lisp lists and expressions, and because of the language used, has the added bonuses of portability, small size, customizability, and clarity. The study of formal grammar and regular expressions has shown us with those topics the utility, robustness, and sometimes elegance of regular languages and Lisp. The same approach can also be applied to variety of other functional programming languages. Finally, the use of automata as a symbolic representation for verification has been investigated in other contexts (e.g., [9]). Therefore, to obtain these kind of particular but interesting results are of substantial and growing interest for many applied problems in symbolic computations [10].

REFERENCES

[1] M. Sipser, "Theory of Computation," 3rd Edition, Course Technology, 2012.

[2] M. Bojańczyk and T. Colcombet, "Tree-Walking Automata Cannot Be Determinized," *Theoretical Computer Science*, Vol. 350, No. 2-3, 2006, pp. 164-170. doi:10.1016/j.tcs.2005.10.031

[3] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi, "Tree Automata Techniques and Applications," 2007.

[4] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi, "Tree Automata Techniques and Applications II," 2007.

[5] C.-H. Chen, "A Neural Network Arhitecture for Syntax Analysis," *IEEE Transactions of Neural Networks*, Vol. 10, No. 1, 1999, pp. 94-114. doi:10.1109/72.737497

[6] J. Power, "Notes on Formal Language Theory and Parsing," National University of Ireland, Maynooth, Kildare, 2002.

[7] I. Bagrak and O. Shivers, "trx: Regular-Tree Expressions, Now in Scheme," *Scheme Workshop*, September 2004.

[8] F. Yu, *et al.*, "Symbolic String Verification," *SPIN*'08 *Proceedings of the* 15*th International Workshop on Model Checking Software*, pp. 306-324.

[9] A. Bouajjani, B. Johnson, M. Nilsson and T. Touili, "Regular Model Checking," *Proceedings of the* 12*th International Conference on Computer Aided Verification*, 2007, pp. 403-418.

[10] L. Segoufin and V. Vianu, "Validating Streaming XML Documents," *ACM*, 2002, pp. 53-64.

