Extending Regular Expressions

Steven Michael Kearns

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
1990
ABSTRACT

Extending Regular Expressions

Steven Michael Kearns

Regular expressions are used in many applications to specify patterns because they can be compiled into very efficient one-pass pattern matchers; at the same time, they can specify a wide range of patterns of practical interest. The thesis extends regular expressions in 4 ways. First, we give a simple, efficient algorithm for matching an extended regular expression language. The language includes a number of new operators for expressing patterns, including operators that specify the context of a match, and operators that can require arbitrary lookahead. The new operators make it feasible to specify many more patterns than practical before, but the new matching algorithm retains the efficiency of one-pass pattern matching. Second, we show how to extract a parse from a successful match, which provides significantly more information from a match than possible before. Third, we describe practical optimizations for efficiently implementing the matching and parse extraction algorithms. Fourth, we demonstrate how the new pattern matching capabilities can be integrated with an existing programming language, by describing a program generator for pattern matching applications. The program generator is called TLex. We describe the results of its use in several real-life applications.
# Table of Contents

List of Figures.................................................................................................................. iv  
List of Tables ...................................................................................................................... v  
Acknowledgements ........................................................................................................... vi  
1. Introduction .................................................................................................................... 1  
2. Motivation ...................................................................................................................... 4  
   2.1. A Quick Introduction .......................................................................................... 4  
   2.2. Why Extend Regular Expressions? ................................................................. 7  
   2.3. Why Restrict Regular Expressions? ............................................................... 10  
   2.4. Parse Extraction ............................................................................................... 12  
3. Previous Work ............................................................................................................... 14  
   3.1. Matching Restricted Regular Expressions .................................................... 14  
   3.2. Matching Extended Regular Expressions ..................................................... 15  
   3.3. VLSI and Regular Expressions ....................................................................... 17  
   3.4. Other Extended Languages ............................................................................ 17  
   3.5. Other Extensions to Regular Expressions ................................................... 21  
   3.6. Parse Extraction ............................................................................................... 23  
   3.7. Languages Integrating Pattern Matching ...................................................... 25  
   3.8. Tools Using Regular Expressions ................................................................. 29  
4. TLex ............................................................................................................................. 31  
   4.1. Why Another Language? .................................................................................. 31  
   4.2. Overview of TLex .............................................................................................. 33  
   4.3. The TLex Pattern Language .......................................................................... 36  
   4.4. Rules and Parse Trees ...................................................................................... 44
8.1. Specific Contributions of the Thesis ........................................... 92
8.2. Future Research ........................................................................ 93

Bibliography ...................................................................................... 96

Appendix A. The Matching Formalism .............................................. 99
   A.1. Formally Defining Patterns ...................................................... 99
   A.2. Defining More Patterns ......................................................... 102
   A.3. Simplifying the Problem .......................................................... 105
   A.4. Prefix() and Postfix() ............................................................. 114

Appendix B. A Formal Discussion of Next() ...................................... 119
   B.1. Correspondence ..................................................................... 119
   B.2. Correctness of SEQ ............................................................... 125
   B.3. Correctness of * ................................................................. 126
   B.4. Correctness of DEFCALL ..................................................... 128
   B.5. Correctness of [x] ................................................................. 129
   B.6. Correctness of SET_LONGER ............................................... 130
   B.7. Correctness of SET_LONGEST ............................................. 135
   B.8. Worst-Case Complexity of Next() ....................................... 137
   B.9. Optimizing Next() ............................................................... 140
   B.10. Listing of Optimized Next() ................................................ 154

Appendix C. A Formal Discussion of Parse Extraction ...................... 159
   C.1. Defining the Problem Formally ............................................. 159
   C.2. Solving the Problem ............................................................ 160
   C.3. Solving the Problem Again ................................................... 165
   C.4. Optimizing Parse() ............................................................ 172
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.5. Proof of the Complexity of Parse()</td>
<td>178</td>
</tr>
<tr>
<td>Appendix D. Details of TLex</td>
<td>183</td>
</tr>
<tr>
<td>Appendix E. WC2 in TLex</td>
<td>194</td>
</tr>
<tr>
<td>Appendix F. The BBall Application</td>
<td>198</td>
</tr>
<tr>
<td>Appendix G. Sample BBall Input</td>
<td>211</td>
</tr>
<tr>
<td>Appendix H. Benchmark Programs</td>
<td>216</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1: A DFA</td>
<td>5</td>
</tr>
<tr>
<td>4-1: A Simple TLex Input File</td>
<td>34</td>
</tr>
<tr>
<td>4-2: TLex Overview</td>
<td>35</td>
</tr>
<tr>
<td>4-3: WC2 Performance</td>
<td>50</td>
</tr>
<tr>
<td>4-4: BBall Timing</td>
<td>53</td>
</tr>
<tr>
<td>4-5: Fast-BBall Timing</td>
<td>54</td>
</tr>
<tr>
<td>8-1: Locating TLex</td>
<td>91</td>
</tr>
<tr>
<td>A-1: A Pattern</td>
<td>110</td>
</tr>
<tr>
<td>A-2: Compacting a Pattern</td>
<td>111</td>
</tr>
<tr>
<td>A-3: Representing Definitions</td>
<td>112</td>
</tr>
<tr>
<td>A-4: Representing Recursion</td>
<td>113</td>
</tr>
<tr>
<td>B-1: Two States</td>
<td>144</td>
</tr>
<tr>
<td>B-2: Two States Compacted</td>
<td>145</td>
</tr>
<tr>
<td>D-1: Text Positions</td>
<td>184</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1:  TLex Pattern Language Summary</td>
<td>38</td>
</tr>
<tr>
<td>4-2:  Behavior of BBall</td>
<td>52</td>
</tr>
<tr>
<td>6-1:  Tested Patterns</td>
<td>76</td>
</tr>
<tr>
<td>6-2:  Matching Using Lazy Evaluation</td>
<td>77</td>
</tr>
<tr>
<td>6-3:  Matching Using a DFA</td>
<td>78</td>
</tr>
<tr>
<td>7-1:  Testing Parse()</td>
<td>89</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank all of those people who helped me complete this thesis. Foremost among these is my adviser, Gerald Q. Maguire Jr.. He offered sound advice, occasional prodding, and unflagging encouragement. Thanks also go to the other members of my thesis committee and thesis proposal committee: Michael Foster, Zvi Galil, Sal Stolfo, Mike Lesk, Thomas Stern and Fletcher Griffis. A number of students at Columbia helped make the PhD process a fun one: David Kurlander, Tony Weida, George Wolberg, Ari Gross, Tom Ellman, Nat Polish, Jonathan Smith, Peter Karp, and my great officemates, Shu-Wie Chen and John Ioannidis. Special thanks are due the people at HLH who really made TLex happen: Seth Haberman and George Lilly, who set me on the path resulting in this thesis; Scot Shinderman, who helped implement the earliest version of TLex; and Wayne Berke and Isaac Dimitrovsky, the first, brave users of TLex and the source of numerous suggestions. Wayne Berke was the author of the BBall application discussed in the thesis.

The impact of the thesis committee on the final form of this thesis must be acknowledged. The high level, intuitive presentation of chapters 5, 6, and 7 was motivated by the thesis committee, who insisted on this format. They deserve all credit/blame that results.
1. Introduction

Regular expressions are an elegant way to specify patterns. Regular expressions are important because they can be compiled into very fast, one-pass pattern matchers, no matter how complicated the regular expression. A language defined by a regular expression is called a regular language. Regular languages are considered very well understood, from a theoretical perspective.

Myriad areas of computer science make use of regular expressions. The most popular use is in text searching and parsing. Other areas include VLSI and operating systems, for specifying the sharing of resources; and protocol analysis, for specifying and verifying protocol behavior. Specific examples will be presented later.

Unfortunately, most all applications use “restricted regular expressions” as the pattern language because there are a number of simple, efficient algorithms for matching them. Adding additional operators to the restricted regular expression language results in an “extended regular expression” language. The goal is to make it easier to write patterns while retaining the efficient pattern matching of restricted regular expressions. Any extended regular expression can be rewritten as an equivalent restricted regular expression, theoretically. However, the equivalent restricted regular expression may be significantly larger and more complicated [Abrahamson 1987].
And in practice, it is quite easy to write an extended regular expression which is too hard for any human to convert into an
1. Introduction

Equivalent restricted regular expression. Thus, an extended regular expression language would be preferable to the restricted regular expression language if there was a simple, efficient algorithm for matching the former.

Because regular expressions are so fundamental in computer science, an improvement in the use of regular expressions can have wide ranging impact. This thesis is concerned with extending regular expressions in several principal ways.

First, we give a simple recursive algorithm for matching an extended regular expression pattern. The additional operators provided in the extended regular expression language include operators that specify a class of context-sensitive patterns, as well as operators which require arbitrary lookahead. The extended language makes it feasible to specify many more patterns than practical before, but the new matching algorithm retains the efficiency of one-pass pattern matching.

Second, we show how to extract a parse from a successful match, which provides significantly more information from a match than possible before.

Third, we describe practical optimizations for implementing efficient versions of the matching and parse extraction algorithms.

Fourth, we demonstrate how the new pattern matching capabilities can be integrated with an existing programming language, by describing a program generator for pattern matching applications.
1. Introduction

The program generator is called TLex. It has been implemented, and the results of its use in several real-life applications are described.

The thesis is organized into 8 chapters plus a number of appendices. Chapter 2 motivates the need for an extended regular expression language. Chapter 3 discusses previous work. Chapter 4 introduces the TLex language from the user’s perspective, to provide an overview of how the innovations of the thesis can be practically employed. Chapter 5 discusses the overall architecture of the pattern matching and parse extraction algorithms. Chapter 6 presents the pattern matching algorithm, along with a correctness argument and complexity analysis. Chapter 7 contains the same for the parse extraction algorithm. Chapter 8 concludes the thesis and suggests avenues for future research. Appendices A, B, C, and D contain much greater detail and formalism about the matching formalism, the matching algorithm, the parsing algorithm, and the TLex language, respectively.
2. Motivation

This chapter gives a quick overview of restricted and extended regular expressions, as well as deterministic finite automata. Then it motivates the need for the additional operators of an extended regular expression language. Finally, it demonstrates the utility of parse extraction.

2.1. A Quick Introduction

Regular expressions will be described later in great detail; for now we give a quick introduction. A restricted regular expression is a pattern recursively composed from strings and the pattern operators SEQ, *, +, OR, NOTCLASS, and DEFINE. Text in quotes (“hi there”), or single characters, are strings of characters that must be matched exactly. A parenthesized expression like (SEQ p1 p2) has a pattern operator, SEQ, in the first position and parameters in successive positions. (SEQ p1 p2) matches any string formed by concatenating a string matching p1 with a string matching p2. (OR p1 p2) matches any string matching p1 or p2. (* p1) matches 0 or more consecutive occurrences of strings matching p1. (+ p1) matches 1 or more consecutive occurrences. (NOTCLASS c1 c2) matches any character except those matching c1 or c2. (DEFINE <name> p1) defines NAME as shorthand for the pattern p1, but recursion is not allowed in a restricted regular expression.

Extended regular expressions, as defined in this thesis, permit even more pattern operators. (AND p1 p2) matches any string matching p1
2. Motivation

and p2. (ALL p1 MINUS p2) matches any string which matches p1 but
not p2. DEFINE patterns may be recursive (though they are
implemented using bounded recursion, which is discussed in section
4.3.2). (CSRP letter1 p1 pright) matches any string matching p1, but
only when a prefix of the string to the right matches pright and the
character to the left matches letter1. CSRPN, CTRP, and CTRPN are
variants which will be detailed later. If p1 matches 2 or more strings
of different lengths, starting at the same spot in the input string, then
(SHORTEST p1) matches only the shortest such string, while
(LONGEST p1) matches only the longest. (SHORTER p1) and
(LONGER p1) match exactly what p1 does, but prefer the shorter or
longer match. Finally, we will also use a macro definition such as
(MACRO MYMAC (SEQ “d” $1)). To use this macro, write
(MYMAC (OR a b)); this gets expanded to the body of MYMAC, and $1
gets replaced by the parameter to MYMAC to produce
(SEQ “d” (OR a b)).

A deterministic finite automata (DFA) is illustrated here:

Figure 2-1: A DFA

The circles are states of the DFA. The start state is marked with an
asterisk, the accepting states have two concentric circles. The arrows
between states are called transitions. Each state has exactly 1
transition leaving it for each possible character in the input alphabet.
However, we have only drawn the interesting transitions above. All
the missing transitions go to a “death state” from which the accepting state can never be reached.

The DFA begins in the start state; then it reads the next input character and takes the transition with the corresponding label. At the end of the input the DFA reports SUCCESS if it is in the accepting state, and FAILURE otherwise. In order to reach the accepting state, the DFA above must read an “f”, followed by an even number of “o”, followed by “d”. We can describe this DFA by listing the strings it matches, but it is an infinite list: “food”, “fooood”, “fooooood”, ...

Another way to describe what it accepts is with this regular expression:

\[(SEQ \text{"f"} \ (+ \text{"oo"}) \ "d")\]

Every DFA can be described by a regular expression, and every regular expression can be converted into a DFA. However, sometimes the conversion can be very tricky. For example, the pattern

\[(SEQ \ (* \ ANY) \ b \ (OR \ a \ b) \ (OR \ a \ b) \ (OR \ a \ b) \ (OR \ a \ b) )\]

requires at least 15 states when converted into a DFA. See [Hopcroft and Ullman 1979] for additional examples and the full theory of regular expressions and finite automata.

It is easy to see how a pattern encoded as a DFA can be searched for efficiently. The computer can efficiently simulate a DFA by precompiling a table mapping from the current state and input character to the next state. Then each input character can be
processed by a simple table lookup. Normally, patterns are most easily specified as regular expressions; thus they must go through a compilation process to be converted into a DFA.

2.2. Why Extend Regular Expressions?

Restricted regular expressions are theoretically capable of specifying anything an extended regular expression can; in practice, a pattern language including extended operators substantially increases the number of patterns that can be specified by programmers. In addition, using more powerful operators means all patterns are more natural to specify, easier to understand, and usually shorter. Persuasive arguments for a powerful pattern language can be found in [Lipson 1984]. We will demonstrate the advantage with several examples.

2.2.1. Matching A C-comment

A C-comment is a string that starts with “/*” and continues until the first occurrence of “*/”. A straightforward attempt to write a regular expression matching a C-comment looks like this:

\begin{verbatim}
(define c-comment (seq "/*" anystr "*/")
\end{verbatim}

This definition matches all C-comments; unfortunately, it also matches some strings which are not C-comments! The challenge is writing a definition which matches “/* hi there */” but not “/* hi there */ /* more text */”. In the pattern language defined in this thesis, the previous definition will prefer to match legal C-comments because anystr prefers to match the smallest possible string; however,
if the definition is used as a subpattern of another pattern it may
choose to match an illegal C-comment. For example, the pattern

\[(SEQ \text{C-comment} \ "!!!")\]

will match the text "/*comment*/ /*comment*/!!" which is probably
wrong.

A restricted regular expression for correctly matching a C-comment
looks like this:

\[
(\text{DEFINE C-comment} (SEQ \"/**\"
  (* (SEQ (* (\text{NOTCLASS} \"*\")
        (+ \"*\")
        (\text{NOTCLASS} \"*\" \"/\")
      )
    )
  )
)
\]

The restricted regular expression for a C-comment is nearly impossible
to read and challenging to derive. In contrast, the extended regular
expression for a C-comment is relatively straightforward. We can use
the following macro:

\[
(\text{MACRO DoesNotHave} (\text{ALL ANYSTR}
\text{MINUS (SEQ ANYSTR }$1\text{ ANYSTR)})
)
\]

(DoesNotHave $1) matches any string that does not contain a
substring matching the pattern substituted for $1. Now defining a
C-comment is easy, because a C-comment is any string that starts with
a "/*", ends with a "*/", and lacks a "*/" in the middle.:

\[
(\text{DEFINE C-comment} (SEQ \"/**\"
  (\text{DoesNotHave} \"*/\")
  \"*/\")
)
\]

An even more readable definition for a C-comment makes use of the
\text{SHORTEST} pattern:

\[
(\text{DEFINE C-comment} (\text{SHORTEST (SEQ \"/** ANYSTR \"*/\")})
)
2. Motivation

This example demonstrates two of the extended operators: SHORTEST and ALL/MINUS.

2.2.2. Matching A Line of All Vowels

Another extended operator is (AND p1 p2), which matches whatever matches both p1 and p2. It can be indispensable for reusing patterns. Pretend a pattern library contains two definitions: Line, a pattern matching any line of a file; and AllVowels, a pattern matching strings containing only vowels. To match a line of a file which contains only vowels using a restricted regular expression, the pattern library would be useless. A totally new pattern would have to be written. Using restricted regular expressions, there is no direct way to reuse the Line and AllVowel patterns to match the combination. In contrast, if the AND pattern is available, one can simply write the pattern

\[(\text{AND Line AllVowels})\]

to solve the problem.

2.2.3. Matching a Number

[Lesk and Schmidt 1986] points out that most patterns are context sensitive, and that usually the LONGEST match is the most useful. This thesis is the first to realize that the LONGEST and context-sensitive-right patterns can be implemented as general regular expression operators. The thesis gives an efficient implementation of the general pattern operators LONGEST, CTRP, CTRPN, CSRP, and CSRPN. The latter four provide context sensitive capabilities. The pattern (CSRPN \text{letter1 p pright}), for example, matches strings
2. **Motivation**

matching p, as long as no prefix of the string to the right matches pright, and the character to the left does not match letter1.

We give a simple example of the use of LONGEST and CSRPN. An obvious definition for matching a number is (+ Digit), matching 1 or more digits. This definition can be troublesome because when fed the input string “1234 balloons” it can match “1”, “2”, “3”, “4”, “12”, “23”, “234”, etc... in addition to the one we want, “1234”. Really, a number is 1 or more digits without a digit to the left or right. **Restricted regular expressions have absolutely no way to specify such a pattern**, because they have no concept of the context of a match!

However, with the extended regular expression language one can express the right context sensitivity using CSRPN or LONGEST. (LONGEST (+ Digit)) extends a match of 1 or more digits as long as possible. In the input “1234 balloons” it would match “1234”, “234”, “34”, and “4”. The pattern (CSRPN Digit (+ Digit) Digit) matches 1 or more digits as long as there is no Digit to the left or right. In the input “1234 balloons” it would match only “1234”.

### 2.3. Why Restrict Regular Expressions?

We previously pointed out that this thesis is the first realization that LONGEST and some context sensitive operators could be implemented as general regular expression operators. Surprisingly, the AND and ALL/MINUS patterns, which have long been recognized as general regular expression operators, are rarely provided in tools that use regular expressions. In fact, most tools provide only restricted regular
expressions, those that can be recursively constructed using simple strings, OR, SEQ, and *. Why are the other powerful operators excluded? We give the main arguments with refutations:

- **(Argument 1)** Any pattern that can be specified with an extended regular expression can also be specified using a restricted regular expression. (The matching algorithm presented in this thesis can be seen as a constructive proof of this assertion: it can compile an extended regular expression into a deterministic finite automata, which is equivalent in power to the language of restricted regular expressions.)

  (Refutation 1) Using more powerful operators means patterns are more natural to specify, easier to understand, and usually shorter. Consider the C-comment example in the previous section for graphic illustration. In addition, one should not confuse theoretical possibility with practical possibility. In principle, any predicate calculus theorem can be tested for consistency against a set of axioms. In reality, there is just a small class of theorems for which this is practical. A similar truth holds for regular expressions. The class of patterns which can be successfully written by programmers is much larger, in practice, when the extended regular expression language is used.

- **(Argument 2)** The extended operators are so powerful that people might specify things which are too large to compile into a deterministic finite automata. If regular expression p2 compiles into a finite automata with n states, (ALL p1 MINUS p2) may compile into a finite automata with $O(n)$ states. (AND p1 p2) may require $O(mn)$ states if p1 and p2 require m and n states respectively [Hopcroft and Ullman 1979].

  (Refutation 2) In practice this exponential and quadratic blowup is a worst case projection which rarely occurs. [Leiss 1985] explored in great detail the consequences of including AND and ALL/MINUS. He found several conditions on the structure of a pattern which slow the quadratic and exponential blowups.

Even a restricted regular expression may require too many states to be compiled. Most importantly, if two regular expressions specify the same pattern then they will require **exactly the same** number of states when compiled, even if one of the regular
2. Motivation

expressions uses extended operators. (This assumes that state
minimization [Hopcroft and Ullman 1979] is performed.)
However, the extended regular expression might be significantly
shorter and simpler than the equivalent restricted regular
expression, so it is preferred.

o (Argument 3) The usual algorithms for compiling regular
expressions do not extend easily to include the extended
operators.

(Refutation 3) This thesis presents a relatively straightforward
and efficient algorithm for matching an extended regular
expression language that includes the extended operators.

2.4. Parse Extraction

A regular expression for matching an object usually documents the
structure of the object being matched; this fact is the key insight for
getting more information out of a match. Most tools that use regular
expressions can return the string that matched a pattern. Much more
useful is knowing which subparts of the matching string correspond to
which subparts of a pattern. Knowing this correspondence is parsing
the string in a meaningful way.

A simple example demonstrates the utility of parse extraction. A
simple way to specify a decimal number is

(DEFINE DecimalNum (SEQ (+ Digit) Period (+ Digit)))

This defines a decimal number as a sequence of 1 or more digits,
followed by a period, followed by 1 or more digits. Consider matching
this pattern against the string “1234.87”. Previous tools can report “it
matched!”; with parse extraction, we can find out that “1234” matched
the first (+ Digit), “87” matched the second; “1” matched the first
iteration of the first (+ Digit), “2” matched the second iteration, etc..
This type of information is often exactly what is needed to make use of a match. In our example we can easily find the integer and decimal parts of the DecimalNum, thanks to parse extraction.

A more sophisticated example is matching the following display of basketball standings:

<table>
<thead>
<tr>
<th>PACIFIC DIV.</th>
<th>W</th>
<th>L</th>
<th>PCT</th>
<th>GB</th>
<th>HOME</th>
<th>ROAD</th>
<th>DIV</th>
<th>STREAK</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA LAKERS</td>
<td>44</td>
<td>14</td>
<td>.759</td>
<td>-</td>
<td>26-4</td>
<td>18-10</td>
<td>12-4</td>
<td>WON</td>
</tr>
<tr>
<td>PORTLAND</td>
<td>41</td>
<td>18</td>
<td>.695</td>
<td>3 1/2</td>
<td>26-4</td>
<td>15-14</td>
<td>13-6</td>
<td>WON</td>
</tr>
<tr>
<td>PHOENIX</td>
<td>39</td>
<td>19</td>
<td>.672</td>
<td>5</td>
<td>25-5</td>
<td>14-14</td>
<td>15-5</td>
<td>WON</td>
</tr>
<tr>
<td>SEATTLE</td>
<td>30</td>
<td>29</td>
<td>.508</td>
<td>14</td>
<td>21-8</td>
<td>9-21</td>
<td>5-11</td>
<td>LOST</td>
</tr>
<tr>
<td>GOLDEN STATE</td>
<td>27</td>
<td>32</td>
<td>.458</td>
<td>17</td>
<td>22-8</td>
<td>5-24</td>
<td>7-10</td>
<td>WON</td>
</tr>
<tr>
<td>LA CLIPPERS</td>
<td>24</td>
<td>36</td>
<td>.400</td>
<td>21</td>
<td>15-12</td>
<td>9-24</td>
<td>5-12</td>
<td>LOST</td>
</tr>
<tr>
<td>SACRAMENTO</td>
<td>18</td>
<td>42</td>
<td>.300</td>
<td>27</td>
<td>13-17</td>
<td>5-25</td>
<td>6-15</td>
<td>WON</td>
</tr>
</tbody>
</table>

From a pattern that matches basketball standings (see ruleset 6 of bbball.tlx, in appendix F) and a parse tree created from a match, one can easily find out how many relevant lines there are, one can access any line, and one can access the team name and any other field on a specific line. Furthermore, in a field like “GB” one can easily determine if it is of the form “-”, “5”, or “14 1/2”, information which is necessary to convert it into a standard internal form.

An important aspect of using the information from parse extraction is how a programmer accesses this information. In the description of the TLex language we describe a Parse Tree abstract data type that offers a convenient interface.
3. Previous Work

There is an extensive theoretical understanding of finite automata and regular expressions. [Hopcroft and Ullman 1979] provides a good exposition of the main theoretical results; [Aho et al. 1986] contains a readable exposition of the principal algorithms.

A restricted regular expression is a regular expression recursively composed of the primitive string patterns plus the pattern operators *, SEQ, and OR. In practice, the operators +, ?, DEFINE, CLASS, and NOTCLASS are usually offered, but these are essentially simple macros for the previous operators. The thesis uses “extended regular expressions” to refer to patterns which may include the additional operators AND, ALL/MINUS, SHORTEST...LONGER, CSRP..CTRPN, and bounded recursion. However, extended regular expressions are also used in the literature to refer to a number of languages, each with different operator mixes, some of which are not even translatable to finite automata. We use “extended regular expression” exclusively in the former sense.

3.1. Matching Restricted Regular Expressions

The most popular method for compiling regular expressions is the subset construction and its variants. The subset construction cannot handle extended regular expressions directly, but the algorithm presented here can be thought of as a descendant of the subset construction. [Thompson 1968] describes the subset construction.
3. Previous Work


[Anzai 1977] gives a matrix method for compiling restricted regular expressions. The space requirements are large, but the matrix operations are all Boolean operations; as a result, this algorithm might be appropriate for high speed hardware compilation of restricted regular expressions.

Gnu Emacs [Stallman 1987] provides a restricted regular expression search algorithm that could theoretically be extended to handle some of the extended operators described herein. It searches all possible matches of a string against a regular expression using recursive backtracking. This technique works well on small strings, but on long strings it can become hopelessly slow, especially when extended operators are involved. A simple example illustrates the pitfalls of recursive backtracking: the simple pattern

\[(SEQ\ ANYSTR\ ANYSTR\ ANYSTR\ c)\]

which matches any string ending in “c”, requires $O()$ possibilities to be explored on a string of length $n$. This translates into hours for even a single page of text. Such performance compares poorly to a finite automata based search, which requires $O(n)$ operations on a string of length $n$, requiring a fraction of a second for a page of text.

3.2. Matching Extended Regular Expressions

A recursive algorithm for compiling a regular expression (which may include the AND and ALL/MINUS operators) into a DFA is discussed
3. Previous Work

in [McNaughton and Yamada 1960] and [Hopcroft and Ullman 1979]. The method is **bottom-up** recursive compilation. For example, to compile the pattern (AND p1 p2) into a DFA, first compile p1 and p2 into DFAs, and then combine the two DFAs using a simple algorithm that examines the Cartesian product of the sets of states of p1 and p2. It has a horrible worst case complexity, and usually does much more work than necessary.

The solution to Exercise 3.23 in [Hopcroft and Ullman 1979] introduces an $O()$ dynamic programming method for matching a specific string against a regular expression which may include the AND and ALL/MINUS operators, where $n$ is the sum of the length of the input and the length of the regular expression. Unfortunately, this algorithm can be impractical for strings as small as 1000 characters because of the $O()$ complexity. The method used to implement CPDS, discussed below, is actually a variant of this dynamic programming method.

[Brzozowski 1964] gives a very elegant recursive algorithm for compiling regular expressions which may include the AND and ALL/MINUS operators. Unfortunately, it is hard to implement efficiently [Berry and Sethi 1986] and it appears unsuitable for implementing parse extraction (because it mangles the structure of the regular expression being matched).

Many people mistakenly believe that the Unix search utility egrep allows the AND and ALL/MINUS operators to be used. In fact, none of the Unix programs grep, egrep, and fgrep, allow AND and
3. Previous Work

ALL/MINUS [UNIX 1986]. Grep does not even allow restricted regular expressions (the star operator can only apply to a single character, for example). Egrep matches restricted regular expressions, i.e. without AND and ALL/MINUS. Fgrep just searches for sets of fixed strings.

The matching algorithm in the thesis is reasonably simple, offers very fast performance, and has the advantage that it can be implemented using “lazy transition evaluation” (which will be fully explained in section B.9.5). It is also straightforward to extend the algorithm to new extended operators as their utility becomes apparent. Most important, the matching algorithm in the thesis was designed to provide exactly the information needed for efficient parse extraction.

3.3. VLSI and Regular Expressions

[Foster 1984] describes recursive VLSI building blocks for simulating a fixed restricted regular expression; these logic circuits take advantage of the same recursive properties of regular expressions as the matching algorithm presented in the thesis. However, Foster matches restricted regular expressions, while the algorithm presented in the thesis matches extended regular expressions.

[Wakabayashi et al. 1984] shows how to make a PROGRAMMABLE VLSI restricted regular expression recognizer; their design is bettered in [Harao and Paisan 1987]. The former paper uses essentially the same recursive cells as Foster’s, but introduces a skip line to speed up the parallel operation of the recognizer.
3.4. Other Extended Languages

Instead of trying to implement extended regular expressions as a DFA, perhaps it would be better to choose a more powerful model of computation. For example, context free languages can do everything that regular languages can, plus recursion. Unfortunately, context free languages are not closed under intersection or complement [Hopcroft and Ullman 1979], implying that the AND and ALL/MINUS extended operators could not be implemented, in general. However, the real problem with moving to a more powerful language model is efficiency. A regular expression, after compilation into a DFA, can be matched in time proportional to the length of the input with a very small constant. The more powerful, interesting languages cannot approach this ideal, even when they are used to match a regular language. For example, we will discuss SNOBOL4 [Griswold et al. 1971], Icon [Griswold and Griswold 1983], and ROPE [Lipson 1984], three powerful pattern languages implemented using recursive backtracking. Only for simple patterns on smaller strings can recursive backtracking search compare to the speed of DFA-based search. We will discuss a powerful language, CPDS [Liu 1981], which offers $O(\cdot)$ search with a number of extended pattern operators. On small strings this algorithm may be competitive, even with a complicated pattern. However, for matching complicated patterns against long strings, the optimal efficiency of DFA-based search is crucial.
SNOBOL4 was the first general purpose language specifically designed to integrate string pattern matching [Griswold et al. 1971]. Patterns are specified procedurally, which means that a pattern is a small program complete with variables and control flow. If a pattern can match a string in several different ways, all the ways are automatically tried one after another using backtracking. Thus patterns which can possibly match in many ways can be very expensive to search for, even exponentially slow. However, the SNOBOL4 pattern language is significantly more powerful than a regular language. Its increased power is largely due to the Immediate Assignment operator. The latter allows a previous part of a match to influence the remaining match, and it supports parse extraction. For example, in SNOBOL4 one can write a pattern equivalent to the following (we translate SNOBOL4 syntax into TLex-like syntax for consistency):

\[
(\text{SEQ } (\text{BIND} (* \text{ANY}) \text{ TO } J) \ J)
\]

It matches any string whose first half is exactly the same as the second half. After matching a substring to \((* \text{ ANY})\), SNOBOL4 “binds” the substring to string variable \(J\). The second occurrence of \(J\) refers to this specific string. Of course, all possible bindings to \(J\) might be tried, one after another.

Icon is a successor to SNOBOL4. The authors of SNOBOL4 realized that SNOBOL4 contains two separate languages, one for specifying patterns and another for general purpose programming [Griswold and Hanson 1980]. Icon combines the two languages into a single language by extending the general purpose language with the automatic
3. Previous Work

backtracking feature of the pattern language. For example, functions can return a sequence of values; the next is automatically requested when the previous one fails. Thanks to this feature, the general purpose language can be used to specify patterns. Like SNOBOL4, the pattern language is significantly more powerful than a regular language. A function with arbitrary control flow and state information can be written to match a pattern. [Griswold and Griswold 1983] is the definitive reference to Icon.

K.C. Liu took a fresh look at SNOBOL4 and Icon, and decided that the problem with these languages was their procedural specification of patterns, which made them as unreadable and hard to modify as most procedural programs. CPDS is a pattern language he proposed instead, in [Liu 1981] and [Liu 1986]. CPDS patterns can have all of the restricted regular expression operators, and in addition the AND, ALL/MINUS, and Immediate Assignment (like that of SNOBOL4) operators. It provides most of the pattern matching power of SNOBOL4 with a straightforward, declarative semantics. Furthermore, instead of using recursive backtracking to search for pattern matches, as SNOBOL4 and Icon use, he implements CPDS using a worst-case $O(n)$ (time and space) dynamic programming method, where $n$ is the length of the string being matched. (It is important to realize that this complexity result assumes that the pattern size is constant. In fact, this assumption understates the algorithm’s complexity by hiding the dependency on pattern complexity in the constant term of the $O(n)$ notation.) CPDS also offers the Immediate Assignment operator, like the one SNOBOL4 has. However, if this
operator is used in a pattern, the worst case efficiency of the matching algorithm becomes polynomial.

Lipson also felt that the problem with SNOBOL4 was the procedural definition of patterns. In [Lipson 1984] he defined a pattern language called ROPE with even more operators than CPDS. ROPE offers the power of SNOBOL4 with greater expressiveness and a simple, declarative semantics. As a result, patterns are easier to write and understand, as well as being more modular. He showed how some of the features of ROPE might be implemented using recursive backtracking and some auxiliary storage, but otherwise did not actually implement the language.

The pattern language implemented in the thesis definitely follows in the tradition of CPDS and ROPE, preferring declarative pattern specification over procedural. It offers the best worst-case performance of any of these extended languages. The pattern language in the thesis is missing the true recursion and Immediate Assignment found in CPDS, but offers a number of pattern operators not found in CPDS (specifically CSRP, CSRPN, LONGEST, SHORTEST, LONGER, SHORTER). The most sophisticated pattern matching tasks require the flexibility of the SNOBOL4 and Icon languages, but there is a significant cost: procedurally specified patterns can be very hard to understand and maintain, and they can require inordinate amounts of time to match.
3. Previous Work

3.5. Other Extensions to Regular Expressions

The extended regular expression language described in this thesis is not the only conceivable one. This section describes a number of other languages built on regular expressions for a variety of purposes. The operators implemented in the thesis were chosen because they seemed the most general; however, specialized applications may demand specialized languages.

[Lipson 1984] presents a pattern language, ROPE, which is decidedly more powerful than that of regular expressions, but does not suggest a way of efficiently implementing it. ROPE includes variants of all the extended operators presented in the thesis; in addition, there is a permutation operator for matching a permutation of the subpatterns; Immediate Assignment (discussed previously); and a few more operators for preferring some matches over others. This thesis can be seen as identifying and efficiently implementing a significant subset of the Lipson operators.

[Abrahamson 1987] examines regular expressions extended with “gotos” and Booleans. A regular expression with these features can still be compiled into a finite automata. He also suggests that the expressiveness of an operator can be measured by comparing the sizes of two patterns that match the same thing, where the first pattern uses the extended operator and the second does not. He shows that a regular expression with “gotos” and Booleans could be much shorter than an equivalent one without.
3. Previous Work

[Shaw 1980] is an excellent survey of the use of regular expressions for specifying concurrent systems. Path expressions, event expressions, and flow expressions are just three of the variants that are produced by extending regular expressions with various operators. The most important extension is the shuffle operator. “pat1 shuffle pat2” matches any string which can be made by interleaving a string matching pat1 and a string matching pat2. The shuffle closure operator takes a pattern p1 as an argument and matches any string which can be made by interleaving 0 or more strings matching p1. Regular expressions extended by the shuffle operator are still regular, but adding the shuffle closure operator makes the resulting language more powerful than that of regular expressions. Shaw introduces a number of other synchronization operators, several of which form a language as powerful as a Turing machine when combined with regular expressions. See [Shaw 1980] for more details.

[Holzmann 1982] discusses a formalism based on regular expressions for verifying properties of protocols. He invents several new operators for this purpose, including a variant of the shuffle operator discussed previously. These operators are too powerful, in general, to explore automatically without causing an exponential blowup in space and time. Thus the theory relies on symbolic manipulation of the extended regular expressions to verify the protocols.

[Brazma and Kinber 1986] discusses the “generalized regular expression”, which includes the natural numbers and addition. The
resulting language is more powerful than regular expressions; it is used for deducing programs from examples.

[Suzuki et al. 1986] explores the consequence of allowing regular expressions to iterate a countably infinite number of times. With this theoretical addition they define operators similar to those in temporal logic, for example an “eventually” operator.

3.6. Parse Extraction

Parse extraction finds out which parts of a subpattern match which substrings of the matched string. The usefulness of this type of information has been noted many times before. In context free parsing, which is often used in compiling computer languages, the main goal of matching is to build a parse tree which encodes this information. However, parse extraction has been largely ignored for regular expressions. Until this thesis there were no algorithms for extracting a parse from a match found with a DFA based search. (One explanation might be that restricted regular expressions are limited to expressing simple patterns in applications rarely requiring parse information. Hopefully, this thesis also removes that limitation.)

In the realm of context free parsing, parse extraction is considered the main goal of matching. For example, if a grammar for ADA does not match a program, the program has a syntax error. When the grammar matches a program, a parse is extracted and used to compile the program. Most context free matching uses bounded lookahead, which makes it practical for parse extraction to occur simultaneously with
matching because backtracking is avoided. The most powerful context
free matching algorithms can search for general context free
grammars, but they require backtracking or simultaneous exploration
of multiple options. Such algorithms require a separate parse
extraction phase, usually operating in time proportional to the square
of the size of the match. [Graham et al. 1980] is an excellent portal
into the literature on general context-free matching.

SNOBOL4, Icon, and Gnu Emacs’ regular expression search both use
recursive backtracking to search for patterns. Through the use of the
Immediate Assignment operator in SNOBOL4, which assigns to a
string variable the substring currently matching a subpattern,
information similar to that provided by parse extraction can be
extracted from a match. Gnu Emacs has a similar capability: roughly
speaking, the last string matching any parenthesized expression in the
pattern can be extracted. The simple, but potentially very slow
recursive backtracking algorithm used for matching in SNOBOL4,
Icon, and Gnu Emacs allows Immediate Assignment to be
implemented easily.

CPDS matches much more cleverly than the simple recursive
backtracking matchers. In return it operates with a much better
worst-case performance than SNOBOL4, ICON, and Gnu Emacs, while
retaining the ability to match Immediate Assignment patterns. Thus
CPDS has parsing abilities similar to the recursive backtracking
matchers, and a more predictable runtime. However, the worst-case
performance of CPDS can still be too slow for practical use.
3. Previous Work

As will be detailed later, the parse extraction algorithm in the thesis operates in time proportional to the size of the match times the depth of the pattern. This is quite efficient, especially in combination with DFA based pattern search. In addition, it can provide more parse information than Immediate Assignment (such as the string that matches each iteration of loop, instead of the string that matches the last iteration) and packages it in a more convenient form (the parse tree abstract data type discussed in detail in chapter 4).

3.7. Languages Integrating Pattern Matching

There have been a number of significant attempts at integrating a general purpose pattern matching capability with a general purpose language. Since the thesis introduces another one, TLEX, we review existing tools.

SNOBOL4, a language specifically designed to integrate string pattern matching, was introduced previously. The SNOBOL4 general purpose language has a convenient set of operations on strings. However, SNOBOL4 has no notable control flow features for constructing pattern matching applications.

Icon is a successor to SNOBOL4. It was also discussed previously. Its string operations and lack of specialized control structures parallel those of SNOBOL4. It offers instead a standard set of control structures (i.e. if..then, while..do) to enable the programmer to easily construct a customized control structure. In some applications this
flexibility is crucial; in other applications it is yet another burden on
the programmer.

[Liu 1986] shows how to integrate CPDS into the Pascal language.
The Pascal programmer can extract information from a successful
match by assigning a submatch to a string variable, just as in
SNOBOL4. There are no special control structures for using a match.

[Lipson 1984] suggests a few minor programming constructs that
might make it slightly easier to use the result of a match. Specifically,
he suggests four rather complicated variants of the if...then statement
which were apparently designed to minimize the nesting of usual
if...then statements. As such, they seem to be cosmetic additions that
offer little help to the programmer of a pattern matching application.
More useful are his suggestions for parameterizing patterns. They
provide a controlled way to pass information to and from a pattern and
they limit the scope of Immediate Assignments.

Lex was designed as a tool for generating lexical analyzers [Lesk and
Schmidt 1986]. Since TLex is most like Lex, we describe it in some
detail. Lexical analysis is the first stage of compilation and involves
partitioning the input into tokens in preparation for parsing. Patterns
are specified with restricted regular expressions, which are compiled
into a very efficient (essentially linear time) search engine. In
addition, right context sensitivity is allowed at the top level of a
pattern, and only the top level of a pattern matches the longest string
possible. The latter two features are implemented outside the
formalism of regular expressions, in contrast to the approach to be
described in the thesis. The algorithms of Lex are described in [Aho et al. 1986]; however, [Paxson 1988] points out that Lex’s algorithm for context sensitivity is incorrect in some cases. Specifically, he points out that the pattern

\[(\text{CSRP} (+ \text{x}) (\text{SEQ} \text{x y}))\]

(which we have translated from the Lex pattern language to the language in the thesis) which is supposed to match “1 or more x’s when there is an xy to the right”, actually reports that the “xx” prefix of “xxy” matches the pattern. This is wrong, since there is no “xy” to the right of “xx” in “xxy”.

In contrast, the thesis implements LONGEST and CSRP much more generally than Lex. LONGEST and CSRP can appear in arbitrary subpatterns of a pattern, not just the top level. They are first class regular expression operators, just like SEQ, *, and OR. The matching algorithm in the thesis, unlike that in Lex, correctly matches CSRP and LONGEST patterns in all cases and on arbitrary subpatterns. A lexical analyzer built with the algorithms in this thesis could retain all of the speed of Lex while offering a more expressive pattern language and access to a parse of the match from within a rule’s action.

Lex supplies a significant control structure specifically designed to support lexical analysis. We describe it in simplified form: a Lex file contains a number of pattern/action pairs. A prefix of the remaining input is matched against the patterns. The pattern which matches the longest prefix is selected, and the corresponding action is executed. Then the matched input is discarded and the process repeats. The
actions are ordinary C code, but inside an action the Lex programmer can access the substring matching the rule’s pattern. No parse of the match is available. The Lex control structure had a direct and obvious impact on that of TLex.

Lex was designed to work especially well with Yacc [Johnson 1975], a tool that generates parsers. Yacc patterns are a subclass of the context free grammars, LALR(1) (left recursive with a look ahead of one token). The parsing algorithm is very efficient thanks to the restriction to single token lookahead, but the language to be matched has to be carefully designed because of this restriction. Actions written in C code are interspersed among the patterns; the parser operates in a bottom-up fashion so the actions are executed in a bottom-up fashion. This combines a control structure with a way to access the parse tree, because Yacc makes it easy for an action to access the values returned by previous actions at the same level of the parse tree.

NewYacc [Purtilo and Callahan 1989] improves Yacc by creating a parse tree before calling any actions. This allows the programmer to execute actions in a variety of orders, and even allows multiple passes through the parse tree. Otherwise, NewYacc behaves similarly to Yacc.

AWK is a language designed to make data file transformation programs quick and easy to write [Aho et al. 1988]. It is an interpreted general purpose language that is reminiscent of (but different from) C. AWK processes a file a line at a time, and automatically breaks up a line into a number of fields, which it detects by looking for a
user-definable delimiter. AWK makes it easy to access the ith field, and offers convenient tests matching a restricted regular expression against a field. AWK does not produce a parse tree as the result of a match. AWK is also limited by its line-at-a-time orientation and its somewhat restrictive concept of breaking the line up into discrete fields. Of course, these are exactly the control structures that make it so convenient for "quick and dirty" text processing programs.

3.8. Tools Using Regular Expressions

In this section we provide a sampling of the myriad application of regular expressions. The advances in this thesis can be used to make enhanced versions of many of these tools.

[Pike 1987] describes SAM, a text editor where text selection is done using a sequence of regular expressions, and all editing operations apply to the currently selected text.

The Unix operating system [Unix 1986] provides a number of tools that use regular expressions. The program Grep searches a file for lines matching a certain pattern; Lex generates lexical analyzers and is discussed above; the programs Sed and Vi are text editors that provide regular expressions to support search and replace operations.

In [Belli and Grosspietsch 1983], regular expressions are used to specify the correct inputs to a hardware circuit. The specification is then used to generate a circuit which finds and corrects erroneous runtime input.
3. Previous Work

[Berry and Gonthier 1988] describes, with few details, how they compile the Esterel language, a synchronous and reactive language, using the Brzozowski method of compiling extended regular expressions. The resulting combination of a program and a finite-state machine offers extremely fast synchronization.

[Jarvis 1976] shows how to use regular expressions in “syntactic recognition” of features in line drawings. Relevant features of a line drawing are extracted and converted into a long string describing the drawing. To search for a feature of interest the feature is coded as a regular expression and the long string is searched for matches to the regular expression.

[Anantharaman et al. 1985] describes a method for compiling path expressions, which are built from regular expressions, into VLSI synchronizers. The method makes it easy to convert readable specifications of the interaction of asynchronous circuits, for example a bus arbitration scheme, into an efficient VLSI implementation.
There are a variety of applications requiring pattern matching on a sequence of text or binary values. Some of the most familiar are lexical analysis and parsing, used in compilers; converting data files from one format to another; finding and extracting useful information in unstructured text; and extracting relevant information from debugging traces.

Because so many applications require pattern matching, many tools have been created to help the programmer solve them; we reviewed many in the previous chapter. TLex, which stands for Tyrannosaurus Lex, uses the algorithms described in this thesis to provide a pattern matching tool with unique abilities. To appreciate the special features of TLex, we compare it to the existing tools along four dimensions:

- How are patterns specified? What patterns can be specified?
- How efficiently can the input be searched for one or more patterns?
- What information is returned by a successful match? How easy is it to make use of such information?
- What control structures are provided to make it easier to write pattern matching applications?

The four dimensions are interdependent. For example, choosing a powerful pattern language makes efficient search less likely.
4.1. Why Another Language?

Given the range of capable tools presented in the previous chapter, one may question the need for any new tools. However, consider searching a day’s output of wire-service stories for some specific pieces of information, such as financial information, sports scores and statistics, or recipes. Such an application poses requirements not fulfilled by the existing tools:

- The “input text” is very large, on the order of several megabytes or more, emphasizing the need for very efficient pattern matching. This means that any search technique which requires $O()$ time or space is unacceptable, unless the constant is small and the input can be broken into a sequence of small, separately searched paragraphs.

- The language used to specify patterns should be as powerful as possible, yet modular and readable to facilitate maintenance. Restrictions such as “lookahead of 1 token” are bearable for the well designed syntax of modern programming languages, but are unacceptable when looking for myriad patterns in free text.

- Patterns of interest in the text may overlap; they cannot be restricted to arbitrary limits such as “a line at a time”, or even “a sentence at a time”.

- It is not enough just to find patterns of interest in the text. The system should provide maximum help in extracting random parts of a successful match. Otherwise, the programmer would be forced to write an additional parser for each item matched. For example, if a pattern matches an address, it should be easy for the programmer to access the zip code without writing any more code.

Analyzing the existing tools, SNOBOL4 and Icon have the problems of being non-standard general purpose languages; offering pattern matching which can be dramatically slow, especially when patterns get
complicated or the text is long; providing relatively unreadable patterns because the patterns are procedurally specified; providing only basic support for using a successful match. CPDS is attractive, but its $O()$ efficiency restricts its use to relatively small strings. The ROPE language supports readable, reusable patterns but shares the other disadvantages of SNOBOL4 and Icon. In addition it has not been implemented. Lex offers attractive efficiency and control structure, but it offers almost no support for extracting information from a match, and its pattern language is too restrictive for a sophisticated class of patterns. Yacc and NewYacc patterns are restricted to “look ahead 1”, making them too cumbersome to use for free text searching. AWK’s pattern language is essentially the same as Lex’s, and it further suffers from control structures which can be inappropriate for sophisticated applications, as we discussed above.

TLex was designed for applications similar to the wire-service application. It provides a very expressive pattern language, yet the patterns can still be matched with optimal efficiency (linear in the size of the input). It appears to the user program as a sophisticated subroutine, and provides a conceptually simple but effective pattern-action control structure. Actions are written in C or C++ code, and TLex automatically constructs a parse tree representing a match that can be conveniently queried inside the corresponding action.

The next sections describe the unique aspects of TLex. For a detailed language manual, see [Kearns 90].
4. TLex

4.2. Overview of TLex

TLex is a code generator and a runtime library designed to provide a programmer with efficient and easy-to-use string pattern matching. TLex was inspired by Lex, but differs from it in a number of important respects.

To use TLex, the programmer prepares a TLex input file. This file contains ordinary C code in the first section, a list of definitions and macros in the second, and a set of pattern/action rules in each succeeding section.
Here is a very simple TLex file, that counts the number of lines in a file.

```c
long count;
main()
{    
    initTLex("myfile.tld", 0, myfileTLexActions);
    doTLex(TLEX_FILE_BUFFER, "input", 0, 0, 1);
    deinitTLex();
}
```

```
(define CR  10)
(define StartOfBuffer
  (csrp Start empty empty))
(define StartLine
  (csrp (or CR Start) empty empty))
(define End (csrpn empty any))
```

```
StartOfBuffer
  ===>  
{  count = 0;  /* initialize count */  }

StartLine
  ===>  
{  count = count + 1;  }

End
  ===>  
{  printf("the Line Count is \%d\n", count);  }
```

Figure 4-1: A Simple TLex Input File

We define StartOfBuffer and End to be patterns that match at the very beginning and end of the file being matched. StartLine matches at the beginning of each line. The application simply runs the first ruleset on the file called “input”; as a result, the number of lines in the file is printed out.
Because TLex is a code generator, using it can be complex. Here is an overview of the process:

![TLex Overview](image)

TLex is a compiler that accepts a file with a .tlx extension and produces 2 files: a .tld data file, which has the compiled form of the patterns in the .tlx file, and a .c file which has the rule actions and user code from the .tlx file converted to compilable C or C++ code.

The user compiles the .c file and links it with tlex.lib, the TLex runtime library, to produce a runnable application. The end result is a user application using the TLex runtime system for efficient pattern matching and retrieval.

When the user application is run, the application may initialize the TLex runtime library to read from a previously compiled .tld file and execute one or more sets of rules on a file or buffer in memory.

For debugging patterns, we provide the program tlexdbg ("tlex debug"), which accepts a data file and an input file as parameters and runs the .tld (data) file on an input file. Tlexdbg runs each ruleset, from first to last, reporting which rules match, where the rules match, and the parse tree extracted from each match. Tlexdbg does not call the rule actions, which means the .c file produced by tlex does not have to be compiled.
4.3. The TLex Pattern Language

The TLex pattern language is technically as powerful as restricted regular expressions plus one added operator that matches the end of the text. However, to the programmer it appears to be a language more powerful than an unrestricted context free grammar. Besides the restricted regular expression operators of *, SEQ, and OR, TLex offers the additional operators AND, ALL/MINUS, LONGER, LONGEST, SHORTER, SHORTEST, CSRP, CSRPN, CTRP, and CTRPN, all of which will be explained shortly. TLex also simulates recursion using bounded recursion (The limitations of bounded recursion are discussed in section 4.3.2). TLex uses regular expressions in a way that provides infinite lookahead. In contrast, a general context free grammar offers true recursion, SEQ, and OR. It cannot offer AND or ALL/MINUS since the context free grammars are not closed under intersection or complement. Furthermore, a general context free grammar has no concept of lookahead. This justifies our contention that the TLex grammar appears more powerful than a general context free grammar. The only thing the latter can do that the former cannot is true recursion, which TLex simulates using bounded recursion.

On the other hand, the TLex pattern language is significantly less powerful than the ROPE, SNOBOL4, and Icon languages. A valuable feature missing in TLex, but present in SNOBOL4, Icon, and CPDS, is Immediate Assignment (explained previously), which prevents TLex from matching patterns such as “a word followed by the same word” or “the most often repeated word”. TLex also does not support the
dynamic creation of patterns at runtime; instead, all patterns which
are to be matched must be listed at compilation time. TLex ignores
this feature in order to implement very fast pattern matching.

4.3.1. Pattern Operators

The TLex pattern language is an extension of restricted regular
expressions. Here we give a succinct description of the TLex pattern
language; previous sections motivated its particular features. The
TLex pattern language is recursively defined. The basic unit is that of
a pattern. The TLex pattern language uses a syntax similar to Lisp,
where most statements are a parenthesized list of items. The first
item in the list is an operator name, and the rest of the items are
arguments to the operator. For example, in the pattern (seq a any b)
the operator is “seq” and the arguments to “seq” are the patterns “a”,
“any”, and “b”.

Table 4-1.-- TLex Pattern Language Summary

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>matches the letter “a”.</td>
</tr>
<tr>
<td>23</td>
<td>matches character with ascii code 23.</td>
</tr>
<tr>
<td>(from 48 to 57)</td>
<td>matches any character with ascii code from 48 to 57.</td>
</tr>
<tr>
<td>'your sign here'</td>
<td>matches the string in quotes.</td>
</tr>
<tr>
<td>&quot;your sign here&quot;</td>
<td>matches the string in quotes.</td>
</tr>
<tr>
<td>(notclass b 34 &quot;*&quot;)</td>
<td>matches any character</td>
</tr>
</tbody>
</table>
IV. TLex

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(class a 23 &quot;+&quot; )</td>
<td>matches any listed character.</td>
</tr>
<tr>
<td>empty</td>
<td>matches all 0 length substrings.</td>
</tr>
<tr>
<td>any</td>
<td>matches any character.</td>
</tr>
<tr>
<td>(seq pat1 pat2 pat3 ... )</td>
<td>matches a match to pat1 followed by a match to pat2, pat3, ...</td>
</tr>
<tr>
<td>[ pat1 pat2 pat3 ...]</td>
<td></td>
</tr>
<tr>
<td>(* pat)</td>
<td>matches 0 or more consecutive occurrences of matches to pat; &quot;*&quot; prefers to match as many times as possible, &quot;**&quot; as few times as possible.</td>
</tr>
<tr>
<td>(** pat)</td>
<td></td>
</tr>
<tr>
<td>(+ pat)</td>
<td>matches 1 or more consecutive occurrences of matches to pat; &quot;+&quot; prefers to match as many times as possible, &quot;++&quot; as few times as possible.</td>
</tr>
<tr>
<td>(++ pat)</td>
<td></td>
</tr>
<tr>
<td>(? pat)</td>
<td>optionally matches pat; &quot;?&quot; prefers to match, &quot;??&quot; prefers not to.</td>
</tr>
<tr>
<td>(?? pat)</td>
<td></td>
</tr>
<tr>
<td>(atleast n pat)</td>
<td>matches at least n, exactly n, or at most n consecutive occurrences of matches to pat; prefers to match as many as possible.</td>
</tr>
<tr>
<td>(exactly n pat)</td>
<td></td>
</tr>
<tr>
<td>(atmost n pat)</td>
<td></td>
</tr>
<tr>
<td>(or pat1 pat2 pat3 ... )</td>
<td>matches what matches pat1, pat2, pat3, ...; prefers to match the leftmost pattern.</td>
</tr>
<tr>
<td>(and pat1 pat2 pat3 ... )</td>
<td>matches what simultaneously matches pat1 and pat2 and pat3...; prefers longer matches.</td>
</tr>
<tr>
<td>(all pat1 minus pat2 pat3 ... )</td>
<td>matches anything that matches pat1 but does not match pat2, pat3, ...; prefers longer matches.</td>
</tr>
</tbody>
</table>
(define name pat1) defines "name" to stand for pat1.

(macro name2 pat) defines "name2" to be a new pattern operator. pat may contain patterns $1..$9, which stand for the first through ninth arguments to name2 in future uses.

(csrp pat patRight)
(csrp patLeft pat patRight) matches what pat matches when patRight matches a prefix of the string to the right, and if patLeft is present, patLeft matches the character to the left.

(csrp pn pat patRight)
(csrp pn patLeft pat patRight) matches what pat matches when no prefix of the string to the right matches patRight, and if patLeft is present, the character to the left does not match patLeft.

(ctrp pat patRight)
(ctrp patLeft pat patRight) matches what pat matches when patRight matches a prefix of the string that starts where the match to pat starts, and if patLeft is present, patLeft matches the character to the left.

(ctrpn pat patRight)
(ctrpn patLeft pat patRight) matches what pat matches when no prefix of the string starting where the match to pat starts matches patRight, and if patLeft is present, the character to the left does not match patLeft.

(shorter pat) matches what pat matches, but prefers shorter or longer matches.

(longer pat) matches what pat matches when pat matches several substrings starting at the same position, matches only the shortest or longest such substring.
4.3.2. More on Definitions and Bounded Recursion

Definitions may be recursive (or mutually recursive). However, TLex implements these definitions approximately, by in effect repeatedly substituting in the definitions a fixed number of times. To understand the limitations of bounded recursion, consider the recursive definition of a set of matching braces:

\[
\text{(define MP } (\text{ or empty seq '{' MP '}')))
\]

TLex treats this as the following pattern, assuming a maximum recursion depth of 3:

\[
\text{(define MP (or empty seq '{' or empty seq '{' or empty seq '{' or empty seq '{' or empty seq '{' or empty seq '{' or empty seq '{' (or empty (seq '{' NONE '}') '}') '}'))}
\]

TLex has converted the pattern into a non-recursive pattern. Notice that the final substitution replaces MP by NONE, a pattern that does not match anything. If TLex implemented true recursion, MP could match an infinite number of strings: the empty string, "[]", "{{}}", "{{{{}}}}", ..., "{{{{{{{{{{}}}}}}}}}}"..., However, TLex implements bounded recursion, and one can see that the expanded pattern matches the empty string, "[]", "{{}}", and "{{{{}}}}".

In typical regular expression applications, bounded recursion is adequate. For example, to match nested parentheses in text one might expect the large majority of occurrences will have a maximum nesting less than 3. For C programs created by humans, it is rare to find a nesting of braces deeper than 7.

Unfortunately, bounded recursion may lead to huge patterns. Consider the definition of TwoT:

```
(define TwoT (or T (seq TwoT TwoT)))
```

TwoT matches all strings of T's. When implemented using bounded recursion the size of the pattern essentially doubles every time the maximum recursion depth increases by 1. As a result of this exponential growth in the size of the patterns, bounded recursion of a useful depth may be unfeasible to match.

4.3.3. The Start of the Input

TLex “hallucinates” a special character before the first character in the input, which the TLex programmer can refer to with the definition “Start”. Start defaults to character 128, but the TLex programmer can redefine it if necessary by defining Start appropriately. This character can never be directly matched except by the patLeft of a csrp(n) or ctrp(n). Its purpose is to allow the specification of patterns that begin at the start of the input.

Examples:
To match the beginning of the input:
```
(define StartOfInput (csrp Start empty empty))
```
To match the end of the input:
To match the beginning of a line (including the first line):

```
(define CR (or 10 13))
(define StartOfLine (csrp (or Start CR) empty empty))
```

Another way to match the beginning of a line (including the first line) involves redefining Start to CR:

```
(define Start 10)
(define CR (or 10 13))
(define StartOfLine (csrp CR empty empty))
```

### 4.3.4. Sample Patterns and Macros

In the following subsections we give a few examples in which we show a pattern, a string, and what parts of the string are matched by the pattern. Here is an example:

<table>
<thead>
<tr>
<th>pattern</th>
<th>input string</th>
</tr>
</thead>
<tbody>
<tr>
<td>(seq a any any b)</td>
<td>aababbbaabbbbababaababbaabaaaabb</td>
</tr>
</tbody>
</table>

The sequences of carats (^) show a range of characters matched by the pattern on the left. Sometimes a pattern matches a zero length range of characters, namely a single position between two characters. In this case we put “><” under the position; the points of the “>” and “<” point to the position.

*O* (upto p) uses the context sensitive right pattern (introduced below) to match the string up to the start of a substring matching p, or up to the end of the string, whichever comes first.

```
(macro upto
  (shortest (csrp (* any) (or $1 End)))))
```
To match a word one can use the following definitions:

\[
\text{(define alphanumeric (or (from "0" to "9")}
\]
\[
\text{ (from A to Z))}
\]
\[
\text{(define nonAN (notclass alphanumeric))}
\]
\[
\text{(define word (csrp nonAN (+ alphanumeric) nonAN))}
\]

When given the input “Hi there. My name is steve! Wow” word can match:

\[
>><>><><><<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><}<
but never a HalfFloat when we can match a FullFloat. The following
ctrpn pattern accomplishes this:

(or FullFloat (ctrpn HalfFloat FullFloat)).

4.4. Rules and Parse Trees

With just the TLex pattern language, TLex would be a fancy way to
find places in the text that match patterns. When we add the
capability to carry out actions after matching a pattern, things get
interesting. When we make available a parse tree describing the match
in detail, things get exciting!

The basic unit of control in TLex is the rule. All rules have the
following form:

\[
\text{pat1}
\text{options: <option>}
\Rightarrow
\{ \text{C-expression} \}
\]

“pat1” is a pattern as defined in the pattern language section. The
(optional) options to the rule allow the programmer to disallow
overlapping matches and/or inhibit parse extraction. After a match of
pat1 is found, TLex creates a parse tree describing the match, and then
calls the C-expression. The parse tree remembers which substrings of
the input match which subpatterns of pat1. Inside the C-expression
the user may write special expressions which access the parse tree.
These special expressions are detailed in appendix D; in this section we
give an example of their use.
As an example, the following rule, with associated definitions, matches a list of decimal numbers and prints out the total of all numbers which are prefixed by a dollar sign. For example, on the input “23 $34 $3 5 3 $1” it prints out “$38”.

(define digit (from “0” to “9”))
(define number (csrpn digit (+ digit) digit)
(define space (class 32 9 10 12 13))
(define NumberList (+-1 [(?-1 “$”) number-1 (* space)]))

NumberList-1 ==>
{
    ss numSS;
    int sum;

    // C++ code
    cout << “read: ”
        << SS[“NumberList-1.+-1”].count()
        << “numbers.  \n”;

    for (sum = 0, numSS = SS[“NumberList-1.+l[0]”];
         numSS;
         numSS = numSS.adv(1))
    {
        if (numSS[“?-1”].thru())
            sum += numSS[“number-1”].tonum();
    }

    cout << “$” << sum << “\n”;
}

The operators +-1, ?-1, and number-1 are all NAMED instances. Any operator can be named (except for empty, any, the primitive patterns, and the patLeft or patRight of a context sensitive pattern), by appending a “-<num>” to the operator name. Doing so informs TLex that one might be interested in accessing the parse tree rooted at that pattern, and provides a convenient name to use in accessing this parse tree. In essence, naming is specifying the desired detail of the parse tree that TLex should create from a match.
The expression $SS["NumberList-1.+-1"]$.count() is called a parse access. It is a C++ expression which finds the parse tree representing $+-1$ inside the parse tree representing $NumberList-1$ inside the parse tree $SS$, and returns the number of times the loop was iterated. $SS$ is the parse tree TLex creates during parse extraction to represent the whole match.

The expression

$$numSS = SS["NumberList-1.+-1[0]"]$$

initializes $numSS$ to the parse tree representing the first iteration of $+-1$. Each time through the loop $numSS$ gets set to the next iteration with the expression

$$numSS = numSS.adv(1)$$

The loop terminates when $numSS$ becomes empty, which will happen when each iteration of $+-1$ has been processed.

Inside the loop,

$$numSS["?-1"].thru()$$

returns TRUE when there is a dollar sign before the number. To convert the matched number into a number (it is matched as a string of digits, remember) the program uses the expression

$$numSS["number-1"].tonum()$$

BBall.tlx, in appendix F, is a large, sophisticated example that illustrates the full range of parse accesses possible in a rule action.
4.4.1. Predicting the Parse

It is possible that one match may be parsed in multiple ways. The simplest example is parsing

\[(\text{seq} (*1 \text{ any}) (*2 \text{ any}))\]

to “12345”. One possible parse is \(*1 = “12”\) and \(*2 = “345”\). Another possibility is \(*1 = “”\) and \(*2 = “12345”\). For this example there are actually 6 different possibilities, any of which are correct parses. TLex operates according to a few simple rules that permit one to predict the actual parse. This is an important capability, as it gives the programmer useful control.

- If \(p\) is one of the \(*, +, ?, \text{atmost}, \text{or atleast}\) patterns, the parse prefers to loop through \(p\), instead of leaving or skipping \(p\). If \(p\) is one of the \(**, ++, \text{or ??}\) patterns, the parse prefers to leave or skip \(p\), instead of looping through \(p\).

- Whenever choosing between several OR alternatives, the parse chooses the first (leftmost) possible alternative.

- The patterns (\(\text{longer} \ p\)) and (\(\text{shorter} \ p\)) prefer to match the longest and shortest matches to \(p\) that allow the rest of the pattern to match successfully.

- \(\text{(and} \ ...)\) and (\(\text{all} \ ... \ \text{minus} \ ...)\) prefer to match the longest strings that allow the rest of the pattern to match successfully.

- anystr prefers to match as few characters as possible.

- The preferences of outer patterns take precedence over the preferences of inner patterns. In \(\text{seq} \ p1 \ p2\), the preferences of \(p1\) take precedence over \(p2\).

Examples:

\[(\text{seq} (*1 \text{ any}) (*2 \text{ any})) \text{ to “12345”:}\]
\[\*1 = “12345”, \ *2 = “”\]

\[(\text{seq} (\text{shorter} (*1 \text{ any})) (*2 \text{ any})) \text{ to “12345”:}\]
\[\*1 = “”, \ *2 = “12345”\]
4. TLex

A robust implementation of TLex exists. It has two parts: a compiler written in 11,000 lines of C++ code, and a runtime library written in 6,000 lines of C. This section presents a practical view of TLex. We characterize its performance on two sample applications; we describe a number of applications using TLex; and we discuss lessons learned from the use of TLex. All experiments were run on a Sun 4/390, and all reported timings are the sum of the (user + system) times extracted using one of the Unix “time” commands or system calls. These essentially represent CPU time.

4.5.1. Words, Sentences, and Lines

In order to compare TLex against a number of other pattern matching tools, we implemented the same application with each tool.
Specifically, the problem was to count the number of words, sentences, and lines, and to print out the longest occurrence of each. Finding an application that all the tools can implement tends to restrict one’s choice to simple applications.

The application, called WC2, was implemented using TLex, FTLex, GAWK, Icon, Flex, Yacc, and RLex. TLex has an option to pre-compile the pattern into a more efficient form; this increases compile time but provides faster and more consistent searching time. We call the program that uses TLex in this way “FTLex”, for Fast TLex. (Sections B.9.4 and B.9.5 explain the different searching modes in detail.)

GAWK is the Gnu version of AWK; version 8 of Icon was used; and Flex is an especially fast and compact version of Lex [Paxson 1988]. RLex is another pattern matching tool by the author that offers essentially the same pattern language and parse tree abstraction as TLex. However, it searches for patterns using recursive backtracking and it offers no special control structures. Thus RLex is good for isolating the pattern matching and control aspects of TLex from the parse tree and expressiveness aspects.

For each tool we measured the number of seconds that each tool requires, as the average of 2 runs. The input was a text of 270,600 characters, close to the size of this thesis, containing 41,464 words, 6,328 lines, and 4,884 sentences. The source code for the TLex version of WC2 is in appendix E. Figure 4-3 displays the results.
Although TLex was designed for much more sophisticated applications than WC2, its performance is similar to most of the other programs. Only Flex significantly betters it, and only GAWK is significantly bettered. The pre-compiled program, FTLex, was about the same speed as TLex. This indicates that most of the time is being spent in parsing and processing the matches, not searching for them. The next example illustrates much better the advantage that can result from pre-compiling an application.

4.5.2. Basketball

For a more realistic benchmark we took the TLex input file for an actual application and stripped it of all but the TLex pattern searches and the parse tree accesses inside the rules. Doing so isolated the pattern matching aspects of an actual application. The resulting TLex application, called BBall, searches sports stories for a set of game scores, player statistics, team standings, etc. The source code to BBall is in appendix F. Notice that it has a large number of patterns.

It would be nice to compare TLex to the other tools on this task. However, the task is complicated enough that it is unreasonably difficult to implement it in any of the other languages. For example, Yacc’s bottom up control structure is much too primitive to handle the necessary manipulations. Flex can probably locate the same patterns,
but extracting the relevant information is onerous because Flex lacks the parsing capability of TLex. Because many patterns extend over more than 1 line, and because the logical fields being extracted do not correspond to the simple fields in AWK, implementing BBall in AWK is unrealistic. A program could be written in RLex or Icon to accomplish the same thing as BBall; however, an RLex program would essentially recreate the built-in control structure of TLex, and because of the large patterns would perform worse. The Icon program would be faced with the same obstacles as the RLex program; in addition, the Icon program must explicitly build up a parse tree. Finally, the BBall application is typical in that patterns have to be continually improved as unexpected examples occur. As a result, the declarative nature of TLex patterns is a distinct advantage that the procedurally specified Icon patterns do not share.

The BBall application runs each of eight rulesets on the input text, in turn. “Fast-BBall” is the same application, using the pre-compilation feature of TLex. Each run requires at least two passes through the input, plus time for extracting a parse when a match is detected, plus the time used in an action to access the resulting parse tree. In the benchmark, the input text consisted of 195,000 characters, representing a collection of basketball stories (see the excerpt in appendix G). BBall was compiled and run on a Sun 4/390. Table 4-2 summarizes the performance of BBall on the input text.
Table 4-2.-- Behavior of BBall

<table>
<thead>
<tr>
<th></th>
<th>TLex Compile Time</th>
<th>Match Time</th>
<th>Parse &amp; Access Time</th>
<th>Total Time</th>
<th>Total Matches</th>
<th>Total States + Substates</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBall</td>
<td>10 sec</td>
<td>85 sec</td>
<td>11 sec</td>
<td>96 sec</td>
<td>634</td>
<td>47,147</td>
</tr>
<tr>
<td>Fast-BBall</td>
<td>7450 sec (~ 2 hour)</td>
<td>4.4 sec</td>
<td>6.8 sec</td>
<td>11 sec</td>
<td>634</td>
<td>535,178</td>
</tr>
</tbody>
</table>

The “Total Matches” is the number of matches for which TLex extracted a parse and called an action while running all 8 rulesets. The “Total States + Substates” entry lists the total number of states and substates that were discovered while running the 8 rulesets. For Fast BBall, the “Total States + Substates” is the number of states and substates discovered after fully pre-compiling the rulesets. Figures 4-4 and 4-5 break down the time spent on each ruleset:

Figure 4-4: BBall Timing

Figure 4-5: Fast-BBall Timing
We can make some observations from these measurements.

- The compile time is quite reasonable for BBall, allowing an efficient edit-test-debug loop while developing patterns.

- Although BBall contains many patterns, the space used by TLex routines is essentially linear in the size of the input. TLex saves a four-byte state for each character in the input; this dwarfs the 32,181 states + substates which are discovered during matching.

- In this application, the matching time dominates the parsing and action time. As a result, Fast-BBall dramatically outperforms BBall, by a factor of 9.

- The speed of matching is dependent on the composition of the pattern in BBall. This is not the case in Fast-BBall.

4.5.3. Other Applications

A number of applications based on TLex have been in daily use for more than a year. Many of the applications involve searching news stories for a set of specific facts. For example, there is one for each of Baseball, Football, College and Professional Basketball. The applications extract late-breaking scores and game summaries, and update statistics. Another application reads the Dow Jones News Wire and extracts a comprehensive set of stock and commodity prices.

TLex has also been used to normalize a huge set of addresses, and to validate and parse a set of data files. The advanced operators of TLex are particularly useful in data validation. If one expects each line of a file to match the DataLine pattern, for example, then one can detect all misformatted lines by matching the pattern

(ALL Line MINUS DataLine)
It should be mentioned that TLex is small and efficient enough to be used on personal computers. In fact, all of the applications we described run on PCs.

4.5.4. Discussion

One surprising fact shown by the many TLex applications is that the advanced operators AND, ALL/MINUS, LONGEST, SHORTEST, LONGER, and SHORTER were rarely used. Pattern programmers preferred to use the restricted operators, the CSRP patterns, and the macros and definitions; in addition, they were very grateful for the ability to control preferences using (** p) versus (* p), (?? p) versus (? p), etc.

The promise of maintainability offered by TLex showed itself again and again. In one case, a set of patterns for matching college basketball stories was converted to a different sport, hockey, in a few hours.

The most important feature of TLex, in practice, was the parse extraction and the convenient access provided by the parse tree access functions. In effect, these features turn flat text into meaningful structures.

The very fast compilation TLex offers was crucial to its use. For many applications, developing a pattern is an experimental science because there is no formal definition for the input, just a number of typical cases. This necessitates a constant edit-compile-debug cycle.
TLex was actually fast enough, in default mode, for all the applications it was used in. However, programmers are always clamoring for more speed. As a result, it is important and useful to have the pre-compiled mode. For example, while patterns are being created fast compilation with slow execution is quite reasonable. In the final application what is often desired is the fastest execution, even at the cost of a very long compilation and a large space requirement for the compiled patterns.
5. The Matching Architecture

In this chapter we give an overview of how the matching and parsing algorithms described in the thesis work together to solve a pattern matching task.

5.1. The Goal of Pattern Matching

What exactly is the goal of a pattern matcher? There are several possibilities: all possible matches of a pattern in the input (there may be none of these for an input string of characters); or the leftmost, rightmost, or an arbitrary match in the input. There is an orthogonal decision to be made about what it means to find a match. Again, there are several possibilities: find the starting position; find the ending position; find both the starting and ending positions; produce a full parse of the match; produce all possible parses of a match. To explain the distinction between the last two, we briefly describe a “parse”.

Whereas a match just reports the starting and/or ending position of a substring of s matching p, a parse returns much more information. A parse reports what subparts of p match what substrings of the whole match. For example, a match might report that the string “Joe Smith” matches the pattern \((\text{SEQ Name Space Name})\) while a parse would add that “Joe” matches the first Name, and “Smith” matches the second. Parsing a match makes pattern matching significantly more useful.
Some matches may be parsed in two different ways. A pattern that allows this is called an “ambiguous grammar” in context free parsing. A trivial example is the match of “aaabbb” against (SEQ ANYSTR ANYSTR). One possible parse is “aaa” matching the first ANYSTR and “bbb” the second. Another possibility is “aaabbb” matching the first ANYSTR and “” matching the second. Both are correct parses.

The thesis describes a procedure for finding the starting positions of all matches of a pattern p in the input s; it also describes how to extract one parse from any of the starting positions. The next section presents an overview of how the algorithms described in the thesis can be used for this purpose.

5.2. Pattern Matching with Next() and Parse()

It is shown how to convert any pattern p into another pattern \( p_{rev} \), which has the characteristic that whenever p matches a substring, \( p_{rev} \) matches the reverse of that substring. Reversing an extended regular expression produces another extended regular expression, which is a straightforward way of saying that extended regular expressions are closed under reversal.

An algorithm is presented to convert any pattern p into a function \( \text{func}_p \) which calculates, for an arbitrary set \( \{S\} \) of positions in s, the predicate \( \{S\}..j \text{ matches } p \). This predicate is explained in full detail later, but essentially \( \{S\}..j \text{ matches } p \) is TRUE iff there is an i in \( \{S\} \) such that \( i..j \text{ matches } p \). (\( i..j \text{ matches } p \) when the text between
positions i and j match p.) Furthermore, \( \text{func}_p \) is incremental in that the calculation of \( \{S\}_{..j+1} \) matches p uses the result of \( \{S\}_{..j} \) matches p.

The function Next(p, ...), which is the subject of the next chapter, is \( \text{func}_p \).

The two techniques are then combined as follows: we create \( p_{\text{rev}} \) and \( \text{func}_{p_{\text{rev}}} \) from p; we reverse the input string s to get \( s_{\text{rev}} \); we choose the set of ALL positions in \( s_{\text{rev}} \) as \( [S] \), writing it as \( \{\text{ALL}\} \); then we use \( \text{func}_{p_{\text{rev}}} \) to calculate \( \{\text{ALL}\}_{..j} \) matches \( p_{\text{rev}} \) for each j in \( s_{\text{rev}} \), using \( s_{\text{rev}} \) as the input string. Any j for which \( \{\text{ALL}\}_{..j} \) matches \( p_{\text{rev}} \) in \( s_{\text{rev}} \) marks a starting position of a match of p in s! To see this, realize that \( \{\text{ALL}\}_{..j} \) matches \( p_{\text{rev}} \) in \( s_{\text{rev}} \) precisely when some substring \( i..j \) of \( s_{\text{rev}} \) matches \( p_{\text{rev}} \). (We know this from the definition of \( \{S\}_{..j} \) matches p, and the definition of \( \{\text{ALL}\} \).) Whenever \( i..j \) of \( s_{\text{rev}} \) matches \( p_{\text{rev}} \), the reversed substring \( j..i \) of s matches p, from the definition of \( p_{\text{rev}} \).

Therefore j is the starting position of a match of p in s.

EXAMPLE:

As a very simple example, we will search the string “I ran my foot into the foobar” for the pattern (seq “foo” (? “bar”)), which is the string “foo” optionally followed by the string “bar”. Calling the input string s and the pattern p, \( s_{\text{rev}} \) becomes “raboof eht otni toof ym nar I”, and \( p_{\text{rev}} \) becomes (seq (? “rab”) “oof”). (We explain how to reverse a pattern later in great detail.) Searching for an optional “rab” followed by “oof”, and marking the end of such matches, we get the following:

\[
\text{text: raboof eht otni toof ym nar I}
\text{p}_{\text{rev}} \text{ matches: } \quad \text{<>} \quad \text{<>}
\]
Reversing the string, we get:

```
text:  I ran my foot into the foobar
p matches:          ><            ><
```

The marks now indicate the start of all matches to \( p \) in \( s \)!

END EXAMPLE.

The first pass through the input finds all positions ending a match to \( p_{rev} \) in \( s_{rev} \). Let \( j \) be any such position. The elegant recursive algorithm Parse(), the subject of chapter 7, constructs a parse tree representing a match to \( p_{rev} \) ending at position \( j \) in \( s_{rev} \). Finally, a trivial transformation converts this parse tree into a parse tree describing a match to \( p \) starting at position \( j \) in \( s \). (In practice the last two steps are combined into one.)

5.3. Pattern Matching Performance

The running time of the algorithms is discussed later in detail, but a summary shows the efficiency of the process. The reverse pass can be done in time proportional to the size of \( s \). Extracting a parse requires saving the sequence of states created during the reverse pass; there is one state saved per input character. Saving the sequence of states requires space essentially proportional to \( s \), thanks to optimizations we shall describe. Extracting a parse takes time proportional to the length of the match times the “depth” of \( p \). (The depth of \( p \) is roughly the deepest level of nesting of parenthesis in the extended regular expression describing \( p \).) Because of the excellent running times and
5. The Matching Architecture

space requirements of these algorithms, it becomes feasible to match non trivial patterns against large texts.

To give a better feeling for the powerful calculations taking place, note that finding all matches of a pattern $p$ against input $s$ with $n$ characters essentially checks all possible substrings of $s$ against $p$. There are approximately $\binom{n+|p|-1}{|p|-1}$ substrings of $s$; when the input is 100 thousand characters there are nearly 10 billion possible substrings to check.

5.4. Discussion

The astute reader may realize that we can run $\text{func}_p$ on the string $s$ in a left to right pass (i.e. the “normal” direction) and thereby find all ending positions $j$ of a match of $p$ in $s$, without having to reverse the pattern or the input string or the parse tree. What is the advantage of doing all the reversals? The advantage is that reversing the pattern $p$ converts a number of advanced operators that seem to require lookahead (namely $\text{LONGER}$, $\text{LONGEST}$, $\text{CSRP}$, $\text{CSRPN}$, $\text{CTR}$, $\text{CTRPN}$) into operators that require “lookbehind”. The latter is relatively easy to implement, but no efficient way is known to implement the former. However, lookahead appears to be significantly more useful in practice. Thus the reversing trick is a central innovation of the thesis which makes it possible to implement advanced operators which seem to require lookahead.

On the other hand, since the first phase of the algorithm is essentially a reverse pass through the input, the whole input must be present
before any matching is done. This characteristic means that the algorithm is inappropriate for some applications that involve processing a stream of data as it arrives - so called “online processing”. However, even in such cases the Next() and Parse() algorithms in the thesis can still be useful. The LONGER and LONGEST operators are no longer implementable, but the AND, SHORTER, SHORTEST, and ALL/MINUS operators are still available, and the operators specifying right context are replaced by operators specifying left context. The thesis concentrates on the more difficult case because right context sensitivity and the LONGER and LONGEST patterns seem to be preferred in practical applications.

5.5. Simplifying the Problem

The TLex pattern language has many operators: SEQ, OR, *, ATMOST, ATLEAST, EXACTLY, EMPTY, x, ANY, ANYSTR, NOTCLASS, FROM/TO, +, DEFINE, AND, ALL/MINUS, SHORTER, LONGER, SHORTEST, LONGEST, CSRP, CSRPN, CTRP, CTRPN. Implementing TLex is greatly simplified by selecting a simpler set of operators and implementing all the patterns in terms of these. We implement the following set of simpler patterns, which enables us to provide the 24 TLex operators using only 14 primitives: SEQ, OR, *, EMPTY, {x}, ANYSTR, DEFCALL, MINI-AND, MINI-NOT, SET_SHORTER, SET_LONGER, SET_SHORTEST, SET_LONGEST, CSLP. Furthermore, the SET_ algorithms are very similar to one another, so effectively the problem is simplified to implementing 11 primitives. (ANYSTR can easily be replaced by (** ANY), but because
it is such a widely used pattern it deserves special implementation. A similar argument could be made for strings.)

As described in the overview of the pattern matching architecture, the first stage involves reversing the patterns. This section describes the translation process from the general TLex language to a simpler set of patterns, and it describes how to reverse a pattern. In practice both stages are combined into one.

5.5.1. Reversing a Pattern

Before defining the reverse of a pattern, we define how to reverse a string. If an input string $s$ has positions from 0 to $n$, the reverse string $s_{rev}$ also has positions from 0 to $n$, which fall between characters. However, the $i$th position of $s$ becomes the $(n-i)$th position of $s_{rev}$, and the $i$th character of $s$ becomes the $(n-i-1)$th character of $s_{rev}$. Now we can define the reverse of a pattern.

**Definition 5.5.1:Rev:**

- $i..j$ matches $(rev\ p)$ in $s_{rev}$ if $n-j..n-i$ matches $p$ in $s$.
- $i..j$ matches $p$ in $s$ if $n-j..n-i$ matches $(rev\ p)$ in $s_{rev}$.

**end Definition.**

$(rev\ (rev\ p))$ is $p$, which can be seen by twice applying the definition of reverse, so reversing a pattern twice leaves it unchanged.

We can then prove the following formulae, which show how to recursively convert any regular expression $p$ into a regular expression equivalent to $(rev\ p)$.
Theorem 5.5.1: Reverse:

- \( \text{rev} \{x\} = \{x\} \)
- \( \text{rev} \ \text{EMPTY} = \text{EMPTY} \)
- \( \text{rev} \ \text{ANYSTR} = \text{ANYSTR} \)
- \( \text{rev} \ (\text{DEFCALL} \ p) = (\text{DEFCALL} \ (\text{rev} \ p)) \)
- \( \text{rev} \ (\text{OR} \ p1 \ p2) = (\text{OR} \ (\text{rev} \ p1) \ (\text{rev} \ p2)) \)
- \( \text{rev} \ (\text{SEQ} \ p1 \ p2) = (\text{SEQ} \ (\text{rev} \ p2) \ (\text{rev} \ p1)) \)
- \( \text{rev} \ (* \ p1) = (* \ (\text{rev} \ p1)) \)
- \( \text{rev} \ (\text{AND} \ p1 \ p2) = (\text{AND} \ (\text{rev} \ p1) \ (\text{rev} \ p2)) \)
- \( \text{rev} \ (\text{ALL} \ p1 \ \text{MINUS} \ p2) = (\text{ALL} \ (\text{rev} \ p1) \ \text{MINUS} \ (\text{rev} \ p2)) \)
- \( \text{rev} \ (\text{LONGER} \ p1) = (\text{SET}_\text{LONGER} \ (\text{rev} \ p1)) \)
- \( \text{rev} \ (\text{SHORTER} \ p1) = (\text{SET}_\text{SHORTER} \ (\text{rev} \ p1)) \)
- \( \text{rev} \ (\text{LONGEST} \ p1) = (\text{SET}_\text{LONGEST} \ (\text{rev} \ p1)) \)
- \( \text{rev} \ (\text{SHORTEST} \ p1) = (\text{SET}_\text{SHORTEST} \ (\text{rev} \ p1)) \)
- \( \text{rev} \ (\text{CSRP} \ p\left p \ p\right) = \)
  \( \text{(CSLP} \ (\text{rev} \ p\right) \ (\text{rev} \ p) \ (\text{rev} \ p\left)) \)
- \( \text{rev} \ (\text{CSRPN} \ p\left p \ p\right) = \)
  \( \text{(CSLPN} \ (\text{rev} \ p\right) \ (\text{rev} \ p) \ (\text{rev} \ p\left)) \)
- \( \text{rev} \ (\text{CTRP} \ p\left p \ p\right) = \)
  \( \text{(CTLP} \ (\text{rev} \ p\right) \ (\text{rev} \ p) \ (\text{rev} \ p\left)) \)
- \( \text{rev} \ (\text{CTRPN} \ p\left p \ p\right) = \)
  \( \text{(CTLPN} \ (\text{rev} \ p\right) \ (\text{rev} \ p) \ (\text{rev} \ p\left)) \)

go to end Theorem.

Notice that the reversed patterns look very similar to the original.

Appendix A contains a proof of this theorem. Some new operators are introduced in the theorem, but they are simple reversals of the previous operators. For example, \((\text{CSLP} \ p\left p \ p\right)\) matches what \(p\) matches when a suffix of the string to the left matches \(p\left\) and the single character to the right matches
5. The Matching Architecture

patRight. (SET_LONGEST pat1) is the reverse of (LONGEST pat1): if several substrings ending at the same position match pat1, then (SET_LONGEST pat1) matches only the longest such substring.

5.5.2. Translating a pattern

The TLex language allows the operators SEQ, OR, AND, and ALL/MINUS to have more than 2 subpatterns. For ease of explication and programming, it is convenient to convert such patterns into equivalent patterns where each operator has exactly 2 subpatterns. Since these operators are associative, this poses no problem: when OP is one of SEQ, OR, AND, (OP p1 p2 p3) becomes (OP p1 (OP p2 p3)).

The next theorem summarizes the translations that replace a number of the TLex operators by combinations of the simpler operators. Some of them are to be applied before reversal, some after. Several rules may have to be applied to a pattern to reduce it to the implemented primitives.

Theorem 5.5.2: Subpatterns:

1. \( x \_ \) (the set with one character, \( x \})
2. \( \text{ANY} \_ \) (the set of all characters)
3. \( \text{NOTCLASS} \ a \ b \ c \) \_ (the set of all characters) - \{a b c\}
4. \( \text{FROM} \ x \ to \ y \) \_ (the set of characters from \( x \) to \( y \})
5. “anything” \_ (SEQ a n y t h i n g)
6. \( + \ p \) \_ (SEQ \( p \) (* \( p \) ))
7. \( \text{CTLP pleft} \ p \ \{x\} \) \_ (SEQ \( p \) (CSLP pleft EMPTY \{x\}))
The first five and last four equalities are obvious. The first four just make explicit on the right, the set of characters on the left. The last four are basic macro expansions. Theorem 5.5.2:Subpatterns:(6) is directly from the definition of +. Theorem 5.5.2:Subpatterns:(13) is not actually used in our implementation, but it shows how \((\text{SHORTEST } p1)\) can be rewritten using other advanced operators. See appendix A for proofs of the remaining equalities.

Cases (10) and (11) may be surprising: why are these surrounded by \text{SET_LONGER}, and what do the pattern MINI-AND and MINI-NOT match? To understand these patterns one must consult the formalism in appendix A. Trying to give an intuitive explanation of their
behavior is harder than understanding the formalism that explains it simply.
6. Next()

In this chapter we give a function Next(p, ...) which corresponds to an arbitrary extended regular expression p. First we list the function Next(), then we give an intuitive justification for it. A formal specification and correctness proof can be found in appendix B.

6.1. Listing of Next()

function Next(p, instate, Char, TF) returns outstate:
begin
1    outstate.inp ♦ TF;
2
3    case p of
4          EMPTY:
5              outstate.q0 ♦ 0;
6              outstate.accepts ♦ TF;
7          {x}:
8              outstate.accepts ♦ (instate.q0 = 1);
9              outstate.q0 ♦ if TF and (Char in {x}) then 1 else 0;
10          ANYSTR:
11              outstate.q0 ♦ if TF or (instate.q0 = 1) then 1 else 0;
12              outstate.accepts ♦ (outstate.q0 = 1);
13          (SEQ p1 p2):
14              outstate.q1 ♦ Next(p1, instate.q1, Char, TF);
15              outstate.q2 ♦
16                  Next(p2, instate.q2, Char, outstate.q1.accepts);
17              outstate.accepts ♦ outstate.q2.accepts;
18          (OR p1 p2):
19              outstate.q1 ♦ Next(p1, instate.q1, Char, TF);
20              outstate.q2 ♦ Next(p2, instate.q2, Char, TF);
21              outstate.accepts ♦ outstate.q1.accepts or
22                  outstate.q2.accepts;
23          (* p1):
24              tempstate ♦ Next(p1, instate.q1, Char, FALSE);
25              outstate.q1 ♦ Next(p1, instate.q1, Char,
26                  TF or tempstate.accepts);
27              outstate.accepts ♦ TF or tempstate.accepts;
(DEFCALL p1):
    increment RecCount(p);
    if RecCount(p) > MAX_RECURSION then
        outstate.q0 ♦ 0;
        outstate.accepts ♦ FALSE;
    else
        outstate.q1 ♦ Next(p1, instate.q1, Char, TF);
        outstate.accepts ♦ outstate.q1.accepts;
    decrement RecCount(p);

(CSLP p1 p2 {x}):;
    outstate.q1 ♦ Next(p1, instate.q1, Char, TRUE);
    outstate.q2 ♦ Next(p2, instate.q2, Char,
    TF and outstate.q1.accepts);
    outstate.accepts ♦ outstate.q2.accepts and Char in {x};

(MINI-AND p1 p2):
    outstate.q1 ♦ Next(p1, instate.q1, Char, TF);
    outstate.q2 ♦ Next(p2, instate.q2, Char, TF);
    outstate.accepts ♦ outstate.q1.accepts and
    outstate.q2.accepts;

(MINI-NOT p1):
    outstate.q1 ♦ Next(p1, instate.q1, Char, TF);
    outstate.accepts ♦ not outstate.q1.accepts;

(SET_SHORTER p1):
    CMP ♦ '<';  ALL ♦ FALSE; goto label (SET_ p1);

(SET_LONGER p1):
    CMP ♦ '>';  ALL ♦ FALSE; goto label (SET_ p1);

(SET_SHORTEST p1):
    CMP ♦ '<';  ALL ♦ TRUE; goto label (SET_ p1);

(SET_LONGEST p1):
    CMP ♦ '>';  ALL ♦ TRUE; goto label (SET_ p1);
6. Next()

6.2. Intuition of Next()

Consider the following pattern:

\[
(\text{SEQ} (* \text{“Test”}) (\text{OR} \text{“Foo”} \\
\text{EMPTY}))
\]

We will be interested in specific places in this pattern, so we arbitrarily number these places:

\[
0 (\text{SEQ} 1 (* \text{“Test”}) 2 (\text{OR} 3 \text{“Foo”} \text{4} \\
\text{EMPTY} 6) 7) 8
\]
Call this pattern PP. (Actually we should also number the locations between each letter of “Test” and “Foo”, but for readability we do not.) There are an infinite number of “paths” through PP, where a path describes how a string can match this pattern. For example, the string “TestTestFoo” matches PP by entering the pattern, looping twice around the *, taking the first choice of the OR, and then exiting the pattern. The empty string matches PP by entering the pattern, looping zero times through *, taking the second choice of the OR, and exiting the pattern.

Roughly speaking, a state q of PP records the places that can be reached after reading some sequence of letters. For example, after reading “Test” the state of PP records that there are paths to each of 1, 2, 3, 5, 6, 7, 8. This is useful because if there is a path to the last place in PP, place 8, then we know the input string matches PP.

The function Next(p, q, Char, TF) assumes that q encodes a list of places reachable before reading Char; it also assumes that TF is TRUE when the first place in p is reachable before reading Char. Next() returns a new state, call it outq, which encodes 3 things: a new list of places reachable after reading Char; an outq.inp boolean which is TRUE when the first place of p is reachable before reading Char; and an outq.accepts boolean which is TRUE when the last place of p is reachable before reading Char.

Now the {x} case of Next() is understandable. The end of {x} can be reached when the start of {x} is reached before reading Char, and Char
is in \{x\}. Next() makes outstate.q0 be 1 when the end of \{x\} is reachable, so outstate.q0 gets 1 whenever TF is TRUE (the start of \{x\} is reached before reading Char) and (Char in \{x\}). Next() must set outstate.accepts to TRUE when the end of \{x\} is reached before reading Char. But this is TRUE when the end of \{x\} is reached after reading the previous Char, and this information is encoded as (instate.q0 = 1).

We can also see how Next() may be defined recursively. Assume that \( p = (\text{OR} \ p_1 \ p_2) \), and if \( q \) encodes a list of places in \( p \) then \( q.q_1 \) encodes the subset of places in \( p_1 \) and \( q.q_2 \) encodes the subset of places in \( p_2 \).

\[
\text{function } \text{Next}((\text{OR} \ p_1 \ p_2), \ q, \ \text{Char}, \ \text{TF}) \ \text{returns } \text{outq}: \\
\text{outq.inp} \leftarrow \text{TF}; \\
\text{outq.q}_1 \leftarrow \text{Next}(p_1, \ q.q_1, \ \text{Char}, \ \text{TF}); \\
\text{outq.q}_2 \leftarrow \text{Next}(p_2, \ q.q_2, \ \text{Char}, \ \text{TF}); \\
\text{outq.accepts} \leftarrow \text{outq.q}_1.accepts \text{ or } \text{outq.q}_2.accepts;
\]

This code works because the next list of places in \( (\text{OR} \ p_1 \ p_2) \) can be calculated by asking \( p_1 \) and \( p_2 \) for their next list of places. If TF is true then the start of \( (\text{OR} \ p_1 \ p_2) \) is reachable; since the starts of \( p_1 \) and \( p_2 \) are also reachable whenever the start of \( (\text{OR} \ p_1 \ p_2) \) is, the TF parameter to \text{Next}((\text{OR} \ p_1 \ p_2),...) becomes the TF parameter to \text{Next}(p_1,...) and \text{Next}(p_2,...). Similarly, if the last place of \( p_1 \) or \( p_2 \) is reachable, then the last place in \( (\text{OR} \ p_1 \ p_2) \) is reachable, so outq.accepts is TRUE when either outq.q1.accepts or outq.q2.accepts is TRUE.

Similar arguments justify all the other cases of Next(), except for the \text{SET_} cases. The latter are used to implement AND, ALL/MINUS, SHORTER..LONGEST. The procedure for (\text{SET_} \text{pat1}) is more
6. *Next()*

complicated than the other cases because we must keep a separate list of places for each time the start of pat1 is reachable. For the details of these cases, which are rather complicated, refer to the formalized arguments in appendix B.

Now the parameters to *Next()* make more sense. *Instate* remembers the state of *p* after reading some input text. *Char* is the next character to read. *TF* is TRUE if the start of *p* can be reached before reading *Char*.

The semantics of a state *q* are now understandable. A state is a record structure with several fields. The *state.inp* and *state.accepts* fields are Booleans. The *state.q1* and *state.q2* fields represent substates. For example, if *st* is a state of (SEQ *p1* *p2*), *st.q1* is a state of *p1* and *st.q2* is a state of *p2*. The *state.qseq* field holds a sequence of substates; *state.qseq.insize* is the number of elements in this sequence, *state.qseq[i]* is the *i*th substate (0 indexed) in *state.qseq*. The *state.q0* field holds an integer representing the state of an ANYSTR, EMPTY, or \{x\} pattern.

6.3. The Initial State

The correspondence proof in appendix B assumes an initial state with certain properties. Specifically:

- If *q* is the initial state for any pattern with subpatterns *p1* and *p2*, *q.q1* and *q.q2* must be the initial states of *p1* and *p2*.

- If *q* is the initial state of an ANYSTR, EMPTY, or \{x\} pattern, then *q.q0* must be 0 (with the intuition that no places are yet reachable).
If q is the initial state of a SET pattern, q.qseq must be the empty sequence. (For example, SET_LONGEST expects q.1.qseq to contain a state for each 0 ≤ i ≤ -1; no such i exists, so q.1.qseq should be empty.)

The “.inp” and “.accepts” fields of the states can be set arbitrarily in an initial state; notice that calculation of q_j from q_{j-1} never requires the values of q_{j-1}.inp or of q_{j-1}.accepts. The following function, InitialState(p), returns the initial state of pattern p which establishes the conditions listed above.
6.4. Using Next() to Match

We show explicitly how Next() can be used to calculate \{S\}..j matches p for an arbitrary set \{S\} of starting positions in input string s. Remember that \{1,4,99\}..16 matches p when s[1..16] matches p or s[4..16] matches p. The idea is to tell Next() that the first place in p is reachable at positions 1, 4, and 99, and to see if, as a result, the last
place in \( p \) is reachable at position 16. If \( q_{15} \) encodes the places reachable after reading the character at position 15, then

\[ q_{16} = \text{Next}(p, q_{15}, s_{16}, \text{TF}_{16}) \]

encodes the places reachable after reading the character at position 16, assuming that \( \text{TF}_{16} \) is TRUE iff the starting place of \( p \) can be reached at position 16. Thus, one can start with the initial state (call it \( q_{1} \)) and repeatedly calculate

\[ q_{j} = \text{Next}(p, q_{j-1}, s_{j}, (j \in \{1,4,99\})) \]

until \( q_{16} \) is calculated. \( q_{16}.\text{accepts} \) will be TRUE if the last place in \( p \) is reachable right before reading \( s_{16} \), which is what we wanted to know; in general, \( q_{j}.\text{accepts} \) is TRUE exactly when \( \{1.4.99\}..j \) matches \( p \). All of this is shown more formally in appendix B.

6.5. Worst-Case Complexity of Next()

The worst-case time and space requirements of Next() are analyzed in appendix B, but we summarize them here: when \( p \) does not use a SET_ pattern, Next(\( p, \ldots \)) requires time and space proportional to the size of \( p \). When \( p \) includes a SET_ pattern, Next(\( p, \ldots \)) may require super-exponential time and space. Fortunately, the optimizations discussed in section B.9 make Next() efficient for patterns that are usefully large, even if they include SET_ patterns.

6.6. Experimental Results for Next()

The previous section described the worst-case behavior of the unoptimized version of Next(). In practice, there are two different
types of optimizations that make Next() an efficient, practical
algorithm. Both are discussed in detail in section B.9; however, we
summarized the ideas here. The first, called lazy evaluation, uses a
cache of the results of recent calls to Next(). If a call to Next()
duplicates a previous call, the cached result is immediately returned.
In practice this duplication happens often, so this technique saves vast
amounts of time. The second optimization technique is to completely
precompile the Next() function into an efficient lookup table: in other
words, a DFA is created which corresponds to the pattern that Next()
represents.

This section presents experimental results that demonstrate the
behavior of the two types of optimized programs. In general, the lazy
evaluation method compiles faster than the DFA method, but the DFA
method matches faster.

For a number of patterns we will describe:

- how many states are discovered during the match of a long text.
  This gives an indication of a pattern’s complexity.

- matching time when using lazy evaluation with various cache
  sizes. All times are in seconds, and are calculated by summing
  the “user” and “system” times returned by appropriate calls to
  the Unix time functions. All experiments took place on a Sun
  4/390.

- matching time when using DFA search.

Table 6-1.-- Tested Patterns

<table>
<thead>
<tr>
<th>Name for</th>
<th>Description</th>
<th>What it Demonstrates</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatTlex</td>
<td>Matches a sequence of TLex patterns to a depth of 5. See bench3.tlx in appendix H.</td>
<td>Large, explosively recursive pattern. A grammar for a real language.</td>
</tr>
<tr>
<td>PatBBall</td>
<td>The third rule of the BBall application. See bench1.tlx in appendix H.</td>
<td>The largest pattern of a real application.</td>
</tr>
<tr>
<td>StringPat</td>
<td>(OR &quot;hello there&quot; &quot;happy day&quot;) See Ruleset 1 of bench2.tlx in appendix H.</td>
<td>Matching simple strings.</td>
</tr>
<tr>
<td>WholePat</td>
<td>Matches the whole input as an alternating sequence of words and non-words. See Ruleset 2 of bench2.tlx in appendix H.</td>
<td>A very long match. Requires a very large Parse Tree.</td>
</tr>
<tr>
<td>CCom1</td>
<td>A C-comment written as a restricted regular expression. See Ruleset 3 of bench2.tlx</td>
<td>CCom1, CCom2 are two different ways to write the same pattern.</td>
</tr>
<tr>
<td>CCom2</td>
<td>A C-comment written using SHORTEST. See Ruleset 4 of bench2.tlx</td>
<td>SHORTEST.</td>
</tr>
<tr>
<td>CCom3</td>
<td>A C-comment as the preferred match. See Ruleset 5 of bench2.tlx</td>
<td>Almost the same as CCom1 and CCom2. ANYSTR.</td>
</tr>
<tr>
<td>Short3</td>
<td>The shortest sequence of characters with 3 a’s and 3 b’s.</td>
<td>SHORTEST and AND.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Ruleset References</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Shorter3</td>
<td>Any sequence of characters with 3 a’s and 3 b’s, but prefer shorter sequences. See Ruleset 7 of bench2.tlx</td>
<td>SHORTER.</td>
</tr>
<tr>
<td>Long3</td>
<td>Longest sequence of characters without 3 a’s and without 3 b’s. See Ruleset 8 of bench2.tlx</td>
<td>LONGEST, ALL/MINUS</td>
</tr>
<tr>
<td>Longer3</td>
<td>Any sequence of characters without 3 a’s and without 3 b’s, but prefer the longer. See Ruleset 9 of bench2.tlx</td>
<td>LONGER, ALL/MINUS</td>
</tr>
<tr>
<td>SentPat</td>
<td>A sentence containing a word without vowels. See Ruleset 10 of bench2.tlx</td>
<td>Combining definitions with AND and ALL/MINUS. CSRPN.</td>
</tr>
<tr>
<td>MatchParen</td>
<td>Matching parentheses with a maximum recursion depth of 6. See Ruleset 11 of bench2.tlx</td>
<td>Non-explosive recursion.</td>
</tr>
</tbody>
</table>
Table 6-2 summarizes the results of matching using the lazy evaluation method. The cache size is the maximum number of transitions that can be cached.

Table 6-2.-- Matching Using Lazy Evaluation

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input Size</th>
<th>States &amp; Substate Created</th>
<th>Time with Cache Size 8192</th>
<th>Time with Cache Size 16,384</th>
<th>Time with Cache Size 32,768</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatTlex</td>
<td>49,760</td>
<td>57,824</td>
<td>161.8</td>
<td>46.4</td>
<td>31.4</td>
</tr>
<tr>
<td>PatBBall</td>
<td>195,000</td>
<td>24,972</td>
<td>235.4</td>
<td>49.9</td>
<td>26.3</td>
</tr>
<tr>
<td>StringPat</td>
<td>270,600</td>
<td>48</td>
<td>2.5</td>
<td>2.5</td>
<td>2.4</td>
</tr>
<tr>
<td>WholePat</td>
<td>270,600</td>
<td>96</td>
<td>4.2</td>
<td>4.3</td>
<td>4.2</td>
</tr>
<tr>
<td>CCom1</td>
<td>270,600</td>
<td>250</td>
<td>1.9</td>
<td>1.9</td>
<td>2.3</td>
</tr>
<tr>
<td>CCom2</td>
<td>270,600</td>
<td>101</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
</tr>
<tr>
<td>CCom3</td>
<td>270,600</td>
<td>34</td>
<td>2.0</td>
<td>2.2</td>
<td>1.9</td>
</tr>
<tr>
<td>Short3</td>
<td>270,600</td>
<td>2,948</td>
<td>2.3</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>Shorter3</td>
<td>270,600</td>
<td>2,988</td>
<td>2.3</td>
<td>2.3</td>
<td>2.4</td>
</tr>
<tr>
<td>Long3</td>
<td>270,600</td>
<td>3,024</td>
<td>2.3</td>
<td>2.3</td>
<td>2.4</td>
</tr>
<tr>
<td>Longer3</td>
<td>270,600</td>
<td>3,024</td>
<td>2.1</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>SentPat</td>
<td>270,600</td>
<td>12,182</td>
<td>6.4</td>
<td>5.3</td>
<td>5.0</td>
</tr>
<tr>
<td>MatchParen</td>
<td>270,600</td>
<td>1,750</td>
<td>2.5</td>
<td>2.1</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Notice that TLex is able to match a genuine TLex input file with a recursion depth of 5, albeit slowly. This is impressive, since PatTlex exhibits explosive recursion. Notice also that the beneficial effects of
caching are strikingly shown by PatBBall and PatTlex, because they are large patterns. The small patterns show little improvement due to caching because most or all of their transitions fit in the cache. Finally, notice that matching time increases for the more complicated patterns.

To demonstrate the speedup made possible by compiling the patterns into a DFA, using the Next() routine, we repeat the tests with fully compiled patterns.

Table 6-3.-- Matching Using a DFA

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input Size</th>
<th>States &amp; Substates Created</th>
<th>Transition Table Size</th>
<th>Compile Time</th>
<th>Matching Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatTlex</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PatBBall</td>
<td>195,000</td>
<td>366,466</td>
<td>551,819</td>
<td>6:15:56.6</td>
<td>.6</td>
</tr>
<tr>
<td>StringPat</td>
<td>270,600</td>
<td>56</td>
<td>179</td>
<td>.2</td>
<td>.7</td>
</tr>
<tr>
<td>WholePat</td>
<td>270,600</td>
<td>106</td>
<td>81</td>
<td>.1</td>
<td>.7</td>
</tr>
<tr>
<td>CCom1</td>
<td>270,600</td>
<td>303</td>
<td>189</td>
<td>.2</td>
<td>.7</td>
</tr>
<tr>
<td>CCom2</td>
<td>270,600</td>
<td>212</td>
<td>188</td>
<td>.2</td>
<td>.7</td>
</tr>
<tr>
<td>CCom3</td>
<td>270,600</td>
<td>44</td>
<td>85</td>
<td>.2</td>
<td>.7</td>
</tr>
<tr>
<td>Short3</td>
<td>270,600</td>
<td>10,258</td>
<td>6,235</td>
<td>1.6</td>
<td>.7</td>
</tr>
<tr>
<td>Shorter3</td>
<td>270,600</td>
<td>10,298</td>
<td>6,235</td>
<td>2.1</td>
<td>.7</td>
</tr>
<tr>
<td>Long3</td>
<td>270,600</td>
<td>10,334</td>
<td>6,235</td>
<td>2.3</td>
<td>.7</td>
</tr>
<tr>
<td>Longer3</td>
<td>270,600</td>
<td>10,334</td>
<td>6,235</td>
<td>2.4</td>
<td>.7</td>
</tr>
<tr>
<td>SentPat</td>
<td>270,600</td>
<td>32,822</td>
<td>15,721</td>
<td>23.8</td>
<td>.8</td>
</tr>
</tbody>
</table>
The compile time is the time to create and compact the DFA. The transition table size is the number of 32-bit words used to store the compacted DFA transitions. The compile times and match times are in seconds, except that the compile time for PatBBall is expressed as hour:min:seconds. The PatTlex pattern could not be successfully compiled because its equivalent DFA is too huge. (We stopped its compilation after it had created a huge number of states and used up hours of time). This is not entirely surprising, since it is a large pattern with explosive recursion, and we must use a recursion depth of 5. This failure illustrates the main limitation of using DFA matching.

Table 6-3 demonstrates that when compiled into a DFA, all patterns require time proportional to the length of the input to match, irrespective of the pattern’s complexity. The DFA matcher is anywhere from 4 to 50 times faster than the lazy evaluation matcher. For large patterns the compile time and space requirements can be excessive.
7. Parse Extraction

7.1. Listing of Parse()

function Parse(q, p, PP, n) returns N;
begin
  case p of
  EMPTY:
    N ♦ n;
  {x}:
    N ♦ n-1;
  ANYSTR:
    N ♦ n;
    until Proj(qN, p, PP).inp do
      N ♦ N-1;
  (SEQ p1 p2):
    N ♦ Parse(q, p1, PP, Parse(q, p2, PP, n));
  (OR p1 p2):
    if Proj(qN, p1, PP).accepts then
      N ♦ Parse(q, p1, PP, n);
    else
      N ♦ Parse(q, p2, PP, n);
  (* p1):
    N ♦ n;
    if (prefer more loops) then
      while Proj(qN, p1, PP).accepts do
        N ♦ Parse(q, p1, PP, N);
    else (prefer less loops)
      until Proj(qN, p, PP).inp do
        N ♦ Parse(q, p1, PP, N);
  (DEFCALL p1):
    N ♦ Parse(q, p1, PP, n);
  (MINI-NOT p1):
    N ♦ n;
    until Proj(qN, p, PP).inp do
      N ♦ N-1;
  (MINI-AND p1 p2):
    N ♦ Parse(q, p2, PP, n);
    N ♦ Parse(q, p1, PP, n);
  (CSLP p1 p2 p3):
    N ♦ Parse(q, p2, PP, n);
7. Parse()

\[(\text{SET\_SHORTEST } p1)\]:
\[(\text{SET\_SHORTER } p1)\]:
\[N \triangledown \text{Parse(\text{NewStates}(q, p, PP, n), pl, pl, n)};\]

\[\text{end case;}\]
\[\text{end;}\]

\[\text{function NewStates}(q, p, PP, n)\]
\[\text{returns a sequence of states } \text{qq;}\]
\[\text{begin}\]
\[\text{/** select the last state */}\]
\[qtemp \triangledown \text{Proj}(q_n, p, PP);\]
\[\text{if } p \text{ is (SET\_SHORTER } p1) \text{ or (SET\_SHORTEST } p1) \text{ then}\]
\[\text{lasti } \triangledown \text{the largest index } i \text{ such that}\]
\[qtemp.qseq[i].\text{accepts}\]
\[\text{else if } p \text{ is (SET\_LONGER } p1) \text{ or (SET\_LONGEST } p1) \text{ then}\]
\[\text{lasti } \triangledown \text{the smallest index } i \text{ such that}\]
\[qtemp.qseq[i].\text{accepts}\]
\[\text{qq}_n \triangledown qtemp.qseq[\text{lasti}]\]
\[\text{/** select preceding states */}\]
\[N \triangledown n;\]
\[\text{while } (\text{! qq}_N.\text{inp}) \text{ do}\]
\[\text{begin}\]
\[N \triangledown N - 1;\]
\[qtemp \triangledown \text{Proj}(q_N, p, PP);\]
\[\text{lasti } \triangledown \text{Back}(qtemp.qseq, \text{lasti});\]
\[\text{qq}_N \triangledown qtemp.qseq[\text{lasti}]\]
\[\text{end;}\]
\[\text{end;}\]

\[\text{function Back(Qseq, i) returns}\]
\[\text{(the index of the } i\text{th non-DEAD\_STATE in Qseq).}\]

\[\text{Notes: the notation } \text{“qq}_k \triangledown q2;” \text{ assigns } q2 \text{ to the } k\text{th}\]
\[\text{element of } \text{qq. } q.qseq \text{ and } \text{qq} \text{ are sequences of states; } q \text{ and}\]
\[\text{q2 are states; } PP, p, p1, \text{ and } p2 \text{ are patterns; } n, N, \text{ and}\]
\[\text{lasti are integer positions in the input}\]

7.2. Intuition of Parse()

Chapter 5 shows how the Next() algorithm can be used to find the
ending positions of all matches of some pattern p in a string s. This
section discusses what it means to **extract a parse** from some ending position.

A pattern can be thought of as encoding a set of possible paths.

\[(SEQ (* f) b (OR c d))\]

is a set of paths that go through \(f\) 0 or more times, then go through \(b\), and then go through \(c\) or \(d\). Extracting a parse of \(p\) ending at position \(n\), given a set of starting positions \(\{S\}\), means finding a path **backwards through** \(p\) ending at position \(n\) and starting at a position \(N\) in \(\{S\}\). We call this a backparse of \(p\) from \(n\) to \(N\).

The recursive function `Parse(q, p, PP, n)` solves this problem. This section presents an intuitive understanding of how `Parse()` works; a formal correctness proof is in appendix C. \(q\) is the sequence of states created in calculating \(\{S\}\). \(j\) matches \(PP\) for each \(j\) (as was discussed in section 6.2) in some input string \(s\); \(p\) is a subpattern of \(PP\); and \(n\) is a position at which the last place in \(p\) is reachable. `Parse()` returns an \(N\) for which the first place in \(p\) is reachable at position \(N\), and \(s[N..n]\) matches \(p\). While `Parse()` does not return the whole backparse through \(p\), it returns the ending position of such a backparse; furthermore, `Parse()` operates recursively so the whole backparse can be reconstructed if desired.

The simplest cases for `Parse()` are `EMPTY` and \(\{x\}\). `Parse()` returns \(N=n\) for `EMPTY` because if the end of `EMPTY` is reachable at position \(n\), then the start of `EMPTY` must also be reachable at position \(n\).
Similarly, if the end of \{x\} is reachable at position n, the start must be reachable at position n-1.

Parse() takes advantage of the recursive nature of a backparse. For example, to find a backparse of (SEQ p1 p2) from n, first find a backparse of p2 from n to some N’, and then find a backparse of p1 from N’ to some N; then N..n must be a backparse of (SEQ p1 p2).

Other patterns can be similarly analyzed recursively. However, the OR pattern illustrates a challenge: a backparse of (OR p1 p2) from n is either a backparse of p1 from n or a backparse of p2 from n. But which one? Instead of a potentially exponential search for a good backparse, we can use information discovered during matching to help make the correct choice. Specifically, if the last place of p1 is reachable at position n, then there is a backparse of p1 from n. Otherwise, there must be a backparse of p2 from n. Recalling the discussion of Next(), the last place of p1 is reachable at position n when \(q_n\).accepts is TRUE, where \(q_n\) is the state of p1 when PP is in state \(q_n\).

Actually, in several places Parse() uses the function Proj\((q_n, p, PP)\) to return the state of p when PP is in state \(q_n\). Parse() also uses the fact, from the discussion of Next(), that Proj\((q_n, p, PP)\).inp is TRUE when the start of p is reachable at position n. A formal definition for Proj() is in appendix C.

The SET_ cases, which also require the auxiliary procedure NewStates(), is rather unintuitive and tricky to understand. The basic idea is to recursively call Parse() changing PP to p1 and creating an
appropriate new sequence of states to replace q. The interested reader should refer to appendix C for a formal discussion.

7.3. Termination of (* p1)

Parse() actually gives two equally correct algorithms for (* p1). It often happens that there are multiple ways to backparse a pattern. Backparsing through (* p1) illustrates one of the choices: it may happen that a backparse can proceed back through p1, or leave (* p1) immediately. The former choice we characterize as “preferring more loops” and the latter choice the opposite. See section 4.4.1 for a related discussion.

The algorithm for (* p1) may not terminate, but this horrible possibility can easily be avoided or detected. The problem occurs if the call to Parse(S, p1, PP, n) returns an N=n. In this case, it is clear we have an infinite loop. Of course, if p1 cannot match the empty string, then the N returned will never equal n and the problem never arises. Here are two examples of patterns that might cause parse to loop forever:

(1) (* (OR EMPTY a))

(2) (* (** Digit))

This suggests the first method of avoiding infinite looping: check that p1 does not match the empty string. If it does not, then no infinite looping is possible while parsing p1. A similar method is to rewrite p1 as

(ALL p1 MINUS EMPTY)
Since

\((* \ p1)\) is exactly equivalent to \((* (\text{ALL} \ p1 \ \text{MINUS} \ \text{EMPTY}))\)

and since \((\text{ALL} \ p1 \ \text{MINUS} \ \text{EMPTY})\) does not match the empty string, this pattern will never cause an infinite loop. [Foster 1989] gives yet another algorithm for rewriting \((* \ p1)\) to prevent this problem.

7.4. Preferences in Parse1()

There are only four places in the function where Parse() may have a choice between different parses. The first place is in ANYSTR. The algorithm exits the ANYSTR as quickly as possible, which causes the ANYSTR to prefer shorter matches. The second place is in OR. Parse() may have a choice of backparsing through the p1 or p2 subpatterns. It prefers to match the p1 subpattern. The third place is in \((* \ p1)\) and has already been discussed. The fourth place is in SET_LONGER and SET_SHORTER, specifically in NewStates(). The arguments in appendix C show that the longest and shortest successful parses are preferred.

7.5. Building a Parse Tree

Although the Parse() function does a great deal of work, the reader may notice that it does not actually create a parse tree. This section sketches out how to actually create a parse tree using Parse(), at a user-definable level of detail. It should be emphasized that the TLex programmer never deals directly with the parse tree. Instead, it is
treated as an abstract type, and a number of convenient operators are provided that make it easy to use.

A parse tree at a user-definable level of detail may be an order of magnitude smaller than a full parse tree; TLex uses naming information (see section 4.4) to build only those parts of the parse tree that the user is interested in.

To integrate parse tree creation into Parse(), the simplest approach would be to add code at the end of Parse(). At that point, N and n have the respective starting and ending positions of the substring matching p, and a node of the parse tree can be built for p. Patterns have to be marked to indicate if a node should be constructed for it. The current TLex implementation actually surrounds each named pattern p with a special (BIND p) pattern. (BIND p) behaves just like p, but in addition stores information (such as the name of the operator) which is used to build a parse tree node.

Finally, it should be noted that creating a parse tree is just one of the possible ways to use Parse(). Another possibility is calling a set of actions in a bottom up fashion, as Yacc does.

7.6. Optimizing Parse()

Parse() is fairly simple and efficient, but the calls to Proj() can be optimized. Essentially, Proj() is calculated incrementally reusing information from previous calls. Section C.4 contains the gory details and the optimized Parse() function.
This section analyzes the worst case performance of the optimized version of Parse(). To be concrete, we will assume that a match of pattern p against a substring s produces a full parse tree PT, and we will calculate the worst case time of Parse() as a function of maxdepth(p), size(s), size(PT), and LL(PT). The maxdepth(p) function is the maximum depth of any pattern in p, when p is viewed as a tree, which is the maximum number of pattern nodes from the root of p to a leaf of p. The size(s) function returns the length of the string involved in the parse. For example, if the substring from positions 8..80 match a pattern, size(s) is 72. size(PT) is the number of nodes in the parse tree. The leaf length, LL(PT), is equal to the total number of characters matched by all leaf nodes in PT. One might think that this should equal size(s), since the parse tree just divides the characters up to show what substrings match what subpatterns. The culprit is MINI-AND. Each branch of MINI-AND may match the same substring in different ways. In a pattern including MINI-AND, LL(PT) ≠ size(s). In a pattern without MINI-AND, LL(PT) = size(s).

The absolute fastest possible time for parse extraction is the time to create PT, which is proportional to size(PT). Our time for Parse() will actually require an additional factor proportional to LL(PT), which corresponds to tracing through PT one character at a time.
Theorem 7.7: ParseTime:

The worst case time of Parse() to extract a full parse tree PT for a match of pattern PP against a substring s is

\[ O(\text{size}(PT)) + O(\text{maxdepth}(PP) \approx \text{LL}(PT)) \].

end Theorem.

The proof of this theorem is in section C.5.

Calculating the space usage is straightforward. It requires the definition of MNS(PP), which is the maximum nesting of SET_patterns within PP. For example, MNS(PP) is 2 if PP is the following pattern:

\[(\text{SET\_LONGER} \ (\text{SEQ} \ (* \ a) \ (\text{SET\_SHORTEST} \ (\text{OR} \ (* \ b) \ a))))\]

Theorem 7.7: ParseSpace:

The worst case space usage of Parse() to extract a full parse tree PT for a match of pattern PP against a substring s is

\[ O(\text{size}(PT)) + \text{size}(s) \approx \text{MNS}(PP) \].

end Theorem.

The proof of this theorem is in section C.5.

7.8. Experimental Results for Parse()

Because worst case analysis may not give an accurate picture of expected results, and to facilitate comparison with future algorithms, this section presents experimental results describing the actual behavior of Parse(). We use the exact same patterns and machine as were used in experimenting with Next(); see table 6-1 for a description of each pattern.
Here are the results. The “Total Match Length” is calculated by summing the lengths of each string matched by the pattern. Be aware that the “Total Parse Time” includes the time it takes the lazy evaluation version of TLex to scan the input for ending positions that match, in addition to the time spent in Parse(). The “Total Parse Time2” is the time it takes the DFA based searcher to scan the input for ending positions that match, plus the time spent in Parse(). The scanning is more efficient in the DFA based search because of some implementation details, but the Parse() time should be nearly the same in both cases. In other words, all of the improvement of “Total Parse Time2” over “Total Parse Time” is due to faster scanning. Therefore, the “Total Parse Time2” gives a better indication of Parse() behavior, while “Total Parse Time” gives a better indication of the expected behavior of the whole parsing process when lazy evaluation is used. Also note that the behavior of Parse() is unrelated to the cache size used in matching.
We can observe that Parse() times are reasonably fast. Also, the times are generally proportional to the total match length (with an appropriate fudge factor for the depth or complexity of the pattern). As we would expect, they are relatively insensitive to the length of the input (compare CCom3 and Short3) and the number of matches (compare Long3 and Short3).
8. Conclusions and Future Work

The thesis has presented efficient algorithms for matching extended regular expressions and extracting a parse tree from a successful match. As a result, it is now possible to use extended regular expressions in the myriad applications already using restricted regular expressions. The extended regular expression language provides a quantum increase in expressiveness over restricted regular expressions. Parse extraction provides a quantum increase in the potential for making use of a successful match. We have demonstrated the practicality and utility of these algorithms by designing and implementing TLex, a language for efficiently extracting information out of large inputs. TLex has been successfully used in a number of real-life applications.

One way to visualize how the algorithms of the thesis compare to previous pattern matching algorithms is with figure 8-1.

Figure 8-1: Locating TLex

The ultimate goal of a pattern matching tool is an extremely expressive pattern language that can be matched and parsed in linear time. Current tools offer a spectrum of solutions that define an efficiency/expressiveness tradeoff (the dashed line). Icon is at one extreme, favoring expressiveness over efficiency. It offers a full programming language for defining patterns, and it searches for
patterns using recursive backtracking. TLex now anchors the other extreme, favoring efficiency over expressiveness. While its pattern language cannot compare to that of Icon, the TLex language is more expressive than all the other tools (excepting the lack of true recursion). In addition the TLex pattern language has the distinct advantage of being specified declaratively, in comparison to the procedural specifications of Icon. Best of all, the algorithms in TLex offer nearly optimal matching and parsing speed. An analogy to cars seems apt: TLex is the most luxurious car you can buy and still be able to drive at 200 mph.

8.1. Specific Contributions of the Thesis

Although it was apparent from 1975 (when Lex was created) that the LONGEST pattern and right context were very useful in writing patterns, this thesis is the first realization that these operators can be implemented as general regular expression operators, as general as SEQ, OR, and *.

The trick of reversing the pattern and input, which effectively turns lookahead into lookbehind and thus makes it possible to implement LONGEST and right context patterns, is a contribution which promises to have applicability to other pattern matching algorithms. For example, it can be applied to various context free recognizers. Specifically, any language closed under reversal is a candidate for this innovation.
8. Conclusions and Future Work

[Lipson 1984] set out to design the best possible pattern specification language. However, Lipson did not suggest how the language might be efficiently implemented. This thesis identifies and shows how to implement a significant subset of Lipson’s ideal language with nearly optimal efficiency.

An extensive formalization of pattern matching appears in the thesis. This is a contribution in itself, as it is quite likely that concepts like Prefix() and Postfix() will have wider application.

The idea of creating a parse tree from a match is an improvement over the SNOBOL4, Icon, and CPDS idea of Immediate Assignment, when the goal is to extract information from a match. A parse tree can be stored in a single variable; parts of the tree which are unneeded for a certain application can be ignored. In contrast, using Immediate Assignment one must explicitly assign each potentially desired piece of information to a different string variable. The parse tree abstraction can probably be used in languages that currently use Immediate Assignment for extracting information from a match.

We describe many practical optimizations that enable the Next() and Parse() algorithms to be practical and convenient. Furthermore, we have actually constructed the optimized algorithms and shown their practical application.

This thesis is the first to realize that it is possible to extract a parse from a match of a string to a finite automata, and it gives an efficient
and elegant algorithm which does so. This fundamental contribution permits fast matching and convenient use of the resulting matches.

## 8.2. Future Research

Parse extraction can also be applied to some of the previous algorithms for matching restricted regular expressions. If the subset construction is used to compile the restricted regular expressions, the parse extraction algorithm in the thesis can easily be adapted.

The VLSI circuits in [Wakabayashi et al. 1984] illustrate a technique which could be incorporated into Next() to permit superior parallelization. The circuits utilize a “skip” line which can be easily translated into a software algorithm. In fact, the first version of the Next() algorithm worked essentially this way. It was changed to the current design because few computers are parallel, and the current design is simpler to describe than the previous. In general, parallel regular expression search is unexplored, as is parallel compilation of regular expressions. Both subjects deserve increased attention in the future.

The TLex pattern language could be improved with a more advanced macro capability. It has proven to be one of the most useful organizing constructs available to the TLex programmer. The interaction of macros with naming especially needs to be investigated.

The current implementations of the TLex search engine offers two modes of searching. It uses lazy transition evaluation (see section
8. Conclusions and Future Work

B.9.5) by default, and DFA compilation (see section B.9.4) as an option. Lazy transition evaluation provides fast compilation and efficient memory use at the expense of slower execution. One implication of this is that the time to search the first few characters can be orders of magnitude slower than the time to search following characters. Future versions of TLex should permit a ruleset to be “trained” by running it on sample input and then saving the resulting state tables and transition caches. This third mode will provide faster initial speed since the caches and the state table will be initially filled with the most important data. As a result, the programmer will be able to trade-off compilation time, execution time, and storage requirements along a much greater range than possible with the two existing modes.

RLex is a pattern matching library being developed for C and C++ programmers. It contains essentially the same pattern language and parse tree accesses as TLex, without the built in control structure. RLex is intended for all the “little” parsing tasks programmers face. The intended applications include parsing file names, network addresses, data files, and command lines. It uses a compact recursive backtracking method to search for the patterns; consequently, the RLex compiler comprises only 15k of code, and the RLex runtime search engine is of a similar size. For added convenience, RLex can be integrated into a library of string operations. In C++, the combination is very powerful.

The most useful addition to the TLex pattern matching language would be left context. If one could figure out how to efficiently match
patterns with both right and left context, simultaneously, than the era of truly reusable patterns would be here.

It might be possible to implement Parse() even more efficiently. A cursory analysis suggests that Parse() can be thought of as a finite automata whose input text is the sequence of DFA states found in matching the pattern, and whose states represent positions in the pattern being parsed. This suggests that it might be possible to implement Parse() to run in linear time; however, much work needs to be done to see if the resulting automata is small enough to be feasibly calculated and stored.
Bibliography


R. Pike: The Text Editor Sam, Software - Practice and Experience, 17(11), November 1987, pp. 813-845.


K. Thompson: Regular Expression Search Algorithm, Comm. ACM 11(6), 1968, pp. 419-422.


Appendix A
The Matching Formalism

In preparation for a formal discussion of the correctness of Next() and Parse() algorithms, this chapter formalizes the meanings of the myriad patterns.

A.1. Formally Defining Patterns

Readable definitions of TLex patterns were given in chapter 4. Here we formally define the patterns. Consider a finite input string which is a sequence of tokens. The tokens are drawn from a finite, fixed alphabet. A “pattern” is defined as a set of substrings of the input string. A substring “matches” a pattern iff it is in the set of substrings that define the pattern. A “regular expression” is one way to specify a pattern. We will use the word “pattern” interchangeably with the phrase “regular expression”.

Normally, the alphabet is the ASCII character set, the input string is a text file to be searched, and examples of patterns include “a line of the input file”, “any word of the input file”, “an occurrence of the word JOE in the input file”, and “the first decimal number in the input file”.

Our formalization contains a subtle difference from the way regular expressions are usually introduced. Traditionally a pattern is considered to be a set of strings; instead, we have defined a pattern to be a set of substrings which must be drawn from a given input string. In our formalism it makes sense to define the pattern “the last word in
Appendix A: The Matching Formalism

the input string”; in the traditional formalism this cannot be specified. Similarly, the LONGEST, CTRP, and CSRP patterns, discussed below, make sense in our formalism but not the traditional.

When we talk about substrings, we will refer to “positions” in the input string; these fall BETWEEN characters of the string. A substring i..j of input string s is the sequence of characters between positions i and j. Position i comes before the ith character, where the first position in the string is 0 (and the first character in the string is called the zeroth). Thus i..i is an empty substring for all i, and a one character substring i..i+1 is just the ith character, also written si. The substring 0..n is equal to the whole string s which is assumed to have n-1 characters. The concatenation of two substrings i..j and j..m, written i..j;j..m, is just i..m. Two substrings i..j and k..m can only be concatenated if j = k.

Later we introduce context sensitive patterns, whose definition refers to position “n”, the last position in the input. When we say “i..j matches p” we really mean i..j is in pattern p for the input string 0..n. A pattern p will be defined by giving a rigorous definition for “i..j matches p”.

About notation: regular expressions are written in “LISP notation”; this makes it easy to add new language constructs and emphasizes their recursive nature. Theorems and Definitions are numbered with the section they appear in, followed by a descriptive name. Thus “Definition 7.2:Dogs” refers to the definition in section 7.2 labeled “Dogs”.
Regular expressions are recursively built from the following formally defined operators.

**Definition A.1: Pattern Operators:**

- \( i..j \) matches \((\text{SEQ } p1 \ p2) / \exists k: i \leq k \leq j: (i..k \) matches \(p1 \) and \( k..j \) matches \(p2)\)
- \( i..j \) matches \((\text{OR } p1 \ p2) / (i..j \) matches \(p1 \) or \( i..j \) matches \(p2)\)
- \( i..j \) matches \((\text{OR } p1) / (i=j) \) or \( \exists k: i \leq k < j: (i..k \) matches \((\text{OR } p1) \) and \( k..j \) matches \(p1)\)
- \( i..j \) matches \(\text{EMPTY} / i=j\)
- \( i..j \) matches token \(x / (i+1 = j) \) and \( s_i = x \)
- \( i..j \) matches token \(x / (i+1 = j)\)
- \( i..j \) matches \(\text{ANYSTR} / \text{TRUE}\).

The next set of operators are convenient additions to a pattern language; however, they can all be mechanically translated into simple combinations of the previous operators.

- \( i..j \) matches \((\text{NOTCLASS } a \ b \ z) / (j-i = 1) \) and \( s_i = a \) and \( s_i = b \) and \( s_i = z \)
- \( i..j \) matches \((\text{FROM } a \ TO \ z) / (j-i = 1) \) and \( s_i = a \) and \( s_i \leq z \)
- \( i..j \) matches \((\text{+ } p1) / i..j \) matches \((\text{SEQ } p1 \ (\text{+ } p1))\)
- \( i..j \) matches \(\text{Foo where Foo is defined as} / (\text{DEFINE } Foo \ p1) / i..j \) matches \(p1\)
- \( i..j \) matches \(\text{"anything"} / i..j \) matches \((\text{SEQ } a \ n \ y \ t \ h \ i \ n \ g)\)

Technically, the next set of pattern operators only increase the power of the language to be equivalent to the previous operators plus the END operator (which matches the end of the input). However, using the following operators one can write patterns that are far too complicated for humans to write using only the basic operators plus END. Thus, the following operators substantially increase the expressiveness of the pattern language.

- \( i..j \) matches \((\text{AND } p1 \ p2) / (i..j \) matches \(p1 \) and \( i..j \) matches \(p2)\)
Appendix A: The Matching Formalism

- i..j matches (ALL p1 MINUS p2) / i..j matches p1 and not (i..j matches p2).

- i..j matches (SHORTEST p1) / (i..j matches p1) and \( \forall k: i < k < j: \text{not (i..k matches p1)} \)

- i..j in 0..n matches (LONGEST p1) / (i..j matches p1) and \( \forall k: j < k < n: \text{not (i..k matches p1)} \)

- i..j matches (SHORTER p1) / i..j matches p1.

- i..j matches (LONGER p1) / i..j matches p1.

The formalism does not capture all of the semantics of SHORTER or LONGER, which is that (SHORTER p1) matches the shortest string matching p1 that permits the rest of the pattern to match. (SHORTER p1) and p1 match exactly the same thing; however, during parse extraction we extract the shortest possible match that permits the rest of the pattern to match.

- i..j in 0..n matches (CSRP p left p right) / (i..j matches p) and \( \exists k: j \leq k \leq n: (j..k matches p right) \) and i-1..i matches p left and p left is a set of characters

- i..j in 0..n matches (CSRPN p left p right) / (i..j matches p) and \( \forall k: j \leq k \leq n: (not j..k matches p right) \) and i-1..i matches p left and p left is a set of characters

- i..j in 0..n matches (CTRP p left p right) / (i..j matches p) and \( \exists k: i \leq k \leq n: (i..k matches p right) \) and i-1..i matches p left and p left is a set of characters

- i..j in 0..n matches (CTRPN p left p right) / (i..j matches p) and \( \forall k: i \leq k \leq n: (not i..k matches p right) \) and i-1..i matches p left and p left is a set of characters

- i..j in 0..n matches (CSRP p right) / i..j matches (CSRP any p right)

- i..j in 0..n matches (CSRPN p right) / i..j matches (CSRPN None p right)
  (None is the empty set of characters)

- i..j in 0..n matches (CTRP p right) / i..j matches (CTRP any p right)

- i..j in 0..n matches (CTRPN p right) / i..j matches (CTRPN None p right)

end Definition.
A.2. Defining More Patterns

In Definition A.1:PatternOperators we defined each pattern \( p \) as a predicate

\[
(i..j \text{ matches } p)
\]

We actually need a more general notion:

\[
({S})..j \text{ matches } p).
\]

\( {S} \) is a set of starting positions in a string \( s \) (which has positions from 0 to \( n \)), \( j \) is an ending position, and the elements of \( {S} \) range from 0..\( n \), but only the values in \( {S} \) which are \( \leq j \) matter. The next definition shows how to extend the previous definitions to this more general notion. Intuitively, \( \{1,4,9,99\}..16 \) matches \( p_1 \) iff \( 1..16 \) matches \( p_1 \), \( 4..16 \) matches \( p_1 \), or \( 9..16 \) matches \( p_1 \).

**Definition A.2:IsExistential:**

\( p \) is Existential iff it can be defined as a predicate \( i..j \text{ matches } p \).

*end Definition.*

Each of the patterns introduced in Definition A.1:PatternOperators is an Existential pattern. Furthermore, Definition A.1:PatternOperators mainly defines \( i..j \text{ matches } p \) in terms of \( x1..x2 \text{ matches } p_1 \) and \( x3..x4 \text{ matches } p_2 \), so each of the previous operators expects its subpatterns to be Existential. For an Existential pattern \( p \), we can define \( {S}..j \text{ matches } p \) in a way that matches the intuitive meaning:
Definition A.2: Existential:
Assuming p is Existential; then

\{S\}..j matches p / 3i:i in \{S\}:(i..j matches p).

end Definition.

If a pattern p is not Existential then i..j matches p is not defined; obviously, Definition A.2: Existential would not make any sense in this case. We will use two pattern operators which are not Existential to implement the TLex patterns. We define them now:

Definition A.2: NonExistentialPatterns:

- \{S\}..j matches (MINI-AND p1 p2) / \{S\}..j matches p1 and \{S\}..j matches p2
- \{S\}..j matches (MINI-NOT p1) / not \{S\}..j matches p1

end Definition.

There are a few Existential patterns which are not available to the pattern programmer but are used in implementing the previous patterns. We define them here for completeness. They are basically the mirror images of the operators LONGEST, SHORTEST, CSRNP, and CSRPN.

Definition A.2: OtherPatterns:

- i..j in 0..n matches (CSLP pleft p pright) / (i..j matches p) and 3k:0≤k≤i:(k..i matches pleft) and j..j+1 matches pright and pright is a set of characters
  CSLP is the Context Sensitive Left-Prefix pattern.
- i..j in 0..n matches (CSLPN pleft p pright) / (i..j matches p) and 3k:0≤k≤i: not(k..i matches pleft) and not j..j+1 matches pright and pright is a set of characters
  CSLPN is the Context Sensitive Left-Prefix-Not pattern.
Appendix A: The Matching Formalism

- $i..j$ in $0..n$ matches (CTLP $p_{\text{left}}$ $p$ $p_{\text{right}}$) / $i..j$ matches $p$ and $\exists k:0\leq k\leq j:(k..j$ matches $p_{\text{left}}$) and $j..j+1$ matches $p_{\text{right}}$ and $p_{\text{right}}$ is a set of characters

CTLP is the Context Total Left-Prefix pattern.

- $i..j$ in $0..n$ matches (CTLPN $p_{\text{left}}$ $p$ $p_{\text{right}}$) / $i..j$ matches $p$ and $\forall k:0\leq k\leq j:$(not $k..j$ matches $p_{\text{left}}$) and $j..j+1$ matches $p_{\text{right}}$ and $p_{\text{right}}$ is a set of characters

CTLPN is the Context Total Left-Prefix-Not pattern.

- $i..j$ matches (SET_SHORTER $p_1$) / $\{i\}..j$ matches $p_1$

- $i..j$ matches (SET_LONGER $p_1$) / $\{i\}..j$ matches $p_1$

- $i..j$ matches (SET_SHORTEST $p_1$) / $\{i\}..j$ matches $p_1$ and not $\exists k:i<k\leq j:($\{$k\}..j$ matches $p_1$)$

- $i..j$ matches (SET_LONGEST $p_1$) / $\{i\}..j$ matches $p_1$ and not $\exists k:0\leq k<i:($\{$k\}..j$ matches $p_1$)$

end Definition.

All of these patterns are Existential. The SET_ patterns do not assume that their subpatterns are Existential.

A.3. Simplifying the Problem

As described in section 5.5, the first stage involves reversing the patterns. This section proves the translation process from the general TLex language to simpler, reversed set of patterns described in section 5.5

A.3.1. Proof of Theorem 5.5.1:Reverse

For a sample proof we will show that the formula for SEQ is correct.
Appendix A: The Matching Formalism

i..j matches (SEQ p1 p2) in s

// { def of SEQ }
∃k:(i..k matches p1 in s and k..j matches p2 in s)

// { def of rev }
∃k:(n-k..n-i matches (rev p1) in s_{rev} and
  n-j..n-k matches (rev p2) in s_{rev} )

// substitute n-k for k
∃(n-k):(k..n-i matches (rev p1) in s_{rev} and
  n-j..k matches (rev p2) in s_{rev} )

// Existence of n-k is Existence of k
∃k:(k..n-i matches (rev p1) in s_{rev} and
  n-j..k matches (rev p2) in s_{rev} )

// { def of SEQ }
 n-j..n-i matches (SEQ (rev p2) (rev p1)) in s_{rev}

For another proof we show that the formula for (rev (CSRPN pleft p pright)) is correct.

i..j matches (CSRPN pleft p pright) in s

// { def of CSRPN }
i..j matches p in s and ∀k:j≤k≤n:(not j..k matches pright in s) and
i-1..i matches pleft in s

// { def of rev }
n-j..n-i matches (rev p) in s_{rev} and
 ∀k:j≤k≤n:(not n-k..n-j matches (rev pr) in s_{rev} ) and
n-i..n-i+1 matches (rev pleft) in s_{rev}
Appendix A: The Matching Formalism

// { multiply j≤k≤n by -1 and add n to all sides }

n-j..n-i matches (rev p) in srev and
∀k: n-j..n-k..0: (not n-k..n-j matches (rev pr) in srev)
n-i..n-i+1 matches (rev pleft) in srev

// { substitute n-k for k }

n-j..n-i matches (rev p) in srev and
∀(n-k): n-j..k..0: (not k..n-j matches (rev pr) in srev)
n-i..n-i+1 matches (rev pleft) in srev

// { simplify }

n-j..n-i matches (rev p) in srev and
∀k: 0≤k≤n-j: (not k..n-j matches (rev pr) in srev)
n-i..n-i+1 matches (rev pleft) in srev

// { def of CSLPN }

n-j..n-i matches (CSLPN (rev pright) (rev p) (rev pleft)) in srev

A.3.2. Proof of Theorem 5.5.2: Subpatterns

In order to prove the remaining equalities, we will actually show that

{S}..j matches P1 / {S}..j matches P2

where P1 is the pattern on the left and P2 is the pattern on the right.

Proof of Theorem 5.5.2: Subpatterns: (7):

{S}..j matches (SEQ p (CSLP pleft EMPTY {x}))

// { Theorem B.1: RecursiveMatch for SEQ }

{i | {S}..i matches p}..j matches (CSLP pleft EMPTY {x})
Appendix A: The Matching Formalism

/\ Theorem B.1:RecursiveMatch for CSLP /

\[ i | \{S\}.i \text{ matches } p \text{ and } \{\text{ALL}\}.i \text{ matches } p \text{left}.j \text{ matches }\]
\[ \text{EMPTY} \]
and \ j..j+1 \text{ matches } \{x\}

/\ Theorem B.1:RecursiveMatch for EMPTY /

\[ \{S\}.j \text{ matches } p \text{ and } \{\text{ALL}\}.j \text{ matches } p \text{left} \text{ and } j..j+1 \text{ matches } \{x\} \]

/\ Theorem B.1:RecursiveMatch for CTLP /

\[ \{S\}.j \text{ matches } (\text{CTLP } p \text{left }{x})\]

end Proof.

Proof of Theorem 5.5.2:Subpatterns:(8):

\[ \{S\}.j \text{ matches } (\text{CTLP } (\text{MINI-NOT } p \text{left}) p \text{ (NOTCLASS } {x}))\]

/\ Theorem B.1:RecursiveMatch for CTLP /

\[ \{S\}.j \text{ matches } p \text{ and } \{\text{ALL}\}.j \text{ matches } (\text{MINI-NOT } p \text{left}) \text{ and } \]
\[ j..j+1 \text{ matches } (\text{NOTCLASS } {x})\]

/\ Theorem B.1:RecursiveMatch for MINI-NOT, def of NOTCLASS /

\[ \{S\}.j \text{ matches } p \text{ and not } \{\text{ALL}\}.j \text{ matches } p \text{left} \text{ and } \]
not \ j..j+1 \text{ matches } \{x\}

/\ Theorem B.1:RecursiveMatch for CTLPN /

\[ \{S\}.j \text{ matches } (\text{CTLPN } p \text{left }{x})\]

end Proof.

It is intended that Theorem 5.5.2:Subpatterns:(7) be applied after
Theorem 5.5.2:Subpatterns:(8), so that a CTLPN can be converted to a
CSLP. The proof of Theorem 5.5.2:Subpatterns:(9) is exactly like the
proof of Theorem 5.5.2:Subpatterns:(8).
Proof of Theorem 5.5.2:Subpatterns:(10):

{S}..j matches (SET_LONGER (MINI-AND p1 p2))

/* Theorem B.1:RecursiveMatch for SET-LONGER */

\exists i : i in \{S\} : {i}..j matches (MINI-AND p1 p2)

/* Theorem B.1:RecursiveMatch for MINI-AND */

\exists i : i in \{S\} : {i}..j matches p1 and {i}..j matches p2

/* Theorem B.1:RecursiveMatch for AND */

{S}..j matches (AND p1 p2)

end Proof.

The proof of Theorem 5.5.2:Subpatterns:(11) is exactly like the proof of Theorem 5.5.2:Subpatterns:(10). It might seem, intuitively, that (AND p1 p2) could be simply translated as (MINI-AND p1 p2). An example disproves the intuition: on the input “cat”, \{0,1\}..3 matches (MINI-AND (SEQ c a t) (SEQ a t)), but \{0,1\}..3 does NOT match (AND (SEQ c a t) (SEQ a t)). Of course, no string can match both “cat” and “at” at the same time.

Theorem 5.5.2:Subpatterns:(12) says that we translate the name of a pattern definition into (DEFCALL p) where p is the body of the pattern definition. For example, if Digit is defined by (DEFINE Digit (from '0' to '9')), then the pattern

(SEQ Digit '.' Digit)

becomes:

(SEQ (DEFCALL (from '0' to '9'))

' '

(DEFCALL (from '0' to '9')))
Recursive pattern definitions are repeatedly substituted in to a fixed depth, and then the recursive definition is replaced by {}, the set of no characters, which no string matches. The next section describes how patterns can be represented as a directed graph to maximize sharing of patterns (and eventually transitions).

Converting Theorem 5.5.2:Subpatterns:(13) into words, it says the shortest string matching p1 is a string which matches p1 but which no prefix shorter by 1 or more characters matches p1.

A.3.3. Representing a Pattern

By representing patterns as directed graphs, subpatterns which look the same can be represented by the same graph. This saves pattern storage space and makes bounded recursion easier to implement. We introduce the pattern representation here because it is crucial to understanding how later algorithms treat recursive definitions.

A pattern is actually represented as a directed graph, each node of which represents a pattern: it is a record which contains the type of the pattern, any pattern specific data, and potentially pointers to subpatterns. Here is how (SEQ a (OR b c)) appears:

Figure A-1: A Pattern
Subpatterns which appear more than once can be represented once, with several nodes pointing to it. For example, the pattern 
\((\text{SEQ} (\text{OR} \ a \ b) (\text{OR} \ a \ b) \ b)\) can be represented like this:

![Figure A-2: Compacting a Pattern](image)

In particular, each pattern definition is only represented once. Instead of copying the body of the pattern definition, as suggested previously, all DEFCALLs for a definition “Foo” actually point to the single representation of the body of “Foo”. The pattern 
\((\text{SEQ} \ \text{Digit} \ \text{Period} \ \text{Digit})\) would appear as follows:

![Figure A-3: Representing Definitions](image)

Recursive or mutually recursive pattern definitions are represented as a cyclic graph. For example, the recursive pattern definition for a sequence of matched parentheses,

\[
(\text{DEFINE} \ \text{MatchParen} \ (\text{SEQ} \ ' (' \ (\text{OR} \ \text{EMPTY} \ \text{MatchParen}) ') ')'))
\]

is actually represented like this:

![Figure A-4: Representing Recursion](image)

The RecCount field in each pattern definition is used to implement bounded recursion. All routines that traverse the pattern definition
increment RecCount upon entering the pattern and decrement RecCount upon exiting. If RecCount exceeds some threshold (MAX_RECURSION), then the routine treats the pattern as {}, which does not match any string. There will be examples of this technique in the Next(), InitialState(), and Parse() procedures given later.

A.4. Prefix() and Postfix()

Though not strictly necessary for understanding Next(), the Prefix() and Postfix() functions are crucial in understanding Parse(). This section introduces them and derives a number of useful properties.

Consider the pattern

\[(\text{SEQ } (\text{OR } x \; (^* y)) \; z)\]

It is natural to think of a regular expression as listing a set of paths. The sample pattern specifies all those paths which go through x and then z, plus those paths which go through y zero or more times and then go through z. The path concept is useful because if you write down, in order, the letter patterns that lie on any path, the resulting string will be a match to the regular expression. For example, the path that goes through y 3 times and then goes through z corresponds to the string “yyyyz” which indeed matches the regular expression.

Given the idea of a path, the Prefix() and Postfix functions arise naturally. Prefix(p1, PP) assumes that p1 is some subpattern of PP, and it returns a regular expression denoting all paths from the start of PP to the start of p1. Postfix(p1, PP) returns a regular expression
denoting all paths from the start of \( PP \) to the **end** of \( p_1 \). Here are some examples:

\[
PP = (SEQ \ z \ (OR \ x \ (* \ y)))
\]

- \( \text{Prefix}(z, PP) = \text{EMPTY} \)
- \( \text{Postfix}(z, PP) = z \)
- \( \text{Prefix}(x, PP) = z \)
- \( \text{Postfix}(x, PP) = (SEQ \ x \ z) \)
- \( \text{Prefix}(y, PP) = (SEQ \ z \ (* \ y)) \)
- \( \text{Postfix}(y, PP) = (SEQ \ (SEQ \ z \ (* \ y)) \ y) \)
- \( \text{Prefix}((OR \ x \ (* \ y)), PP) = z \)
- \( \text{Prefix}(PP, PP) = \text{EMPTY} \)
- \( \text{Postfix}(PP, PP) = PP \)

It must be stressed that \( p_1 \) is a **specific** subpattern of \( PP \): in the following pattern, what should \( \text{Prefix}(y, PP) \) return?

\[
PP = (SEQ \ (OR \ y \ (* \ y)) \ y)
\]

The answer is that we should have specified exactly which subpattern is being referred to, for example by using subscripts. So we should have asked for \( \text{Prefix}(y_3, PP) \), where

\[
PP = (SEQ \ (OR \ y_1 \ (* \ y_2)) \ y_3)
\]

Like all good functions, \( \text{Prefix}() \) can be specified recursively. We say that \( \text{parent}(p) \) is the specific pattern that contains \( p \) as an immediate subpattern. Thus \( \text{parent} \ (y_1) \) in the last \( PP \) is \( (OR \ y_1 \ (* \ y_2)) \), and the \( \text{parent}() \) of the latter is \( PP \).

**Definition A.4:** Prefix:

Assuming that \( p_1 \) is a subpattern of \( PP \); then

\( \text{Prefix}(p_1, PP) \) is \( \text{EMPTY} \) when \( p_1 = PP \).
Appendix A: The Matching Formalism

Prefix(p1, PP) is Prefix(parent(p1), PP)
    when parent(p1) = (SEQ p1 p2), (OR p1 p2), (OR p2 p1),
    (MINI-AND p1 p2), (MINI-AND p2 p1), (DEFCALL p1),
    (MINI-NOT p1), (SET_ p1).

Prefix(p1, PP) is (SEQ Prefix(parent(p1), PP) p2)
    when parent(p1) is (SEQ p2 p1).

Prefix(p1, PP) is (SEQ Prefix(parent(p1), PP) parent(p1))
    when parent(p1) is (* p1).

Prefix(p1, PP) is (CSLP pleft Prefix(parent(p1), PP) any)
    when parent(p1) is (CSLP pleft p1 {x}).

(Notice that this is an inductive definition which goes from patterns to the patterns that contain them. Thus the base case is Prefix(PP, PP).)

end Definition.

Postfix() can be defined in terms of Prefix(). It is intuitive that any path to the end of a pattern first goes to the start of the pattern and then goes through the pattern.

Definition A.4: Postfix:

Assuming that p1 is a subpattern of PP; then

Postfix(p1, PP) is (SEQ Prefix(p1, PP) p1)

end Definition.

Following sections mainly refer to the following lemma.

Lemma A.4: FixFormula

1. Prefix((DEFCALL p1), PP) = Prefix(p1)
2. {S}..j matches Postfix((DEFCALL p1), PP)
   / {S}..j matches Postfix(p1, PP)
   if RecCount((DEFCALL p1)) < MAX_RECURSION
   / FALSE otherwise
3. in (SEQ p1 p2), Prefix(p2, PP) = Postfix(p1, PP)
4. Prefix(p1, PP) = Prefix((SEQ p1 p2), PP)
5. Postfix(p2, PP) = Postfix((SEQ p1 p2), PP)
Appendix A: The Matching Formalism

(6) \text{Prefix((OR p1 p2), PP)} = \text{Prefix(p1, PP)} = \text{Prefix(p2, PP)}

(7) \{S\}..j matches Postfix((OR p1 p2), PP) / 
    \{S\}..j matches Postfix(p1, PP) or 
    \{S\}..j matches Postfix(p2, PP)

(8) \text{Prefix((MINI-AND p1 p2), PP)} = \text{Prefix(p1, PP)} 
    = \text{Prefix(p2, PP)}

(9) \{S\}..j matches Postfix((MINI-AND p1 p2), PP) / 
    \{S\}..j matches Postfix(p1, PP) and 
    \{S\}..j matches Postfix(p2, PP)

(10) \text{Prefix(p1, PP)} = \text{Postfix(* p1), PP)}

(11) \{S\}..j matches Prefix(* p1), PP) _ 
    \{S\}..j matches Prefix(p1, PP)

(12) \{S\}..j matches Postfix(* p1), PP) / 
    \{S\}..j matches Prefix(* p1), PP) or 
    \{S\}..j matches Postfix(p1, PP)

(13) \text{Prefix(EMPTY, PP)} = \text{Postfix(EMPTY, PP)}

(14) \text{Prefix((MINI-NOT p1), PP)} = \text{Prefix(p1, PP)}

(15) \{S\}..j matches Postfix((MINI-NOT p1), PP) / 
    \text{not } \{S\}..j matches Postfix(p1, PP)

(16) \{S\}..j matches Prefix(p2, PP) _ 
    \{S\}..j matches Prefix((CSLP p1 p2 \{x\}), PP)

(17) \{S\}..j matches Postfix((CSLP p1 p2 \{x\}), PP) _ 
    \{S\}..j matches Postfix(p2, PP)

end Lemma.

Proof:

All rules involving Postfix(p, PP) rules can be proven by substituting 
the equivalent (SEQ Prefix(p, PP) p) and then using Definition 
A.4:Prefix and Theorem B.1:RecursiveMatch. All rules involving 
Prefix() can be proven by applying the Definition A.4:Prefix as much as 
possible, and then applying Theorem B.1:RecursiveMatch. As an 
example we prove Lemma A.4:FixFormula(7):
Appendix A: The Matching Formalism

\{S\}..j matches Postfix((OR p1 p2), PP) / 

\{S\}..j matches Postfix(p1, PP) or \{S\}..j matches Postfix(p2, PP)

/ \{ def of Postfix \}

\{S\}..j matches (SEQ Prefix((OR p1 p2), PP) (OR p1 p2)) / 

\{S\}..j matches (SEQ Prefix(p1, PP) p1) or
\{S\}..j matches (SEQ Prefix(p2, PP) p2)

/ \{ def of Prefix \}

\{S\}..j matches (SEQ Prefix((OR p1 p2), PP) (OR p1 p2)) / 

\{S\}..j matches (SEQ Prefix((OR p1 p2), PP) p1) or
\{S\}..j matches (SEQ Prefix((OR p1 p2), PP) p2)

/ \{ LET OR1 stand for (OR p1 p2), apply def of \{S\}..j matches SEQ \}

\{i | \{S\}..i matches OR1\}..j matches (OR p1 p2) / 

\{i | \{S\}..i matches OR1\}..j matches p1 or
\{i | \{S\}..i matches OR1\}..j matches p2

_ \{ generalize formula to arbitrary \{T\} \}

\{T\}..j matches (OR p1 p2) / 

\{T\}..j matches p1 or \{T\}..j matches p2

/ \{ Theorem B.1:RecursiveMatch(OR) \}

TRUE
Appendix B
A Formal Discussion of Next()

B.1. Correspondence

Next() is a finite automata function, which we now formalize.

Abstractly, a finite automata is a function from a current state and an input character to a new state. What makes it a finite automata is the fact that there are only a finite number of different states. Each state may be accepting or not accepting.

Finite Automata $M: \text{instate} \times \text{Char} \times \text{outstate'}$
where outstate' may be accepting or not accepting.

We will actually consider a finite automata to be a function

$M: \text{instate} \times \text{Char} \times \text{TF} \times \text{outstate'}$

where TF is a Boolean, called the new-arrival Boolean because of the intuition discussed in the previous section. This is not a significant extension, since the Cartesian product of the set of characters and a Boolean can be considered a set of characters twice as big as the set of characters alone.

The output of finite automata $M$ in response to input

$<s_0,TF_0>...<s_n,TF_n>$

each tuple an input character and a TF Boolean, is $q_n$, where $q_1...q_n$ is the sequence of states such that $q_i = M(q_{i-1}, s_i, TF_i)$, and $q_1$ is the initial state of $M$. For any state $q$ returned by $M$, $q$.accepts is TRUE iff $q$ is an accepting state.
We will be converting a regular expression $p$ into a finite automata function $M$. Intuitively, we say that function $M$ \textbf{corresponds} to pattern $p$ (abbreviated $M$ corr $p$) when:

For all strings $s$ (with positions from $0..n$), input $<s_0, TF_0><s_1, TF_1>...<s_j, TF_j>$, sequence of states $q$ where $q_j = M(q_{j-1}, s_j, TF_j)$ for all $j$:

$M(q_{j-1}, s_j, TF_j).\text{accepts} \iff$

$(0..j \text{ matches } p \text{ and } TF_0 = \text{TRUE}) \text{ or }$

$(1..j \text{ matches } p \text{ and } TF_1 = \text{TRUE}) \text{ or }$

...$

$(j-1..j \text{ matches } p \text{ and } TF_{j-1} = \text{TRUE}) \text{ or }$

$(j..j \text{ matches } p \text{ and } TF_j = \text{TRUE})$

The numbering of the input, i.e. $<s_i, TF_i>$, is meant to suggest that the new-arrival $TF_i$ arrives before character $s_i$. Though counterintuitive, this definition simplifies the algorithms.

“Correspondence” is a useful concept because to test $\{1, 4, 9, 99\}..16$ matches $p_1$, we set $TF_1, TF_4, TF_9,$ and $TF_{99}$ to TRUE, while setting all others to false, and then we calculate $q_{16}$ using finite automata function $M$. If $M$ corr $p_1$, then $q_{16}.\text{accepts}$ iff $\{1, 4, 9, 99\}..16$ matches $p_1$.

We formalize all of this with the following, slightly more general, definition:

**Definition B.1:Corr:**

Assuming finite automata function Next($p,...$); arbitrary set of starting positions $\{S\}$ (each position $\_0$); arbitrary input string $s$ (with positions from $0..n$); arbitrary $j \_0$; and sequence of states $q$ such that

$\forall k:0 \leq k \leq j: (q_k = $ Next$(p, q_{k-1}, s_k, k \in \{S\}))$; then

Next($p, ...$) corr $p$ iff

$q_j.\text{accepts} / \{S\}..j$ matches $p$ and $(q_j.\text{inp} / j \in \{S\})$

\textbf{end Definition.}
Appendix B: A Formal Discussion of Next()

Correspondence will be the criterion for the correctness of Next(). For example, we will show that our algorithm for Next((SEQ p1 p2), ...) is correct by showing that Next((SEQ p1 p2), ...) corresponds to (SEQ p1 p2).

In general, each case of Next() is a recursive routine, where Next(p, ...) is calculated as some combination of the Next() functions of the subpatterns of p. As a first step we show how {S}.j matches p can be computed as a function of the subpatterns of p.

**Theorem B.1: RecursiveMatch:**

Assuming p1 and p2 are arbitrary patterns; {ALL} represents the set of all integers _ 0; then

1. {S}.j matches (SEQ p1 p2) / {i | {S}.i matches p1}.j matches p2
2. {S}.j matches (OR p1 p2) / {S}.i matches p1 or {S}.j matches p2
3. {S}.j matches (* p1) / j in {Q}<j+1>, where {Q}<j> = {i|i<j and (i in {S} or {Q}<i}.i matches p1})
4. {S}.j matches (CSLP p1 p2 p3) / {i}{ALL}.i matches p1 and i in {S}.j matches p2 and j..j+1 matches p3
5. {S}.j matches (CSLPN p1 p2 p3) / {i}i in {S} and not {ALL}.i matches p1}.j matches p2 and not j..j+1 matches p3
6. {S}.j matches (CTLP p1 p2 p3) / {S}.j matches p2 and {ALL}.j matches p1 and j..j+1 matches p3
7. {S}.j matches (CTLPN p1 p2) / {S}.j matches p2 and not {ALL}.j matches p1 and not j..j+1 matches p3
8. {S}.j matches (AND p1 p2) / ∃i:i in {S};({i}.j matches p1 and {i}.j matches p2)
Appendix B: A Formal Discussion of Next()

(9) \{S\}..j matches (ALL p1 MINUS p2) /
    \exists i : i in \{S\} : (\{i\}..j matches p1 and not \{i\}..j matches p2)

(10) \{S\}..j matches (MINI-AND p1 p2) /
    \{S\}..j matches p1 and \{S\}..j matches p2

(11) \{S\}..j matches (MINI-NOT p1) /
    not \{S\}..j matches p1

(12) \{S\}..j matches (SET_LONGER p1) /
    \exists i : i in \{S\} : (\{i\}..j matches p1)

(13) \{S\}..j matches (SET_LONGEST p1) /
    \exists i : i in \{S\} : (\{i\}..j matches p1 and
    \forall k : k < i : (not \{k\}..j matches p1)),

(14) \{S\}..j matches (SET_SHORTER p1) /
    \exists i : i in \{S\} : (\{i\}..j matches p1)

(15) \{S\}..j matches (SET_SHORTEST p1) /
    \exists i : i in \{S\} : (\{i\}..j matches p1 and
    not \exists k : i < k < j : (\{k\}..j matches p1))

(16) \{S\}..j matches \{x\} /
    j-1 in \{S\} and s_{j-1} in \{x\}

(17) \{S\}..j matches EMPTY /
    j in \{S\}

(18) \{S\}..j matches ANYSTR /
    \exists k : (k \leq j and k in \{S\})

end Theorem.

Before we prove the theorem, we have to extend the definition of SEQ, OR, \(^*,\) CTLP, CTLPN, CSLP, CSLPN, AND, ALL/MINUS. They have been defined already when p1 and p2 are Existential patterns. But when p1, for example, is not Existential the previous definitions do not apply. We may as well define these cases so that Theorem B.1:RecursiveMatch is automatically TRUE. The next definition does so.
Appendix B: A Formal Discussion of Next()

**Definition B.1: ExtendExistential:***

Assuming $p_1$ and $p_2$ are not Existential, and $p$ is one of

- $(SEQ\ p_1\ p_2)$,
- $(OR\ p_1\ p_2)$,
- $(*\ p_1)$,
- $(CTLP\ p_1\ p_2\ p_3)$,
- $(CTLPN\ p_1\ p_2\ p_3)$,
- $(CSLP\ p_1\ p_2\ p_3)$,
- $(CSLPN\ p_1\ p_2\ p_3)$,
- $(AND\ p_1\ p_2)$,
- $(ALL\ p_1\ MINUS\ p_2)$;

then

$\{S\}..j$ matches $p$ is defined as given by Theorem B.1: RecursiveMatch, (1) through (9).

**end Definition.**

Now the proof of Theorem B.1: RecursiveMatch can proceed.

**Proof of Theorem B.1: RecursiveMatch:**

We have already proved, by Definition ExtendExistential, the case of $p_1$ or $p_2$ not Existential. Now, we have to prove the theorem assuming that $p_1$ and $p_2$ are Existential. (16), (17), and (18) are trivially true. (10) through (15) are true by definition. (8) and (9) simplify to their original definitions when $p_1$ and $p_2$ are Existential. We prove two of the hardest remaining cases, for $(SEQ\ p_1\ p_2)$ and for $(*\ p_1)$.

**Proof of B.1: RecursiveMatch: (SEQ):**

$\{S\}..j$ matches $(SEQ\ p_1\ p_2)$

/* def of $\{S\}..j$ matches $(SEQ\ p_1\ p_2)$ */

$\exists k:\forall k \in \{S\}: (k..j$ matches $(SEQ\ p_1\ p_2))$

/* def of $k..j$ matches $(SEQ\ p_1\ p_2)$ */

$\exists k:\forall k \in \{S\}: (\exists i: (k..i$ matches $p_1$ and $i..j$ matches $p_2))$

/* pull $\exists$ out, push $\exists$ in */

$\exists i: (\exists k \in \{S\}: (k..i$ matches $p_1)):(i..j$ matches $p_2)$

/* $p_1$ Existential */

$\exists i: (\{S\}..i$ matches $p_1):(i..j$ matches $p_2)$
Appendix B: A Formal Discussion of Next()

\[
/ \{ p2 \text{ Existential} \}

\{ i \mid \{S\}.i \text{ matches } p1 \}.j \text{ matches } p2
\]

end Proof.

Proof of B.1: RecursiveMatch(\(*)\):

\[
\{S\}.j \text{ matches } (*) p1
\]

\[
/ \exists k : k \in \{S\}: (k..j \text{ matches } (*) p1)
\]

\[
/ \text{ def of } (*) p1 \}

\[
\exists k : k \in \{S\}: (k = j \text{ or } \exists i : k \leq i < j : (k..i \text{ matches } (*) p1 \text{ and } i..j \text{ matches } p1))
\]

\[
/ \text{ split } \exists k : \}

\[
\exists k : k \in \{S\}: (k = j) \text{ or } \exists k : k \in \{S\}: (\exists i : k \leq i < j : (k..i \text{ matches } (*) p1 \text{ and } i..j \text{ matches } p1))
\]

\[
/ \text{ 1 point rule } \}

\[
\exists k : k \in \{S\}: (\exists i : k \leq i < j : (k..i \text{ matches } (*) p1 \text{ and } i..j \text{ matches } p1))
\]

\[
/ \text{ pull } \exists k \text{ out, push } \exists k \text{ in } \}

\[
\exists i : i < j \text{ and } \exists k : k \in \{S\} \text{ and } k \leq i : (k..i \text{ matches } (*) p1) : (i..j \text{ matches } p1)
\]

\[
/ \text{ simplify } \}

\[
\exists i : i < j \text{ and } \{S\}.i \text{ matches } (*) p1 : i..j \text{ matches } p1
\]
Appendix B: A Formal Discussion of Next()

/ j in {S} or
   {i | i<j and {S}..i matches (* p1)..j matches p1
/ { define {Q<j} }
  j in {S} or {Q<j}..j matches p1,
  where {Q<j} = {i | i<j and {S}..i matches (* p1)}
/ { use new definition for {S}..i matches (* p1) }
  j in {S} or {Q<j}..j matches p1,
  where {Q<j} = {i | i<j and (i in {S} or {Q<i}..i matches p1)}
/ { definition of {Q<j} }
  j in {Q<j+1}
end Proof.

Now we can prove that Next(p,...) corresponds to p. To prove correspondence, assume an s, j, {S}, and q as specified by Definition B.1:Corr, and show that qj.inp equals (j in {S}) and that qj.accepts equals {S}..j matches p. The proof proceeds by cases, one case for each type of pattern. The case for each pattern with subpatterns, for example (SEQ p1 p2), assumes that Next(p1,...) corr p1 and Next(p2,...) corr p2; in addition there is an assumption about the initial state. The patterns without subpatterns (ANYSTR, EMPTY, {x}) prove correspondence assuming something about the initial state. Thus, the proof by cases amounts to a structural induction proof which ultimately rests on the correctness of the initial state. The latter topic was dealt with in section 6.3. Only the most interesting cases are
shown in subsequent sections; interested readers can prove the others analogously.

B.2. Correctness of SEQ

Theorem B.2: CorrSEQ:

Assuming Next(p1, ...) corr p1 and Next(p2, ...) corr p2; initial state of (SEQ p1 p2), q_{initial}, ensures that q_{initial}.q1 and q_{initial}.q2 are the initial states of p1 and p2; then

Next((SEQ p1 p2), ...) corr (SEQ p1 p2).

end Theorem.

Proof:

Assuming arbitrary s, {S}, j, q such that

\[ \forall k: (q_k = \text{Next}((\text{SEQ} p1 p2), q_{k-1}, s_k, k \in \{S\})) \]

we can substitute q_k for outstate, q_{k-1} for instate, s_k for Char, and (k in {S}) for TF into the relevant lines of Next():

function Next((SEQ p1 p2), q_{k-1}, s_k, k \in \{S\}) returns q_k:

1|   q_k.inp ♦ k in \{S\};
21|   q_k.q1 ♦ Next(p1, q_{k-1}.q1, s_k, k \in \{S\});
22|   q_k.q2 ♦ Next(p2, q_{k-1}.q2, s_k, q_k.q1.accepts);
23|   q_k.accepts ♦ q_k.q2.accepts;

This substitution is valid for all k, including k=j. Therefore Line 1 establishes the first requirement of Next((SEQ p1 p2),...) corr (SEQ p1 p2), namely that

\[ q_j.inp = j \text{ in } \{S\} \]

Because Next(p1,...) corr p1, and line 2 is true for all 0≤ k≤ j, we know

\[ q_k.q1.accepts/{S}.k \text{ matches } p1/k \text{ in } \{i \mid \{S\}.i \text{ matches } p1\} \]

This result, the assumption Next(p2,...) corr p2, and line 22 ensure

\[ q_k.q2.accepts/{i \mid \{S\}.i \text{ matches } p1}.k \text{ matches } p2 \]
The formula on the right is the same as

\[
{\{S\}}_k \text{ matches } (\text{SEQ } p_1 \text{ p}_2)
\]

according to Theorem B.1:RecursiveMatch. This result (with \(k=j\)), and line 23, complete the proof of correspondence. (The assumption about initialization states ensures that \(q_1.q_1\) and \(q_1.q_2\) are the initial states of \(p_1\) and \(p_2\), which in turn allows one to apply Definition B.1:Corr to \(p_1\) and \(p_2\).)

B.3. Correctness of *

**Theorem B.3:CorrStar:**

Assuming \(\text{Next}(p_1, \ldots) \text{ corr } p_1\); initial state of \((\ast p_1)\), \(q_{\text{initial}}\), ensures that \(q_{\text{initial}}.q_1\) is the initial states of \(p_1\); then

\[
\text{Next}((\ast p_1), \ldots) \text{ corr } (\ast p_1).
\]

**end Theorem.**

Proof:

Assuming appropriate \(s\), \(j\), \(\{S\}\), and \(q\), we can substitute \(q_k\) for outstate, \(q_{k-1}\) for instate, \(s_k\) for Char, and \((k \in \{S\})\) for TF into the relevant lines of \(\text{Next}()\):

```plaintext
function Next((\ast p_1), q_{k-1}, s_k, k \in \{S\}) \text{ returns } q_k:
1  | q_k.inp \(\ast k \in \{S\};
41 | \text{tempstate} \(\ast \text{Next}(p_1, q_{k-1}.q_1, s_k, \text{FALSE});
42 | q_k.q_1 \(\ast \text{Next}(p_1, q_{k-1}.q_1, s_k,
        \text{ k in } \{S\} \text{ or tempstate.accepts});
43 | q_k.accepts \(\ast k \in \{S\} \text{ or tempstate.accepts};
```
We need some lemmas before continuing with the proof:

**Lemma B.3:1:** \{Q<j\} _ {i | i<j and (i in {S} or {Q<i}.i matches p1)}

**Lemma B.3:2:** \( \forall k: k<j: (k in \{Q<k+1\}/k in \{Q<j\}) \)

**Lemma B.3:3:** \( j in \{Q<j\} / FALSE \)

**Lemma B.3:4:** \( \forall k: 0 \leq k \leq j: (q_k.q_1 = Next(p_1, q_{k-1}.q_1, s_k, k in \{Q<k+1\})) \)

Lemma B.3:1 is just the definition of \{Q<j\}. The next two lemmas are simple applications of this definition. The last lemma must be proved by induction. The base case is trivial, because when \( j \) is -1 there is no \( k \) such that \( 0 \leq k \leq -1 \). For the inductive case we assume that Lemma B.3:4 is true for \( j=n-1 \), and we prove that

\[ q_n.q_1 = Next(p_1, q_{n-1}.q_1, s_{n-1}, n in \{Q<n+1\}) \]

From lines 41 and 42 (with \( n \) substituted for \( k \)), the inductive assumption, Lemma B.3:2, Lemma B.3:3, and Definition B.1:Corr, we conclude that tempstate.accepts equals \{Q<k\}.k matches p1. Realizing that

\[ n in \{S\} or \{Q<n\}.n matches p1 / n in \{Q<n+1\} \]

and substituting the right side as the last argument to Next() in line 42, the line becomes:

\[ q_n.q_1 = Next(p_1, q_{n-1}.q_1, s_n, n in \{Q<n+1\}) \]

This completes the inductive proof of Lemma B.3:4.

In the course of proving Lemma B.3:4 we showed that tempstate.accepts / \{Q<k\}.k matches p1

Combining this result with line 43 shows that

\[ q_k.accepts / (k in \{S\} or \{Q<k\}.k matches p1) \]
Applying Lemma B.3:1 we can rewrite this as
\[ q_k.\text{accepts} \mid k \in \{Q<k+1\} \]
Applying Theorem B.1:RecursiveMatch proves that
\[ q_k.\text{accepts} \mid \{S\}.k \text{ matches } (* p1) \]
Since one of the possible values of k is j, and since line 1 establishes the other half of the proof of correspondence, we have successfully proved correspondence.

B.4. Correctness of **DEFCALL**

As discussed in the section on representing patterns, when the RecCount() field of a pattern definition exceeds MAX_RECURSION, p1 should be treated as \{\}. The increment and decrement (lines 51 and 58) update RecCount() to be the current number of nesting levels of the pattern definition being called. When p1 is treated as \{\}, we always set outstate.q1 to 0 and outstate.accepts to FALSE because \{\} never accepts, no matter what the input. When p1 is treated as p1, a call to Next((DEFCALL p1),...) is treated as a call to p1, which is obviously correct.

B.5. Correctness of **{x}**

We will next show the proof of correctness for a primitive pattern, \{x\}, which matches any single character in the set \{x\}. The only additional aspect of interest is specifying an initial state.

**Theorem B.5:CorrX:**

\[ \text{Next}({x}, \ldots) \text{ corr } {x} \text{ if the initial state is 0.} \]
Appendix B: A Formal Discussion of Next()

end Theorem.

Proof:

Assume appropriate \( S \), \( j \), input \( s \), states \( q \). By substituting (into lines 1, 8 and 9) \( s_k \) for Char, \( (k \text{ in } S) \) for TF, \( q_{k-1} \) for instate, and \( q_k \) for outstate, we see:

\[
q_k.q0 = 1 \quad / \quad k \text{ in } S \text{ and } s_k \text{ in } x
\]
\[
q_k.inp \quad / \quad k \text{ in } S
\]
\[
q_k.accepts \quad / \quad (q_{k-1}.q0 = 1)
\]

Reusing the first result,

\[
q_k.accepts \quad / \quad (k-1 \text{ in } S \text{ and } s_{k-1} \text{ in } x).
\]

This is exactly the definition of \( S..k \) matches \( x \) given in Theorem B.1:RecursiveMatch. By Definition B.1:Corr, Next(\( x \), ...) corr \( x \).

B.6. Correctness of SET_LONGER

In this section we justify the algorithm for SET_LONGER. The state of a SET_LONGER at position \( j \), \( q_j \), has several components: a state \( q_j.q1 \), representing Next(\( p1, ... \)) on the input \( <s_0,\text{FALSE}>,...<s_{j-1},\text{FALSE}> \); a sequence of states \( q_j.qseq \), which has \( q_j.qseq.size \) elements indexed from 0; a sequence of Booleans \( q_j.marked \), which has an element corresponding to each element of \( q_j.qseq \); and the usual Booleans \( q_j.accepts \) and \( q_j.inp \).

Since we are analyzing SET_LONGER, we examine lines 100-143 of Next() under the assumption that ALL is FALSE and CMP is Greater-Than. Assuming appropriate \( s, j, S \), and \( q \), we can substitute
FALSE for ALL, '>' for CMP, $q_k$ for outstate, $q_{k-1}$ for instate, $s_k$ for Char, and ($k \in \{S\}$) for TF, into the relevant lines of Next():

```c
/** remove DEAD_STATEs **/
in_seq ♦ $q_{k-1}.qseq$ with all DEAD_STATEs deleted;
in_marked ♦ $q_{k-1}.marked$ with all marks corresponding to DEAD_STATEs deleted;
in_size ♦ in_seq.size;

/** calculate next states **/
$q_k.q1$ ♦ Next(p1, $q_{k-1}.q1$, $s_k$, FALSE);
for each state in_seq[i] do
  $q_k.qseq[i]$ ♦ Next(p1, in_seq[i], $s_k$, FALSE);
  $q_k.marked[i]$ ♦ in_marked[i];

/** take care of new-arrival **/
if ((k in \{S\})) then
  $q_k.qseq[in_size]$ ♦
  Next(p1, $q_{k-1}.q1$, $s_k$, TRUE);
  $q_k.marked[in_size]$ ♦ (k in \{S\});

/** zero younger or older duplicates **/
for all i, y such that (i > y) do
  if ($q_k.qseq[i] = q_k.qseq[y]$) then
    $q_k.qseq[i]$ ♦ DEAD_STATE;

/** calculate accepts **/
$q_k.accepts$ ♦ $\exists i : (q_k.qseq[i].accepts$ and
  no y such that (i > y) has
  $q_k.qseq[y].accepts$ and
  $q_k.marked[i])$;
```

These lines are true for all $k$ from 0 to $j$. Notice that the lines involving marked[i] can be safely ignored under the SET_LONGER assumptions. For any $i$, marked[i] is initially set to TRUE, and at each succeeding iteration it is copied, so it remains TRUE. If we ignore such lines, and if we temporarily ignore the deletion and creation of DEAD_STATEs, the lines become simple to analyze:
Before proving correspondence formally, we first give some intuition.
Recall that
\[
{\{S\}}..j \text{ matches (SET_LONGER p1)} / \exists \ i : \{i\}..j \text{ matches p1}
\]
Intuitively, \{3,9,12\}..j matches (SET_LONGER p1) is tested by individually testing \{3\}..j matches p1, \{9\}..j matches p1, and \{12\}..j matches p1. \(q_j.qseq\) is a sequence of states of p1, one state for each \(i \leq j\) in \{S\}. \(q_j.qseq[0].accepts\) is TRUE when \{3\}..j matches p1, \(q_j.qseq[1].accepts\) is TRUE when \{9\}..j matches p1, etc. If we consider \{S\} ordered from smallest (called the “oldest”) starting position to largest (“youngest”), and we let \{S_i\} be the set containing only the ith element of \{S\}, then \(q_j.qseq[i].accepts\) is TRUE if \{S_i\}..j matches p1, using 0 based indexing.

The algorithm sets \(q_j.accepts\) to TRUE when any of the \(q_j.qseq[i].accepts\) is TRUE. From the previous discussion this happens when \(\exists \ i : \{i\}..j \text{ matches p1}\), which is exactly the definition of \{S\}..j.
matches (SET_LONGER p). Therefore, if \( q_j \cdot \text{qseq} \) is as described, then
\[ \text{Next}((\text{SET_LONGER} p), \ldots) \text{ corr} (\text{SET_LONGER} p). \]

Here is the same argument, formalized. Consider \( q_k \cdot \text{qseq}[i] \), and let \( m \) be the iteration at which the \( i \)th element of \( q_k \cdot \text{qseq} \) was first installed.

For all \( k \) from \( m+1 \) to \( j \), line 113 applies to \( q_k \cdot \text{qseq}[i] \):
\[
\forall k: m+1 \leq k \leq j: (q_k \cdot \text{qseq}[i] = \text{Next}(p1, \text{in}_\text{seq}[i], s_k, \text{FALSE}))
\]

For \( k=m \), lines 121-123 apply, and we see that \( m \) is in \( \{S\} \):
\[
\text{if } k=m \text{ then } q_k \cdot \text{qseq}[i] = \text{Next}(p1, q_{k-1} \cdot q1, s_k, \text{TRUE})
\]

Finally, for \( k \) from 0 to \( m-1 \), line 111 applies:
\[
\forall k: 0 \leq k \leq m-1: (q_k \cdot q1 \cdot \text{Next}(p1, q_{k-1} \cdot q1, s_k, \text{FALSE}))
\]

We put all these results together by defining sequence of states \( Q \) such that \( Q_k \) is \( q_k \cdot q1 \) for \( k<m \) and \( Q_k \) is \( q_k \cdot \text{qseq}[i] \) for \( m \leq k \leq j \). Now we can say
\[
\forall k: 0 \leq k \leq j: (Q_k = \text{Next}(p1, Q_{k-1}, s_k, (k=m)))
\]

Applying Definition B.1:Corr, we conclude
\[ q_k \cdot \text{qseq}[i] \cdot \text{accepts} = Q_k \cdot \text{accepts} = \{m\}..k \text{ matches } p1 \]

This is important: we just showed that each state \( q_k \cdot \text{qseq}[i] \) calculates the match of \( \{m\}..k \text{ matches } p1 \) for some \( m \) in \( \{S\} \). We did not show it, but the converse is also true: for each \( m \) in \( \{S\} \) there is a state \( q_k \cdot \text{qseq}[i] \) calculating \( \{m\}..k \text{ matches } p1 \).

Now, lines 141-143 simplify to
\[
q_k \cdot \text{accepts} \overset{\exists i: (q_k \cdot \text{qseq}[i] \cdot \text{accepts})}{\leq 141}
\]

because the existence of the least \( i \) is the same as the existence of any \( i \). (Lines 142-143 are more debris applicable only to SET_LONGEST
Appendix B: A Formal Discussion of Next()

and SET_SHORTEST.) Plugging in our discovery about $q_k.qseq[i].accepts$ shows

$$q_k.accepts = \exists i: (\exists \ m: \ m \ in \ [S]: \ \{m\}.k \ matches \ p1)$$

which simplifies to

$$q_k.accepts = \{S\}.k \ matches \ (SET_LONGER \ p1)$$

Specializing $k$ to $j$ establishes the main proof obligation for showing correspondence. We have successfully formalized the previous intuitive argument.

To clear up the details involving DEAD_STATE, notice that a state in $q_k.qseq$ is only reset to DEAD_STATE when there is an identical state earlier in the sequence. Since Next$(p1, \ldots)$ is deterministic, and all states in $q_k.qseq$ are called with the same arguments, two identical states will remain identical as $k$ increases. Thus it is wasteful to calculate both. The one that is kept is the one added to qseq earliest, which we call the “older” state. The “younger” state is set to DEAD_STATE, and is ignored in further processing thanks to lines 102-103 of Next(). The reason for preferring “older” states to “younger” and for not deleting DEAD_STATEs immediately will be clarified in the chapter on Parse(). Essentially, keeping the older state allows us the trace back the longer match during parse extraction. Also note that deleting duplicates is more than an “optimization”. Removing duplicates ensures that each state of $p1$ can appear at most once in qseq, which in turn ensures that (SET_LONGER $p1$) has only a finite number of different states when $p1$ has a finite number of states.
The algorithm for SET_SHORTER is almost exactly the same as for SET_LONGER. Only one line is different: instead of zeroing younger duplicates we zero older duplicates. See line 131. As a result, the correctness argument for SET_LONGER translates with trivial modifications into a correctness argument for SET_SHORTER.

**B.7. Correctness of SET_LONGEST**

In this section we argue the correctness of the algorithm for SET_LONGEST and SET_SHORTEST. The careful reader will notice that the only difference between these cases and the SET_LONGER and SET_SHORTEST cases is that ALL is set to TRUE instead of FALSE. Therefore in this section we merely point out the differences that result from this change.

As before, state \( q_k.qseq[i] \) represents \( \{m\}.k \) matches \( p1 \) for some \( m \).

However, in the SET_LONGEST case \( qseq \) has a state for every \( m \leq k \), whereas in the SET_LONGER case \( qseq \) has a state for \( m \) iff \( m \in \{S\} \).

The lines involving marked[] cannot be ignored; marked[i] is TRUE iff \( qseq[i] \) represents an \( m \) in \( \{S\} \).

The reason for adding a new state to \( q_k.qseq \) at every \( k \) is that to test if \( \{i\}.k \) matches (SET_LONGEST \( p1 \), \( \{i\}.k \) must match \( p1 \) and \( \{m\}.k \) must NOT match \( p1 \) for any \( m \) less than \( i \). Therefore we must keep track of the matches from all possible \( m \) (which we do by adding a new state to \( q_k.qseq \) at each \( k \)), and we must differentiate the \( m \) in \( \{S\} \) from \( m \) not in \( \{S\} \) (which we do using marked[]).
Finally, we analyze the computation of $q_k.accepts$. Here are the relevant lines:

```c
/** calculate accepts **/
q_k.accepts \iff \exists i: (q_k.qseq[i].accepts and no y such that (i > y) has
q_k.qseq[y].accepts and q_k.marked[i]);
```

The computation is the same as for (SET_LONGER p1), except for the addition of two extra conditions for accepting. Lines 141-143 specify that $i$ is the least index of an accepting state in $qseq$. Since $q_k.qseq$ is ordered from “oldest” to “youngest”, the oldest accepting state is found at the minimum $i$ such that $q_k.qseq[i]$ accepts. If there is no “oldest” accepting state, then there is NO accepting state, so $q_k.accepts$ should clearly be FALSE for the same reasons as given for (SET_LONGER p1). If there is an accepting state, there is clearly an oldest. The oldest accepting state has the neat feature that all older states do not accept. Mathematically, if $q_k.qseq[i]$ has the oldest accepting state, and it represents $\{m\}..k$ matches $p1$, this feature is expressed as follows:

$$
\forall y: y < m: (\text{not } \{y\}..k \text{ matches } p1)
$$

If $q_k.marked[i]$ is TRUE, then it represents $\{m\}..k$ matches $p1$ for some $m$ in $\{S\}$. Mathematically, we can conclude:

$$
\exists m: m \in \{S\}: (\{m\}..k \text{ matches } p1) \text{ and } \\
\forall y: y < m: (\text{not } \{y\}..k \text{ matches } p1))
$$
This equation is exactly the definition of \(\{S\}..k \text{ matches} (\text{SET\_LONGEST } p1)\).

The algorithm for \text{SET\_SHORTEST} is nearly identical to the one just discussed. Only two lines change, lines 131 and 142. The former line zeros older duplicates instead of younger, and line 142 sets \(q_k.\text{accept}\) to \(\text{TRUE}\) iff the youngest accepting state is marked, instead of the oldest. The modifications to the correctness argument for \text{SET\_LONGEST} are easy to find.

**B.8. Worst-Case Complexity of Next()**

The running time and space requirements of Next() will be analyzed for a single call to Next(). Practically speaking, the analysis is of little value; the performance of Next(p,...) depends crucially on the pattern \(p\) being processed and the caching optimizations to be described in the subsequent section. Also, the usual case behavior is radically different from the worst case. Analyzing the expected performance is too difficult; however, section 6.6 gives experimental results that describe the actual performance of the optimized Next() function. In the current section, we analyze the worst case behavior of Next(p,...) assuming no caching but with two cases detailing the composition of \(p\): (1) \(p\) contains no \text{SET\_} patterns; (2) \(p\) contains \text{SET\_} patterns.

The Next() algorithm is fairly simple. Except for the \text{SET\_} patterns, each line is either a recursive call to Next() or the computation of a Boolean expression. In fact, the running time of Next() can best be described as proportional to the size of the instate parameter;
similarly, since Next() uses essentially no space but it creates an outstate, we say that Next() uses space proportional to instate. Now we justify these claims and try to relate the size of instate to the size of the pattern.

The instate parameter can be thought of as a tree. States of ANYSTR, EMPTY, and \{x\} contain 0 substates; states of * and MINI-NOT contain 1 substate; states of SEQ, OR, MINI-AND, and CSLP contain 2 substates; states of SET_ patterns may contain an large number of substates. The size of a state is defined by the following formula:

\[
\text{size(state)} = \begin{cases} 
1 & \text{if state of a ANYSTR, EMPTY, \{x\}} \\
1 + \text{size(substate)} + \text{size(substate)} & \text{if state of a (*) pattern} \\
1 + \text{size(state.q1)} + \sum_{i} \text{size(state.qseq[i])} & \text{if state of a SET_ pattern} \\
1 + \text{size(state.q1)} + \text{size(state.q2)} & \text{else}
\end{cases}
\]

Essentially size() just counts the number of nodes in the tree representing the state. The only surprising feature of the size() formula is the fact that the size of a state for * counts its substate twice. The reason for this is that we want size(state) to equal the number of calls to Next(). The * pattern calls Next() twice on its subpattern. This may seem more reasonable if one views (\* p1) using its recursive expansion, in which p1 appears twice:

\[
(\* p1) = (\text{OR EMPTY (SEQ p1 (* p1))})
\]

With this definition, it should be clear that the number of calls to Next() is essentially the same as size(instate). Except for SET_, each case of Next() requires only a constant amount of additional processing per call to Next(). The SET_ patterns require a troubling loop (see
Appendix B: A Formal Discussion of Next()

lines 104-105 of Next()) which seems to require time proportional to the square of the number of substates. However, the search can be implemented using a fancier data structure for storing the substates (such as an ordered hash table) to reduce the time to a constant charge per substate. In practice such an enhancement is undesirable since the number of substates is typically small (≤ 25 for the examples we have looked at), as is the constant factor on the comparisons being computed (essentially 1 machine instruction using the state representation in section B.9.3). Thus, we conclude that running time of Next() is proportional to the size of instate.

**Theorem B.8: NextTime:**

Next(p, instate, Char, TF) requires worst case time $O(size(instate))$.

*end Theorem.*

Since each call to Next() returns exactly one state, the size of outstate() is bounded by the number of calls to Next(), which is bounded by size(instate).

**Theorem B.8: NextSpace:**

Next(p, instate, Char, TF) requires worst case space $O(size(instate))$.

*end Theorem.*

To relate size(instate) to the size of the pattern p that Next(p,...) corresponds to, we first distinguish the case of p containing no SET_ patterns. This is a serious restriction, since it essentially reduces the language to that of restricted regular expressions plus CSLP, CSLPN,
CTLP, and CTLPN. On the other hand, this case is important for comparing Next() to algorithms which implement restricted regular expressions.

When \( p \) contains no \texttt{SET_} patterns, a state of \( p \) has one substate for each subpattern. Thus, \( \text{size(state of } p \text{)} \) is equal to the size of \( p \), where the size of \( p \) is the number of patterns in it (counting patterns twice inside a \texttt{*}, recursively).

When \( p \) contains \texttt{SET_} patterns, the situation is much worse. A state for (\texttt{SET_} \( p_1 \)) may theoretically have as many substates as \( p_1 \) has when \( p_1 \) is compiled into a deterministic finite automata. Even worse, a (\texttt{SET_} \( p_1 \)) pattern can conceivably compile to more than \( n \) states when \( p_1 \) compiles to \( n \) states, because each subset of states of \( p_1 \) is a possible state of (\texttt{SET_} \( p_1 \)). These two results imply that an extended regular expression using nested \texttt{SET_} patterns may contain a super-exponential number of states. Fortunately, most patterns do not behave this badly!

## B.9. Optimizing Next()

Without clever engineering, the Next() algorithm is too slow and memory consumptive to be of practical importance. Here we describe a number of relatively simple optimizations that make Next() efficient. Then we present experimental results which give some idea of the expected behavior of the optimized Next().
B.9.1. Simplifying to Primitive Patterns

One of the simplest patterns is the primitive pattern \{x\}, which matches a set of characters. Often a complicated pattern can be simplified to \{x\}, saving execution time and space. For example, 

\( (\text{OR } a \ b \ c \ d) \) is the same as \{a,b,c,d\}, and 

\( (\text{All Digit minus (from '5' to '9')} ) \) is the same as \{0,1,2,3,4\}. The general recursive rules are simple:

\[
\text{If } p_1 \text{ and } p_2 \text{ are } \{x\} \text{ and } \{y\} \text{ then } \\
(\text{OR } p_1 \ p_2) \text{ is } \{x\} \text{ union } \{y\}
\]

\[
\text{If } p_1 \text{ and } p_2 \text{ are } \{x\} \text{ and } \{y\} \text{ then } \\
(\text{AND } p_1 \ p_2) \text{ is } \{x\} \text{ intersection } \{y\}
\]

\[
\text{If } p_1 \text{ and } p_2 \text{ are } \{x\} \text{ and } \{y\} \text{ then } \\
(\text{ALL } p_1 \ MINUS \ p_2) \text{ is } \{x\} \text{ set-minus } \{y\}
\]

\[
\text{If } p_1 \text{ is } \{x\} \text{ then } \\
(\text{SET}_i p_1) \text{ for any of the SET_ patterns is } \{x\}.
\]

B.9.2. Eliminating SET Patterns

The SET_ patterns are one of the main culprits sapping the efficiency of Next(). Sometimes a SET_SHORTER or SET_LONGER can be eliminated without changing the behavior of a pattern.

Example:

\[
(\text{SET_LONGER} \ (\text{MINI-AND} \ (\text{SET-SHORTER} \ (\text{SEQ} \ a \ \text{anystr})) \\
(\text{SEQ} \ \text{anystr} \ b)))
\]

is equivalent to

\[
(\text{SET_LONGER} \ (\text{MINI-AND} \ (\text{SEQ} \ a \ \text{anystr}) \\
(\text{SEQ} \ \text{anystr} \ b)))
\]

end Example.

Next() calculates \{S\}..j matches (SET_LONGER p1) by separately calculating [i]..j matches p1 for each i in \{S\}. If one can ensure that \{S\} has at most 1 element, then \{S\}..j matches (SET_LONGER p1) is
same as \([S].j\) matches \(p1\); the SET_LONGER can be eliminated. The following formula describe a predicate SAFE\((p)\) which is TRUE if pattern \(\text{Next}(p,...)\) is guaranteed to calculate \([S].j\) matches \(p\) for an \([S]\) with at most 1 element. They can be proven by referring to the rules in Theorem B.1:RecursiveMatch.

\[
\text{If } p \text{ is of the form (SET_SHORTER } p1) \text{ or (SET_LONGER } p1) \text{, and SAFE}(p), \text{ then } p \text{ can be replaced by } p1.
\]

\[
\text{If } p \text{ is of the form (SET_SHORTER } p1), (\text{SET_LONGER } p1), (\text{SET_SHORTEST } p1), \text{ or (SET_LONGEST } p1), \text{ then SAFE}(p1).
\]

\[
\text{If } \text{SAFE}((\text{OR } p1 p2)) \text{ or } \text{SAFE}((\text{MINI-AND } p1 p2)) \text{ then } \text{SAFE}(p1) \text{ and } \text{SAFE}(p2).
\]

\[
\text{If } \text{SAFE}((\text{SEQ } p1 p2)) \text{ or } \text{SAFE}((\text{MINI-NOT } p1)) \text{ or } \text{SAFE}((\text{CSLP pleft p1 pright})) \text{ then } \text{SAFE}(p1).
\]

\[
\text{if } \text{SAFE}(\text{Foo}) \text{ for ALL calls to definition Foo, then } \text{SAFE}(\text{<body of Foo>}).
\]

**B.9.3. Representing States Compactly**

One of the most fearsome inefficiencies of the Next() function is storing the states, which is necessary for parse extraction. Each state is essentially a tree at least as large as the pattern. With input of a million characters, the space needed to store a million individual states would be prohibitive. Furthermore, in a later section we will discuss caching of transitions, which requires quickly comparing two states to see if they are equal. Equality on states can be recursively defined: two states are equal if corresponding fields are equal and corresponding substates are equal. Unfortunately, a naive representation of states would require time proportional to the size of the states to test equality, which would be unacceptable in practice.
Fortunately, there is a representation for states that solves both problems. Consider representing an state \( q \), as discussed in section 6.2, using the following record:

\[
\text{Type St is record}
\]

\[
\begin{align*}
\text{inp : Boolean; } \\
\text{accepts : Boolean; } \\
\text{SubStates : iSeqSt;} \\
\text{StateVal : integer; }
\end{align*}
\]

\[
\text{Type iSeqSt is an index (pointer) to a sequence of St;}
\]

The general idea is that all the substates of a state are stored in the sequence of states pointed to by SubStates. This permits two or more states having the same substates to share a single copy of the substates. Here is how to map any abstract state \( q \) (where an abstract state is as described in section 6.2) into the St representation:

**Definition B.9.3: StateRep:**

Assuming \( T(\text{seq of St}) \) is a function that converts every unique sequence of St into a unique iSeqSt; \( L(\text{iSeqSt}) \) is the inverse of \( T() \), converting an iSeqSt back into the corresponding sequence of St; then

Concrete St Q represents abstract state q iff:

- Q.inp \( / \) q.inp, and
- Q.accepts \( / \) q.accepts, and
- if q is a state of SEQ, OR, MINI-AND, or CSLP then
  - \( L(Q.\text{SubStates})[0] \) represents q.q1, and
  - \( L(Q.\text{SubStates})[1] \) represents q.q2, and
  - Q.StateVal = 0
- else if q is a state of *, DEFCALL, or MINI-NOT then
  - \( L(Q.\text{SubStates})[0] \) represents q.q1, and
  - Q.StateVal = 0
- else if q is a state of a SET_pattern then
  - \( L(Q.\text{SubStates})[0] \) represents q.q1, and
  - \( L(Q.\text{SubStates})[i+1] \) represents q.qseq[i], and
  - Q.StateVal = 0
- else if q is a state of an EMPTY, ANYSTR, or \{x\} then
  - Q.StateVal = q.q0, and
  - Q.SubStates = nil

end Definition.
To illustrate the space savings made possible by the representation, consider figure B-1.:  

Figure B-1: Two States

These two states illustrate redundancy we can take advantage of. Notice that the sequence of states under the topmost node of Sample State 1 is the same as the sequence of states under the topmost node of Sample State 2. Also, the leaf sequences of both states are the same. We can convert the two trees representing the states into a directed acyclic graph (DAG) that represents the same thing more concisely:

Figure B-2: Two States Compacted

In this example, it requires 14 states to represent two states as two separate trees, but only 6 states to represent the two states as a DAG, a dramatic decrease which becomes even more dramatic as the number of states increases.

When T() and L() are implemented intelligently, no sequence of St will be stored twice. T can be implemented as a hash table whose key is a sequence of St and whose data is an iSeqSt, so T() operates in constant time. L() can be implemented as a simple array indexed by an iSeqSt with elements which are sequences of St, so it too operates in constant time. When T(SeqSt1) is called, and a sequence of states equal to SeqSt1 has previously been inserted into the hash table, T() returns
the iSeqSt already allocated for SeqSt1. If SeqSt1 is not presently in
the hash table, SeqSt1 is inserted into L’s array, indexed by a new
iSeqSt1, and then iSeqSt1 is stored in M’s hash table indexed by the
key SeqSt1.

Happily, this compact state representation also makes it possible to
compare two abstract states in constant time. The following definition
and theorem describes how:

**Definition B.9.3:EqualSt:**

Assuming Q1 and Q2 are St; then

\[(Q1 = Q2) \iff ((Q1.inp = Q2.inp) \quad \text{and} \quad
(Q1.accepts = Q2.accepts) \quad \text{and} \quad
(Q1.SubState = Q2.SubState) \quad \text{and} \quad
(Q1.StateVal = Q2.StateVal))\]

**end Definition.**

**Theorem B.9.3:EqualStates:**

Assuming Q1 and Q2 are St representing abstract states q1 and q2; then

\[(q1 = q2) \iff (Q1 = Q2)\]

**end Theorem.**

Proving this theorem requires a structural induction on the structure
of St, which we leave to the interested reader.

Integrating the representation into Next() is straightforward. Instead
of accepting and returning an abstract state, Next() accepts and
returns an St representing the state. We rewrite parts of the Next()
function to illustrate the idea:
Appendix B: A Formal Discussion of Next()

function Next(p, inSt, Char, TF) returns outSt:
begin
  1 outSt.inp ♦ TF;
  2 outSt.SubStates ♦ nil; /* default value */
  3 outSt.StateVal ♦ 0; /* default value */
  4 MySeqSt ♦ []; /* initialize to the empty sequence */

  case p of
    EMPTY:
      11 outSt.StateVal ♦ 0;
      12 outSt.accepts ♦ TF;
    ....
    (SEQ p1 p2):
      41 MySeqSt[0] ♦ Next(p1, L(inSt.SubStates)[0], Char, TF);
      42 MySeqSt[1] ♦ Next(p2, L(inSt.SubStates)[1], Char,
                      MySeqSt[0].accepts);
      44 outSt.SubStates ♦ T(MySeqSt);
      45 outSt.accepts ♦ MySeqSt[1].accepts;
    ....
  end function.

The code is similar to the original version of Next(). In the SEQ case, for instance, we call Next() on the substates (actually, the sub-St's) and store the results in MySeqSt[0] and MySeqSt[1] instead of in the .q1 and .q2 fields of an abstract state. Then we convert MySeqSt to an iSeqSt using the T() function, and we store this index in outSt.SubStates.

B.9.4. Creating a DFA

One of the most important features of a regular expression is that any regular expression can be converted into a DFA. In many applications this eliminates the need for extensive optimization of Next(). After compiling a pattern into a DFA, Next() never has to be called. Instead, the DFA can be simulated in a tight loop using a fast and simple
lookup table. Even if the compilation takes many hours, the pattern can be searched for as often as desired, very quickly.

In practice, there are several factors which discourage DFA creation.

- Compilation can take hours for complicated patterns. Debugging a pattern requires an edit-compile-test cycle, and a lengthy compile phase makes this process unbearable.

- A DFA for a complicated pattern can require a huge amount of space.

The next section describes a method, lazy transition evaluation, of avoiding DFA creation while retaining reasonable matching efficiency. TLex uses lazy transition evaluation as the default, but there is an option to use full DFA compilation. In this section, we describe how to use Next() to create a DFA.

To create a DFA, one must find all possible states, and all possible transitions between states. The following algorithm creates a DFA calculating \( \{ \text{ALL} \}..j \) matches \( p \), storing the resulting set of states in States and the transitions in Transitions:

\[
\begin{align*}
\text{Transitions} & \uparrow \{\}\; ; \\
\text{States} & \uparrow \{\text{initial\_state\ of\ } p\}\; ; \\
\text{ExploredStates} & \uparrow \{\}; \\
\text{while}(\text{ExploredStates} \_ \_ \_ \text{States}) \_ \text{do} & \\
\text{state1} & \uparrow \text{any state in} \ (\text{States} \_ \text{ExploredStates}); \\
\text{for each letter } c \text{ in the alphabet do} & \\
\text{state2} & \uparrow \text{Next}(p, \text{state1}, c, \text{TRUE}); \\
\text{Transitions} & \uparrow \{\text{states1} \_ i \_ \_ \_ \text{state2}\}; \\
\text{States} & \uparrow \text{States} + \{\text{state2}\}; \\
\text{ExploredStates} & \uparrow \text{ExploredStates} + \{\text{state1}\}; \\
\end{align*}
\]

This algorithm is fairly simple. It calculates all transitions from all states. What makes it non-trivial is the fact that we do not know all the states until we are finished compiling! As we discover states they
are added to the States set. The set ExploredStates remembers the states we have already explored, and we continue until every state has been explored (and thus every state reachable from the initial state has been discovered). The while-loop terminates because \( p \) has a finite number of states.

In practice one needs a clever compact representation for the compiled transitions, and a compact representation for the states like the one described above. In addition, in some applications state minimization might be useful. See [Aho et al. 1986] for a full discussion of state minimization and representing transitions.

### B.9.5. Caching Transitions

A transition describes what the next state is, based on the current state, the input character, and the new-arrival Boolean. \( \text{Next}(p, q, \text{Char}, \text{TF}) \) can be considered a routine for calculating transitions. A finite automata with \( n \) states has \( |a| \sim n \) transitions, where \( |a| \) is the number of different input characters and TF Booleans (typically \( |a| = 256 \sim 2 = 512 \)). Therefore, storing the transitions of a finite automata can be even more difficult than storing the states.

Storing all transitions seems especially wasteful when one realizes that most of them are rarely or never used, even on very long strings. For example, a finite automata used to search text stores a transition from every state when the input character is 'Z'. If there are no Z's in the input, these transitions just take up space. If there are a few Z's, only the Z transitions from a few states are relevant.
Furthermore, finite automata tend to obey an 80/20 law, where a small percentage of the transitions are used a large percentage of the time. For example, in searching for a string like "hello", most of the time a finite automata is in the initial state, looking for an "h".

The technique of "Lazy Transition Evaluation" [Aho et al. 1986, 128] takes advantage of these characteristics to make pattern matching more efficient. The previous section described the usual paradigm of regular expression search: compile a regular expression into a DFA, finding and storing all states and all transitions. The transitions and states are calculated by repeatedly calling the Next() function on each state with all possible inputs. This stage can be extremely time consuming! However, after compiling into a DFA, the stored states and transitions can be used in place of the Next() function to quickly find matches to the regular expression in an input string.

Lazy evaluation offers a different paradigm. In the first stage we find only the initial state. This requires very little time and space. In the next stage, we match the input string by continuously calling Next() on the current state and the current input character. The only states stored are those states that are discovered in the course of processing the input. (The states must be stored for use by the parse extraction algorithm.) None of the transitions MUST be stored, since we can always recalculate the next state using the Next() function. However, it is significantly faster to access a stored transition than it is to consult the Next() function. As a result, we modify Next() to maintain a cache of the most recently used transitions. If a transition is in the
cache, Next() uses it to return the correct outstate; otherwise, Next() calculates outstate as usual and inserts the calculated transition into the cache.

Lazy evaluation essentially postpones the compilation stage until the pattern matching stage. However, it can be more efficient because only the transitions and states actually used in matching are compiled. The size of the transition cache can be varied to trade off space for time. With a transition cache large enough to store all the transitions, lazy evaluation is at least as efficient as first compiling and then matching (except compiling has been delayed until runtime).

Typically, pattern matching using lazy evaluation appears slow at the start, until the cache gets filled and the “working set” of states is discovered. Then, pattern matching proceeds almost as quickly as in the precompiled paradigm, even with a relatively small transition cache.

The recursive nature of Next() makes transition caching even more valuable than as described in [Aho et al. 1986]. Even if the desired transition from a state is not in the cache, the desired transition from a substate may be present, and a recursive call to Next() will find it. Furthermore, because our patterns are stored as a DAG (see section A.3.3), the transitions cached for a definition or primitive pattern are useful each place the definition appears in a pattern.

Integrating transition caching into Next() is easy. At the start of the function we check if the transition to calculate is already in the cache; if so, we immediately return it. Otherwise, Next() calculates as usual,
except before returning it inserts the newly calculated transition into
the cache. Here are the new lines of Next():

```plaintext
function Next(p, inSt, Char, TF) returns outSt:
  begin
    new if Is_In_Transition_Cache(p, inSt, Char, TF) then
      new outSt ♦ Lookup_In_Transition_Cache(p, inSt, Char, TF);
      new return;
    end
    ..... normal Next() code .....;
    new Insert_Into_Transition_Cache(p, inSt, Char, TF, OutSt);
  end function.
```

Caching, and the compact representation of states described in section
B.9.3, interact in a very subtle way. The fundamental assumption of
transition caching is that the arguments p, inSt, Char, and TF
uniquely determine outSt. If Next() were truly a function without side
effects, this would be true; however, Next() may increment the
RecCount field of a number of patterns, which is a side effect not
obviously represented in the arguments to Next(). Is it possible that
two calls to Next(), with the same arguments but a different spectrum
of RecCount values, can return different OutSt? If so, the caching
scheme described is incorrect.

Fortunately, the answer is no when a simple assumption is made:

(assumption) Each state of a DEFCALL when the DEFCALL is
treated as {} is different from each state of the same
DEFCALL when the DEFCALL is treated as a call to subpattern
p1.

We can see this as follows. If two states q are the same, then obviously
their corresponding substates must be the same. If two calls to Next()
have the same arguments, then the two calls proceed exactly the same
until a DEFCALL is executed and the RecCount in one case is different
from the other. At this point, if (assumption) is true, then the input
state q in one call must be different from the input state q' in the other
call. But this contradicts the observation that equal states have equal
substates; therefore such a situation can never happen. Basically we
have shown that the state q adequately encodes all relevant RecCount
values. We have also learned that any transition cached for a bounded
recursive pattern at one depth can never apply to the behavior of the
pattern at another depth.

B.9.6. Character Equivalence Classes

In this section we explain an optimization, equivalence classes of
characters, that can dramatically reduce the number of possible
transitions. The idea appears in [Paxson 1988]; we can apply it better
thanks to the recursive nature of Next().

A transition cache illuminates another source of inefficiency. Consider
the pattern Num, defined as follows:

\begin{verbatim}
(define Num (seq (+ (from '0' to '9'))
  '.'
  (* (from '0' to '9'))))
\end{verbatim}

Notice that this pattern treats all digits the same, and it treats every
color character which is neither a digit nor a period the same. When we say
that Num “treats the characters X and Y the same” we really mean
that Next(Num, instate, X, TF) = Next(Num, instate, Y, TF) for every
combination of instate and TF value. In other words, the characters X
and Y cause the same transitions, no matter what the state and no
matter what the TF value.
Since characters X and Y always cause the same transitions, it is wasteful to store transitions for both X and Y. Furthermore, we should be able to use any transitions stored for X in calculating a transition for Y. If we take advantage of equivalence classes, each state of Num is the source of 6 transitions (3 equivalence classes times 2 possible TF values) instead of the usual 512 possible transitions. Obviously, taking advantage of equivalence classes could dramatically increase the probability of “hits” in the transition cache!

One can see how character equivalence classes arise by examining Next(): the only place where the behavior of Next() depends on the input character is the case of pattern \{x\}. The \{x\} case treats all characters in \{x\} the same, and all other characters the same. If one can show that characters X and Y are treated the same by all the \{x\} patterns in a pattern, then we can conclude that X and Y are treated the same in the whole pattern.

We make use of a pattern p’s equivalence classes of characters as follows. When Next(p, instate, Char, TF) is called we translate Char to the smallest character in the equivalence class containing Char. This is perfectly legal since Next(p,...) treats both characters the same by definition.

Although characters could be translated recursively at every pattern, we implement translations only at the root of the main pattern, each definition, and each \{x\} pattern. We restrict it to these cases to limit the space taken up by the translation tables while maximizing the benefit of translation. Translating at the root of the main pattern
coalesces most of the non-alphabetic characters, in typical text processing patterns. Translating at definitions is valuable because if the definitions are used in multiple locations the translation is useful in each place. Translating at each \{x\} pattern means an arbitrary set of characters can be stored as a single character. All characters in \{x\} translate to the same character, all other characters translate to a different character.

To translate at a pattern \(p\), we precalculate an array of characters \(AR_p\), where \(AR_p[x]\) gives the lowest character in the equivalence class containing \(x\). It remains to describe how to calculate the equivalence class of characters of a pattern. In a pattern \(p\), let \(\text{PrimitivesOf}(p)\) be the set of all patterns \(\{x\}\) in \(p\). Construct the coarsest set (i.e., the fewest number) of equivalence classes \(e_i\) such that if characters \(f\) and \(g\) are in \(e_i\), and \(\{x\}\) is any pattern in \(\text{PrimitivesOf}(p)\), then either \(f\) and \(g\) are both in \(\{x\}\) or both NOT in \(\{x\}\).

**B.10. Listing of Optimized Next()**

This section lists an optimized version of Next() and InitialState(); it includes a (TRANSLATE \(p1\)) case, which implements the optimization discussed in section B.9.6; the lazy evaluation idea of section B.9.5; and the compact state representation of section B.9.3. Also, we pretend that each state has a “marked” field, though it is only relevant for the state of a SET_ pattern. Finally, sequences such as MySeqSt are indexed from 0.
function Next(p, inSt, Char, TF) returns outSt:
    begin
        if Is_In_Transition_Cache(p, inSt, Char, TF) then
            outSt ◆ Lookup_In_Transition_Cache(p, inSt, Char, TF);
            return;
        outSt.inp ◆ TF;
        outSt.SubStates ◆ nil; /* default value */
        outSt.StateVal ◆ 0; /* default value */
        MySeqSt ◆ []; /* initialize to the empty sequence */
        case p of
            EMPT:
                outSt.StateVal ◆ 0;
                outSt.accepts ◆ TF;
            {x}:
                outSt.accepts ◆ (inSt.StateVal = 1);
                outSt.StateVal ◆ if TF and (Char in {x}) then 1 else 0;
            ANYSTR:
                outSt.StateVal ◆
                    if TF or (inSt.StateVal = 1) then 1 else 0;
                outSt.accepts ◆ (outSt.StateVal = 1);
            (SEQ p1 p2):
                MySeqSt[0] ◆ Next(p1, L(inSt.SubStates)[0], Char, TF);
                MySeqSt[1] ◆ Next(p2, L(inSt.SubStates)[1], Char,
                    MySeqSt[0].accepts);
                outSt.SubStates ◆ T(MySeqSt);
                outSt.accepts ◆ MySeqSt[1].accepts;
            (OR p1 p2):
                MySeqSt[0] ◆ Next(p1, L(inSt.SubStates)[0], Char, TF);
                MySeqSt[1] ◆ Next(p2, L(inSt.SubStates)[1], Char, TF);
                outSt.SubStates ◆ T(MySeqSt);
                outSt.accepts ◆ MySeqSt[0].accepts or MySeqSt[1].accepts;
            (* p1):
                tempSt ◆ Next(p1, L(inSt.SubStates)[0], Char, FALSE);
                MySeqSt[0] ◆ Next(p1, L(inSt.SubStates)[0], Char,
                    TF or tempSt.accepts);
                outSt.SubStates ◆ T(MySeqSt);
                outSt.accepts ◆ TF or tempSt.accepts;
            (DEFCALL p1):
                increment RecCount(p);
                if RecCount(p) > MAX_RECURSION then
                    outSt.StateVal ◆ 0;
                    outSt.accepts ◆ FALSE;
                else
                    MySeqSt[0] ◆
                        Next(p1, L(inSt.SubStates)[0], Char, TF);
Appendix B: A Formal Discussion of Next()

outSt.SubStates ♦ T(MySeqSt);
outSt.accepts ♦ MySeqSt[0].accepts;
decrement RecCount(p);

(TRANSLA p1):
    newChar ♦ smallest character in the
             equivalence class of p1 containing Char;
    outSt ♦ Next(p1, inSt, newChar, TF);

(CSLP p1 p2 {x}):
    MySeqSt[0] ♦ Next(p1, L(inSt.SubStates)[0], Char, TRUE);
    MySeqSt[1] ♦ Next(p2, L(inSt.SubStates)[1], Char,
                   TF and MySeqSt[0].accepts);
    outSt.SubStates ♦ T(MySeqSt);
    outSt.accepts ♦ MySeqSt[1].accepts and Char in {x};

(MINI-AND p1 p2):
    MySeqSt[0] ♦ Next(p1, L(inSt.SubStates)[0], Char, TF);
    MySeqSt[1] ♦ Next(p2, L(inSt.SubStates)[1], Char, TF);
    outSt.SubStates ♦ T(MySeqSt);
    outSt.accepts ♦ MySeqSt[0].accepts and
                   MySeqSt[1].accepts;

(MINI-NOT p1):
    MySeqSt[0] ♦ Next(p1, L(inSt.SubStates)[0], Char, TF);
    outSt.SubStates ♦ T(MySeqSt);
    outSt.accepts ♦ not MySeqSt[0].accepts;

(SET_SHORTER p1):
    CMP ♦ '<'; ALL ♦ FALSE; goto label (SET_ p1);

(SET_LONGER p1):
    CMP ♦ '>'; ALL ♦ FALSE; goto label (SET_ p1);

(SET_SHORTEST p1):
    CMP ♦ '<'; ALL ♦ TRUE; goto label (SET_ p1);

(SET_LONGEST p1):
    CMP ♦ '>'; ALL ♦ TRUE; goto label (SET_ p1);
(SET p1):
   /** remove DEAD_STATEs **/
   TempSeqSt ♦ L(inSt.SubStates) with
   all DEAD_STATEs deleted;
   in_size ♦ TempSeqSt.size;

   /** calculate next states **/
   for i from 0 to in_size-1 do
      MySeqSt[i] ♦ Next(p1, TempSeqSt[i], Char, FALSE);
      MySeqSt[i].marked ♦ TempSeqSt[i].marked;

   /** take care of new-arrival **/
   if (ALL or TF) then
      MySeqSt[in_size] ♦
      Next(p1, TempSeqSt[0], Char, TRUE);
      MySeqSt[in_size].marked ♦ TF;

   /** zero younger or older duplicates **/
   for all i, j such that (i CMP j) do
      if (MySeqSt[i] = MySeqSt[j] except for the
         marked fields, which may differ) then
         MySeqSt[i] ♦ DEAD_STATE;

   OutSt.SubStates = T(MySeqSt);

   /** calculate accepts **/
   OutSt.accepts ♦ ∃i:(MySeqSt[i].accepts and
      no j such that (i CMP j) has
      MySeqSt[j].accepts and
      MySeqSt[i].marked);

end case;

Insert_Into_Transition_Cache(p, inSt, Char, TF, OutSt);

end function.
function InitialState(p) returns outSt;
begin
  outSt.SubStates ♦ nil; /* default value */
  outSt.StateVal ♦ 0; /* default value */
  MySeqSt ♦ []; /* initialize to the empty sequence */
  outSt.inp ♦ FALSE;
  outSt.accepts ♦ FALSE;

  case p of:
    (CSLP p1 p2):
    (OR p1 p2):
    (MINI-AND p1 p2):
      (SEQ p1 p2):
        MySeqSt[0] ♦ InitialState(p1);
        MySeqSt[1] ♦ InitialState(p2);
        OutSt.SubStates ♦ T(MySeqSt);
      (DEFCALL p1):
        increment RecCount(p);
        if RecCount(p) > MAX_RECURSION then
          OutSt.StateVal ♦ 0;
        else
          MySeqSt[0] ♦ InitialState(p1);
          OutSt.SubStates ♦ T(MySeqSt);
          decrement RecCount(p);
    (MINI-NOT p1):
      (* p1):
      (SET_SHORTEST p1):
      (SET_SHORTER p1):
      (SET_LONGEST p1):
      (SET_LONGER p1):
        MySeqSt[0] ♦ InitialState(p1);
        OutSt.SubStates ♦ T(MySeqSt);
    EMPTY:
    {x} :
    ANYSTR:
      OutSt.StateVal ♦ 0;
  end case;
end function;
Appendix C
A Formal Discussion of Parse Extraction

C.1. Defining the Problem Formally

Assuming that \{S\}..n matches p, we define a function \text{Parse0}(\{S\}, p, n) returning an integer \( N \) in \{S\} such that \( N..n \) matches \( p \). Actually, in order to write a recursive algorithm for \text{Parse0}(\{S\}, p, n) we have to define a more general function:

\textbf{Definition C.1:Parse1:}

Assuming \( p \) is a subpattern of \( PP \) but \( p \) is not a proper subpattern of a \text{SET}_\text{-} pattern; if \( p \) is a \text{MINI\text{-}AND} or \text{MINI\text{-}NOT} then \text{SAFE}(p); \{S\}..n \) matches \text{Postfix}(p, PP); then

\text{Parse1}(\{S\}, p, PP, n) returns \( N \) such that \{N\}..n matches \( p \) and \{S\}..N matches \text{Prefix}(p, PP)

\textbf{end Definition.}

A word about the assumptions: the first two rather unintuitive restrictions arise because \text{Next()} handles \text{SET}_\text{-} patterns differently than other patterns, and \text{SET}_\text{-} patterns are crucial for using \text{MINI\text{-}AND} and \text{MINI\text{-}NOT}. (This explains nothing, but hopefully it suggests why special treatment of these patterns can be expected.) The first assumption is used in evolving \text{Parse1()} to \text{Parse()} (which happens later). The second assumption is used to prove the \text{MINI\text{-}AND} and \text{MINI\text{-}NOT} cases of \text{Parse1()}. Essentially, it codifies the practice of surrounding each \text{MINI\text{-}AND} and \text{MINI\text{-}NOT} with a \text{SET}_\text{-} pattern, because doing so ensures this assumption. Thus the
second assumption is not at all restrictive in using Parse1(). The third assumption is necessary to ensure the existence of an N for Parse1() to return.

A special case of Parse1(), namely Parse1({S}, PP, PP, n), is equivalent to Parse0(). Prefix(PP, PP) = EMPTY, Postfix(PP, PP) = PP, and thus

Parse1({S}, PP, PP, n) returns an N such that

N..n matches PP and

[S]..N matches EMPTY

Since [S]..N matches EMPTY is the same as (N in {S}), we see

Parse0({S}, PP, n) = Parse1({S}, PP, PP, n)

C.2. Solving the Problem

Like all good algorithms, Parse1() can be recursively specified.

function Parse1({S}, p, PP, n) returns N;
begin
  case p of
    EMPTY:
      N ♦ n;
    {x}:
      N ♦ n-1;
    ANYSTR:
      if {S}..n matches Prefix(p, PP) then N ♦ n
      else N ♦ Parse1({S}, p, PP, n-1)
    (SEQ p1 p2):
      N ♦ Parse1({S}, p1, PP, Parse1({S}, p2, PP, n))
    (OR p1 p2):
      if {S}..n matches Postfix(p1, PP) then
        N ♦ Parse1({S}, p1, PP, n);
      else
        N ♦ Parse1({S}, p2, PP, n);
    (* p1):
      if (prefer more loops) then
        if {S}..n matches Postfix(p1, PP) then
          N ♦ Parse1({S}, p, PP, Parse1({S}, p1, PP, n));
else
    N ♦ n;
else (prefer less loops)
    if {S}..n matches Prefix(p, PP) then
        N ♦ n;
    else
        Parse1({S}, p, PP, Parse1({S}, p1, PP, n))
    end case;
end;

The reader may notice that many lines are not very algorithmic; for example, the MINI-NOT case and the SET_ case. This will be remedied in the next version of Parse(), which is the subject of the following section.

Proof of Correctness of Parse1():

The proof is a structural induction on patterns. We prove correctness of the ANYSTR, EMPTY, and {x} cases, and then we prove the correctness of the other cases by assuming that Parse1() behaves correctly during the recursive calls to subpatterns. Each case can assume that {S}..n matches Postfix(p, PP) upon entry, but each case must ensure that {S}..n' matches Postfix(p1, PP) before any recursive
call to Parse1([S], p1, PP, n'). For brevity we only include the proofs for a few interesting cases.

- **case p = \{x\}:**

  Since \{S\}..n matches Postfix(\{x\}, PP), Definition A.4:Postfix tells us
  \{S\}..n matches (SEQ Prefix(\{x\}, PP) \{x\})

  which, when we invoke Theorem B.1:RecursiveMatch and manipulate, tells us
  \{S\}..n-1 matches Prefix(p, PP) and n-1..n matches p

  Therefore, n-1 is an excellent value to return from Parse1().

- **case p = (SEQ p1 p2):**

  Since \{S\}..n matches Postfix(p, PP), Lemma A.4:FixFormula(5) ensures that we may call Parse1([S], p2, PP,n). This call will return an N' such that \{S\}..N' matches Prefix(p2, PP) and N'..n matches p2. Lemma A.4:FixFormula(3) then ensures that we may call Parse1([S], p1, PP, N') with this N'. This call returns an N such that N..n matches p1 and \{S\}..N matches Prefix(p1, PP). Lemma A.4:FixFormula(4) then shows that

  \{S\}..N matches Prefix((SEQ p1 p2), PP)

  Since N..N' matches p1 and N'..n matches p2, Definition A.1:PatternOperators ensures that

  N..n matches (SEQ p1 p2)

- **case p = (*) p1:**
Appendix C: A Formal Discussion of Parse Extraction

We will not prove the correctness of this case, since it is similar to the SEQ case. For the interested reader, though, we give a hint: Lemma A.4:FixFormula(12) is used to prove the following important fact:

\[ \{S\}_{..n} \text{ matches } \text{Postfix}(\{p\}_{1}), \text{PP} \] and
\[ \text{not } \{S\}_{..n} \text{ matches } \text{Postfix}(p_{1}, \text{PP}) \]

\[ \{S\}_{..n} \text{ matches } \text{Prefix}(\{p\}_{1}), \text{PP} \]

o case \( p = (\text{DEFCALL } p_{1}) \):

The interesting aspect of this case is that we do not check the RecCount fields. Intuitively, we will never call Parse1() on a DEFCALL unless there is a backparse through \( p_{1} \). Since this only occurs when \( p_{1} \) is not \( {} \), we never have to worry about RecCount.

More formally, if \( \{S\}_{..n} \text{ matches } \text{Postfix}(\{\text{DEFCALL } p_{1}\}, \text{PP}) \) then
\[ \{S\}_{..n} \text{ matches } \text{SEQ Prefix}(p, \text{PP}) (\text{DEFCALL } p_{1}) \]

The pattern on the right cannot match anything if RecCount(\( p \)) > MAX_RECURSION, because in this case \( \{\text{DEFCALL } p_{1}\} = {} \). This contradicts the assumption upon entering Parse1(), which shows that RecCount(\( p \)) \( \leq \) MAX_RECURSION. Thus there is no need to increment and decrement the RecCount field.

o case \( p = (\text{MINI-NOT } p_{1}) \) or \( (\text{MINI-AND } p_{1} p_{2}) \)

In the MINI-NOT case, notice that Parse1() does not parse through \( p_{1} \) recursively. The reason is simple: \( \{T\}..n \text{ matches } \text{MINI-NOT } p_{1} \) exactly when there are no possible parses of \( \{T\}..n \) through \( p_{1} \).

Therefore it is meaningless to talk about a parse through the \( p_{1} \) pattern of a MINI-NOT.
Appendix C: A Formal Discussion of Parse Extraction

The MINI-AND case best illustrates the need for the second assumption in Definition C.1: Parse1: SAFE(p), which really ensures that whenever Next() calculates \( \{T\}..k \) matches (MINI-AND p1 p2), \( \{T\} \) has at most one element. Section B.9.2 contains the discussion of SAFE(). Now we illustrate the use of the assumption in backparsing through a MINI-AND.

One expects that the Parse1() calls to p1 and p2 should both return the same N. If they do not, which is the correct one to return? Fortunately, the assumption SAFE(p) establishes that the same N will be returned. Calling the values returned by the two recursive calls \( N_I \) and \( N_2 \), respectively, after both calls we know

\[
\{N_I\}..n \text{ matches } p1 \text{ and } \{S\}..N_I \text{ matches } \text{Prefix}(p_1, PP) \quad \text{and} \\
\{N_2\}..n \text{ matches } p2 \text{ and } \{S\}..N_2 \text{ matches } \text{Prefix}(p_2, PP)
\]

We want to find an N such that

\[
\{N\}..n \text{ matches } (\text{MINI-AND } p1 \ p2) \text{ and } \{S\}..N \text{ matches } \text{Prefix}(p, PP)
\]

which Theorem B.1: RecursiveMatch converts to

\[
\{N\}..n \text{ matches } p1 \text{ and } \{N\}..n \text{ matches } p2 \text{ and} \\
\{S\}..N \text{ matches } \text{Prefix}(p, PP)
\]

The third predicate is easy: if \( N = N_I \) or \( N = N_2 \), Lemma A.4:FixFormula(8) establishes it directly. However, the first two predicates are TRUE if

\[
N = N_I = N_2
\]

SAFE(p) ensures this because any computation of \( \{T\}..n \) matches p will have a \( \{T\} \) with at most 1 element. In our case

\[
\{T\} = \{i \mid \{S\}..i \text{ matches } \text{Prefix}(p, PP)\}
\]
Appendix C: A Formal Discussion of Parse Extraction

Since both $N_1$ and $N_2$ are in \{T\}, and since \{T\} has at most 1 element, \{T\} must have exactly 1 element equal to both $N_1$ and $N_2$.

end Proof of Correctness of Parse1().

C.3. Solving the Problem Again

In this section we apply a correctness preserving transformation to Parse1() to derive a function Parse() which does the same thing. However, Parse() will be a true executable algorithm.

One suspicious feature of Parse1() is that it makes no appreciable use of the sequence of states $q$ created during matching. $q$ is a sequence of states for which

$$q_k = \text{Next}(PP, q_{k-1}, s_{k-1}, k \text{ in } \{S\}) \text{ and}$$

$q_k$.accepts iff $\{S\}..k$ matches $PP$ and

$q_k$.inp iff $k \text{ in } \{S\}$

Furthermore, a state $q_k$ of a pattern $PP$ encodes the state of all the subpatterns of $PP$. We define the function $\text{Proj}(q_k, p, PP)$ to return the state of subpattern $p$ when $PP$ is in state $q_k$. Here is a formal definition.

Definition C.3: Proj:

Assuming that $p_1$ is a subpattern of $PP$; that $p_1$ is not a strict subpattern of a \textsc{set}_ pattern in $PP$; then

$\text{Proj}(q_k, p_1, PP)$ is $q_k$ when $p_1 = PP$

$\text{Proj}(q_k, p_1, PP)$ is $\text{Proj}(q_k, \text{parent}(p_1), PP).q_1$

when $\text{parent}(p_1) = (\text{SEQ} p_1 p_2), (\text{OR} p_1 p_2), (* p_1), (\text{MINI-AND} p_1 p_2), (\text{DEFCALL} p_1), (\text{MINI-NOT} p_1)$
Appendix C: A Formal Discussion of Parse Extraction

\[ \text{Proj}(q_k, p_2, PP) = \text{Proj}(q_k, \text{parent}(p_1), PP).q_2 \]

when \( \text{parent}(p_1) = (\text{SEQ} p_1 p_2), (\text{OR} p_1 p_2), (\text{MINI-AND} p_1 p_2), (\text{CTLP} p_1 p_2 p_3) \)

(Notice that this is a recursive definition which goes from patterns to the patterns that contain them. Thus the base case is \( \text{Proj}(q_k, PP, PP). \))

end Definition.

Now we give the most important theorem in this section, which is actually a fact about \( \text{Next}() \). It describes the interpretation of \( \text{Proj}(q_k, p, PP).\inp \) and \( \text{Proj}(q_k, p, PP).\accepts \):

**Theorem C.3: ProjMeaning:**

Assuming that \( p \) is a subpattern of \( PP; \)
that \( p \) is not a strict subpattern of any SET_ pattern in \( PP; \)
\( q \) is a sequence of states, and for any \( k \)
\[ q_k = \text{Next}(PP, q_{k-1}, s_k, k \in \{S\}) \]
\[ q_k.\text{accepts} \iff \{S\}..k \text{ matches } PP \text{ and } q_k.\text{inp} \iff k \in \{S\}; \]
then

(1) \( \forall k: \text{Proj}(q_k, p, PP) = \text{Next}(p, \text{Proj}(q_{k-1}, p, PP), s_k, k \in \{T\}) \)
where \( (k \in \{T\}) = \{S\}..k \text{ matches } \text{Prefix}(p, PP) \)
(thus \( \{T\} = \{i | \{S\}..i \text{ matches } \text{Prefix}(p, PP)\} \))

(2) \( \text{Proj}(q_k, p, PP).\inp = \{S\}..k \text{ matches } \text{Prefix}(p, PP) \)

(3) \( \text{Proj}(q_k, p, PP).\accepts = \{S\}..k \text{ matches } \text{Postfix}(p, PP) \)

end Theorem.

Proof:

Actually, conclusions (2) and (3) follow from (1) as follows: Definition B.1:Corr applied to (1), with \( \text{Proj}() \) substituted for \( q_j \) and \( \{T\} \)
substituted for \( \{S\} \), gives

\[ \text{Proj}(q_k, p, PP).\inp / k \in \{T\} \]
\[ \text{Proj}(q_k, p, PP).\accepts / \{T\}..k \text{ matches } p \]
Substituting the definition of \( \{ T \} \), simplifying, and applying Definition A.4:Postfix, converts the first result to (2) and the second result to (3).

Proving (1) requires an induction over all patterns \( p \). The base case is \( p = \text{PP} \); substituting \( \text{PP} \) for \( p \) and using Definition A.4:Prefix and Definition A.4:Postfix, (1) becomes a copy of the assumption, and the base case is proved.

The inductive case of (1) requires proving, for each type of pattern \( p \) (except for \( \text{SET}_- \) patterns), that (1) is true on the subpatterns of \( p \) when (1) is true on \( p \). As an example we carry out the proof for \( p = (\text{SEQ} \ p_1 \ p_2) \). Proofs for the other cases are omitted, but their proofs are very similar.

Since we may assume (1) holds on \( p \), we may reuse the results of section B.2 (where the correctness of \( \text{Next}((\text{SEQ} \ p_1 \ p_2),...) \) was proved), except that we substitute \( \text{Proj}(q_k, p, \text{PP}) \) for \( q_k \), \( \text{Proj}(q_{k-1}, p, \text{PP}) \) for \( q_{k-1} \), and \( \{ T \} \) for \( \{ S \} \). Remember that \( p \) is \( (\text{SEQ} \ p_1 \ p_2) \). Rewriting the appropriate lines of \( \text{Next}() \), as we did in section B.2, produces this:

```
function Next((SEQ \ p_1 \ p_2), Proj(q_{k-1}, p, \text{PP}), s_k, k \in \{ T \})
    returns Proj(q_k, p, \text{PP}): 
    1|        Proj(q_k, p, \text{PP}).\text{inp} \& k \in \{ T \};
    21|        Proj(q_k, p, \text{PP}).\text{q1} \&
    2|            \text{Next}(p_1, Proj(q_{k-1}, p, \text{PP}).\text{q1}, s_k, k \in \{ T \});
    22|        Proj(q_k, p, \text{PP}).\text{q2} \&
    3|            \text{Next}(p_2, Proj(q_{k-1}, p, \text{PP}).\text{q2}, s_k,
    23|            Proj(q_k, p, \text{PP}).\text{q1}.\text{accepts});
    4|        Proj(q_k, p, \text{PP}).\text{accepts} \& Proj(q_k, p, \text{PP}).\text{q2}.\text{accepts};
```
Appendix C: A Formal Discussion of Parse Extraction

These should be viewed as a set of linked equations, true for all k.

Only lines 21 and 22 are of interest; to these lines we apply Definition
C.3:Proj to get

\[
\begin{align*}
21 & \quad \text{Proj}(q_k, p_1, PP) \; \diamond \; \\
& \quad \text{Next}(p_1, \text{Proj}(q_{k-1}, p_1, PP), s_k, k \in \{T\})
\end{align*}
\]

\[
\begin{align*}
22 & \quad \text{Proj}(q_k, p_2, PP) \; \diamond \; \\
& \quad \text{Next}(p_2, \text{Proj}(q_{k-1}, p_2, PP), s_k, \\
& \quad \quad \text{Proj}(q_k, p_1, PP).\text{accepts})
\end{align*}
\]

We are practically finished. Line 21 becomes (1) (with p1 for p) if we can show that

\[(k \in \{T\}) \div (k \in \{T'\}),\]

where \((k \in \{T\}) \div \{S\}.k \text{ matches } \text{Prefix}(p, PP)\)

where \((k \in \{T'\}) \div \{S\}.k \text{ matches } \text{Prefix}(p_1, PP)\)

Since p is (SEQ p1 p2), the truth of this is found in Lemma A.4:FixFormula(4).

Line 22 becomes (1) (with p2 for p) if we can show that

\[\text{Proj}(q_k, p_1, PP).\text{accepts} \div \{S\}.k \text{ matches } \text{Prefix}(p_2, PP)\]

The previous analysis of line 21 allows us to conclude that

\[\text{Proj}(q_k, p_1, PP).\text{accepts} \div \{S\}.k \text{ matches } \text{Postfix}(p_1, PP)\]

Then a simple application of Lemma A.4:FixFormula(3) completes the proof:

\[\text{Proj}(q_k, p_1, PP).\text{accepts} \div \{S\}.k \text{ matches } \text{Prefix}(p_2, PP)\]

end Proof of Theorem C.3:ProjMeaning.

Thanks to Theorem C.3:ProjMeaning, we can replace all the lines of Parse1() that read “if \{S\}.n \text{ matches } \text{Postfix}(p_1, PP)” by the equivalent “if \text{Proj}(q_n, p_1, PP).\text{accepts}”.
Similarly, “if \( \{S\}..n \) matches Prefix(p, PP)” becomes

“if Proj(q_n, p, PP).inp”

These transformations suffice to remove all non-algorithmic lines from Parse1(), except for the lines in the SET_ case. As promised, we must specify and prove the correctness of the function NewStates(q, p, PP, n). Here is the specification for NewStates():

**Definition C.3: NewStates:**

Assuming p, PP, q, \( \{S\} \), n satisfy the assumptions of Parse1; p is a subpattern of PP with the form (SET_ p1); then

NewStates(q, p, PP, n) returns a sequence of states qq such that

1. \( \text{qq}_k \) is a state of p1 and
2. \( \text{qq}_k = \text{Next}(p1, \text{qq}_{k-1}, s_k, k \text{ in } \{N\}) \);
3. \( \{N\} \) is a set of positions with only 1 element, N;
4. \( \{S\}..N \) matches Prefix(p, PP);
5. \( \{N\}..n \) matches p;
6. \( \{N\}..n \) matches p1,

**end Definition.**

If NewStates() behaves according to the definition, the recursive call in the SET_ case of Parse() is correct. First, the preconditions for calling Parse() are established by (6) (which ensures that \( \{N\}..n \) matches Postfix(p1, p1)) and (1) and (2). The N’ returned by the recursive call must be the same N mentioned in the definition. To see this, note that Parse(\( \{N\}, p1, p1, n \)) returns an N’ such that \( \{N’\}..n \) matches p1 and \( \{N\}..N’ \) matches Prefix(p1, p1), which is EMPTY; then \( \{N\}..N’ \) matches EMPTY is equivalent to \( N’ = N \). Finally, (4) and (5) are exactly the requirements of an N returned by Parse(q, (SET_ p1), PP, n).

To understand the NewStates() algorithm we will show, more intuitively than formally, how it is derived. The reader is encouraged
to review the discussion of \texttt{Next((SET\_LONGEST...))} and \texttt{Next((SET\_LONGER...))} in sections B.6 and B.7.

The challenge of \texttt{NewStates()} springs from the fact that each state $Q_k$ of a (SET\_ p1) pattern has many substates for p1, in $Q_k.qseq$. Each $qq_k$ is one of substates in $Q_k.qseq$ - but which one? Finding $qq_n$ is easy, because consequences (1), (2), and (6) imply that $qq_n.\text{accepts}$ must be TRUE. (One of $Q_n.qseq[i].\text{accepts}$ is TRUE because $Q_n.\text{accepts}$ is TRUE. The latter is true because of Theorem C.3:ProjMeaning and because $\{S\}..n$ matches $\text{Postfix((SET\_ p1), PP)}$.)

Given a $qq_k$ which is found in $Q_k.qseq[i]$, we describe how to find $qq_{k-1}$ in $Q_{k-1}.qseq$. Consequence (2) is the key. It implies that

$$Q_k.qseq[i] = \text{Next(p1, } Q_{k-1}.qseq[j], s_k, \text{ k in \{N\} ) for some } j$$

Exactly which $j$ can be determined by closely examining the SET\_ case of \texttt{Next()}. It is the $j$ such that if all DEAD\_STATEs are deleted from $Q_{k-1}.qseq$, the state at index $j$ becomes the state at index $i$.

(Importantly, this observation also ensures that $Q_{k-1}.qseq[j]$ is never a DEAD\_STATE.) Define $\text{Back}(Q_{k-1}.qseq, i)$ to return this $j$. We conclude that if $qq_k$ is found in $Q_k.qseq[i]$, then

$$qq_{k-1} \text{ is found in } Q_{k-1}.qseq[\text{Back}(Q_{k-1}.qseq, i)].$$

The reason for keeping DEAD\_STATEs is to make it fast and easy to compute $\text{Back()}$. Function $\text{Back}(Qseq, i)$ can be calculated as (the index of the ith non-DEAD\_STATE in Qseq). It can be calculated in constant time for a suitable representation of Qseq. However, TLex implements $\text{Back()}$ using time proportional to the length of Qseq, by looping through Qseq and counting non-DEAD\_STATEs. In practice
Qseq is short and the overhead of the loop is very small so this is a workable solution.

When is it safe to stop? Actually, we only need $qq_N \ldots qq_n$, because by examining Parse() one can see that if Parse() returns $N$, it never accesses a $q_i$ such that $i < N$. This is an important optimization. However, how do we know which state is $qq_N$? Just as $qq_n$ was identified as a state in $Q_n.qseq$ for which the accepts field was TRUE, $qq_N$ is a state for which the inp field is TRUE. Therefore we stop when $qq_k.inp$ is TRUE, and then we know that this $k$ is $N$.

The algorithm for NewStates() given in section 7.1 is essentially a transcription of the previous discussion.

To show that the $\{N\}$ of Definition C.3:NewStates is a set with exactly 1 element, recall the correctness proof for Next((SET_ p1)...). Each state $q_{temp_j}.qseq[i]$ calculates $\{m\}..j$ matches $p_1$ for some set $\{m\}$ of one element. Thus the particular sequence we select will also calculate $\{N\}..j$ matches $p_1$ for a set $\{N\}$ of one element.

$\{S\}..N$ matches prefix($p$, PP), consequence (5), is TRUE because $qq_N.inp$ is TRUE, which implies that Proj($q_N$, $p$, PP).inp is TRUE, which implies (5).

We still must show that these patterns prefer the shorter and longer matches that permit the rest of the pattern to parse. We argue the SET_LONGER case; SET_SHORTER is analogous. Function NewStates() prefers longer matches when $p$ is SET_LONGER for a simple reason. When $qq_n$ is selected, $q_{temp_n}.qseq$ may have an
accepting state at index i representing \{m1\}..n matches p1, and an accepting state at index j representing \{m2\}..n matches p1. Assume that m1 is the least such position. Next((SET_LONGER p1)...)) ensures that i < j if m1 < m2, and NewStates() chooses the least index, namely i. Thus the least m1 is always selected, and thus the longer match is preferred, as desired.

Since \{S\} is no longer explicitly mentioned in Parse(), its parameter can be dropped. We replace it with q, a sequence of states. Also, we replace some recursive calls by equivalent iterations. The new function, Parse(), is the result of applying these transformations, whose correctness we just verified, to Parse1(). Thus Theorem C.3:Parse needs no additional proof.

**Theorem C.3:Parse:**

Assuming p, PP, \{S\}, and n are as assumed in Definition C.1:Parse1; q is a sequence of states, and for any k

\[q_k = \text{Next}(PP, q_{k-1}, s_k, k \in \{S\});\]

then

\[\text{Parse1}\{(S), p, PP, n\} = \text{Parse}(q, p, PP, n)\]

end Theorem.

The function Parse() is listed in section 7.1.

**C.4. Optimizing Parse()**

Parse() is fairly simple and efficient, but the calls to Proj() can be optimized. We will introduce a new parameter to Parse(), PS, of type ProjStack. PS is passed by reference, unlike all other parameters, so a call to Parse(...PS...) or a call to NewStates(...PS...) can modify PS. A
ProjStack is basically a stack of tuples \(<\text{State, OneTwo}>\). The State component is a state, the OneTwo component is a Boolean. A ProjStack supports the following (constant time) operations:

- **NewPS()** returns an empty stack.
- **TopPS(ProjStack)** returns the tuple at the top of the stack but does not modify ProjStack.
- **LengthPS(ProjStack)** returns the number of tuples in the stack.
- \(PS[i]\) accesses the \(i\)th tuple in the stack, where the TopPS() tuple is at index LengthPS().
- **PushPS(state, OneTwo, ProjStack)** modifies ProjStack by pushing a tuple \(<\text{state, OneTwo}>\) on top.
- **SoftPushPS(state, OneTwo, ProjStack)** returns a new ProjStack equal to (ProjStack with tuple \(<\text{state, OneTwo}>\) pushed on top).
- **PopPS(ProjStack)** modifies ProjStack by popping the top tuple.
- **SoftPopPS(ProjStack)** returns a new ProjStack equal to ProjStack with the top tuple popped off.

The operations with a “Soft” prefix are non-destructive routines which are used only in definitions. The purpose of PS is to take advantage of the incremental computation of Proj() expressed by Definition C.3:Proj. It also speeds calculation of Proj() when no incremental computation is possible. The following predicate describes how PS and Proj() relate.
Predicate C.4: InvProjStack(S, p, PP, PS):

(1) \(\text{TopPS(PS).State has Proj(S, p, PP)}\)

(2) \(\text{TopPS(PS).OneTwo is TRUE if SoftPopPS(PS).State.q1 has TopPS(PS).State}\)

(3) \(\text{TopPS(PS).OneTwo is FALSE if SoftPopPS(PS).State.q2 has TopPS(PS).State}\)

(4) \(\text{InvProjStack(S, Parent(p), PP, PS) unless p = PP}\)

end Predicate.

The routine RefreshPS(PS, S), which we list now, assumes that PS is a ProjStack satisfying InvProjStack(SomeS, p, PP, PS) for some random state SomeS; it efficiently establishes InvProjStack(S, p, PP, PS) and returns the modified PS.

\[
\text{function RefreshPS(PS, S) returns PS; begin}
\]

\[
\text{PS[1].State} \in S;
\]

\[
\text{for i = 2 to LengthPS(PS) do}
\]

\[
\text{PS[i].State} = \text{if PS[i].OneTwo then PS[i-1].State.q1 else PS[i-1].State.q2}
\]

end.

The following invariant describes how PS relates to the Parse() function:

Invariant C.4: PS:

Upon entry to Parse(q, p1, PP, PS, n), PS satisfies InvProjStack(qn, p1, PP, PS).

Upon returning N from Parse(q, plsub PP, PS, n), PS satisfies InvProjStack(qn, pl, PP, PS), where plsub is an immediate subpattern of pl.

end Invariant.

To understand more simply the reason for Invariant PS, consider the following corollary of Invariant C.4: PS:
Corollary C.4: QuickProj:

Whenever InvProjStack(q_m, p, PP, PS) is TRUE:

- Proj(q_m, p, PP) may be calculated by TopPS(PS).State.
- Proj(q_m, p1, PP) for p1 the p1 subpattern of p may be calculated by TopPS(PS).State.q1.
- Proj(q_m, p2, PP) for p2 the p2 subpattern of p may be calculated by TopPS(PS).State.q2.
- InvProjStack(q_n, p, PP, PS) is TRUE upon entry to Parse()
- InvProjStack(q_N, p, PP, PS) is TRUE after a call to Parse() returns N.
- InvProjStack(q_N, p, PP, PS) is TRUE after calling RefreshPS(PS, q_N).

end Corollary.

Thus PS enables Proj() to be calculated in constant time in most cases. The functions GO1() and GO2() provide concise ways to call Parse() recursively and maintain Invariant PS.

Definition C.4: GO1 and GO2:

GO1(PS) is a function which executes
PushPS(TopPS(PS).State.q1, TRUE) and then returns the modified PS.

GO2(PS) is a function which executes
PushPS(TopPS(PS).State.q2, FALSE) and then returns the modified PS

end Definition.

Corollary C.4: GO:

Assuming we are inside a call to Parse(SomeQ, p, PP, PS, n1); p1 and p2 are the p1 and p2 subpatterns of p; p is not a SET_ pattern; InvProjStack(q_m, p, PP, PS) is TRUE; then

Calling Parse(q, p1, PP, GO1(PS), m) maintains Invariant C.4:PS;
Calling Parse(q, p2, PP, GO2(PS), m) maintains Invariant C.4:PS;

end Corollary.
C.4.1. The Optimized Version of Parse()

Using the previous results, we can rewrite Parse() and NewStates() as follows:

```plaintext
function Parse(q, p, PP, PS, n) returns N;
begin
    case p of
    EMPTY:
    N := n;
{x}:
    N := n - 1;
    RefreshPS(PS, q_N);
    ANYSTR:
    N := n;
    until TopPS(PS).State.inp do
        N := N - 1;
        RefreshPS(PS, q_N);
    (SEQ p1 p2):
    N := Parse(q, p1, PP, GO1(PS),
               Parse(q, p2, PP, GO2(PS), n));
    (OR p1 p2):
    if TopPS(PS).State.q1.accepts then
        N := Parse(q, p1, PP, GO1(PS), n);
    else
        N := Parse(q, p2, PP, GO2(PS), n);
    (* p1):
    N := n;
    if (prefer more loops) then
        while TopPS(PS).State.q1.accepts do
            N := Parse(q, p1, PP, GO1(PS), N);
    else (prefer less loops)
        until TopPS(PS).State.inp do
            N := Parse(q, p1, PP, GO1(PS), N);
    (DEFCALL p1):
    N := Parse(q, p1, PP, GO1(PS), n);
    (MINI-NOT p1):
    N := n;
    until TopPS(PS).State.inp do
        N := N - 1;
        RefreshPS(PS, q_N);
    (MINI-AND p1 p2):
    N := Parse(q, p2, PP, GO2(PS), n);
    N := Parse(q, p1, PP, GO1(PS), n);
    (CSLP p1 p2 p3):
```
Appendix C: A Formal Discussion of Parse Extraction

N ♦ Parse(q, p2, PP, GO2(PS), n);

(SET_LONGEST p1):
(SET_LONGER p1):
(SET_SHORTEST p1):
(SET_SHORTER p1):
newqq ♦ NewStates(q, p, PP, PS, n);
PSnew ♦ PushPS(newqq, FALSE, NewPS());
N ♦ Parse(newqq, p1, p1, PSnew, n);

end case;
PopPS(PS);
end;

function NewStates(q, p, PP, PS, n) returns qq;
begin
/** select the last state **/
qtemp ♦ TopPS(PS).State;
if p is (SHORTER p1) or (SHORTEST p1) then
  lasti ♦ the largest index i such that
  qtemp.qseq[i].accepts
else if p is (LONGER p1) or (LONGEST p1) then
  lasti ♦ the smallest index i such that q
  qtemp.qseq[i].accepts
qqn ♦ qtemp.qseq[lasti]
/** select preceding states **/
N ♦ n;
while (! qqN.inp) do
  begin
    N ♦ N - 1;
    RefreshPS(PS, qqN);
    qtemp ♦ TopPS(PS).State;
    lasti ♦ Back(qtemp.qseq, lasti);
    qqN ♦ qtemp.qseq[lasti];
  end;
end;

C.4.2. Correctness of the Optimized Parse()

Using the results developed above, it is easy to show that the new
versions of Parse() and NewStates() calculate the same thing as the
old. The correctness argument will have two parts. First we show that
Invariant C.4:PS is maintained. Then we use the correctness
preserving transformations of Corollary C.4:QuickProj to replace the Proj() calls by more efficient procedures.

First, we show that Parse() maintains Invariant C.4:PS. All calls to Parse() now include a new parameter, PS. Any call to a p1 or p2 subpattern passes GO1(PS) or GO2(PS) respectively. As argued in Corollary C.4:GO, this maintains Invariant PS on the subpatterns of p. (Actually, the SET_ patterns are an exception which will be discussed separately.) Each case of Parse() will ensure InvProjStack(qN, p, PP, PS). For example, the patterns {x}, MINI-NOT, and ANYSTR which assign to N call RefreshPS(PS, qN) directly after, which establishes InvProjStack(). The other cases assign to N from a recursive Parse() call; since we just showed that Invariant C.4:PS is true for recursive calls, InvProjStack() is established as a result of Invariant C.4:PS. At the end of Parse() each case executes the PopPS() statement. The combination of InvProjStack(qN, p, PP, PS) and PS ♦ PopPS(PS) assures that InvProjStack(qN, parent(p), PP, PS) is true. We conclude that Parse() maintains Invariant C.4:PS.

Using Corollary C.4:Quickproj we can replace all Proj() calls by appropriate accesses to PS. That is all the transformations necessary for converting the old versions of Parse() and NewStates() into the new versions, except that we have to discuss the SET_ patterns in greater detail.

The idea of the SET_ cases of Parse() is to recursively call Parse() with p1 becoming the new PP and the result of NewStates() becoming the
new q. In addition, we want PSnew to become the new PS. Invariant C.4:PS requires that PSnew be a stack with one element, whose State component is newqqn where newqqn is a state of p1. Invariant C.4:PS allows the bottommost stack element to have an arbitrary OneTwo component. Thus, PSnew can be created as listed. The SET_ cases establish InvProjState(qN, p, PP, PS) because the call to NewStates() establishes InvProjState() for the same N as returned by the recursive call to Parse() on p1. (The next paragraph shows this.)

Function NewStates() is very similar to the old version. It accepts a new PS parameter, which is an input/output variable; the Proj() calls become accesses to PS; and RefreshPS() is called when N changes. All of these transformations were previously justified. A new side effect of NewStates() is establishing InvProjState(qN, p, PP, PS) upon exit. If n=N upon exit then InvProjState() is true because it is true upon entry and it is never changed. If N < n then InvProjState() is true because RefreshPS(PS, qN) is called after changing N.

\section*{C.5. Proof of the Complexity of Parse()}

\subsection*{C.5.1. Proof of Theorem 7.7:ParseTime}

Since each node of a parse tree corresponds to one call to Parse(), we should analyze the maximum time for one call to Parse(), ignoring recursive calls to Parse(). Then the worst case time of Parse() would be proportional to the maximum time for one call times the size of the parse tree. This explains the origin of the $O(\text{size}(PT))$ factor. Actually, in calculating the maximum time for one call to Parse() we also ignore
the calls to RefreshPS(); all such calls are factored out and reflected in the $O(\text{maxdepth}(\text{PP}) \approx \text{LL}(\text{PT}))$ factor. Now we proceed to calculate the maximum time for one call to Parse(), ignoring recursive calls to Parse() and calls to Refresh(). Then we explain how RefreshPS() calls can be factored out.

For the operators EMPTY, \{x\}, SEQ, OR, DEFCALL, MINI-AND, and CSLP, the analysis is trivial. These patterns use only constant overhead per Parse() call. The pattern * involves a loop, but we can charge each piece of constant time loop overhead to the recursive Parse() call in the loop. The MINI-NOT and ANYSTR loops require constant time per RefreshPS() call which is counted elsewhere.

To calculate the time required by the SET_ cases we only have to analyze the time required by NewStates(), since PushPS() and NewPS() can operate in constant time, and the only other statement is a recursive call to Parse(). After factoring out the RefreshPS() calls and proportional overhead (see the discussion to follow) NewStates() requires only constant time.

Thus, every case of Parse() require constant cost per Parse() call, when recursive Parse() calls and RefreshPS() calls are ignored. Summing over all calls to parse we find the total time for a Parse() call, ignoring RefreshPS() calls, is $O(\text{size}(\text{PT}))$.

Now we have to argue that all calls to RefreshPS() and constant overhead per RefreshPS() call are counted in $O(\text{maxdepth}(\text{PP}) \approx \text{LL}(\text{PT}))$. First we analyze the running time ignoring
the RefreshPS() calls within NewStates(), then we show that the latter
calls can safely be ignored. The analysis of RefreshPS() requires the
following definitions:

**Definition C.5:Depth:**

Assume patterns p and PP are viewed as trees pTree and PPTree;
and p is in IN PP, so that pTree is a tree within PPTree:

\[
\text{depth}(p, PP) = \text{the number of tree nodes from the root of pTree to the root of PPTree.}
\]

\[
\text{maxdepth}(PP) = \text{the maximum depth}(p, PP) \text{ over all patterns } p \text{ in PP.}
\]

end Definition.

We will use a number of obvious facts about depth() and maxdepth()
which we do not formally prove:

**Lemma C.5:DepthFacts:**

If psub is IN p is IN PP, then depth(psub, PP) = depth(psub, p) + depth(p, PP).

\[
\text{maxdepth}(PP) \geq \text{maxdepth}(p) + \text{depth}(p, PP).
\]

for any p in PP, \( \text{depth}(p, PP) \leq \text{maxdepth}(PP) \).

end Lemma.

The leaf nodes of PT represent one of the patterns \{x\}, EMPTY,
ANYSTR, or MINI-NOT. The EMPTY case does not call RefreshPS(),
but the cases for the other patterns do. In fact, ignoring for a moment
the RefreshPS() calls in NewStates(), all RefreshPS() calls occur in
creating the leaf nodes of PT. The ANYSTR and MINI-NOT cases call
RefreshPS() size(s1) times, where size(s1) is the length of the string
matching the ANYSTR or MINI-NOT. The \{x\} pattern calls
RefreshPS() once, and it matches a string of size 1. We conclude that the total number of calls to RefreshPS() is bounded by LL(PT).

The cost of one RefreshPS(PS, S) call is proportional to the length of PS. The latter is depth(p, PP) when RefreshPS() is called from within Parse(p, n, q, PP, PS), which can be discerned from Invariant C.4:PS and the definition of function RefreshPS(). Thus, all RefreshPS() calls inside Parse(p, n, q, PP, PS) can be bounded by time proportional to maxdepth(PP), and the total time for all RefreshPS() calls is $O(\maxdepth(PP) \propto LL(PT))$.

There is one detail to tie up: when NewStates() is called from within Parse((SET_ p1), n, q, PP, PS), RefreshPS() is called size(s1) times, where size(s1) is the length of the substring matching (SET_ p1). The previous analysis ignored these calls. Fortunately, we can show that the time required by these calls save at least an equal amount of time in future calls to RefreshPS(). If this is true, the previous analysis is still correct.

A call to Parse((SET_ p1), n, q, PP, PS) recursively calls Parse(p1, n, newqq, p1, PSnew); p1 is smaller than PP, and PSnew is smaller than PS. In fact, a call to RefreshPS() from within the recursive call requires $O(\maxdepth(p1))$ time, while our previous analysis counted each such call as $O(\maxdepth(PP))$ time. The previous analysis counted at least size(s1) calls to RefreshPS() within the recursive call, since s1 is the string matching (SET_ p1), and we counted $LL((SET_ p1))$ calls inside p1, and size(s1) ≤ $LL((SET_ p1))$. Therefore, we overestimated the time for RefreshPS() calls by at least
$O(\text{size}(s_1) \propto (\text{maxdepth}(PP) - \text{maxdepth}(p_1)))$. By Lemma C.5:DepthFacts the overestimate is more than $O(\text{size}(S_1)\propto \text{depth}(p_1, PP))$. However, the calls to RefreshPS() from NewStates() requires less than $O(\text{size}(s_1)\propto \text{depth}(p_1, PP))$ time. Thus, the “overestimate” actually compensates for the RefreshPS() calls within NewStates().

end Proof.

C.5.2. Proof of Theorem 7.7:ParseSpace:

If a parse tree, PT, is created by Parse(), then of course $O(\text{size}(PT))$ space is needed for PT. There is some further space needed during Parse() execution for the ProjStack data structure and the state sequences returned by NewStates(). The former requires $O(\text{maxdepth}(PP))$ space, which can be absorbed by the $O(\text{size}(PT))$ factor. The space to store the state sequence returned by NewStates() is needed only as long as the recursive call executes, and the length of the state sequence is bounded by size(s). Therefore, at most $O(\text{size}(s) \propto MNS(PP) \propto \text{size_of_1_state})$ space is needed for the state sequences. Using the state representation discussed as an optimization of Next(), size_of_1_state is a 32 bit integer, which counts as constant space. Thus the state sequences require $O(\text{size}(s) \propto MNS(PP))$ space during Parse() execution. The total space is the sum of the two factors.
end Proof.
Appendix D
Details of TLex

This section presents detailed descriptions of the parse tree access functions.

D.1. Accessing the Parse Tree

When an action is executed, a parse tree giving details about the match is stored in the variable SS, which is local to the action. SS is of type ss, which is an abstract parse tree type. This means it cannot be accessed directly, only through a set of functions soon to be described.

Any initialized parse tree, say ss2, is associated with a pattern, call it p. ss2 remembers the starting and ending positions of the substring matching p; in addition, if p has named subpatterns, ss2 may hold parse trees representing the matches of these subpatterns. In this manual, we will refer to “the substring matching ss2” when we really mean the substring, remembered by ss2, matching the pattern associated with ss2. An uninitialized parse tree is one that does not represent any pattern. Obviously, uninitialized parse trees normally result from errors.

The input text is considered an array of characters, whose first index is 0. TLex, however, refers to POSITIONS in the text, which conceptually fall BETWEEN characters. Position i is to the left of character i. A substring between start and end includes exactly those characters between positions start and end. Thus an empty substring
Appendix D: Details of TLex

has start=end; the substring which is the same as character i has start = i, end = i+1. The following diagram may clarify this:

Figure D-1: Text Positions

TLex is responsible for creating and destroying the parse tree; it is created right before the action is called and destroyed as soon as the action terminates.

Subsequent sections describe the various functions for accessing a parse tree. In the following, consider foo to be an arbitrary parse tree, of type ss.

D.2. Assignment and Argument Passing

Objects of type ss can be efficiently assigned and passed by value. Converting an ss into an integer, which is possible in C++, returns TRUE iff ss is initialized.

D.3. Display

```c
void foo.print_binding()
```

This function prints out the parse tree foo, showing sub-parse trees indented under the main parse tree. This function may be useful for debugging. Here is a typical printout for a match of the pattern “main” with theses definitions:
(define Space ' )
(define Digit (from "0" to "9"))
(define number (csrpn Digit (+ Digit) Digit))
(define decNumber (seq-1 number-1 . number-2))
(define main decNumber-1)

When “main-1” is matched against the string “5.6” and the resulting parse tree is printed out, a display like this results:

```
main-1: [0 - 3]: count 424
  "5.6"
decNumber-1: [0 - 3]: count 0
  "5.6"
  seq-1: [0 - 3]: count 0
    "5.6"
      number-1: [0 - 1]: count 0
        "5"
      number-2: [2 - 3]: count 0
        "6"
```

Each entry shows the name of the named operator that was parsed through, the starting and ending positions of the match through that operator, the “count” value (which is only relevant for OR and LOOPING patterns), and on the next line the actual text of the match.

**D.4. Access of Individual Characters**

```
foo.string(longnumber i)
```

Returns the character at index i in the substring that matches foo. The first character in the substring (assuming the substring has at least 1 character in it) is indexed by 0. If the array reference is outside the substring, a fatal error may occur in rare circumstances, so avoid this situation. If foo is uninitialized, it errors.
D.5. Conversion to a C-string

<table>
<thead>
<tr>
<th>char * foo.toString()</th>
</tr>
</thead>
<tbody>
<tr>
<td>char * foo.toString(void * mem, longnumber memsize)</td>
</tr>
</tbody>
</table>

The short version converts foo into a newly allocated C-string and returns it. It is the user’s responsibility to properly free() or “delete” the string returned. The long version converts foo into a C-string using the memory pointed to by mem. At most memsize-1 characters and the trailing zero byte are copied. If foo matches the empty string, “”, then the empty string is returned or written to mem. If foo is uninitialized, it errors.

D.6. Start, End, and Length

<table>
<thead>
<tr>
<th>longnumber foo.start()</th>
</tr>
</thead>
<tbody>
<tr>
<td>longnumber foo.end()</td>
</tr>
<tr>
<td>longnumber foo.length()</td>
</tr>
</tbody>
</table>

foo.start() returns the position of the start of the substring foo, which is before the first character in the substring. This is also the index of the first character in the substring. If foo is uninitialized, it errors.

foo.end() returns the position marking the end of the substring. If foo is uninitialized, it errors.
foo.length() returns the number of characters in the substring, as a longnumber. If foo is uninitialized, it errors.

D.7. Conversion to Longnumber

longnumber foo.tonum()

Returns the numeric equivalent of foo. Leading spaces are ignored; the number goes from the first nonspace character in foo to the first non-digit character or the end of foo. If the first nonspace character is a “-”, a negative number is returned. If a prefix of foo is not a number, 0 is returned. Note that foo.tonum() returns a longnumber. (This function is essentially equivalent to the Unix function “atol”, except that .tonum() assumes a decimal base.) If foo is uninitialized, it errors.

D.8. Specialized Loop Accesses

longnumber foo.count()

ss foo[longnumber i]

These operators require that foo represent a match for one of the looping patterns:

- (exactly-3 pat)
- (atmost-3 pat)
- (atleast-3 pat)
- (*-4 pat)
Appendix D: Details of TLex  194

o (**-4 pat)

o (+-5 pat)

o (++-5 pat)

If so, foo.count() returns the number of times pat was traversed, which is also the number of iterations of the loop. foo[i] returns an ss matching the ith iteration of the loop (0-based). If i is out of bounds then an uninitialized ss is returned. Note that a loop iteration is not a loop, so for example foo[i].count() is not meaningful if foo represents a looping pattern. Of course, foo[i] is a substring and thus can be operated on with .start(), .end(), etc..

If foo is uninitialized or if foo does not represent a looping pattern, an error is reported.

D.9. Specialized Loop Iteration Access

ss foo.adv(longnumber j)

This operator requires that foo represent a match for one of the iterations of a looping pattern, i.e. foo = bar[i] when bar represents a looping pattern.

If foo is an ss for the ith iteration of some loop, then foo.adv(j) returns the (ith + j) iteration of the loop, or an uninitialized ss if there is no such iteration. This syntax compensates for the fact that accessing the ith iteration of a loop, such as bar[i], takes i operations. Using foo.adv(n), one can advance n iterations with only n operations. Since
n is usually 1 this is more efficient. The following example shows how to efficiently loop through the loop iterations of bar, when bar is an ss matching a loop:

```c
ss fooiter;
for (fooiter = bar[0];
    fooiter;
    fooiter = fooiter.adv(1))
    use(fooiter);
```

(The Boolean test converts fooiter to an integer, returning TRUE iff fooiter is initialized (as was described in section D.2.)) Note that fooiter.adv(i) does not change fooiter, it returns a new ss.

If foo is uninitialized or foo does not represent a loop iteration, or if “j” is negative, an error is reported.

D.10. Specialized Optional Test

```c
int foo.thru()
```

When foo represents one of the patterns

- o (?-1 pat1)
- o (?-2 pat2)

then foo.thru() returns TRUE iff the subpattern was matched. If foo is uninitialized an error is reported.

D.11. Specialized OR Access

```c
longnumber foo.which()
```
Appendix D: Details of TLex

When foo represents

\[ o \ (or-1 \ pat1 \ pat2 \ pat3 \ pat4 \ ...) \]

then foo.which() returns the index of the subpattern of or-1 that was parsed. This is a 1 based index, so if pat1 was parsed or-1.which() returns 1. If or-1 is uninitialized an error is reported.

D.12. Nesting Accesses

Because patterns are recursive and nested with parentheses, there is a natural notion of 1 pattern containing another: p1 contains p2 when p2 is a subpattern of p1, or a subpattern of p1 contains p2. In the example below bar-2 is contained by seq-1, or-1, and foo.

\[
\begin{align*}
  (define \ bar \ (seq-3 \ c \ d)) \\
  (define \ foo \ (or-1 \ (seq-1 \ a \ bar-2) \\ \ \ (* \ (notclass \ 13))) ) \\
  (define \ main \ (*-1 \ (seq \ a \ b \\ \ (+-1 \ (or \ a \ (seq-1 \ b \ foo-2)))))
\end{align*}
\]

Given X an ss, we can access any ss contained in X by writing X[“Y”] where Y is the name of the desired pattern inside X.

C++: ss operator[](char* access_string)

For example, if ff is an ss matching foo, then the following are all the legal accesses of an ss contained in ff: ff[“or-1”], ff[“seq-1”], ff[“bar-2”]. Whitespace is not allowed in the access string.

Nested accesses can be composed: ff[“or-1”][“bar-2”] returns the bar-2 inside the or-1 inside ff. However, this expression returns the same ss as ff[“bar-2”]. TLex provides a simplified syntax that allows consecutive nested accesses to be combined: just combine the strings
and separate the names with dots, like a field access in C++:
ff[“or-1”][“bar-2”] can be written ff[“or-1.bar-2”].

Sometimes consecutive accesses cannot be shortened. There is an ambiguity when we want to access a pattern that is located inside a looping pattern. For example, assume mm is an ss associated with pattern main, defined above, and we want to get the value of seq-1 inside main. Actually there are many ss associated with seq-1. For each time through the *-1 loop there may be, for each time through the +1 loop, a seq-1 ss. Thus to access a specific ss for seq-1 we must first specify which loop iteration of +1 and *-1 to consider. The syntax for this is the obvious one: assuming we want the first value associated with seq-1 the second time through the *-1 loop, we ask for it with the following statement:

C++: mm[*1[1].+1[0].seq-1]

Notice that iteration indexing begins at 1. What if one wanted to put a variable or other expression returning a number as the *-1 subscript? One cannot use “*-1[i].+1[0].seq-1” because TLex does not know about “i”. Instead, one must return to consecutive accesses:

C++: mm[*1][i][+1[0].seq-1]
Another case is accessing items bound inside definitions. In order to access an ss inside a definition, prefix the ss with the name of the definition. (This syntax allows named operators to have the same name in different definitions, without causing confusion.) For example, accessing the seq-3 ss inside bar in iteration 1 of +1 in iteration j of *-1 is done as follows:

C++: foo[“*-1”][j][“+1[1].foo-2.bar-2.seq-3”]

D.13. Accessing Macros

Macro calls cannot be named. Treat macro calls as if the substitution has been textually done, and access the operators in the macro directly.

Example:

If “double” is a macro defined as (seq-1 $1 $1x), then in the ss X representing the pattern (or-1 (double foo-2) “hi”), valid accesses include X[“or-1”], X[“seq-1”], X[“foo-2”].

D.14. Access Errors

What about error conditions? If rr is an ss for or-1 in our example pattern above, it is possible that the second branch of or-1 was parsed, not the first. What happens if rr[“bar-2”] is executed in this case? An error will occur, and a message printed stating that TLex could not find an ss for “bar-2” inside rr. The programmer should check that rr.which() is 1 before requesting rr[“bar-2”]. An error is also reported if one asks for an ss which will never be there, for example: rr[“or-1”], rr[“barr-2”]. Of course, accessing an ss inside an uninitialized ss results in an error.
Appendix D: Details of TLex  199

Design note: We could have returned an uninitialized ss for illegal accesses. This would provide a convenient way for the programmer to test the path of a match. However, it has the undesirable effect of delaying the reporting of errors until the programmer attempts to use the uninitialized ss returned. Upon experimentation, erroring immediately appeared much more desirable.
Appendix E
WC2 in TLex

This appendix contains listing of the WC2 programs written in TLex.

The RLex version is almost exactly the same as the TLex version.

/* wc2.tlx
*/

/* arbitrary user code goes here. */
#include "malloc.h"
#include "string.h"

main (int argc,char ** argv)
/**********
expects the data file to be "wc2.tld" and the
input file to be passed as the first argument.
**********/
{
 int i;

 if (argc == 1)
   {
     printf("error*** no filename specified\n");
     return 1;
   }

 PRINTDOTS = FALSE;
 PRINTTIMING = FALSE;

 initTLex("wc2.tld", argv[1], wc2TLexActions);

 wc2_init();
doTLex(TLEX_SAME_BUFFER, argv[1], 0, 0, 1);

deinitTLex();

} /* globals for the longest so far */

int wordLongestLen, lineLongestLen, sentLongestLen;
int wordCount, lineCount, sentCount;
char * wordLongest, * lineLongest, * sentLongest;

wc2_init()
{
   wordCount = lineCount = sentCount = 0;
   wordLongestLen = lineLongestLen = sentLongestLen = 0;
   wordLongest = malloc(3);
   lineLongest = malloc(3);

sentLongest = malloc(3);
wordLongest[0] = lineLongest[0] = sentLongest[0] = 0;
}

/* definitions and macros */

(define end (csrpn empty any))

(define StartLine (csrp (class Start CR) empty empty))

(define CR 10)

(define EOS (class "." ":" "?" "!"))

(define space (class 32 10 9 12 13))
(define notspace (notclass space))

(define Word (csrpn notspace (+ notspace) notspace))
Appendix E: WC2 in TLex 202

(define Line [StartLine (* (notclass CR))])
(define Sentence (csrp (class EOS Start)
   [{* (notclass EOS)) EOS]
   empty))

-----------------------------------------------------------------

/* ruleset 1 */

Word-1 ==> 
{
   ss myword = SS;
   wordCount++;
   if (myword.length() > wordLongestLen)
      {
         wordLongestLen = myword.length();
         free(wordLongest);
         wordLongest = myword.tostring();
      }
}

/* line */

Line-1 ==> 
{
   ss myline = SS;
   lineCount++;
   if (myline.length() > lineLongestLen)
      {
         lineLongestLen = myline.length();
         free(lineLongest);
         lineLongest = myline.tostring();
      }
}

/* sentence */

Sentence-1 ==> 
{
   ss mysent = SS;
   sentCount++;
   if (mysent.length() > sentLongestLen)
      {
         sentLongestLen = mysent.length();
         free(sentLongest);
         sentLongest = mysent.tostring();
      }
Appendix E: WC2 in TLex

```c
end

printf("words: %d   lines: %d   sentences: %d
",
    wordCount, lineCount, sentCount);
printf("longest word: %s
", wordLongest);
printf("longest line: \n%s
", lineLongest+1);
printf("longest Sentence: \n%s\n", sentLongest+1);
```

/**************************** EOF ****************************/

Appendix F
The BBall Application

This appendix contains a listing of the TLex input file for the BBall application. We have translated all parse tree accesses from the original C version to the C++ version, because only the syntax for the latter was given in the thesis.

```c
/*
   bball.tlx - tlex version
*/

#include <math.h>
#include "malloc.h"
#include "string.h"
#include "tlexlib.h"

int miscellaneous(int count)
{
    return 1;
}

main (int argc,char ** argv)
{**********
    expects the data file to be "bball.tld" and the input file to be passed as the first argument. 
    **********/
    { int i;
      if (argc == 1)
      {
        printf("error*** no filename specified\n");
        return 1;
      }
      PRINTDOTS = FALSE;
      PRINTTIMING = TRUE;
      transitionCacheTLex(5);  /* boost transition cache size */
      initTLex("bball.tld", argv[1], bballTLexActions);
      for (i = 1; i <= 8; i++)
        doTLex(TLEXSAME_BUFFER, argv[1], 0, 0, i);
      deinitTLex();
}
Appendix F: The BBall Application

*/ definitions and macros */

/**************
digit:
 matches a digit from 0..9.
**************/

(define digit (from "0" to "9"))

/**************
letter:
 matches a lowercase or uppercase letter of the alphabet.
**************/

(define letter (or (from a to z) 
(from A to Z)))

/*************
word:
 matches a string of letters
*************/

(define word (csrpn letter (+ letter) letter))

/*************
um:
 matches a string of digits
*************/

(define num (csrpn digit (+ digit) digit))

/************
decnum:
 a number with an optional . and more digits.
************/

(define decnum (or (csrpn num-1 .) 
(seq num-2 
 . 
 num-3)))

(define statnum (or (csrpn num-1 dot) 
(seq num-2 dot num-3) (seq dot num-4)))

/* spwire.tlx --
 TLEX file to recognize patterns from the sports wire */

(define tab 9) 
(define cr (or 10 (csrpn 13 10) (seq 13 10)))
(define notcr (notclass 10 13))
(define space 32)
(define dollar 36)
(define percent 37)
(define amper 38)
(define leftpar 40)
(define rightpar 41)
(define aster 42)
(define plus 43)
(define comma 44)
(define dash 45)
(define dot 46)
(define slash 47)
(define colon 58)
(define larrow 174)
(define rarrow 175)
(define soh 1)
(define stx 2)
(define etx 3)
(define eot 4)
(define ctln 14)
(define dcl 17)
(define sync 22)
(define tfi 31)

(define lowercase (from a to z))
(define uppercase (from A to Z))
(define alphanum (or letter digit))
(define wchar (or space tab))
(define wschars (+ wchar))
(define wspace (* wchar))
(define wsoorcr (or wchar cr))
(define wsoorcrs (+ wsoorcr))
(define notws (notclass 10 13 32 9))
(define emptylin (seq wspace cr))

(define time (seq (+ digit) colon (+ digit)) )

/* general purpose macros */

/ *(firstch p): matches a line
* which has a first part matching p
* (this is used to recognize game summary lines)
 */
(macro firstch (seq $1 filler cr))

(macro teamnames (csrpn letter (or $1 $2) letter))

/* definitions specific to BASKETBALL */

/* team definitions */

(define Atlan (teamnames "ATLANTA" "ATL"))
(define Bosto (teamnames "BOSTON" "BOS"))
(define Charl (teamnames "CHARLOTTE" "CHA"))
(define Chica (teamnames "CHICAGO" "CHI"))
Appendix F: The BBall Application

(define Cleve (teamnames "CLEVELAND" "CLE"))
(define Dalla (teamnames "DALLAS" "DAL"))
(define Denve (teamnames "DENVER" "DEN"))
(define Detro (teamnames "DETROIT" "DET"))
(define Golde (teamnames "GOLDEN STATE" "GSW"))
(define Houst (teamnames "HOUSTON" "HOU"))
(define India (teamnames "INDIANA" "IND"))
(define Lacli (teamnames (or "L.A. CLIPPERS" "LA CLIPPERS") "LAC"))
(define Lalak (teamnames (or "L.A. LAKERS" "LA LAKERS") "LAL"))
(define Miami (teamnames "MIAMI" "MIA"))
(define Minne (teamnames "MINNESOTA" "MIN"))
(define Njnet (teamnames "NEW JERSEY" "NJN"))
(define Nykni (teamnames "NEW YORK" "NYK"))
(define Orlan (teamnames "ORLANDO" "ORL"))
(define Phila (teamnames "PHILADELPHIA" "PHI"))
(define Phoen (teamnames "PHOENIX" "PHO"))
(define Portl (teamnames "PORTLAND" "POR"))
(define Sacra (teamnames "SACRAMENTO" "SAC"))
(define Sanan (teamnames "SAN ANTONIO" "SAS"))
(define Seatt (teamnames "SEATTLE" "SEA"))
(define Utahj (teamnames "UTAH" "UTH"))
(define Wasli (teamnames "WASHINGTON" "WAS"))

(define Team (or-1
  Atlan
  Bosto
  Charl
  Chica
  Cleve
  Dalla
  Denve
  Detro
  Golde
  Houst
  India
  Lacli
  Lalak
  Miami
  Milwa
  Minne
  Njnet
  Nykni
  Orlan
  Phila
  Phoen
  Portl
  Sacra
  Sanan
  Seatt
  Utahj
  Wasli
))

/* definitions specifically for RULE SET 3 (Pro Basketball Finals) */
(define Scorenum
  (seq wschars num-1))

(define Overtime
  (*-1 Scorenum-1))

(define Linescore
  (seq wspace Team (exactly-1 4 Scorenum-1) Overtime-1 Scorenum-2
    (* (notclass 13 10 (from "0" to "9"))) cr))

(define HighType
  (seq "HIGH" wschars (or-1 "SCORERS" "REBOUND" "ASSISTS")
    colon))

(define HighHeader
  (seq (and Team (exactly 3 letter)) wspace dash))

(define Extraline
  (seq wspace (all (seq notws any any) minus Team "HIG")
    (* notcr) cr))

(define HighEntry
  (seq HighHeader (* notcr) cr (* Extraline)))

/* definitions specifically for RULE SET 4
(Pro Basketball Summaries) */

(define Teamstr
  (seq letter (* (or wschar letter amper comma dot)) letter))

/* definitions specifically for RULE SET 5
(Pro Basketball Lineups) */

(define Player
  (seq (? (seq word wschars)) letter
    (* (or letter dot dash)) letter))

/* position abbrevs: guard, center, forward */
(define Posletter (class G C F))

(define Lineuplin
  (seq wspace Posletter-1 wschars Player-1 wschars
    Posletter-2 wschars Player-2 wspace cr))

/* definitions specifically for RULE SET 7
(Pro Basketball Schedules) */

(define SchedLine
  (seq Team-1 wschars "AT" wschars Team-2
    wschars time-1 (* notcr) cr))

/* definitions for RULE SET 6 & 8
(Pro Basketball Standings & Leaders) */

(define numspace (seq wspace num))
Appendix F: The BBall Application

(define Win numspace)
(define Loss numspace)
(define Pct (seq wspace statnum))
(define Record (seq wspace num-l "-" num-2))
(define WL_Vect (or-1 "WON" "LOST"))
(define Frac (seq wspace num-2 "/" num-3))
(define GB (or-1 dash
    num-1
    (seq num-2 Frac)
    Frac))

(define filler (* notcr))
(define wordplus (csrpn (+ (or comma dot ":" amper letter))
letter))

/* Non Standard Teams */

(define line (seq-1 Win-1 Loss-1 Pct-1 wspace GB-1
    wspace Record-1 wspace Record-2
    wspace Record-3 wspace WL_Vect wspace num-1))

(define Eastern "EASTERN")
(define Western "WESTERN")
(define Atlantic "ATLANTIC")
(define Central "CENTRAL")
(define Midwest "MIDWEST")
(define Pacific "PACIFIC")

(define Div "DIV.")
(define Lit_Conf "CONFERENCE")
(define Pro_Bask_Div (or-1 Atlantic Central Midwest Pacific))
(define Pro_Bask_Conf (or-1 Eastern Western))

(define pfst_record (seq-1 (?-1 "X-"") (?-2 "Y-"") wspace
    Team-1 wspace line-1
    filler cr))

(define Categories (or-1 "SCORING" "ASSISTS" "REBOUNDING"))

(define att_comp (seq-1 num-2 "/" num-3))
(define Stat (or-1 num-l decnum-l att_comp-l))
(define Stats (+-1 (seq wspace Stat-1))

(define PName (+-1 (seq wspace wordplus)))
(define PlayerTeam (seq PName-1 "," wspace Team-1
    (?-1 (seq dash Team-2))))

-------------------------------------------------------------------

/* RULE SET 1 (Pro Basketball Score Updates) */

(seq "NBA" filler-1)

=/>
Appendix F: The BBall Application

char *text;
miscellaneous(500);
text = SS["filler-1"].to_string();
miscellaneous(100);
delete text;
}

/* RULE SET 2 (Pro Basketball Game Notes) */
(seq "NOTE" (? colon) wspace filler-1)
===>
{
  char *text;
miscellaneous(500);

text = SS["filler-1"].to_string();
miscellaneous(100);
delete text;
}

/* RULE SET 3 (Pro Basketball Finals) */
(seq cr Linescore-1 Linescore-2)
===>
{
  longnumber hteam_pts, vteam_pts, h_total, v_total;
  char refbuf[512];
  int num_ot, i;
miscellaneous(500);
miscellaneous(1000);

  if (SS["Linescore-1.Overtime-1.*-1"].count !=
      SS["Linescore-2.Overtime-1.*-1"].count())
    printf("error 101\n");

  for (i=0; i < 4; i++) {
    vteam_pts = SS["Linescore-1.exactly-1"]\[i\]
      ["Scorenum-1.num-1"].tonum();
    hteam_pts = SS["Linescore-2.exactly-1"]\[i\]
      ["Scorenum-1.num-1"].tonum();
    miscellaneous(1000);
  }

  num_ot = SS["Linescore-1.Overtime-1.*-1"].count();
  if (num_ot > 0) {
    miscellaneous(4000);
    for (i=0; i < num_ot; i++) {

Appendix F: The BBall Application

longnumber h_ot, v_ot;
miscellaneous(200);

v_ot = SS["Linescore-1.Overtime-1.*-1"][i]["Scorenum-1.num-1"].tonum();
h_ot = SS["Linescore-2.Overtime-1.*-1"][i]["Scorenum-1.num-1"].tonum();

miscellaneous(2000);

v_total = SS["Linescore-1.Scorenum-2.num-1"].tonum();
h_total = SS["Linescore-2.Scorenum-2.num-1"].tonum();

miscellaneous(2000);

(seq HighType-1 (+-1 (seq wspace HighEntry-1)))

{ char * text;
  int hightype, nhigh, i;
  char refbuf[512];

  hightype = SS["HighType-1.or-1"].which();

  miscellaneous(200);
  nhigh = SS["+-1"].count();
  for (i=0; i < nhigh; i++) {
    miscellaneous(200);
    text = SS["+-1"][i]["HighEntry-1"].tostring();
    miscellaneous(1000);
    free(text);
  }
  miscellaneous(200);
}

/* RULE SET 4 (Pro Basketball Summaries) */

(seq wspace Teamstr wschars num comma wspace Teamstr wschars num (* notcr) cr "-----" filler cr (* emptylin)
  (+-1 (firstch (notclass 10 13 32 9 12))))

{ char *text;
  miscellaneous(2000);
  text = SS["+-1"].tostring();
/* RULE SET 5 (Pro Basketball Lineups) */

(seq cr wspace Team wschars Team wspace cr (* emptylin) (exactly-1 5 Lineuplin-1))
===>
{
    char * str;
    int i;
    char refbuf[512];

    for (i=0; i < 5; i++) {
        str = SS["exactly-1"[i]["Lineuplin-1.Posletter-1"].tostring();
        free(str);
        str = SS["exactly-1"[i]["Lineuplin-1.Player-1"].tostring();
        free(str);
        str = SS["exactly-1"[i]["Lineuplin-1.Posletter-2"].tostring();
        free(str);
        str = SS["exactly-1"[i]["Lineuplin-1.Player-2"].tostring();
        free(str);
    }
}

/* RULE SET 6 (Pro Basketball Standings) */

(seq Pro_Bask_Conf-1 wspace Lit_Conf)
===>
{
    int conf_ord;

    conf_ord = SS["Pro_Bask_Conf-1.or-1"].which();
}

(seq Pro_Bask_Div-1 wspace Div)
===>
{
int div_ord;

div_ord = SS["Pro_Bask_Div-1.or-1"].which();
}

(+1  pfst_record-1)

proc
{
    char *conf_ptr, *div_ptr;
    static char refbuf[512];
    float pct, GB;
    int div_ord;
    int nteams, i, team_ord;
    char *teams, *strpct;
    int win, loss;
    int clinched;
    ss tempss;

    /* Calculate Proper Division */
    miscellaneous(1000);
    miscellaneous(200);

    /* Now loop through and write teams */
    nteams = SS["+-1"].count();
    miscellaneous(200);
    for (i=0; i < nteams; i++) {
        tempss = SS["+-1"][i]["pfst_record-1"];
        miscellaneous(100);
        teams = tempss["Team-1"].toString();

        /* Convert to ordinal */
        clinched = tempss["?-1"].thru();
        win = tempss["line-1.Win-1"].tonum();
        loss = tempss["line-1.Loss-1"].tonum();
        strpct = tempss["line-1.Pct-1"].toString();
        free(strpct);
        switch (tempss["line-1.GB-1.or-1"].which()) {
            case 1:   GB = 0 ; break;
            case 2:    GB = tempss["line-1.GB-1.num-1"].tonum();
                        break;
            case 3:    GB = tempss["line-1.GB-1.num-2"].tonum() + 0.5;
                        break;
            case 4:   GB = .5; break;
            default:   break;
        }
        team_ord = tempss["Team-1.or-1"].which() - 1;
miscellaneous(1000);
free(teams);
miscellaneous(6000);
}
miscellaneous(600);
}

/*----------------------------------------------------------
/* RULE SET 7 (Pro Basketball Schedules) */
*/

(seq "SCHEDULE" wsorcrs "FOR" wschars word comma wspace word-1
wschars
num-1)

===>
{
  longnumber valid_date;
  char *month;

  month = SS["word-1"].tostring();
  valid_date = SS["num-1"].tonum();
  free(month);
  miscellaneous(20);
}

(+1 SchedLine-1)

===>
{
  longnumber vord, hord;
  char * gametime;
  int nscheds, i;
  ss tempss;

  nscheds = SS["+1"].count();
  for (i=0; i < nscheds; i++) {
    tempss = SS["+1"][i]["SchedLine-1"];
    vord = (longnumber) tempss["Team-1.or-1"].which() -1;
    hord = (longnumber) tempss["Team-2.or-1"].which() -1;
    gametime = tempss["time-1"].tostring();
    miscellaneous(3000);
    free(gametime);
  }
}

/*----------------------------------------------------------
/* RULE SET 8 (Pro Basketball Leaders) */
*/

Categories-1

===>
{
  int cat_ord;

  cat_ord = SS["Categories-1.or-1"].which();
}
Appendix F: The BBall Application

(+ - (seq PlayerTeam-1 Stats-1 wspace cr))
==>
{
  char *teamstr, *p1,*p2;
  int nplayers,nstats,cat_ord;
  char * player, * val;
  longnumber team;
  int leader_type, stat_type, numval;
  int nleaders, nfields, i, j,k;
  static char refbuf[512], namebuf[25];
  ss tempss, temp2ss;

  miscellaneous(1000);

  nplayers = SS["+-1"].count();
  for (i=0; i < nplayers; i++) {
    tempss = SS["+-1"][i];
    k = tempss["PlayerTeam-1.PName-1.+-1"].count();
    p1 = tempss["PlayerTeam-1.PName-1.+-1[0]"].tostring();
    free(p1);
    for (j=1; j < k; j++) {
      p2 = tempss["PlayerTeam-1.PName-1.+-1"][j].tostring();
      miscellaneous(2);
      free(p2);
    }

    player = tempss["PlayerTeam-1.PName-1"].tostring();
    if  (tempss["PlayerTeam-1.?-1"].thru() > 0)
      team = (longnumber)
        tempss["PlayerTeam-1.?-1.Team-2.or-1"].which() -1;
    else
      team = (longnumber)
        tempss["PlayerTeam-1.Team-1.or-1"].which() -1;

    teamstr = tempss["PlayerTeam-1.Team-1"].tostring();

    miscellaneous(2000);
    free(player);
    free(teamstr);
  
  nstats = tempss["Stats-1.+-1"].count();
  /* how many stats per line */
  k = 0;
  for (j=0; j < nstats; j++) {
    temp2ss = tempss["Stats-1.+-1"][j];
    stat_type = temp2ss["Stats-1.or-1"].which();
    if (stat_type < 3) {
      /* These are the simple numbers */
      val = temp2ss["Stats-1"].tostring();
      miscellaneous(1000);
      free(val); }
    else {
      /* These are the kicking type stats 11/11 */
      /* More than one value per stat */
    }
}
numval = (longnumber)
    temp2ss[“Stats-1.att_comp-1.num-2“].tonum();
miscellaneous(2000);
numval = (longnumber)
    temp2ss[“Stats-1.att_comp-1.num-3“].tonum();
miscellaneous(2000);
}
}
miscellaneous(1000);
}

/*************** EOF ****************/************
Appendix G
Sample BBall Input

-NBA--
bc-nba-u-sacramento CHASAC06207006043001 03-09 0000

NBA   CHARLOTTE       62
      SACRAMENTO       70  4:30 LEFT, 3RD QTR

st 03-09-90 00:01 et

-NBA--
bc-nba-u-goldenstate CLEGSW05608906075101 03-09 0000

NBA   CLEVELAND       56
      GOLDEN STATE      89  7:51 LEFT, 3RD QTR

st 03-09-90 02:22 et

9521-TEXT-
BC-BKP-STND-NBASTAND-R 03-09 0000

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
NATIONAL BASKETBALL ASSOCIATION STANDINGS
THRU MARCH 8TH
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

EASTERN CONFERENCE

<table>
<thead>
<tr>
<th>ATLANTIC DIV.</th>
<th>W</th>
<th>L</th>
<th>PCT</th>
<th>GB</th>
<th>HOME</th>
<th>ROAD</th>
<th>DIV</th>
<th>STREAK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>PHILADELPHIA</td>
<td>38</td>
<td>22</td>
<td>.633</td>
<td>-</td>
<td>24-4</td>
<td>14-18</td>
<td>14-4</td>
<td>WON 2</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>37</td>
<td>22</td>
<td>.627</td>
<td>1/2</td>
<td>23-5</td>
<td>14-17</td>
<td>11-6</td>
<td>LOST 2</td>
</tr>
<tr>
<td>BOSTON</td>
<td>34</td>
<td>24</td>
<td>.586</td>
<td>3</td>
<td>19-10</td>
<td>15-14</td>
<td>10-5</td>
<td>LOST 2</td>
</tr>
<tr>
<td>WASHINGTON</td>
<td>23</td>
<td>38</td>
<td>.377</td>
<td>15 1/2</td>
<td>15-15</td>
<td>8-23</td>
<td>7-12</td>
<td>LOST 2</td>
</tr>
<tr>
<td>NEW JERSEY</td>
<td>15</td>
<td>44</td>
<td>.254</td>
<td>22 1/2</td>
<td>11-18</td>
<td>4-26</td>
<td>7-9</td>
<td>LOST 1</td>
</tr>
<tr>
<td>MIAMI</td>
<td>14</td>
<td>47</td>
<td>.230</td>
<td>24 1/2</td>
<td>9-23</td>
<td>5-24</td>
<td>3-16</td>
<td>WON 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CENTRAL DIV.</th>
<th>W</th>
<th>L</th>
<th>PCT</th>
<th>GB</th>
<th>HOME</th>
<th>ROAD</th>
<th>DIV</th>
<th>STREAK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>DETROIT</td>
<td>45</td>
<td>15</td>
<td>.750</td>
<td>-</td>
<td>28-3</td>
<td>17-12</td>
<td>16-5</td>
<td>WON 6</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>38</td>
<td>21</td>
<td>.644</td>
<td>6 1/2</td>
<td>23-3</td>
<td>15-18</td>
<td>12-7</td>
<td>LOST 1</td>
</tr>
</tbody>
</table>
Appendix G: Sample BBall Input

<table>
<thead>
<tr>
<th>Team</th>
<th>W</th>
<th>L</th>
<th>PCT</th>
<th>GB</th>
<th>HOME</th>
<th>ROAD</th>
<th>DIV</th>
<th>STREAK</th>
</tr>
</thead>
<tbody>
<tr>
<td>MILWAUKEE</td>
<td>32</td>
<td>27</td>
<td>.542</td>
<td>12</td>
<td>1/2</td>
<td>20-10</td>
<td>12-17</td>
<td>9-10</td>
</tr>
<tr>
<td>INDIANA</td>
<td>31</td>
<td>29</td>
<td>.517</td>
<td>14</td>
<td></td>
<td>21-9</td>
<td>10-20</td>
<td>13-7</td>
</tr>
<tr>
<td>ATLANTA</td>
<td>28</td>
<td>31</td>
<td>.475</td>
<td>16</td>
<td>1/2</td>
<td>19-10</td>
<td>9-21</td>
<td>12-11</td>
</tr>
<tr>
<td>CLEVELAND</td>
<td>25</td>
<td>34</td>
<td>.424</td>
<td>19</td>
<td>1/2</td>
<td>17-12</td>
<td>8-22</td>
<td>6-13</td>
</tr>
<tr>
<td>ORLANDO</td>
<td>16</td>
<td>44</td>
<td>.267</td>
<td></td>
<td></td>
<td>12-18</td>
<td>4-26</td>
<td>4-19</td>
</tr>
</tbody>
</table>

WESTERN CONFERENCE

MIDWEST DIV.  W    L    PCT  GB       HOME    ROAD   DIV   STREAK
---------   --- --- --- -- ------- --- ---- ---- ---- ----

<table>
<thead>
<tr>
<th>Team</th>
<th>W</th>
<th>L</th>
<th>PCT</th>
<th>GB</th>
<th>HOME</th>
<th>ROAD</th>
<th>DIV</th>
<th>STREAK</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTAH</td>
<td>43</td>
<td>17</td>
<td>.717</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAN ANTONIO</td>
<td>38</td>
<td>20</td>
<td>.655</td>
<td>4</td>
<td>24-5</td>
<td>14-5</td>
<td>14-5</td>
<td>LOST 2</td>
</tr>
<tr>
<td>DENVER</td>
<td>33</td>
<td>26</td>
<td>.559</td>
<td>9</td>
<td>23-8</td>
<td>10-8</td>
<td>10-8</td>
<td>WON 4</td>
</tr>
<tr>
<td>DALLAS</td>
<td>32</td>
<td>27</td>
<td>.542</td>
<td>10</td>
<td>21-9</td>
<td>11-8</td>
<td>10-9</td>
<td>LOST 2</td>
</tr>
<tr>
<td>HOUSTON</td>
<td>29</td>
<td>31</td>
<td>.483</td>
<td>14</td>
<td>23-7</td>
<td>6-24</td>
<td>6-12</td>
<td>WON 5</td>
</tr>
<tr>
<td>MINNESOTA</td>
<td>16</td>
<td>43</td>
<td>.271</td>
<td>26</td>
<td>14-18</td>
<td>2-25</td>
<td>4-14</td>
<td>WON 1</td>
</tr>
<tr>
<td>CHARLOTTE</td>
<td>10</td>
<td>48</td>
<td>.172</td>
<td>32</td>
<td>9-21</td>
<td>1-27</td>
<td>5-15</td>
<td></td>
</tr>
</tbody>
</table>

PACIFIC DIV.  W    L    PCT  GB       HOME    ROAD | DIV   STREAK
---------   --- --- --- -- ------- --- ---- ---- ---- ---- ----

<table>
<thead>
<tr>
<th>Team</th>
<th>W</th>
<th>L</th>
<th>PCT</th>
<th>GB</th>
<th>HOME</th>
<th>ROAD</th>
<th>DIV</th>
<th>STREAK</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA LAKERS</td>
<td>44</td>
<td>14</td>
<td>.759</td>
<td>26</td>
<td>18-10</td>
<td>12-4</td>
<td></td>
<td>WON 1</td>
</tr>
<tr>
<td>PORTLAND</td>
<td>41</td>
<td>18</td>
<td>.695</td>
<td>3</td>
<td>26-4</td>
<td>15-14</td>
<td>13-6</td>
<td>WON 3</td>
</tr>
<tr>
<td>PHOENIX</td>
<td>39</td>
<td>19</td>
<td>.672</td>
<td>5</td>
<td>25-5</td>
<td>14-14</td>
<td>15-5</td>
<td>WON 3</td>
</tr>
<tr>
<td>SEATTLE</td>
<td>30</td>
<td>29</td>
<td>.508</td>
<td>14</td>
<td>21-8</td>
<td>9-21</td>
<td>5-11</td>
<td>LOST 1</td>
</tr>
<tr>
<td>GOLDEN STATE</td>
<td>27</td>
<td>32</td>
<td>.458</td>
<td>17</td>
<td>22-8</td>
<td>5-24</td>
<td>7-10</td>
<td>WON 2</td>
</tr>
<tr>
<td>LA CLIPPERS</td>
<td>24</td>
<td>36</td>
<td>.400</td>
<td>21</td>
<td>15-12</td>
<td>9-24</td>
<td>5-12</td>
<td>LOST 1</td>
</tr>
<tr>
<td>SACRAMENTO</td>
<td>18</td>
<td>42</td>
<td>.300</td>
<td>27</td>
<td>13-17</td>
<td>5-25</td>
<td>6-15</td>
<td>WON 1</td>
</tr>
</tbody>
</table>

FRIDAY, MARCH 9TH SCHEDULE

DETROIT AT NEW JERSEY          7:30
WASHINGTON AT BOSTON           7:30 - A
PORTLAND AT PHILADELPHIA       7:30
PHOENIX AT INDIANA             7:30
SEATTLE AT ATLANTA             7:30
UTAH AT MILWAUKEE              9:00
GOLDEN STATE AT L.A. LAKERS    10:30

A - AT HARTFORD CIVIC CENTER

(ALL TIMES ARE EASTERN STANDARD)

st 03-09-90 00:54 et
s9494-TEXT-
BC-BKP-RCPS-KINGS-R  03-09 0000
SPORTSTICKER NBA RECAP

SACRAMENTO 111, CHARLOTTE 102

Wayman Tisdale scored 33 points and set a club record in Sacramento's 111-102 victory over Charlotte. Tisdale outscored the Hornets, 20-19, in the third quarter, tying the KC-Sacramento record for points in quarter set by Eddie Johnson in 1986. Tisdale's nine field goals in the quarter was a team record. Antoine Carr had 24 points and Vinny Del Negro added 22 points and 13 assists as the Kings beat the Hornets for the third time in three meetings this season. Charlotte's Rex Chapman tied his career-high with 38 points.

NATIONAL BASKETBALL ASSOCIATION LEADERS THROUGH MARCH 8TH

SCORING

<table>
<thead>
<tr>
<th>PLAYER</th>
<th>GP</th>
<th>PTS</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael Jordan, Chi</td>
<td>59</td>
<td>1961</td>
<td>33.2</td>
</tr>
<tr>
<td>Karl Malone, UTH</td>
<td>60</td>
<td>1830</td>
<td>30.5</td>
</tr>
<tr>
<td>Patrick Ewing, NYK</td>
<td>1417</td>
<td>24.4</td>
<td></td>
</tr>
<tr>
<td>Akeem Olajuwon, Hou</td>
<td>60</td>
<td>1445</td>
<td>24.1</td>
</tr>
<tr>
<td>Jeff Malone, Was</td>
<td>57</td>
<td>1370</td>
<td>24.0</td>
</tr>
<tr>
<td>David Robinson, SAS</td>
<td>58</td>
<td>1385</td>
<td>23.9</td>
</tr>
<tr>
<td>Larry Bird, Bos</td>
<td>54</td>
<td>1253</td>
<td>23.2</td>
</tr>
<tr>
<td>Dale Ellis, Sea</td>
<td>32</td>
<td>741</td>
<td>23.2</td>
</tr>
<tr>
<td>Ricky Pierce, MIL</td>
<td>45</td>
<td>1041</td>
<td>23.1</td>
</tr>
<tr>
<td>Clyde Drexler, POR</td>
<td>50</td>
<td>1156</td>
<td>23.1</td>
</tr>
<tr>
<td>Terry Cummings, SAS</td>
<td>58</td>
<td>1333</td>
<td>23.0</td>
</tr>
<tr>
<td>Tony Campbell, Min</td>
<td>59</td>
<td>1351</td>
<td>22.9</td>
</tr>
<tr>
<td>Xavier McDaniell, Sea</td>
<td>50</td>
<td>1138</td>
<td>22.8</td>
</tr>
</tbody>
</table>
Appendix G: Sample BBall Input

<table>
<thead>
<tr>
<th>PLAYER</th>
<th>GP</th>
<th>REB</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAYMAN TISDALE, SAC</td>
<td>58</td>
<td>1312</td>
<td>22.6</td>
</tr>
<tr>
<td>MITCH RICHMOND, GSW</td>
<td>57</td>
<td>1279</td>
<td>22.4</td>
</tr>
<tr>
<td>RON HARPER, LAC</td>
<td>36</td>
<td>806</td>
<td>22.4</td>
</tr>
<tr>
<td>MAGIC JOHNSON, LAL</td>
<td>57</td>
<td>1240</td>
<td>21.8</td>
</tr>
<tr>
<td>KEVIN JOHNSON, PHO</td>
<td>50</td>
<td>1081</td>
<td>21.6</td>
</tr>
<tr>
<td>BERNARD KING, WAS</td>
<td>61</td>
<td>1292</td>
<td>21.2</td>
</tr>
<tr>
<td>JAMES WORTHY, LAL</td>
<td>58</td>
<td>1211</td>
<td>20.9</td>
</tr>
</tbody>
</table>

### REBOUNDING

<table>
<thead>
<tr>
<th>PLAYER</th>
<th>GP</th>
<th>REB</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>AKEEM OLAJUWON, HOU</td>
<td>60</td>
<td>824</td>
<td>13.7</td>
</tr>
<tr>
<td>ROY TARPLEY, DAL</td>
<td>24</td>
<td>309</td>
<td>12.9</td>
</tr>
<tr>
<td>CHARLES OAKLEY, NYK</td>
<td>55</td>
<td>655</td>
<td>11.9</td>
</tr>
<tr>
<td>DAVID ROBINSON, SAS</td>
<td>58</td>
<td>687</td>
<td>11.8</td>
</tr>
<tr>
<td>CHARLES BARKLEY, PHI</td>
<td>58</td>
<td>663</td>
<td>11.4</td>
</tr>
<tr>
<td>KARL MALONE, UTH</td>
<td>60</td>
<td>649</td>
<td>10.8</td>
</tr>
<tr>
<td>RONY SEIKALY, MIA</td>
<td>59</td>
<td>624</td>
<td>10.6</td>
</tr>
<tr>
<td>PATRICK EWING, NYK</td>
<td>59</td>
<td>621</td>
<td>10.5</td>
</tr>
<tr>
<td>ROBERT PARISH, BOS</td>
<td>58</td>
<td>596</td>
<td>10.3</td>
</tr>
<tr>
<td>SAM BOWIE, NJN</td>
<td>53</td>
<td>546</td>
<td>10.3</td>
</tr>
<tr>
<td>MICHAEL CAGE, SEA</td>
<td>59</td>
<td>586</td>
<td>9.9</td>
</tr>
<tr>
<td>BUCK WILLIAMS, POR</td>
<td>59</td>
<td>579</td>
<td>9.8</td>
</tr>
<tr>
<td>MOSES MALONE, ATL</td>
<td>58</td>
<td>561</td>
<td>9.7</td>
</tr>
<tr>
<td>DENNIS RODMAN, DET</td>
<td>60</td>
<td>582</td>
<td>9.7</td>
</tr>
<tr>
<td>JAMES DON</td>
<td>49</td>
<td>444</td>
<td>9.1</td>
</tr>
<tr>
<td>LAFAYETTE LEVER, DEN</td>
<td>57</td>
<td>508</td>
<td>8.9</td>
</tr>
<tr>
<td>AC GREEN, LAL</td>
<td>58</td>
<td>509</td>
<td>8.8</td>
</tr>
<tr>
<td>DARRELL WALKER, WAS</td>
<td>61</td>
<td>537</td>
<td>8.8</td>
</tr>
<tr>
<td>JR REID, CHA</td>
<td>58</td>
<td>501</td>
<td>8.6</td>
</tr>
<tr>
<td>MIKE GMINSKI, PHI</td>
<td>59</td>
<td>500</td>
<td>8.5</td>
</tr>
</tbody>
</table>

### ASSISTS

<table>
<thead>
<tr>
<th>PLAYER</th>
<th>GP</th>
<th>AST</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOHN STOCKTON, UTH</td>
<td>56</td>
<td>798</td>
<td>14.3</td>
</tr>
<tr>
<td>MAGIC JOHNSON, LAL</td>
<td>57</td>
<td>668</td>
<td>11.7</td>
</tr>
<tr>
<td>KEVIN JOHNSON, PHO</td>
<td>50</td>
<td>560</td>
<td>11.2</td>
</tr>
<tr>
<td>TYRONE BOGUES, CHA</td>
<td>58</td>
<td>616</td>
<td>10.6</td>
</tr>
<tr>
<td>GARY GRANT, LAC</td>
<td>44</td>
<td>434</td>
<td>9.9</td>
</tr>
<tr>
<td>ISIAH THOMAS, DET</td>
<td>59</td>
<td>565</td>
<td>9.6</td>
</tr>
<tr>
<td>TERRY PORTER, POR</td>
<td>59</td>
<td>553</td>
<td>9.4</td>
</tr>
<tr>
<td>MARK PRICE, CLE</td>
<td>50</td>
<td>422</td>
<td>8.4</td>
</tr>
<tr>
<td>TIM HARDAWAY, GSW</td>
<td>56</td>
<td>467</td>
<td>8.3</td>
</tr>
<tr>
<td>MARK JACKSON, NYK</td>
<td>59</td>
<td>454</td>
<td>7.7</td>
</tr>
<tr>
<td>Name</td>
<td>Team</td>
<td>Height</td>
<td>Weight</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>Eric Floyd</td>
<td>Hou</td>
<td>60</td>
<td>454</td>
</tr>
<tr>
<td>Vern Fleming</td>
<td>Ind</td>
<td>60</td>
<td>455</td>
</tr>
<tr>
<td>Darrell Walker</td>
<td>Was</td>
<td>61</td>
<td>466</td>
</tr>
<tr>
<td>Larry Bird</td>
<td>Bos</td>
<td>54</td>
<td>400</td>
</tr>
<tr>
<td>Johnny Dawkins</td>
<td>Phi</td>
<td>59</td>
<td>428</td>
</tr>
<tr>
<td>Nate McMillan</td>
<td>Sea</td>
<td>59</td>
<td>430</td>
</tr>
<tr>
<td>Derek Harper</td>
<td>Dal</td>
<td>59</td>
<td>423</td>
</tr>
<tr>
<td>Sherman Douglas</td>
<td>Mia</td>
<td>61</td>
<td>431</td>
</tr>
<tr>
<td>Kenny Smith</td>
<td>Atl</td>
<td>58</td>
<td>380</td>
</tr>
<tr>
<td>Michael Jordan</td>
<td>Chi</td>
<td>59</td>
<td>377</td>
</tr>
<tr>
<td>Lafayette Lever</td>
<td>Den</td>
<td>57</td>
<td>365</td>
</tr>
<tr>
<td>Spud Webb</td>
<td>Atl</td>
<td>59</td>
<td>371</td>
</tr>
<tr>
<td>Dennis Johnson</td>
<td>Bos</td>
<td>54</td>
<td>342</td>
</tr>
<tr>
<td>Clyde Drexler</td>
<td>Por</td>
<td>50</td>
<td>314</td>
</tr>
<tr>
<td>Michael Adams</td>
<td>Den</td>
<td>55</td>
<td>340</td>
</tr>
</tbody>
</table>
Appendix H
Benchmark Programs

This appendix contains the TLex files used to benchmark the Next() and Parse() routines.

/* bench1.tlx */

main (int argc,char ** argv)
{
    int i;
    if (argc == 1)
    {
        printf("error*** no filename specified\n");
        return 1;
    }

    PRINTDOTS = TRUE;
    PRINTTIMING = TRUE;

    initTLex("bench1.tld", argv[1], bench1TLexActions);

    for (i = 1; i <= 8; i++)
        doTLex(TLEX_SAME_BUFFER, argv[1], 0, 0, i);

   _deinitTLex();
}

/* definitions and macros */

/*******
digit:
    matches a digit from 0..9.
********/

(define digit (from "0" to "9"))

/*******
letter:
    matches a lowercase or uppercase letter of the alphabet.
********/

(define letter (or (from a to z)))
(from A to Z))

****************
num:
  matches a string of digits
****************/

(define num (csrpn digit (+ digit) digit))

(define tab 9)
(define cr (class 10 13))
(define notcr (notclass 10 13))
(define space 32)
(define colon 58)
(define dash 45)

(define wschar (class space tab))
(define wschars (+ wschar))
(define wspace (* wschar))
(define notws (notclass 10 13 32 9))

/* general purpose macros */

  /* (firstch p): matches a line
  * which has a first part matching p
  * (this is used to recognize game summary lines)
  */

(macro teamnames (csrpn letter (or $1 $2) letter))

/* definitions specific to BASKETBALL */

/* team definitions */

(define Atlan (teamnames "ATLANTA" "ATL"))
(define Bosto (teamnames "BOSTON" "BOS"))
(define Charl (teamnames "CHARLOTTE" "CHA"))
(define Chica (teamnames "CHICAGO" "CHI"))
(define Cleve (teamnames "CLEVELAND" "CLE"))
(define Dalla (teamnames "DALLAS" "DAL"))
(define Denve (teamnames "DENVER" "DEN"))
(define Detro (teamnames "DETROIT" "DET"))
(define Golde (teamnames "GOLDEN STATE" "GSW"))
(define Houst (teamnames "HOUSTON" "HOU"))
(define India (teamnames "INDIANA" "IND"))
(define Lacli (teamnames (or "L.A. CLIPPERS" "LA CLIPPERS"
    "LAC"))
(define Lalak (teamnames (or "L.A. LAKERS" "LA LAKERS" "LAL")))
(define Miami (teamnames "MIAMI" "MIA"))
(define Milwa (teamnames "MILWAUKEE" "MIL"))
(define Minne (teamnames "MINNESOTA" "MIN"))
(define Njnet (teamnames "NEW JERSEY" "NJN"))
(define Nykni (teamnames "NEW YORK" "NYK"))
(define Orlan (teamnames "ORLANDO" "ORL"))
(define Phila (teamnames "PHILADELPHIA" "PHI"))
(define Phoen (teamnames "PHOENIX" "PHO"))
(define Portl (teamnames "PORTLAND" "POR"))
(define Sacra (teamnames "SACRAMENTO" "SAC"))
(define Sanan (teamnames "SAN ANTONIO" "SAS"))
(define Seatt (teamnames "SEATTLE" "SEA"))
(define Utahj (teamnames "UTAH" "UTH"))
(define Washi (teamnames "WASHINGTON" "WAS"))

(define Team (or-l
  Atlan
  Bosto
  Charl
  Chica
  Cleve
  Dalla
  Denve
  Detro
  Golde
  Houst
  India
  Lacli
  Lalak
  Miami
  Milwa
  Minne
  Njnet
  Nykni
  Orlan
  Phila
  Phoen
  Portl
  Sacra
  Sanan
  Seatt
  Utahj
  Washi
))

/* definitions specifically for RULE SET 3 (Pro Basketball Finals) */

(define Scorenum
  (seq wschars num-1))

(define Overtime
  (*-1 Scorenum-1))

(define Linescore
  (seq wspace Team (exactly-1 4 Scorenum-1) Overtime-1 Scorenum-2
    (* (notclass 13 10 (from "0" to "9"))) cr))

(define HighType
  (seq "HIGH" wschars (or-l "SCORERS" "REBOUND" "ASSISTS")
    colon))

(define HighHeader
  (seq (and Team (exactly 3 letter)) wspace dash))

(define Extraline
Appendix H: Benchmark Programs

```
(seq wspace (all (seq notws any any) minus Team "HIG")
  (* notcr) cr))

(define HighEntry
  (seq HighHeader (* notcr) cr (* Extraline)))

----------------------------------------
------------------
/* RULE SET 1 (Pro Basketball Final Scores) */
(or-1 (seq cr Linescore-1 Linescore-2)
  (seq HighType-1 (+-1 (seq wspace HighEntry-1))))
==> 
{}
/*************** EOF *****************************/

```
/* bench2.tlx */

main (int argc,char ** argv)
{
    int i;

    if (argc == 1)
    {
        printf("error*** no filename specified\n");
        return 1;
    }

    PRINTDOTS = TRUE;
    PRINTTIMING = TRUE;

    initTLex("bench2.tld", argv[1], bench2TLexActions);

    for (i = 1; i <= 8; i++)
        doTLex(TLEX_SAME_BUFFER, argv[1], 0, 0, i);

    deinitTLex();
}

--------------------------------------------------------

/* definitions and macros */

/**************
digit:
    matches a digit from 0..9.
***************/

(define digit (from "0" to "9"))

/**************
letter:
    matches a lowercase or uppercase letter of the alphabet.
***************/

(define letter (or (from a to z)
    (from A to Z)))

/**************
um:
    matches a string of digits
***************/

(define num (csrpn digit (+ digit) digit))

/**************
(contains p)
matches all strings containing a match to p
**************************

(macro contains (seq anystr $1 anystr))

************
miscellaneous characters
************

(define tab 9)
(define cr (class 10 13))
(define notcr (notclass 10 13))
(define space 32)
(define colon 58)
(define dash 45)

************
whitespace and non-whitespace
************

(define Wschar (class space tab cr))
(define Ws (+ Wschar))
(define OWs (* Wschar))
(define NonWschar (notclass space tab cr))
(define NonWs (+ NonWschar))
(define NonOws (* NonWschar))

/* ruleset 2 specific */
(define MyWord NonWs)
(define MyNonWord Ws)

/* rulesets 3, 4, and 5 */
(define CComment1 (seq '//*
  (* (seq (* (notclass '*))
       (+ '*)
       (notclass '*/'))))
  (* (notclass '*'))
  (+ '*') '/')))
(define CComment2 (shortest (seq "/*" anystr "/")))
(define CComment3 (seq "/*" anystr "/"))

/* ruleset 10 */
(define Vowels (class a e i o u A E I O U))
(define WordLetter (class letter "'" '-'))
(define NonWordLetter (notclass WordLetter))
(define EndSentence (class . "?:" :!))
(define NotEndSentence (notclass EndSentence))
(define WordNoVowels (all Word minus (contains Vowels)))
(define Sentence (csrp (or Start EndSentence)
  (seq (+ NotEndSentence) EndSentence)
  empty))
(define SentenceWithWordNoVowels (and Sentence
  (contains WordNoVowels)))

/* ruleset 11 */
(define lpar "(")
(define rpar ")")
(define notpar (notclass lpar rpar))
(define MatchParen (seq lpar (* (or notpar MatchParen-2)) rpar))

/* ruleset 1 */
/* simple string */
(or "hello there"
 "happy day")
====>
{}

/* ruleset 2 */
/* parse whole text as a sequence of words */
(+ -1 (seq MyWord -1 MyNonWord))
====>
{}

/* ruleset 3 */
/* C Comment as a restricted regular expression */
CComment1
====>
{}

/* ruleset 4 */
/* C Comment as an extended regular expression */
CComment2
====>
{}

/* ruleset 5 */
/* C Comment as the preferred match */
CComment3
====>
{}``
Appendix H: Benchmark Programs

-----------------------------------------------------------
/* ruleset 6 */
/* shortest sequence of characters with 3 a's and 3 b's */

(shortest (and (seq anystr a anystr a anystr a anystr)
               (seq anystr b anystr b anystr b anystr)))
====>
{}
-----------------------------------------------------------

/* ruleset 7 */
/* a sequence of characters with 3 a's and 3 b's, prefer shorter sequences */

(shorter (and (contains [a anystr a anystr a])
              (contains [b anystr b anystr b])))
====>
{}
-----------------------------------------------------------

/* ruleset 8 */
/* longest sequence of characters without 3 a's and without 3 b's */

(longest (all anystr
          minus (contains [a anystr a anystr a])
                 (contains [b anystr b anystr b])))
====>
{}
-----------------------------------------------------------

/* ruleset 9 */
/* sequence of characters without 3 a's and without 3 b's, prefer longer */

(longer (all anystr
          minus (contains [a anystr a anystr a])
                 (contains [b anystr b anystr b])))
====>
{}
-----------------------------------------------------------

/* ruleset 10 */
/* sentences containing a word without vowels */
SentenceWithWordNoVowels
====>
{}

/* ruleset 11 */
/* matching parentheses */

MatchParen
====>
{}

/*************** EOF *************************/
/* bench3.tlx */

main (int argc, char ** argv)
{ int i;

if (argc == 1)
{ printf("error*** no filename specified\n");
  return 1;
}

PRINTDOTS = TRUE;
PRINTTIMING = TRUE;

initTLex("bench2.tld", argv[1], bench2TLexActions);

for (i = 1; i <= 8; i++)
doTLex(TLEX_SAME_BUFFER, argv[1], 0, 0, i);

deinitTLex();
}

/*****************************/

/* definitions and macros */

/********************
digit:
  matches a digit from 0..9.
********************/

(define digit (from "0" to "9"))

/********************
letter:
  matches a lowercase or uppercase letter of the alphabet.
********************/

(define letter (or (from a to z)
  (from A to Z)))

/********************
um:
  matches a string of digits
********************/

(define num (csrpn digit (+ digit) digit))

/********************
  (contains p)
matches all strings containing a match to p
******************************************************************************

(macro contains (seq anystr $1 anystr))

/************
miscellaneous characters
************/

(define tab 9)
(define cr (class 10 13))
(define notcr (notclass 10 13))
(define space 32)
(define colon 58)
(define dash 45)

/**************/
whitespace and non-whitespace
**************/

(define Wschar (class space tab cr))
(define Ws (+ Wschar))
(define OWs (* Wschar))
(define NonWschar (notclass space tab cr))
(define NonWs (+ NonWschar))
(define NonOws (* NonWschar))

/********************/
tlex input
**********************/

(define Quote "")  /* single quote */
(define Quote2 "")  /* double quote */
(define BS 92)       /* back slash */
(define Digit (from '0' to '9'))
(define MiscCode [Myanystr -1 '-----' (+ '-' )])
(define Myanystr anystr)
(define Define1 (or 1 defines 2 macros 3))
(define Separator [ws '-----' (+ '-' ) ws])
(define RuleStuff (or-89 nameit-1 Rule-1))

(define Rule [ws pat-23 ws Options-16 ws (+ '=' ) '=>' ws CCode-1 ws])
(define Options (?-88 ['option' (? s) (? :) (*-9 SubOption-1)])
(define SubOption [ws (or-76 nameit-2
  'Bind' 'NoBind' 'Overlap' 'NoOverlap')
  ws])
(define nameit ['(' 'nameit' ws ident-16 ')])

(define CCode ['{ CCode1}]
(define CCode1 [|** CCode2 |}])
(define CCode2 [** (notclass '{ '}' 'Quote Quote2 '/'))
  (or CCode wschar (seq '/' (notclass '*' '/'))
  (seq Quote2 Cstring1-1 Quote2)
  (seq Quote Cstring2-2 Quote2))}
(define ident (seq letter
 (+ (class letter _ Digit)))))

(define ident2 (seq ident-18 (?-12 ['-' number-28]) ))

(define ws (* wschar))

(define ws2 (+ wschar))

(define wschar (or (class 32 12 10 9 13)
 CComment
 ['//' (* (notclass 10 12))))

(define CComment (seq '/*' (* (seq (* (notclass '*'))
 (+ '*') (notclass '* ' '/'))) (* (notclass '*')) (+ '*') '/'))

(define number (+ Digit))

(define primpat (or-92 primpat2-14
 (seq '(' ws 'from' ws2 primpat2-11 ws2
 'to' ws2 primpat2-12 ws ')')))

(define primpat2 (or-9 number-55
 (notclass Wschar [' ''] ' ' '(' '+ ' '* ' '?' Quote Quote2)
 (seq Quote2 any Quote2)
 (seq Quote any Quote) ))

(define operator (seq (or-3 'or' '*' '**' '?' '??' '+' '++' 'seq'
 'all' 'and'
 [(or-4 'ctr' 'csr') (?-1 p) (?-2 n)]
 'atmost' 'atleast' 'exactly'
 'shortest' 'longest' 'shorter' 'longer'
 'class' 'notclass'
 ident )
 (?-11 ['- ' number-2]) ))

(define Cstring1 (* (or (notclass Quote2 BS) [BS any]))))

(define Cstring2 (* (or (notclass Quote) [BS any]))))

(define string1 (+ (notclass Quote2)))

(define string2 (+ (notclass Quote)))

(define patlist (seq pat-77 (*-6 [ws2 pat-7])))

(define pat (or-8 'empty' 'anystr' 'any'
 ident2-2 primpat-3
 (seq '$' (from '1' to '9') (?-55 'x'))
 ['(' ws operator-1 ws2
 patlist-2 ws ')']
 (seq Quote2 string1-1 Quote2)
 (seq Quote string2-2 Quote)
 ['[' ws patlist-1 ws ']] )

()}
(define main (+-1 defines-1))

(define defines
  (seq ws '(' ws 'define' ws2 ident-1 ws2 pat-1 ws ')') ws))

(define macros
  (seq ws '(' ws 'macro' ws2 ident-1 ws2 pat-1 ws ')') ws))

/* ruleset 1 */
/* sequence of tlex defines */

defines-1
  ===>

{}

/******************** EOF *********************/