(3) Change any myss[i] calls, where myss is a loop iteration, to myss.adv(i).

(4) You probably want to update context sensitive patterns to take advantage of single character left context.

access of the next loop iteration has a new syntax: ss.adv(1) instead of ss[1].

Upgrading from Version to Version

To convert from version 38 to version 68:

(1) Use the program conv.exe to do most of the conversion. The syntax is "conv <infile> <outfile>"

(2) Delete any macros for csrp, csrpn, smallest.

(3) Delete the definitions for anystr and BL.

(4) Change the macro "end" to read:
(macro end (csrpn empty any)).

(5) Replace any use of the "not" pattern with either "notclass" or "all/minus". For example:
(not "a") becomes (notclass "a")
(not (or a b 23)) becomes (notclass a b 23)
(and p1 (not p3) (not p4)) becomes (all p1 minus p3 p4)

(6) Replace any use of csr by a csrp or csrpn. If really neccessary, you can use the macro
(macro csr (csrp $1 (seq $2 anystr End))), along with
(define End (csrpn empty any))

(7) If you need a pattern which never matches, use (notclass any)

(8) If you used "fake recursion", convert the pattern to use real recursion and change the pattern accesses accordingly. If you do not know what fake recursion is, don't worry about it.

(9) Remove any declarations of TLex constants and functions. These are now accessible in tlexlib.h; a line including tlexlib.h is automatically inserted into the geneated C file.

(10) Rewrite any doTLex() calls to the new format.

(11) You probably want to update context sensitive patterns to take advantage of single character left context.

To convert from version 58 to version 68:

(1) Change the .count() calls which apply to an "or" to .which().

(2) Change the .count() calls which apply to an "?" or "??" to .thru().

Appendix

Version to Version Differences

Here are the differences between TLex v.32 and TLex v.58:

- substrings are now called parse trees.
- A new syntax for accessing parse trees.
- The match to the rule is accessed through SS instead of BL or implicitly.
- The patterns shorter, longer, shortest, longest have been implemented.
- csr has been abolished; csrp and csrpn are implemented as basic patterns instead of using csr.
- all/minus replaces not.
- There are new operators ??, **, ++.
- bounded recursion is supported.
- 'and' and 'all/minus' now match the longer string instead of a random string.
- ? and ?? are now treated more like an "or" than a loop.
- We save more information during the matching phase to simplify parse extraction. Thus we may use up more space during matching, but parse extraction should be faster.
- All rulesets are loaded at once.
- The tlex compiler is written in C++ for easy maintainability.
- The syntax for calling doTLex() has changed slightly.
- start.tlx no longer has any important declarations; everything is in the include file tlexlib.h.

Here are the differences between TLex v.58 and TLex v.68

- The csrp(n), ctrp(n) operators now take an optional additional parameter, patLeft, which can specify one character of left context.
- The specialized accesses of ?-1 and or-1 have new names: .thru() instead of .count(), and .which() instead of .count(). The incremental

In addition to tlex.lib the programmer must include the object file made from the .c file created by tlex. Finally, it is the responsibility of the programmer to provide a main() program.

```
Example:  (UNIX) (comments are in quotes)
tlex testtlx.tlx
      "compiles tlex input file testtlx.tlx,
      producing testtlx.tld and testtlx.c"
cc testtlx.c -o testtlx tlex.lib
      "compiles testtlx.c, links with tlex.lib,
      and produces the executable program `testtlx'"
testtlx....
      "the arguments and behaviour of testtlx depends
      on how the programmer defined the main() routine."
```

```
Example:  (IBM, Microsoft C) (comments are in quotes)
tlex testtlx.tlx
      "compiles tlex input file testtlx.tlx,
      producing testtlx.tld and testtlx.c"
cl -AL -c /G2 testtlx.c
      "compiles testtlx.c in large model for 80286"
link /NOI /STACK:15000 testtlx.obj,testtlx.exe,,tlex.lib;
      "links testtlx.obj with tlex.lib, and
      produces the executable program `testtlx'"
testtlx....
      "the arguments and behaviour of testtlx depend
      on how the programmer defined the main() routine."
```

```
 A REPORT for Rule:1, Ruleset 2:
============================
1: London dealers said the currency markets were !concerned
Rule: [238 - 247]: count 0
"concerned"
    and-1: [238 - 247]: count 0
    "concerned"
      atleast-2: [238 - 247]: count 9
      "concerned"
            loop-iteration: [238 - 239]: count 0
            "c"
            loop-iteration: [239 - 240]: count 0
            "o"
            loop-iteration: [240 - 241]: count 0
            "n"
            loop-iteration: [241 - 242]: count 0
            "c"
            loop-iteration: [242 - 243]: count 0
            "e"
            loop-iteration: [243 - 244]: count 0
            "r"
            loop-iteration: [244 - 245]: count 0
            "n"
            loop-iteration: [245 - 246]: count 0
            "e"
            loop-iteration: [246 - 247]: count 0
            "d"
```

The report program prints out which ruleset and rule it matched; then the line the match occurred on. The exclamation point is put right before the first character in the match.

Then comes a printout of rule match, showing each substring that was found during parse extraction. See the previous description of the ss procedure “print_binding” for an explanation of the remaining printout.

9.4. The TLex Runtime System

In order to make an application that uses the .tld files created by tlex, the programmer must include the tlex runtime system during linking. At the moment the tlex runtime system is in a library file, tlex.lib. All the declarations needed to interface to the runtime system are in the file tlexlib.h. The file tlexlib.h must be accessible by the C compiler because the C++ file that tlex generates always #includes tlexlib.h; it may be put in the current directory or the “include” directory. You may want to persuse tlexlib.h for useful functions and constants.

IMPORTANT NOTE: When using the TLex runtime library with Microsoft C, it is crucial that you use the /STACKSIZE option to the linker to increase the size of the stack. /STACKSIZE:15000 works.

-p, -P: “print dots”
Print dots during matching, which shows progress made during time-consuming matches.

-t , -T: “print timing”
Print timing.

-c <num>, -C <num>: “set cache size”
Set transition cache to use 1000 * (2**<num) * 16 bytes. For large rulesets, increasing this number makes matching faster. The default is approximately <num> = 3.

-r <num>, -R <num>: “set recursion level”
Set recursion level to <num>. The default is 3.

<num>: “run specific ruleset”
Only run ruleset <num>. If no <num> is given, every ruleset is run, from first to last.

Example: tlexdbg foo.tld foo.txt 3 -L -r 5 -p

This line will search the input file foo.txt for the patterns in the third ruleset of the TLex data file foo.tld. Tlexdbg will print out dots as it searches, it will translate the input to lowercase as it matches, and it will use a recursion bound of 5.

Tlexdbg runs the data file on the inputfile, printing out a report of each match instead of calling the rule actions. Here is a typical report for the pattern

```
(and-1 word (atleast-2 5 any))
```

with a description of the various parts of the report:

previous versions. There is no longer an upper limit on the number of loop iterations TLex remembers, as there was in previous versions.

9.3. The Programs tlex and tlexdbg

Here are detailed descriptions of the programs "tlex" and "tlexdbg".

```
tlex infile.tlx [-u] [-o] [-n] [-d datafile.tld] [-c cfile.c]
```

The first argument is the TLex rulefile. It must have the extension ".tlx". The optional arguments, which may appear in any order, mean the following:

-o, -O : "only recompile the C file"
  If you made a change to the C-code in a .tlx file, but no change to any pattern, then you can use this option to speed recompilation.

-d datafile.tld : "name the output data file datafile.tld".
  By default, tlex puts the output data file in a file with the same name as infile.tlx but with the extension ".tld". Use the -d option to name the output something else; however, the output data file must have the extension ".tld".

-c cfile.c : "name the output c file cfile.c"
  By default, tlex puts the output C file in a file with the same name as infile.tlx but with the extension ".c". Use the -c option to name the output something else; however, the output C file must have the extension ".c".

-n, -N : "make an NFA data-file".
  This option is obsolete

-u, -U : "unoptimize"
  This option is obsolete.

```
tlexdbg datafile.tld inputfile [options]
```

The first argument is the TLex data file, which should have the extension ".tld". The second argument is the input file to match the data file against. The options are as follows:

-l, -L: “lowercase”
  Match the lowercase of the input.

```
recursionMaxTLex(int j)
```

TLex implements recursion by using bounded recursion. A definition can be nexted within itself upto j times; the next nesting of the definition is treated as a pattern which does not match anything. This procedure can be used to set the recursion bound to j. It must be called before calling doTLex().

```
transitionCacheTLex(int j)
```

Set the transition cache to 1000 * (2**j) * 16 bytes. For large patterns, larger j's imples faster matching. The default transition cache is approximately j=3. The transition cache exists between the initTLex() and deinitTLex() calls.

```
extern bit8 TLex_Translation_Table[];
initTransTLex(int code);
```

The TLex_Translation_Table enables the tlex programmer to map any character to any other character. This is useful, for example, in mapping uppercase letters to lowercase. TLex_Translation_Table[i] has the translation of character i. The table has entries from 0 to 255. It is even possible to switch letters: for example, setting TLex_Translation_Table['9'] to '1' and setting TLex_Translation_Table['1'] to '9' will cause Tlex to treat nines in the input as ones, and ones as nines.

Initially, TLex_Translation_Table maps every character to itself. The routine initTransTlex() may be used to alter TLex_Translation_Table, or the programmer may alter it directly. Calling initTransTLex(0) resets the table to map every character to itself. Calling initTransTLex(1) resets the table to map uppercase letters to lowercase letters.

```
lockTLex(int rulesetnum)
unlockTLex(int rulesetnum)
```

These calls do nothing; they are present for backwards compatibility with previous versions.

```
maxloopTLex(int itercount)
```

This procedure does nothing; it is present for backward compatibility with

```
Example:

void testTLexAction testTLexActions;

initTLex("test.tld", "tlexout2.dat", testTLexActions);
      /* initializes TLex to read from the data file test.tld and
       the input file tlexout2.dat.*/

doTLex(TLEX_FILE_BUFFER, "yen.txt", 0, 0, 3);
      /* calls the third ruleset on the whole file yen.txt. */

doTLex(TLEX_FILE_BUFFER, "yen.txt", 100, 200, 5);
      /* calls the fifth ruleset on the text in yen.txt
        between positions 100 and 200. */

char * myblock;
myblock = get_text_from_somewhere();
doTLex(TLEX_MEMORY_BUFFER, myblock, 0, 0, 1);
      /* calls the third ruleset on the text in myblock. */

deinitTLex();
      /* deinitializes TLex. */
```

9.2. Advanced Procedures

```
inhibitTLex(int rulenum, long position)
```

inhibitTLex() tells TLex to ignore all future matches to the rule with index "rulenum" that start before “position”. Internally, each rule has a current "inhibit position". Matches to a rule that start before the rule’s inhibit position are ignored. This routine is used internally by the Overlap/NoOverlap processor, so the programmer must be aware that the inhibit position of a rule can change if the rule has the NoOverlap option set and it is matched.

The programmer can use inhibitTLex() to better control the matching process. If rules 5, 6, and 8 are matched at position p, and the action of rule 5 calls inhibitTLex(6, p+1) and inhibitTLex(8, p+1), then the actions of rule 6 and rule 8 will NOT be called. The inhibit position of a rule is checked right before the rule action is called.

Calling inhibitTLex(0, p) sets the global inhibit position to p. In effect this makes sure that no rule will match until position p.

Each recursive call to doTLex() gets its own set of inhibit positions, initialized to the first position in the input. As a result, the programmer does not have to worry about a recursive call messing up the current inhibit positions.

```
doTLex(int buffertype,  /* type of buffer to read from */
       char * buffer,   /* where the input is */
       longnumber start,/* starting position of search */
       longnumber end,  /* position of end of search */
       int rulesetnum)  /* which ruleset to do */
```

Call doTLex() after calling initTLex() and before calling deinitTLex(), to run a ruleset. Running a ruleset means searching the subset of "buffer" indicated by “subs” for matches to patterns declared in ruleset “rulesetnum”, calling the rule actions when a match is found. See the section "Calling Rules" for more discussion of what running a ruleset means.

"buffertype" is one of TLEX_MEMORY_BUFFER or TLEX_FILE_BUFFER or TLEX_SAME_BUFFER. These constants are defined in tlelib.h. If buffertype is TLEX_MEMORY_BUFFER then buffer points to a block of memory containing the text to search. If buffertype is TLEX_FILE_BUFFER then buffer is a string giving the name of the file to search. It is quite common to run multiple rulesets on the same ruleset, and to call doTLex() on a subset of the same input, from within a rule action. In these cases, it is time-consuming to reopen the input each time. This motivates the next buffertype: if buffertype is TLEX_SAME_BUFFER then buffer is ignored and the “current buffer” is used again. The “current buffer” is only set during a doTLex() call and when initTLex() is called with a non-nil buffer parameter.

Parameter “start” and “end” are text positions that allow you to restrict the search to a subset of the input. A file with filesize bytes goes from position 0 to position filesize. A memory block is considered an array of bytes going from position 0 to position memory_block_size. If start = end = 0, then the whole memory block or file is searched; in this case the size of a memory block is found by searching for a terminating zero byte, and the zero byte is not considered part of the input.

"rulesetnum" is the 1-based index of the ruleset to run.

What about recursion? Recursion, which in this case would be calling doTLex() while executing the action part of some rule, IS allowed. Another type of recursion, calling initTLex() while executing an action in a different rulefile, is NOT allowed. However, one program may use two or more different tlex data files, as long as initTLex() and deinitTLex() are called for one file before calling initTLex() for another file.

When filling out subs in preparation for calling doTLex(), remember to consider the requirements of right context sensitivity. See the discussion of defining "word" in the section describing context sensitive right patterns.

9. Programmer's Interface Reference

This section describes the routines for running a ruleset and controlling the resulting behaviour.

9.1. Fundamental Procedures

```
typedef void (*funcActionPtr)(int, int, ss);
initTLex(char * datafile, char * buffer,
            funcActionPtr fAP)
```

```
deinitTLex()
```

Call initTLex() to load in the essential parts of "datafile", in preparation for calling doTLex. "datafile" should be the name of a data file created by the tlex compiler. Call deinitTLex() to unload the information loaded by initTLex(). Every initTLex() call should have exactly one matching deinitTLex() call. The third argument to initTLex() is the name of a procedure: if the datafile is "foo.tld" then pass fooTLexActions. (tlex automagically generates the fooTLexActions() procedure for you, by appending the name of the output data file with "TLexActions". This third argument is needed so that one program may use more than 1 rulefile.)

If buffer is non zero, then it is the name of a file to open. An input file can be specified here to open a file on which a number of rulesets will run. Then doTLex() can be called with TLEX_SAME_BUFFER for each ruleset, avoiding the need to reopen the file each time a ruleset is called. If buffer is 0, then no file is pre-opened and the user should specify the input in the next call to doTLex().

Example:

```
void fooTLexActions(int ruleset, int rulenum, ss SS);
      // forward declare fooTLexActions

main(argc, argv)
...
{
      initTLex("foo.tld", argv[1], fooTLexActions);
      ....
      deinitTLex();
}
```

Rulesets also provide a convenient organizing method for the rules. One ruleset can locate a header, for instance, and then choose among a number of rulesets based on the information in the header.

Rulesets can be named in a way similar to rules. If a ruleset contains (nameit RulesetFoo), then TLex defines RulesetFoo to be a constant referring to that ruleset. This constant may be used by the functions that require a ruleset number, such as doTLex().

8.2.3. Running a Ruleset

TLex essentially creates a powerful subroutine for the programmer's application. Typically, the programmer calls this subroutine using doTLex(...), telling it what input to read and which ruleset to run. Here we describe what doTLex() does when it "runs a ruleset". The full description of the programmer's interface to TLex can be found in Appendix A. Here we give an overview of the behavior of doTLex().

The function doTLex() first scans the input for occurrences of patterns in the requested ruleset, recording which rules from the ruleset match at which positions. Then doTLex() calls the actions for all rules which start a match at position 0; then all rules which start a match at position 1 have their actions called, then position 2, etc.. If several rules start a match at the same position, they are called in the order of their appearance in the ruleset, from lowest numbered rule to highest. For example, if rules 2, 3, and 5 match at a position, the action for rule 2 is called first, then the action for rule 3, and then the action for rule 5.

```
/* pseudo code for the main loop */
for (position from 0 to end) do
    rulematches = set of rules that match at position,
                  if any.
    for (rule in rulematches from first to last) do
            extract parse and call action for rule;
```

Note: If a rule has the NoBind option set, the main loop does not extract a parse before calling the rule. If a rule has the NoOverlap option set, then a rule is only matched if it is not overlapped from a previous match to the same rule. Calls to inhibitTLex(), described in the next section, can also alter the behavior of the main loop.

The Overlap and NoOverlap options control how TLex ignores matches. Consider defining a word as (csrpn (+ letter) letter). If TLex found **every** match from left to right, when given the text:

```
"This is sample text"
```

and asked to find a word, it would first find "This". Then, unfortunately, it would find "his", followed by "is" and "s". It is usual that after matching a rule, we want any subsequent match to start after the end of the previous match. By giving the NoOverlap option, as in the example to follow, we get the desired behavior:

```
word
options: NoOverlap
==>
{}
```

On our sample input this rule would match "This", "is", "sample", and finally "text". Actually, since NoOverlap is the default, the "options: NoOverlap" line is extraneous. If the Overlap option is given, on our sample input this rule would match "This", "his", "is", "s", "is", "s", etc.

The Bind and NoBind options deal with parsing. Compared with finding the start of the matches in the input, parsing takes a long time. By giving the NoBind option, one can tell TLex not to parse when it finds a match for that rule. One can still find out the location of the start of the match, by calling SS.start(), but no other parse accessing functions should be used. The Bind option, which is the default, instructs TLex to extract a parse before calling the action.

A match to a rule with the NoBind option set is treated as a zero length match starting and ending at the start of the actual match. This is important to know if the rule also has the NoOverlap option set; in effect, the NoBind option nullifies the NoOverlap option.

The (nameit <name>) option provides a way to symbolically refer to a rule in a call to InhibitTLex(). TLex defines <name>, in the output .c file, to be the number of the rule.

8.2.2. Rulesets

A number of rules may be put together in a **ruleset**. All rules in a ruleset are searched simultaneously. The effective limit of how many rules can be in a ruleset is determined by the complexity of the rules and their mutual interactions. The space required for compiling a bunch of rules can theoretically grow exponentially if they interact harmfully. If this happens, just split the ruleset into a set of smaller rulesets.

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.

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. After experimentation, erroring immediately appears much more desirable.

8.2. The TLex Control Structure

The basic element of the TLex control structure is the rule, which is composed of a pattern and a C or C++ action, with an optional set of options. Here is a typical rule, which counts how many words ending in “ed” are in the input and keeps the length of the longest word found.

```
(and Word-1 (seq anystr "ed"))
options: NoOverlap Bind (nameit EDrule)
==>
{   // C++ code
    count = count + 1;
    if (SS["Word-1"].length() > largestSoFar)
       largestSoFar = SS["Word-1"].length();
}
```

Essentially, each time the pattern is located in the input, a parse is extracted and the corresponding action is executed. The parse tree can be accessed in an action through the predefined variable SS of type ss, using the access functions described previously.

8.2.1. Rule Options

Each rule can be given 0 or more options. To specify one or more rule options, include the following line after the rule pattern and before the “==>”:

```
options: <option1_name> <option2_name> etc..
```

The possible option names are “Overlap”, “NoOverlap”, “Bind”, “NoBind”, and (nameit <foo>), where foo is an arbitrary identifier. Defaults are “NoOverlap” and “Bind”.

with dots, like a field access in C++: ff[“or-1”][“bar-2”] can be written ff[“or-1.bar-2”]. In C this looks like RS(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"]
C:   RS(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"]
C:   RS(RI(RS(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"]
C:   RS(RI(RS(foo, "*-1"), j), "+-1[1].foo-2.bar-2.seq-3")
```

8.1.12. 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 C++
accesses include X["or-1"], X["seq-1"], X["foo-2"].
```

8.1.13. 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.

then foo.thru() returns TRUE iff the subpattern was matched. If foo is uninitialized an error is reported.

8.1.10. Specialized OR Access

```
posnumber foo.which()         (C++)
posnumber ssWhich(ss foo)     (C)
```

When foo represents a parse tree for

- (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 then or-1.which() returns 1.

8.1.11. 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.

```
(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))))))
```

Given X an ss, we can access any ss contained in X by writing X[“Y”] in C++, or RS(X, “Y”) in C, where Y is the name of the desired pattern inside X.

```
ss operator[](char* access_string)     (C++)
ss RS(ss foo, char * access_string)    (C)
```

For example, if ff is an ss matching foo, then the following are all the legal C++ 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

8.1.8. Specialized Loop Iteration Access

```
ss foo.adv(longnumber j)          (C++)
ss ssAdv(ss foo, longnumber j)    (C)
```

This operator requires that foo represent a match for one of the iterations of a looping pattern, i.e. foo = bar[i] where 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++ code
ss fooiter;
for (fooiter = bar[0];
     fooiter;
     fooiter = fooiter.adv(1))
   use(fooiter);

/* C code */
ss fooiter;
for (fooiter = RI(bar, 0);
     ssInited(fooiter);
     fooiter = ssAdv(fooiter, 1))
   use(fooiter);
```

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.

8.1.9. Specialized Optional Test

```
int foo.thru()        (C++)
int ssThru(ss foo)    (C)
```

When foo represents one of the patterns

- (?-1 pat1)
- (??-2 pat2)

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.

8.1.7. Specialized Loop Accesses

```
poslongnumber foo.count()       (C++)
poslongnumber ssCount(ss foo)   (C)
```

```
ss foo[longnumber i]         (C++)
ss RI(ss foo, longnumber i)  (C)
```

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)
- (**-4 pat)
- (+-5 pat)
- (++-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 there is no such iteration 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.

8.1.4. Convert to a C-string

```
char * foo.tostring()       (C++)
char * ssTostring(ss foo)   (C)
```

```
char * foo.tostring(void * mem, longnumber memsize)     (C++)
char * ssTemps(ss foo, void * mem, longnumber memsize)  (C)
```

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.

8.1.5. Start, End, and Length

```
longnumber foo.start()       (C++)
longnumber ssStart(ss foo)   (C)
```

```
longnumber foo.end()         (C++)
longnumber ssEnd(ss foo)     (C)
```

```
longnumber foo.length()      (C++)
longnumber ssLength(ss foo)  (C)
```

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.

8.1.6. Convert to Longnumber

```
longnumber foo.tonum()       (C++)
longnumber ssTonum(ss foo)   (C)
```

Returns the numeric equivalent of foo. Leading spaces are ignored; the number

8.1.2. Display

```
void foo.print_binding()     (C++)
void print_binding(ss foo)   (C)
```

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))
(define decNumber (seq-1 number-1 . number-2))
(define main (seq Space decNumber-1))
```

When "main" is matched against the string " 5.6" and the resulting parse tree is printout out, a display like this results:

```
main: [40 - 44]: count 424
" 5.6"
    decNumber-1: [41 - 44]: count 0
    "5.6"
        seq-1: [41 - 44]: count 0
        "5.6"
            number-1: [41 - 42]: count 0
            "5"
            number-2: [43 - 44]: 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. The "count" value is the 1-based index of the subpattern that was parsed, for an OR pattern; for a LOOPING pattern the "count" value gives the number of iterations of the loop.

8.1.3. Access Individual Characters

```
foo.string(longnumber i)         (C++)
ssString(ss foo, longnumber i)   (C)
```

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.

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 has start=end; the substring which is the same as character i has start = i, end = i+1. See figure 8.

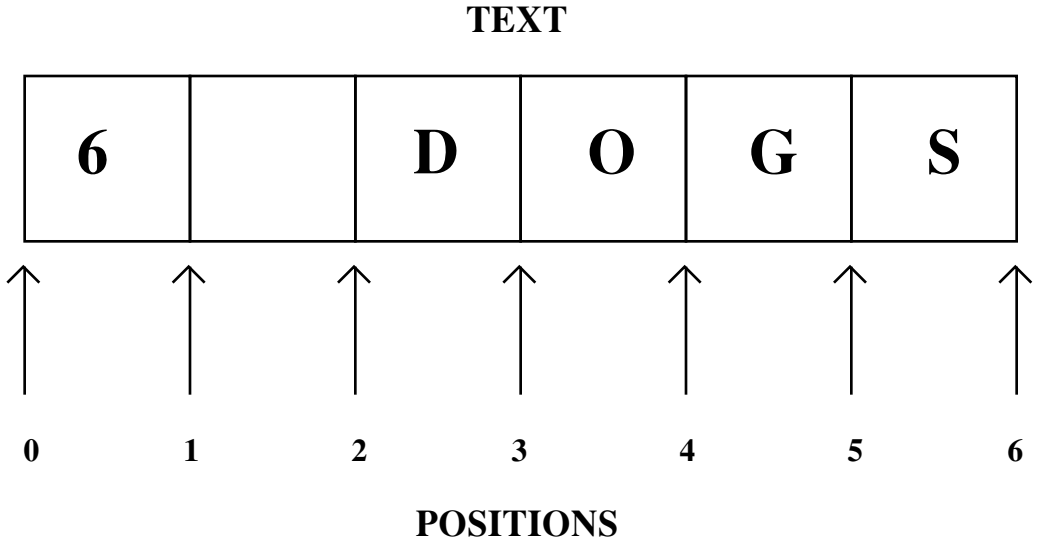


Figure 8: Text Positions

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 various functions for accessing a parse tree. In the following, consider foo to be an arbitrary parse tree, of type ss.

8.1.1. Assignment and Argument Passing

```
int(ss foo)            (C++)
int ssInited(ss foo)   (C)
```

Objects of type ss can be efficiently assigned and passed by value. ssInited(foo) returns TRUE iff ss is initialized. Converting an ss into an integer, which is possible in C++, returns TRUE iff ss is initialized. In C++ the following code calls Doit() only if foo is initialized:

```
if (foo) Doit();
```

essence, naming is specifying the desired detail of the parse tree resulting from a match.

The expression SS["decNumber-1.number-1"].tostring() is called a parse access. It is a C++ expression which finds the parse tree representing number-1 inside the parse tree representing decNumber-1 inside the parse tree SS, and converts it to a C string. SS is the parse tree TLex creates during parse extraction to represent the whole pattern. All the possible accesses will be described in this section.

8.1. 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.

8. Rules and Parse Tree Reference

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:

```
pat1
options: <option>
==>
{ C or C++ expression }
```

"pat1" is a pattern as defined in the pattern language section. The (optional) options to the rule are described later. The C-expression should be enclosed in braces. It can be an arbitrary set of C or C++ statements. The spacing, including carriage returns, is irrelevant. Of course, the user is encouraged to format patterns to maximize readability.

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 will be detailed in this section.

As an example, the following rule, with associated definitions, matches a decimal number and prints out the integer part.

```
(define digit (from "0" to "9"))
(define number (csrpn (+ digit) digit)
(define decNumber (seq-1 number-1 . number-2))

(seq (all any minus digit) decNumber-1)
==>
   {
    char * mystring;

    // C++ code
    mystring = SS["decNumber-1.number-1"].tostring();

    cout << "found a decimal number;\n";
    cout << "integer part is" << mystring << "\n";

    delete mystring;
   }
```

The operators seq-1, number-1, number-2, and decNumber-1 are all NAMED. Any operator can be named (except for the primitive patterns and EMPTY and ANY), 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

to match is preferred. These will be illustrated in greater detail in the section on predicting the parse. Here we contrast shorter with shortest, and longer with longest.

(SEQ (+ Digit) (OR “3” “!”)), matches a string of digits followed by “3” or “!”. On the input string “12345!” the leftmost match could be either “123” or “12345”. Both are correct; the one actually returned depends on details of the implementation. In contrast, the pattern

(SEQ (SHORTER (+ Digit)) (OR “3” “!”))

says to prefer to match shorter initial strings of Digits, which means on the same input string “123” would be matched.

(SEQ (LONGER (+ Digit)) Digit) on the input string “12345!” produces the leftmost match “12345”. In contrast (SEQ (LONGEST (+ Digit)) Digit) on the same input string would not match at all, since (LONGEST (+ Digit)) insists on matching “12345”, leaving no numbers to match the final Digit.

matches patLeft. patLeft should be a character or set of characters. (ctrp pat patRight) is allows any character to the left.

(ctrpn patLeft pat patRight) matches what pat matches when no prefix of the string starting where pat starts matches patRight, and the character to the left does not match patLeft. patLeft should be a character or set of characters. (ctrpn pat patRight) allows any character to the left.

```
Examples:

(ctrp Word Address) would match the first word of an Address,
given appropriate definitions for Address and Word.

Assume FullFloat matches numbers like "123.456" and HalfFloat
matches numbers like "789.". To define a pattern which matches
floats of either type, one might consider
(or FullFloat HalfFloat). Unfortunately, on the input "456.789
times" this MIGHT match "456." as a HalfFloat. We really want to
match a FullFloat or a HalfFloat, but never a HalfFloat when we
can match a FullFloat. The following CTRPN pattern accomplishes
this:

(or FullFloat (ctrpn HalfFloat FullFloat)).
```

7.12. SHORTER and LONGEST Patterns

(shorter pat)
(longer pat)
(shortest pat)
(longest pat)

As amply illustrated by the previous examples, one pattern can match many different strings, all starting at the same position. For example, (+ digit) matches several different strings starting at the front of the input "1234": "1", "12", "123", "1234". The longest and shortest operators restrict the pattern to matching the longest and shortest possible strings from each starting position. So (LONGEST (+ digit)), starting at the front of "1234" would match only "1234", whereas (SHORTEST (+ digit)) would match only "1".

(SHORTEST (SEQ ANYSTR b)) matches the shortest string ending in b. On the input string "aaaaabaaab" the leftmost match is "aaaaab".

The patterns (LONGER pat) and (SHORTER pat) match exactly what pat does; however, the longer or shorter match that allows the rest of the pattern

```
(define whitespace (+ (or 9 10 12 13 32)))
(define Digit (from "0" to "9"))
(define nonDigit (all Digit minus (from "0" to "9")))
(define number (csrpn Digit (+ Digit) Digit))
(define numberList [number (* (seq nonDigit number))])
(define numberListLongest (csrpn numberList
                            (seq nonDigit num)))

On the input "this is a sequence of numbers: 100 200,300   400."
numberList would match:                      ^^^^^^^^^^^
                                             ^^^^^^^
                                             ^^^
                                                 ^^^^^^^
                                                 ^^^
                                                     ^^^   ^^^

On the input "this is a sequence of numbers: 100 200,300   400."
numberListLongest would match:               ^^^^^^^^^^^
                                                 ^^^^^^^
                                                     ^^^   ^^^


To match a word:

(define whitespace (+ (or 9 10 12 13 32)))
(define alphanumeric (or (from "0" to "9")
                         (from A to Z)
                         (from a to z))
(define word (csrpn alphanumeric (+ alphanumeric) alphanumeric))

When given the input "Hi there. My name is steve! Wow"
word                  ^^ ^^^^^  ^^ ^^^^ ^^ ^^^^^  ^^^
```

7.11. Context Total Right Patterns

(ctrp pat patRight)
(ctrpn pat patRight)
(ctrp patLeft pat patRight)
(ctrpn patLeft pat patRight)

Ctrp stands for "context total right prefix", while ctrpn stands for "context total right prefix not". They are variants of the csrp and csrpn patterns. While the latter patterns consider the context starting at the end of pat, the ctrp(n) patterns consider the context starting at the start of pat.

(ctrp patLeft pat patRight) matches what pat matches when a prefix of the string starting where pat starts matches patRight, and the character to the left

Really, we want to say that a number matches any substring which has a nondigit to the left and right and digits in the middle - but we do not want the nondigits to be part of what we consider a number to be. In other words, a number is a sequence of digits, but only in the context where the letters to the right and left are nondigits. We call this a context sensitive pattern.

Unfortunately, it does not seem possible to efficiently match full context sensitive patterns. However, TLex allow the specification of arbitrary right context and a single character of left context.

(csrp patLeft pat patRight) matches what pat matches when a prefix of the string to the right matches patRight, and the single character to the left matches patLeft. patLeft should specify a character or set of characters. (csrp pat patRight) is a simplified version that allows any character on the left.

(csrpn patLeft pat patRight) matches what pat matches when no prefix of the string to the right matches patRight, and the character to the left does not match patLeft. patLeft should specify a character or set of characters. (csrpn pat patRight) is a simplified version that allows any character on the left.

TLex “hallucinates” a special character before the first character of the input, to help in matching the start of the input. This character can never be directly matched, it can only match a left context pattern. It can be accessed through the predefined definition “Start”. “Start” defaults to ascii value 128, but the programmer may redefine it to any other value just by defining Start.

```
Examples:

To match the start of the input:

(define Begin (csrp Start empty empty))

To match the start of each line (including the first):

(define CR 10)
(define LineStart (csrp (or CR Start) empty empty))

Another way to match the start of each line involves redefining
Start:

(define Start CR)

(define LineStart (csrp CR empty empty))

To match a number or a list of numbers:
```

```
(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.
    (upto "foo") on the string "I ate foo or foo" matches:
        "I ate foo or foo"
         ^^^^^^ ^^^^^^ ^^
          ^^^^^  ^^^^^  ^
           ^^^^   ^^^^  ><
            ^^^    ^^^
             ^^     ^^
              ^      ^
              ><     ><

(macro upto
    (shortest (csrp (* any) (or $1 end))))
```

7.10. Context Sensitive Right Patterns

(csrp pat patRight)
(csrpn pat patRight)
(csrp patLeft pat patRight)
(csrpn patLeft pat patRight)

Csrp stands for "context sensitive right prefix", while csrpn stands for "context sensitive right prefix not". Context sensitive patterns are crucial for even everyday pattern matching. For example, to match a number we can try (+ Digit). But on the string " 1234 " this matches the substrings "1", "2", "234" in addition to the one we want, "1234". Another try is:

```
(define Number (seq nonDigit (+ Digit) nonDigit))
```

The problem with this is that it matches not only numbers but also the characters to the right and left of the number. Thus if we defined a list of numbers as follows:

```
(define NumberList (+ (seq Number Space)))
```

then NumberList would not match " 12 33 47 62 " because the space after 12 would be matched by "number" and then "Space" has nothing to match. (+ Number) also fails because the space after 12 and before 33 would have to be part of both "Number" patterns. While it is possible to write a new NumberList pattern as a restricted regular expression, it is not possible to reuse the Number pattern in creating it.

After a macro like (macro name23 pat0) is defined, one can use it like any other operator. (name23 pat1 pat2) would be substituted by pat0, with all occurrences of $1 in pat0 substituted by pat1 and all occurrences of $2 in pat0 substituted by pat2, etc.. This simple substitution sometimes causes problems with naming (Naming is introduced later, but we describe the problem here.). Consider the following macro:

```
(macro double (seq $1 $1))
```

This macro takes one argument and matches two consecutive occurrences of it. But if it is used as follows, trouble arises:

```
(double (seq-1 a b))
```

This gets expanded to (seq (seq-1 a b) (seq-1 a b)) which is illegal, because it has two occurrences of the named operator seq-1. In order to avoid this problem, the macro writer may use the patterns $1x ... $9x in addition to $1 ... $9. The former patterns stand for the 1st through 9th arguments, unnamed. Thus, we could rewrite double as follows:

```
(macro double (seq $1x $1))
```

Then, (double (seq-1 a b)) would be expanded to be:

```
(seq (seq a b) (seq-1 a b))
```

The first substitution of $1 is unnamed, removing the double occurrence of seq-1.

Examples:

The (MyShortest p) pattern will match, for a given starting position, the shortest string from that position that matches p. Another way of saying this is "(MyShortest p) matches a substring s when p matches s and p does not match a prefix of s." It can be implemented as a macro defined as follows:

```
(macro MyShortest
    (all $1
     minus (seq $1x (+ any))))
```

(upto2 p) matches anything up to but not including all of a substring matching p;

```
(macro upto2
    (all anystr minus (seq anystr $1 anystr)))
```

(upto2 "123") on the string "I bought 4123" matches every substring that ends before the 3 or earlier.

7.8. Definitions and Bounded Recursion

```
(define name pat1)
```

The "define" pattern lets one define a name for a pattern. This makes patterns more readable, efficient, shorter, and reusable. Defines should only appear in the definition section of a TLex input file. Recursive or eventually recursive definitions are allowed, but they are implemented by limiting the recursion to a fixed depth, which defaults to 3. A full discussion of bounded recursion was presented above. The name being defined must be a sequence of letters and numbers and underscores at least 2 letters long, in order to distinguish definition names from single character patterns. (Technically the first letter must be a letter in a-z or A-Z, and the rest of the letters can be a-z or A-Z or 0-9 or underscore.) The order of the definitions in a file is immaterial, but it is good style to define subpatterns before the patterns that use them.

```
Examples:

(define Digit (from 48 to 57))

(define Non_Digit (all any minus Digit))

(define Number0 (seq Non_digit (+ Digit) Non_digit))

The csrpn pattern is defined later.  Here we define "End" to match
the very end of the string.

(define End (csrpn empty any))
```

7.9. Macros

```
(macro name23 pat)
```

TLex provides a primitive macro capability. A macro provides a simple way to create new pattern operators as combinations of existing operators. Macros, like definitions, can only appear in the definition section of the TLex input file. (macro name23 pat) defines name23 to be a macro equal to pat, just like a definition. However, unlike a definition, pat may contain the special patterns $1, $1x, ... , $9, $9x. These stand for the 1st..9th arguments to the macro when the macro is later encountered, which will be patterns.

```
(and (seq b a b b (* any))
     (seq (* any) b a b a))
 matches any substring which starts with "babb" and ends
 with "baba":
                          aababbaaaabbbbababaabababbaabaaaabb
                            ^^^^^^^^^^^^^^^
                            ^^^^^^^^^^^^^^^^^
```

7.7. The ALL/MINUS Pattern

```
(all pat1 minus pat2 pat3 ...)
```

The all/minus pattern matches any string which matches pat1 but **does not** match pat2 or pat3. It prefers to match longer strings. The all/minus pattern replaces the "not" pattern, which was a tricky operator present in an older version of TLex. For example, most people expect (not a) to match every character but "a". However, it really matches every string except for "a", including the empty string and all strings longer than 1 character. Using all/minus one would write (all any minus a), which matches exactly what one would expect.

As an example, (all anystr minus (seq a a)) matches all strings which do not match (seq a a). This includes the empty string, all strings of length 1, 3, 4, ... , as well as all strings of length two where both letters are not "a".

The best way to write "all characters except for c" is (notclass c). Another way is (all any minus c). They both compile to the same thing (because TLex optimizes such all/minus patterns) but the "notclass" form is more concise and less error-prone.

```
Examples:

(all anystr minus (* any))
    matches no strings.

(all VariableName minus Keyword) matches all VariableNames,
except those that are also Keywords.

(all (exactly 5 any)
 minus (seq (* any) b (* any) b (* any)))
matches any 5 letter substring that doesn't have 2 b's in it:
                        aababbaaaabbbbababaabababbaabaaaabb
                             ^^^^^              ^^^^^
                              ^^^^^              ^^^^^
                                                  ^^^^^
                                                   ^^^^^
```

```
Examples:

                          aababbaaaabbbbababaababbaabaaaabb
(+ (seq a b))              ^^       ^^   ^^   ^^    ^^   ^^
                             ^^          ^^^^   ^^
                            ^^^^           ^^ ^^^^

                          aababbaaaabbbbababaababbaabaaaabb
(seq b                        ^^    ^^           ^^       ^^
     b                        ^^^^   ^^          ^^^^
     (* (seq a a)))           ^^^^^^  ^^

                          aababbaaaabbbbababaababbaabaaaabb
(seq a b b                    ^^^   ^^^          ^^^      ^^^
     (atmost 1                ^^^^^              ^^^^^
             (seq a a)))
```

7.5. The OR Pattern

```
(or pat1 pat2 pat3 ...)
```

The "or" pattern matches a substring which matches pat1 or pat2 or pat3. (or pat empty) is the same as (? pat) for example. (* (or a b)) matches any substring made up of just a's and b's.

```
Examples:
                                aababbaaaabbbbababaababbaabaaaabb
(or (exactly 3 a)                     ^^^ ^^^               ^^^
    (atleast 3 b))                     ^^^ ^^^               ^^^
                                          ^^^^
                                aababbaaaabbbbababaababbaabaaaabb
(seq b b (or (seq a a a)              ^^^^^   ^^^^^^
             (seq a b a b)))
```

7.6. The AND Pattern

```
(and pat1 pat2 pat3 ...)
```

The "and" pattern matches any substring which matches pat1 and pat2 and pat3. It prefers to match longer strings.

```
examples:

(and (seq (* any) b (* any) b (* any))
     (seq (* any) a (* any) a (* any)))
 matches any substring with atleast 2 b's and atleast 2 a's.
```

7.4. Looping Patterns

(* pat)
(** pat)
(+ pat)
(++ pat)
(? pat)
(?? pat)
(atleast n pat)
(exactly n pat)
(atmost n pat)

These are the looping patterns. (atleast n pat) matches n or more consecutive occurrences of pattern pat. (atmost n pat) matches n or fewer consecutive occurrences of pattern pat. (exactly n pat) matches exactly n consecutive occurrences of pat. (* pat) is shorthand for (atleast 0 pat), (+ pat) is shorthand for (atleast 1 pat), (? pat) is shorthand for (atmost 1 pat), which essentially makes pat optional. ("*", "?", and "+" are standard restricted regular expression functions.) The operators **, ++, and ?? are exactly like *, +, and ?, except while the latter prefer to match as many times as possible, the former prefer to match as few times as possible. The difference will be illustrated in greater detail in the section on parse extraction, section 4.6.15.

Notice that (+ pat) is the same as (seq pat (* pat)). Notice (? pat) is the same as (or pat empty).

7.3. The SEQ Pattern

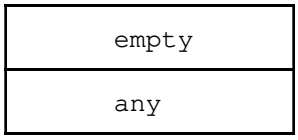
(seq pat1 pat2 pat3 ...)
[pat1 pat2 pat3 ....]

The sequence pattern matches any substring whose first part matches pat1, whose next part matches pat2, etc. [pat1 pat2 pat3 ...] is a shorthand syntax for (seq pat1 pat2 pat3 ...).

```
Examples:
                        aababbaaaabbbbabababaababbaabaaaabb
(seq a b b)                ^^^   ^^^         ^^^      ^^^


                        aababbaaaabbbbabababaababbaabaaaabb
(seq a (seq a b) b)           ^^^^                  ^^^^

It is easy to see that (seq a (seq a b) b) matches the same
substrings as (seq a a b b).

                        aababbaaaabbbbabababaababbaabaaaabb
(seq a b b a a a a)        ^^^^^^^
```

```
c   matches the lowercase c.

"+" matches the plus character.

'   matches the apostrophe character.

(from 48 to 57) matches any single digit.

(from "0" to "9") matches any single digit.

(class "0" "1" "2" "3" "4" "5" "6" "7" "8" "9") matches any digit

(notclass a b c) matches any letter except for "a", "b", and "c".

(from a to z) matches any lowercase letter of the alphabet.

'hi there'  matches the substring <hi there>.

""""   matches the double quote character.

(class Digits Letters)   matches all digits and letters, assuming
appropriate definitions for Digits and Letters.
```

7.2. EMPTY and ANY



The "empty" pattern matches all 0 length substrings; it usually appears in an "or" statement.

The "any" pattern matches any character. Since the TLex alphabet is all the characters from 0 to 255, any is the same as (from 0 to 255).

```
Examples:
                                 aababbaaaabbbbababaababbaabaaaabb
(seq a b empty any b)             ^^^^    ^^^^ ^^^^ ^^^^

                                 aababbaaaabbbbababaababbaabaaaabb
(seq a any any b)                 ^^^^  ^^^^   ^^^^ ^^^^     ^^^^
                                        ^^^^                  ^^^^
                                         ^^^^

                                 aababbaaaabbbbababaababbaabaaaabb
empty                           >< >< >< >< >< >< >< >< >< >< >< ><
                                 >< >< >< >< >< >< >< >< >< >< ><
                                  >< >< >< >< >< >< >< >< >< >< ><
```

7.1. The Primitive Patterns

a
23
(from 48 to 57)
'your sign here'
"your sign here"
(notclass b 34 "*")
(class a 23 "+")

The primitive patterns match a single character, a set of characters, or a string literal. Any single character, except for "(", ")", "[", "]", "{", "'", "`", "+", "*", "?", digits, carriage return, linefeed, tab, and space characters are patterns that match that single character. **Any** single character in single or double quotes matches that character.

A sequence of digits, in other words a number, matches the character with ASCII value equal to the number. Note that 0 matches ASCII character 0, while "0" matches ASCII character 48 (zero).

A string literal in single or double quotes, i.e. "your sign here", matches the text between the quotes.

(from num1 to num2) matches all the characters from num1 to num2 inclusive, where num1 and num2 are ASCII numbers or single non-special letters. It does not matter which of num1 or num2 is bigger.

(class a 23 "+") matches "a", the character with ASCII code 23, or "+". Use the class pattern to define a set of characters to be matched.
(notclass b 34 "*") matches any character except for the ones listed. The arguments to class or notclass can be any primitive pattern, or a definition for such a pattern.

```
Examples:

9   matches the tab character.

10  matches the newline character.

13  matches the carriage return character.
```

7. TLex Pattern Language Reference

The TLex pattern language is an extension of restricted regular expressions. Here is a description of the TLex language. For most of the examples we will take as basic characters “a” and “b”. The basic alphabet of TLex, however, is all the character values from 0 to 255. Multi-byte character sets pose no theoretical difficulty, but might require significant re-engineering of TLex to be efficiently handled.

The TLex pattern language is recursively defined. The basic unit is that of a pattern. Here we describe the different classes of patterns. 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”.

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

```
   pattern                                  input string
-----------------------      -----------------------------------
                              aababbaaaabbbbababaababbaabaaaabb
(seq a any any b)              ^^^^  ^^^^   ^^^^ ^^^^     ^^^^
                                      ^^^^                 ^^^^
                                       ^^^^
```

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.

6. Summary

TLex offers optimally fast matching, and nearly optimal parsing speed. At the same time it offers a pattern language which is more expressive than any previous tools using regular expressions, and close in expressiveness to slower tools that use a more powerful model of pattern matching. The automatic parse tree creation, and convenient access to the parse tree provided by the parse tree abstract data type vastly simplify the creation of pattern matching applications. Thus, while maintaining nearly optimal speed, TLex sets a new standard of expressiveness for pattern matching tools.

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.

5.3. 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, hockey, and 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.

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

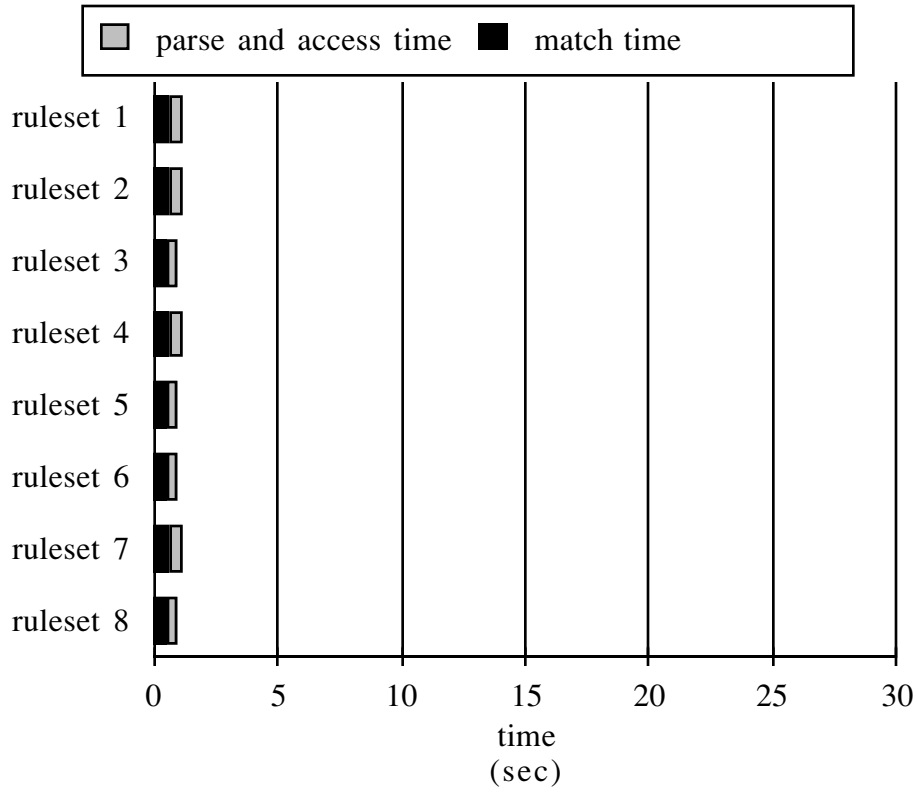


Figure 7: Fast-BBall Timing per Ruleset

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.

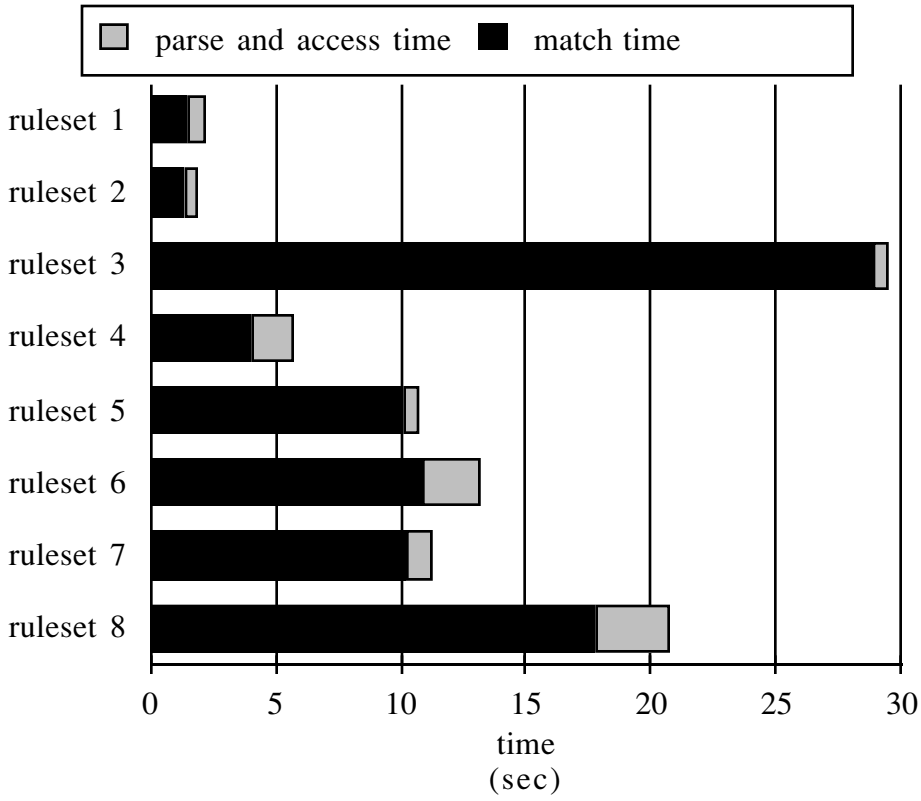


Figure 6: BBall Timing per Ruleset

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 much 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 and statistical tables. BBall was compiled and run on a Sun 4/390. Table 2 summarizes the performance of BBall on the input text.

Table 2.-- Behavior of BBall

	TLex Compile Time	**Match Time**	**Parse & Access Time**	**Total Time**	**Total Matches**	**Total States + Substates**
BBall	10 sec	85 sec	11 sec	96 sec	634	47,147
Fast-BBall	7450 sec (~ 2 hour)	4.4 sec	6.8 sec	11 sec	634	535,178

"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 6 and 7 break down the time spent on each ruleset:

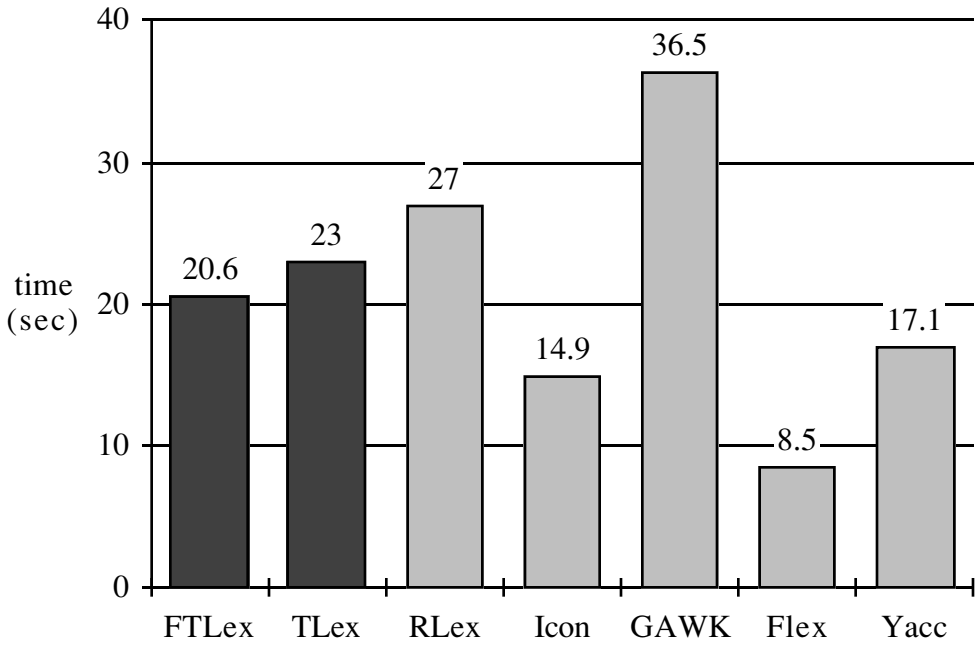


Figure 5: WC2 Performance

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 worse. 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.

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 too large (1700 lines) to include here, but we can quantify its size: it has 8 sets of rules, defines 98 patterns, and makes 55 pattern accesses, some in loops.

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, NewYacc's control structure is too primitive to handle the necessary manipulations. Flex can probably locate the

5. Experience with TLex

This section presents the performance of TLex and attempts to compare it to other pattern matching tools.

TLex has two parts: a compiler written in 11,000 lines of C++ code, and a runtime library written in 6,000 lines. We characterize its performance on two sample applications; we describe a number of applications that use TLex; and we discuss lessons learned from its real world use. 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.

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. (Technically, the pre-compiled form is that of a deterministic finite automata, while the slower form is that of a non-deterministic finite automata using lazy transition evaluation; see the thesis for details.) We call the program that uses TLex in this way "FTLex", for Fast TLex. 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. Figure 5 displays the results.

(seq (shorter (*-1 any)) (*-2 any)) to "12345":
  *-1 = "", *-2 = "12345"

(seq (shorter (*-1 any)) (shorter (*-2 any))) to "12345":
  *-1 = "", *-2 = ""

(seq (longer (shorter (*-1 any))) (*-2 any)) to "12345":
  *-1 = "12345", *-2 = ""

(seq (*-2 digit) (?-2 "+")) to "1234+":
  *-2 = "1234", ?-2 = "+"

(seq (*-2 digit) (??-2 "+")) to "1234+":
  *-2 = "1234", ??-2 = ""

(seq (++-2 digit) (?-2 "+")) to "1234+":
  ++-2 = "1", ?-2 = ""

(seq-2 (and-1 (seq a anystr)
              (seq anystr a))
       anystr)
    to "abbabbababbababb":
  seq-2 = "abbabbababbaba"
  and-1 = "abbabbababbaba"

(or-1 "abba" "abbabba") to "abbabba":
  or-1 = "abba"

(longer (or-1 "abba" "abbabba")) to "abbabba":
  or-1 = "abbabba"

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()
```

One can see how the automatic creation of a parse tree, and the parse tree abstract data type that provides convenient access to it, vastly simplifies the process of creating pattern matching applications.

4.1. Predicting the Parse

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

```
(seq (*-1 any) (*-2 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 *, +, ?, atmost, or atleast patterns, the parse prefers to loop through p, instead of leaving or skipping p. If p is one of the **, ++, 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 (longer p) and (shorter p) prefer to match the longest and shortest matches to p that allow the rest of the pattern to match successfully.

- (and ...) and (all ... 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 (seq p1 p2), the preferences of p1 take precedence over p2.

Examples:

(seq (*-1 any) (*-2 any)) to "12345":
*-1 = "12345", *-2 = ""

```
(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.+-1[0]"];
         numSS;
         numSS = numSS.adv(1))
        {
         if (numSS["?-1"].thru())
             sum += numSS["number-1"].tonum();
        }

    cout << "$" << sum << "\n";
   }
```

Figure 4: A Sample Rule

The operators +-1, ?-1, and number-1 are all instances of **named** operators. Any operator can be named 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 addition, naming essentially specifies the desired detail of the parse tree that TLex should create from a match. Knowing this speeds the creation of the parse tree and minimizes the space needed to store it.

The data type "ss" is an abstract data type used to provide access to a parse tree. 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 +-1 loop was repeated. 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.

4. Rules and Parse Trees

This section presents an overview of TLex rules and parse trees; a later section discusses each in greater detail.

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 that can access a parse tree describing the match in detail, a truly useful tool results.

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

```
pat1
==>
{ C or C++ expression }
```

"pat1" is a pattern written in the TLex pattern language. 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 later; in this section we give an example of their use.

As an example, the rule and associated definitions in figure 4 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".

```
(upto "foo") on the string "I ate foo or foo" matches:
        "I ate foo or foo"
         ^^^^^^ ^^^^^^ ^^
          ^^^^^  ^^^^^  ^
           ^^^^   ^^^^  ><
            ^^^    ^^^
             ^^     ^^
              ^      ^
              ><     ><
```

• To match a word one can use the following definitions:

```
(define alphanumeric (or (from "0" to "9")
                         (from A to Z)
                         (from a to z))
(define nonAN (notclass alphanumeric))
(define word (csrp nonAN (+ alphanumeric) nonAN))

When given the input "Hi there. My name is steve! Wow"
word can match:          ^^^^^  ^^ ^^^^ ^^ ^^^^^
```

There seems to be a problem with this example. Both "Hi" and "Wow" are not matched! "Hi" isn't matched because there is no nonAN to the left of it. "Wow" isn't matched because there is no nonAN to the right of it. This problem often arises when one wants to make sure that something does NOT appear on the right or left. It is easy to forget about the start or the end of the text. We can redefine word as follows to solve these problems:

```
(define word2
        (csrpn alphanumeric (+ alphanumeric) alphanumeric))

When given the input "Hi there. My name is steve! Wow"
word2 can match:      ^^ ^^^^^  ^^ ^^^^ ^^ ^^^^^  ^^^
```

• Assume FullFloat matches numbers like "123.456" and HalfFloat matches numbers like "789.". To define a pattern which matches floats of either type, one might consider (or FullFloat HalfFloat). Unfortunately, on the input "456.789 times" this **might** match "456." as a HalfFloat. We really want to match a FullFloat or a HalfFloat, but never a HalfFloat when we can match a FullFloat. The following ctrpn pattern accomplishes this:

```
(or FullFloat (ctrpn HalfFloat FullFloat)).
```

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 if there is more than 1 recursive call to the pattern being defined. 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 implement when a pattern calls itself two or more times.

3.3. 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:

```
   pattern                                  input string
------------------------      ------------------------------------
                               aababbaaaabbbbababaababbaabaaaabb
(seq a any any b)               ^^^^  ^^^^   ^^^^ ^^^^     ^^^^
                                       ^^^^                 ^^^^
                                        ^^^^
```

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.

• (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))))
```

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

`(shortest pat)`
 	
`(longest pat)`
	when pat matches several substrings starting at the same position, matches only the shortest or longest such substring.

3.2. Limitations of 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, and then substituting a pattern that matches nothing. To understand the limitations of bounded recursion, consider the recursive definition of a set of matching braces:

```
(define MP  (or  empty
                 (seq '{' MP '}')))
```

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

```
(define MP
   (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

(or pat1 pat2 pat3 ...)	matches what matches pat1, pat2, pat3, ...; prefers to match the leftmost pattern.
(and pat1 pat2 pat3 ...)	matches what simultaneously matches pat1 and pat2 and pat3...; prefers longer matches.
(all pat1 minus pat2 pat3 ...)	matches anything that matches pat1 but does not match pat2, pat3, ...; prefers longer matches.
(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.
(csrpn pat patRight) (csrpn 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.

a	matches the letter “a”.
23	matches character with ascii code 23.
(from A to Z)	matches any character between A and Z.
‘your sign here’	matches the string in quotes.
“your sign here”	matches the string in quotes.
(notclass b 34 “*”)	matches any character except the listed ones.
(class a 23 “+”)	matches any listed character.
empty	matches all 0 length substrings.
any	matches any character.
(seq pat1 pat2 pat3 ...) [pat1 pat2 pat3 ...]	matches a match to pat1 followed by a match to pat2, pat3, ...
(* pat) (** pat)	matches 0 or more consecutive occurrences of matches to pat; “*” prefers to match as many times as possible, “**” as few times as possible.
(+ pat) (++ pat)	matches 1 or more consecutive occurrences of matches to pat; “+” prefers to match as many times as possible, “++” as few times as possible.
(? pat) (?? pat)	optionally matches pat; “?” prefers to match, “??” prefers not to.
(atleast n pat) (exactly n pat) (atmost n pat)	matches at least n, exactly n, or at most n consecutive occurrences of matches to pat; prefers to match as many as possible.

3. TLex Patterns

In this section we introduce and summarize the TLex pattern langauge; a later section discusses each operator in detail.

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 below). TLex uses regular expressions in a way that provides unlimited 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 Icon language. A valuable feature missing in TLex, but present in Icon, is **immediate assignment**; the latter allows 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. Icon allows dynamic pattern building, however. Although less powerful than Icon, TLex can be significantly faster and easier to use.

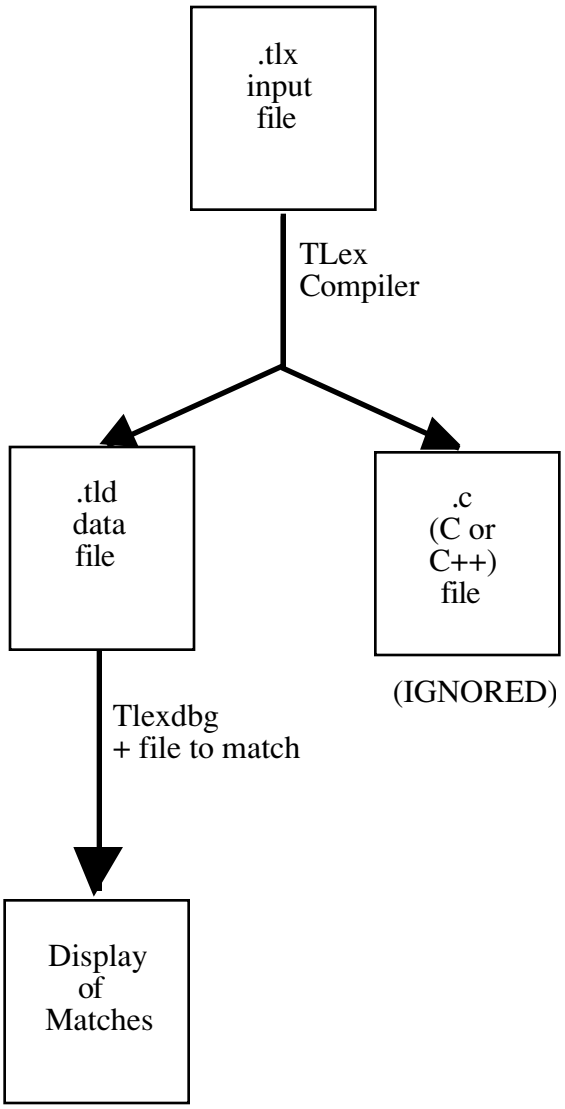
3.1. Pattern Operators

The TLex pattern language is an extension of restricted regular expressions. Here we give a succinct description of the TLex pattern language. 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 1.-- TLex Pattern Language Summary

PATTERN	DESCRIPTION

means the .c file produced by the TLex compiler does not have to be compiled. Here is a diagram of the debugging process:



Because TLex is a code generator, using it can be complex. Figure 3 gives an overview of the process:

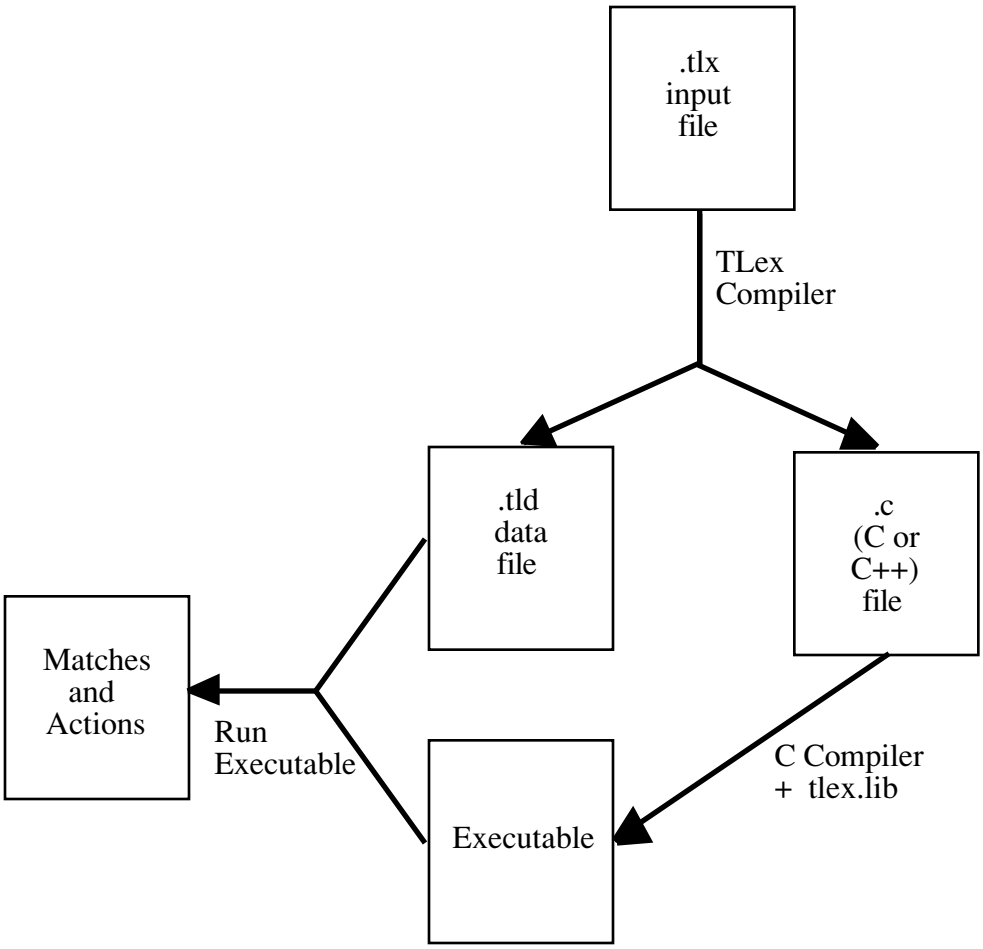


Figure 3: TLex Overview

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. Generating separate files for the patterns and actions simplifies debugging (see below) and permits patterns to be loaded only when in use.

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.

For debugging patterns, the program TLexdbg ("tlex debug") accepts a .tld file and an input file as parameters and matches the patterns against the input. TLexdbg reports which rules match, where the rules match, and the parse tree extracted from each match. TLexdbg does not call the rule actions, which

```
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 2: 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.

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 or 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 schematic view of a typical TLex input file:

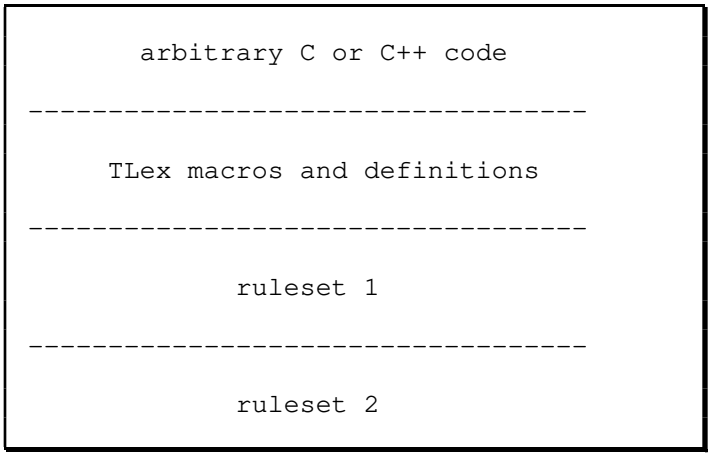


Figure 1: Schematic TLex Input File

Figure 2 shows a very simple TLex file that counts the number of lines in a file.

1. Introduction

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. TLex, which stands for Tyrannosaurus Lex, is another tool for this purpose.

Consider searching a day's output of wire-service stories for specific pieces of information, such as financial information, sports scores and statistics, or recipes. Such an application poses requirements not fulfilled by existing pattern matching 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 $O(n^2)$ time or space (or worse) is unacceptable.

- 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.

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 action.

TLex v.68
User's Manual

Columbia University Technical Report
CUCS-037-90

Steven M. Kearns

Columbia University,
450 Computer Science,
500 W. 120th St.
New York, NY 10027, U.S.A.

HLH Associates
1 W. 85th St. Apt. 3D
New York, NY 10024

Abstract:

The TLex Language, a regular expression-based language for finding and extracting information from text, is described. TLex features a pattern language that includes intersection, complement, and context sensitive operators. In addition, TLex automatically creates a parse tree from a successful match and offers convenient functions for manipulating it.