LexGen – A Lexical Scanner Generator

Contents

1 Introduction

LexGen is a Forth program that converts a specification of keywords and regular expressions into a set of data tables that can be used in a lexical scanner or pattern matcher implemented as a Deterministic Finite State automaton (DFA). This note does not describe how LexGen works but how to use it. To understand the software see the algorithms for converting regular expressions into a DFA in the Red Dragon book, reference [1], as described in section 3.9, p134 to 146. The data structures generated are shown in fig 3.47 in the book. Another useful reference is [2] which available as a free download.

LexGen is written in ANS Forth to generate data for use in Forth applications, although it may easily be amended to generate data files for any other language. A suitable Forth system for running LexGen is GForth but it should work with any system compliant with the ANS Forth standard that also includes the Core Extension, Double-Number, File-Access, Programming-Tools, Search-Order and String word sets. Descriptions in this document assume the use of GForth. As written it also requires a case-insensitive Forth system.

LexGen is typically used to provide a lexical scanner to use with parsers generated by such tools as Grace [3] or Gray [4], but does have wider uses.

The latest version is 2.3, see section 11.

2 Installation and test

The following assumes the use of GForth.

a. Download a release from GitHub

b. Create a working directory called `lexgen` somewhere, for example under GForth itself.

c. Unzip the downloaded file into the `lexgen` directory.

d. Start Gforth with lexgen as the working directory.

e. Type (suppress warnings to avoid confusion caused by irritating redefinition messages):

```
warnings off
fpath path= ./|./../
s" test/lexgentest.fth" included
```

which should result in the following output from GForth

```
Gforth 0.7.9_20170308, Copyright (C) 1995-2016 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `help' for basic help

warnings off  ok
fpath path= ./|./../  ok
s" test/runtest.fth" included

Loading LexGen 2.3.0 ...


LexGen 2.3.0 loaded successfully. <0>
Decorating the syntax tree...
```

```
Building the transition table...
.....................................................................
.....................................................................
.....................................................
Number of FSM states: 225
Generating output data...
Saving data to file...
Run completed

.....................................................................
.....................................................

LexGen state transition table & file scanner tested.

...
LexGen source scanner tested.
```

The important messages indicating a working system are:

```
Run completed
LexGen state transition table & file scanner tested.
LexGen source scanner tested.
```

With other Forth systems the file path may need to be changed in the file `lexgentest.fth`. when the file `teststt.fth` is included (in two places). Other systems will also generate different warning messages.

A successful test will have generated a file called `teststt.fth` in the test directory.

3 Portability

The LexGen test program has been run successfully using GForth, VFX Forth, SwiftForth, Win32Forth , iForth and BigForth under 32 bit Windows 7 and XP. It has not been tested on 64 bit systems and other operating systems; feedback on use of LexGen on such systems would be appreciated.

4 Files provided

See GitHub for the set of downloaded files

5 The User Interface

5.1 Introduction

In version 2.0 onwards the user interface has been greatly simplified by the addition of an interface that permits specification of patterns as simple regular expressions in a conventional format instead of the reverse Polish format used in previous versions. A description of the old, now deprecated, interface is in Appendix A.

The set of patterns/symbols/strings that will be recognised by a target scanner are specified in one or more files that must be included following the loading of LexGen. The data specified for LexGen falls into two types:

- Definition of constants (also called tokens) to be returned by the target scanner when a particular keyword or pattern has been recognised. Typically this is in a separate file, e.g. see the provided file

`testtokens.fth`, as these constants will also be needed by accompanying software such as a parser generated by Grace or Gray.

- Specification of the keywords or patterns to be recognised by the target scanner; this is the subject of the next sub-section.

5.2 Input file format

Typically the specification of patterns to be recognised by the generate state transition tables and scanner will be contained in a single file containing the following items (see the file `testlexinput.fth` for examples):

1. The name of the output file, and (optionally) the scanner file to be used and whether formatting of the output data is required.

2. The maximum character value to be used. For ASCII this is 127, it is recommended that this be kept below 256 otherwise the generated tables may be huge.

3. Case sensitivity. This may be changed as required throughout the file.

4. Named regular expressions.

5. Association of constant values with the patterns to be recognised i.e. the value to be returned by the target scanner when a particular pattern has been recognised.

6. A call to initiate the generation of the tables by using the word `lexgen`.

This order is not mandatory, for example items 3, 4 and 5 may be intermingled as desired as long as named items are defined before they are used. Item 2 must be before 3, 4 and 5. Item 6 must be last.

The definitions of constants or tokens referred to in item 5 are recommended to be in a separate file. This is because LexGen is typically used in conjunction with another tool such as Grace which also uses these tokens and it is vital that the token values are consistent between the output from the tools. This is most easily achieved by each tool using the same token file.

5.3 Regular expression format

A conventional regular expression format is used for specification of patterns to be recognised. The following features are supported (the term ‘item’ is used to represent a character, a sequence of characters or a previously defined regular expression):

1. Concatenation - consecutive items are automatically concatenated together. e.g. the pattern `abc` represents concatenated characters `'a'`, `'b'`, and `'c'`.

2. Alternation - specified by inserting character `'|'` between items to represent alternatives, e.g. `a|b` is either character `'a'` or character `'b'`.

3. Zero or more repetitions - specified by appending an `'*'` character to the item to be repeated, e.g. `a*b` means zero or more `'a'` characters followed by a single `'b'` character.

4. One or more repetitions - specified by appending a `'+'` character to the item to be repeated e.g. `a+b` means one or more `'a'` characters before a single `'b'` character.

5. Optional item - specified by appending a `'?'` character to the optional item, e.g. `ab?c` means that the `'b'` character is optional.

6. Grouping - items may be grouped together by enclosing them in parentheses e.g. `(abc)` groups the concatenated characters `'a'`, `'b'` and `'c'`. `(abc)?` means that the group is optional.

7. Character classes - are represented using square brackets. e.g. `[abc]` represents a single character that may be `'a'`, `'b'` or `'c'`. The class may include both single characters and ranges e.g. `[_a-z0-9]` means a single character that may be an underscore or lower case alphabetic character or a digit.

8. Negated character classes - are represented using square brackets with the first enclosed character being `'^'`. These are like character classes but represent characters that are excluded from the pattern e.g. `[^abc]` means any character that is not an `'a'`, `'b'` or `'c'`.

9. Metacharacters that are specified by preceding them with a `'\'` character[1]. e.g. `\*` represents a `'*'` character. This is to avoid `'*'` being recognised as a repetition operator. The set of metacharacters is: `\[ \] \( \) \| \* \+ \- \? \# \{ \} and \\`. These must be used both within regular expressions and character classes.

10. Escaped characters are also preceded with a `'\'` character[2], these are mainly to represent control characters. e.g. `\s` represents a space, `\n` a line feed, `\r` a carriage return, `\t` a tab, `\0` a zero. In addition `\cx` represents a control character whose value is given by (`x mod 32`) e.g. `\cH` represents control H which has the value 8. `\d{...}` is the decimal equivalent of the integer string between the braces e.g. `\d{10}` is the same as `\n`. Similarly `\x{...}` converts the enclosed string as a hexadecimal number e.g. `\x{0A}` is also the same as `\n`.

11. Previously defined regular expressions which have been defined by the word `regex` (see the glossary below). Such a word must be enclosed in braces when it is used in another regular expression e.g. if we had defined foo as:

    ```
    regex foo ab*c
    ```

    then we would use `foo` by writing e.g. `x{foo}yz` which is equivalent to the regular expression `xab*cyz`

5.4 Example input file

An example file can be seen in the file `testlexinput.fth` which specifies a subset of C keywords, operators, identifiers and numbers; as well as other items purely for test purposes. This file contains constants (tokens) of the form `"begin"` which are defined in a separate file called `testtokens.fth`.

5.5 Glossary of User words

(These are grouped by function type in approximate order of use rather than alphabetical order).

`setOutputFile ( caddr u -- )` sets the output file name for the generated state transition tables or lexical scanner.

`setScannerFile (caddr u -- )` sets the name of the scanner file to be appended to the generated state transition tables if `no-scanner` has <u>not</u> been used.

`no-scanner ( -- )` stipulates that a scanner file must not be appended to the generated state transition tables. This overrides any file name set by `setScannerFile`.

1 A `\` followed by any other character e.g. `\q` is currently treated as concatenated characters `'\'` and `'q'` (but don't rely on this, it may change in future versions if more metacharacters or escaped characters are provided - better to use `\\q`).

`formatted` `( -- )` stipulates that the state transition tables are saved in tabular format. By default they are unformatted.

`setMaxChar` `( u -- )` Specifies the maximum character value to be used. The default is 127. If used it must occur before definition of any patterns.

`case-insensitive` `( -- )` Makes the following symbols, characters and character classes case insensitive, e.g program = PROGRAM = PrOgRaM.

`case-sensitive` `( -- )` Makes the following symbols, characters and character classes case sensitive i.e. program <> PROGRAM. This is the default state. Note that `case-sensitive` and `case-insensitive` may be used as often as required for different symbols.

`regex` `( "<spaces>name<spaces>regexp<spaces>eol" -- )` Defines a name and assigns the following regular expression to the name. The regular expression is terminated by the end of the line. e.g. usage is:

```
regex foo ab*c
```

Following definition of the name, `regex` removes leading and trailing spaces from the following regular expression. When `name` is used in another regular expression it must be enclosed in braces e.g. `{foo}`. Execution of name leaves the syntax tree corresponding to the regular expression, or a clone of it, on the stack. Comments are not permitted in the line.

`==>` `( tree? n "<spaces>regexp<spaces>eol" -- tree2 )` Associates the value n with the following regular expression. The regular expression is terminated by the end of the line. Leading and trailing spaces are removed from the regular expression. The input `tree` is marked as optional as it will not be present when ==> is used for the first time. As LexGen is using the stack while processing the input file, users must be careful when using the stack for their own purposes. Comments are not permitted in the line.

`lexgen` `( tree -- )` Initiates the whole process of generating scanner data and saving the tables in the specified output file. This must follow all regular expression definitions and is usually used at the end of the LexGen input file as shown in `lexgentest.fth`.

`token` `( "<spaces>name" -- )` To define token values as sequential constants, normally starting at 1. `name` is created as a constant which leaves its value on the stack when executed. For example in a file tokens.fth the following may be written:

```
token  "begin"
token  "end"
etc
```

would define `"begin"` as 1, `"end"` as 2 etc. The word `token` is included for user convenience, use of `token` is not mandatory, redefine it or use a different word if required. (It is strongly recommended that the lowest token value is 1 as anything higher will have an effect on set sizes used by LexGen.)

`token>` `( "<spaces>name" -- )` Exactly the same as `token`. It has been introduced to facilitate use of LexGen with Grace - see the Grace user guide for an explanation of its use.

5.6 Use of regular expressions

Named regular expressions

Use `regex` to name a regular expression for use in other regular expressions e.g.

```
regex letter       [a-zA-Z]
regex digit        [0-9]
regex identifier   {letter}({letter}|{digit}|_)*
```

When a named regular expression is used as above, it must be enclosed in braces `{...}` to distinguish it from a sequence of characters, for example using `letter` instead of `{letter}` would insert the separate characters `'l' 'e' 't' 't' 'e' 'r'` into the regular expression. The regular expression will be terminated at the end of the line. A future enhancement may permit free-form multi-line layout.

White space

Removing leading white space from the regular expression permits neat layout for readability. Removing trailing spaces prevents a space being accidentally included at the end of the regular expression. Any spaces within the regular expression are left in place and form part of the regular expression. If a leading or trailing space is required in the regular expression either use `\s` or enclose a space in parentheses.

Associating constants with regular expressions

When a lexical scanner is used and a pattern representing a keyword is recognised, the scanner returns a constant (i.e. token) to the calling program, therefore there has to be a way of associating tokens with a regular expression representing the pattern. This is done with the `==>` operator.

The rest of the line after `==>` is assumed to be a regular expression, therefore comments are not permitted. An example of using a named regular expression is:

```
"id"      ==>       {identifier}
```

where `"id"` is a previously declared token. Note that use of quotes round the constant is not mandatory - as in Forth any name can be used, using quotes is merely the author's convention.

Inclusion of comments

As `regex` and `==>` use the rest of the current line as the regular expression, comments are not permitted at the end of the line e.g. given this association:

```
"ws"      ==>       [\s\t\n\r]+         \ White space
```

the intended comment and spaces before it would be included in the regular expression. If there is a demand from users this restriction may be removed in a future version.

5.7 Ambiguity resolution

If there is some ambiguity in the regular expression definitions then the first defined regular expression will be recognised instead of the later ones. For example given the following definitions:

```
regex digit        [0-9]
regex allButSpace  [^\0-\s]

regex integer      {digit}+
regex forth-word   {allButSpace}+
```

```
"integer"    ==>   {integer}
"forth-word" ==>   {forth-word}
```

the character string 1234 could either be recognised as an `integer` or as a `forth-word`. However, as the definition of regular expression `integer` occurs first, LexGen will recognise the string 1234 as an `integer`. Such ambiguity resolution permits less rigorous definition of regular expressions and results in fewer states in the final DFA and hence in smaller output tables for the application scanner.

6 Running LexGen

LexGen is run by simply including `lexgenloader.fth` followed by the user written tokens file and input file (see section 5.2) containing the patterns to be recognised by the generated state transition tables and scanner. An example of a typical LexGen run is given near the start of the test program `lexgentest.fth`.

7 Output from LexGen

LexGen generates state transition tables for a lexical scanner. These can be generated with or without one of the scanners provided in the LexGen download (see section 8.1) by the use or not of the word `no-scanner` in the LexGen input files. By default the supplied file scanner is used. Alternatively if a user requires a different scanner e.g. the supplied source scanner or one written by the user, this may be incorporated in the output file by the use of the word `setScannerFile`. By default the tables are unformatted - this may be overriden by the use of the word `formatted` to give data in a tabular format.

The data is saved as offsets into the tables, if the provided scanner is used, these offsets are converted into absolute addresses before use.

8 The scanner for target applications

8.1 The scanners included in the download

As already noted, two target scanners are provided in the files `filescanner.fth` and `sourcescanner.fth` one of which may be appended to the state transition tables. These scanners are not likely to be suitable for all target applications but can serve as models for alternatives. The scanners are line oriented, i.e. patterns or symbols will not be recognised across line boundaries.

The two scanners are provided for two situations:

- where the target source being scanned is in a separate file not containing executable Forth when `filescanner.fth` should be used.
- where the target source is incorporated into a Forth source code file when `sourcescanner.fth` should be included.

Both scanners are used in the test program. The different way they are used is shown in section 8.3.

8.2 Glossary of scanner words

The following words are provided to use the scanner:

`open-source ( caddr u -- )` or `( x x -- )` is defined differently in the two scanners to enable code using the target scanner to be largely independent of the two scanners.

a. In the file scanner `open-source` opens the file named in the string specified by `(caddr u)` and initiates the scanner ready for scanning the file.

b. in the source scanner `open-source` simply drops the top two stack items.

`next-token ( -- caddr u tok )` runs the DFA to recognise the next pattern in the file opened by `open-source` and returns both the pattern as `(caddr u)` and its token `tok` which was specified in the original tokens file. If no valid symbol has been recognised, the value 0 is returned with `(caddr u)` giving the unrecognised character(s). If the end of the file has been reached it returns `( x 0 tok )` where `tok` is the end of file token, `eof-tok`.

`close-source ( -- )` Closes the source file

`eof-tok ( -- n )` This is a Forth `VALUE` preset to -1. The value of `eof-tok` may be changed using the Forth word `TO` e.g. `99 to eof-tok`

In addition the following may prove useful:

`next-line ( -- f )` Abandons the current line and fetches the next line from the source file. Returns true if successful, false indicates end of file. `next-line` could be useful when a comment has been recognised in the source file.

`scan-file` which is a test word that has different definitions in the two scanner files. Following loading of the state transition tables and the scanner, `scan-file` will scan text and list all the recognised symbols and their token values. This word can be provided as a visual check that the state transition tables are correct.

In the file scanner:

`scan-file ( caddr u -- )` scans the text file named `(caddr u ).`

In the source scanner:

`scan-file ( -- )` scans the text following it in the source file until either the end of the file or some user defined mechanism terminates scanning.

8.3 Testing LexGen output

The state transition table generated by LexGen can be visually tested by using the word `scan-file` that is defined in each of the scanners. This can be demonstrated by the following, assuming that the test program has been run successfully (see section 2).

The test program will have generated a file called `testst.fth` in the lexgen/test directory. Type the following (assuming the working directory is `lexgen`):

```
s" test/testst.fth" included
s" src/filescanner.fth" included
s" test/test.txt" scan-file
```

where `test.txt` contains a list of symbols that should be recognised by the scanner. This should result in the symbols (including white space) and their token values being displayed until the end of the file is reached.

To carry out the same test with the source scanner type:

```
s" test/testst.fth" included
s" src/sourcescanner.fth" included
s" test/test2.txt" included
```

Which should result in the same display as before. Note that the first word in `test2.txt` is `scan-file` which scans the rest of `test2.txt`.

9 Possible problems

a. The transition table can be huge, depending on the size of the character set and the number of target symbols to be recognised. There is a chance that the Forth system does not provide sufficient dataspace for this. The solution is, if possible, to increase the size of dataspace available or to use a Forth system with more dataspace.

b. The syntax tree can become very deep which can be a problem as it is traversed recursively. This could lead to return stack overflow if the Forth system used has too small a return stack. If so that is likely to crash the system. The solution is, if possible, to increase the size of the return stack or to use a Forth system with a bigger return stack.

10 Future possible enhancements

a. Free form multi-line layout including comments

b. Generation of more compact, probably slower, state transition tables.

c. Addition of a state minimisation algorithm as described in Aho, Sethi and Ullman.

11 Version history

Version 2.3 10 January 2018. Transferred to GitHub, no changes in functionality.

Version 2.2 21 May 2011. Definition of `token>` added. File `sourcescanner.fth` added to the download. File scanner.fth renamed filescanner.fth. Test program extended.

Version 2.1 31 August 2010. User interface simplified by removal of `:regex` and change in the semantics of `regex` and `==>`.

Version 2.0 20 August 2010. New user interface permits regular expressions in conventional format. Addition of options to generate unformatted output files and append a scanner to the output file.

Version 1.2 10 April 2008, new user words: `token 'lit'` and `yields`. Minor documentation corrections

Version 1.1 30 Dec 2006, improved ambiguity resolution.
Version 1.0 19 Dec 2006, initial release

12 ANS Forth requirements

LexGen requires a case-insensitive ANS Forth system and the following words from the ANS Forth standard:

Core word set: `! #> #S ' (* + +! +LOOP , - . ." / /MOD 0< 0= 1+ 1- 2! 2* 2/ 2@ 2DROP 2DUP : ; < <# = > >BODY >IN >NUMBER >R ?DUP @ ABORT ABORT" ABS ALIGN ALIGNED ALLOT AND BASE BEGIN BL C! C, C@ CELL+ CELLS CHAR CHAR+ CHARS CONSTANT COUNT CR CREATE DECIMAL DO DOES> DROP DUP ELSE EMIT ENVIRONMENT? EVALUATE EXECUTE EXIT FIND HERE I IF IMMEDIATE INVERT J KEY LEAVE LOOP LSHIFT MAX MIN MOD MOVE NEGATE OR OVER POSTPONE QUIT R> R@ RECURSE REPEAT ROT RSHIFT S" SIGN SOURCE SPACE SPACES STATE SWAP THEN TYPE U< UNLOOP UNTIL VARIABLE WHILE WORD XOR [['] [CHAR]]`

Core extension word set: `.( .R 0<> 0> 2>R 2R> 2R@ :NONAME <> ?DO CASE COMPILE, ERASE FALSE HEX MARKER NIP PAD PARSE REFILL TO TRUE TUCK U> VALUE WITHIN \`

Double-number word set: `2VARIABLE`

File-access word set: `CLOSE-FILE CREATE-FILE FILE-SIZE INCLUDED OPEN-FILE R/O READ-FILE READ-LINE S" W/O WRITE-FILE WRITE-LINE`

Memory allocation word set: `ALLOCATE FREE`

Programming-tools word set: `.S`

Programming-tools extension word set: `[ELSE] [IF] [THEN]`

Search-order word set: `GET-CURRENT GET-ORDER SEARCH-WORDLIST SET-CURRENT SET_ORDER WORDLIST`

Search-order extension word set: `PREVIOUS`

String word set: `-TRAILING /STRING CMOVE COMPARE`

13 References

1. "Compilers Principles, Techniques and Tool", Aho, Sethi and Ullman
2. "Basics of Compiler Design", Torben Mogensen, www.diku.dk/~torbenm/Basics, free download.
3. Grace from www.qlikz.org/forth/grace
4. Gray by Anton Ertl from www.complang.tuwien.ac.at/forth/gray5.zip

Appendix A - The original user interface

A1 Specification of patterns to be recognised

This appendix describes the original user interface that was suppplied as the main interface up to version 1.2. It has now been superseded but is still available, therefore it has been moved into this appendix. However its use is deprecated and it will be removed in a future release.

A2 Layout of definition file

Symbols and regular expressions to be recognised are specified in the file lexinput.fth. Another filename can be used but lexgen.fth will need to be amended accordingly. It is best to use the example lexinput.fth as a template, the following examples are taken from this file. The outline structure of this file is:

1. Define the output file name and maximum character value to be used in the scanner. Without these defaults will be used i.e. lextables.fth and 127 respectively.
2. Define character sets and classes for use in regular expressions e.g.

```
char a char z [..]      \ defines a character set
char A char Z [..+]     \ adds a range of characters to the set
charClass letter        \ defines a character class called letter
```

3. Define regular expressions e.g. to define an identifier (the notation is explained in the next section)

```
letter letter digit <|> <*> <.> regexp identifier
```

   (note that character sets, classes and regular expressions may be intermingled as desired but must be complete before the next part).

4. Specify the symbols and regular expressions between the bracketing words `begin-symbols` and `end-symbols` and the tokens to be returned by the scanner when the pattern has been recognised by the scanner e.g.

```
begin-symbols
    "void"  symbol void
    "<<"    symbol <<
    "id"    denotes identifier
end-symbols syntaxtree
```

   which says that the scanner should return the token `"void"` when it recognised the word `void` in its input text stream. Tokens will have been defined in file tokens.fth. Use of the file tokens.fth is not essential but advisable so that the same definitions can be used in both LexGen and the scanner.

5. Run LexGen with:

```
syntaxtree lexgen
```

A3 Regular expressions

As indicated above the usual regular expression notation is not used in LexGen. Instead a forth-like reverse Polish notation is used with the following operators

LexGen	Replaces	Meaning
<.>	(not used)	Concatenation
<|>	|	Alternatives
<*>	*	Zero or more occurrences
<+>	+	One or more occurrence
<?>	?	Optional (zero or one)
'char' a	a	Character

The usual regular expression operators have been replaced by <*> etc to distinguish them from standard Forth words. Examples of their use are:

Usual notation	LexGen notation
ab	'char' a 'char' b <.>
ab*	'char' a 'char' b <*> <.>
(ab)*	'char' a 'char' b <.> <*>
a|b	'char' a 'char' b <|>
a?|(ab)+	'char' a <?> 'char' a 'char' b <.> <+> <|>

Character sets can be created by using the words `[new] [..] [..+] [+] [-]` and `charClass` which are described in the glossary e.g.

```
char a char z [..] char _ [+] charClass idletter
```

Note the difference between char and 'char', char is a standard Forth word that leaves a character value on the stack and is useful in LexGen for specification of character sets; whereas 'char' is a LexGen word that creates a leaf node representing a character value for a syntax tree and leaves the address of that node on the stack for use in regular expressions. Confusing the two will likely lead to a system crash.

A4 Ambiguity resolution

If there is some ambiguity in the regular expression definitions then the first defined regular expression will be recognised instead of the later ones. For example given the following definitions:

```
char 0 char 9 [..] charClass digit
33 127 [..] charClass allButSpace

digit <+>       regexp integer
allButSpace <+> regexp forth-word

"integer"    denotes integer
"forth-word" denotes forth-word
```

the character string 1234 could either be recognised as an `integer` or as a `forth-word`. However, as the definition of regular expression `integer` occurs first, LexGen will recognise the string 1234 as an `integer`. Such ambiguity resolution permits less rigorous definition of regular expressions and results in fewer states in the final DFA and hence in smaller output tables for the application scanner.

A5 Glossary of user words

(These are grouped by function type in approximate order of use rather than alphabetical order).

`setOutputFile` `( caddr u -- )` set the output file name

`setMaxChar` `( u -- )` Specifies the maximum character value to be used. The default is 127. If used it must occur before definition of any character sets

`[new]` `( c -- set )` creates a new character set containing c

`[..]` `( c1 c2 -- set )` creates a new character set including the range c1 to c2

`[..+]` `( set c1 c2 -- set' )` adds character range c1 to c2 to an existing set

`[+]` `( set c -- set' )` adds character c to set

`[-]` `( set c -- set' )` removes character c from set

`charClass ( set "<spaces>name" -- )` creates a character class called `name`.
`name` execution: `( -- tree )` creates a 1 node tree for the set.

`'char' ( "<spaces>name" -- tree )` creates a 1 node tree for the first character in `name` (the rest of `name` is discarded as for `char`)

`'lit' ( u --- tree )` creates a 1 node tree for the value u, u must be in the range 0 to the value set by `setMaxChar` (`'lit'` added in version 1.2 of LexGen for a non-character based application).

`<.> ( tree1 tree2 -- tree3 )` concatenates two syntax trees into one

`<|> ( tree1 tree2 -- tree3 )` or's two syntax trees into one

`<*> ( tree1 -- tree2 )` 0 or more repetitions of tree1

`<+> ( tree1 -- tree2 )` 1 or more repetitions of tree1

`<?> ( tree1 -- tree2 )` 0 or 1 occurrence of tree1

`regexp ( tree "<spaces>name" -- )` creates a name for a regular expression.
`name` execution `( -- tree )`, `name` is used in other regular expressions and by `denotes`.

`case-insensitive ( -- )` Makes the following symbols case insensitive,
e.g program = PROGRAM = PrOgRaM. This applies to symbols only, not character sets or single characters. Note that case-sensitive and case-insensitive may be used as often as required for different symbols.

`case-sensitive ( -- )` Makes the following symbols case sensitive i.e.
program <> PROGRAM. This is the default state. Note that case-sensitive and case-insensitive may be used as often as required for different symbols.

`begin-symbols ( -- )` starts the association list of symbols and regular expressions with tokens

`symbol ( [tree1] token "<spaces>name" -- tree )` constructs a syntax tree for `name`, token is the value to be returned by the scanner when `name` is recognised (see the note below). Use of `symbol` is shorthand for concatenation of characters e.g.

```
123 symbol xyz
```

is exactly equivalent to

```
'char' x 'char' y <.> 'char' z <.> regexp xyz
123 denotes xyz
```

`denotes ( [tree1] token "<spaces>name" -- tree )` Associates a token with the regular expression called `name` and incorporates it into the overall syntax tree (see the note below).

`yields ( [tree1] token tree2 -- tree )` Associates a token with tree2 and incorporates tree2 into the overall syntax tree (see the note below). (`yields` added in version 1.2 of LexGen for a non-character based application)

Note that the first use of `symbol, denotes` or `yields` does not require the syntax tree `tree1` to be on the stack. Also `symbol, denotes and yields` should only be used between `begin-symbols` and `end-symbols.`

`end-symbols` `( tree "<spaces>name" -- )` ends the list of symbols and regular expressions and creates a `name` that, when executed, leaves tree on the stack

`lexgen` `( tree -- )` Starts the whole process of generating scanner data.

`token` `( "<spaces>name" -- )` To define token values sequentially starting with 1. `name` is created as a constant which leaves its value on the stack when executed. `token` is included for user convenience, use of `token` is not mandatory, redefine it if required. For example in a file tokens.fth the following may be written:

```
token  "begin"
token  "end"
etc
```

would define `"begin"` as 1, `"end"` as 2 etc.