LexGen – A Lexical Scanner Generator

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 algorithms used see the algorithms for converting regular expressions into a DFA in the Red Dragon book on compilers ('Compilers, Principles Techniques and Tools' by Aho, Sethi and Ullman, 1986) as described in section 3.9, p134 to 146. The data structures generated are shown in fig 3.47 in the book.

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 to use for 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 wordlists. Descriptions assume the use of GForth. As written it also requires a case-insensitive Forth system.

2 Installation and files provided

Unzip the file lexgen.zip into a directory called lexgen in the gforth directory. Files may be placed in any directory you wish but then the file paths used in the file lexgen.fth will need to be amended. The files provided are:

- lexgen.fth which loads the rest of the LexGen files which are:
  - ansify.fth provides definitions of some common but non-standard words
  - mini_oof.fth provides simple object oriented facilities (by Bernd Paysan)
  - extended_mini_oof.fth extends mini-oof
  - sets.fth for working with sets (modified from code written by Anton Ertl in gray.fs)
  - shellsort.fth
  - syntaxtree.fth builds and handles syntax trees
  - transitiontable.fth builds a transition table for a DFA from the syntax tree
  - lexarrays.fth compresses the transition table into the output data structure
  - savetables.fth writes the data to a file
  - userinterface.fth defines the words for users
- scanner.fth and scanner2.fth are example scanner files for target applications
- an example using symbols from the C programming language
  - tokens.fth a user file for the target application giving token values for keywords etc
  - lexinput.fth a user file containing the keywords , symbols and regular expressions for C
  - test1.txt a source file for testing recognition of C keywords etc
  - lextables.fth the generated data files

To run the example start gforth and type:

```
marker restart
s" lexgen\lexgen.fth" included
restart
s" lexgen\scanner.fth" included
s" lexgen\test1.txt" scan
```

This should result in a list of recognised symbols scrolling up the screen. Users of other Forth systems will probably have to amend file paths to get it to work as there is no standard for handling directories.

3 Specification of patterns to be recognised

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
```

4 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 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.

5 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 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`)

`<.> ( 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-sensitive ( -- )` Sets a flag to make following symbols case insensitive
e.g program = PROGRAM = PrOgRaM. Applies to symbols only, not character sets or single characters

`case-insensitive ( -- )` Clears the case sensitivity flag for following symbols
i.e. program <> PROGRAM. This is the default state

`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. Note that the first use of `symbol` does not require a tree on the stack. `symbol` should only be used between `begin-symbols` and `end-symbols`. `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. Note that the first use of `symbol` does not require a tree on the stack. `denotes` should only be used between `begin-symbols` and `end-symbols`

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

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

6 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 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.

c. Use of `char` instead of `'char'` and vice versa.

7 Output from LexGen

Data is output in an ASCII text file that may be easily included into a target application. This output consists of numbers followed by `,` (the Forth comma word) for the Forth text interpreter to compile the number into dataspace. These numbers are indexes into the data arrays. How target applications choose to use this data is beyond the scope of this note. Should a different format be required this should be easily achievable by modification of the file savetables.fth, for example:

- If the tables are small enough (fewer than 256 rows) to use `c,` rather than `,` to reduce the amount of dataspace required by using bytes instead of cells. The scanner would need to be changed accordingly.
- To save data as a binary file that would be faster to load into the target application
- To save data for a different target language

8 The scanner for target applications

Two scanners are provided as examples:

a. The first in scanner.fth loads the data, converts the index numbers to absolute addresses before scanning input test. This is faster, when scanning than …

b. scanner2.fth that loads the data and uses the indexes unchanged. This would have to be used should LexGen output data be converted to use byte data – in this case the scanner itself would also need to be adjusted to access bytes.

As already described above, to run the scanner, once loaded type:

```
s" lexgen\test1.txt" scan
```

9 Future possible enhancements

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

b. Use of standard notation for regular expressions.