Recap

In summary:

- The specification of a lexical analyser generator consists of two parts:
    1. Specification of tokens – done through regular expressions.
    2. Specification of actions - done through action routines.
- The lexical analyser generator:
    - Processes the regular expressions and forms a graph called DFA.
    - Copies the action routines without any change.
    - Adds a driver routine whose behaviour we described.

  These three things put together constitutes the lexical analyser.

Issues

- What are regular expressions? How can they be used to describe tokens?
- How can regular expresions be converted to DFA?

Introduction to Regular Expressions

A regular expressions denote a set of strings, also called *a language*. For example, **a*b** denotes the language $\{$**b, ab, aab, aaab, . . .**$\}$. We denote the language of a regular expression r as $L(r)$.

A single character is a regular expression.

- Examples: **a**, **Z**, **n**, **t**.
- Denotes a singleton set containing the character. **a** denotes the set $\{$**a**$\}$.

Introduction to Regular Expressions

ϵ is a regular expression.

- Denotes $\{\epsilon\}$, the set containing the empty string.

Introduction to Regular Expressions

If r and s are regular expressions then $r|s$ is a regular expression.

- Examples: $\mathbf{a}|\mathbf{b}|\ \ldots|\mathbf{z}|\mathbf{A}|\mathbf{B}|\ldots|\mathbf{Z}$ and $\mathbf{0}|\mathbf{1}|\ldots|\mathbf{9}$. Let us call these regular expressions **LETTER** and **DIGIT**.
- $L(r|s)$ is the union of strings in $L(r)$ and $L(s)$.

Introduction to Regular Expressions

If r and s are regular expressions then rs is a regular expression.

- Examples: `begin` – with an assumed associativity.
- `{LETTER}({LETTER}|{DIGIT})*`.
    - Notice that the braces required around `LETTER` is a lex requirement and denotes that it is a synonym for a regular expression and not the literal `LETTER`.
- $L(rs)$ is the concatenation of strings x and y such that $x \in L(r)$ and $y \in L(s)$.

Introduction to Regular Expressions

If r is a regular expressions then r^* is a regular expression.

- Examples: `({LETTER}|{DIGIT})*`
- $L(r^*)$ is the concatenation of zero or more strings from $L(r)$. Concatenation of zero strings is defined to be the null string.

Introduction to Regular Expressions

If r is a regular expressions then (r) is a regular expression. Parentheses are used for grouping.

- Examples: `({LETTER}|{DIGIT})*`
- The language denoted by (r) is $L(r)$.

Introduction to Regular Expressions

Shorthand: If r is a regular expressions then r^+ is a regular expression.

- Examples: {`DIGIT`}`+`
- $L(r^+)$ is the concatenation of one or more strings from $L(r)$.
- $r^+ = rr^*$.

Introduction to Regular Expressions

Shorthand: If r is a regular expressions then $r?$ is a regular expression.

- Examples: {`DIGIT`}`?` denotes zero or one occurrence of a digit.
- $r?$ stands for zero or one occurrence of strings in r.
- $r? = \epsilon | r$

Regular expressions provided by Lex

Expression	Describes	Example
c	any character c	a
\c	character c literally	*
"s"	string s literally	"**"
.	any character except newline	a.*b
^	beginning of a line	^abc
$	end of line	abc$
[s]	any character in s	[abc]
[^s]	any character not in s	[^abc]
r*	zero or more r's	a*
r+	one or more r's	a+
r?	zero or one r	a?
$r_1 r_2$	r_1 then r_2	ab
$r_1 \| r_2$	r_1 or r_2	a|b
(r)	r	(a|b)
r_1 / r_2	r_1 when followed by r_2	abc/123

Example of token specification in Lex

```
[ \t\n]+                       {/*no action, no return*/}
if                             {return(IF);}
then                           {return(THEN);}
else                           {return(ELSE);}
{letter}({letter}|{digit})*    {yylval=install_id(); return(ID);}

-?{digit}+(\.{digit}+)?(E[+-]?{digit}+)?
                               {yylval=atof(yytext); return(NUM);}

"<"                            {yylval=LT; return(RELOP);}
"<="                           {yylval=LE; return(RELOP);}
"+"                            {yylval=PLUS; return(ADDOP);}
"*"                            {yylval=MULT; return(MULOP);}
```

A Larger Example

1. Starting from a input position, detect the longest lexeme that could match a pattern.

   Example: Return `begin` and not `b`, `be`, `beg` . . . .

   Where has this decision been incorporated in our description of the generated lexical analyser?
2. If a lexeme matches more than one patterns, declare the lexeme to have matched the earliest pattern.

   Example: The state numbered 5 corresponds to the pattern `begin` and not `identifier`.

LEXICAL ERRORS

Primarily of two kinds:

1. Lexemes whose length exceed the bound specified by the language.
   - In (old time) Fortran, an identifier more than 7 characters long is a lexical error.
   - Most languages have a bound on the precision of numeric constants. A constant whose length exceeds this bound is a lexical error.
2. Illegal characters in the program.
   - The characters `~`, `&` and `@` occuring in a Pascal program (but not within a string or a comment) are lexical errors.
3. Unterminated strings or comments.

Handling Lexical Errors

The action taken on detection of an error are:

1. Issue an appropriate error message.
2. - Error of the first type—the entire lexeme is read and then truncated to the specified length. Generates a warning.
   - Error of the second type—
     - Skip illegal character—this is what was discussed earlier. What does `flexc++` do?
     - A possibility which is rarely practiced—pass the character to the parser which has better knowledge of the context in which error has occurred. This opens up more possibilities of recovery - replacement instead of deletion.
   - Error of the third type—wait till end of file and issue error message.