0.1 Patterns

Section **??** already discussed the format of the pattern file. In this section, it is described how to specify patterns by the use of regular expressions. The first releases of queχ relied a core that is produced by flex. From version 0.9.0 on queχ provides its own core engine—leaving the flex engine as an option. For the sake of established habits the descriptions of regular expressions are kept conform with the world of lex/flex.

Exceptions are marked with a ~~f~~`lex!`-sign, meaning 'not to be used when creating a flex core engine[1]. Complications that are mentioned which are not true for queχ are marke with the ~~q~~`ue`χ`!`-sign. Note further, that what is called 'start conditions' in lex or flex is replaced with the more powerful concept of *modes* in quex. Thus, there is no discussion on start conditions in the context of patterns. The explanation is divided into the consideration of context-free expressions and context-dependent expressions.

- `R/S` an R but only if it is followed by an `S`. The text matched by `S` is included when determining whether this rule is the "longest match", but is then returned to the input before the action is executed. So the action only sees the text matched by `R`. This type of pattern is called "trailing context".

  There are some combinations of '`R/S`' that 'flex' cannot match correctly; see notes in the Deficiencies / Bugs section below regarding "dangerous trailing context" in the flex manual [**?**]. No such deficiencies are know about the generated core engine produced by queχ!

- `^R` an `R`, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).

- `R\$` an R, but only at the end of a line (i.e., just before a newline). Equivalent to "R/\n".

  Note that flex's notion of "newline" is exactly whatever the C compiler used to compile flex interprets '\n' as; in particular, on some DOS systems you must either filter out \r's in the input yourself, or explicitly use R/\r\n for "r$".

  ... [*explanation of flex' start-conditions is left out, since queχ provides its own concepts of modes.*]

- `<<EOF>>` an end-of-file. ... [*discussion of start conditions is left out.*].

Note that inside of a character class, all regular expression operators lose their special meaning except escape ('\') and the character class operators, '-', ']', and, at the beginning of the class, '^'.

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence. For example,

```
foo|bar*
```

is the same as

[1] At places where flex does not provide the particular feature, of course, additions to the syntax were made. For people, who insist on using a flex created engine, it is essential that they do not rely on features such as sophisticated pre-conditions, or unicode related expressions.

```
(foo)|(ba(r*))
```

since the '*' operator has higher precedence than concatenation, and concatenation higher than alternation ('—'). This pattern therefore matches *either* the string "foo" *or* the string "ba" followed by zero-or-more r's. To match "foo" or zero-or-more "bar"'s, use:

```
foo|(bar)*
```

and to match zero-or-more "foo"'s-or-"bar"'s:

```
(foo|bar)*
```

[*NOTE: In queχ the following character class expressions may be deleted in later versions, because for unicode implementations they require some more complicated interaction with the definitions of so called 'locals'.*

In addition to characters and ranges of characters, character classes can also contain character class "expressions". These are expressions enclosed inside '[': and ':'] delimiters (which themselves must appear between the '[' and ']' of the character class; other elements may occur inside the character class, too). The valid expressions are:

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

These expressions all designate a set of characters equivalent to the corresponding standard C 'isXXX' function. For example, '[:alnum:]' designates those characters for which 'isalnum()' returns true - i.e., any alphabetic or numeric. Some systems don't provide 'isblank()', so flex defines '[:blank:]' as a blank or a tab.

For example, the following character classes are all equivalent:

```
[[:alnum:]]
[[:alpha:][:digit:]
[[:alpha:]0-9]
[a-zA-Z0-9]
```

If your scanner is case-insensitive (the '-i' flag), then '[:upper:]' and '[:lower:]' are equivalent to '[:alpha:]'.

Some notes on patterns:

- A negated character class such as the example "[^A-Z]" above *will match a newline* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., "[^A-Z\n]"). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like [^"]* can match the entire input unless there's another quote in the input.

- A rule can have at most one instance of trailing context (the '/' operator or the '$' operator). The start condition, '^', and "<<EOF>>" patterns can only occur at the beginning of a pattern, and, as well as with '/' and '$', cannot be grouped inside parentheses. A '^' which does not occur at the beginning of a rule or a '$' which does not occur at the end of a rule loses its special properties and is treated as a normal character.

  The following are illegal:

  ```
  foo/bar\$
  <sc1>foo<sc2>bar
  ```

  Note that the first of these, can be written "foo/bar\n".

  The following will result in '$' or '^' being treated as a normal character:

  ```
  foo|(bar\$)
  foo|^bar
  ```

0.1.1 Context Free Regular Expressions

Context free regular expressions match against an input independent on what come before or after it. For example the regular expression "`for`" will match against the letters `f`, `o`, and `r` independent if there was a whitespace or whatsoever before it or after it. This is the 'usual' way to define patterns. More sophisticated techniques are explained in the subsequent section. This sections explains how to define simple *chains of characters* and `operations on chains of characters` to combine them into powerful patterns.

Chains of Characters:

- `x` matches the character 'x'.

  That means, lonestanding characters match simply the character that they represent. This is true, as long as those characters are not operators by which regular expressions describe some fancy mechanisms—see below.

- `.` matches any character (byte) except newline

- `[xyz]` a "character class" or "character set"; in this case, the pattern matches either an 'x', a 'y', or a 'z'. The brackets '[' and ']' are examples for lonestanding characters that are operators. If they are to be matched quotes or backslashes have to be used as shown below.

- `[abj-oZ]` a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'. The minus '-' determines the range specification. Its right hand side is the start of the range its right hand sinde the end of the range (here 'j-o' means from 'j' to 'o').

- `[^A-Z]` a "negated character class", i.e., any character but those in the class. In this case, any character *except* an uppercase letter.

- `[^A-Z\n]` any character *except* an uppercase letter or a newline.
- `"[xyz]\"foo"'` the literal string: '[xyz]"foo'.

  That is, inside quotes the characters which are used as operators for regular expressions can be applied in their original sense. A '[' stands for code point 91 (hex. 5B), matches against a '[' and does not mean 'open character set'.
- `\X` if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \X. Otherwise, a literal 'X' (used to escape operators such as '*')
- `\0` a NULL character (ASCII/Unicode code point 0).
- `\123` the character with octal value 123.
- `\x2a` the character with hexadecimal value '2a'.

Operations on Character Chains:

Let `R` and `S` be character chains (maybe consisting out of a single character) specified as mentioned above.

- `R*` *zero* or more occurencies of `R`.
- `R+` *one* or more `R`'s
- `R?` *zero* or *one* `R`. That means, there maybe an `R` or not.
- `R{2,5}` anywhere from two to five `R`'s
- `R{2,}` two or more `R`'s.
- `R{4}` exactly 4 `R`'s.
- `(R)` match an `R`; parentheses are used to override precedence (see below).
- `RS` the regular expression `R` followed by the regular expression `S`; called concatenation or sequence.
- `R|S` either an R or an S, i.e. `R` and `S` are two valid alternatives.
- `{NAME}` the expansion of the "NAME" definition (see above).

0.1.2 Pre- and Post-Conditions