- Table of Contents
- Previous Chapter

Chapter 6: Technical documentation (Sat, 28 Dec 2013 18:21:17 +0100)

Starting point of the analysis is the parser. The parser needs access to `States` and `Rules`. The parser is described here.

6.1: Notation, Terminology

6.1.0.1: Notations

- `BOL`: Begin-of-line is abbreviated to BOL.
- `LOP`: the lookahead operator (`/`) is abbreviated to LOP.
- `RE`: A regular expression.

- `Ax`: accept count, indicating that *x* state transitions have been performed since the initial accepting state (`A0`).
- `Ax`: incrementing accept count. The accept count of this state is incremented each time this state is reached again
- `Fx`: A final state with an associated accept count.
- *pre-A-state*: any NFA state in a pattern preceding its `A0` state.
- *post-A-state*: any NFA state in a pattern at or beyond its `A0` state.
- *LA operator*: the *lookahead* operator (`'/'`).
- `x[y]`: at state `x` an empty transition to state `y` is allowed. See figure 2.

6.1.0.2: Example patterns with an LA operator

When an `F` state of a LOP pattern is reached and there is no continuation at that point then the pattern until the A0 state is considered matched and is returned.

- The pattern `a/a+`:

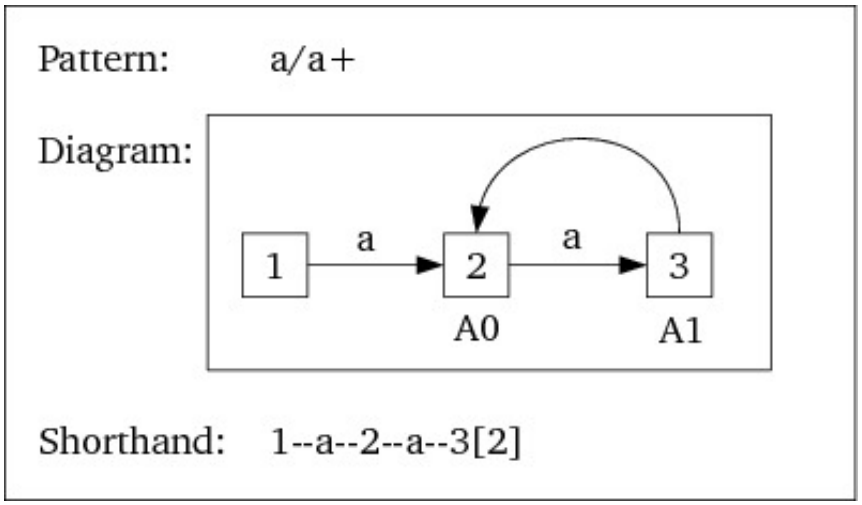

Figure 2: The pattern a/a+

The accept count of this state is A0, and state 2 is its Final state

- The pattern `a*/aaa`: Start counting from the `A0` state until the F-state has been reached.

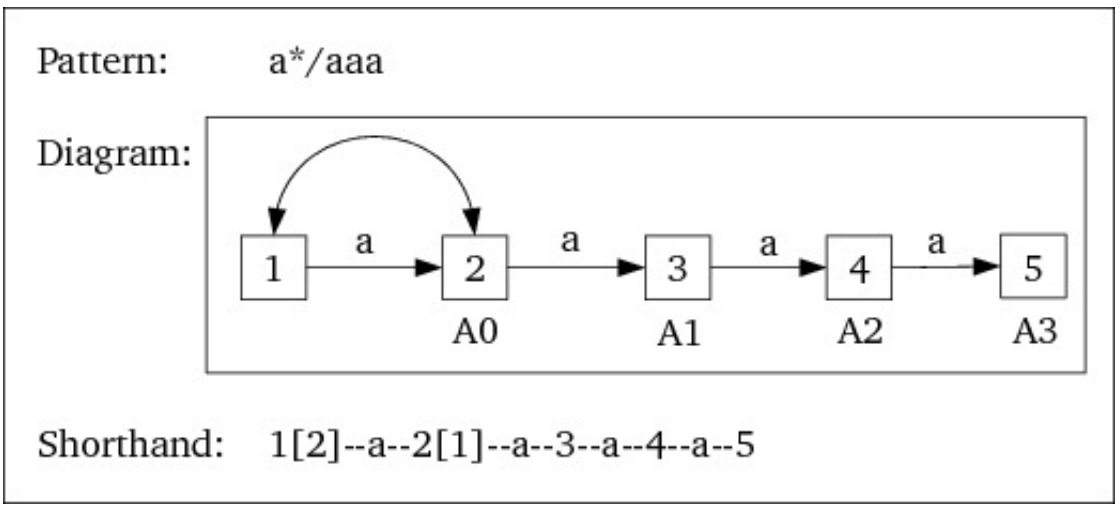

Figure 3: The pattern a*/aaa

The resulting DFA becomes:

	Input Ch		
StateSet	a	Fin/Acc	#steps
12	123	A0	
123	1234	A1	1
1234	12345	A2	2
12345	12345	F, A3	3

6.2: The parser

The parser is called by `flexc++` as follows:

```
Parser parser(rules, states);
parser.parse();
```

The constructor performs no other tasks than initializing the Parser object. `Rules` and `states` are still empty at this point.

The `parse` function is generated by bisonc++. It's better understood from the grammar description.

`parser/grammar`: The grammar's start rule is

```
input:
    opt_directives
    section_delimiter
    rules
;
```

`parser/inc/rules`: The directives are not covered here yet. A rule is defined as:

```
rules:
    rules _rule
|
    // empty
;
```

As a notational convention, all non-terminals not used outside of the file in which they are defined start with an underscore. Furthermore, wherever possible rules are defined before they are used. This cannot always be accomplished, e.g., when defining indirectly recursive rules.

A `_rule` is a rule definition followed by a newline:

```
_rule:
    _rule_def '\n' reset
;
```

The `reset` non-terminal merely calls `Parser::reset`, preparing the parser for the next rule, resetting various data members to their original, begin-of-line states. See `parser/reset.cc` for details.

A `_rule_definition` may be empty, but usually it consists of a rule optionally valid for a set of mini-scanners (`optMs_rule`, or it is a rule or series of rules embedded in a mini-scanner compound (`msComound`):

```
_rule_def:
    // only an empty line is OK
|
```

```
        // recovery from errors is OK: continue at the next line
        error
    |
        optMs_rule
    |
        msCompound
    ;
```

The `msCompund` isn't interesting by itself, as it repeats `optMs_rule`, but without the mini-scanner specifications for the individual rules.

An `optMs_rule` is a rule (`_optMs_rule`), optionally followed by an action:

```
    optMs_rule:
            // just regular expressions, without an action block
        _optMs_rule
    |
        // the scanner returns a block if it encounters a non-blank character
        // on a line after ws, unless the character is a '|'
        _optMs_rule action
        {
            assignBlock();
        }
    ;
```

An `_optMS_rule` defines one or more rules, sharing the same action block:

```
    _optMs_rule:                            // after ORNL continue parsing
        _optMs_rule ORNL '\n' reset _optMs_rule // at a new line
        {
            orAction();
        }
    |
        _optMs_regex
        {
            addRule($1, true);  // true: reset the miniscanner spec to INITIAL
        }
    ;
```

And an `_optMS_regex`, finally, is a regular expression or EOF-pattern, optionally prefixed by a mini scanner specification:

```
    _optMs_regex:
        ms_spec regexOrEOF
        {
            $$ = $2;
        }
    |
        regexOrEOF
    ;
```

A `regexOrEOF` defines exactly that:

```
regexOrEOF:
    _regex
|
    EOF_PATTERN
    {
        $$ = eofPattern();
    }
;
```

Actions are defined here; the `Rules` class is described here; Regular expressions (i.e., `regex`) are described here.

6.3: Semantic Data, struct DataType and the class SemUnion

TODO

6.3.1: The class Pattern

A `PatternVal` is derived from `SemVal`. It mainly consists of static factory functions returning shared pointers to `SemVal` objects (`spSemVal`).

`PatternVal` objects contain (and offer, through the members `begin` and `end`) index values in the `States::d_states` of `State` objects. The value returned by `PatternVal::begin()` is the index of the first `State` defining the pattern, the value returned by `PattenVal::end()` is the index of the last (so: `State::FINAL`) `State` of the pattern (cf. figure 4).

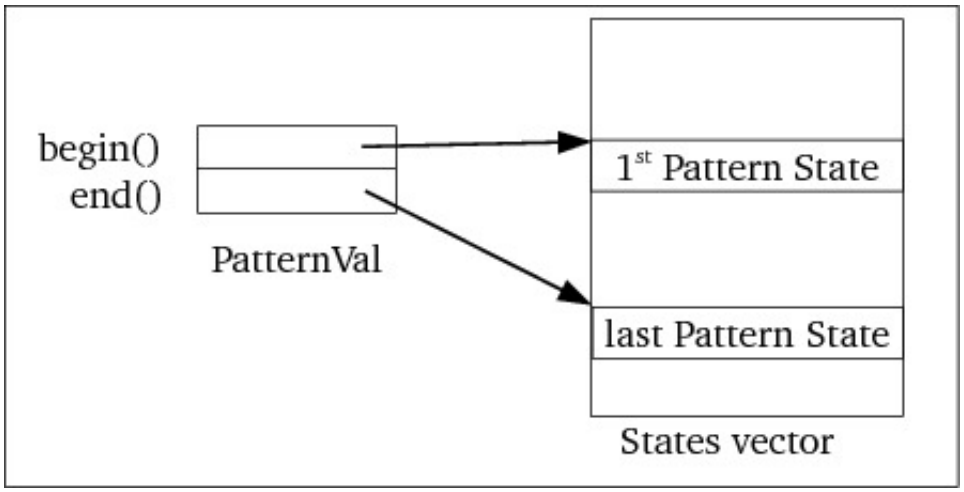

Figure 4: PatternVal objects

`parser/rules/pattern`: the following descriptions of the working of some basic patterns illustrate how the vector of `State` objects in `States` is filled. Based on this description other pattern's implementations should be evident.

A basic pattern is the plain character. The plain character pattern is processed as follows:

- First the indices of two free `State` vector locations are requested (see the `next2` description in the `States` clas).

- The `State` at the first state index is then set to a state containing the plain character, linking to the next free state which has been initialized to the `FINAL` state by `States::next2`.
- Then the `PatternVal` is embedded in a `SemVal` (see the `SemVal` class description).
- The `SemVal` is then returned in a `std::shared_ptr` (cf. figure 5).

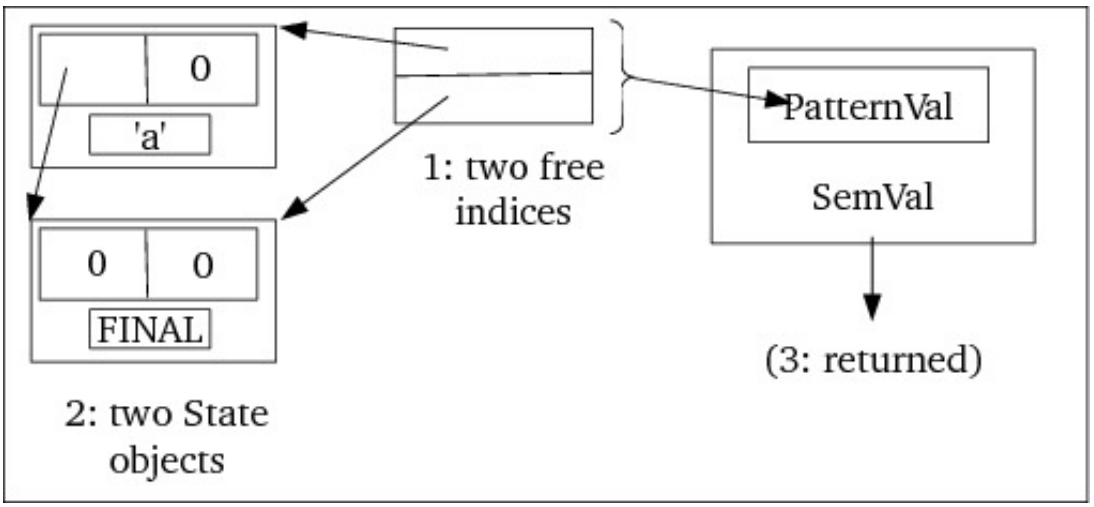

Figure 5: A plain character pattern

Concatenation of two patterns always produces a free `State` that can be returned by, e.g., `State::nex2`. The states, defining a pattern, therefore, do not have to be consecutively stored in the `States`'s vector of `State` objects (see figure 6).

- Concatenation starts with two `SemVal` objects.
- The `SemVal` objects are downcast to, resp. a `lhs PatternVal` and a `rhs PatternVal`.
- With lookahead patterns, the `lhs` pattern may be an *accepting state*. I.e., once the full pattern has been recognized only the lhs is actually matched. E.g., after recognizing `a/b` `a` is returned as matched, as `a` is the accepting pattern. When concatenating the lhs's end state disappears and is replaced by the rhs's begin state. Therefore:
- The rhs's begin state's accepting flag is set to the lhs's end state's accepting flag.
- The `lhs.end()` state is assigned the value of the `rhs.begin()` state
- The `rhs.begin()` state is marked as free
- A new `PatternVal` is returned as `SemVal *` having its `begin` index set to `lhs.begin()` and its `end` idex set to `rhs.end()`.

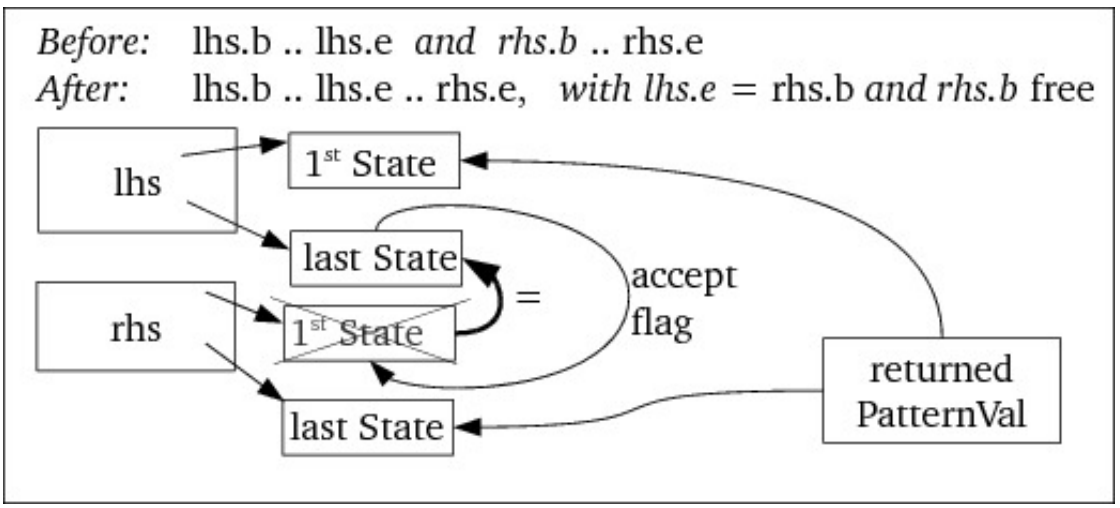

Figure 6: Concatenating Patterns

6.3.2: The class CharClass

TODO

6.3.3: The class Interval

TODO

6.4: Start Conditions and the class StartConditions

TODO

6.5: Code (action) blocks

The block of **C++** code is simply parsed from the first `{` until the final matching `}`, acknowledging comment, strings and character constants.

The scanner merely recognizes the beginning of an action (block), returning `Parser::BLOCK` from its `handleCharAfterBlanks` member. Then, `Parser::block` calls `Scanner::lex()` until it observes the end of the action statement or block.

The parser stores the block's content in its `d_block` member for later processing by `Rules::add`.

6.6: The class State

Objects of the class `State` contain one single element of a pattern. A `State` contains/offers:

- `d_type`: the type represented by the State (EMPTY, FINAL, CHARSET, BOL), the internally used `UNDETERMINED__` and `EOF__` types, or the ascii-value of a single character;
- `d_flag`: the state-type: `NO_LOP` if the RE for this state does not use the LOP; `ACCEPT` if this state is the first state following the LOP (reached via an empty transition from the last state before the LOP); `PRE` for all states preceding the `ACCEPT` state; `POST` for all states following the `ACCEPT` state.
- `d_rule`: an index into the Rules object to the rule object defining the pattern;
- A shared pointer to `StateData`. The class `StateData` is derived from SemVal, and contains two indices: the `State` indices of the states following the current `State`. The second index (`d_next2`) may be 0, indicating that there is no second continuation state. A second continuation state occurs when the *alternate* operator (`'|'`) is used in a pattern.

Starting from a pattern's initial state all its subsequent states can therefore be reached by visiting the indices stored in `StateData`.

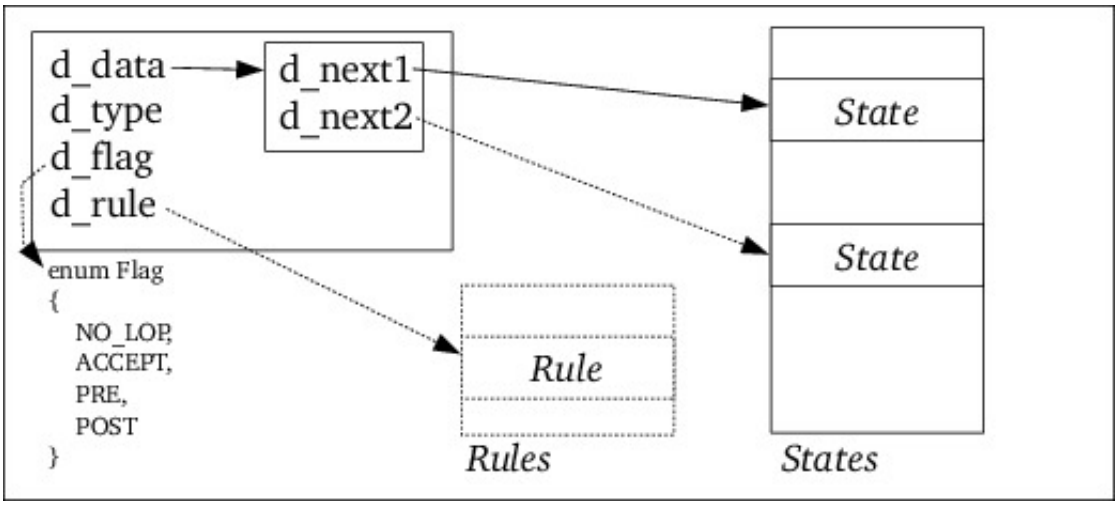

Figure 7: The State class

6.7: States

The `States` class object holds all the states defining all patterns (`d_states`). When a new pattern is requested the member `next` or `next2` can be invoked to retrieve the next or next two free states. Since concatenating pattern produces free states (cf. the description of the pattern concatenation) a vector of free state indices is maintained by the `States` class as well (`d_free`).

The member `next2` returns a pair of free state indices, the second state index refers to a `State` that has been initialized to the end-of-pattern state: its state type is `State::FINAL` and its two successor (next) fields are set to 0, indicating that there is no continuation from this state.

Patterns consisting of multiple states (like the pattern `ab`, consisting of the state holding `a` and the state holding `b`) are stored in a linked-list like structure, defined in the `States`'s `d_state` vector. Its representation is as follows (cf. figure 8:

- The `next1()` member of the state containing the `a` pattern returns the index of the state containing the `b` pattern.
- The `next1()` member of the state containing the `b` pattern returns the index of the final state
- The final state's `next1` and `next2` members return 0
- Except for the pattern using the `|`-operator (`pattern | pattern`) all `next2` members return 0. The implementation of the `|`-operator is described here.

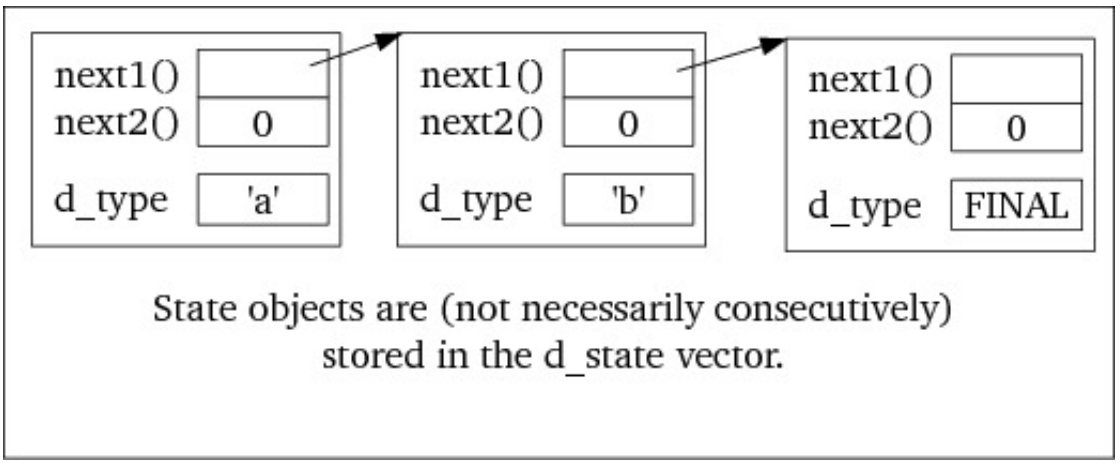

Figure 8: Catenating States

6.8: Rules and the class Rule

All rules are accessible from the `Rules` object. It contains a reference to the *states* (see here), and a vector `d_rules` containing the information of each of the rules, and a hash table linking a final state index back to its rule number (see figure 9)

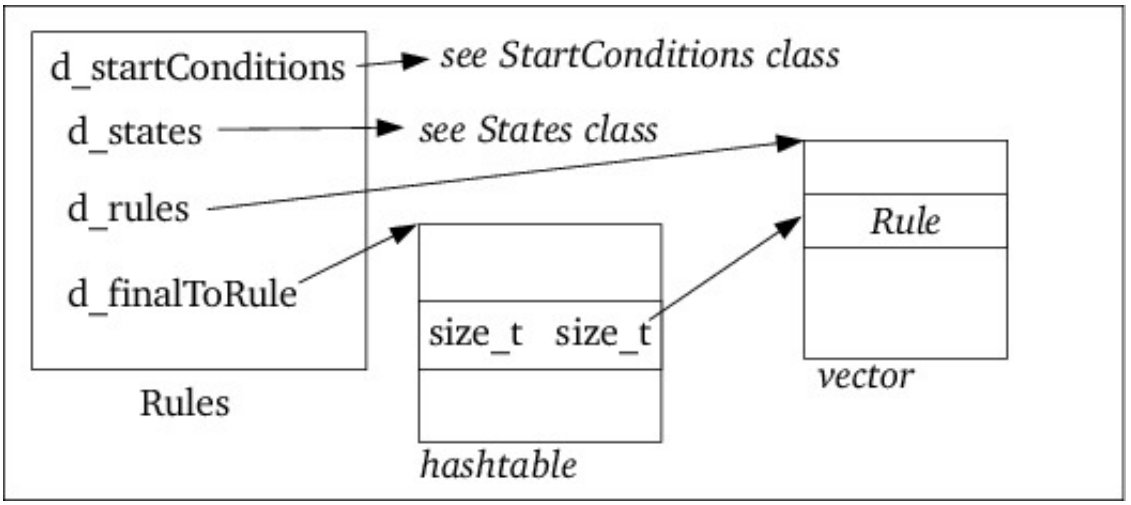

Figure 9: The Rules class

When the parser has detected a rule it calls `Rules::add`. A `Rule` object is added to `d_rules`, storing begin and end state indices, a flag indicating whether or not this rule's RE started at BOL and a *viable* flag, indicating whether the rule can at all be matched. This flag is initially `false`, but it is set to `true` if the rule is mentioned in at least one of the final states of any of the DFAs. A rule also contains the code of any action block that's associated with it (empty if there are no actions), see also section 6.5.

`Rules::add` also stores the association between the rule's final state and rule index in its `unordered_map d_finalToRule`. Furthermore, it calls `d_startConditions.add(ruleIdx)` to store the rule index at the currently active start conditions. See section 6.4 for the class `StartConditions`.

States of rules using the LOP need access to the rule to which they belong. For those rules `Rules::setRuleIndices` is called to assign the State's `d_rule` field.

The `Rule` object themselves have an organization shown in figure 10.

Rules starting with `^` can only be matched at *begin-of-line* (BOL). The data member `d_bol` of such rules is set to `true`.

Rules that are matched in some DFA state are *viable*: such rules can be matched. The `d_viable` data member of such rules is set to `true`. A rule's `d_viable` data member may be set to true at some point during a DFA construction, but during the DFA construction another (earlier) rule might be matched in the same DFA state. In those cases the viable state of the later rule is reset to `false`. Once all DFAs have been constructed a simple visit of the `Rules` provides information about their viability.

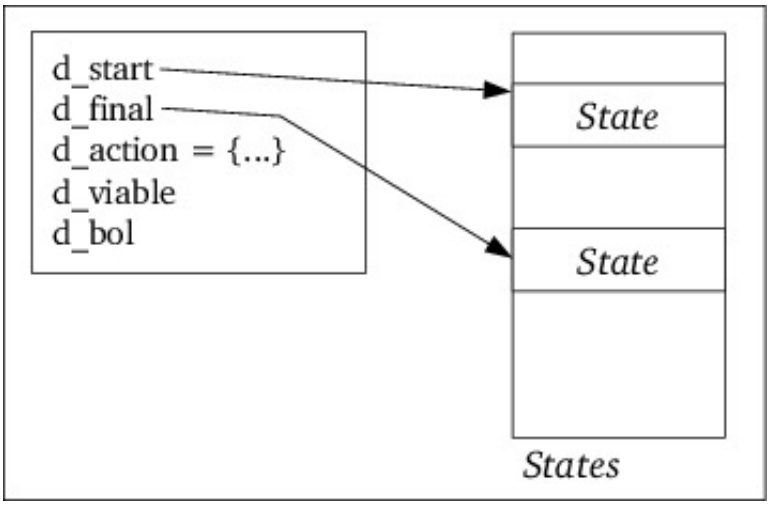

Figure 10: The Rule class data members

6.9: Patterns

`parser/rules/pattern`: Patterns define the regular expressions that can be matched. All patterns start at `pattern_ws` and consist of:

```
pattern_ws:
    pattern opt_ws mode_block
;
```

Following the recognition of a pattern, the scanner is reset to its `block` mode, allowing it to recognize a **C++** action block. Blocks are defined here.

The following patterns are defined (more extensive descriptions follow):

- `EOF_PATTERN` - recognized by the lexer, matching `<<EOF>>`.
- `STRING` - recognized by the lexer, matching a literal string.
- `SECTION_DELIMITER` - recognized by the lexer. The `%%` sequence in fact ends the rule section

  TO BE INVESTIGATED.

- `character_class`: a self-defined or predefined character class like `[a-c]` or `[[alpha]]`.
- `plain_characters`: any plain character, like `a` in `ape`.
- `ESCAPE_SEQUENCE`: characters defined by escape sequences, like `\x2a`. To the parser they are plain characters.
- `'.'`: the any-character-but-newline matching pattern.
- `pattern pattern`: two patterns immediately following each other. These two patterns have the precedence of `CHARACTER`, and are combined left-associatively.
- `pattern '|' pattern` Two alternative continuations.
- `pattern quantifier`: quantifiers are `*`, `+` and `?`.
- `'(' incParen pattern ')' decParen` a pattern nested in parentheses
- `pattern '{' start_interval_m interval '}' regex_block_m` a repetition using curly braces (an interval repetition)
- `pattern '/' pattern` a lookahead pattern ($ is handled by the scanner)

All patterns are handled by `PatternVal` functions. E.g., a `STRING` is handled by `PatternVal::str`, a character class by `PatternVal::charSet`, etc. See here for information about the `PatternVal` class.

6.10: Ranges

When processing characters in regular expressions subranges are defined. An expression like `` `am' `` defines the ranges ascii-0 to (but not including) `'a'`; `'a'`; `'b'` through `'l'`; `'m'`; and `'n'` through ascii-255.

Likewise, sets may define ranges, like `[[:digits:]]`, defining all ascii characters preceding the digits; all decimal digits; and all ascii characters following the digits.

Rather than having a NFA/DFA having entries for each of these characters the NFA/DFA's column-dimension (see section 6.13) can be reduced, often considerably, by letting each column refer to a *set* of characters, rather than individual characters.

The `Ranges` object takes care of defining and manipulating the actual subsets. Its data organization is given in figure 11.

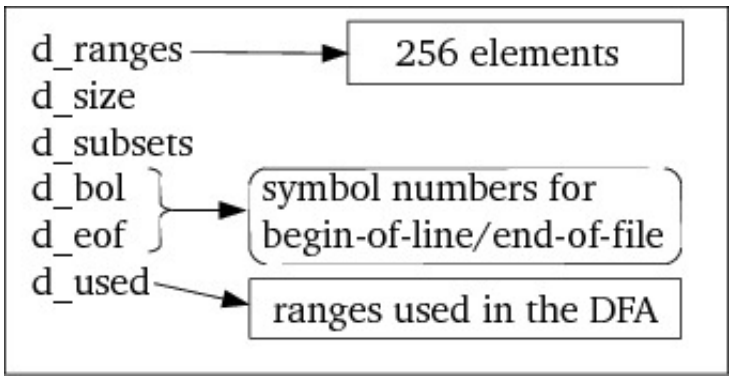

Figure 11: The class Ranges data members

Initially `d_ranges` is filled with all zeroes. Once a range of characters or a single character is defined in a pattern, it is added to the `Ranges` object (functions `add` accepting single characters or strings). The occurrence counts of the added characters are incremented if necessary. De functions `update`, `collision`, and `countRanges` handle the updating.

Details of the algorithm are not covered here, until the need for this arises. For the time being consult the sources for details.

In the user interface the important members are `rangeOf, rangeOfBOL` and `rangeofEOF`, returning the column indices in DFAs to which input characters belong.

6.11: The class TailCount

The class `TailCount` collects the tail sizes of rules using the LOP. Its organization is shown in figure 12. `TailCount` objects are used in the rows of DFAs (cf. section 6.14).

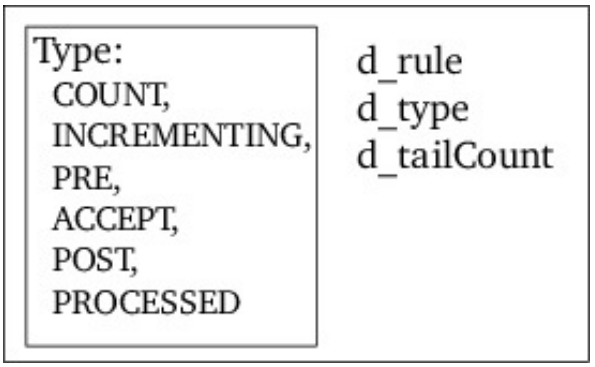

Figure 12: The TailCount class

A `TailCount` object has three data members:

- `d_rule`: this data member holds the index of the rule to which the `TailCount` object refers.
- `d_type`: this is a `TailCount::Type` (a public accessible type). Its values are bit-flagged values, and the values PRE, ACCEPT, and POST may be combined with the values COUNT and INCREMENTING.
  The values PRE, ACCEPT, and POST indicate that the DFA row containing the `TailCount` object represents NFA states before (PRE), at (ACCEPT) or beyond (POST) the LOP.
  The flag COUNT is set when the `TailCount`'s `d_tailCount` value is a fixed value and the flag `INCREMENTING` is set when the current length of the tail must be incremented (cf. section 6.15).
  The flag `PROCESSED` is set by members of the class `DFA` (cf. sections 6.13 and 6.13.1) when a `DFA` row's tail count has been determined or is otherwise known.
- `d_tailCount`: the tail length (so far) of a matched text. Its value is only valid if the COUNT flag is also set in `d_type`.

6.12: The class DFAs

The regular expression patterns define non-finite state automata (NFA) which must be converted to deterministic finite state automata (DFA). Each mini scanner has its own DFA and the class `DFAs` builds the various DFAs for each of the mini scanners. The DFA construction needs access to the rules, states and character ranges, which are available to the `DFAs` object) (cf. figure 13)

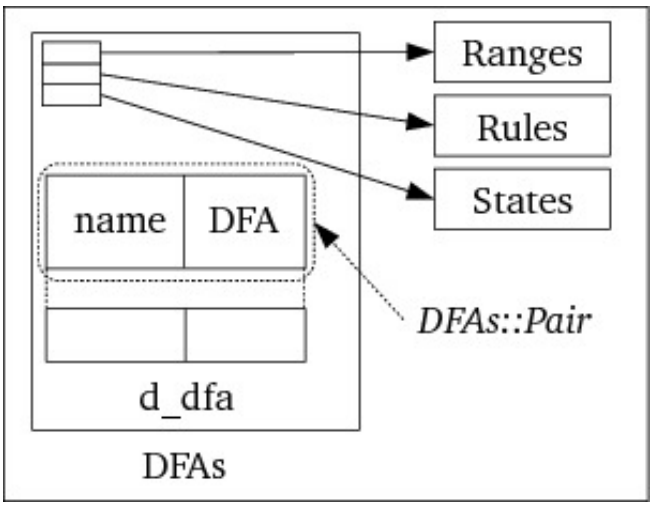

Figure 13: The DFAs class organization

The DFAs object's real work is performed by its build member. The build member visits all the elements of Rules, passing each of the start conditions (stored in the Rules object) to buildDFA. For each of the start conditions, holding the indices of the rules that are defined for the various start conditions, a DFA is constructed for all of the rules of that start condition (i.e., mini-scanner) (cf. figure 6.4).

The function buildDFA performs two tasks:

- It adds another element to its d_dfa vector. Each element of d_dfa contains
  - the name of the mini scanner (obtained from the Rules's NameVector), and
  - an initialized DFA object (see section 6.13)
- Next it calls the DFA function build for the just initialized DFA, passing it the vector of indices of the start states for that mini scanner. Build will construct the mini scanner's DFA, see section 6.13.

6.13: The DFA

The conversion from NFA to DFA is the job of the class DFA object (cf. figure 14)

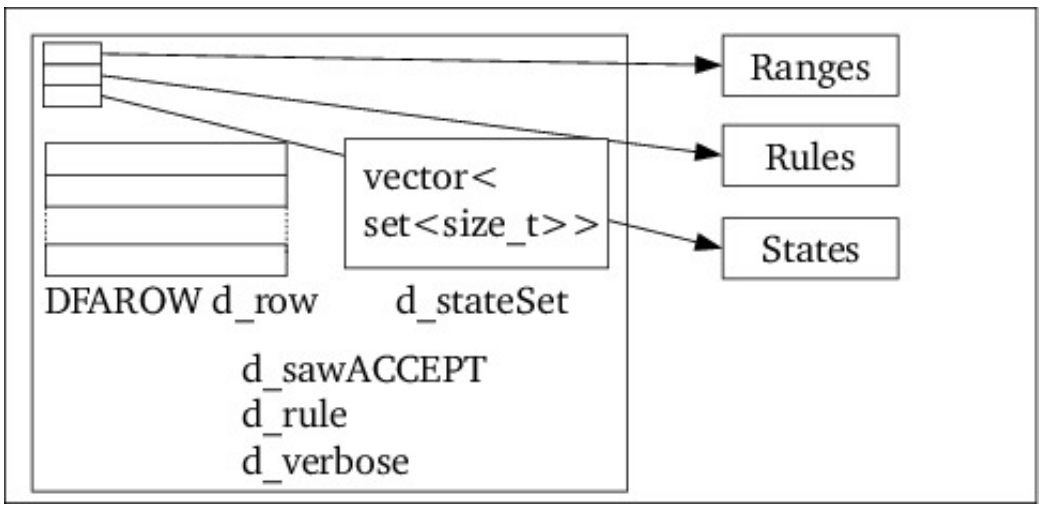

Figure 14: The DFA class data members

The DFA uses the externally available rules, states and character ranges and builds a vector of DFARows, named d_row. It has the following data members:

- d_row: the vector of DFARow objects, defining the rows of the DFA (cf. section 6.14);
- d_stateSet: a vector of sets of State indices. There are as many sets in this vector as there are elements in d_row, and each element holds the indices of the State objects in d_states that are represented by the matching DFARow object.
- d_sawAccept: a bool set to *true* while processing TailCount objects once a row representing an ACCEPT state has been encountered. TailCounts are processed by computeTailCounts, see below.
- d_rule: the rule index represented by a TailCount object in d_row[0].
- d_verbose: shadows the presence of the --verbose program flag.

Building the DFA from the NFA is the task of `DFA::build`.

Each row of `d_row` defines a state in the DFA. The Rule numbers of the Rules defining a mini scanner received as `build`'s `vector<size_t>` parameter.

`DFA::build` initially stores the start states of the rules of its mini scanner in its `d_stateSet[0]`th element, which is a set (cf. figure 15). This is done by `DFA::fillStartSet`

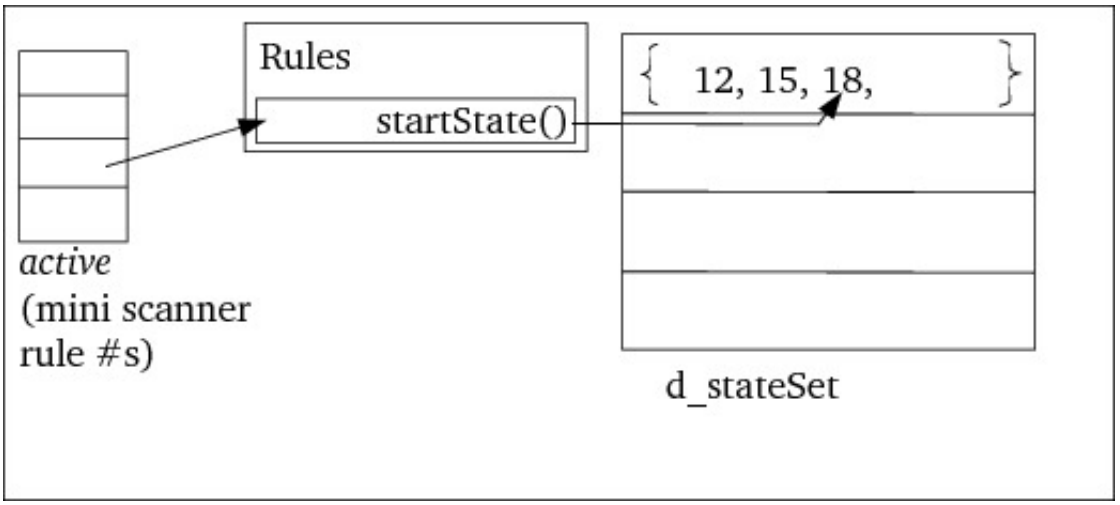

Figure 15: Creating the start states for row 0

Next, the e-closure of this initial set of states is computed. The e-closure algorithm is described in ASU's dragon book (1986, figure 3.26). It essentially adds all states that can be reached from each element in the current set of states on an empty transition. The e-closure is computed by `States::eClosure`.

At this point there is an element in `DFA::d_stateSet`, but not yet an element in `DFA::d_row`. By adding a `DFARow` (see section 6.14) to `d_row` we associate a `DFARow` with an initial set of states.

Once the new DFA has been added to `d_row` its transitions are probed by `DFARow::transitions` (see section 6.14).

DFA::keepUniqueRows

Having determined the transitions `build` proceeds to remove implied/identical rows, calling `DFA::keepUniqueRows`. This latter function visits each of the rows of the DFA, testing whether an existing row has the same transitions and final state information as the current row. `Same transitions' means that the transitions of the current (under inspection) row are present in an earlier row; `same final state information' means that the current row is a final state for the same rule(s) as an earlier row. In such situations the current row can be merged with the earlier row, keeping the earlier row. The (later) row can then be removed as transitions from it are identical to those from the earlier row. This may happen, as the NFA construction algorithm may define more empty edges than strictly necessary, sometimes resulting in additional rows in the DFAs. As an example, consider the pattern `(a|ab)+/(a|ba)+`, producing the DFA

Input Chars

StateSet a b Final Accept

0	1			
1	2	3		
2	2	3	0:1	
3	4	5		
4	2	3	0:2	
5	6			0:++1
6	7	5	0	
7	7	5	0	

Rows 6 and 7 are identical, as are rows 2 and 4. For row 4 the (erroneous, if |TAIL| should be as short as possible) |TAIL| = 2 is shown, resulting from `aba` being interpreted as HEAD: `a` and TAIL: `ba`.

But when |TAIL| should be minimized `aba` should be interpreted as HEAD: `ab` and TAIL `a`, resulting in transitions 0 -> 1 -> 3 -> 2, with |TAIL| = 1. This happens when row 4 is merged to row 2. Having merged the rows, former transitions to the now removed rows must of course be updated to the merging row. So row 3 must transit to 2 when receiving input symbol `a`. The member `shrinkDFA` handles the shrinkage of the DFA. In this example the final DFA becomes:

	Input Chars			
StateSet	a	b	Final	Accept
0	1			
1	2	3		
2	2	3	0:1	
3	2	4		
4	5			0:++1
5	5	4	0	

keepViableTailCounts

Tail counts of rules which cannot be matched (i.e., for rules not having the `viable` attribute set) are removed by this function. Rule's `TailCount` elements were determined by the `DFARow`'s constructor (see sections 6.14 and 6.11).

computeTailCounts

Finally, any remaining `TailCount` elements are processed by `computeTailCounts`. Each `TailCount` element is processed as follows (cf. figure 16)

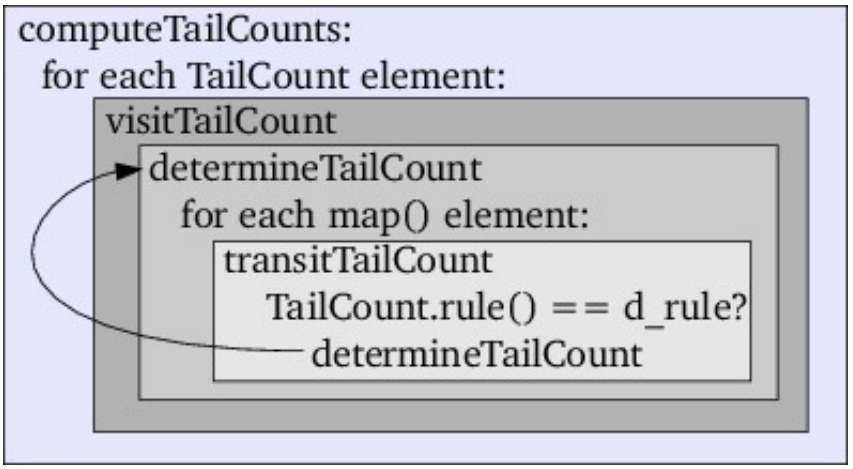

Figure 16: Computing TailCount values

- `visitTailCount`: initializes `d_sawACCEPT` to `false` and `d_row` to the `TailCount`'s rule index, and then calls `determineTailCount` to compute the tail sizes for the various rows of the DFA.
- `determineTailCount`: calls `setTailCount` to determine the action to take for the `TailCount` object that's passed to it. The logic behind this function's decision is shown in section 6.13.1. If the current tail count is computed as the previous tail count + 1 (i.e., COUNT is set), then the computed value may conflict with a previously computed value (cf. section 6.15.3). In that case the count is set to zero, and a warning is issued. The warning is generated by `setNextTailCount`, called from `setTailCount`. This procedure is subject to change in future releases of `flexc++`.
  If the `TailCount` values were updated then `determineTailCount` proceeds by calling `transitTailCount` for each of the row's transition map's elements.
- `transitTailCount`: if the destination row of a transition has a `TailCount` element referring to the row stored in DFA's `d_row`, then the tail count for that row is determined by (indirect recursively) calling `determineTailCount`.

6.13.1: The logic used in 'setTailCount'

The following table shows the decisions/actions to make/perform for the various combinations of `d_sawAccept` and `TailCount` flag values. The flag `PROCESSED` is defined in `TailCount` as well and is set once a `TailCount`'s COUNT or INCREMENTING `d_flag` has been determined.

```
See also setTailCount.cc

Flag Logic (all alternatives are ignored if PROCESSED has been set as well).

 d_sawAccept   PRE     ACCEPT  POST
-------------------------------------------------------------------------
initialize count for rows representing ACCEPT for the 1st time:

 2   0         0       1      0          - set COUNT, d_accCount = 0,
                                             d_sawAccept = true
 3   0         0       1      1          - set COUNT, d_accCount = 0
                                          d_sawAccept = true
 6   0         1       1      0          - set COUNT, d_accCount = 0,
                                          d_sawAccept = true
 7   0         1       1      1          - set COUNT, d_accCount = 0,
                                          d_sawAccept = true
```

```
------------------------------------------------------------------------
set INCREMENTING for rows only representing POST states:

 1   0          0        0      1       - set INCREMENTING
 9   1          0        0      1       - set INCREMENTING
------------------------------------------------------------------------
increment the count for rows representing PRE/ACCEPT and POST states

 5   0          1        0      1          - set COUNT
                                            d_accCount = previousCount + 1
11   1          0        1      1          - set COUNT
                                            d_accCount = previousCount + 1
13   1          1        0      1          - set COUNT
                                            d_accCount = previousCount + 1
15   1          1        1      1          - set COUNT
                                            d_accCount = previousCount + 1
------------------------------------------------------------------------
only PRE/ACCEPT states, and ACCEPT has already been seen: no action
10    1         0        1      0       - no action
14    1         1        1      0       - no action
------------------------------------------------------------------------
only PRE states: no action
 4    0         1        0      0       - no action
12    1         1        0      0       - no action
------------------------------------------------------------------------
these states cannot occur with LOP rules as LOP rules either set PRE, ACCEPT
or POST
 0    0         0        0      0       -       no action (does not occur)
 8    1         0        0      0       -       no action (does not occur)
------------------------------------------------------------------------
```

6.14: The rows of the DFA: DFAROW

Rows of DFA matrices relate (ranges of) input characters to rows (states) to transit to (see figure 17).

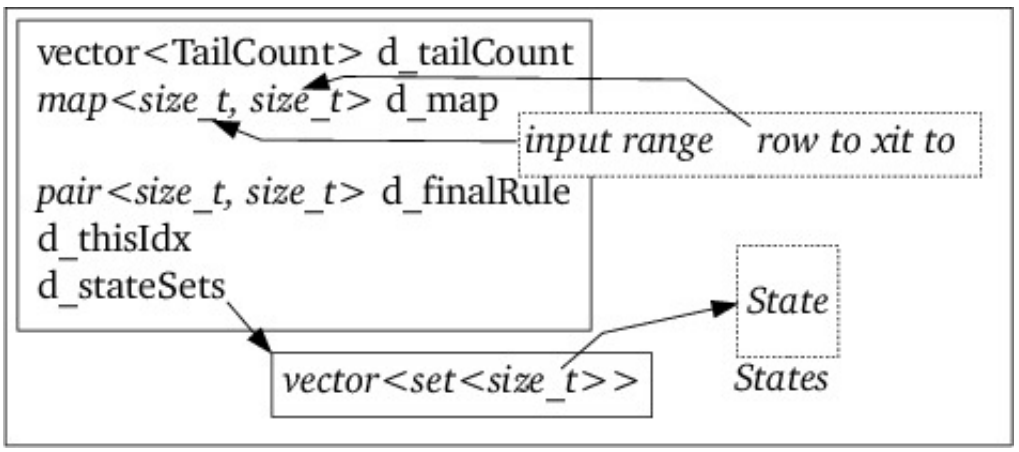

Figure 17: The data of the class DFARow

The DFARow's row specification itself is defined by a set of states accessible as d_stateSets[d_thisIdx]. The values of this set define the (e-closure of the) states of this row (see section 6.13).

Each of these states may have a transition to another state when observing a character from one of the input ranges. These states to transit to in turn define a set of states.

Transitions from this row are probed for each of the character ranges (in Ranges, see 6.10) by DFARow::transitions. See section 6.14 for a description of this function. What it boils down to: transitions may add new elements to d_stateSet, causing the iteration to continue until eventually there are as many d_row elements as there are d_transitions elements.

If this set of states to transit to is not yet present in the DFA then a new row is added to the DFA. Adding rows is the task of DFA::build, defining and adding new sets of States is the responsibility of DFARow::transition.

- d_tailCount: if the current row is visited while recognizing a rule using the LOP then d_tailCount contains a TailCount element for that rule. See section 6.15 for a description of the used algorithm and section 6.11 for a description of the class TailCount.
- d_map: the DFARow's unordered_map d_map defines the relationship between an input character range (the map's key) and the row to transit to when a character from that input range has been observed.
- d_finalRule: the elements of the pair d_finalRule are indices of the Rules for which this DFARow represents a final state. The *first* element of the pair refers to a rule whose RE starts at BOL, the *second* element refers to a rule whose RE does not explicitly start at BOL. When a value equals UINT_MAX then the current row is no final state for such a rule (BOL or not BOL).
- d_thisIdx: the index of the current DFARow element in its DFA matrix.
- d_stateSets: this data member is a pointer to a vector containing sets of states represented by the rows of the DFA. Element d_stateSets[d_thisIdx] is the set of states represented by the current DFA row.

6.15: Patterns using the lookahead operator (LOP)

6.15.1: The LOP handling algorithm

In this section we describe the algorithm that is used to analyze REs using the LOP. It would be nice if at some point we could prove its correctness. Some of its steps can be shown to be correct, but that doesn't hold true for other parts of the algorithm, which may or may not be correct.

First we'll provide examples showing how the RE can be analyzed, concentrating on various complexities encountered with LOPs. At the end a description of the current algorithm is given.

Comment and suggestions for improvement are (more than) welcome!

Consider the RE shown in figure 18:

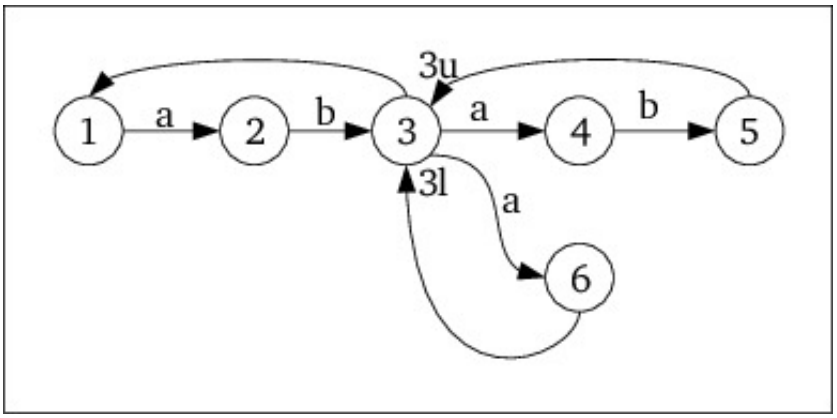

Figure 18: The RE (ab)+/(ab)+|a+

The labels *3u* and *3l* represent the upper and lower routes from state 3. Coming from state 5 the only acceptable route is via *a* to 4, coming from 6 an *a* takes you back to state 6. This representation greatly simplifies the NFA, avoiding many empty transitions. When in state 3 paths continue back to state 1, or to states 4 and 6.

Before constructing the DFA it's a good idea to concentrate on the transitions themselves, as this simplifies the DFA construction. The transitions from each NFA state are shown in the following table:

NFA transitions for `(ab)+/(ab)+|a+`

State	a	b
1	2	
2		1,3
3	2,3l,4,6	
3u	4	
3l	3l,6	
4		3u,5
5	4	
6	3l,6	

From the above table the DFA can now easily be constructed. Starting at row 0, representing all states accessible from NFA state 1 determine the union of states reacheable from the current set on input characters `a` and `b`. NFA states 3u and 3l are *swallowed* by state 3 if these states are elements of a set also containing state 3. This holds true in general. Here is the initial DFA:

DFA construction (part I)

Row	NFA states	a	b
0	1	1	

1	2		2
2	1,3	3	
3	2,3l,4,6	4	5
4	3l,6	4	
5	1,3,5	3	

While in states 1, 2 or 3 the LOP has not yet been passed and we remain in the *head* (`H`) of the RE. A transition from state 3 to either states 4 or 6 takes us into the RE's *tail* (`T`). Following the DFA's transitions find the first row where state 3, the state corresponding to the LOP, is reached. This is DFA row 2 (`DFA[2]`). This row receives a *tail* length (`|T|`) of 0: no LA-tail has as yet been collected at this point.

Continuing from this row transitions may enter the LA-tail. So transitions from here into the tail states 4, 5 and 6 must increment the tail's size. While in the tail each transition increases `|T|`. So, from `DFA[2]` the transition on `a` to `DFA[3]` must increment `|T|` to 1.

From `DFA[3]` transitions are possible to `DFA[4]` (on `a`) and `DFA[5]` (on `b`). Concentrating on the latter: this is yet another possible step into the tail, so it is associated with `|T| == 2`. However, it's also possible that we've retraced our steps here, and have in fact returned back to states 1 and 3. The continuation from `DFA[5]` on `a` implies that we've done so.

Why? This is because an additional rule applies to the *tail* computation: its length must always be as short as possible. Having seen `ab` (taking us to `DFA[2]`); then another `ab` (taking us to `DFA[5]`); then *another* `a` allows us to merge the second `ab` into the RE's head, considering the final (so far) `a` to be the first element of the tail again. This is also shown by the transition count: moving from `DFA[5]` to `DFA[3]` takes us (correctly) back to `|T| == 1`.

When we're in `DFA[3]` an `a` rather than a `b` might be observed. This takes us to `DFA[4]`. From here there's no going back to pre-3 states: `DFA[4]` consists of NFA states 3l and 6, and both are beyond state 3. We've reached the `a+` alternative of the RE's tail, and from now on *any* `a` will increase `|T|`. This allows us to formulate the general rule: *once a DFA state is reached merely consisting of post-LOP states, there is no fixed tail-count, but every transition increases* `|T|`. Using a `+` to indicate an incrementing DFA row we associate a `+` with `DFA[4]`.

What happens if, e.g., in `DFA[4]` or `DFA[2]` a `b` is encountered? Then no further continuations are possible. What to do in those situations depends on the set of NFA states defining the DFA rows.

`DFA[2]`'s set of NFA states does not contain state 5 or 6, which are the RE's end states. In that case the RE hasn't been matched and the resulting scanner must revert to some sort of recovery procedure (which is, by the way, the *echo first* procedure, which is not relevant for this section). `DFA[3], DFA[4]` and `DFA[5]`, however, *do* contain a final NFA state and are therefore considered *Final* DFA states. When no more input characters can be matched and a final DFA state has been reached we flag *success*, and have matched the RE. `|T|` is as computed so far (either a fixed value or a variable (incrementing) value when we're in a `F` and `+` DFA state.

The following table shows the full `DFA`, including *accept counts* (`Acc`) and *final state* (`Final`) indicators:

DFA construction (part II)

Row	NFA states	Acc	a	b	Final
0	1		1		
1	2			2	
2	1,3	0	3		
3	2,3l,4,6	1	4	5	F
4	3l,6	+	4		F
5	1,3,5	2	3		F

6.15.2: LOP handling: a more extensive example

In this section we present the slightly more complex RE `((ab|ba)+/(ab+|b+a)+)`, following the steps leading up to its DFA. The NFA representing the RE is provided in figure 19:

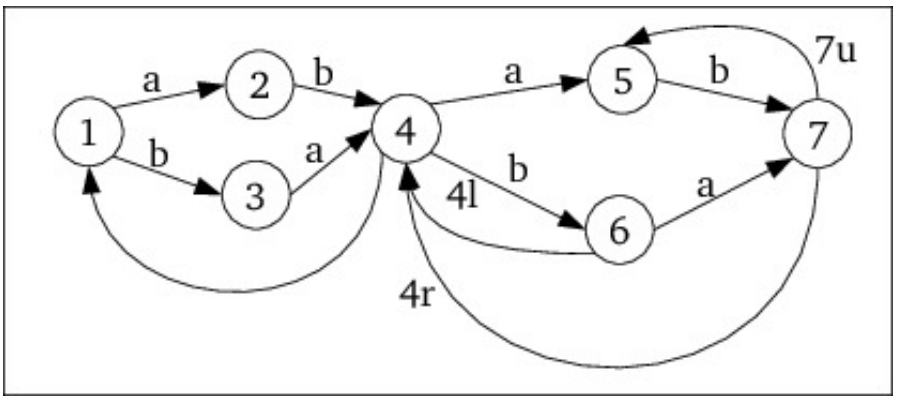

Figure 19: The RE (ab|ba)+/(ab+|b+a)+

Like before, some edges have restricted use. From state 4 state 1 can immediately be reached, without further input; state 5 can be reached on a, state 6 can be reached on b. But edge 4l can only be used from state 6 to return to state 6 on a b input character. Coming from state 5 edge 7u is immediately available, but 7u is not available coming from state 6. Arriving at state 7 the lower edge (labeled 4r) can always be taken, but from its destination (state 4) only the right edges are possible (to either states 5 or 6.

The corresponding state transition matrix is:

NFA transitions for `(ab|ba)+/(ab+|b+a)+`

State	a	b
1	2	3
2		1,4
3	1,4	

4	2,5	3,4l,6
4r	5	4l,6
4l		4l,6
5		4r,5,7
6	4r,7	4l,6
7	5	4l,6
7u		4r,5,7

Now the DFA can be constructed:

DFA for (ab|ba)+/(ab+|b+a)+

Row	NFA states	Acc	a	b	Final
0	1		1	2	
1	2			3	
2	3		3		
3	1,4	0	4	5	
4	2,5	1		6	
5	3,4l,6	1	7	8	
6	1,4,5,7	2	4	9	F
7	1,4,7	2	4	5	F
8	4l,6	+	10	8	
9	3,4r,5,6,7	3	6	11	F
10	4r,7	+	12	8	F
11	4r,5,6,7	+	13	11	F
12	5	+		8	
13	4r,5,7	+	12	11	F

6.15.3: LOP handling: a problem case

Consider the RE shown in figure 20: `(ab|a)+/(ba|a)+`.

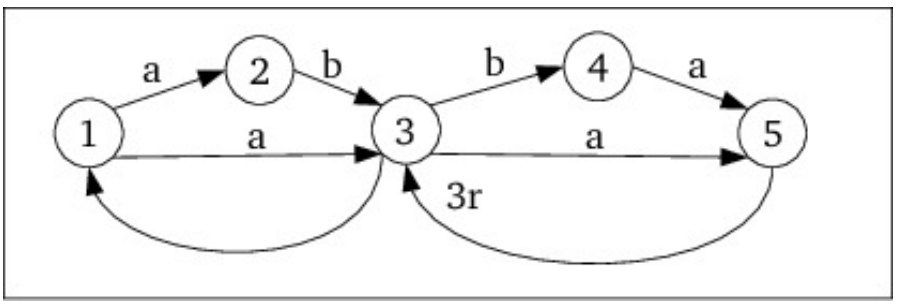

Figure 20: The RE (ab|a)+/(ba|a)+

Its state transition matrix looks like this:

NFA transitions for `(ab|a)+/(ba|a)+`

State	a	b
1	1,2,3	
2		1,3
3	1,2,3,5	4
4	3r,5	
5	3r,5	4

Now we're going to construct its DFA. We're doing this row by row. The first two rows are easy:

DFA for (ab|a)+/(ba|a)+

Row	NFA states	Acc	a	b	Final
0	1		1		
1	1,2,3	0	2	3	

`DFA[2]` is constructed next: on `a` we reach one of the states 1, 2, 3 or 5, and it can easily be verified that an `a` in that state keeps us in `DFA[2]`.

What about `DFA[3]`? A `b` in `DFA[1]` takes us to state 1, 3. or 4. Since state 5 takes us to state 4 on a `b`, the `b` transition of `DFA[3]` equals the `b` transition of `DFA[2]`.

Adding `DFA[2]` and `DFA[3]` to our previous DFA results in:

DFA for (ab|a)+/(ba|a)+

Row	NFA states	Acc	a	b	Final
0	1		1		
1	1,2,3	0	2	3	
2	1,2,3,5	1	2	3	F
3	1,3,4	1/2	2	4	

Looking at the Acc entry of `DFA[3]` we see that our tail computation algorithm fails. A `b` character in `DFA[1]` takes us to `DFA[3]` with `|T| == 1`, but an `a` character in `DFA[1]` followed by a `b` character takes us via `DFA[2]` to `DFA[3]` as well, implying `|T| == 3`. Bad news!

Before looking in detail at what's going on here let's first complete the DFA. We need two more rows:

DFA for (ab|a)+/(ba|a)+

Row	NFA states	Acc	a	b	Final
0	1		1		
1	1,2,3	0	2	3	
2	1,2,3,5	1	2	3	F
3	1,3,4	1/2	2	4	
4	4	+	5		
5	3r,5	+	5	4	F

Analyzing the problem it is soon clear that the problem is caused by the two rows having equal transitions. They cannot be merged (and merging wouldn't solve the problem either), because `DFA[1]` is not a final state while `DFA[2]` is.

But we *can* conclude something here. In `DFA[2]` when there is no continuation we are in a final state which can only be reached after recognizing a final `a`. In that case, using the `shortest tail' guideline, `|T|` must clearly be equal to 1. This also holds true when multiple `a` characters are encountered. No matter how many `a` characters, once we're in `DFA[2]` we stay there and `|T|` correctly remains at 1.

An interesting event happens when reading `b` characters. Following a `a` characters and a `b` we reach `DFA[3]`. As we've seen, an `a` there takes us back to `DFA[2]`. But what about a `b`? In this case we can actually conclude that we *must* have reached the tail. The only way to have two consecutive `b` characters is by reaching state 4, after which we can only stay in the tail. The transition from `DFA[3]` to `DFA[4]` on `b` illustrates this: `DFA[4]` is a `+` Acc state, as is `DFA[5]` and once we're in `DFA[4]` or `DFA[5]` there's no escape from the tail: they are only `+` Acc states.

So we *know* that the *first* time we reach state 4 `|T|` *must* be 1. All earlier patterns can be accomodated by the RE's head. Backtracking from `DFA[4]` it is clear that it can only be reached (for the first time) via a transition from `DFA[3]`, which is our conflicting state.

But `DFA[3]` is conflicting *only* because the algorithm itself has no knowledge about the `shortest tail' guidline, and so it considers `DFA[3]` as representing states 1, 3, or 4. But it *can't* be state 4, since `DFA[4]` represents state 4, and there's no transition from state 4 to state 4. Consequently, although the DFA construction algorithm adds state 4 to `DFA[4]` this is *semantically* impossible and thus it can be removed from the set of states represented by `DFA[4]`.

Now `DFA[3]` consists of states 1 and 3, which corresponds to `|T| == 0`. This removes the conflict.

For now it is conjectured that conflicts caused by multiple Acc values for `DFA` rows can be solved by changing their Acc values to 0.

This is no proof, of course. But here's another RE, having two conflicts which are solved by the above conjecture (which the reader may verify for him/herself):

```
(abc|ab|a)+/(cba|ba|a)+
```

Several other, fairly complex REs can thus be handled by our algorithm, which is described in the next section.

6.15.4: The steps of the LOP handling algorithm

Now that we've seen several examples, it's time for a description of the steps of the LOP-handling algorithm. The description provided here concentrates on processing a single RE. In reality there will be multiple REs requiring RE-numbers to be associated with the elements of the Accept count, B/A and Final columns (see below).

- Starting point is the NFA representation of a RE.
- Create a table where each row represents an NFA state and where the columns are the characters (in practice: character ranges) of the input alphabet. The elements of this table are sets of states that can be reached from the current (row) state given a particular input character.
- The DFA consists of rows having several columns.
    - The first column contains sets of NFA states. The first column of `DFA[0]` defines the set of states consisting of the RE's initial NFA state and all states that can be reached from the initial state using an empty transition.
    - Associate `B` with NFA states that are states *before* the LOP, associate `A` with NFA states that are states *after* the LOP.
    - The DFA's last but one column is called the B/A column. It indicates the kind of NFA states that are found in the DFA row's current set. It elements contain
        - `A` if the set merely contains A-type of states;
        - `B` if the set merely contains B-type of states;
        - `B/A` if the set contains A- *and* B-type of states;
    - The second column is the *Accept count* (ACC) column. Transiting from `DFA[0]` it contains 0 for the first row whose set of NFA states contains the NFA state associated with the LOP.
    - Following column two there are as many columns as their are input characters (EOF might be defined as a pseudo input character). Each element indicates the DFA row to transit to given the current row and input character column. If no transition is defined for a particular combination the entry is left blank.
- Each unique set of NFA states defines a DFA row of its own.
- The DFA's final column is called the *Final* column: it is empty except for those rows where the sets of NFA states contain the RE's final state(s). In those cases flag in this column that those rows represent final rows. If no continuation is possible in such rows then a RE has been matched. See also section 6.16.
- Back to the ACC row. Mere `B` rows have no ACC entry. Relevant transitions from the ACC == 0 row reach either B/A or A rows. Consider the RE `a+/a+` represented by the following DFA:

DFA for a+/a+

Row	NFA states	ACC	a	B/A	Final

0	1		1	B	
1	1,2	0	2	B	
2	1,2,3		2	B/A	F

To reach the final state the input must reach `DFA[2]`. At this point we've transited 1 step into the tail, and so `|T| == 1`. In theory subsequent a's could be used to enlarge the tail, but applying the `shortest tail' heuristic results in a `|T| == 1`, no matter how many a-characters follow. Transitions from a row having an ACC count to a B/A type of row increment the ACC count of the originating row. The above DFA therefore becomes:

DFA for a+/a+

Row	NFA states	ACC	a	B/A	Final
0	1		1	B	
1	1,2	0	2	B	
2	1,2,3	1	2	B/A	F

Consequently, once the RE is accepted `|T|` has a fixed length of 1.

- B/A type-A DFA rows merely consist of states in the RE's tail. Each transition here must necessarily increment the tail. Mere A-type of B/A rows therefore receive an ACC +, indicating that each transition into this row increments `|T|`.
- When transitions return to an earlier DFA row having a lower ACC then this indicates that a final sequences of characters, previously considered to be in the RE's tail can be merged with the HEAD: the earlier row's ACC will again be used. RE `(ab)+/(ab)+` is an example:

DFA for (ab)+/(ab)+

Row	NFA states	ACC	a	b	B/A	Final
0	1		1		B	
1	2			2	B	
2	1,3	0	3		B	
3	2,4	1		4	B/A	
4	1,3,5	2	3		B/A	F

In this DFA an `a` in row 4 takes us back to row 2, merging a previously read `ab` with the head and interpreting the just-read `a` as the first character of the tail, now again having length 1.

- (Conjecture) In some REs (see section 6.15.3) DFA rows receive multiple ACC values. This is caused by a theoretically possible transition to an A-type state, which is in practice prevented by the `shortest tail' heuristic. An ACC value 0 must be assigned to rows having multiple ACC values. This only applies to B/A-type rows, not to mere A-type rows.

6.16: Final states and ^xyz vs. xyz patterns

Patterns starting with ^ and not starting with ^ are different rules. A rules like `^a` and `a` can both be

matched, depending on whether the a is found at begin of line (BOL) or not.

Generally, when two DFA rows represent the final states of multiple REs (in the extreme case: rule 1 could be a and rule 2 could also be a) then the lexical scanner generator abandons rule 2 for that DFA row.

If one of the two rules start with a BOL-marker (^), however, then then DFA row can be used for both rules. This is the single exception to the rule that a DFA row can only be associated with a single RE's final state.

This affects the implementation of the DFA rows: it's only necessary to store two indices indicating the rules for which the DFA row represents a final state.

By default the indices are 0, indicating that the DFA row is not a final state for an RE. At most two rules can be used in a particular state: a rule starting at BOL and a rule not starting at BOL. The `first` element of the pair indicates the index for rules starting at BOL, the `second` element of the pair is used for rules not starting at BOL. When a DFA state could be the final state for multiple rules of the same type (BOL or not BOL), then the rule having the lowest index is used. By default the elements of the `d_finalRule` pair are initialized to `UINT_MAX`, indicating that this row does not represent a final state.

When a regular expression has been matched at a DFA row having two rule indices then the runtime implementation checks it `d_atBOL` member to see if the matched RE started at BOL. If so, the BOL-RE is matched; otherwise the non-BOL rule is matched. See also section 6.18.

6.17: Generating Code

The code generated by `flexc++` must read input, match the input with some regular expressin and return the associated token. A catch-all escape is provided by the implicitly provided rule echoing all non-matched input to the standard output stream.

The class `Generator` handles code generation for the class `Scanner` (or whatever other name is defined). The `Generator`'s constructor currently defines a fixed output file name (`STATICS`) to which all static data are written. This output stream is accessed throught the `Generator::d_out` data member. This setup must eventually be modified.

The generated code consists of four parts:

- The character-table (actually: range-table). See section 6.17.1.
- The DFAs are generated by `Generator::dfas`.

  For each of the DFAs the function `Generator::dfa` is called. Next the information about the Final and Accept counts is written by `Generator::outFinAcs` and finally the DFA entry points are generated by `Generator::dfaEntryPoints`.

  - The function `Generator::dfa` receives as its first argument a `DFAs::Pair` (see figure 13), containing the name of the mini scanner associated with the DFA and the DFA itself. The member function first checks whether the DFA has already been processed. This only happens for `INITIAL`, and the current test is rather blunt by the function `dfa` calling `std::find`.

If this DFA has not yet entered the `s_dfa[]` array then

- The name of the current mini scanner is stored in `startStates` (to allow checking for repetitive handling);
- The row number of the first row of the current DFA in `s_dfa` is saved in the vector `dfaOffsets`;
- `Generator::dfaRow` is called for each row of the DFA (see figure 17). This latter function writes the vector of DFA rows to transit to (i.e., rows relative to the current DFA's start row, so not the actual rows in `s_dfa`) for each of the character ranges (calling `Generator::dfaTransitions`), followed by the row's `FinAc` data: a value != -1 indicates the rule for which this row is a FINAL state.

  The row's final two values are begin and end indices in s_accept, holding information about a row's accept state. -1 indicates `not an accept state'

- The function `Generator::outFinAcs`

  TO DO

- The function `Generator::dfaEntryPoints`

  TO DO

- The function `Generator::declarations`

  TO DO

- The function `Generator::actions`

  TO DO

6.17.1: The range-table

The character-table translates input characters to ranges. Each input character (by default the 256 extended ascii-characters) is associated with a character *range*. Character *range* indices are then used as column indices of the DFA tables (see sections 6.10 and 6.13).

The function `Generator::charTable` defines in `d_out` the static data member `size_t const ScannerBase::s_ranges[]`. This array has 256 elements, so each character (cast to type `unsigned char`) can be used as an index into this array, returning its range-number.

In addition to real input characters, the scanner may return two pseudo range values: `rangeOfBOL` is the range matching the special `character' `BOL`, returned when a begin-of-line is sensed, and `rangeOfEOF` which is returned when EOF was sensed (e.g., when the `<<EOF>>` rule was used). These BOL and EOF tokens must be returned by `nextChar` when BOL or EOF was sensed, and the DFA's recognizes their ranges. The ranges `rangeEOF` and `rangeBOL` are declared in the scanner class's data members section.

If DFA's don't recognize BOL or EOF characters then the default action is performed: BOL is ignored and EOF results in switching back to the previous stream or in returning token 0 (and ending the scanning-process).

The code generator adds code handing BOL and EOF to scanners using these pseudo characters. The code is left out of the generated scanner if these pseudo characters are not used.

Range tables are generated by `generator/chartable.cc`.

6.17.2: The DFAs

The function `Generator::dfas`, defined in `generator/dfas` defines defines in `d_out` the static data member `int const ScannerBase::s_dfa[][dfaCols()]`, where `dfaCols` returns the number of columns of the DFA matrices.

All DFAs are accessed through this single `s_dfa` matrix. Each individual DFA starts at a specific row of `s_dfa`. The first DFA to be written is associated with the `INITIAL` scanner: `INITIAL` is always defined and contains all rules not explicitly associated with a mini scanner.

The matrix `s_dfa` contains the rows of *all* DFAs ordered by start state.

The `enum class BEGIN` defines the symbolic names of the start states Its constant `INITIAL` always receives value 0.

Each row contains the row to transit to if the column's character range was sensed. Row numbers are relative to the used DFA. There are as many elements in the rows of the `s_dfa` table as there are character ranges *plus* two. These final elements represent the begin and end indices in the array `s_accept`, holding information about a row's accept state (see the next section).

The *base locations* for the various mini scanners are defined in the static array `s_dfaBase`. Its initial value matches the `INITIAL` scanner, and points to the first `s_dfa` row. Additional values are defined if mini scanners were used and point at the initial rows in `s_dfa` for these mini scanners. Here is an example of a `enum class Begin` and matching `s_dfaBase`:

```
enum class Begin
{
    INITIAL,
    str,
};

std::vector<int const (*)[9]> const ScannerBase::s_dfaBase =
{
    { s_dfa + 0 },
    { s_dfa + 6 },
};
```

The `INITIAL` scanner's dfa rows start at the top, the `str` mini scanner starts at row index 6 of `s_dfa`.

6.17.3: The Final-Accept info array

Indices at the end of the `s_dfa` rows refer to rows in the `s_finAcInfo` array, generated by `Generator::outFinAcs` (file `generator/outfinacs`).

The first index holds the first relevant `s_accept` row, the second index holds the index of the row that is

not relevant anymore. If `begin == end` then this row does not represent an accepting or final state.

The columns of this matrix are labeled `R, I` and `A`, indicate:

- `R`: the rule number that was matched if there is no continuation for the next input-range value. If this rule is matched its action block is executed next.
- `I`: the info-field. Its values are three bit-flags indicated whether the row represents a final row; whether the associated rule uses the LOP; and whether the state is an incrementing accept state for the associated rule. These values can be (values 1 and 5 do not occur):
  - `0`: not a final state, not using the LOP, not an incrementing state;
  - `2`: not a final state, using the LOP, not an incrementing state;
  - `3`: not a final state, using the LOP, an incrementing state;
  - `4`: a final state, not using the LOP, not an incrementing state;
  - `6`: a final state, using the LOP, not an incrementing state;
  - `7`: a final state, using the LOP, an incrementing state;
- `A`: the accept count, which is 0 except for I-values 2 and 6 where it contains a fixed Accept Count (ACC): in the run-time implementation the length of the RE's LOP-tail is (re)set to the ACC value every time the associated row is reached. With incrementing states the RE's tail is incremented each time such a DFA row is reached (see also section 6.15).

  TODO: the next section must be edited according to the above description

Here is an example of rules, transition matrix and `s_finAcInfo` array: There are three rules, two using the LA operator:

```
abd+/def+
abd+/d
abd+
```

The matching transition matrix shows that states 3 through 6 have FinAc vectors associated with them (discussed below):

```
{-1, 1,-1,-1,-1,-1,-1,-1,-1,   0, 0},  // 0
{-1,-1, 2,-1,-1,-1,-1,-1,-1,   0, 0},  // 1
{-1,-1,-1,-1, 3,-1,-1,-1,-1,   0, 0},  // 2
{-1,-1,-1,-1, 4,-1,-1,-1,-1,   0, 1},  // 3
{-1,-1,-1,-1, 4, 5,-1,-1,-1,   1, 3},  // 4
{-1,-1,-1,-1,-1,-1, 6,-1,-1,   3, 4},  // 5
{-1,-1,-1,-1,-1,-1, 6,-1,-1,   4, 5},  // 6



{-1, 1,-1,-1,-1,-1,-1,-1,-1,   0, 0},  // 0 a
{-1,-1, 2,-1,-1,-1,-1,-1,-1,   0, 0},  // 1 b
{-1,-1,-1,-1, 3,-1,-1,-1,-1,   0, 0},  // 2 d
{-1,-1,-1,-1, 4,-1,-1,-1,-1,   0, 1},  // 3 abd+: match 2, full length
                                         { 2,-1,-1, 0},

{-1,-1,-1,-1, 4, 5,-1,-1,-1,   1, 3},  // 4 abcd/+d
                                           no cont.: match rule 2
                                           cont: accept count = 1
```

```
                                        { 1, 1, 1, 0},
                                        { 2,-1,-1, 0},

    {-1,-1,-1,-1,-1,-1, 6,-1,-1,   3, 4},  // 5
                                            no final state, acc.
                                            count: 2, incrementing
                                        { 0,-2, 2, 1},

    {-1,-1,-1,-1,-1,-1, 6,-1,-1,   4, 5},  // 6

                                        { 0,-1,-1, 0},
```

The FinAc matrix is:

```
{ 2,-1,-1, 0},
{ 1, 1, 1, 0},
{ 2,-1,-1, 0},
{ 0,-2, 2, 1},
{ 0,-1,-1, 0},
```

- Its first row indicates what may happen in state 3: if there's no continuation in this state (i.e., after recognizing a, b, d then rule 2 is matched, recognizing `abd`).
- Its second and third rows indicate what may happen in state 4. This state is reached after matching `abdd`. While more d's are matched we stay in state 4. If an `e` character is read, transit to state 5. Otherwise, there is a choice between matching rules 1 or 2. Since the lexer uses greedy matching it will match rule 2, and matching rule 1, having a tail of 1 character is ignore.
- State 5's FinAc vector is `{ 0,-2, 2, 1}`. Here we've recognized `abd+de`. It indicates that this is not a final state, and that the current accept count equals 2, and that this is an incrementing accept state.

All rows are final states for rule 0. The first row has an accept count of 1 and is a final state if there's no continuation possible from the matching state.

The second row is an incrementing accept state. While transitions are in accept states the accept count, starting at 1, is incremented.

The third row represents a plain accepting state. No LA operator.

6.18: Run-time operations

The `lex` function works as follows:

- All its variables have been initialized at construction time. In particular, initially the run-time variable `d_atBOL` is set to `true`.
- In a loop (until EOF) characters are retrieved until a rule is matched (see below)
- Once a rule has been matched its matching code block is executed.

Processing characters:

The next character from the input is obtained in `d_char`. At the beginning of the character processing loop `d_char` must have received a value. At EOF the character is given the (pseudo) character value `AT_EOF`.

Next, the associated range (`d_range`) is determined as `s_ranges[d_char]`.

The variable `d_dfaBase` points at the dfa currently used. The variable `d_dfaIdx` holds the index of the current row in the current dfa.

The expression

```
nextIdx = (d_dfa + d_dfaIdx)[d_range];
```

provides the next `d_dfaIdx` value. If unequal -1 do:

```
d_dfaIdx = nextIdx;
d_char = nextChar();
```

If equal -1 then there is no continuation for the current rule.

Depending on the current state and action several action types can be returned by the run-time function `actionType__`:

- `CONTINUE`: there exists a continuation from the current state/range combination;
- `MATCH`: a final state has been reached and there's no continuation possible. In this case a rule has been matched
- `ECHO1ST`: no continuation is possible for the current range, and no final state was reached, the input buffer contains one or more characters.
- `ECHOCHAR`: no continuation is possible for the current range, no final state was reached and the input buffer is still empty
- `RETURN`: input exhausted, `lex` returns 0.

The following table shows how actions are determined by `actionType__`. The function `actionType__` starts it tests at the first row of the table, and stops as soon as a condition hold true, returning the action type:

Action determined by `actionType__`

	Input Character	
Performed Test	not AT_EOF	AT_EOF
transition OK	CONTINUE	CONTINUE
match found	MATCH	MATCH
buffer size	ECHO1ST	ECHO1ST
buffer empty	ECHOCHAR	RETURN

For each of the returned action types actions must be performed:

- `CONTINUE:` store the input character (not being `AT_EOF`) in the buffer, continue scanning
- `MATCH:` push the just read char back to the input, copy the non-pseudo character (i.e., other than AT_EOF) to the match-buffer, set `d_atBOL` to `true` if the last character in the `d_matched` buffer is `'\n'`, execute the rule's actions, continue scanning (but the action may end `lex`'s call).
- `ECHO1ST:` echo the 1st char. of the buffer to stderr, set `d_atBOL` to `true` if that character equals `'\n'`, otherwise to `false`, return all but the first character in the buffer back to the input, continue scanning.
- `ECHOCHAR:` echo the just read character to stderr, set `d_atBOL` to `true` if that character equals `'\n'`, otherwise to `false`, continue scanning.
- `RETURN:` the function `lex` returns with token value 0.

The data member `d_atBOL` is also set to `true` at a stream-switch: when switching to another stream or returning to a stacked stream. This handles the borderline case where a file's last line is not properly terminated with a `'\n'` character.

> TODO: Subsequent text of this section may require extensive editing/rewriting

In *all* case where the last two elements of the current `s_dfa` row are unequal special actions must be performed. Using the last but one's value as `d_finAcIdx` do as follows:

- If there is a continuation for the current character range (so `(d_dfaBase + d_dfaIdx) [d_charRange]` is unequal -1) then inspect the `s_finAcInfo + d_finAcIdx`'s A and I values.
  - If I equals 1 store A's value and increment it at each next transition until a rule has been matched. This determines the LA tail-length;
- Otherwise (no continuation):
  - If I equals 1 then the A count incremented so far is the LA's tail length.
  - If I equals 0 and A is non-negative then A's value is the LA's tail length
  - If F is positive then it is the accept count of the current rule

    > TODO: maybe this and the previous option are mutually exclusive and only one of them occurs in practice.

    If F equals -1 then accept the matched text as is.
    If it equals -2 we're not at a final state. If there's no continuation, push back all but the first character, echo the 1st char and continue scanning with the INITAL scanner at state 0.

If a final state has been reached then possibly push back the LA tail and execute the actions of the rule that was matched. Using the currently active miniscanner, start matching the next rule.

6.19: Generating code

Code is generated by an object of the class `Generator`. The generator generates the following files (using default names, all defaults can be overruled by directives and options):

- `Scannerbase.h.` This file defines the scanner's base class. It is always rewritten and declares data members used by the scanner.

- `Scanner.h`. This file represents the generated scanner's interface. It inherits from `ScannerBase` and is written only if not existing.
- `lex.cc` This file contains the implementation of the scanner function and any support functions it may need. This file is always rewritten.

Each of these three files has a skeleton, found in `/usr/share/flexc++` which is copied to the generated code, embedding the following elements into the generated files:

- The *static data* are inserted into the generated `lex.cc` file. Static data are not used by other components of the scanner, and could be embedded in the anonymous namespace;
- The *actions* are inserted into the generated `lex.cc` as well.
- The *declarations* are inserted into the generated `Scannerbase.h` file.

The member `genLex` generates the lexer file `lex.cc`. It recognizes `$insert` lines starting in column 1 and performs the matching action, inserting appropriate elements into the generated file.

Other generating members act similarly:

The member `genBase` generates the scanner base file `Scannerbase.h`, the member `genScanner` generates `Scanner.h`.

6.20: Reflex: refreshing flexc++

When a new **flexc**++ release is prepared (resulting from a minor or major upgrade or even rewrite) several steps should be performed before the new version is released. **Flexc**++ should at least be able to regenerate its own lexical scanner function `lex()`.

In **flexc**++'s distribution the shell script `reflex` is found. In an interactive process it performs the required steps by which **flexc**++ recreates its own scanner. The script assumes that a new binary program, representing the latest changes, is available as `./tmp/bin/flexc++`, relative to the directory in which the `reflex` script is found. When creating `flexc++` with the provided `build` script, these assumptions are automatically met.

The `reflex` script then performs the following steps:

- It optionally creates and always cleans a working directory `./self`.
- It copies `scanner/lexer` to `./self`, allowing **flexc**++ to create a separate `lex.cc` and `Scannerbase.h`.
- In `./self` it executes

  ```
  ../tmp/bin/flexc++ -S ../skeletons lexer
  ```

  by which the new **flexc**++ program creates a scanner from its own scanner specification file. At this point realize that the new **flexc**++ program still uses a `lex()` function that was created by an older scanner generator.

- Before taking this step *make sure that the original `lex.cc` and `scannerbase.h` files were properly backed up*.
  The files `lex.cc` and `Scannerbase.h`, created in the previous step are now copied to the `./scanner` directory.

- All of the sources of the `Scanner` and `Parser` classes as well as `flexc++.cc` itself must now be recompiled, unless absolutely no changes were made to the `Scanner`'s internal data organization were made, in which case `build program` will only recompile the modified source file `./scanner/lex.cc`.
  At this point a new **flexc++** program has been created, using a `lex.cc` implementation generated by the latest release of **flexc++**.

- Once again execute in `./self`

  ```
  ../tmp/bin/flexc++ -S ../skeletons lexer
  ```

  by which **flexc++** creates a scanner from its own scanner specification file, now using its own generated implementation of the `lex()` function in `lex.cc`.

- If `diff ./scanner/lex.cc self/` only shows different time stamps and if `diff ./scanner/Scannerbase.h self/` also only shows different time stamps, then **flexc++** could recreate its own matching function and a stable new release has been created.

- Table of Contents
- Previous Chapter
