Queχ
A Mode Oriented Directly Coded Lexical Analyser Generator
Version 0.11.3

(C) 2006, 2007 Frank-René Schäfer
fschaef@users.sourceforge.net

August 11, 2007

Contents

1. Introduction

The queχ program generates a lexical analyser that scans text and identifies patterns. The result of this lexical analysis is a list of *tokens*. A token is a piece of atomic information directly relating to a pattern, or an *event*. It consists of a type-identifier, i.e. the *token type*, and content wich is extracted from the text fragment that matched the pattern.

Figure 1.1 shows the principle of lexical analysis. The lexical analyser receives a stream of characters "if(x> 3.1) { ..." and produces a list of tokens that tells what the stream signifies. A first token tells that there was an `if` statement, the second token tells that there was an opening bracket, the third one tells that there was an identifier with the content `x`, and so on.

In 'real' programming languages the token stream is received by a parser that interpretes the given input according to a specific grammar. However, for simple scripting languages this token stream might be treated immediately. Using a lexical analyser generator for handcrafted ad-hoc scripting languages has the advantages that it can be developped faster and it is much easier and safer to provide some flexibility and power. This is to be demonstrated in the following chapters.

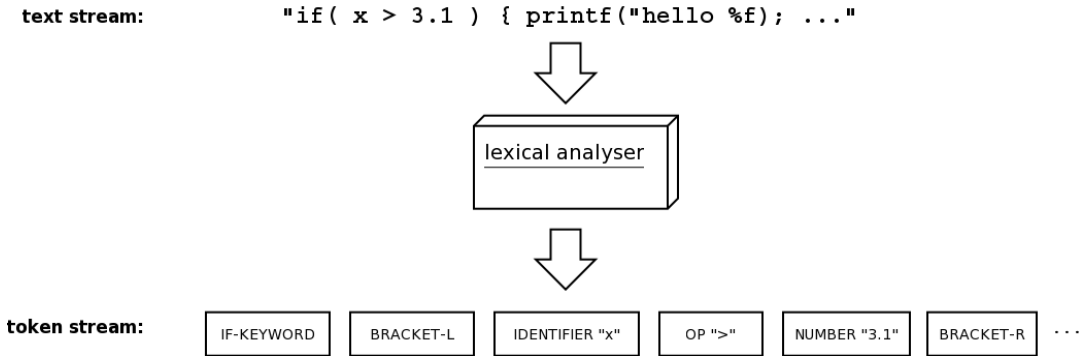

Figure 1.1.: The process of lexical analysis

The following features distinguish queχ from the traditional lexical analysers such as lex or flex:

- *Ease*. A simple as well as a complicated lexical analyzer can be specified in a very elegant and transparent manner. Do not get confused over the set of features and philosophies. If you do not use them, then simple skip the concerning regions in the text. Start from the ready-to-rumble examples in the ./DEMO subdirectory.
- A generator for a *directly coded lexical analyzer* featuring pre- and post-condtions. The generated lexical analyzer is up to 2.5 times faster than an analyzer created by flex/lex.

- Sophisticated *lexical modes* in which only exclusively specified patterns are active. In contrast to normal 'lex' modes they provide the following functionality:
  - *Inheritance relationships* between lexical analyser modes. This allows the systematic inclusion of patterns from other modes, as well as convinient transition control.
  - *Mode transition control*, i.e. restriction can be made to what mode a certain mode can exit or from which mode it can be entered. This helps to avoid that the lexical analyser drops by accident into an unwanted lexical analyses mode.
  - *Mode transition events*, i.e. event handlers can be defined for the events of exiting or entering from or to a particular mode.
  - *Indentation events*, i.e it is possible to provide an event handler for the event of the first appearing non-whitespace in a line. This event handling happens quasi-paralell to the pattern matching.
- A default general purpose *token* class. Additionally, Queχ provides an interface to run the lexical analyser with a user-defined token class.
- A *token queue* so that tokens can be communicated without returning from the analyser function. The token queue is a key for the production of '*implicit tokens*', i.e. tokens that do not relate directly to characters in an analysed character stream. Those tokens are derived from context. This again, is a key for defining redundancy reduced languages.
- Automatic *line* and *column numbering*. These features can be turned off, in case that one fears performance loss. It can be determined excactly from which line and which column a recognized pattern started and where it ended.

The present text first explains briefly the basic concepts of lexical analysis in queχ. Here, a short review is given on lexical analysis, but then it concentrates on the introduction of the features mentioned above. The subsequent chapter dicusses a simple example of a complete application for lexical analysis. Finally, the last chapter elaborates on the formal usage of all features of queχ.

1.1. Installation

Before, one can start with the installation of queχ, one has to make sure that Python (http://www.python.org) and flex (http://flex.sourceforge.net) are both installed. Most linux distributions provide handy rpm packages of those two on CD. Then, there are only two things to do:

1. Extracting the file `quex-x.x.x.tar.gz` into a directory that fits your little heart's desires.
2. Setting the environment variable `QUEX_PATH` in your system environment to the place where you installed queχ. If you are using a Unix system and the bash-shell add the following line to your `.bashrc`-file:

```
export QUEX_PATH=the/directory/where/quex/was/installed/
```

if you installed queχ in the directory given on the right hand side of the assignment.

3. Make a link from the file `$QUEX_PATH/quex.py` to `$EXECPATH/quex` where `$EXECPATH` is a path where executables can be found by your system. If you work on a unix system, you might want to type

```
> ln -s the/directory/where/quex/was/installed/quex.py \
       /usr/local/bin/quex
```

You might want to ensure executable rights with

```
> chmod a+rx the/directory/where/quex/was/installed/quex.py
> chmod a+rx /usr/local/bin/quex
```

4. Supplying your c++ compiler with the include path '$QUEX_PATH/templates'. If you are using g++ simply add the option

```
-I$(QUEX_PATH)/code_base
```

to the list of compiler flags. An example of how this is done can be observed in the test applications which come with the distribution of queχ.

That is all. Now, you should either copy the directories `DEMO/*` to a place where you want to work on it, or simply change directory to there. These directories contain sample applications 000, 001, ... Change to the directory of the sample applications and type '`make`'. If everything is setup propperly you will get your first queχ-made lexical analyser executable in the frame of some seconds. The 'normal' Makefile in these directories creates a lexical analyzer with a quex-core engine. If you want to create a core engine via flex you need to type '`make -f Makefile.flex_core`' thus relying on the second makefile.

The example applications depict easy ways to specify traditional lexical analysers, they show some special features of queχ such as mode transitions, and more. The application in `DEMO/002` shows a sample lexical analyser the deals with indentation blocks, i.e. higher and lower indentation opens and closes function blocks. Play a little with the sample applications and then continue with the remainder of this document.

1.2. Licensing Information

The Software is distributed under LPGL, provided the following restriction:

- There is no license for products targeting military purpose.

1.3. The Name of the Game

The name queχ has obviously a lexo-graphical relation to lex and flex—the most known tools for generating lexical analysers at the time of this writing. The 'Qu' at the beginning of the word stands for 'quick', so hopefully the new features of queχ and the ability to created directly coded engines will generate much faster lexical analysers. Also, the process of programming shall be much quicker by means of handy shortcuts that quex provides and the elegance that problems can be solved.

The last letter in the name 'χ' is a the greek lowercase letter 'chi'. It is intended to remind the author of this text that he actually wanted to create a better TEX[1] system with optimized source code appearance. When he realized that traditional lexical analysers did not provide the functionality to express solutions easily he started working on queχ—and dived into the subject much deeper than he ever thought he would.

[1] As Donald Knuth explains it[], the last letter in TEXis an uppercase 'chi' so the name is to be pronounced like the German word 'ach' (it's actually more a sound Germans utter, rather than a word with a dedicated meaning). Analogously, the χ at the end of queχ should not be pronounced like the 'x' of the German word 'nix'.

2. Basics

Lexical analysis is all about patterns. The following section recalls some basics on the mechanisms of pattern matching. For a detailed description see [], [], and []. It also explains how tokens are communicated to the caller of the lexical analyzer. Further, it introduces the queχ's concept of lexical analyzer modes. Eventually, a final section explains the control and handling of mode transitions.

2.1. Pattern Matching

Pattern matching can be described by a finite state automaton where incoming characters play the role of triggers. Each incoming character, i.e. trigger, determines wether the state machine transits into a new state or it remains in the same state. Consider figure 2.1 that depicts a state machine to match a floating point number. In its initial state it waits for a digit (the START state). If no digit arrives, the state machine goes into the FAILED state. It can be said for sure, that this was no number. If a digit arrives, though, in transits into 'DIGITS-A'. As long as digits arrive, it remains in this state. If a non-digit comes in it transits into the ACCEPTANCE state, since already enough digits have arrived as to judge that it is a number. However, if a dot arrived, then the state machine enters into the DOT state. If after the dot a new digit arrives, it enters into the state DIGITS-B. Note, that in the states DIGITS-A, DOT, or DIGITS-B a character from the 'else' set (non-expected characters) lets the machine still transit into an ACCEPTANCE state, since enough characters have arrived to judge that it is a number.

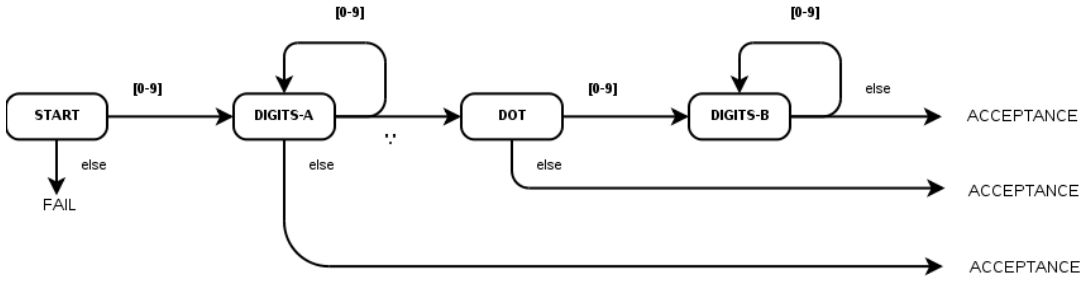

Figure 2.1.: A state machine to detect a floating point number pattern.

In real world lexical analysis a whole bank of state machines are active at the same time. Starting from a position x_0 in the character stream, a *step in lexical analysis* consists of the following procedures:

1. All state machines (one for each pattern) are set into their START state.
2. Starting at position x_0, characters are eaten and the state machines transit their internal states. Some of them enter the FAIL state and drop out, others enter the ACCEPTANCE state and drop out.
3. At some point in time x_1, each of state machines either enters the FAIL state or the ACCEPTANCE state[1]. At this point in time a decision is to be made about 'who is the winner.'
   a) If all state machines reach the FAIL state, then it can be set that from position x_0 the character stream is not sound. The lexical analyser could only send an 'error-token'. One looks at the next character after x_0, i.e. $x_0 = x_0 + 1$ and goes back to step 1.
   b) If one or more state machines reache the ACCEPTANCE state, then the pattern that has eaten the most characters, i.e. lived the longest is considered the winner (the matcher). The show continues.
4. The matcher pattern has an associated *pattern match action* to be executed when the pattern matches. Usually, this interprets the character stream from position x_0 to position x_1 and puts the information into a token, i.e. some signal as output of the lexical analyser.

In brief, a step in lexical analysis starts at a position x_0 in the character stream. After eating some characters, at some position x_1, it can be decided what pattern matches the best, and eventually the action related to the winning pattern is executed. Then lexical analysis starts at position x_1 all over again until the end of the character stream.

In principle, the functioning of a lexical analyser is determined by a 1) a set of patterns and 2) by their related actions. For most practical applications though one requires additionally a definition of token identifiers because the 'signals' need to be understood by another unit (e.g. the parser).

Traditionally, lexical analysers send a single token for an identified pattern in the character stream. Practically, this means that, after a pattern match has happened the 'token-getter-function' returns the token that was identified. Queχ, though, implements a more flexible approach which is better suited for implicit tokens and mode transitions. The following section shows how tokens are communicated by a lexical analyser created with the queχ program.

2.2. The Token Queue

Queχ promotes the idea of implicit tokens, i.e. tokens that are produced without any pattern match. This concept raised from the fact that some tokens are, in fact, redundant and can be derived from context. From the perspective of the source code appearance of a

[1] Let's assume that the end-of-stream character be included into the pattern definitions, so that this statement holds.

programming language, a redundancy free language is much clearer than one that forces the programmer to specify information over and over again. Consider the example of the \begin-\end-regions in LaTeX as shown in the subsequent listing.

```
\begin{itemize}
\item this is the first item.
\item this is the second one.
...
\item this is the last item.
\end{itemize}
```

Logically, the begin and end tokens could have been derived from the fact that an \item token pops up in normal text mode. Implicit tokens means that the lexical analyser can read between the lines. It sees things that are meant to be there from the context. Implementing the concept of implicit token generation inside lexical analysis rather than in a parsers allows the parser to constitute a unit that reasons logically, i.e. it considers distinct signals as input, it has a clear set of grammatical rules, and produces a clear understanding of the information (e.g. a syntax tree of a program). In *Quex*'s philosophy words that are implicitly thought to be *inserted* into the context of a sentence, are treated by lexical analysis. *Multiple-Meanings* (TODO: check this wording), i.e. the *interpretation* of a signal/token due to its context is expressed by grammar.

Traditionally, the parser calls a function of the form 'get_token()'; the lexical analyser does one single pattern match, and then returns the token that represents the matched pattern—or an error, if no pattern matched. This mechanism is not sufficient when implicit tokens are involved, because implicit tokens can pop up out of nowhere in the character stream. When a pattern matches a whole set of such implicit tokens may be ready to be sent.

Figure 2.2 shows the information flow when a lexical analyser generated by queχ parses a character stream. The user begins and requests a new token using member function `get_token()`. The engine's object start then the lexical analysis. During the analysis, the pattern matching actions may send a whole set of tokens into the *token-queue*. When the pattern matching actions decide to come back, the `get_token()` function returns the first token that was entered into the queue.

The subsequent call to `get_token()`, though, does not trigger a lexical analysis since there are still tokens in the queue. Until the queue is empty, the function only returns tokens from the queue. When the queue is empty, the lexical analysis starts again in the same manner as described above. From the caller's side the process appears to be a reading from a continuous token stream. From inside the pattern match action the process appears to be a sending process, where tokens are sent to some recipient.

2.3. Lexical Analysis Modes

In many cases the behavior of the lexical analysers can be efficiently described by means of modes for lexical analysis. A standard example are format strings. Inside format strings, there is no need for the lexical patterns that are necessary when parsing an algorithm. They

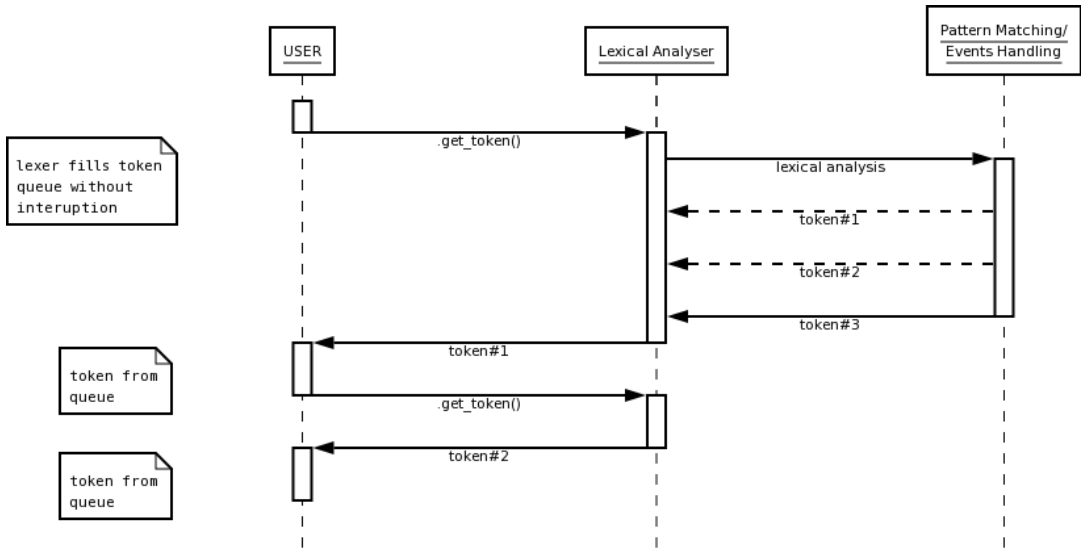

Figure 2.2.: Sequence diagram displaying the token-passing from the pattern-match action to the caller of `get_token()`.

are actually better turned off. Thus, a mechanism is needed that is able to control the activation and deactivation of groups of patterns to be matched.

A lexical analyser mode is a state of the lexical analyser where only a particular set of patterns is actively waiting to be matched against. A lexical analyser needs to be in exactly one mode at a particular point in time. Thus, a lexical analyser mode corresponds to the way that the input stream is currently transformed into signals, i.e. tokens. To handle the detection and influencing of mode transitions, Queχ provides event handling mechanisms as described in the following sections.

A potential disadvantages of modes is confusion. With traditional lexical analyser generators, such as flex, the end-user does not see more than a token stream. He has no inside on the current lexical analyser modes. He cannot sense or control the mode transitions which are currently made. The mode transitions are somwhere hidden in the pattern match actions. GNU Flex's *start conditions* are similar to modes. But, the only way two modes A and B modes can be related in flex is by letting a pattern be active in in A and B. There is no convienient mechanism to say: 'let B overtake all patterns of A'. This is where mode inheritance relationships of Queχ provide clear convenience advantages as described in the section after the following.

2.3.1. Mode Inheritance

Object oriented programming introduced the concept of *inheritance*[], in the sense that two classes can have a 'is-a' relationship. The class of 'cars' may for example be related to the class of 'vehicles' by an 'is-a' relationship. This means that 'cars' inherit all attributes

of the broader class 'vehicles' and may add some specific attributes to it. Such inheritance relationships are directional. In our example this means that every object that belongs to the class of cars also belongs to the class of vehicles (every car is a vehicle). However, not every object that belongs to the class of vehicles (e.g. a submarine) does belong to the the class of cars. Mode inheritance in queχ works similar and effects the following three mechanisms:

- *Patterns*: If a mode B is derived[2] from a mode A, then it overtakes all of its patterns and the correspondent pattern match actions. This is practical if one wants to ensure that a certain mode does react on certain patterns. In this case it has to be derived from the mode that shows the desired behavior.

  Rule: If mode B is derived from A and both have a common pattern XYZ where mode B relates it to a pattern action XYZ_B and mode A relates it to a pattern action XYZ_A, then the pattern action XYZ_B is canceled. *A base pattern-action pair overrides any derived pattern action pair of the same pattern.*

- *Event Handlers*. If a mode B is derived from a mode A, then B's event handler will be executed after A's event handlers. For example, A may have an event handler '`on_entry`' for entering mode A. Mode B itself might have an event handler for '`on_entry`'. If the lexical analyser now enters mode B, the event handler of its base mode A is first executed and then the event handler of class B itself.

  Rule: *Event handlers are executed in sequence from base to derived classes.*

- *Mode Access*: If a mode B is derived from a mode A, then any permission of entry or exit to A is also a permission for B to enter or exit. The philosophy behind this is that that due to pattern inheritance mode B will behave like mode A, so it can be trusted the same way that one trusts A.

  Rule: *Entrys and exits that are allowed for a base class are also allowed for all its derived classes.*

Figure 2.3 shows an example where a mode A acts as a base mode for mode B. Internally, queχ creates a new mode B* that contains the patterns that were explicitly givent by the definition of B and the patterns that are inherited from A. The pattern match action of XYZ from mode B is omitted, since mode B has to behave like mode A for patterns specified in mode A. The `on_entry` and `on_exit` functions are of mode A are pasted into the own event handlers so that they are executed before them. What is not displayed in the figure is access inheritance. When Queχ checks the entry and exits of a mode it consideres automatically also the base classes. If there is no base mode that acts like a door opener to a particular mode, then transits will cause a run-time error.

In conclusion, it can be said that inheritance is a very useful tool to implement some 'standard' behavior in a base mode. Derived modes can then comply to this standard simply by being derived from this mode. Also, for the control of mode transitions the is-a

[2]The statements 'B is derived from A', 'A is base class of B', and 'B inherits from A' are equivalent in meaning in the context of object oriented programming. In the same way the term 'derivation', 'base class', and 'inheritance' shall express mode relations in mode oriented lexical analysis.

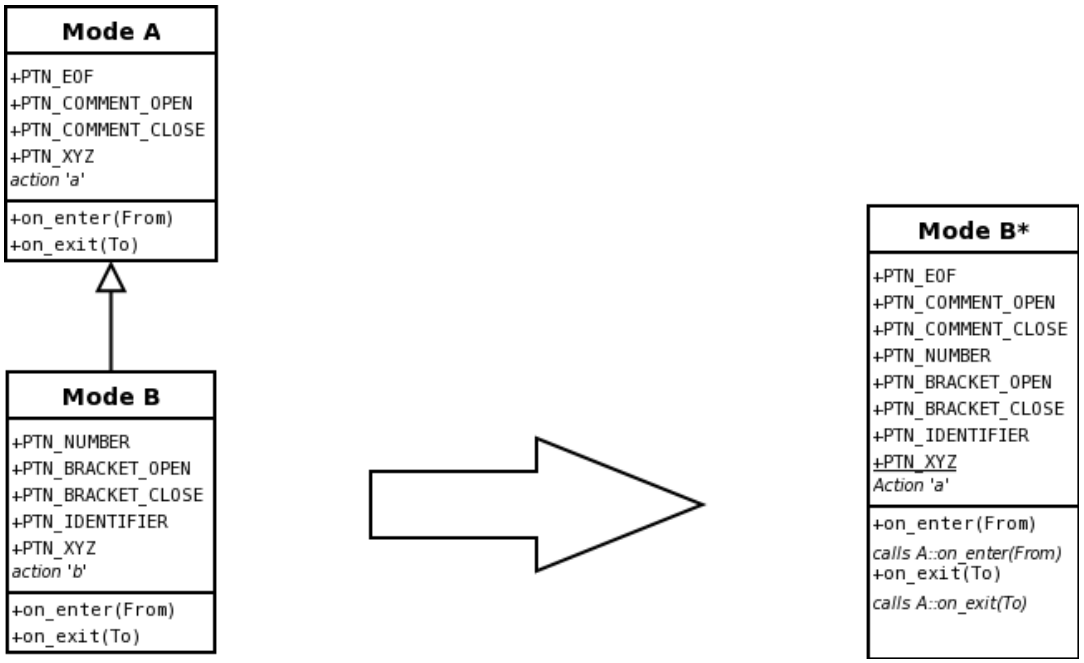

Figure 2.3.: Mode Inheritance.

relationships of inheritance provides much flexibility. Mode inheritance allows to think about mode relationships very naturally and helps to avoid code duplication.

2.3.2. Mode Transitions

In Queχ modes can be changed by two basic mechanisms. First, modes can simple be *changed*. This makes sense, if the new mode knows exactly to what modes it has to change and this does not depend on the previous mode. Another mechanism is that of *pushing and popping*. This makes sense, if the new mode shall return to the previous mode. It is therefore similar to a 'call-subroutine' in procedural programming languages.

Figure 2.4 shows a typical mode transition. Mode X has some patterns and one of them matched. Inside the pattern match action a transition to mode Y is requested. This triggers the event handler `on_exit()` of mode X before any transition is made. Here the user can send tokens, change some internal variables or write some debug output. Then the event handler `on_entry()` of entered mode Y is called. Again the user can provide some actions related to the entrance of that mode.

The ability to define event handlers for entering and exiting modes not only supports implicit token production, but also supports transparency. It can now exactly be traced how modes are transited. In order to *restrict* the transition from and to lexical modes, queχ provides transition control mechanisms. In the header of a definition of a mode X it can be specified from what modes the mode X can be entered and to what modes the mode X can exit. If a mode transition happens that is not conform to this restrictions a run-time error is

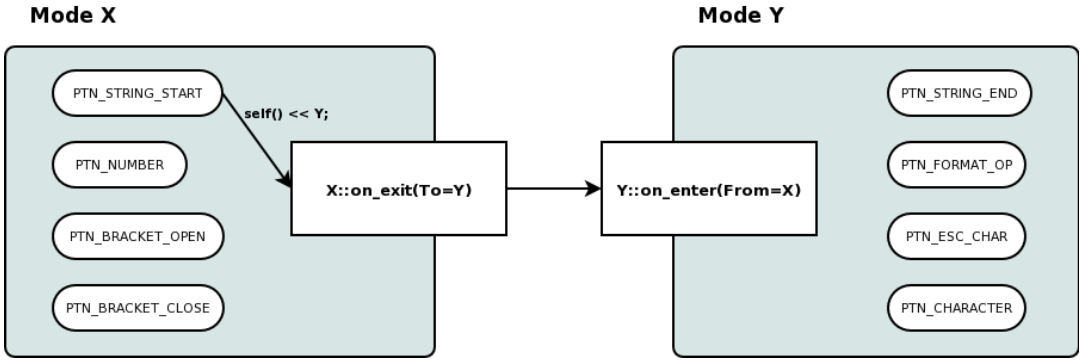

Figure 2.4.: Event handling during lexical mode transitions.

triggered. This helps to prevent the lexical analyser to slide from one mode to another due to some thoughtlessly designed patterns.

Figure 2.5 shows an example, where mode transition control comes very handy. Here, there is a lexical mode for parsing the content of a FUNCTION, a mode for COMMENT and one for DOCU-mentation parsing, a mode for parsing a character STRING and one for MATH. The design here says, that the comment mode can only be activated from the FUNCTION mode and can only leave to FUNCTION mode. Similarly for the STRING and the MATH mode. Queχ provides the feature, that for COMMENT mode, for example, one can specify that it can only be entered from FUNCTION mode, thus preventing the DOCU, STRING and the MATH mode from ever transiting into COMMENT mode. Also, it can be specified that the COMMENT mode can only exit to FUNCTION or DOCU mode, thus preventing it from dropping into STRING or MATH mode.

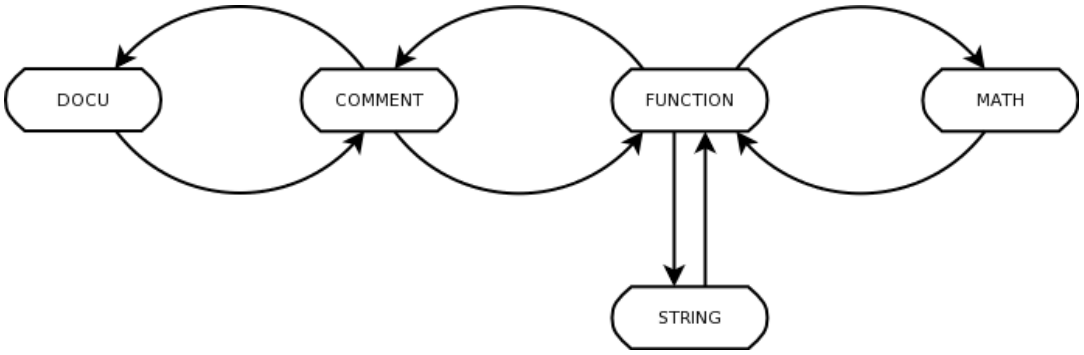

Figure 2.5.: Restricted mode transitions.

Extensive mode transitions without a control mechanism as the one implemented is queχ is highly error prone. The advantage here is that the restrictions on the mode transitions are *explicitly* specified in the description of the lexical analyser. The alternative would be to create an external document and review the behavior of the lexical analyser with respect

to mode transitions whenever an error is assumed from being caused by straying mode transitions. Practically, solid handling of extensive mode transitions in lexical analysis are not possible except with a control mechanism as provided by queχ.

2.4. Indentation

With the rise of the Python programming language the use of indentation as scope delimiter has become popular. Indeed, it maybe the most efficient method to delimit scope with the least amount of additional characters [**?**], such as '{' and '}' in C-styled languages. Queχ provides a convinient mechanism to handle indentations which runs quasi in paralell to the pattern matching: *indentation events*.

Figure 2.6 displays in an example the principle of indentation events. Whenever the lexical analyser reaches the first non-whitespace in a line, an indentation event (indicated as a little star in the figure) is triggered. The lexical analyser engine then calls a user defined indentation handler. The numbers at the indentation events indicate the number of characters that the indentation spans. They are passed to the user's indentation handler as arguments. The indentation handler, then, can then keep track of indentation blocks or whatever his little heart desires.

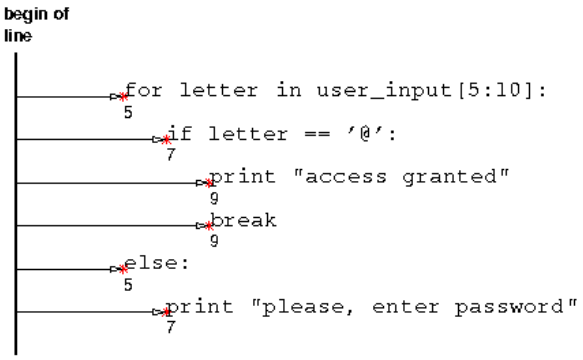

Figure 2.6.: Principle of Indentation Events

Note, that it is not trivial to express indentation in terms of pattern action pairs solely based on regular expressions. It is not enough to define a pattern such as

```
P_INDENTATION_CATCHER    "\n"[ ]*
```

That is a newline followed by whitespace. Imagine, one introduces a comment sign such as the traditional '#' for 'comment until newline'. The comment eating pattern would be at first glance:

```
P_COMMENT_EATER    "#"[^\n]*\n
```

That is a '#' followed by anything but newline and then one newline. The action related to this pattern would have to put pack the last newline. Otherwise the indentation catcher which starts with a newline can never trigger. In this particular case, this problem can be solved by deleting the last newline from the comment eater pattern, knowing that after 'as many not-newline as possible' there must be a newline, i.e.

```
P_COMMENT_EATER    "#"[^\n]*
```

The last newline is then eaten by the indentation catcher. However, the main problem remains: with such a design *patterns need to know about each other* to work propperly! In an environment of many different modes which are additionally related by inheritance, it is difficult to guarantee that all patterns 'respect' the quirks of each other. They better work independently. The indentation events of queχ provide a means that allows one to avoid inter-pattern dependencies.

Similarly, catching indentation with 'pre-condition newline plus whitespace', i.e. `^[ \t]*` is fragile, in a sense that another pattern that contains newline plus whitespace might hinder this pattern from triggering. In a lexical analyzer with dozens or hundreds of patterns this becomes quickly unmanageable. Errors that arise from patterns defined somewhere else are very hard to find and require a lot of insight into the actual process of lexical analysis. Using the '`on_indentation`' event handler ends up in a much clearer and safer design. For more information about the pre-condition newline pitfall see section 4.1.2, page ??.

Caveat:

Note, that if a pattern contains more than one newline then only the indentation event concerning the last newline ist triggered! Imagine a pattern

```
STRANGER    "␣"*"hello"[\n]+"␣"*"world"[\n]+"␣"*"how␣are␣you?"
```

then the following pattern would match

```
  hello
world
    how are you?
```

If this matches, then the lines of 'hello' and 'world' do not trigger an indentation event. So, when dealing with indentation based scoping such strange things are best avoided[3].

2.5. Line and Column Counting

Any compiler or lexical analyzer imposes rules on the text that is has to treat. Most of thoses texts are written by humans and humans make sometimes errors and disrespect rules. A gentle compiler tells its user about his errors and tells it also about the place where the error occured. Here line numbering becomes crucial. Additionally columns can be counted. This provides even more detailed information.

[3]Probably, it is anyway hard to think of a case where such pattern definitions are not to be avoided.

There is general default algorithm to count lines and columns which is always applied in case that there is no better alternative. Queχ analyzes the patterns and adapts the way of counting lines and columns according to special characteristics[4]. If the pattern contains for example a fixed number of newlines, than only a fixed number is added and no newlines are counted at runtime. The mechanisms for line and column counting are optimized for the most 'reasonable' pattern characteristics. There are strange cases[5], though, for which a slightly better counting mechanism might be found. For reasonable applications as known from popular programming languages the mechanisms of counting should be optimal.

Note, that line and column counting can be turned off individually by pre-processor switches. By these switches the generated analyzer is 'naked' of any counting mechanism and it is likely that it runs a little faster. For serious applications, though, at least line number counting should be in place for error reporting.

2.6. Summary

This chapter discussed the major features of queχ. To the knowledge of the author of this text, at the time of this writing, there is no lexical analyser generator that matches these features. Moreover, once one gets used to think in terms of 'sending tokens', 'mode-transitions', and once one gets used to define modes that implement standard behavior for other modes using inheritance, then it becomes hard to get back to the cumbersome thinking of plain lex or flex. The little overhead required to learn new concepts comes with the great advantage of being able to design the lexical analysis process in a very natural way[6].

[4] Actually, even the indentation count algorithm is adapted to profit from knowledge about the patterns internal structure.

[5] Example: A pattern that contains a newline which is followed by a fixed number of characters. The determination of this in the context of post-conditions is complicated. On the other hand, such patterns are considered strange and seldom, so the expected gain with an optimized algorithm was considered neglectable by the author.

[6] Well, at least 'natural' to the author of this text.

3. Practical Usage

This practical examples guides through the phases of creating a lexical analyser using queχ. The files of this example can be found in the directory `DEMO/001/` of the queχ distribution. The file `simple.qx` contains the description of the lexical analyzer, `lexer.cpp` contains a lexical analyzer application, and a `Makefile` allows the convinient creation of the whole lexical analyzer application. Copy-pasting this example may be a good starting point for the work with queχ. A good approach is to devide the development of a lexical analyzer into the two steps of *design* and *implementation*. The first step consists of three minor activities:

1. *Definition of Patterns*. Patterns can be defined directly on the pattern action pair line, but it is much more intuitive to define patterns in terms of regular expressions [] in a separate data section. Those patterns act like constants that allow to understand quickly for what a pattern/action pair stands. Using a name 'NOT_NEWLINE+' is simply visually more appealing than writing [^\n]+. Those patterns can be defined very easily in terms of regular expressions.

2. *Definition of Modes*. This is best done with a drawing program or by hand on paper. One should be clear about how lexer modes are named, how they relate to each other and what transitions are possible. This design will be the basis for the later coding of modes.

3. *Definition of Token-IDs*. We finally want to send tokens and those tokens must have identifiers. For this reason, the user needs to specify names of tokens in a token section, so that queχ can create ids for them.

Once, these simple steps are accomplished the implementation of modes in terms of pattern-action pairs can be started. The example discussed in this section develops the following files that are required for generating a lexical analyser:

`simple.qx` which contains all information queχ needs to create a lexical analyzer.

`lexer.cpp` which uses the created lexical analyser in a mini example application.

To follow the explanations of this chapter the directory `DEMO/001` is best copied into a user domain for further examination and modifications.

3.1. Patterns

In the simple example, only very basic patterns have to be defined. There is a pattern for *whitespace*, consisting of space, tabulator and newline. There are two bracket types for

```
define {
    // patterns of interest for all modes
    P_WHITESPACE                [ \t\n]
    P_IDENTIFIER                [_a-zA-Z]+
    P_NUMBER                    [0-9]+
    // string mode related patterns
    P_STRING_DELIMITER                          "\""
    P_BACKSLASHED_STRING_DELIMITER              "\\\""
    P_BACKSLASHED_BACKSLASH                     "\\\\"
}
```

Figure 3.1.: Definition of pattern shorthands in a `define`-section.

opening and closing brackets, an assignment operator =, the two keywords `if` and `struct`, identifiers consisting of letters from A-Z in capital and lower-case, floating point numbers and string delimeters. The listing in figure ?? shows the content of the `define` section in `simple.qx` describing all required patterns in terms of regular expressions.

The filename including the extension is, of course, totally freely chooseable. Note, that all pattern names are 'global'. They are not local to a specific mode. That means, that all modes can access the pattern definitions. Note also, that inside the `define`-range only '//'-comments are allowed. The `P_`-prefix was chosen to indicate to the reader, that this is a pattern definition. There is no danger of interfering with the program namespace. These shorthand names are only considered for the definition of regular expressions. The later generated engine does not contain them as variables.

3.2. Token-ID

With a similar ease as patterns, token-ids can be specified. A `token`-section containing token-ids only has to contain a newline separated list of token names. Queχ will create a set of constants with unique numerical values. Figure 3.2 shows the definition of token-ids for the example.

Note, that pastes a prefix in front of the numerical constants for the tokens. A token-id of `STRUCT` will be called `TOK_STRUCT`, if `TOK` is specified as token-prefix. This is discussed briefly in the section 4.12. It is also important to remind that the token-ids *are* relevant to the namespace. They name constant variables inside the namespace `quex`. It is not advisable to set the token-id prefix to an empty string and name a token-id '`i`' or '`x`' because those names are likely to interfer with names of other variables.

3.3. Modes

The design of modes, their relations and the transitions is something to be done informally with a drawing program or on a sheet of paper. It is basically a help for the user to develop

```
token {
    BRACKET_O
    BRACKET_C
    CURLY_BRACKET_O
    CURLY_BRACKET_C
    OP_ASSIGNMENT
    IF
    STRUCT
    SEMICOLON
    IDENTIFIER
    NUMBER
    STRING
}
```

Figure 3.2.: Definition of token-ids in a `token`-section.

a clear design of the lexical analysis process. In our simple example, there are only three modes:

1. `END_OF_FILE`: This mode describes the reaction to end-of-file. All modes in the simple example share this behavior, so they are derived from this mode. On the other hand, this is more an *abstract* mode[1]. The lexical analyser shall never be in an `END_OF_FILE` mode, but in a derived mode of it.

2. `PROGRAM`: In this mode, we detect program related tokens. It is derived from `END_OF_FILE`. Thus, an if an end-of-file arrives in this mode it behaves propperly. When a string delimiter arrives, this mode shall transit into the `STRING_READER` mode.

3. `STRING_READER`: This mode shall also treat end-of-file in the 'standard' way, so it is also derived from `END_OF_FILE`. However, all patterns for number detection, keywords and brackets are totally meaningless inside the string. A string-delimiter shall trigger the return to the `PROGRAM` mode.

The inheritance relationships in this example are displayed in figure 3.3. Both 'real' modes `PROGRAM` and `STRING_READER` are derived from `END_OF_FILE` so that they implement the standard behavior for the end-of-file event.

In this examples their are only two modes in which the lexical analyser can be. The posible transitions are shown in figure ??. `PROGRAM` mode can transit into `STRING_READER` mode and vice versa. The mode inheritances and transitions are only pinpointed in non-machine readable form. This information can be used to write the first stubs of the modes in which one will later fill in the pattern-action pairs.

The three code-stubs for the three modes are shown in figure ??. Until now, there are no pattern action pairs defined inside the modes. The dots `...` inside the curly braces will later

[1] In the C++ sense of the word, there is no 'real' existing mode of that type. Only derived modes can exists.

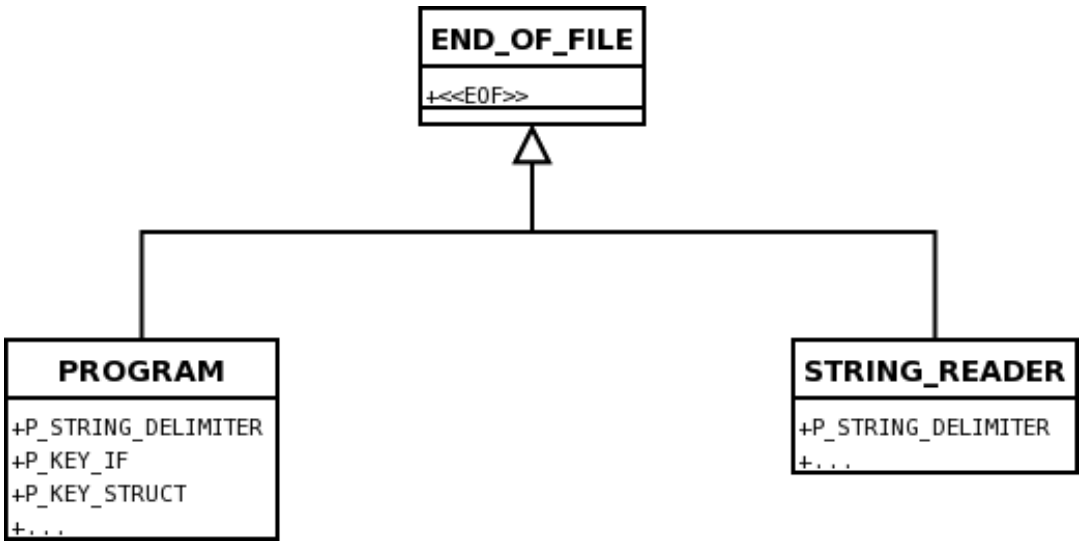

Figure 3.3.: Inheritance relationships of the modes in the example lexical analyser.

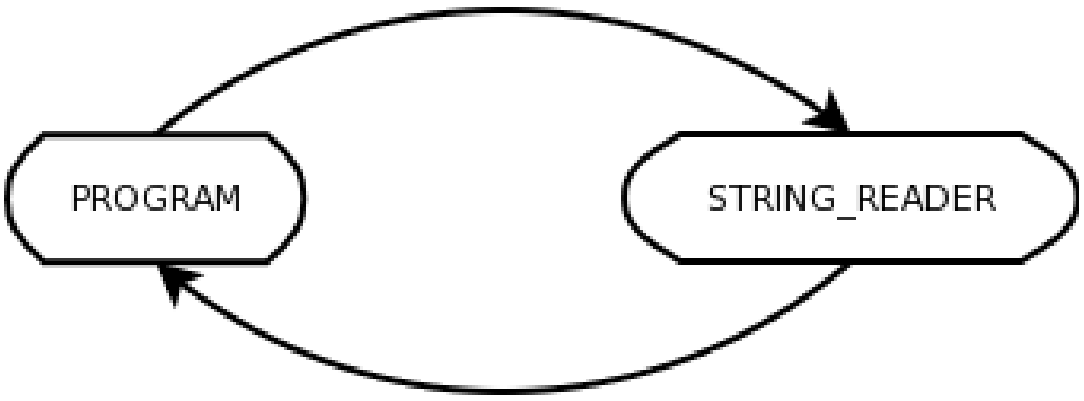

Figure 3.4.: Scratch of mode transitions in the simple example.

```
mode END_OF_FILE :
<inheritable: only>
{
...
}

mode PROGRAM :
      END_OF_FILE
<exit:  STRING_READER>
<entry: STRING_READER>
{
...
}

mode STRING_READER :
      END_OF_FILE
<exit:  PROGRAM>
<entry: PROGRAM>
{
...
}
```

Figure 3.5.: Example code stubs for lexical analyser modes.

be replaced by the pattern-action pairs. First of all, mode END_OF_FILE is defined. It does not inherit any mode, but can only be used as a base mode. Thus the option

```
<inheritable: only>
```

is specified in edgy brackets. Second, the PROGRAM mode is derived from END_OF_FILE which is indicated by its name following the colon. The options

```
<exit:    STRING\_READER>
<entry:   STRING\_READER>
```

tell that this mode can be entered from STRING_READER mode, and that it can exit to this mode. Any transition to another mode (that is not derived from STRING_READER) is detected as an error[2]. Analogously, the STRING_READER mode is defined specifying its base class and the allowed transitions to PROGRAM.

Note, that all options given here are indeed *optional*. They are not necessary to define pattern-action pairs, and they do not contribute to the functioning of the generated lexical analyser. However, they allow to check wether mode definitions and mode transitions are according to the design or not. The detection of usage against design specifications is of great help to avoid errors or find their causes.

3.4. Pattern-Action Pairs

The previous sections explained the definitions of patterns, token-ids, and mode characteristics. In this section the core of the lexical analysis is discussed: pattern-action pairs. Lexical analysis has as a result a stream of tokens that tell about atomic information chunks in the input stream. The most simple pattern-action pair consists of sending a specific token whenever a particular pattern occurs. Consider for example figure 3.6. It shows the definition of a mode that can handle the end of a file. The only pattern of this mode is <<EOF>>. The action to be executed if this pattern occurs is embraced by curly brackets. Inside the brackets a special token is sent, a token with the id:

```
quex::token::ID_TERMINATION
```

telling that lexical analysis is over. Note, that in queχ one uses the keyword `self` and not a `this`-pointer to refer to the lexical analyser. Recall that queχ is a code generator, not a compiler. Internally, the code fragments that are specified as pattern-action pairs or event handlers may be pasted by queχ into objects other than the lexical analyser (e.g. the lexical-analyser-mode-information structures). Also the user might derive his own class from the lexical analyser class, so there would be a type-issue. To unify the access to the lexical analyser, all places were code is pasted allow to use `self` as a variable that represents the lexical analyser. The rule here is: say '`self.`' not '`this->`'.

[2]Note, that inside queχ `assert()` commands are used. According to ISO9899 (ANSI-C) the command only takes effect, if NDEBUG is not defined. If it is, the mode transitions are consequently not checked by the queχ engine. The general rule applies here also: only define NDEBUG if it is 100% safe to say that no use-case will violate the design.

```
mode END_OF_FILE :
<inheritable: only>
{
    <<EOF>> {
        self.send(quex::token::ID_TERMINATION);
        RETURN;
    }
}
```

Figure 3.6.: The END_OF_FILE mode handling the end-of-file pattern.

A final RETURN statement in the pattern-action pair says that the flow of control can be turned back to the caller of function get_token(). Note again, that return is not used. The RETURN macro ensures that the token queue is in a sound state before returning from the lexical analysis. The rule here is: say 'RETURN' not 'return'.

The pattern-action pairs of the PROGRAM mode displayed in figure 3.7 are a little more diverse. Note also, that for simple patterns the operator '=>' is used to express the sending of a token of a specific id. The whitespace pattern does not cause any token to be sent and one does not return to the user, thus it is followed by an empty pair of curly brackets. The pattern P_IDENTIFIER creates a token that contains the string that was matched. P_NUMBER also stores a number that corresponds the lexeme that was matched. The pattern P_STRING_DELIMITER does not send a token, but initiates a mode transition to the mode STRING_READER.

The STRING_READER mode's pattern-action pairs are shown in figure 3.8. This mode makes use of a special member of the lexical analyser object: *the accumulator*. This object allows to accumulate text from incoming patterns and flush them when desired. When this mode is entered, the accumulator is emptied (see on_entry). When this mode is left, then the accumulator is flushed (see on_exit). This means, that if the accumulated text is not empty, it is send as a token with a user defined token-id. The STRING_READER mode sends the accumulated text with a token-id TKN_STRING.

The arrival of a string delimiter causes a mode transition back to mode PROGRAM. A backslashed string delimiter allows to have a string delimter inside the the string, so it does not cause a return to PROGRAM mode. instead, it adds a quote to the accumulator. The default action '.' specifies that any other incoming text is simply accumulated. Even in this simple example it becomes clear that the specification of entry and exit event handlers facilitates the specification of mode transitions. It can be said for sure, that wherever in the code a command triggers an exit to another mode, the accumulator is flushed.

This section now finished the specification of file simple.qx as an input file for the queχ lexical analyser generator. The next section tells how to call queχ to generate a full fledged engine that does the lexical analysis as required.

```
mode PROGRAM :
      END_OF_FILE
<exit:  STRING_READER>
<entry: STRING_READER>
{
    "{"           => TKN_CURLY_BRACKET_O;
    "}"           => TKN_CURLY_BRACKET_C;
    "="           => TKN_OP_ASSIGNMENT;
    "struct"      => TKN_STRUCT;
    "if"          => TKN_IF;
    ";"           => TKN_SEMICOLON;
    {P_NUMBER}    => TKN_NUMBER(atoi(yytext));
    {P_IDENTIFIER} { self.send(TKN_IDENTIFIER, yytext); RETURN; }

    {P_WHITESPACE} {
    }
    {P_STRING_DELIMITER} {
        self << STRING_READER;
    }
}
```

Figure 3.7.: The PROGRAM mode.

3.5. Calling Quex

With the files from directory $(QUEX_PATH)/DEMO/001/ the creation of the example lexical analyser application is as simple as can be. Type `make` in the command line and the application is built. The Makefile that comes with the sample application is written in a way that makes it very easy to adapt and extend its contents for other user's needs. The core of this make file explains how to call queχ:

```
# (*) definitions
MODE_FILES   = ./in/simple.qx
#
ENGINE_NAME = tiny_lexer
#
ENGINE_SOURCES = $(ENGINE_NAME)                \
                 $(ENGINE_NAME).cpp            \
                 $(ENGINE_NAME)-internal.h     \
                 $(ENGINE_NAME)-token_ids      \
                 $(ENGINE_NAME)-core-engine.cpp
...

# (*) build rule
$(ENGINE_SOURCES): $(PATTERN_FILE) $(MODE_FILES) $(TOKEN_DB)
```

```
mode STRING_READER :
     END_OF_FILE
<exit:  PROGRAM>
<entry: PROGRAM>
{
    on_entry {
        self.accumulator.clear();
    }

    on_exit {
        self.accumulator.flush(TKN_STRING);
    }

    {P_BACKSLASHED_STRING_DELIMITER} {
        self.accumulator.add('\"');
    }

    {P_STRING_DELIMITER} {
        /*.....................................*/ self << PROGRAM;
        RETURN;
    }

    . {
        self.accumulator.add(Lexeme);
    }
}
```

Figure 3.8.: The `STRING_READER` mode.

```
quex --mode-files $(MODE_FILES)      \
     --engine     $(ENGINE_NAME)
```

The build rule says that whenever one of the files containing mode descriptions changes the source code for the lexical analyser has to be rebuild. In our example the only mode file is `simple.qx`. Queχ receives the names of the input file names as 'no-minus' followers of the command line option '`--mode-files`'.

The output of queχ is determined by the option `--engine`. The name that follows it specifies the name of the class that implements the engine. But, it also acts as filestem for all output files. In our example, the engine's name is `tiny_lexer`. Therefore, there will be four files created:

File: `tiny_lexer` containing the header file that contains the class definition of class `quex::tiny_lexer`. This is *the only output file that is of direct interest to the user*. It has to be included in each file that interacts with the produced lexical analyser.

File: `tiny_lexer.cpp` containing the created lexical analyser engine. The user does not need to touch it, but it has to be compiled sometime and linked to the application. The example Makefile does this already. *The user is not directly concerned with this file*.

File: `tiny_lexer-token-ids` containing definitions of token-ids. That means it provides variables with the names of token-ids that carry unique numerical values. It also defines the member function

```
string token::map_id_to_name(const id_type TokenID)
```

which maps a token-id to a token-id name. This comes practical in many stages of the development. However, this file is included inside the header file for the lexer class definition and *the user does not worry about this file at all*.

File (only with flex): `tiny_lexer-internal` is a C-style header file containing extracted information from a file that was initially created by flex. The user is not concerned of this file.

File (only with queχ-core engine): `tiny-lexer-core-engine.cpp` is a generated C++ code file containing the generated lexical analyzer engines for all modes.

Once the engine sources are built by queχ (and possibly by the help of flex), an example application can be built that uses the lexical analyser. The source code for the example is specified in file '`lexer.cpp`'. The content of this file is shown in figure 3.9.

Initially, the lexical analyser needs to be created as an object of type `quex::tiny_lexer`. The filename of the file to be analysed is directly passed to the constructor of the class. Then, one needs to set the initial lexical analysis mode. In our example, `PROGRAM` shall be the initial mode. Since it shall not trigger any mode transitions or protection algorithms, it is set using the function `set_mode_brutally()`.

After printing out the version information, the loop over the token stream starts. In the sample application it does not more than getting the a token using `get_token()` and

```
#include<fstream>
#include<iostream>

// (*) include lexical analyser header
#include <./tiny_lexer>

using namespace std;

int
main(int argc, char** argv)
{
    // (*) create token
    quex::token         Token;
    // (*) create the lexical analyser
    //     if no command line argument is specified user file 'example.txt'
    quex::tiny_lexer*  qlex = new quex::tiny_lexer("example.txt");

    // (*) set the initial mode of the lexical analyser
    qlex->set_mode_brutally(qlex->PROGRAM);

    // (*) print the version
    cout << qlex->version() << endl << endl;

    cout << ",-------------------------------------------------------------------------------
    cout << "| [START]\n";

    int number_of_tokens = 0;
    // (*) loop until the 'termination' token arrives
    do {
        // (*) get next token from the token stream
        qlex->get_token(&Token);

        // (*) print out token information
        //     -- line number and column number
        cout << "(" << qlex->line_number() << ", ";
        cout <<       qlex->column_number() << ")  \t";
        //     -- name of the token
        cout << Token.type_id_name() << endl;

        ++number_of_tokens;

        // (*) check against 'termination'
    } while( Token.type_id() != quex::token::ID_TERMINATION );

    cout << "| [END] number of token = " << number_of_tokens << "\n";
    cout << "`-------------------------------------------------------------------------------

    return 0;
}
```


Figure 3.9.: A sample application using the generated lexical analyser `tiny_lexer`.

.

printing it out. In a more realistic application this token is to be passed to a parser that does its syntactic analysis on it. When a token arrives with an identifier equal to `quex::token::ID_TERMINATION`, then the loop exits and the program terminates.

At this point in time everything is ready for compilation. Using the Makefile coming with the example, typing `make` shall produce a ready-to-run application called `lexer` in the current directory. If one types now

```
> ./lexer example.txt
```

on the command line the following output shall be produced:

```
tiny_lexer: Version 0.0.0-pre-release. Date Sat Jul  1 12:00:20 2006
Generated by Quex 0.0.1

,------------------------------------------------------------------
| [START]
(0, 0)          STRUCT
(0, 7)          IDENTIFIER
(0, 15)         CURLY_BRACKET_O
(1, 2)          IDENTIFIER
(1, 9)          IDENTIFIER
(1, 10)         SEMICOLON
(2, 2)          IDENTIFIER
(2, 9)          IDENTIFIER
(2, 10)         SEMICOLON
(3, 0)          CURLY_BRACKET_C
(3, 1)          SEMICOLON
(5, 0)          IF
(5, 3)          IDENTIFIER
(5, 12)         CURLY_BRACKET_O
(6, 2)          IDENTIFIER
(6, 7)          IDENTIFIER
(6, 14)         NUMBER
(6, 34)         STRING
(6, 35)         SEMICOLON
(7, 0)          CURLY_BRACKET_C
(7, 1)          <TERMINATION>
| [END] number of token = 21
`------------------------------------------------------------------
```

3.6. Summary

This section has shown how to built a lexical analyser with queχ and provided a ready-to-run sample application. It was discussed how to specify patterns in a pattern file, how to announce names for token-identifiers and how to code mode-files that contain information

about modes and pattern-action pairs. After an explanation of how to invoque queχ, a sample application was discussed that uses the created lexical analyser engine. The Makefile that comes with the example can be easily adapted to particular applications and reduces the required effort to the typing of `make` on the command line. In general, the example files `simple.qx` and `lexer.cpp` build a good basis for a new project. In the following chapters various features and options of que$\chi\}$ are discussed. Additionally some discussion will be made about the generated lexical analyser. The current chapter provided enough information to start coding a lexical analyzer. Following chapters become interesting once you tasted the ease of queχ.

4. Formal Ussage

The last chapter provided a functioning example for the usage of queχ. At this point the user should have a clear idea about the basic concepts and how they are applied. This chapter discusses the details of usage and minor features that may not have been mentioned in the previous chapters. The first sections discuss the specification of patterns, mode characteristics, pattern-action pairs, and mode transitions. Section ?? discusses the lexical analyser class that queχ creates with all members and member functions that the user might want to use.

In some cases, the provided functionality of the generated lexical analyser class might not be sufficient, so section 4.8 discusses the formalities to implement a derived class and its usage. The generated code contains some macro switches that allow to control the behavior of the analyser at compile time. Section ?? discusses the usage of those macros. Finally, section 4.12 explains all command line arguments that can be passed to queχ.

4.1. Patterns

Section 3.1 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 solely on a core that was produced by flex. From version 0.10.0 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 mostly conform with the world of lex/flex. However, queχ provides features that flex does not. If it is intended to use flex as a core generating engine on the long run, then please refer to the flex manual [], section 'Patterns'. This section discusses pure queχ syntax. The explanation is divided into the consideration of context-free expressions and context-dependent expressions.

4.1.1. 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` 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 and EOF. Note, that on systems where newline is coded as $0D$, $0A$ this matches also the $0D$ character whenever a newline occurs (subject to possible change).
- `[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 left 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'.

Two special rules have to appear isolatedly, out of the context of regular expressions. With the following two rules the actions for the event of end of file and the failure event can be specified:

- `<<EOF>>` the event of an end-of-file (end of data-stream).
- `<<FAIL>>` the event of failure, i.e. no single pattern matched. Note, this rule is of the 'lex' style, but is only available with the queχ core engine.

This syntax is more a 'recognition' of the traditional `lex` syntax. In fact the two event handlers '`on_failure`' and '`on_end_of_stream`' are a one-to-one correspondance to what is mentioned above. Possibly some later versions will totally dismiss the lex related engine

core, and then also these constructs will disappear in favor of the mentioned two event handlers.

Operations:

Let R and S be regular expressions, i.e. a chain of characters specified in the way mentioned above, or a regular expression as a result from the operations below.

- 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 group operations, i.e. to override precedence, in the same way as the brackets in $(a + b) \cdot c$ override the precedence of multiplication over addition.
- 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 defined pattern "NAME". Recall, that pattern names can be defined in `define` sections (see section 3.1).

Pitfalls

The most dangerous pitfall is related to precedence and length. Note, that a pattern that is defined *before* another pattern has a higher precedence. Also, if a pattern can match a longer chain of characters it wins. Thus, if there are for example two patterns

```
[A-Z]+     => TKN_IDENTIFIER(Lexeme);
"PRINT"    => TKN_KEYWORD_PRINT;
```

then the keyword PRINT will never be matched. This is so, because [A-Z] matches also the character chain PRINT and has a higher precedence, because it is defined first. To illustrate the danger of 'greedy matching', i.e. the fact that length matters, let two patterns be defined as:

```
"Else"             => TKN_KEYWORD_ELSE;
"Else\tAugenstein" => TKN_SWABIAN_LADY(Lexeme);
```

Now, the `Else` statement may be matched, but only if it is not followed by tabulator and `Augenstein`. On the first glance, this case does not seem to be very probable. Sometimes it may be necessary, though, to define delimiters to avoid such confusion. In the very large majority of cases 'greedy matching' is a convienient blessing. Imagine the problem with identifiers, i.e. any chain of alphabetic characters, and a keyword '`for`'. If there was no greedy matching (longest match), then any variable starting with `for` could not propperly be detected, since the first three letters would result in the `for`-keyword token.

As for version 0.10.0, queχ has a deficiency which is due to the parsing algorithm of the mode definition parts. Until now, spaces ' ' cannot be used directly inside mode definitions, i.e.

```
mode {
    [ \t\n]    {
// ... reaction on whitespace ...
    }
}
```

is *not* possible. Please, define a whitespace pattern inside a '`define`' section and use the pattern lookup, as shown below:

```
define {
    WHITESPACE   [ \t\n]
}

...

mode {
    {WHITESPACE} {
// ... reaction on whitespace ...
    }
}
```

Since version 0.10.0, there is indeed a regular expression parser inside quex for the implementation of the quex core engine. It has not been connected, though, to the old engine that was to produce only flex-readable code, without 'understanding' it. Versions to come are likely to catch such pitfalls by parsing the regular expression.

4.1.2. Pre- and Post-Conditions

Additionally to the specification of the pattern to be matched queχ allows to define conditions on the boundary of the pattern. This happens through pre- and post-conditions. First, the trivial pre- and post-conditions for begin of line and end of line are discussed. Then it is shown how to specify whole regular expressions to express conditions on the surroundings of the pattern to be matched. The traditional characters to condition begin and end of line are:

- `^R`: an `R`, but only at the beginning of a line. This condition holds whenever the scan starts at the beginning of the character stream or right after a newline character. This shortcut scans only for a single newline character '\' (hexadecimal $0A$) backwards, independent on how the particular operating system codes the newline. In this case, there is no harm coming from different conventions of newline.

- `R$`: an R, but only at the end of a line or right before the end of the file. Note, that the meaning of this shortcut can be adapted according to the target operating system. Some operating systems, such as DOS and Windows, code a newline as a sequence '\r\n' (hexadecimal $0D$, $0A$), i.e. as two characters. If you want to use this feature on those systems, you need to specify the `--DOS` option on the command line (or in your makefile). Otherwise, `$` will scan only for the newline character $0A$.

  Note, that for the trivial end-of-line post condition the newline coding convention is essential. If newline is coded as $0D, 0A$ then the first $0D$ would discard a pattern that was supposed to be followed by $0A$ only.

For more sophisticated case 'real' regular expressions can be defined to handle pre- and post-conditions. Note, that pre- and post-conditions can only appear at the front and rear of the core pattern. Let `R` be the core regular expression, `Q` the regular expression of the pre-condition, and `S` the regular expression for the post-condition.

- `R/S`: matches an `R`, but only if it is followed by an `S`. If the pattern matches the input is set to the place where `R`. The part that is matched by `S` is available for the next pattern to be matched. `R` is post-conditioned. Note, that the case where the end of `R` matches the beginning of `S` cannot be treated by Version 0.9.0[1].

- `Q/R/`: matches `R` from the current position, but only if it is preceeded by a `Q`. Practically, this means that queχ goes backwards in order to determine if the pre-condition `Q` has matched and then forward to see if `R` has matched. `R` is pre-conditioned. Note, with pre-conditions there is no trailing context problem as with post-conditions above.

- `Q/R/S`: matches `R` from the current position, but only if the preceeding stream matches a `Q` and the following stream matches an `S`. `R` is pre- and post-conditioned.

Pitfalls

A somehow subtle pitfall is related to the begin-of-line pre-condition. When using the '^' sign at the beginning of a line it is tempting to say "do not worry, this matches at *any* begin of line"—well it does not. The important thing to understand is that it matches when the lexical analysis step *starts* at the beginning of a line. The alarm signal should ring if begin-of-line is triggered and a whitespace pattern is defined that includes newline. Consider the following patterns being defined:

[1] The reason for this lies in the nature of state machines. Flex has the exact same problem. To avoid this some type of 'step back from the end of the post-condition' must be implemented.

```
define {
    WHITESPACE      [ \t\n]+
        ....
    GREETING         ^[ \t]*hello
}

mode x_y_z : {
     {WHITESPACE}   => TKN_WHITESPACE;
     {GREETING}     => TKN_GREETING(Lexeme);
}
```

Where the `hello` greeting is to be matched after newline and some possible whitespace. Now, given the character stream:

```
    ...
    hello
```

will *not* send a GREETING token. This is because the whitespace pattern eats the newline before the 'hello-line' and even the whitespace before hello. Now, the next analysis step starts right before `hello` and this is not the beginning of a line. Splitting the whitespace eater into newline and non-newline helps:

```
define {
    WHITESPACE_1    [ \t]+
    WHITESPACE_2    [\n]+
    GREETING        ^[ \t]*hello
}

mode x_y_z : {
     {WHITESPACE_1}   => TKN_WHITESPACE;
     {WHITESPACE_2}   => TKN_WHITESPACE;
     {GREETING}       => TKN_GREETING(Lexeme);
}
```

Now, the first whitespace eating ends right before newline, then the second whitespace eating ends after newline, and the hello-greeting at the beginning of the line can be matched.

Another pitfall is, again, related to precedence. Make sure that if there are two patterns with the same core pattern `R`, then the pre- or post-conditioned patterns must be defined *before* the un-conditioned pattern. Otherwise, the pre- or post-conditioned patterns may never match. Recall section ?? for a detailed discussion on precedence pitfalls.

4.1.3. The Starting Mode `start`

In most practical application it is not a good idea to start in a void initial mode, so queχ requires an initial mode explicitly to be specified. This happens using the `start` keyword, i.e. one has to type somewhere in between the mode definitions (but best at the very beginning):

```
start = PROGRAM
```

Let the mode-stubs be stored in a file called `simple.qx`. The previous sections defined patterns, token-ids, and modes. Together with these definitions one can now start with the specification of pattern-action pairs as discussed in the following section.

4.2. Mode Characteristics

Queχ allows to specify characteristics of modes concerning transitions and inheritance. The keyword `mode` starts the definition of a lexical analyser mode. Then it can be directly followed by a curly bracket that starts the pattern-action pair definitions as shown in figure 4.1a. This is the most simple way to define a mode.

In order to define options or base modes the name of the mode has to be followed by a colon as in the examples in figure 4.1b and 4.1c. Note, that if a mode specifies base modes and options, the options are specified after the base modes.

Base modes directly follow the ':' sign. The list of base modes is whitespace separated. When the first $<$ sign arrives, queχ starts reading the options. Options are bracketed by $<$ and $>$ signs and multiple options are also listed whitespace separatedly. The following options can be specified:

- $<$`inheritable: [ "yes", "no", "only" ]` $>$] restricts the inheritance relationships. If `yes` is specified, it is allowed for any other mode to be derived from this mode. If `no` is specified and another mode tries to derive from this mode, an error will be printed. Specifying `only` tells that this mode cannot be a lexical analyser mode. Its reason for existance is to support derived modes. This is similar to the concept of abstract classes in C++ or interfaces in Java.

- $<$`entry:` *mode-name* $>$ tells that it is allowed that the lexical analyser enters this mode from mode *mode-name* or one of its derived modes. If no entry mode is specified, then all modes are allowed.

- $<$`exit:` *mode-name* $>$ tells that it is allowed that the lexical analyser exits from this mode to mode *mode-name* or one of its derived modes. If no exit mode is specified, then all modes are allowed.

- $<$`restrict: [ "entry", "exit" ]` $>$ turns of the admissibility of derived modes for entry or exit. If `entry` is specified the entry is restricted, so any other mode than the the listed entry modes is forbidden even if it is derived from one of them. Respectively, specifying `exit` restricts the list of exit modes.

Options are optional, i.e. they do not need to be specified. But, they facilitate the debugging and add confidence that the mode transitions are safe and sound. The list of options is then to be followed by an opening curly bracket '{' that opens the list of pattern-action pairs. These are discussed in the following section.

a)

```
mode MY_MODE {
   ...
   // pattern-action pairs of MY_MODE
   ...
}
```

b)

```
mode MY_MODE :
    END_OF_FILE
    WHITESPACE_EATER
    DOCUMENTATION_TRIGGER
    {
   ...
   // pattern-action pairs of MY_MODE
   ...
}
```

c)

```
mode MY_MODE :
    END_OF_FILE
    <inheritable: yes>
    <exit:        OTHER_MODE>
    <restrict:    exit>
    {
   ...
   // pattern-action pairs of MY_MODE
   ...
}
```

Figure 4.1.: Definition of mode characteristics.

4.3. Pattern-Action Pairs

Queχ only allows to specify pattern-action pairs inside modes. As the name says, a pattern-action pair consists of a pattern that is to be matched and a code fragment that is executed when the pattern matches. Patterns are best specified inside the pattern definition sections, so that inside the mode sections only names of patterns occur. However, one can also use regular expressions inside the mode definitions (currently no ' ' spaces, though). The format for pattern-action pairs is as simple as can be:

1. The *pattern* is specified as a regular expression or a pattern name embraced by curly brackets.
2. Then comes a *whitespace* that separates the pattern from the action.
3. Then there must be a *curly bracket followed by whitespace* that delimits the start of the code fragment of the action. Queχ parses then until the *matching closing curly bracket* that terminates the action.

Examples for pattern-action pairs have been given enough in section 3.4, page 24. Note, if you want to define a pattern that contains whitespace, you must define it in the pattern section! Then use the pattern macro in the pattern action pair. Otherwise, queχ might interpret the whitespace as 'end-of-pattern'[2]. There are a few important variables to know for coding the action for a pattern match:

1. `self`: Use the reference `self. + member_name` in order to access a member of the lexical analyser. This works in any place - also in event handlers. Better do not try to use the `this` pointer in order to keep the code uniform. The `self` reference automatically referes to an object of the derived class if such exists, and it points to the lexical analyser even if the code is pasted into member functions of different objects.
2. `RETURN`: Use the `RETURN` macro to return from a pattern action pair. This ensures a synchronization with the token queue.
3. `Lexeme`: The string that matched the pattern of the current action can be accessed through variable `Lexeme`.
4. `LexemeL`: The length of a lexeme that matched a pattern can be accessed through variable `LexemeL`.

4.3.1. Pattern Action Shortcuts

In practical applications for programming languages, the large majority of patterns to be identified are simple and do not require large algorithms to handle them. Keywords, such as '`if`', '`then`', and '`foreach`', operators, such as '+' and '*' and delimiters such as '`;`', '`(`' and '`)`' simply need to send a token telling that such and such appeared. Their pattern action pairs all look similar to the following:

[2]This holds at least until version 0.10.0.

```
...
{P_KEYWORD_IF} {
        self.send(TKN_KEYWORD_IF);
        RETURN;
}
{P_KEYWORD_THEN} {
        self.send(TKN_KEYWORD_THEN);
        RETURN;
}
{P_OPERATOR_PLUS} {
        self.send(TKN_OP_PLUS);
        RETURN;
}
...
```

Queχ allows to write such simple token sendings in a more elegant manner. If a pattern is followed by a => operator and a name *xyz* then this is translated into 'pattern sends token with name *xyz*'. The code as shown above will be internally generated. The above code segment is equivalent to the following:

```
...
{P_KEYWORD_IF}      => TKN_KEYWORD_IF;
{P_KEYWORD_THEN}    => TKN_KEYWORD_THEN;
{P_OPERATOR_PLUS}   => TKN_OP_PLUS;
...
```

For simple patterns that consist of a fixed string and that appear only in one place it does not make sense to define a pattern explicitly. They can also be given directly, i.e. one can write

```
...
"if"      => TKN_KEYWORD_IF;
"then"    => TKN_KEYWORD_THEN;
"+"       => TKN_OP_PLUS;
...
```

thus avoiding long lists of pattern definitions external to the mode definition. Sometimes, though, things are little more complicated. It might be necessary to extract some of the information in the lexeme and store it into the token as in the following pattern action pairs:

```
...
{P_NUMBER} {
   self.send(TKN_NUMBER, atoi(Lexeme));
   RETURN;
}
{P_IDENTIFIER} {
   self.send(TKN_IDENTIFIER, Lexeme);
```

```
    RETURN;
  }
  ...
```

In these cases, one passes arguments to the send function other then the token-id. In order to specify additional arguments to the send functions the token-ids have to be followed by a pair of brackets containing the list of arguments (possibly separated by comma). This way, the above two pattern action pairs can be rewritten elegantly as:

```
  ...
  {P_NUMBER}      => TKN_NUMBER(atoi(Lexeme));
  {P_IDENTIFIER}  => TKN_IDENTIFIER(Lexeme);
  ...
```

These abbreviations not only speed up the generation of a lexical analyser tremendously, but they make the lexical analyser descriptions also very readable. Long lists of pattern-action pairs can be understood and analysed in a few seconds.

4.3.2. Priorities

For a given character stream it might be possible that more than one pattern matches. Therefore, rules are needed that describe the resolution of those 'conflicts'. This section explains the priorities that queχ assigns to a given set of patterns based on length, position, inheritance relationships, and a possible *priority-mark*. Imagine, two patterns:

```
"print"    { ... /* print keyword */ }
"printer"  { ... /* device name   */ }
```

both patterns '`"print"`' and '`"printer"`' match the first five characters of a character stream '`print...`'. The first pattern could match, but what if the stream continuous with '`...er`'? It would never be possible to match an incoming '`printer`' because the first pattern eats the first five characters and the remaining `er` is lost in space. Thus, the rule, followed by most lexical analysers: *The pattern with the longest match proceeds!* Still, there might be more than one pattern that matches the same number of characters, e.g. the two patterns

```
"print" { ... /* print keyword */ }
[a-z]+  { ... /* identifier    */ }
```

match in a character stream '`print(`' exactly five characters. So do we deal with a print keyword or with an identifier? The rule here is: *first come, first serve - patterns that are mentioned first in the code win!*. If a pattern in a base mode and a pattern in a derived mode match with the same number of characters, than *the base mode' pattern proceeds!*. This follows the philosophie that the base mode imposes behavior on the derived modes. Figure 4.2 shows how a pattern is matched in case of multiple patterns matching the current character input stream.

For cases of real urgency, a keyword allows to struck the philosophie of base and derived classes: `PRIORITY-MARK`. A pattern followed by this keyword is *lifted* into the current mode, thus having the priority according to the position in the current mode not of the base mode. For example:

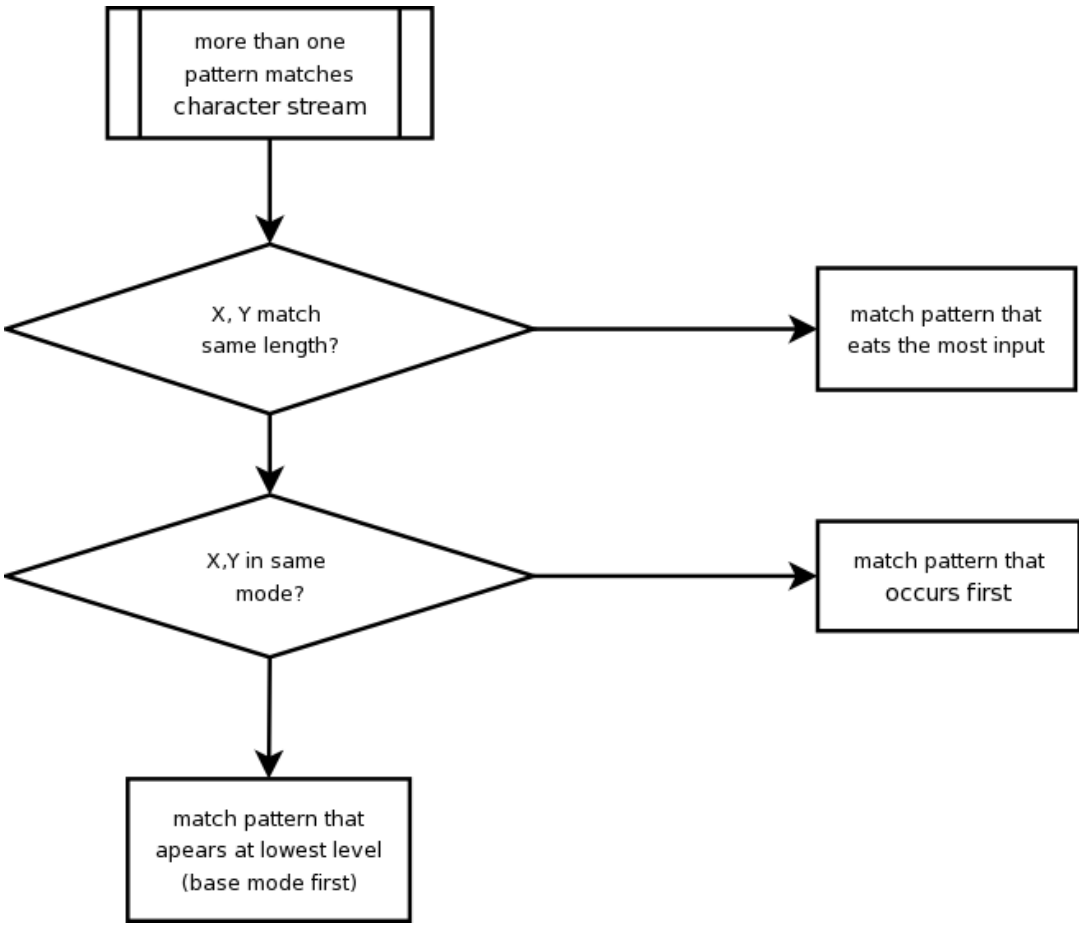

Figure 4.2.: Dispatch of pattern actions in case of concurrent matches.

```
mode my_base {
   ...
   [a-z]+ { ... /* identifier */ }
   ...
}
mode my_derived :
    my_base
{
   ...
   {"print"} { ... /* keyword */ }
   ...
}
```

When the lexical analyser is in the `my_derived` mode, then 'print' is always recognized as an identifier and never as keyword. However, if the `PRIORITY-MARK` is set as in the following code fragment,

```
mode my_base {
   ...
   [a-z]+     { ... /* identifier */ }
   ...
}
mode my_derived :
   my_base
{
   ...
   {"print"} { ... /* keyword */ }
   [a-z]+     PRIORITY-MARK
   ...
}
```

then the '[a-z]+' pattern has a priority of a pattern in the mode `my_derived` *after* the pattern '`"print"`'. The action related to the '[a-z]+' pattern, though, remains. An incoming `print` character stream is now always recognized as keyword. It cannot be overemphasized, that using priority marks allow derived modes to act 'against' the concepts of the base modes. Thus a mode B may be derived from mode A, i.e. 'is-a' mode A, but it behaves different! Priority marks are indecent and a sign of a bad design!

Priority marks can be avoided by splitting the base mode A into two modes A0 and A1, one containing desired patterns and the undesired patterns. Figure 4.3 shows this idea. The original mode can be achieved by derivation from A0 and A1. The mode B can derive from the base mode of desired patterns. This is the clean way to avoid that undesired base class patterns have to high priority —use `PRIORITY-MARK` in case of laziness.

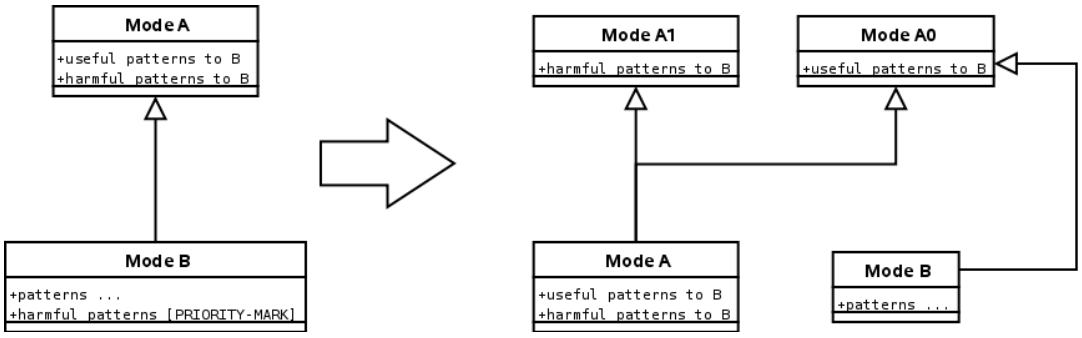

Figure 4.3.: Avoiding PRIORITY-MARK through re-design.

4.3.3. The Match Event

Queχ provides a special event is executed at the moment any match is executed. This allows the user to do some things—for example for debugging purposes— at any pattern match. The event handler name to do this is `on_match`. The code fragment that follows this name is considered to be the action which is executed before any match action is executed. Every mode has its own `on_match` event handler. However, these event handlers can be inherited in the same way as `on_exit` and `on_entry` are inherited.

The event handler receives the two arguments 'Lexeme', the pointer to the first character of the lexeme and 'LexemeL', i.e. the length of the lexeme that was matched. The following shows an example, where the user wants to do some statistics on the number of matches, the number of backslashes and the total length that was matched in the mode 'CORE'.

```
mode CORE :
...
{
    ...
    on_match {
        match_count        += 1;
        backslashe_n       += __count_backslashes_in_string(Lexeme);
        total_match_length += LexemeL;
    }
    ...
}
```

4.4. Mode Transition Events

There are exactly two events that are related to mode transitions: entering and exiting. Accordingly, one can define inside a mode the two handlers `on_entry` and `on_exit`. In addition to the `self` reference, the following variables are available:

`const quex_mode& FromMode`: Only availabe in `on_entry` where this is the mode *from* which the current mode is entered.

`const quex_mode& ToMode`: Only availabe in `on_exit` where this is the mode *to* which the lexical analyser transits.

The `on_exit` function of the mode that is left is always called before the `on_entry` function of the mode that is entered. Note, that it does not make sense inside transitions to use the `RETURN` statement since at this point the token-queue is not to be synchronized with a user's token request. Examples for enter and exit handlers have been given in section 3.4.

4.4.1. Caveat

Note, that initiating explicitly mode transition inside `on_exit` will cause an infinite recursion! If this is intended the mode transition mechanism should be circumvented using the member function `set_mode_brutally()`. Note also, that initiating an explicit mode transition inside `on_entry` *can cause* an infinit recursion, if one initiates a transit into the entered mode. Certainly, such a mode transition does not make sense. In general, mode transitions inside `on_entry` or `on_exit` event handlers are best avoided. Consider the code fragment

```
mode MY_MODE_A {
...
    {P_SOMETHING} {
        self << MY_MODE_B;
    }
...
}
```

meaning that one desires to enter `MY_MODE_B` if pattern `P_SOMETHING` is detected. Imagine, now, because of some twisted mode transitions in the transition event handlers one ends up in a mode `MY_MODE_K`! Not to mention the chain of mode transition event handler calls - such a design makes it hard to conclude from written code to the functionality that it implements. To repeat it: *explicit mode transitions inside* `on_entry` *and* `on_exit` *are best avoided!*

One might recall how uncomfortable one would feel if one mounts a train in Munich, leading to Cologne, and because of some redirections the trains ends up in Warsaw or in Moscow. In a same way that train companies should better not do this to theirs customers, a programmer should not do to himself mode transitions inside `on_entry` and `on_exit`.

Another issue has to do with optimization. Queχ is aware of transition handlers being implemented or not. If no transition handlers are implemented at all then no event handling code is executed. Note, that each entry and exit transition handler requires a dereferecing and a function call—even if the function itself is empty. For fine tuning of speed it is advisable to use *only* entry handlers or *only* exit handlers. The gain in computation time can be computed simply as: probability of mode swith times time for dereferencing and empty function call over two. For reasonable languages (probability of mode change $<$ 1 / 25 characters) consider the speed gain in a range of less than 2%. The derefencing, though, can come costy if the mode change is seldom enough that is causes a cache-miss. Thus, for compilers that compile large fragments of code, this optimization should be considered.

4.5. Indentation Events

Additionally to the `on_entry` and `on_exit` event there is an event for the `first non-whitespace character` that appears in a line: `on_indentation`. In order to trigger this event a dedicated process keeps track of newlines and whitespace that follows it. This process happens virtually in paralell to pattern matching. It knows to states:

State 1 : Newline has occured and since then, there was only whitespace.

State 2 : There was at least one none-whitespace since newline.

At the beginning of lexical analysis, or at the beginning of a line the lexical analyser is in state 1. When the first non-whitespace character appears in a line it enters state 2. The transition from state 1 to state 2, i.e. the appearance of the first non-whitespace character in a line triggers the indentation event and the correspondent event handler is called. The transition from state 2 to state 1 and state 1 to state 2 can also appear inside one single matched pattern[3].

This event handler is called right before the pattern action is executed that belongs to the matched pattern. Inside the event handler, in addition to the `self` reference, the following variable is available:

`const int Indentation` : The distance from the beginning of the line to the first non-whitespace.

Note, that the indentation event can be disabled, but only for once. Using the member function

```
void disable_next_indentation_event();
```

disables the indentation event for the next time. However, after the next prevented indentation handling it is enabled again. This comes handy if one needs to have a 'line-prolonger', such as a Backslash in python and many shell script languages, or the underscore in VisualBasic. The following mode action pair would prevent the subsequent line to be considered for indentation handling:

```
mode SOMETHING {
    ...
    {P_BACKSLASH_FOLLOWED_BY_NEWLINE} {
        self.disable_next_indentation_event();
    }
    ...
}
```

If the subsequent line, though, does not end with a backslash the event handler is automatically active and indentation handling is in place.

[3] Imagine, for example a pattern [\n]+ that matches newlines and whitespace.

4.6. Lexical Analyser Class

Queχ generates a C++ class that implements a lexical analyser as specified by the user. This section discusses the class members and member functions with which the user may interacts. Section 4.6.1 discusses the usage of the lexical analyser from outside, i.e. how the end user of the engine is going to create a lexical analyser and how a character stream is transformed into a token stream. The subsequent sections discuss how modes can be handled using member functions, token handling, the *accumulator*, and some virtual functions to be possibly overriden by a derived class. Any member function can theoretically be used in any context[4].

4.6.1. General User Interface

Before the lexical analyser can be used, one has to create an instance of it. Assuming that 'the_lexer' is the name of the lexical analyser engine (see command line option `--engine`) the following two constructors are provided:

```
the_lexer(const istream* in_stream, ostream* out_stream = 0);

the_lexer(const string& in_filename, ostream* out_stream = 0);
```

The first constructor accepts an input stream of any kind and an optional output stream for unmatched characters. The second constructor directly accepts a filename instead of a stream. The stream or the filename act as the 'root-file' for the lexical analysis. The lexical analyser is now ready to run. Tokens are received by calling the member function

```
token::id_type get_token(token* result_p);
```

It initiates a lexical analyser process and fills the object at `result_p` with the next token. Internally, tokens may be stacked and not every call to `get_token()` initiates an analysis. But the user does not care. He simply receives a sequence of tokens through this function until a token arrives with the the token-id

```
quex::token::ID_TERMINATION
```

This id is by default defined as zero, because also many parsers require that. This id tells that the input has been totally treated and no further token will arrive. In order to keep track of different versions of generated lexers the member function

```
const char* version() const;
```

returns a string telling about the version (see also command line option `--version`), and the date when queχ produced the engine. The current line and column numbers can be accessed using the functions

```
int line_number() const;
```

[4] This is, of course, only true as long as acces rights of `private` or `public` do not forbid it.

```
int line_index() const;

int column_number() const;

int column_index() const;
```

where the '*_number()' functions return the information where the last pattern started, and the '*_index()' functions return the information where the last pattern ended.

4.6.2. Mode Handling

From the given set of modes, queχ creates a set of mode-information objects as members of the lexical analyser class. The names of the members are exactly the same names as the names of the modes. In the example of the last section, the following three objects are members of the class `tiny_lexer`:

```
END_OF_FILE_tag      END_OF_FILE;
STRING_READER_tag    STRING_READER;
PROGRAM_tag          PROGRAM;
```

The three classes `END_OF_FILE_tag`, `STRING_READER_tag`, and `PROGRAM_tag` are all derived from class `quex_mode`. Thus pointers and references to them can do the job, whenever a pointer to a `quex_mode` can. The lexical analyser class provides the following functions to access information about the current mode.

```
quex_mode&        mode();
const string&     mode_name() const;
const int         mode_id() const;
```

The first function returns a reference to the mode-information object that represents the current mode. The second returns a string with the name of the current mode. Those member functions shall be accessed using the `self` member, i.e. for example

```
if( self.mode() != PROGRAM )
    cerr << self.mode_name() << endl;
```

The third function `mode_id()` returns a mode-identifier, i.e. an integer value that is unique for the current mode. This identifier is mainly used by the flex generated engine under the hood. For the end-user it is practically not relevant, but the mode-ids can be mapped to mode objects using the member functions:

```
quex_mode&    map_mode_id_to_mode(const int              ModeID);
const int     map_mode_to_mode_id(const quex_mode\& Mode) const;
```

A direct transition to another mode is initiated through the $<<$-operator or the function `enter_mode()`, i.e.

```
void & operator$<<$(const int MODE_ID);
void & operator$<<$(quex_mode& Mode);
void & enter_mode(quex_mode& TargetMode);
```

The operators return `void` deliberately because mode transitions are not thought to be done in concatination without pattern matching. If so, one can still write multiple mode transitions in a row. In any case of the three above the following three steps are guaranteed:

1. Call `on_exit` handler of current mode.
2. Set the target mode.
3. Call `on_entry` handler of target mode.

Additionally to direct mode transitions modes can be pushed and popped similar to subroutine calls (without arguments). This is provided by the functions:

```
void        push_mode(quex_mode& new_mode);
void        pop_mode();
void        pop_drop_mode();
```

The member function `push_mode(new_mode)` pushed the current mode on a last-in-first-out stack and set the `new_mode` as the current mode. A call to `pop_mode()` pops the last mode from the stack and sets it as the current mode. Note, that the mode transitions with push and pop follow the same mode transition procedure as for entering a mode directly. This means, that the `on_exit` and `on_entry` handler of the source and target mode are called. The function `pop_drop_mode()` pops a mode from the mode-stack, but does not set it as current mode. It is rather dropped and forgotten.

If one wants to avoid the call of exit and enter event handlers, then modes can also set brutally using the member functions:

```
void  set_mode_brutally(const int LexerMode);
void  set_mode_brutally(const quex_mode& Mode);
```

Using these functions only the current mode is adapted, but no event handlers are called. This also means that *mode transition control is turned off*. Inadmissible transitions triggered with these functions cannot be detected during run-time.

Mode Objects

Modes themselves are implemented as objects of classes which are derived from the base class `quex_mode`. Those mode objects have member functions that provide information about the modes and possible transitions:

```
bool  has_base(const quex_mode& Mode,
      bool PrintErrorMsgF = false) const;
bool  has_entry_from(const quex_mode& Mode,
      bool PrintErrorMsgF = false) const;
bool  has_exit_to(const quex_mode& Mode,
      bool PrintErrorMsgF = false) const;
const int     ID; \\
const string  Name; \\
```

The first three member functions allow to get information about the relation to other modes. If the flag PringErrorMsgF is set than the function will print an error message to the standard error output in case that the condition is not matched. This comes very handy when using these functions in asserts or during debugging. The functions can be applied on a given mode object or inside the on_entry and on_exit functions with the this pointer. In a pattern action pair, for example, one might write

```
if( PROGRAM.has_base(self.mode()) )
    cerr << "mode not a base of PROGRAM: " << self.mode_name() << endl;
```

For the end-user these functions are not really relevant, since queχ itself introduces asserts on mode transitions and provides convienient member functions in the lexical analyser class to access information about the current mode.

4.6.3. Token Handling

Section 2.2 elaborated on the idea that the lexical analyser communicates tokens to the user. In the queχ generated lexical analyser the tokens are stored in a token-queue before they are delivered to the caller of get_token(). Inside the pattern-action the send function group allows to send tokens:

```
void  &     send(const token& That);
void  &     send(const token::id_type TokenID);
void  &     send_n(const int N, const token::id_type TokenID);
void  &     send(const token::id_type TokenID, const char* Text);
void  &     send(const token::id_type TokenID, const int Number1);
```

Figure 2.2, page 12, displayed a sequence diagram depicting the communication of tokens. Using the send function group the writer of pattern-action pairs does not need to care about the get_token() function call. He sends tokens and leaves the responsibility of delivery to the engine.

4.6.4. The Accumulator

The accumulator is a member that allows to stock strings to communicate them between pattern-actions. In the practical example in section 3 the string contained in string delimiter marks is accumulated until the on_exit handler was activated, i.e. the STRING_READER mode is left. Insider the handler, the string is flushed into a token with a specific id TKN_STRING. The accumulator provides the following functions:

```
void   add(const char*);
void   add(const char);
void   flush(const token::id\_type TokenID);
void   clear();
```

The add-functions add a string or a character to the accumulated string, the flush() function sends a token with the accumulated string and the specified token-id. Finally, the clear() function clears the accumulated string without sending any token.

4.6.5. Class Body Extensions: `body`

Instead of deriving a class from the lexical analyser, it is sometimes more convenient to simply paste some more content into the class body. This can be done using the keyword `body` outside the mode definitions. The following code, for example, pastes the definition of an integer vector into the class' body, so that it can later be used to keep track of indentation columns.

```
body {
  std::vector<int>     indentation_stack;
}
```

Note, that the namespace prefix `std::` is used. This is because the class definition's body appears inside a header file and in header files one better uses absolute namings, rather than abbreviating using namespaces. The newly defined member can then be used inside any pattern action or event handler via access of the `self` reference, e.g.

```
mode SOMETHING {
    ...
    on_indentation {
        ...
        // close any block that has higher indentation
        while( Indentation < self.indentation_stack.back() ) {
            // send token that indicates: block-end
            self.send(TOKEN_END_OF_BLOCK);
            // cut indentation block from list
            self.indentation_stack.pop_back();
        }

    }
    ...
}
```

4.6.6. Constructor Extensions: `init`

The last section discussed how additional contents can be pasted into the defintion of the class' body that represents the lexical analyser. Naturally, those new contents need to be initialized. This happens inside the constructor of the generated lexical analyser. Queχ allows to specify additional content to be executed inside the constructor using the `init` keyword outside any mode definition. The following code fragment shows a setup as it might occur in a lexical analyser that supports indentation based statement blocks:

```
body {
    std::vector<int>  indentation_stack;
    bool              allow_opening_indentation_f;
}
```

```
init {
    // first indentation at column = 0
    self.indentation_stack.push_back(0);
    // default: do not allow to open a sub-block.
    // only function definitions, if statements, and for loops
    // shoul allow to open a new indentation block in the next line.
    self.allow_opening_indentation_f = false;
}
```

The `body` fragment adds an indentation stack as a member to keep track of line indentations. A second member variable tells if a higher indentation is supposed to appear that opens a statement block. These two variables need to be initialized. Therefore, the `init` fragment contains an initial block starting at column '0'. This indentation block cannot be closed (except by 'end-of-file') because no column can have a negative indentation. Also, by default open sub-blocks is disallowed, so the correspondent member variable is set to `false`. This code fragment will then appear in any constructor of the generated lexical analyser class.

4.6.7. Virtual Member Functions

The generated lexical analyser contains also virtual functions that do not do anything but wait for being overriden by a derived class. Those functions are

```
void    on_action_entry(const char* Pattern, const int Length);
void    on_include(const char* Filename);
void    on_include_exit();
```

Function `on_action_entry(...)` allows to write a user defined function that is executed whenever a pattern matches. The last two functions are executed at the moment when a file is included or one returns from an included file. The default implementation is that they do not do anything. Only if one derives a class from the lexical analyser, then one creates its own handlers for these events. If you do not want to use these functions at all, you are free to outcomment the definition

```
QUEX_OPTION_VIRTUAL_FUNCTION_ON_ACTION_ENTRY
```

4.7. Pasting Header Information: `header`

It is sometimes necessary to provide header file information or macro definitions to the pattern-action pairs or event handlers. To do this queχ allows to specify a `header`. This keyword has to appear outside any mode definition. It is followed by an opening curly bracket, the code to be pasted and a matching closing curly bracket.

4.8. Deriving from Lexical Analyser

The class generated by queχ implements an engine for lexical analysis with some basic functionality as was discussed in the previous sections. Sometimes, though, it is necessary to add features to the lexical analyser. Especially, adding event handlers for file inclusion and pattern matching requires to overide virtual functions of the engine. Adding new features while keeping basic functionality is best done using C++ inheritance, i.e. one derives his own class from the generated lexer class.

The name and definition of this class needs to be known for the creation of the lexical analyser. Thus, one has to specify the command line option `--derived-class` followed by the name of the class and `--derived-class-file` followed by the name of the file that contains the definition of the derived class. The derivation itself happens in the standard C++ way, i.e. one derives publicly from the lexical analyser class:

```
class small_lexer
   : public quex::tiny_lexer {
     small_lexer(const std::string& filename,
                 std::ostream*      output_stream = 0);
     // ...
};
```

Note, that the derived class has to call a base class' constructor from inside its constructor initialization list in order to propperly initialize the lexical analyser object.

```
     small_lexer(const std::string& filename,
                 std::ostream*      output_stream = 0)
        : tiny_lexer(filename, output_stream),
          // ...
       {
         // ...
       }
```

If these caveats are taken care of the user is free to create objects of his derived class and use it the same way as he used the previous plain generated class.

4.9. User defined Token Classes

Queχ provides a token-class that allows the storage of a `string` object, an `int` value. Note, that any complex structure might be transformed later on by the parser based on the given string. In some cases, though, it may be promising to implement a dedicated token class that is optimal in memory and speed with respect to a specific problem. For these cases, queχ allows the user to specify his own token class. The token queue does not require any adaption, since it is implemented as a template. The context of usages of tokens, however, imposes that it complies to some policies:

1. The token class must be specified in namespace `quex`.

2. A `typedef` inside the class must define a type for `id_type`.
3. The following token-ids need to be defined inside the class:

   `ID_UNINITIALIZED` which is preferably set to '-1' so that no interference can occur with positive valued token-ids.

   `ID_TERMINATION` which is preferably set to '0' because many parser generators require a token-id of zero to terminate.
4. A member function that maps token-ids to token-names

   ```
   static const std::string&  map_id_to_name(token::id_type);
   ```

   that maps any token-id to a human readable string. Note, that queχ does generate this function automatically, as long as it is not told not to do so by specifying command line option

   ```
   --user-token-id-file 'filename'
   ```

5. Member functions that set token content, e.g.

   ```
   void  set(token::id_type TokenID, const char*);
   void  set(token::id_type TokenID, int, int);
   void  set(token::id_type TokenID, double);
   void  set(token::id_type TokenID, double, my_type&);
   ```

   As soon as the user defines those functions, the interface for sending those tokens from the lexer is also in place. The magic of templates lets the generated lexer class provide an interface for sending of tokens that is equivalent to the following function definitions:

   ```
   void  send(token::id_type TokenID, const char*);
   void  send(token::id_type TokenID, int, int);
   void  send(token::id_type TokenID, double);
   void  send(token::id_type TokenID, int, my_type&);
   ```

   Thus, inside the pattern action pairs one can send tokens, for example using the `self` reference the following way:

   ```
   {MY_PATTERN} {
       // map lexeme to my_type-object
       my_type tmp(split(Lexeme, ":"), LexemeL);
       self.send(LexemeL, tmp);
       RETURN;
   }
   ```

As long as these conventions are respected the user created token class will interoperate with the framework smoothly. The inner structure of the token class can be freely implemented according to the programmer's optimization concepts. Note, that the name of the token class is also of free choice. When invoquing queχ, the command line option `--token-class` needs to be followed by the user defined token class name. The command line option `--token-class-file` tell queχ the name of the while where this class is defined. As long as this option are not defined queχ will not consider user defined token classes and provide the standard token class.

4.10. Summary of Code Sections

Previous chapters and sections introduced different code sections where patterns, token identifier, modes, and engine code could be specified. The following table summarizes these section in order to serve as an overview. It associates the code section's keyword with its content and the section and page reference of its explanation.

Keyword	Task	section	Page
`mode`	Pattern-action pairs, i.e. the 'behavior' of the lexical analyzer.	4.2 4.3	39 41
`define`	Pattern name defintions, i.e. naming of regular expressions to be used in pattern-action pairs.	3.1	19
`token`	Definition of token-ids.	3.2	20
`header`	Code to be pasted above the engine class.	4.7	54
`body`	Code to be pasted into the engine class definition.	4.6.5	53
`init`	Code to be pasted into the engine's constructor.	4.6.6	53

For convinience, even though, it is not a 'region', let it be mentioned here, that he assignment

```
start = MY_MODE
```

defines the starting mode of the lexical analyzer. If no starting mode is explicitly defined queχ assumes the first mode defined in the source code as starting mode and produces a warning of the console.

4.11. Macro Switches

When queχ creates a lexical analyser it leaves some switches in the code that can be turned on and off in order to control certain charactersistics. First of all, queχ makes heavy use of `asserts` to make sure that the code is executed appropriately. These asserts can be turned off by defining the `NDEBUG` macro. The following macros are queχ specific and can be specified with the '-D' option of your compiler:

`DEBUG_QUEX_MODE_TRANSITIONS` : If defined, the lexical analyser engine displays any mode transition on the standard error output.

DEBUG_QUEX_PATTERN_MATCHES : If defined every pattern match is going to be displayed on the standard error output. Note, that if the lexical analyser is created with option '--debug no' (see section 4.12), then the source code does not contain the code fragments that activate such a printout.

DEBUG_QUEX_TOKEN_SEND : If defined, then every token being sent using the send function group is displayed on standard error output.

QUEX_FOREIGN_TOKEN_ID_DEFINITION : If defined the definition of token-ids, i.e. their numeric values is taken from an external file. This file needs to be specified using the command line option --foreign-token-id-file. Foreign token-id definitions make sense, for example, if the parser generator insists on defining token-ids. If this macro is not defined the given external file is ignored and queχ-generated numerical values are used.

QUEX_NO_SUPPORT_FOR_COLUMN_NUMBER_COUNTING :

QUEX_NO_SUPPORT_FOR_LINE_NUMBER_COUNTING : By defining these macros the user can turn off the column number and line number counting. The analyser may run a little faster then. Note, that in this case the member functions for column number and line number access are no longer available.

4.12. Command Line Options

Main options to control the behavior of the generated lexical analyzer are:

"-i", "--mode-files" *file-list*: default=""

file-list = list of files of the file containing mode definitions (see sections 3.3, 3.4, and 4.6.2)

"--token-prefix" *name*: default=TKN_

name = Name prefix that is to be sticked in front of any name given in the token-id files. For example, if a token-id file contains the name 'COMPLEX' and the token-prefix is 'TOKEN_' then the token-id inside the code will be 'TOKEN_COMPLEX'.

"--token-offset" *number*: default=10000

number = Number where the numeric values for the token-ids start to count.

"--version-id" *name*: default=0.0.0-pre-release

name = arbitrary name of the version that was generated. This string is reported by the version() member function of the lexical analyser.

"--foreign-token-id-file" *filename*: default=""

filename = Name of the file that contains an alternative definition of the numerical values for the token-ids (see also section 4.11).

"-o", "--engine" *name*: default=lexer

name = Name of the lexical analyser class that is to be created inside the namespace quex. This name also determines the filestem of the output files: *name*, *name*-internal.h, *name*-token_ids, and *name*.cpp.

"–debug" False If provided, then code fragments are created to activate the output of every pattern match. Then defining the macro DEBUG_QUEX_PATTERN_MATCHES activates those printouts in the standard error output. Note, that this options produces considerable code overhead. If 'no' is specified, then no such code is created.

"–no-mode-transition-check False Turns off the mode transition check and makes the engine a little faster. During development this option should not be used. But the final lexical analyzer should be created with this option set.

For the support of derivation from the generated lexical analyzer class the following command line options can be used.

"--derived-class" *name*: default=<none>

name = If specified, name of the derived class that the user intends to provide (see section 4.8). Note, specifying this option signalizes that the user wants to derive from the generated class. If this is not desired, this option, and the following, have to be left out.

"--derived-class-file" *filename*: default=""

filename = If specified, name of the file where the derived class is defined. This option makes only sense in the context of option --derived-class.

"--friend-class" *name-list*: default=

name-list = Names of classes that shall be friends of the generated lexical analyser. This is to be used, if other classes need to have access to protected or private members of the analyser. It is only to be used by specialists.

Additionally, there are options for specialists who want to provide their own token-class:

"--token-class" *name*: default=token

name = Name of the token class that the user defined. Note that the token class needs to be specified in namespace quex.

"--token-class-file" *filename*: default=$(QUEX_PATH)/code_base/token

filename = Name of file that contains the definition of the token class.

Even if a non-quex token class is provided, still the token-id generator may be useful. By default, it remains in place. The user, however, can specify the following option to disable it:

`"--user-token-id-file"` *filename*: default=<none>

filename = Name of file that contains the definition of the token-ids and the mapping function from numerical token-ids to `std::string` objects, i.e. human readable names.

There may be cases where the characters used to indicate end-of-stream and begin-of-stream need to be redefined, because the default code points appear in a pattern[5]. Also, note that the '.' regular expression (meaning 'nothing but newline or end of file') needs check for begin-of-buffer and end-of-buffer in the general case. Giving both the same value may come with some speedup, and does not hurt. None of these values should be zero, because this is the buffer limit delimiter. Then the following options allow to modify those values:

`"--begin-of-stream"` *number*: default=`0x19`

`"--end-of-stream"` *number*: default=`0x1A`

The numbers following these options can be either decimal, without any prefix, hexadecimal with a '0x' prefix or octal with a '0o' prefix. If the trivial end-of-line pre-condition (i.e. the '$' at the end of a regular expression) is sused and the lexical analyzer has to run on a system that codes newline as hexadecimal '0D.0A' then you need to specify the following option:

`"--DOS"` : default=`False`

For unicode support it is essential to allow iconv support []. For this the iconv library must be installed on your system. On Unix systems this library is usually present. If a coding other than ASCII is required, specify the following options:

`--iconv` : default=`False`

Enable the use of the iconv library for character stream decoding. This option is a *must* for unicode support.

`--bytes-per-ucs-code-point", "-b"` [*1, 2, 4*]: default=2

With this option the internal representation of character is specified. It determines the byte number per character which compose the lexeme strings and on which the lexical analyzer engine internally operates. The byte number should at least suffice to carry the desired input coding space. You can only specify 1 byte, 2 byte or 4 byte per character.

`--endian` *"little", "big"*: default="<system>"

There are two types of byte ordering for integer number for different CPUs. For creating a lexical analyzer engine on the same CPU type as queχ runs then this option is not required, since queχ finds this out by its own. If you create an engine for a different plattform, you must know its byte ordering scheme, i.e. little endian or big endian, and specify it after '`--endian`'.

[5] As for 'normal' ASCII or Unicode based lexical analyzers, this would most probably not be a good design decision. But, when other, alien, non-unicode codings are to be used, this case is conceivable.

Note, that for well tempered patterns this should be safe even on Unix machines. The generated lexical analyzer, though, might be a litter slower. For version information pass option '`--version`' or '`-v`'. The options '`--help` and '`-h`' are reserved for requesting a help text.

A. Queχ Intern: Generation of the Core Engine

The queχ program is provided in source code with an LPGL license in order to support understanding the process of lexical analysis in its practical application. The source code may be used for teaching or to enhance the functionality or performance of the software. In order to allow programmers to get into queχ' source code programming, the following sections describe the way that queχ generates lexical analysers step by step. Clearly, this chapter is a hacker's guide to queχ. Users interested only in the application of queχ may skip this chapter without hesitating.

The generation of the core engine is separate from the handling of modes and their relationships. In the following sections, only the generation of the core engine, i.e. a lexical analyser for a single mode, is explained. Each mode consists of such an engine. The aggregation of engines for each mode to one single analyser engine is trivial and not subject of further explanation[1]. For further support to get into programming, the unit tests for each sub-module in the source code may be considered. They can be found in the `./TEST/` sub-directory of each module.

Note, that the following operations are only applied if the core engine is not to be generated by the `flex` program. If the core engine is to be produced by flex then queχ only translates the modes into source code readable by `flex`, receives its output, and adapts some headers so that the `flex`-produced engine fits into the queχ's framework. The engine produced by queχ, though, is a directly coded engine, rather than a table driven approach. It is more suited for the handling of unicode and provides advantages in terms of system memory usage and calculation speed.

A.1. Introduction

The following sections elaborate on the process of generating code for a directly coded lexical analyser. Each lexical analyzer mode consists of such an independent code fragment embedded into a function. In a first section, the big picture is discussed. This gives an overview about the major issues involved and build a basis for later sections to refer to. Many books discuss in detail the Thomson-construction and the Hopcroft-Optimization, but few (the author does not know a single one) discusses the construction down to code-generation and its specialities in detail. Queχ was created to handle efficiently Unicode character sets. As a consequence the table-driven approach, such as in lex [] is no longer

[1] The switching between modes is not much more than the bending of a function pointer to the mode's analyzer function.

suitable[2]. Thus, queχ implements a *directly coded lexical analyser* where no state variable is needed and no character-trigger table. It is also a much faster approach, since no memory has to be referenced[3].

A.1.1. The Big Picture

In this section the overal lexical analyser is discussed. This is done before the detailed construction is dealt with, in order to introduce basic ideas that influence the details of the process of creating a lexical analyser. In particular concepts such as pattern priviledges, pre- and post-conditions, as well as state origins are discussed at this place. The lexical analyser to be generated by queχ is a state machine. In this text the following notation for different states of a state machine is chosen:

- A single circle signifies a state, but not an acceptance state.
- A double circle signifies an acceptance state. If a state machine is in this state this means that the pattern matched.
- A double circle where the inner circle is a dashed line indicates a state that is an acceptance state under the condition that pattern's post-condition is met.
- An annotated arrow from one state to another signifies a conditional transition. The character written close to the arrow tell about the input characters that have to occure in order for the transition to be accomplished.
- Any character that is not mentioned as labels to the state transition arrows means that the state machine is left. If the state was an acceptance state the pattern is considered to be matched – otherwise not.
- A flag raised at a state indicates its origin. The combined state machine webs all patterns of a lexical analyser together. This graphical mark allows to keep track of the original patterns involved.

Figure A.1 shows an example of three patterns that are to be matched against: A keyword `WHILE`, a label, i.e. a character sequence terminated by a colon, and an identifier, i.e. a character sequence. An incoming character stream triggers step by step the transition in each of these state machines. Some may bail out. Other may reach the acceptance state. If one state machine reached the acceptance state, one can say that the pattern it represents has matched. For example a character sequence `H`, `E`, `R`, `E`, `':'` lets state machine 0 fail at the first character, state machine 1 reaches acceptance at character ':' and state machine 2 reaches acceptance once character before. Here an essential concept has to be mentioned:

[2] In table driven analysers the table size is the square of the number of characters. Unicode provides currently \$10.FFFF code points. A table for this character set would blow the memory of todays computers (The author writes these lines at the beginning of the 21^{st} century). Compressing the table might slow down the speed of the lexical analyser significantly.

[3] Note, that an expression such as '`vector[i]`' implies generally a multiplication (`offset = i * sizeof(type)`), an addition (`adress = vector + offset`), and a memory referencing operation (`object = *adress`).

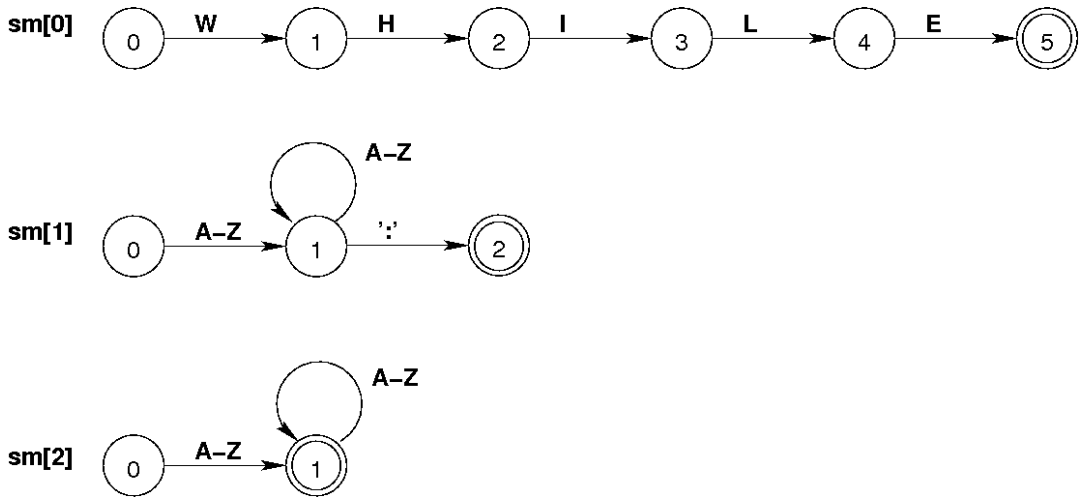

Figure A.1.: Three example patterns implemented as state machines

> **Longest Match:** Queχ produces a lexical analyser that eats as many characters as possible. When a pattern can match more incoming characters than another, it dominates.

This approach is very logical, since otherwise it was impossible to have a keyword `End` and `EndIf` for example. If the shortest match wins, then always `End` is detected, but never `EndIf`. Practically in the example above this means that we do not stop when state machine 2 enters acceptance, but we give state machine 1 a chance match a longer pattern.

Now, consider the character sequence `W`, `H`, `I`, `L`, `E`, `'('`. State machine 1 fails without ever entering acceptance at the last character. State machine 0 enters acceptance at the last `E`, and so does state machine 2. Both match the same amount of characters, but a lexical analyser can only report the detection of one pattern at a particular position in the character stream. There is no general logical rule here that can be applied to determine dominace of one pattern over the other. At this point the decision must be made by the user. Usually, this happens by sequence, i.e. the pattern that was defined first, matches in case of equal length[4].

> **A Pattern Precedence** is a user- (or inheritance-)determined relation between two patterns. It is a statement that tells what pattern preceeds if both match the same amount of characters.

[4]Note, that we are talking here on the 'state machine engine' level. The rules for pattern precedence in derived modes as mentioned in section A.2 are actually translated into priviledge rules on the engine level. The design of queχ is layed out in a way for using a database that maintains pattern-precedencies. However, practically this has no advantage, since base mode precendencies can be expressed by the order that patterns are defined. This means, that pattern that are inherited from based modes simply have to be converted into state machines before the derived mode's patterns. This way it is guaranteed that the base mode's pattern state machines have lower ids and higher precedence.

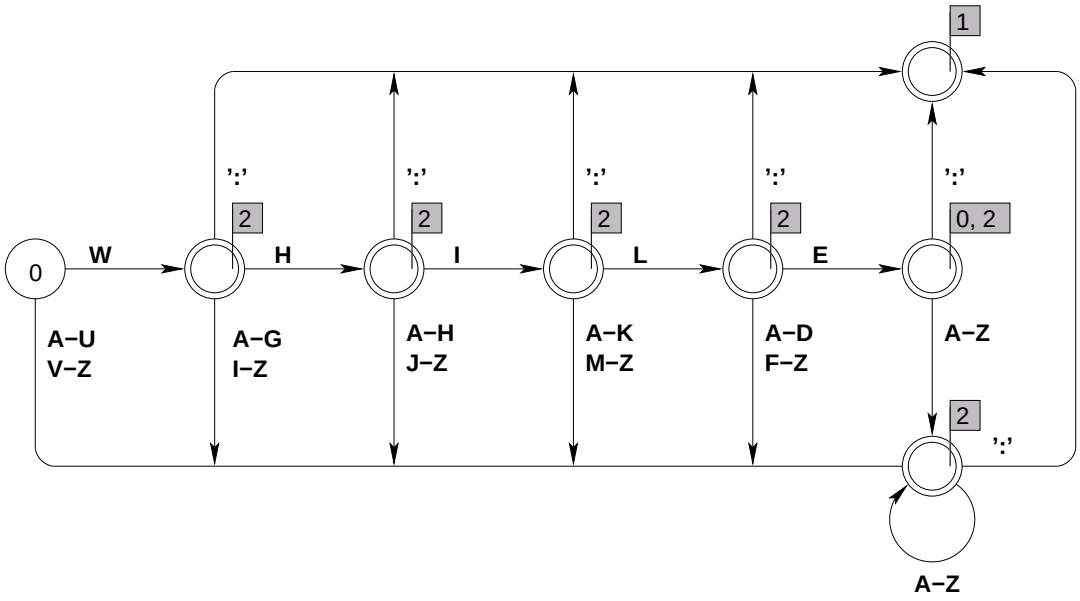

Figure A.2.: State machine matching agains all three patterns from figure A.1

A Dominating Pattern is the pattern in a set of pattern that can match in a particular state, which 'wins', i.e. that has the highest precedence.

The construction of a lexical analyser includes the construction of a single state machine that is equivalent to the bank of patterns which were shown in figure A.1. Figure A.2 shows a state machine which is the result of such a construction. In particular, this involves a construction of a deterministic finite automaton (DFA) from the pattern bank, which is a non-deterministic finite automaton (NFA). The pattern bank allows that the state machine is in multiple states at the same time, a DFA is only in one particular state at a particular time. The state machine that results from this operation is still to be optimized. A so called Hopcroft optimization [] reduces the number of states to a minimum. The result in the abovementioned case is the state machine of figure A.2.

Queχ allows to define patterns that trigger only in a certain environment of characters. The ability to define borders of a pattern that are not part of the pattern itself is implemented by means of so called pre- and post-conditions. The following to subsections introduce these two concepts and how they are modelled by queχ.

Post-Conditions

A post-condition is a condition on the character stream after the 'real' pattern has matched. When the 'real' pattern notifies a match at position X, the analyser needs to proceed in the character stream until it determines whether the subsequent characters match the post-condition. If the post-condition is met it reports 'acceptance'—however, the lexical analyser's

focus is put back to position X.

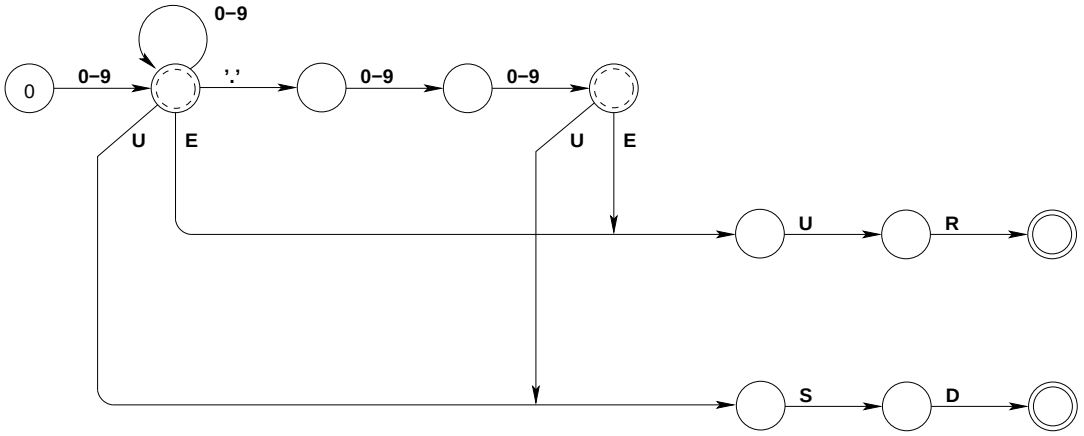

Figure A.3.: State machine matching a post condition pattern.

An example of a post-condition is shown in figure A.3. The rule for this pattern is that when a number comes in that is followed by a currency, then it has to be reported as *amount*, rather than a number. In this example there are two currencies: EUR for 'Euro' and USD for 'US-Dollars'. When a number is matched—with or without a floating point—then we reached the position to which we want to step back, if it was an *amount*. These acceptance states have a dashed inner circle, to indicate that the input position has to be marked, but we're not finished yet. Only after one of the currencies `EUR` or `USD` appeared, finally, an acceptance state is reached. But, the following analysis needs to start at the position, backwards, where the number matched.

Figure A.4 displays the sequence of actions for an incoming character stream. When the `12` has matched as a number the input position '2' is stored but the analysis continues. When `USD` has matched it is now determined, that the previous number was indeed an amount. A token 'AMOUNT(12)' can be reported. But, but one needs to go back to the position where the amount ended, i.e. to position '2'. From there, it detects a currency `USD` and reports a token 'CURRENCY("USD")'. The same thing happens now with the number `45.10` and the currency `EUR` in the remainder of the stream.

Post-conditions are implemented by setting a 'store-input-position mark' at the states where the 'real' pattern matches. The end of the post-condition is an acceptance state. Figure A.5 depicts the states of the amount/currency pattern webbed into a state machine that contains other patterns. The process of combining and optimizing the state machine needs to make sure, that the information about saving input positions and acceptance states remains—here indicated as notes on a flag.

Pre-Conditions

A Pre-Condition is a condition on the character stream before the pattern is actually being parsed. Starting from the current position X, the analyser needs to go backwards to check

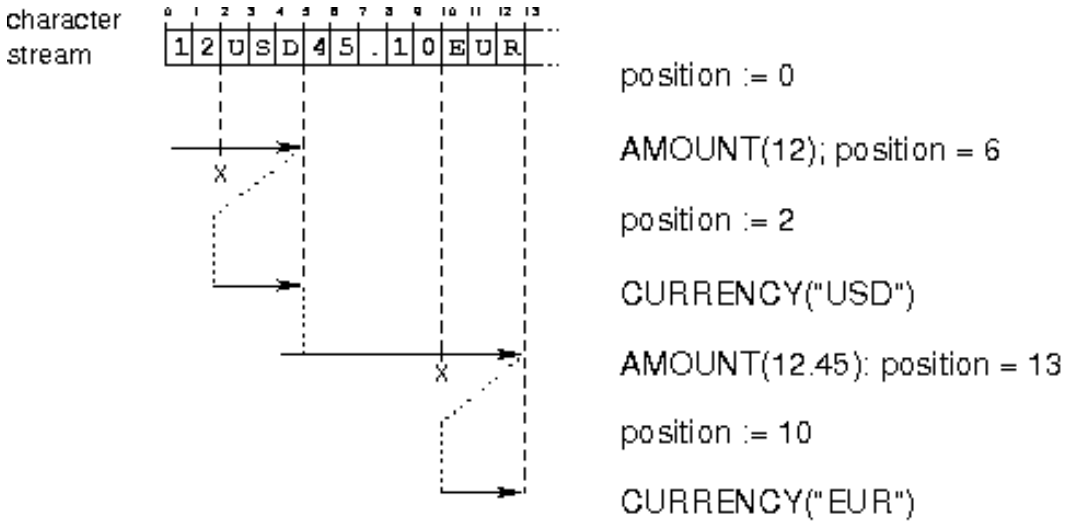

Figure A.4.: Sequence of analysing steps detecting a post-conditioned pattern.

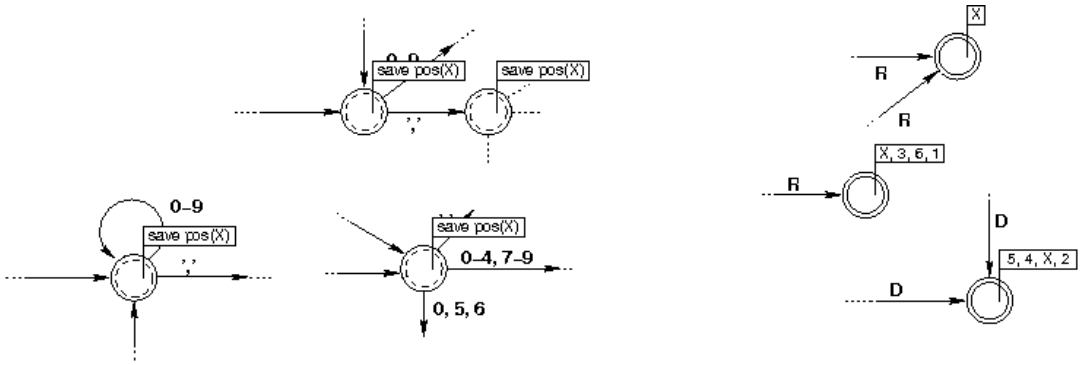

Figure A.5.: States of a post-conditioned pattern in the combined state machine.

whether the pre-condition is met. After determining whether it is met or not, the 'real' pattern is analysed starting from position X.

Intuitively, it does not make sense to try to match against a pattern if it is known from the beginning that the pre-condition is not met. For this reason, it is tempting to consider a single analyzer function for each case of 'pre-condition met' and 'pre-condition not met'. With this approach, though, the total number of analysers to deal with N pre-conditions is 2^N since any combination of 'pre-condition met' and 'pre-condition not met' has to be considered. Very few pre-conditions would then blow the number of analysers beyond what is practical. Queχ could not be a general solution for pre-conditions.

Rather than creating an analyser for each pre-condition constallation, one single analyser containing all pre-conditioned and non-pre-conditioned patterns is created. Pre-conditions are simply implemented by a conditional acceptance at the end of the pattern, i.e.

```
// acceptance state ...
if( pre_condition_X_fulfilled_f ) last_acceptance = X;
...
```

When the analyser reaches the acceptance state of a pre-conditioned pattern it notifies a match only under the condition that the pre-condition flag was raised before. From one point of view, this is inefficient since in some cases the analyser tries to match against a pattern, even though the pre-condition disallows an acceptance. On the other hand, the states of the pre-conditioned patterns are webbed altogether into one single state machine and the overhead for doing so is minimal.

> **TODO:** It would be an improvement to allow the user control when a pre-conditioned pattern constallation is webbed into the combined state machine and when not.

A.1.2. Generation Steps

In detail, the process of generating a lexical analyser consists of the following steps:

1. Constructing state machines for a each specific pattern (section A.2, 70). There are three mechanism involved, known as the Thomson Construction [].

   - Sequentialization to implement chains of patterns (section A.2.1, 71).
   - Parallelization dealing with patterns are bound by a logical 'or' (section A.2.2, 72).
   - Repetition to implement repetitions of patterns. Repetitions can have minimum and maximum bounds of repetition (section A.2.3, 72).

   By means of these mechanisms it is possible to construct a state machine representing a regular expressions. However, the resulting state machine contains possibly ϵ-transitions and is therefore a non-deterministic finite state automaton (NFA). In order to reduce complexity and to speed up the subsequent processing the state machine representation is optimized. The optimized state machine is achieved by

- a conversion from a non-deterministic finite automaton NFA to a deterministic finite automaton (section A.4, 76), a DFA, and
- a so called Hoproft Optimization [] to reduce the state set (section A.5, 80).

These two mechanism, though, are involved again in the later process of generating a single lexial analyser for all patterns involved.

2. Labeling states of the pattern's state machines with 'origins' (section ??, ??). This is necessary, since the next step will combine all state machines into a single one. It must be possible to determine at each state of the state machine to know to which states in the patterns it relates. Especially, the acceptance states need to identify what pattern actually triggered acceptance.

3. Parallelization of pattern state machines (section A.2.2, 72). This builds the combined state machine checking for all patterns at the same time.

4. Conversion of the combined state machine from an NFA to a DFA (section A.4, 76).

5. Minimization of the state set using Hopcroft Optimization (section A.5, 80).

6. Code generation for the given combined state machine (section A.6, 83). This consist of generation of code for

   - for the state transitions of each state involved.
   - for the acceptance states.

   Additionally a framework must be created that iterates over a given character stream, i.e. a buffer management module. This module is described in chapter (section ??, ??).

A.2. Thomson Construction

The lexical analyser that queχ creates is based on regular expressions. In order to create state machines that represents regular expressions, it is necessary to provide operations that combine elementary regular expressions through *concatenation*, *parallelization*, *optionality* and *repetition*. This section discusses how this is accomplished by means of the so called *Thomson Construction* [].

An essential concept is the so called ϵ-transition. By means of this special transitions state machines can be glued in sequence and in parallel. It builds a very basic concept in the Thomson construction for developing a state machine that represents a regular expression.

> **An ϵ-transition** is a transition that does not require any input. If a state A is connect with a state B via an ϵ-transition, then entering state A implies that the state machine enters state B.

In a sense, an ϵ-transition behaves like a 'free-ride' where no input character is necessary to trigger a subsequent state. However, those transitions imply that the state machine become non-deterministic, since it can now be in multiple states at the same time.

A very important principle which facilitated writing algorithms on state machines was that every state-id in every state machine is *unique*. That is, a state machine A cannot have a state-id that also appears in B. Further, state machines are mere *collections* of states. States are mere *mappings* that describe what subsequent state is reached by what trigger. Operations on state machines can then be applied by setting transitions, i.e. changing the mappings of states, and finally collecting all mappings (i.e. states) in a single state machine. The uniqueness of state-ids ensures consistency.

The following sections discuss the elementary operations on state machines that are necessary to describe regular expressions.

A.2.1. Mounting State Machines in Sequence

Mounting two state machines together in sequence is fairly simple. If state machine B is to be mounted after state machine A, then the following is to be done:

1. Clone both state machines, to make sure that we have unique state-ids inside the two machines, and operations won't influence the originals A and B.

2. Mount an epsilon transition from all acceptance states of A to the initial state of B.

3. Disable acceptance of all acceptance states in A.

4. Collect all states of A and B into a single state machine.

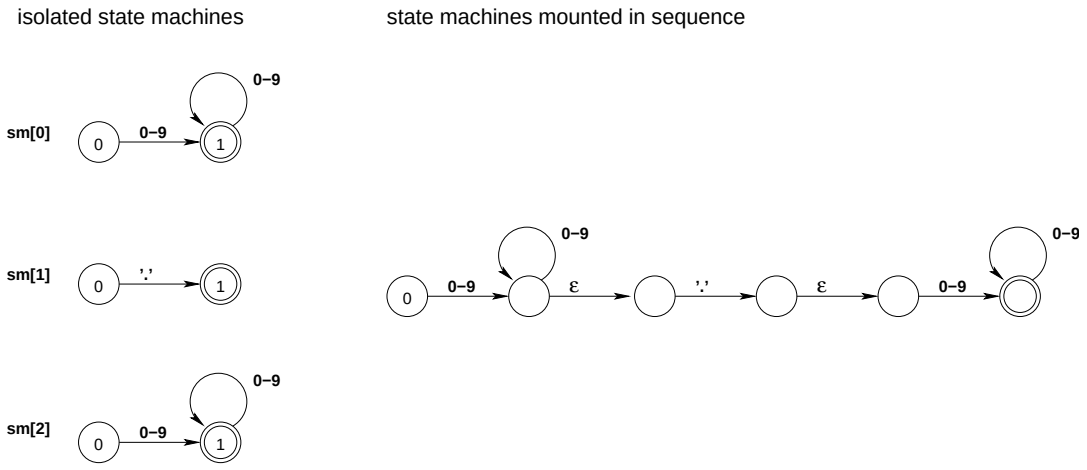

Figure A.6.: Mounting two state machines in a sequence.

These steps imply that the acceptance state of the last state machine remains. In Queχ an algorithm is implemented that does the sequentialization for any number of state machines—not just two—in one beat. In figure A.6 a state machine is created representing a 'matcher' for floating point numbers out of the sequentialized patterns [0-9]+, ., and pattern[0-9]+.

A.2.2. Mounting State Machines in Parallel

In order to implement the logical 'or' operation in regular expressions it must be possible to mount in parallel. Therefore, all acceptance states of the original state machines to be mounted in parallel need to remain acceptance states. All initial states need to remain initial states. However, there can be only one initial state. At this point the ϵ-transition comes handy again. The following steps describe the process of mounting state machines in parallel:

1. Clone both state machines (see ??).
2. Mount an epsilon transition from all acceptance states of all state machines to a new terminal state T. Let T be an acceptance state.
3. Disable original acceptance of all acceptance states in all state machines.
4. Create a new initial state I and mount an ϵ-transition to all initial states of all state machines.

Figure A.7 shows the result of combining three state machines in parallel. By means of this procedure there is only one initial state and one terminal acceptance state for the state machine. Now, regular expression such as `"hello"|"bonjour"` can be implemented as two parallel state machines for `"hello"` and `"bonjour"`.

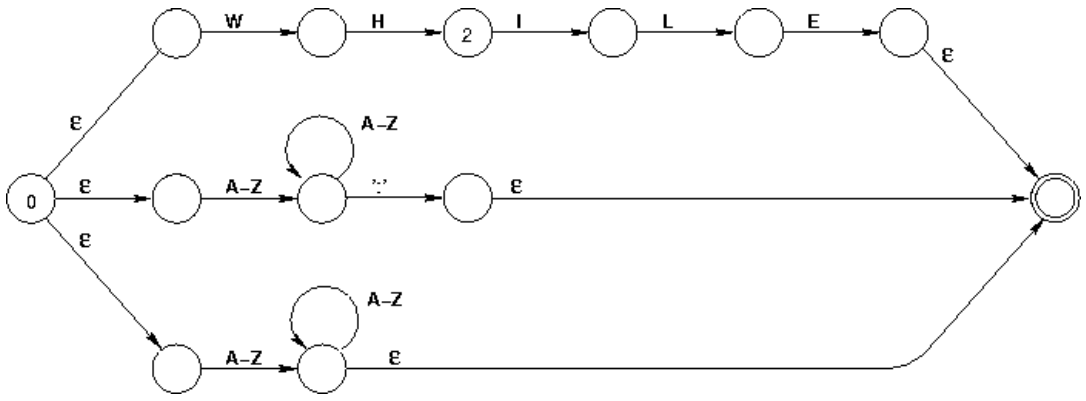

Figure A.7.: Mounting two state machines in parallel.

A.2.3. Implementing Repeated State Machines

Repetition must govern the following scenarios:

`R*`: repetition of a pattern `R` zero or an arbitrary number of times.

`R+`: repetition of a pattern `R` one or an arbitrary number of times.

`R{x}`: repetition of a pattern exactly `x` times.

`R{x,y}`: repetition of a pattern at least x and at maximum y times.

`R{x,}`: repetition of a pattern an arbitrary number of times, but at least `x` times.

`R?`: repetition of a pattern `R` zero or one time.

The first case `R*` is the so called 'Kleene-closure' []. Again, by means of the ϵ-transition this can be achieved by the following steps:

1. Create a new initial state and a new terminal state. The new terminal state is an acceptance state.
2. Mount the new intial state and the new terminal state via ϵ-transition to the old initial state and the old acceptance states. The old acceptance states remain acceptance states.
3. Connect all acceptance states backwards to the old initial state via ϵ transition. This way the same character sequence can be entered in the state machine again.
4. Connect the new initial state forwards to the new terminal state via ϵ-transition.

Figure A.8 shows the example of a Kleene-Closure for the pattern `[A-Z]+":"`, which would match a C-style label. The other forms of repetitition are closely related to that. The one or arbitrary repetition `R+` can be achieved by mounting the same state machine in front of the Kleene closure, ensuring that the pattern has to match at least once. Similarly minimum repetitions are achieved by mounting the state machines N times in front of the Kleene-Closure, where N is the minimum number of times the pattern has to be matched.

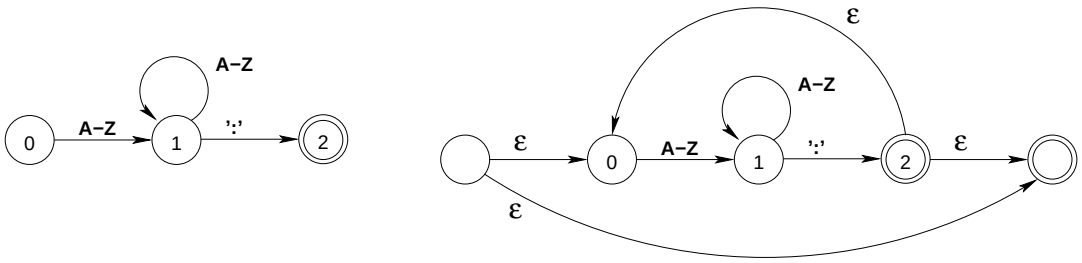

Figure A.8.: Pattern repetion by Kleene Closure.

Where a maximum number M of repetitions is involved, one cannot rely on the Kleene Closure anymore. The state machines have to be mounted M-N times in sequence, where the acceptance states in between remain acceptance states. This way one can be sure that any number of matches from N to M corresponds to an acceptance state.

A special case is the case of where zero and a maximum number of repetions are involved, i.e. `R?` and `R{x,y}` with $x = 0$. In this case, simply the initial state is raised to an acceptance state, and then the usual algorithm for generating a maximum number of repetitions is applied.

A.3. Labeling State Origins

The previous sections discussed how to generate a state machine for one isolated regular expression using the Thomson Construction. The final lexical analyzer, though, needs to have all possible patterns present, i.e. combined into an automaton in order to judge which pattern matches from the current input position. The approach that was chosen in queχ was to produce a single deterministic automaton from the set of state machines for each pattern. This process is described in the following sections.

It was already discussed that information about acceptance (section 2.1, 9), storing of input positions(section A.1.1, 66) and dependence on pre-conditions (section A.1.1, 67) need to be stored as hidden state variables[] inside the states of the isolated pattern state machines. Those informations need to be maintained when the states dissolve into the 'super state machine' that contains all patterns. This happens by labeling them with origin information.

As for acceptance, no special information carrier is required since the Thomson Construction and the Hopcroft Optimization treat them appropriately. A state in the 'super state machine' has acceptance if one of its original states had acceptance and it does not have acceptance if none of its original states had acceptance. It remains to store information about which pattern has raised the acceptance in a particular acceptance state, i.e. some type of pattern-id.

Without post-conditions an acceptance state stands for storing the input position. If the pattern that raised acceptance at this point wins, then the analyzer's focus must be set back to this position and the next analysis starts from this point. With post-conditions acceptance and storing the input position are no longer the same. When the core pattern has ended, the input position must be saved. But, at this point the state machine has not yet reached acceptance. It must first reach the end of the post condition. Then the input position can be put back to the end of the core pattern. Thus a flag is required telling wether or not it is required to store the input position.

The input positions for different post-conditions must be stored in different variables, since only at the end of the match it can be decided which pattern actually succeeded. A flag indicating that the original pattern was post-conditioned tells the code generator to use the pattern-id as indicator for the variable storing the input positions.

The existence of pre-conditions imply the check at an acceptance state if a particular pre-condition has been met. Thus a variable is required containing the pre-condition which has to be met in order to trigger acceptance. A special pre-condition is the begin-of-line pre-condition. It is a trivial pre-condition, since it does not involve an inverse state machine. Thus it is treated as a separate flag.

Inside Queχ, there is a data structure dedicated to store this information, called class `StateOriginInfo`. It contains the following fields:

`state_machine_id` carries the id of the original state machine. This identifies the pattern to which the state belongs.

`state_index` contains the index of the original state.

`store_input_position_f()` gives information about the input position to be stored or not. In the table below referenced as **S**.

`post_conditioned_acceptance_f()` indicates whether the origin of the state relates to a post condition. In the table below referenced as **post**.

`pre_condition_id` indicates the id of the pre-condition that has to be fulfilled so that the acceptance of the state is valid. In the table below referenced as **pre**.

`trivial_precondition_begin_of_line_f` indicates wether the pattern is supposed to start at the beginning of a line, i.e. after newline or the start of the buffer. In the table below referenced as **bol**.

Note, that there is nothing like a flag to indicate 'trivial post-condition end of line'. This is for the simple reason that this 'trivial' post condition is internally translated into a post-condition 'end of line or end of file'. There is no mechanism that could gain some speed here. For the begin of line pre-condition though, one can rely on the last character that was considered.

A state in the 'super state machine' contains a list of origins, each origin corresponds to an original state of the isolated patterns. The following table shows the possible combinations of these parameters together with the acceptance flag **AccF** of the state that may be labeled with these origins:

Original State	State containing the Original State One of its Origins				
	AccF	**S**	**post**	**pre**	**bof**
normal acceptance state	must	yes	—	—	—
normal non-acceptance state	?	—	—	—	—
end of post-conditioned core pattern	?	yes	yes	—	—
end of post-condition	must	—	yes	—	—
acceptance of pre-conditioned pattern	must	yes	—	yes	no!
acceptance of begin-of-line pre-conditioned pattern	must	yes	—	no!	—

Note, that 'must' for the acceptance flag means that the state in the final state machine that contains this origin *has to be* an acceptance state. The '?' means that in the original state machine the state was not an acceptance state, thus the state that contains it together with others may or may not be an acceptance state. Note also, that pre-conditioned patterns and trivially pre-conditioned patterns are mutually exclusive.

In order to help to get easy access to the origin's state type the member function '`type`' returns direct information about the type of the original state, i.e. if it was a normal acceptance, a non-acceptance, a post-conditioned state (end of core pattern), an end of a post

condition or a pre-conditioned acceptance state. In case the constitution does not lead to any valid interpretation, the function replies with an error. This would indicate that some functions do not work propperly. This allows for a double-check that all origins have passed the state machine construction propperly.

A.4. Subset Construction—NFA to DFA

The conversion of a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA) brings the advantage that at a particular point in time the state machine is in one particular state—not in multiple states as with the NFA. This not only reduces complexity, but also decreases memory requirements (to store state indices and pattern match-management) and computation time of the resulting lexical analyser. The key to do so is to combine states that the NFA takes simultaneously into a single state. Whenever a state in the NFA triggers on the same character to more than one state, the states that are reached by this trigger are combined into one single state. The process relies on two sub-processes described in this section: 1) construction of a so called epsilon closure, and 2) determining unique state combinations for the range of all triggers.

The core idea of this process is to find all possible super states of the NFA, i.e. the state combinations that can exist at a particular point in time. Then trigger sets have to be found that trigger the transition from one state combination to the other. The state combinations can then be 'renamed' as a single state and it is sure that the resulting state machine is deterministic. By requesting that the acceptance state of a combined state is acceptance as soon as one of its 'member states' is acceptance the entire mechanism of the NFA is preserved.

The notion of ϵ-triggers enabled a very efficient method to sequentialize, parallelize and repeat state machines. As soon as a state machine contains an ϵ-trigger it becomes a NFA. If a state A transits on an ϵ trigger to a state B, then the state machine is in state A *and* B simultaneously as soon as A is reached, because an ϵ-trigger does not require any input. This is one reason why the result of those operations is an NFA. In order to help with the ϵ-transitions the ϵ-closure is defined:

> An **Epsilon Closure** of a state n is the aggregation of all states that can be reached from n via an epsilon transition.

Consequently, when treating a state, one first collects all states that can immediately be reached via ϵ-transition into a new state. This is shown in figure A.9. This new state $n0$ contains all transitions of all states of the ϵ-closure. It is supposed to be equivalent to the super state of the NFA which is at the same time in all of the states of the epsilon closure. Note, however, even though state 23 in the figure is part of the ϵ-closure, as a target state it is kept isolated refrerred with its 'maiden name' rather then as part of $n0$ [5].

The next step consist in finding state combinations and the trigger sets by which they are reached from the new state $n0$. Figure A.10 shows an example of a state $n0$ where

[5] From a super state point of view, state 23 which is reached through the recursion from itself, is not the same as the state 23 being entered from state 0. Also, $n0$ as being a 'super state ' does not belong to the same 'world' as the original states.

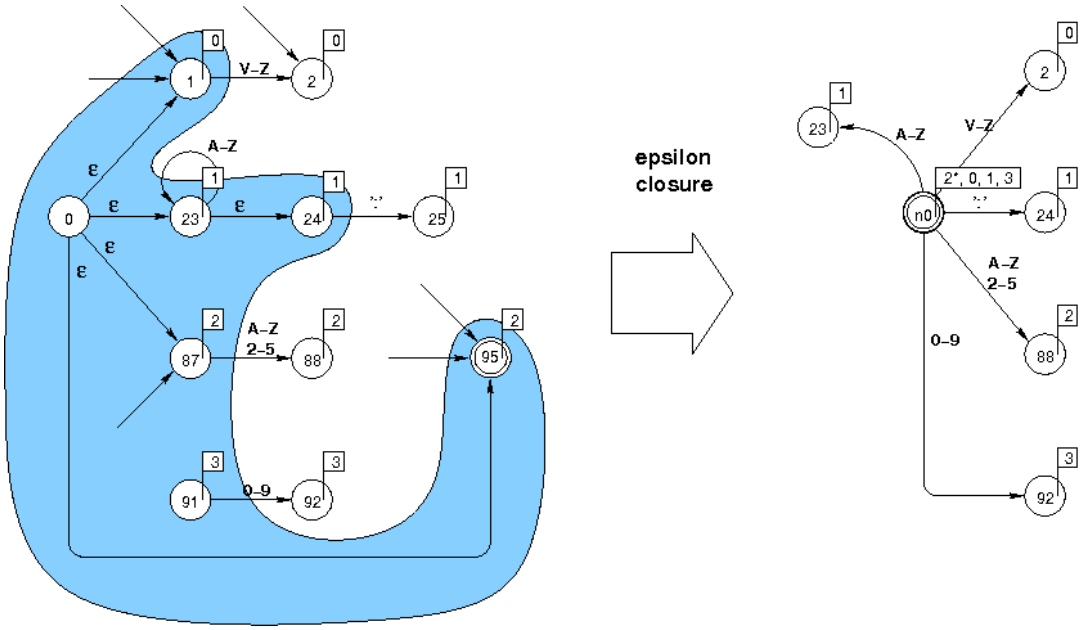

Figure A.9.: The ϵ-closure.

some characters trigger to multiple states at the same time. This means, that the state machine would be in more than one state at the same time, and can, therefore, not be a DFA. On the other hand, one can definitely say that *one particular character* triggers *one state combination*. Combining the states of each character trigger to a state combination, again allows the one-to-one (injective) mapping from a trigger to a state—and the conditions for a DFA are met. The search of a distinct trigger map consist of the following:

> A **distinct trigger map** from a state n is based on target state combinations that are reached by exactly the same trigger. A set of target states can be combined, if there is at least one common trigger (character code point) by which they are reached from state n. When all target state combinations are found, they can be 'renamed' as single states, and the trigger map describes a one-to-one mapping from incoming characters to target states.

Traditionally [], [] the detection of suiteable state combinations considers each single character of the trigger map isolated. This works fine, as long as the code set does not exceed the oldtimer 8-bit ASCII character set. Each state in the NFA requires then 255 checks. For a Unicode-based lexical analyser this behaves significantly different. Unicode 5.0 currently has $10FFFF, i.e. decimal 1.114.111, code points – not to speak about code points for private usage. Consequently about 140.000 times checks would have to be performed for each state in the state machine. In order to avoid such a caste in calculation time, the author of queχ developed a different algorithm based on character ranges as shown in figure A.11. The algorithm identifies state combinations which are related to

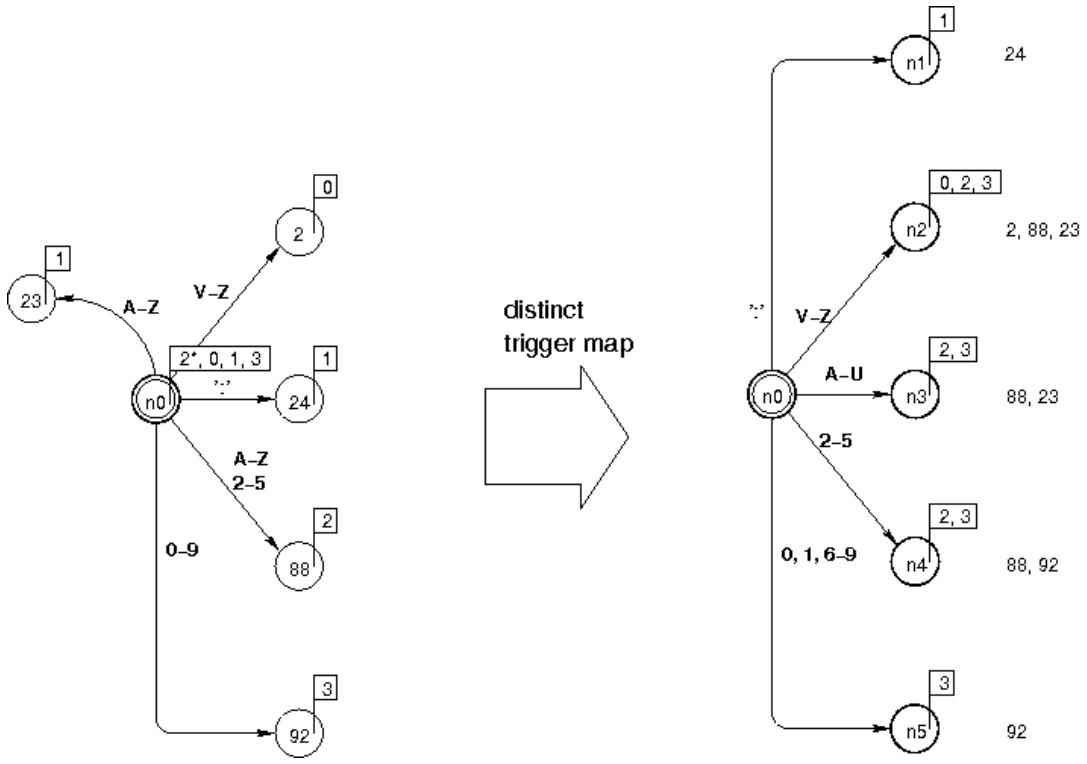

Figure A.10.: Transscription of a state set into a single state of an DFA.

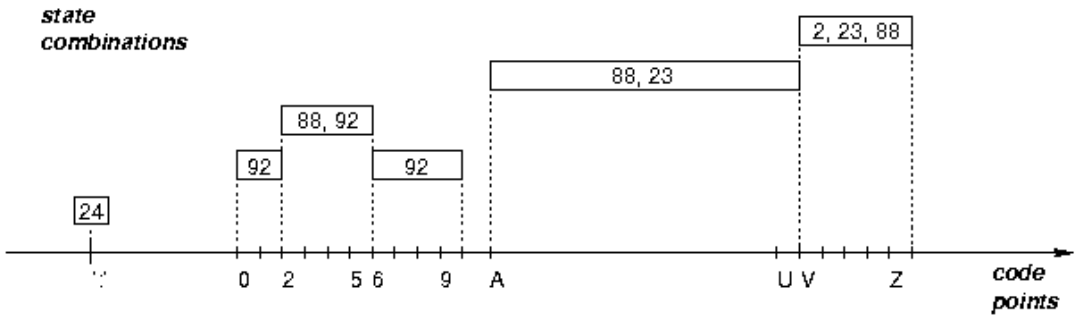

Figure A.11.: Combinations of follow-up states displayed against the code points of a character set – similar to a time history

certain overlapping ranges of characters.

As mentioned in ??, the queχ engine relies on a description of triggers in terms of number ranges. Instead of checking each character, the algorithm considers the numeric ranges $R_{A,B}$ in which a state A triggers to state B. If a range $R_{A,B}$ and $R_{A,B}$ intersect, then B and C are states which are entered simultaneously in the NFA. Thus, they have to be combined in the DFA. For each set of target state combination the intersection of all trigger ranges gives the range on what the combination is triggered. It can now be 'renamed', i.e. a new state can be created that is triggered based on any character that fall into the intersection.

The computation of intersections between all possible target states can also be simplified. Again, the line-up of the ranges on the time line comes handy (figure ??). This time line is implemented by 'begin-trigger-target-state' and 'and-trigger-target-state' commands that appear at certain code points. Finding intersecting ranges then consists only in walking down the timeline, adding and removing target states on any such sommand, and storing the intersecting domains together with the target state combinations. With this approach it is possible to compute most suiteable lexical analyzers in the frame of few seconds on present day computing machines.

Now, that multiple states are combined into one state, the question remains about the acceptance type of the combined state? The answer becomes simple, though, considering that we try to match patterns and that a pattern match relates to an acceptance state:

- If no state in the state combination is an acceptance state, than the state representing the combination *is not an* acceptance state.

- If at least one state in the state combination *is an* acceptance state, than the state representing the combination is an acceptance state. This is so, since a pattern that matched through one path is not dismissed because another path does not match. For example, the regular expression `if|ignore` matches on "if", i.e. the 'first path' as soon as the "f" comes in, even though, the second path expects a "g" and would be non-acceptance. Since it is only required that one pattern matches, the acceptance of the first path is sufficient.

- If two patterns in a combinatino trigger acceptance, this does not pose an exception since there exists preference rules that trigger if two patterns match on the same number of characters (section ??, ??).

The transformation from NFA to DFA is a process that is not only applied during the construction of state machines for isolated patterns, but also on the state machine that combines all patterns to be matched. In this case, the information about the origins have all to be kept. This includes the information about the original state. If a non-acceptance state from pattern X is combined with an acceptance state from pattern Y, we want to be sure, that the lexical analyser does not notify that X matched. A correct way to determine the winner pattern is to consider the origins that were acceptance states, determine the most priviledged pattern that matches (see section ??) and this is the winner.

The following very important assumption has to hold for the process of converting a set of state machines, i.e. a NFA, into a single state machine, i.e. a DFA:

> Let $P_{a,b}$ be the set of paths from a state A to a state B in a state machine (constituting a parallel branch of the NFA). Let $S_{DFA}(A)$ be the set of states that contain A in the resulting DFA, and let $S_{DFA}(B)$ be the set of states that contain B in the resulting DFA. For the DFA to uniquely represent the NFA, one must assume the set of path $P^*(S_{DFA}(A),\ S_{DFA}(B))$ is equal to the original one, i.e.
>
> $$P^*(S_{DFA}(A),\ S_{DFA}(B)) = P_{a,b}$$

A direct consequence of this is that if a state X in a pattern can only be reached over a state Y, then in the resulting state machine this holds equally. The following algorithm demonstrates in pseudo-code the process of constructing a DFA that corresponds to an NFA. It can be applied for single patterns, as well as for the combined state machine that represents the lexical analyser.

```
state-set-worklist = { [ init-state ] }

while worklist not empty:
    new worklist = empty
    for each state-set in worklist:
        ec = epsilon-closure of state-set
        new-worklist = distinct-target-state-combinations of ec
        set triggers for ec

    worklist = new-worklist
```

A.5. Hopcroft Optimization

In some cases, such as in the example shown in figure A.12a, the structure of a state machine can obviously be optimized. Note, that the sole task of the state machine is to determine if the sequence of characters appeared that matches a pattern. In this sense, there is no difference between all acceptance states. As a direct consequence, if there is a set of acceptance states, such as state 1, 2, and 3 in figure A.12a, where all characters trigger to another acceptance state, then those acceptance states can obviously be combined into a single one. This section concretizes this idea and develops the process known as the Hopcroft optimization.

An essential concept of the Hopcroft Optimization is the concept of *equivalent states*. State machines are equivalent if all possible following sequences of triggers lead to the same result. Broken down to states, one requires for any two states A and B to be equivalent that all character streams that cause failure starting A will cause failure starting from B. Further, all character streams that cause acceptance starting from A cause *equivalent acceptance* starting from B. Concretely, this means that for all possible triggers t, one requires that the follow-up state from A on t is in the same state set than from B on t.

a) b)

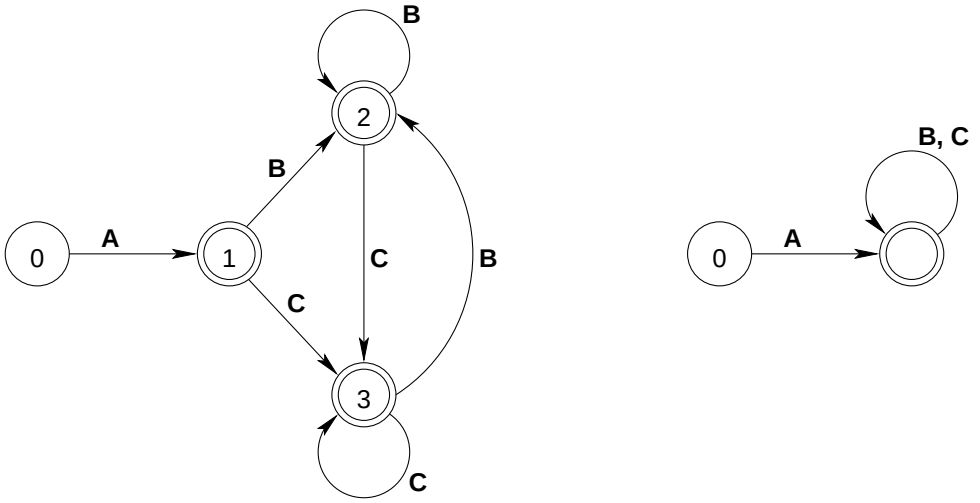

Figure A.12.: Combining equivalent states of a state machine.

In the case of a single pattern, the set of states can be split up into two state sets that build the original worklist: the set of states that are acceptance states and those that are not. These two state sets are the initial worklist. For each state set X in the worklist one has to check wether it behaves 'abnormal', wether it triggers on a character α to a different set of states then the rest. Now, this state set is split into 'normal' states X_0 and 'abornal' states X_1. The state set X from the worklist is then replaced by the two state sets X_0 and X_1. This process can then be repeated until the worklist does not change anymore, i.e. no state sets have to be split up, because the represent equivalent states.

As long as one deals with a single isolated pattern things end up here. For isolated patterns, it only matters if a match occured or not. For the combined state machine, that represents the whole bank of patterns more information is to be considered. Such a case is depicted in figure A.13. Two acceptance state that report different patterns as 'winner' are not equivalent. Thus, the state set of initial equivalent acceptance states needs to be redefined. Note, that the remaining procedure remains the same – one only has to discuss the change of the initial set of equivalent acceptance states for the case that multiple origins are involved. Here, the term *equivalent acceptance* has to be revisited.

> As long as there are no origins involved, then two acceptance states are equivalent, if and only if – well – they are acceptance states.

For the combined state machine, though, it is important to know what pattern, i.e. original state machine, actually reached acceptance, i.e. matched. It is possible that multiple patterns match at the same time. Thus, one has to concentrate on the dominating original pattern (see section ??, page ??).

a) b)

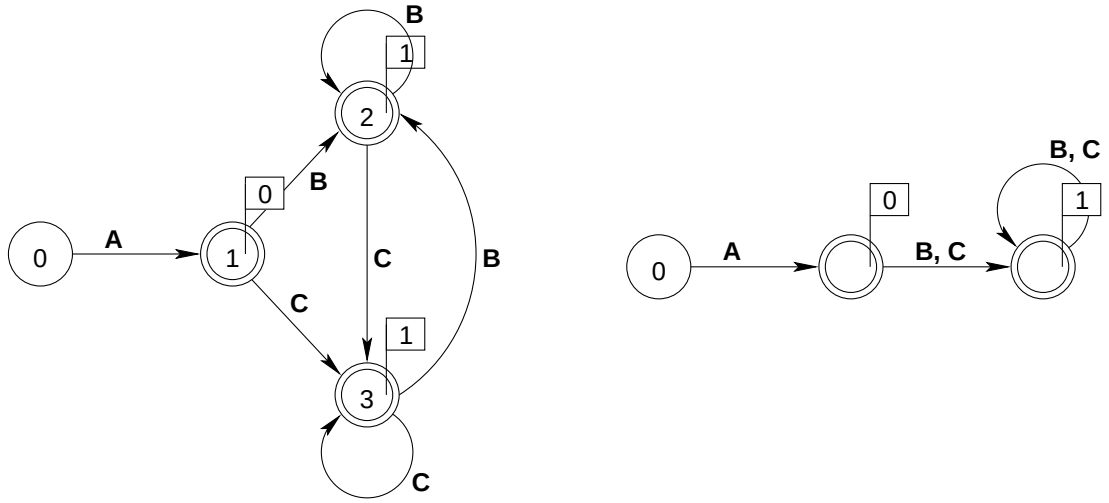

Figure A.13.: Combining equivalent states of a state machine with states of different origin.

> If a state machine contains origins, then two acceptance states are equivalent, if and only if their dominating original patterns are the same.

Post conditions do not change anything at this setup, because the acceptance state of the post-condition is only reached through the state that represents acceptance of the core pattern. Pre-conditions, though, add some restrictions. At this point, it makes sense to imagine the code produced to deal with pre- conditions:

```
    // entry of a state with some pre-conditions:
    if      PreCondition_Pattern5 == True: acceptance = Pattern5
    else if PreCondition_Pattern2 == True: acceptance = Pattern2
    else if PreCondition_Pattern6 == True: acceptance = Pattern6
    else:                                  acceptance = 7
    // lesser priveledged patterns cannot be dealt with:
    //      if PreCondition_Pattern36 == True: acceptance = Pattern36
// does this
    //      if PreCondition_Pattern0 == True:  acceptance = Pattern0
// make sense ?
    ...
```

Let us assume that the pattern priviledges are ordered the same way as they appear in the code fragment, i.e. 5 has the highest priority, if not than comes 2, if not then comes 6, and if the precondition of 6 does not hold then comes the unconditional acceptance of pattern 7. Lesser priviledged patterns (conditioned or unconditioned) do not matter, since the pattern 7 takes it all. Lets call the set of origins from the most priviledged one to the first

unconditional state (if it exists) the *list of possible acceptance origins*. Now, the following can be set about equivalent acceptance states:

> If a state machine contains preconditioned origins, then two acceptance states are equivalent, if and only if their list of possible origins are the same.

The only modification one has to do to the Hopcroft algorithm is to adapt the construction of the initial set of acceptance states. It has to be split according to the above criterion. The splitting procedure considering the triggering to equivalent states remains the same. Here is the algorithm in pseudo-code:

....

Traditionally, the decision wether a set is to be split is made traditionally [] based on single characters. This method works fine for ASCII character streams but becomes uneffective with Unicode (see also page ??). Again, the author developed a method that determines equivalence based on character ranges that cuts down the computation time to a tiny fraction.

A.6. Code Generation

The previous sections discused the construction of an 'annotated' state machine representing the lexical analyser. The final step of code generation remains. Code generation means to write a program that behaves like the state machine. This means, that if a character α appears at the input one has to transit to the same state as the state machine would. Further, if the state machine fails, the program needs to fail. And, if it succeeds, i.e. a pattern matches, then the program need to notify a match. Additionally, some framework needs to be created for the lexical analyser to iterate over an incoming character stream.

One particular problem has not been discussed before. 'Real' lexical analyser search for the longest match, i.e. they try to eat as many characters as possible to achieve a match. It is now conceivable that a pattern A has matched but pattern B is still in the race and has still hope to be matched – it might only need some more characters. Thus the state machine leaves the acceptance state, but *it needs to store information* about the match of A, just in case that pattern B is finally not matched. Consider the two patterns `return` and `[a-z_]+":"`. When a character stream "return" came in, the first pattern matched, but the second is still an option. So, if a string "return_label:" appears one can report that the second pattern matched. But, if the trailing colon is missing, we need to be able to go back and report that there was a return statement. For this reason, we need two variables:

- `last_acceptance`: storing the state index of the last pattern that 'won' by acceptance.
- `last_acceptance_input_position`: storing the position in the the character stream where the last pattern won the match.

Since one incoming character is potentially compared multiple times until it is determined to what follow-up state it triggers, one needs a variable to store the character that just arrived: `input`.

For the case, that post-conditions appear, variables have to be added that store the place where a particular post-condition starts, i.e. the place one has to jump back if the post-condition is successful:

```
last_acceptance_N_input_position
```

where `N` is the numeric identifier of the pre-condition. If pre-conditions are involved one needs to store information wether the pre-conditions are fulfilled or not. Thus variables need to be provided to store the pre-condition results:

```
pre_condition_M_fulfilled_f
```

where `M` is the index of the pre-condition. The lexical analyser core, generated by queχproduces a single function, let it be called `analyse_this()` that is called in order to initiate the lexical analysis:

```
QUEX_ANALYSER_RETURN_TYPE
analyse_this(QUEX_ANALYSER_FUNC_ARGS) {
    // (1) variable definitions ----------------------------------------
    //
    // -- basic required variables
    int                          last_acceptance = -1;
    QUEX_STREAM_POSITION_TYPE    last_acceptance_input_position = (QUEX_STREAM_P
    QUEX_CHARACTER_TYPE          input = (QUEX_CHARACTER_TYPE)(0x00);\n

    // -- variables to deal with post-conditioned patterns (optional)
    //    QUEX_STREAM_POSITION_TYPE  last_acceptance_i_input_position = (QUEX_S
    //    QUEX_STREAM_POSITION_TYPE  last_acceptance_k_input_position = (QUEX_S
    //    QUEX_STREAM_POSITION_TYPE  last_acceptance_l_input_position = (QUEX_S
    // ...

    // -- variables to deal with pre-conditions (optional)
    //    int    pre_condition_i_fulfilled_f = 0;
    //    int    pre_condition_i_fulfilled_f = 0;
    //    int    pre_condition_i_fulfilled_f = 0;
    // ...


    // (2) state transition code ---------------------------------------
    // ...


    // (3) pattern action code / terminal states -----------------------
    //     (possible function return from here)
    // ...
}
```

The way the function is defined leaves opportunities to adapt it to different types of environments, depending on the definitions of the QUEX_...-macros. The following to sections discuss the construction of state transition code and code for the terminal states. Then it has to be discussed how inverse state machines for pre-conditions are translated into code and fit into the picture. A final section explains how to modify the analyser function by the definition of the QUEX_...-macros.

A.6.1. State Transition Header

Each character that is read from a character stream triggers a state transition inside the lexical analyser. This continues until the currently incoming character does not trigger any further transition. Following cases need to be distingushed:

Mismatch The lexical analyser never reached an acceptance state. No explicitly defined pattern has matched. If a default action is defined than it is executed at this point.

Match The lexical analyser reached an acceptance state, some time earlier—or is currently in it. The the action of the dominating pattern has to be executed.

If the current state is an acceptance state, then the id of the winning pattern and the position where it was reached has to be stored. If the current state represents the beginning of a post-condition, then one needs to store the current input position. If later on the post-condition finishes, then one can go back to it.

What pattern actually wins in an acceptance state depends on two issues. First, something that can be compiled into the code: the patterns priviledge. Seconds, something that has to be determined at run-time: dependence on a pre-condition flag. Note, that information about a post-conditioned pattern needs not to be stored, since the path to the acceptance state can only be reached through the state where the core pattern would have raised 'acceptance'. The header of the state transition code of a particular acceptance state looks like the following:

```
QUEX_LABEL__ENTRY_98:
    QUEX_STREAM_TELL(last_acceptance_position);

    // (*) patterns that start now with the post-condition
    QUEX_STREAM_TELL(last_acceptance_a_input_position);
    QUEX_STREAM_TELL(last_acceptance_b_input_position);
    QUEX_STREAM_TELL(last_acceptance_c_input_position);
    ...

    // (*) patterns that dominate, but under a pre-condition
    if( pre_condition_i_fulfilled_f ) {
        last_acceptance = i;
    }
    else if( pre_condition_j_fulfilled_f ) {
        last_acceptance = k;
```

```
    }
    else if( pre_condition_k_fulfilled_f ) {
        last_acceptance = I;
    }
    ...
    else {
        last_acceptance = X;  // the dominating pattern
    }
```

If the state is not an acceptance state, then this header is simply empty. However, both types of states, acceptance and non-acceptance states share the following code fragment:

```
    // (*) get a character from the input stream as a trigger
    QUEX_STREAM_GET(input);

    // ... it follows: the state transitions ...
```

It simply reads the next character from the input stream. It is canned into a macro, so the user may adapt its underlying procedure to his way of representing the character stream (e.g. as a buffer, an object of `istream`, etc.). The code fragment for the state transition has to determine what trigger triggered, and therefore what the subsequent state will be.

A.6.2. State Transition Triggering

The dynamic behavior of a state machine is uniquely determined by the transitions of its states. For a DFA, it can be assumed that it is in one distinct state when an incoming character arrives. The incoming character determines, according to the triggers of the state, what the subsequent state will be. The next to sections discuss the code generation for a state of the lexical analyser state machine. The operations to be coded for a state can be subdivided into writing a state header (section ??) which does some bookkeeping and writing code for the trigger transitions (section ??).

As mentioned earlier, queχ's engine is based on character ranges. This enables a fast algorithm to bracket the incoming character. Similar to binary search[], it determines if the incoming character lies above a border of the middle interval, then higher and lower intervals are checked. Of course, one first checks if the character fits any interval. Imagine an acceptance state for pattern 21 that has the following trigger map:

mismatch	$\longrightarrow$	terminal 21
[A-H], [J-Z]	$\longrightarrow$	state 94
':'	$\longrightarrow$	state 95
I	$\longrightarrow$	state 99

The current implementation produces the following code to implement these transition rules:

```
    if( input >= 58 && input < 91 ) {
        if( input < 65 ) {
            if( input < 59 ) {
                goto QUEX_LABEL__ENTRY_95;
```

```
            } else {
                goto QUEX_LABEL__TERMINAL_21;
            }
        } else {
            if( input < 73 ) {
                goto QUEX_LABEL__ENTRY_94;
            } else {
                if( input < 74 ) {
                    goto QUEX_LABEL__ENTRY_99;
                } else {
                    goto QUEX_LABEL__ENTRY_94;
                }
            }
        }
    }
    // no trigger triggered
    goto QUEX_LABEL__TERMINAL_21;
```

The first if-statement checks wether the `input` lies in the borders of the trigger map. If not, one can directly jump to the terminal state 21. Then one checks wether the input is less than 65, i.e. an A, and then brackets the input down for each interval. This way of generating code seems to leave room for improvement, though. To check wether the input is ':' (character code 58) we test against $>= 58$, < 91, < 65, and < 59, instead of a simple check $== 58$. A hand written transition code may look like the following:

```
    if( input == 58 ) goto QUEX_LABEL_ENTRY_95;
    if( input == 73 ) goto QUEX_LABEL_ENTRZ_99;
    // HERE: input != 73, therefore we can check against A-Z
    if( input >= 65 && input < 91 ) goto QUEX_LABEL_ENTRY_94;
    // no trigger triggered
    goto QUEX_LABEL__TERMINAL_21;
```

However, if one does too many single checks one would loose the advantage of binary bracketting and the result would be much less efficient. The following table shows an estimated cost in terms of comparisons with respect to the two approaches:

	comparisons	
	automatic	hand-written
mismatch (before)	1	3
mismatch (after)	2	4
mismatch (inside)	4	4
[A-H]	5	4
[J-Z]	4	4
':'	4	1
I	5	2

A meaningful comparison of the efficiency of the two approaches, one would have to multiply the results with the probabilities that these characters sets occur. In the above example, the code size of the handwritten transition is much less than the solution created automatically. Determining the optimal algorithm that takes a parameters 'memory-size-speed-tradeoff' and computes the optimal state transition would be an interesting contribution to the queχproject. The author of this text welcomes any propositions to improve the generation of transition code.

B. The Buffer

NOTE: The information here is not complete, since the buffer was completely re-designed. Add to this chapter inconv support and input strategies.

Instead of relying on direct read and write operations from storage devices and transmission lines, it is generally advantegous to profit from the fact that data is usually transported faster when it is transported in some larger blocks. When doing lexical analysis one treats the data stream character by character. To send a 'request' for each character and wait for the reply each time a new character is read, is obviously not a very promissing approach. Independent on what device or line there is in the background the solution that comes handy is: *a buffer in the system memory*.

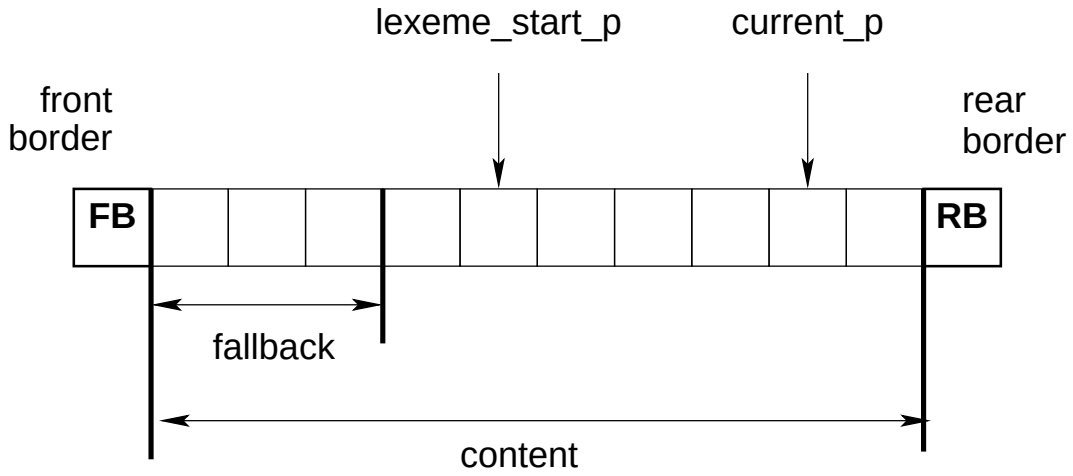

Figure B.1.: Structure of queχ's buffer.

The structure of queχs buffer is shown in figure B.1. The buffer itself is not more than a continous chunk of memory. It has to places to store border characters. One is located at the front to store 'begin of buffer' or 'begin of file'. The other is located at the end to store 'end of buffer' or 'end of file'. However, the 'end of file' character may come anywhere from the start of the buffer to the end. Details about these limit characters are explained below. Thus the actual buffer size is two elements greater than the actual number of content elements. A 'current' pointer iterates from left to right[1] to go forward. The element to which it points designates the current input to the lexical analyser state machine. If it reaches a

[1] In this figure it is 'left-to-right'. This basically means from low memory addresses to higher memory addresses. It had absolutely nothing to to with the directionality of the character encoding.

border new content has to be loaded. The fallback area allows to prevent immediate loads in the opposite directions if one requests to go back immediately after loading.

The following sections explain the mechanisms of queχs buffer, some basic API functions for lexical analysis, the loading procedures, and the buffer creation procedure.

B.1. Programming Interface

The extreme efficiency mentioned in the last section is payed off, though, by some deveilement of the buffer internals to its API. This is not beautiful, from a design prespective, but—well, very efficient. The following operations need to be implemented for the buffer:

- `get_forward()`: increments the pointer to the current character and returns the content. If a limit is reached, then a limiting character code is returned. Again, this has not to be checked before the transition check, because if it is a limit character it drops out anyway.
- `get_backward()`: decrements the pointer to the current character and returns the content. If a limit is reached, then a limiting character code is returned the same way as above.
- `tell_adr()`: returns the position of the 'current' pointer *in memory*.
- `seek_adr(position)`: sets the 'current' pointer to the position given as first argument. The first argument, therefore, designates *a memory address*—not a stream position.
- `seek_offset(const int)`: adds the given offset to the 'current' pointer.

This interface requires some care to be taken[2]. It is not possible to call *get_forward()* blindly and get the whole stream of data. If a limiting character is returned, *it is mandatory* to call `load_forward()` and `get_forward()` again. This spares the comparison against end of buffer and end of file at each 'get'[3]. The function `get_backward()` works in exactly the same manner.

The positioning functions `tell_adr()` and `seek_adr()` are implemented with the same spirit of stingyness. The return and receive memory addresses. Thus, any address taken with 'tell' needs to be adapted at after any call to `load_forward` or `load_backward()`[4]. The `seek_offset()` function currently only adds a value to the 'current' pointer[5]. This function, actually, only used for unit tests. During analysis the absolut addressing functions are applied.

[2] Note, that this API is only 'used' by code that is autogenerated. The human end-user is not confronted with such a 'fragile' interface.

[3] Instead, a 'get' is not more than a pointer increment and a dereferencation

[4] Again, such a requirement on the usage is nearly insane to be demanded from a human user. QueχS code generator, though, has no problem with that.

[5] When the `NDEBUG` flag is not set, it is checked wether the result is inside the boundaries of the buffer

B.2. Mechanisms

The buffer implementation in queχ takes advantage of the mechanism of lexical analysis, and manages to keep the overhead of buffering extremely low. This works as shown in figure B.2. First of all, some content of the information stream in stored in a fixed chunk of memory, the buffer. During the lexical analysis one basically iterates through this chunk. Whenever a new input character is required to determine a state transition, the pointer to the current character, short the 'current' pointer, is increased and the input is assigned the content to what this pointer points. The buffer limits, begin of file, and end of file is determined through special characters. Thus, if the input that is received equals those, then one touched either a buffer limit, the begin of the file, or the end of the file. The trick here is that the transition checks of any state will drop if those characters occur. In any other case the business can continue as usual.

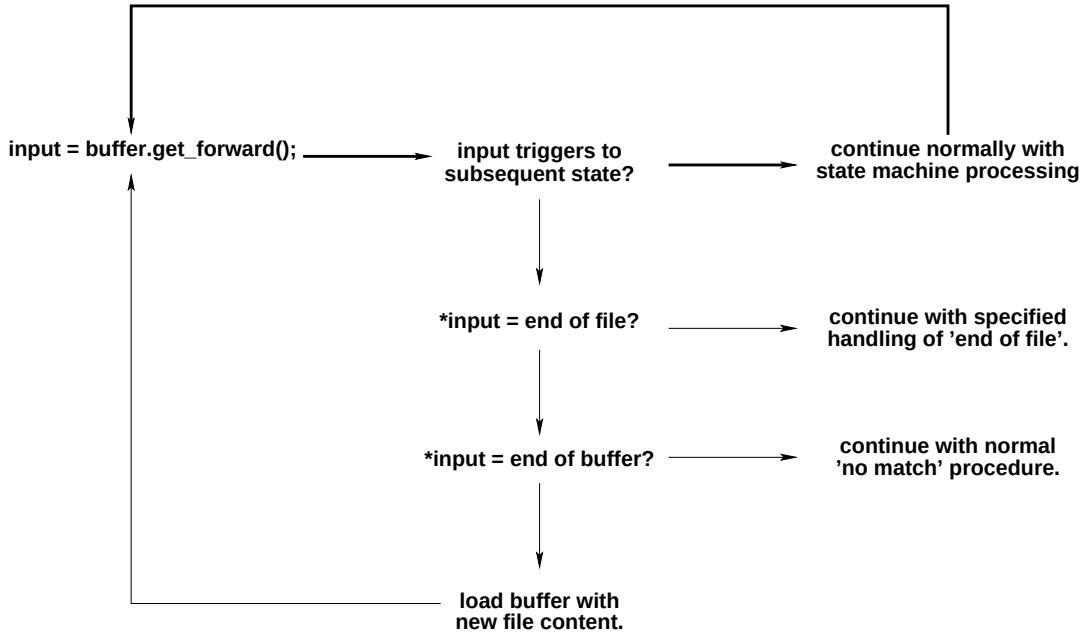

Figure B.2.: Interaction of buffer and the lexical analyser state machine.

The mechanism described above is highly efficient, since there is a *zero overhead* as long as no limit is reached. Only when the buffer limit is reached some overhead occurs for identifying that the drop out occured either due to buffer limit or end of file (begin of file, if one iterates backwards during pre-conditions). Already with a buffer size of 64KB, the buffer limit is only reached every 65536 characters. With an average of 4 operations per character the time overhead should not exceed 0.02% for buffer loading. The speed is basically equivalent to iterating directly through system memory.

B.3. Loading from File

Figure B.3 shows the process of loading the buffer with fresh content in forward direction. Loading data from a device into a buffer is very costy. So, it is a good idea to introduce some fallback area where the end of the current buffer content is stored at the beginning. This prevents an immediate load backwards as soon as a character is required from before the current position. Also, since the current lexeme may be post-treated by some pattern action, the pointer to the lexeme start also has to lie inside the buffer. The maximum of both values defines the fallback border.

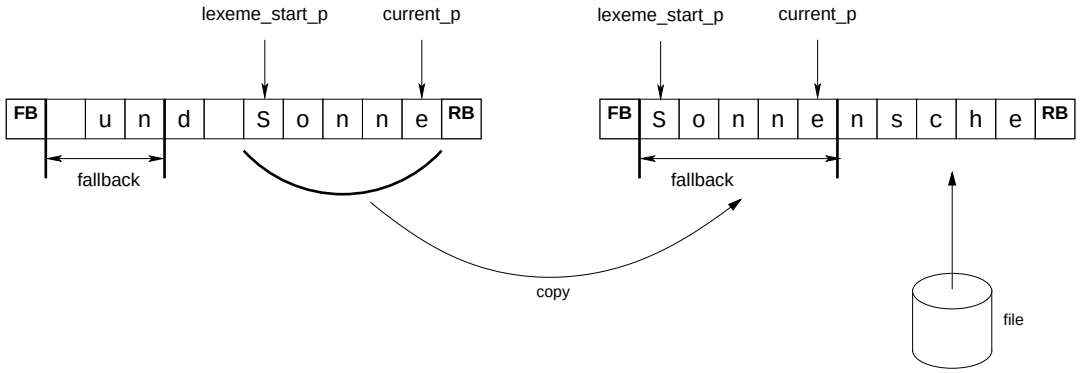

Figure B.3.: Loading new contents 'forwards' from the stream into the buffer.

The new content from the file is then stored behind this border. If an end of file occurs, internally an 'end of file pointer' is set to the position in the buffer where this occurs (most probably before the end of the buffer, but not necessarily). It is stored in a pointer, because it needs to be moved during backward loading. Running through the whole buffer searching for EOF would imply a tremendous slow-down. Also, the end of file character is stored at the position where end of file occured. This way, the `get_forward()` function will return it and the state transition can react on it.

The setting and unsetting of the buffer limit characters and end of file characters at the borders is handled by the load function. Basically, when a border of the buffer does not represent a begin or end of file, it is set to the buffer limit code character. Else, it contains either the begin of file or the end of file character. Note, that the begin of file character, if it occurs, occurs only at the begin of the buffer. The end of file character may occur at any position.

Figure B.4 shows the process of backward loading. Unlike loading forward not such a large amount of memory is newly loaded. Note, that backward loading is only necessary if the fallback buffer was not enough. Thus, it implies some 'abuse of design' that is caught and treated seeminglessly. However, a basic assumption about lexical analysis is that the main direction is `forward`. Instead of loading a large bunch of memory before the current position, loading backwards only loads about the first third of the buffer with backwards content from the file. The two thirds of buffer remain intact. Thus, when lexical analysis finally goes forward again, there is still much room before a forward loading is necessary.

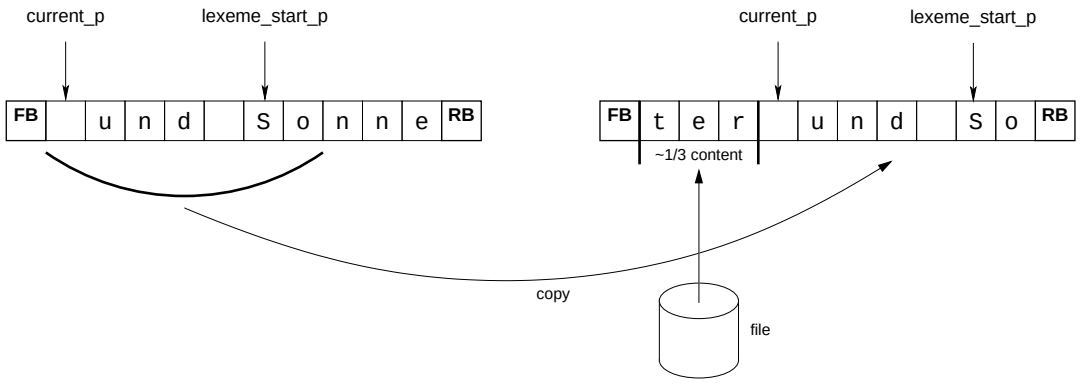

Figure B.4.: Loading new contents 'backwards' from the stream into the buffer.

B.4. Creation of A Buffer Object

Queχs buffer is implemented as a template. By this means a maximum of flexibility is achieved. The following parameters can be passed to the template:

1. `int BOFC`: The character code to designate 'begin of file'.
2. `int EOFC`: The character code to designate 'end of file'.
3. `int BLC`: The character code to designate 'buffer limit'.
4. `class OverflowPolicy`: A class implementing two functions: '`forward`' and '`backward`' to deal with the unlike event of buffer overflow (see section ??).

Those are the parameters to the class `basic_buffer`. The header, though, defines a default setting for the class `buffer`. If these default do not interfere with some abnormal goals (such as using EOF inside, not at the border, of a regular expression), then these values can be adapted. The easiest way to to so is to adapt the '`typedef`' statement in the header file:

```
typedef basic_buffer<0x1, 0x2, 0x0, MyOverflowPolicy> buffer;
```

Then, again `quex::buffer` can be used to name the buffer class at any given place. Constructing a buffer is possible with two basic approaches. The first approach is suitable for systems where dynamic memory allocation is not an issue. It leaves the buffer allocation to the constructor of `buffer`. The second approach can be used for systems where dynamic memory allocation is an issue, such as for embedded systems. In these systems memory is a costy resource and there is no room for slow memory management routines. In this case a pointer to a location in memory can be passed together with the number of bytes that can be taken from it.

B.5. Input Methods