QUEX

FAST UNIVERSAL
LEXICAL ANALYZER GENERATOR

Documentation

Release 0.61.1

Frank-Rene Schaefer

March 02, 2012

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

The quex 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 which is extracted from the text fragment that matched the pattern.

Figure *(this)* 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 compilers for serious programming languages the token stream is received by a parser that interprets 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 advantage that it can be developed faster and it is much easier and safer to provide flexibility and power. This is demonstrated in the following chapters.

The following features distinguish quex 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 by the set of features and philosophies. If you do not use them, then simply skip the concerning sections of 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.
- *Unicode*. The quex engine comes with a sophisticated buffer management which allows to specify converters as buffer fillers. At the time of this writing, the libraries 'iconv' and 'icu' for character code conversion are directly supported.
- 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 convenient transition control.

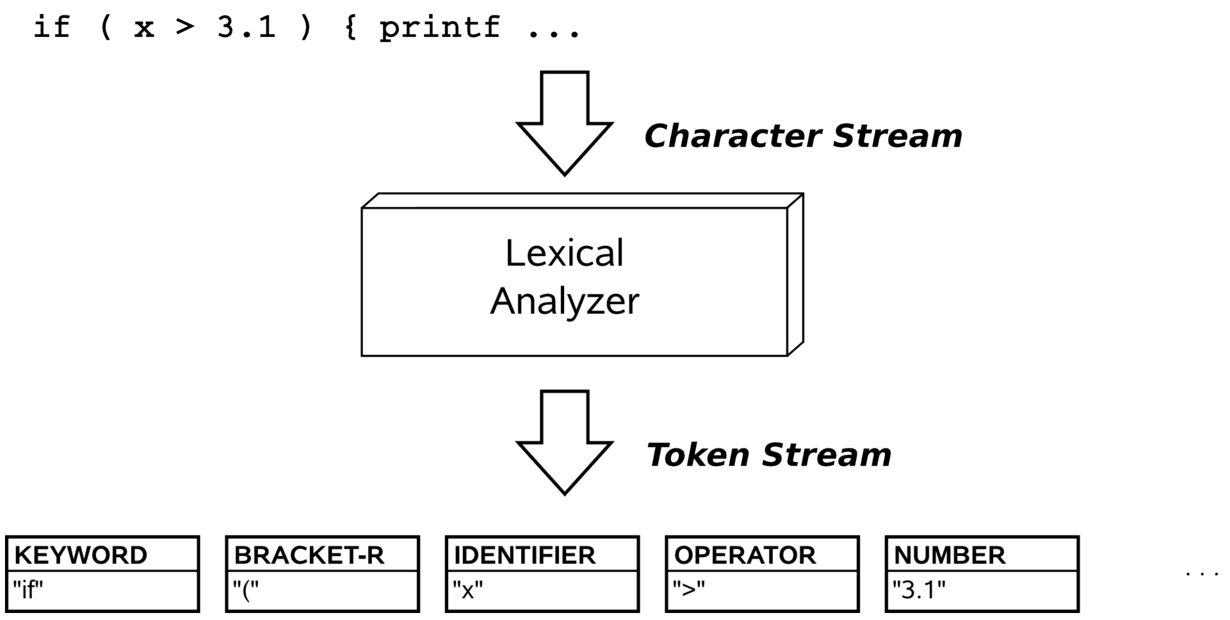


Figure 1.1: Process of lexical analysis.

  - *Transition control*, i.e. restriction can be made to which mode a certain mode can exit or from which mode it can be entered. This prevents the lexical analyser from accidentally dropping into an unwanted lexical analysis 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, Quex 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*. The current line number and column number can be accessed at any time. Quex investigates patterns and determines the most time efficient method to apply changes to column and line numbers. This results in an overhead for counting which is almost not measurable. However, for fine tuning it might as well be turned off.
- *Include stack handling*. For languages where files include other files the quex engine provides a feature that allows to store the analyzer state on a stack, continue analysis on the included file, and restore the analyzer state on return from the included file–without much fuss for the end user.

- *Indentation Events*. As soon as a non-whitespace occurs after a newline a indentation event is fired, that allows convenient means to implement indentation based languages of the Python-like style.
- *Skippers*. For ranges, that are to be skipped, quex can implement optimized small engines for skipping characters that are not of interest for lexical analysis. Examples, as the 'C/C++'-style comments '/' *to* '/' or '//' to newline.
- Automatic generation of transition graphs. Using the *–plot* command line option initiates quex to produce a graphical representation of the underlying state machines.

This text briefly explains the basic concepts of lexical analysis in quex. 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. The final chapter elaborates on the formal usage of all features of quex.

1.1 Installation

On sourceforge there are a variety of Quex packages supporting the major operating systems and distributions at the time of this writing as they are: Microsoft Windows (tm), Mac OS (tm), and Linux (.deb and .rpm based distributions).

This section discusses the major 'pillars' of an installation. It is explains how to install quex from one of the source packages on an arbitrary platform. At the same time, it may provide first hints for troubleshooting. The pillars of a Quex installation are the following:

Python

Before beginning the installation of quex, make sure that Python (http://www.python.org) is installed. Most linux distributions provide handy .rpm or .deb packages.

Warning: Do not install a Python version >= 3.0! Python 3.x is a language different from Python < 3.0. Quex is programmed towards version 2.6.

In order to verify that Python is installed you may open a console (`cmd.exe`, `xterm` or a shel/terminal under Unix). Then type `python` and the output should be similar to:

```
> python
Python 2.6.4 (r264:75706, Jan 30 2010, 22:50:05)
...
>>>
```

Type `quit()` to leave the shell. If you do not get this interactive python shell, then most likely the PATH variable does not contain the path where Python is installed. On the console review the setting of the PATH by:

```
C:\> echo %PATH%
```

on Windows, or:

```
> echo $PATH
```

on Unix. If Python's path is not in the list, then add it. Consult your the documentation of your operating system for further instructions about how to do this.

Quex Distribution

Get a source distribution from `quex.org` or `quex.sourceforge.net`. That is, select a file with the ending '.tgz', '.zip', or '.7z' from the download directory. Extract the files to a directory of your choice.

The QUEX_PATH

Quex knows about where it is installed through the environment variable `QUEX_PATH`. 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/
```

To do the same thing on Windows, go to Control Panel, System, Advanced, Environment Variables. Then add the variable `QUEX_PATH` with the value `C:\Programs\Quex`, provided that quex is installed there.

For clarity, the path must point to the place wher `quex-exe.py` is located.

The Quex Executable

The safest way to do this is to add the content of the `QUEX_PATH` variable to the PATH variable. Thus the systems will search for executables in quex's installation directory. Or, on operating systems that can provide links, make a link:

```
> ln -s $QUEX_PATH/quex-exe.py $EXECUTABLE_PATH/quex
```

where `$EXECUTABLE_PATH` is a path where executables can be found by your system. On a Unix system an appropriate directory is:

```
/usr/local/bin/quex
```

To access this directory, you should be either root or use `sudo`. You can ensure executable rights with:

```
> chmod a+rx $QUEX_PATH/quex-exe.py
> chmod a+rx /usr/local/bin/quex
```

On Windows, the file `quex.bat` should be copied into `C:\WINDOWS\SYSTEM` where most probably executable files can be found.

This is all for the installation of Quex. Your should now be able to type on the command line:

```
> quex --version
```

and get a result similar to:

```
Quex - Fast Universal Lexical Analyzer Generator
Version 0.57.1
(C) 2006-2011 Frank-Rene Schaefer
ABSOLUTELY NO WARRANTY
```

Note, that with the operating system installers there might be problems occuring with previous installations. Thus, when updating better better move any older installation to a place where the system can find them (e.g. in the trash can).

`Compilation`
When compiling Quex-generated code the QUEX_PATH must be provided as an include path, i.e. you must add an `-I` option `-I$QUEX_PATH` on the command line or `-I$(QUEX_PATH)` in a Makefile.

In the sub directories of `$QUEX_PATH/demo` there are many examples of how to do that.

`IConv, or ICU`
If you want to use character set conversion, you need to install one of the supported libraries–currently IBM's ICU <http://icu-project.org/userguide/intro.html> or GNU IConv <http://www.gnu.org/software/libiconv/>[#f1]_.

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, $ldots$. Change to the directory of the sample applications and type `make`. If everything is setup properly, you will get your first quex-made lexical analyser executable in a matter of seconds.

Note: It was reported that a certain development environment called 'VisualStudio' from a company called 'Microsoft' requires the path to python to be set explicitly. Development environments may have their own set of variables that need to be adapted.

The example applications depict easy ways to specify traditional lexical analysers, they show some special features of quex such as mode transitions, and more. Each demo-application deals with a particular feature of quex:

`demo/000`
shows how to setup a lexical analyzer very quickly.

`demo/001`
demonstrates basics on modes and mode transitions.

`demo/002`
contains a sample application for an indentation based language.

`demo/003`
implements a lexical analyzer handling UTF8 encoding. Quex creates an engine that relies

on *converters* ICU and IConv. The converters are used to convert the incoming data stream to unicode. The internal engine still runs on plain unicode code points.

In contrast, directory demo/011 an example is shown how quex creates analyzer engines that do not need converters. The engine itself understands the codec and triggers on its code elements.

`demo/004`
contains a setup to generate a lexical analyser for the 'C' language. Users are encouraged to submit other exciting examples of their language.

`demo/005`
explains how to do lexical analysis with files that include other files.

`demo/006`
contains an example that treats *pseudo ambiguous post contexts* (or, 'dangerous trailing contexts) which are difficult to deal with by means of traditional lexical analyser generators.

`demo/007`
examples showing pattern priorization between base and derived modes as well as applications of `PRIORITY-MARK` and `DELETION`.

`demo/008`
shows how a quex generated lexer is linked to a bison generated parser.

`demo/009`
shows how analyzers can be implemented for `char` and `wchar_t` string streams.

`demo/010`
shows how to access the analyzer's memory buffer directly. It also contains an example, `stdinlexer.cpp`, that shows how to parse the standard input (i.e. the `cin` or `stdin` stream).

`demo/011`
provides an example of how to adapt the character encoding of the lexical analyzer engine without using converters. Example codecs are iso8859-7 and utf8.

`demo/012`
gives an example of how to use multiple lexical analyzers in one single application.

`benchmark`
contains a benchmark suite to measure the performance of the lexical analyzer. As an example a benchmark for a C-lexer is implemented. The suite can build lexical analyzers based on quex, but also as a comparison the same analyzers generated by flex and re2c.

The author of this document suggests that the user looks at these sample applications first before continuing with the remainder of this text. With the background of easy-to-use examples to serve as starting point for their own efforts, it should be natural to get a feeling for the ease of quex.

1.1.1 Visual Studio(tm) Trouble Shooting

At the time of this writing a development tool called 'Visual Studio' by a company called Microsoft(tm) is very popular. It provides a graphical user interface and its users tend to be more graphical-oriented. This section tries to overcome initial troubles related to the usage of Visual Studio.

Quex comes with a variety of demos located in subdirectories of *$QUEX_PATH/demo*. To start, one of the project files must be loaded into Visual Studio. Then, under 'Project' or 'Debug' a choice allows to 'Build the Project'. If this works, it can be run by clicking on a green triangle that points to the right in the top tool bar. The first thing that might fail, is that the build process reports that python cannot be found:

```
1>------ Build started: Project: demo003, Configuration: Debug Win32 ------
1>  'python' is not recognized as an internal or external command,
1>  operable program or batch file.
...
```

To fix this, open your browser (Explorer(tm), Firefox(tm), Opera(tm), or so) and type``www.python.org`` in the address line. Then follow the menus that guide to a download page. Download the latest python version of the 2.* series. Do **not** download anything of version 3.0 or higher. Double click on the downloaded package and follow the instructions. Then open the Microsoft Windows Control Panel. Open the 'Systems' menu. Click on the 'Advanced Tab'; and from there click on 'Environment Variables'. Choose the environment variable 'PATH' and click 'EDIT'. Make sure that the Python's path is included as shown in figure *python_path*. If not, please add the directory where you installed python, e.g. `C:\Python27\;`. The semicolon is the delimiter between directories in the `PATH` variable.

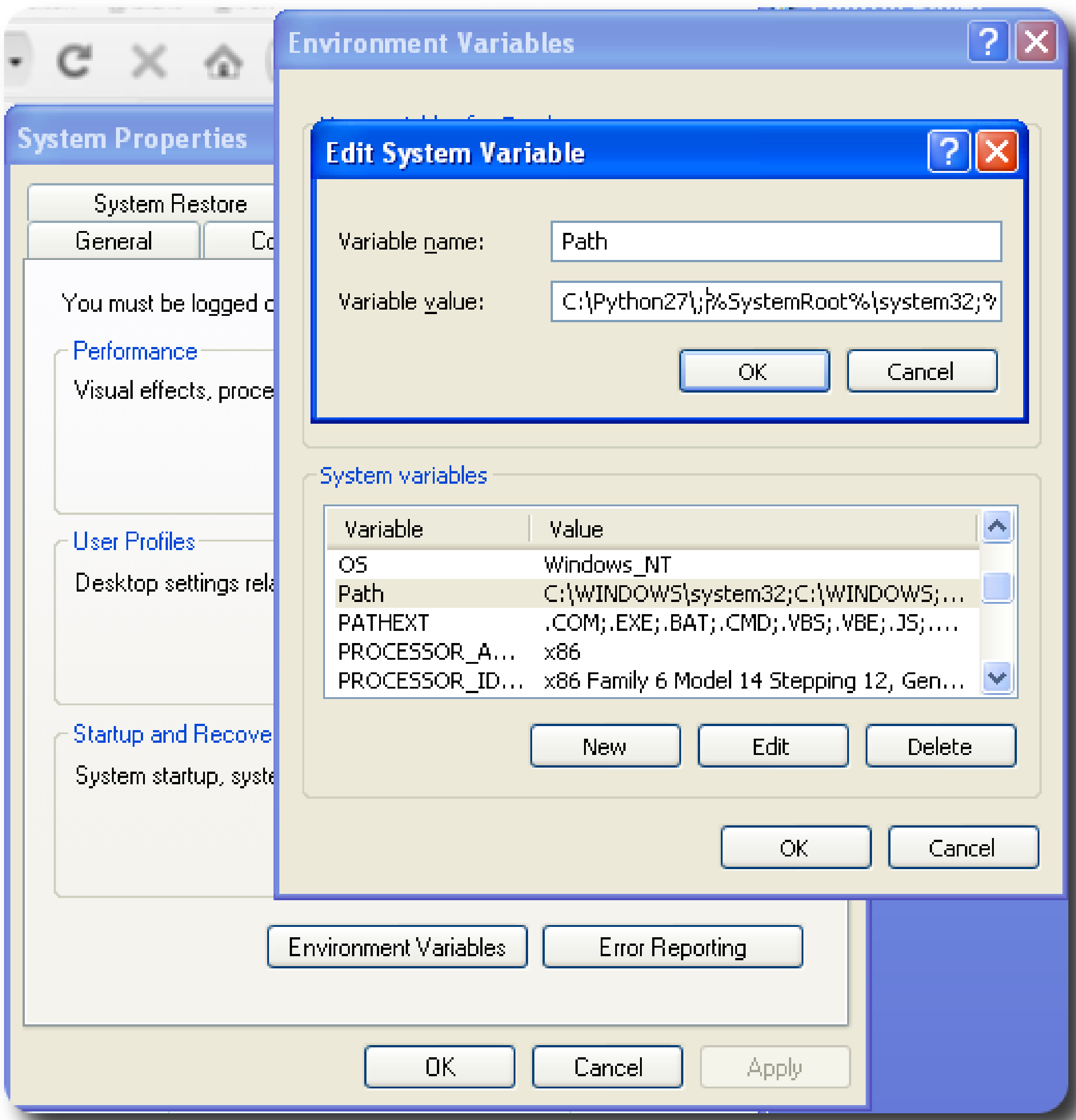
Make sure that a ".qx" file, that contains some quex code, is included in the source file of your project. To involve Quex's code generation in the build process, right click on the ".qx" file and choose 'Properties'. Now, a window opens that shows its properties as shown in *Setting up Quex as code generator.*. The command line tells how Quex is to be called in order to generate the source files. In the example, Quex generates `EasyLexer` and `EasyLexer.cpp`.

If no external components are required the aforementioned explanation should be sufficient. When using an external converter, such as ICU for example, some more work remains to be done. It might happen, that Visual Studio cannot find certain header files, such as `unicode/uconv.h` which is actually an ICU header. The compiler searches for include files in include directories. They search directories can be modified by right-clicking on the project (e.g. 'demo003'). A new window pops up. In the menu for C/C++, click into the field right to 'Additional Include Directories' as show in figure *External include directory.*. Add the additional include directory; In the example it is:

```
"C:\Program Files\icu\include";
```

because the user wishes to use ICU together with Quex and installed ICU at this place.

Configuration parameters of the analyzer can be set in the Menu 'C/C++' at item 'Command Line'. There should be a large text field in the bottom right corner. In the example of demo003, the macro

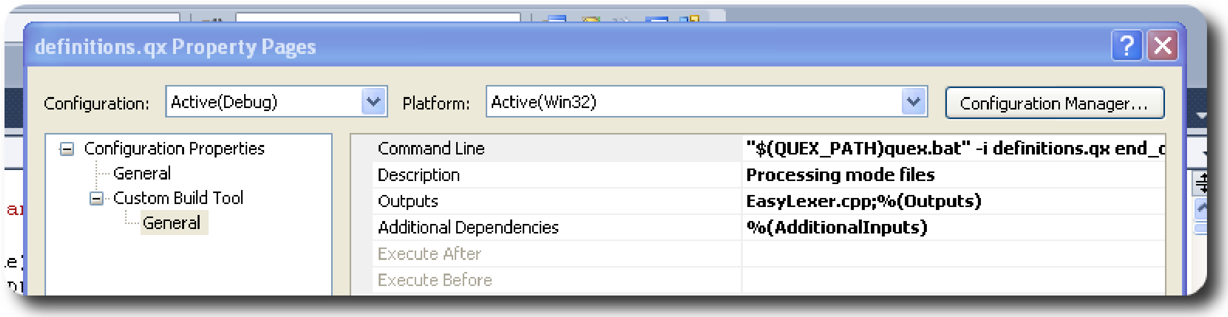


Variable name: Path

Variable value: C:\Python27\;%SystemRoot%\system32;%

Variable	Value
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	x86
PROCESSOR_ID...	x86 Family 6 Model 14 Stepping 12, Gen...

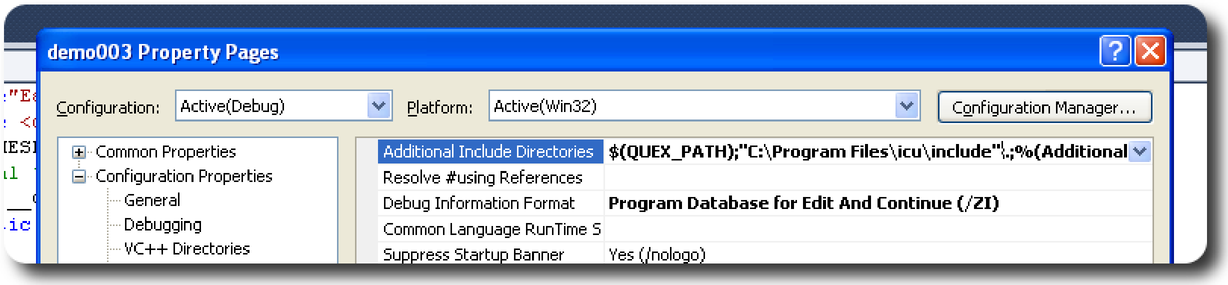
Figure 1.2: Adding python's path to the environment PATH variable.



Configuration: Active(Debug) Platform: Active(Win32)

Command Line	"$(QUEX_PATH)quex.bat" -i definitions.qx end_c
Description	Processing mode files
Outputs	EasyLexer.cpp;%(Outputs)
Additional Dependencies	%(AdditionalInputs)
Execute After	
Execute Before	

Figure 1.3: Setting up Quex as code generator.



Configuration: Active(Debug) Platform: Active(Win32)

Additional Include Directories	$(QUEX_PATH);"C:\Program Files\icu\include";%(Additional
Resolve #using References	
Debug Information Format	Program Database for Edit And Continue (/ZI)
Common Language RunTime S	
Suppress Startup Banner	Yes (/nologo)

Figure 1.4: External include directory.

`ENCODING_NAME` must be defined for the file `example.cpp`. In order to let the lexical analyzer run at full speed, asserts must be disabled, so that at least the following options must added:

```
-DENCODING_NAME=\"UTF8\" -DQUEX_OPTION_ASSERTS_DISABLED
```

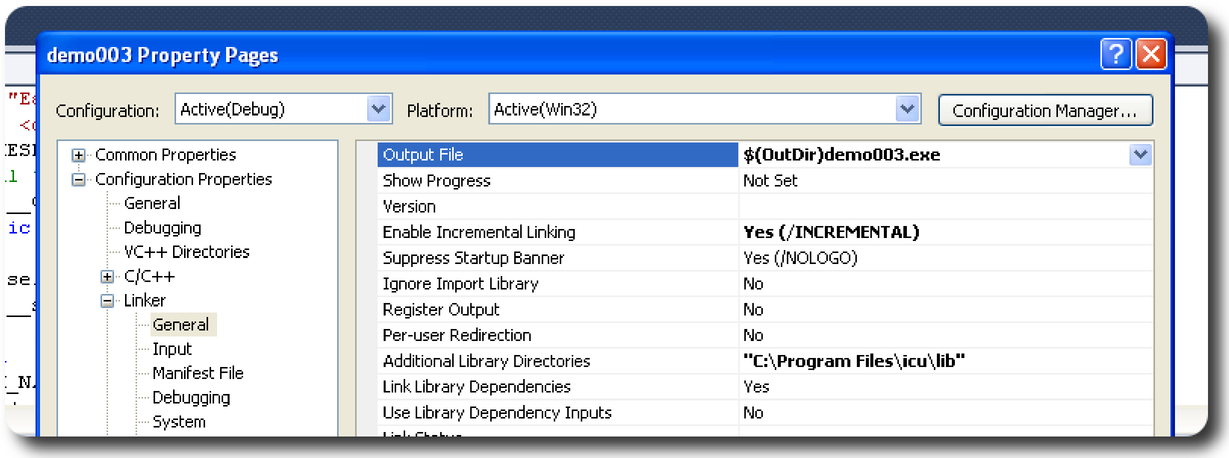
The `-D` flag is used to define the macros. A directly following '=' allows to define a value for the macro.

Analyzers might also rely on other libraries. For this the library itself, as well as the library path must be specified. In the same property window as before, under 'Linker/Input', there is a field called 'Additional Dependencies' that allows to add those libraries. In the example of figure *Adding an external library.* the libraryis:

```
icuuc.lib
```

is added, which is the library entry for ICU's character codec converters. The paths where the Visual Studio searches for additional libraries can be influenced by the entry 'Additional Library Directories' under 'Linker/General' in the same property window. With this setup the example projects should compile and link properly into an executable file.

What remains may be issues with dynamic link libraries (DLLs) that cannot be found at run time. To solve this, the directory where the dynamic link library is located may either be added to the system's path variable, as was done for the python installation path. Or, they might be copied directly into the directory where the compiled program is located (e.g. `demo\003\Debug`). The



Configuration: Active(Debug) Platform: Active(Win32)

Output File	$(OutDir)demo003.exe
Show Progress	Not Set
Version	
Enable Incremental Linking	**Yes (/INCREMENTAL)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**"C:\Program Files\icu\lib"**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No

Figure 1.5: Adding an external library.

later is a good solution for a 'first shot' in case that the development environment shadows the PATH variable.

As a final hint, it might be a good idea to introduce a break point at the last line of the example lexical analyzer. This way to command line window which displays the results remains active. In the command line window, the results of the lexical analysis may be examined.

Plotting

For plotting graphs, the graphviz utility must be installed. It can be downloaded from http://www.graphviz.org. If it still does not work as planned, then make sure that the directory where "dot.exe" is located is inside the global `PATH` environment variable. Closing all applications that may use quex and reopening them ensures that the new setting is communicated correctly to quex.

1.2 Licensing

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

NOTE: There is no license for products targeting military purpose.

Note, that if any part of the LGPL contradicts with the former restrictions, the former restrictions have precedence. The 'no warranty clause' holds in any case. Here is a brief summary of the LGPL:

```
Quex is  free software; you can redistribute  it and/or modify it  under  the
terms  of  the GNU  Lesser  General  Public License  as published by the  Free
Software Foundation; either version  2.1 of the License, or (at your option)
any later version.  This software  is distributed in the  hope that it will  be
```

If this license prevents you from applying quex, please, contact the author fschaef@users.sourceforge.net in order to find an appropriate solution.

1.3 The Name of the Game

The name quex obviously has a lexo-graphical relation to lex and flex–the most well 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 quex and the ability to create directly coded engines will generate much faster lexical analysers. Also, the process of programming shall be much quicker by means of the handy shortcuts and elegant solutions that quex provides.

The last letter '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 quex–and dove much deeper into the subject than he ever thought he would.

Figure 1.6: Calligraphy of the name Quex based on the arabic spelling: 'kaf', 'wau', and 'kha'.

A question closely related to the name might be what the logo actually stands for. It is a calligraphic essay of the author of quex. Based on the arabic spelling of quex (kaf, wau, and kha) the logo consist of a representation in the *squared kufi* style. It is the algorithmic nature of this style that made it possible for a modestly talented artist like me to produce something beautiful. Indeed there is something fascination about arabic calligraphy due to the fact that it combines art and algorithm to produce something esthetically appealing.

1.4 Stability

Quex has grown over the years, gained more and more features and became more and more complex. In the frame of this development many refactorings and redesigns happend in order to main-

[1] As Donald Knuth explains itcite{}, the last letter in TeX is 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 quex should be prononounced like that and not like the 'x' of the German word 'nix'.

tain a transparent clear code structure. The basis for a solid refactoring are automated unit tests and system tests. By means of these tests it is verified that changes to the code or the design do not influence the functioning of the system.

Additionally to the tests for the implemented modules, it is tried to implement for every non-trivial bug and feature request a regression test that ensures that the error does not occur in any future release. Since no release is made without all tests being passed a certain level of quality is guaranteed. Nevertheless, no bug is too small to be reported. It is by bug reports that quex has gained its actual level of maturity.

Note: Please, do not hesitate to report bugs!

The few minutes spend to submit a bug may help to improve the quality for many other users or pinpoint to new interesting issues. In the humble history of quex there have been several cases where small issues let to important design changes. Much of the maturity of quex is due to changes that were made due to bug reports.

Released versions of quex come with a file `unit_test_results.txt` that contains descriptions of all performed tests. It allows the user to overview the level of testing. Any observed lack of coverage may be reported as bug or feature request.

Note: Please, report bugs at

http://sourceforge.net/tracker/?group_id=168259&atid=846112

and submit feature requests at

http://sourceforge.net/tracker/?group_id=168259&atid=846115

Corrections on language or style of the documentation are also welcome. The most convenient way to do this is to download the PDF documentation and use common PDF editor such as Okular http://okular.kde.org, or other freeware tools (see: http://pdfreaders.org/). Comments to the text are best added as *inline note*. The edited source may be submitted to fschaef@users.sourceforge.net.

CHAPTER

TWO

BASIC CONCEPTS

The goal of this section is to familiarize the reader with the basic concepts of lexical analysis and particular implementations in quex. First, it is recalled what lexical analysis is all about. Then, it introduces the concept of pattern-action pairs. It also shows, how those pattern-action pairs can be grouped into modes. Next, the concept of event handlers is explained. One section provides an overview over all code sections of a lexical analyzer description. A final section provides an overview over use case scenarios and how they are to be handled.

2.1 Pattern-Action Pairs

As described earlier, a lexical analyser has to task to analyse the incoming character stream, identify atomic chunks (lexemes) of information and stamp them with a token type. Examples in common programming languages are numbers consisting of a list of digits, identifiers consisting of a list of letters, keywords such as `if`, `else` and so on. This is the simplest form of analysis. However, a lexical analyser might also perform other actions as soon as the input matches a pattern. In general a lexical analyser can be described by a set of patterns related to the actions to be performed when a pattern matches, i.e.

Pattern	Action
list of digits	Return 'Number' token together with the digits.
list of letters	Return 'Identifier' token together with letters.
'for', 'while' etc.	Return keyword token.

Practical applications require a formal language for patterns. Quex requires patterns to be described in regular expressions in the traditional lex/flex style. The patten action pairs in the above list can be defined for quex in the following manner

```
mode MINE {
    [0-9]+          { return TKN_NUMBER(Lexeme); }
    [a-zA-Z_]+      { return TKN_NUMBER(Lexeme); }
    while           { return TKN_KEYWORD_WHILE; }
    for             { return TKN_KEYWORD_FOR; }
}
```

Note, that pattern-action pairs can only occur inside modes. They can only be specified inside a `mode { .. }` definition.

2.2 Modes

In practical languages, there might be ranges in the character stream that do not need to whole set of pattern-action pairs to be active all the time. It might actually be 'disturbing'. Imagine that a compiler would try to analyse 'C' format strings. In this case the detection of numbers, identifiers, etc. has no place in the process. The concept of lexical analyser modes allows to group pattern-action pairs, thus enabling and disabling whole sets of pattern-action pairs. This is critical for 'mini'-languages that are embedded into a main language.

The lexical analyser is exactly in one mode at a specific time. The mode determines what pattern-action pairs are armed to trigger on incoming characters. Additionally, they can provide their own event handlers for different events.

The following code segment shows two modes. The first one shows a normal 'PROGRAM' mode. The second one is a small mode to detect basic string formatting patterns. The '<<' operator is used to trigger the transition from one mode to the other.

```
mode PROGRAM {
    <<EOF>>         { return TKN_TERMINATION; }

    [0-9]+          { return TKN_NUMBER(Lexeme); }
    [a-zA-Z_]+      { return TKN_NUMBER(Lexeme); }
    while           { return TKN_KEYWORD_WHILE; }
    for             { return TKN_KEYWORD_FOR; }
    "\""            { self << MINI; }
    .               { }
}

mode MINI {
    <<EOF>>      { return TKN_TERMINATION; }

    "%s"         { return TKN_FORMAT_CHARP; }
    "%i"         { return TKN_FORMAT_INT; }
    "%f"         { return TKN_FORMAT_FLOAT; }
    "\""         { self << PROGRAM; }
    .            { }
}
```

A potential disadvantage of modes is *confusion*–when used with traditional lexical analyser generators. In flex, for example, the end-user does not see more than a token stream. He has no insight into the current lexical analyser mode. He cannot sense or control the mode transitions that are currently being made. The mode transitions are hidden somewhere in the pattern match actions. GNU Flex's *start conditions* are similar to modes, but the only way two modes, A and B, can be related in flex is via 'inclusion', i.e. by letting a pattern be active in A and B. There is no convienient mechanism to say: 'let B override all patterns of A'. This is where the mode inheritance relationships of Quex provide clear and convenient advantages to be explained in the chapters to come.

2.3 Event Handling

User activity triggered by the lexical analyser is not restricted to 'on pattern match' as it is implemented by pattern-action pairs. Instead, quex allows the definition of event handlers for several events. Those event handlers can contain user written code which is executed as soon as the particular event arrives. Examples for events are 'on_match' which is triggered whenever a pattern matches, 'on_indentation' whenever there is a transition from whitespace to non-whitespace after newline, and so on. Event handlers are tied to modes, thus they are defined inside modes. An event handler of a mode A is only armed as long as the lexical analyser is in that particular mode.

The following example shows how entry and exit of a mode may be used to send tokens that have been accumulated during analysis:

```
mode FORMAT_STRING {
    ...
    on_entry { self.accumulator.clear(); }
    on_exit  {
        self.accumulator.flush(QUEX_TKN_STRING);
        self_send(QUEX_TKN_QUOTE);
    }
    ...
}
```

The 'on_indentation' event handler may be used to create Python-like indentation based languages–as shown in the following example:

```
mode PROGRAM {
    on_indentation {

        if( Indentation > self.indentation_stack.back() ) {
            ...
        }
        while( self.indentation_stack.back() > Indentation ) {
            self_send(QUEX_TKN_BLOCK_CLOSE);
            self.indentation_stack.pop_back();
        }
        ...
    }
    {P_BACKSLASHED_NEWLINE} {
        self.disable_next_indentation_event();
    }
    ...
}
```

The usage details about those example is explained in the sections to come.

2.4 Sections

The definition of a lexical analyzer to be generated by quex happens in several sections. The only section that is mandatory, is the mode section. No pattern-action pairs can be defined outside a mode and no lexical analyzer can work without pattern action pairs. The following list provides an overview about sections that can be defined in a source file to be treated by quex:

mode
A mode section starts with the keyword `mode` and has the following syntax

```
mode mode-name :
     base-mode-1 base-mode-2 ...
     <option-1> <option-2> ...
{
     pattern-1  action-1
     event-1    event-handler-1
     event-2    event-handler-2
     pattern-2  action-2
     pattern-3  action-3
}
```

After the `mode` keyword an identifier must name the mode to be specified. A ':' is followed by a list of whitespace separated list of mode names of base modes (see section <<>>). Then options can be specified as a list of html-like tags, i.e. bracketed in < - > brackets (see section <<>>). An opening curly bracket `{` opens the region for the definition of pattern-action pairs and event handlers (see section <<>>). Finally, a closing `}` terminates the definition of a mode.

define
This section has the syntax

```
define {
    ...
    PATTERN_NAME    pattern-definition
    ...
}
```

The `define` keyword is followed by an opening curly bracked `{`. Then pattern names can be defined as pairs. Each pattern name is an identifier. Note, that the pattern names do not enter any namespace of the generated source code. They are only known inside the mode definitions. The pattern-definition can be any formal description of a lexeme structure using quex's formal language [#f1] (see section <<>>).

token

In this section token identifier can be specified. The definition of token identifiers is optional. The fact that quex warns about undefined token-ids helps to avoid dubios effects of typos, where the analyzer sends token ids that no one catches.

The syntax of this section is

```
token {
     ...
     TOKEN_NAME;
     ...
}
```

The token identifiers need to be separated by semi-colons.

Note: The token identifier in this section are prefix-less. The token prefix, e.g. defined by comand line option `--token-prefix` is automatically pasted in front of the identifier.

```
repeated_token {
     ...
     TOKEN_NAME;
     ...
}

Inside this section the token names are listed that may be sent
via implicit repetition using ``self_send_n(...)``. That is, inside
the token a repetition number is stored and the ``receive()``
function keeps returning the same token identifier until the
repetition number is zero. Only tokens, that appear inside the ``repeated_toke
section may be subject to this mechanism.
```

Additionally to the section defining the behavior of the lexical analyzer there are sections that allow to past code directly into the definition of the engine to be generated. They all follow the pattern:

```
section-name {
     ...
     section content
     ...
}
```

Whatever is contained between the two brackets is pasted in the corresponding location for the given section-name. The available sections are the following:

header
Content of this section is pasted into the header of the generated files. Here, additional include files may be specified or constants may be specified.

body
Extensions to the lexical analyzer class definition. This is useful for adding new class members to the analyzers or declaring `friend`-ship relationships to other classes. For example:

```
body {
        int         my_counter;
        friend void some_function(MyLexer&);
}
```

defines an additional variable `my_counter` and a friend function inside the lexer class' body.

init
Extensions to the lexical analyzer constructor. This is the place to initialize the additional members mentioned in the `body` section. Note, that as in every code fragment, the analyzer itself is referred to via the `self` variable. For example

```
init {
        self.my_counter = 4711;
}
```

Initializes a self declared member of the analyzer `my_counter` to 4711.

A customized token type can easily be defined in a section called

token_type
Defines a customized token class, see *Customized Token Classes*.

Quex supports the inclusion of other files or streams during analyzis. This is done by means of a include stack handler *Include Stack*. It writes the relevant state information into a so called *memento* [1] when it dives into a file and restores its state from it when it comes back. The following sections allow to make additions to the memento scheme of the include handler:

memento
Extensions to the memento class that saves file local data before a sub-file (included file) is handled.

memento_pack
Implicit Variables:

`memento`: Pointer to the memento object.

`self`: Reference to the lexical analyzer object.

`input_handle`: Pointer to Pointer to an input handle, usually a pointer to a pointer to a stream object that is to be included. If `*input_handle == 0x0` this means

[1] See 'Design Patterns' (<<>>).

that the input handle must be assigned based on the `InputName`, i.e. something like:

```
*input_handle = fopen(InputName, "rb");
```

This option, is the reason that `input_handle` is a pointer to a pointer and not just a pointer.

`InputName`: Name of the new data source to be included. This is only to be used if `input_handle == 0x0`.

Code to be treated when the state of a lexical analyzer is stored in a memento.

`memento_unpack`

Implicit Variables:

`memento`: Pointer to the memento object.

`self`: Reference to the lexical analyzer object.

Code to be treated when the state of a lexical analyzer is restored from a memento.

An initial mode `START_MODE` in which the lexical analyzer starts its analysis can be specified via

```
start = START_MODE;
```

2.5 Usage Scenerios

Generated lexical analyzers might be used in different environments and scenarios. Depending on the particular use case, a different way of interaction with the engine is required. In this section, a set of scenarios is identified. The reader is encourage to compare his use case with the ones mentioned below and to apply the appropriate interaction scheme.

1. **File**: The whole stream of data is present at the time where the engine starts its analysis. This is the typical case when the input consists of files provided by some type of file system.

   Interaction: The generator needs to be instantiated with one of the following constructors.

```
CLASS(const std::string&  Filename,
      const char*         InputCodingName      = 0x0,
      bool                ByteOrderReversionF = false);
CLASS(std::istream*       p_input_stream,
      const char*         InputCodingName      = 0x0,
      bool                ByteOrderReversionF = false);
CLASS(std::wistream*      p_input_stream,
      const char*         InputCodingName      = 0x0,
      bool                ByteOrderReversionF = false);
CLASS(std::FILE*          input_fh,
      const char*         InputCodingName      = 0x0,
      bool                ByteOrderReversionF = false);
```

Tokens are received with the functions:

```
QUEX_TYPE_TOKEN*  receive(QUEX_TYPE_TOKEN*  begin,
                          QUEX_TYPE_TOKEN*  end);
```

in case that a 'users_queue' token policy is applied. Or, with

```
void              receive();
void              receive(QUEX_TYPE_TOKEN*   result_p);
```

in case that a 'users_token' token policy is applied. Or, in the default case where a 'queue' token policy is applied the following functions are available:

```
void              receive(QUEX_TYPE_TOKEN*   result_p);
void              receive(QUEX_TYPE_TOKEN**  result_pp);
```

More about token policies can be reviewed in *Token Policy*.

For file inclusion, include stacks can be implemented based on *memento chains* (see *Include Stacks*).

Note: Care might be taken with some implementations of `std:istream`. The number of characters read might not always correspond to the increase of the numeric value of the stream position. Opening files in binary mode, mostly, helps. If not then the engine needs to operate in *strange stream* mode. Then, the compile line option:

```
QUEX_OPTION_STRANGE_ISTREAM_IMPLEMENTATION
```

needs to be defined.

For this scenerio, it is actually not essential that there is a file from which the data is supplied. Likewise, data can be read from a `stringstream` or any other derivate of `istream` or `FILE*`. Essential is that all data is available as soon at the moment the engine expects it.

1. **Syntactically Chunked Input: In this scenario, the engine is fed with** chunks of character streams which appear in a time separate manner. Thus, the engine does not necessarily get a reply when it asks to refill its buffer. In the syntactically chunked case, though, it is assumed that the frame itself is sufficient to categorize the stream into tokens. There are no tails to be appended, so that a lexeme is completed. This may be a frame sequence which is fed to a command line interpreter::

   ```
   frame[time=0]:  [print "hello world"]
   frame[time=1]:  [plot "myfile.dat" u 1:4]
   frame[time=2]:  [display on screen 45, 32 pri]
   ```

   The command line from start to end is passed to the analyzer engine. If there is a keyword `print` and the line ends with "`pri`", then the engine does not have to wait for the next line in order to check wether it completes the `print` keyword. It is imposed that each line is syntactically complete, and thus, `pri` can be considered as an identifier or whatsoever the language proposes (most likely 'error').

This scenario requires direct buffer access. It can be treated by means of *Copying Content*, *Direct Filling*, or *Pointing* as mentioned in the dedicated sections.

1. **Arbitrarily Chunked Input: In this scenario, the analyzer needs to wait** at the point it reaches the end of a character frame because there might be a tail that completes the lexeme:

```
frame[time=0]:  [print "hello world"; plo]
frame[time=1]:  [t "myfile.dat" u 1:4; di]
frame[time=2]:  [splay on screen 45, 32]
```

   This might happen when input comes through a network connection and the frame content is not synchronized with the frame size. This scenario, also, requires direct buffer access. It can be treated by means of *Copying Content*, *Direct Filling*, but not with *Pointing*.

In the view of the author, these use case cover all possible scenarios. However, do not hesitate to write a email if there is a scenario which cannot be handled by one of the abovementioned interaction schemes.

2.6 Number Format

Numbers in quex are specified in a way similar to what is common practice in many programming languages such as C, Python, Java, etc. – with some convenient extensions. Number formats are as follows:

`Example 4711`
Normal decimal integers do not start with '0' and have none of the prefixes below.

`0x, Example 0xC0.FF.EE`
Hexadecimal numbers need to be preceeded by '0x'. The dots inside the hexadecimal numbers are meaningless for the parser, but may facilitate the reading for the human reader.

`0x, Example 0o751`
Octal numbers are preceeded by '0o'.

`0b, Example 0b100.1001`
Binary numbers are preceeded by '0b'. Again, redundant dots may facilitate the human interpretation of the specified number.

`0r, Example 0rXMVIII or 0rvii`
Roman numbers must be preceeded by a '0r' prefix.

CHAPTER

THREE

USAGE

This section introduces the basics for using quex. A minimalist, but complete, example shall display the process of generating a lexical analyser engine with quex. A short section gives an overview over the major sections of a quex definition file. In two separate sections the definition of patterns and their related actions is explained. Pattern-action pairs are grouped in modes. The particularities of mode definitions is explained in a dedicated section. At this point in time the user is familiar with the syntax of the input language of quex. A final section discusses the command line arguments for the invokation of quex on the command, or in Makefiles and project files.

3.1 Minimalist Example

This section shows a minimalist example of a complete lexical analyser. It shows the quex-code and the C++ code which is necessary to create a complete working lexical analyser executable. At this point it is tolerated that the reader might not understand every detail of given code fragments. However, the goal is to provide the reader with quick overview of the related processes. Let us start with a .qx file as input to the lexical analyser generator:

```
header {
#include <cstdlib>  // for: atoi()
}

mode ONE_AND_ONLY
{
    <<EOF>>      => QUEX_TKN_TERMINATION;

    [ \t\r\n]+  { }
    "struct"     => QUEX_TKN_STRUCT;
    "int"        => QUEX_TKN_TYPE_INT;
    "double"     => QUEX_TKN_TYPE_DOUBLE;
    "send"       => QUEX_TKN_SEND;
    "expect"     => QUEX_TKN_EXPECT;
    ";"          => QUEX_TKN_SEMICOLON;
    "{"          => QUEX_TKN_BRACKET_OPEN;
    "}"          => QUEX_TKN_BRACKET_CLOSE;
    [0-9]+       => QUEX_TKN_NUMBER(number=atoi((char*)Lexeme));
    [_a-zA-Z]+   => QUEX_TKN_IDENTIFIER(Lexeme);
}
```

First, a C standard header is included in a *header* section. This code is basically pasted inside the generated code. The included header *cstdlib* declares the function *atoi* which is used in the code fragments below. The keyword *mode* signalizes the definition of a lexical analyser mode. All pattern action pairs need to be related to a mode. In the simple example there is only one mode *ONE_AND_ONLY* that collects all patterns to be matched against. The pattern actions simply 'send' a token as a reaction to a matched pattern. Assume that the content mentioned above is stored in a file called *simple.qx*. Then quex can now be invoked with

```
> quex -i simple.qx -o tiny_lexer
```

This will create some source code to be compiler later on. The following C++ program demonstrates the accesses to lexical analyser engine:

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

#include "tiny_lexer"

int main(int argc, char** argv)
{
    quex::Token*        token_p = 0x0;
    quex::tiny_lexer    qlex("example.txt");

    do {
        qlex.receive(&token_p);        // --token-policy queue
        // token_id = qlex.receive(); // --token-policy single
        std::cout << Token.type_id_name() << endl;

    } while( Token.type_id() != QUEX_TKN_TERMINATION );

    return 0;
}
```

This program creates a lexical analyser which gets its input character stream from a file called *example.txt*. It contains a loop to read tokens from that input stream, prints the token's type and exits as soon as the termination token id is received. Assume that this code is saved in a file called *lexer.cpp*, then the following command would create the executable of the lexical analyser:

```
> $CC  lexer.cpp  tiny_lexer.cpp -I$QUEX_PATH -I. -o lexer
```

The *$CC* needs to be replaced by the compiler that you are using (e.g g++, icpc, sunCC etc.). The two files *tiny_lexer.cpp* and *tiny_lexer-core-engine.cpp* are the files that have been created by quex. They are mostly human readable. Interested users might want to investigate how the analysis work, or derive from it with little adaptions a 'low-end' C version of the analyzer by hand. Anyway, this is all to know about the process of generating a lexical analyzer with quex. The application *lexer* is now ready to rumble. Assume that *example.txt* contains the following content:

```
struct example {
  int    x;
  double y;
};

if ConfigOk {
  send number 4711 hello world;
  expect number 0815 handshake acknowledged;
}
```

Then, a call to *lexer* will produce something like the following output:

```
STRUCT
IDENTIFIER
BRACKET_OPEN
TYPE_INT
IDENTIFIER
SEMICOLON
TYPE_DOUBLE
IDENTIFIER
...
SEMICOLON
EXPECT
IDENTIFIER
NUMBER
IDENTIFIER
IDENTIFIER
SEMICOLON
BRACKET_CLOSE
```

The example is self containing, you can either type it by hand or use the example in the demo/000 directory. If it is required that the produced lexical analyzer is to be distributed in source code then quex can create an independent source package, by

```
> quex -i simple.qx -o tiny_lexer --source-package my-package
```

The source package and the generated lexical analyzer are then located in directory `my-packager`. That for compiling the source package the location of the package has to be passed as an include path with `-I`, i.e.

```
> $CC  lexer.cpp  tiny_lexer.cpp -Imy-package -o lexer
```

The source packager only collects those files which are actually required. Thus the command line instructs quex to create the independent source package should look exactly the same as the 'usual' command line plus the `--source-package` option. Alternatively, the compiler's pre-process could be used to generated a macro-expanded, all-included source file. The GNU Compilers supports this via the '-E' command line option. The command line

```
> cat lexer.cpp >> tiny_lexer.cpp
> g++ tiny_lexer.cpp -I$QUEX_PATH -I. -E -o my-packaged-lexer.cpp
```

The first line appends the user's lexer to the generated engine so that all is together in one single file. The second line uses the compiler to expand all macros and include all include files into one single source file that can now be compiled independently. This, however, might include also some standard library headers which under normal conditions are not required in an independent source package.

3.2 Pattern Definition

Section <<sec-practical-patterns>> already discussed the format of the pattern file. In this section, it is described how to specify patterns by means of regular expressions. The regular expression syntax follows the scheme of what is usually called *traditional UNIX syntax* cite{}, includes elements of *extended POSIX regular expressions* cite{IEEE POSIX Standard 1003.2} and *POSIX bracket expressions* cite{}. This facilitates the migration from and to other lexical analyzer generators and test environments. Additionally, quex provides support for *Unicode Propperties*. A compliance to *Unicode Regular Expressions* cite{Unicode 5.0 Technical Report Standard #18} is currently not targeted, though, because this expressive power is usually not required for compiler generation.

Nevertheless, quex provides features that, for example, flex does not. If it is intended to maintain compatibility of regular expressions with flex, then please refer to the flex manual cite{}, section 'Patterns' and do not use quex-specific constructs. This section discusses pure quex syntax. The explanation is divided into the consideration of context-free expressions and context-dependent expressions.

Quex uses regular expressions to describe patterns and provides its own syntax for filtering and combining character sets. The development of applications running unicode might impose the construction of larger descriptions for patterns. In order to keep mode descriptions clean quex provides a *define* section where patterns can be defined and later on referred to by their identifiers in curly brackets. See the following example:

```
define {
   /* Eating whitespace */
   WHITESPACE    [ \t\n]+
   // An identifier can never start with a number
   IDENTIFIER    [_a-zA-Z][_a-zA-Z0-9]*
}

mode MINE : {
    {WHITESPACE}  { /* do nothing */ }
    {IDENTIFIER}  => TKN_IDENTIFIER(Lexeme);
}
```

Patterns are used to identify atomic chunks of information such as ‘numbers’, ‘variable names’, ‘string constants’, and ‘keywords’. A concrete chain of characters that matches a particular pattern is called a *lexeme*. So ‘0.815’ would be a lexeme that matches a number pattern and ‘print’ might be a lexeme that matches a keyword pattern. The description of patterns by means of a formal language is the subject of the following subsections.

3.2.1 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. Character sets are a form of *alternative* expressions– for one single character. For more sophisticated alternative expressions see the paragraphs below.

[: expression :]

matches a set of characters that result from a character set expression *expression*. Section <<formal/patterns/character-set-expressions>> discusses this feature in detail. In particual `[:alnum:]`, `[:alpha:]` and the like are the character sets as defined as POSIX bracket expressions.

[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). Note, that -

standards for 'range from to' where the character code of the right hand side needs to be greater than the character code of the left hand side.

[^A-Z\n]
a "negated character class", i.e., any character but those in the class. The ^ character indicates *negation* at this point. This expression matches any character *except* an uppercase letter or 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'. Note, than inside strings one can still use the ANSI-C backslashed characters `\n`, `\t`, etc. as well as the Unicode name property `\N`. However, general Unicode property expression `\P` that result in *character sets* are not dealt with inside strings.

\C{ R } or \C(flags){ R }
Applies case folding for the given regular expression or character set 'R'. This basically provides a shorthand for writing regular expressions that need to map upper and lower case patterns, i.e.:

```
\C{select}
```

matches for example:

```
"SELECT", "select", "sElEcT", ...
```

The expression `R` passed to the case folding operation needs to fit the environment in which it was called. If the case folding is applied in a character set expression, then its content must be a character set expression, i.e.:

```
[:\C{[:union([a-z], []):]}:]    // correct
[:\C{[a-z]}:]                     // correct
```

and *not*:

```
[:\C{union([a-z], [])}:]        // wrong
[:\C{a-z}:]                       // wrong
```

The algorithm for case folding follows Unicode Standard Annex #21 "CASE MAPPINGS", Section 1.3. That is for example, the character 'k' is not only folded to 'k' (0x6B) and 'K' (0x4B) but also to '' (0x212A). Additionally, unicode defines case foldings to multi character sequences, such as:

```
    (0390) --> ι(03B9)(0308)(0301)
    (0149) --> (02BC)n(006E)
I    (0049) --> i(0069), (0130), (0131), i(0069)(0307)
    (FB00) --> f(0066)f(0066)
```

```
(FB03) --> f(0066)f(0066)i(0069)
(FB17) --> (0574)(056D)
```

As a speciality of the turkish language, the ‘i’ with and without the dot are not the same. That is, a dotless lowercase ‘i’ is folded to a dotless uppercase ‘I’ and a dotted ‘i’ is mapped to a dotted uppercase ‘I’. This mapping, though, is mutually exclusive with the ‘normal’ case folding and is not active by default. The following flags can be set in order to control the detailed case folding behavior:

s

This flag enables simple case folding *without* the multi-character

m

The *m* flag enables the case folding to multi-character sequences. This flag is not available in character set expressions. In this case the result must be a set of characters and not a set of character sequences.

t

By setting the *t* flag, the turkish case mapping is enabled. Whenever the turkish case folding is an alternative, it is preferred.

The default behavior corresponds to the flags *s* and *m* (`\C{R} \C(sm){R}`) for patterns and *s* (`\C{R} \C(s){R}`) for character sets. Characters that are beyond the scope of the current codec or input character byte width are cut out seeminglessly.

\R{ ... }

Reverse the pattern specified in brackets. If for example, it is specified:

```
"Hello "\R{dlroW} => QUEX_TKN_HELLO_WORD(Lexeme)
```

then the token `HELLO_WORLD` would be sent upon the appearance of ‘Hello World’ in the input stream. This feature is mainly useful for definitions of patterns of right-to-left writing systems such as Arabic, Binti and Hebrew. Chinese, Japanese, as well as ancient Greek, ancient Latin, Egyptian, and Etruscan can be written in both directions.

Note: For some reason, it has caused some confusion in the past, that pattern substitution requires an extra pair of curly brackets, i.e. to reverse what has been defined as `PATTERN` it needs to to be written:

```
\R{{PATTERN}}
```

which reads from inside to outside: expand the pattern definition, then reverse expanded pattern. Inside the curly brackets of `\R{...}` any pattern expression may occur in the well defined manner.

\A{P}

The ‘anti pattern’ of `P`, that is the pattern which matches what `P` cannot not match.

Note: Let L be the set of all lexemes and M the set of lexemes that cause a match of P. Then, the anti pattern A{P} matches all lexemes from the set L subtracted the lexemes of M.

For example the anti pattern of the keyword `if` matches all lexemes that are one letter long and do not start with `i`. Also, matches all lexemes that start with `i` but do not continue with `f`.

`\0`
a NULL character (ASCII/Unicode code point 0). This is to be used with *extreme caution*! The NULL character is also used aa buffer limitting character! See section <<sec-formal-command-line-options>> for specifying a different value for the buffer limit code.

`\U11A0FF`
the character with hexadecimal value 11A0FF. A maximum of *six* hexadecimal digits can be specified. Hexadecimal numbers with less than six digits must either be followed by a non-hex-digit, a delimiter such as `"`, `[`, or `(`, or specified with leading zeroes (i.e. use \U00071F, for hexadecimal 71F). The latter choice is probably the best candidate for an 'established habit'. Hexadecimal digits can contain be uppercase or lowercase letters (from A to F).

`\X7A27`
the character with hexadecimal value 7A27. A maximum of *four* hexadecimal digits can be specified. The delimiting rules are are ananlogous to the rules for *U*.

`\x27`
the character with hexadecimal value 27. A maximum of *two* hexadecimal digits can be specified. The delimiting rules are are ananlogous to the rules for *U*.

`\123`
the character with octal value 123, a maximum of three digits less than 8 can follow the backslash. The delimiting rules are ananlogous to the rules for *U*.

`\a, \b, \f, \n, \r, \t, \r, or \v`
the ANSI-C interpretation of the backslashed character.

`\P{ Unicode Property Expression }`
the set of characters for which the *Unicode Property Expression* holds. Note, that these expressions cannot be used inside quoted strings.

`\N{ UNICODE CHARACTER NAME }`
the code of the character with the given Unicode character name. This is a shortcut for `\P{Name=UNICODE CHARACTER NAME}`. For possible settings of this character see cite{Unicode 5.0}.

`\G{ X }`
the code of the character with the given *General Category* cite{}. This is a shortcut for `\P{General_Category=X}`. Note, that these expressions cannot be used inside quoted strings. For possible settings of the `General_Category` property, see section <<sec-formal-unicode-properties>>.

\E{ Codec Name }
: the subset of unicode characters which is covered by the given codec. Using this is particularly helpful to cut out uncovered characters when a codec engine is used (see *Engine Codec*).

Any character specified as character code, i.e. using *‘, ‘x*, *X*, or *U* are considered to be unicode code points. For applications in english spoken cultures this is identical to the ASCII encoding. For details about unicode code tables consider the standard cite{Unicode50}. Section <<sec-formal-ucs-properties>> is dedicated to an introduction to Unicode properties.

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 quex core engine.

This syntax is more ‘in 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. Much of the syntax is directly based on POSIX extended regular expressions cite{}.

R*
: *zero* or more occurencies of the regular expression `R`.

R+
: *one* or more repetition of the regular expression `R`.

R?
: *zero* or *one* `R`. That means, there maybe an `R` or not.

R{2,5}
: anywhere from two to five repetitions of the regular expressions `R`.

R{2,}
: two or more repetitions of the regular expression `R`.

R{4}
: exactly four repetitions of the regular expression `R`.

(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) * c` override the precedence of multiplication over addition.

`RS`
the regular expression `R` followed by the regular expression `S`. This is usually called a *concatenation* or a *sequence*.

`R|S`
either an `R` or an `S`, i.e. `R` and `S` both match. This is usually called an *alternative*.

`{NAME}`
the expansion of the defined pattern "NAME". Pattern names can be defined in *define* sections (see section <<sec-practical-patterns>>).

3.2.2 Character Set Expressions

Character set expression are a tool to combine, filter and or select character ranges conviniently. The result of a character set expression is a set of characters. Such a set of characters can then be used to express that any of them can occur at a given position of the input stream. The character set expression `[:alpha:]`, for example matches all characters that are letters, i.e. anything from *a* to *z* and *A* to *Z*. It belongs to the POSIX bracket expressions which are explained below. Further, this section explains how sets can be generated from other sets via the operations *union*, *intersection*, *difference*, and *inverse*.

POSIX bracket expressions are basically shortcuts for some more regular expressions that would formally look a bit more clumpsy. Quex provides those expressions bracketed in `[:` and `:]` brackets. They are specified in the table below.

Expression	Meaning	Related Regular Expression
`[:alnum:]`	Alphanumeric characters	`[A-Za-z0-9]`
`[:alpha:]`	Alphabetic characters	`[A-Za-z]`
`[:blank:]`	Space and tab	`[ \t]`
`[:cntrl:]`	Control characters	`[\x00-\x1F\x7F]`
`[:digit:]`	Digits	`[0-9]`
`[:graph:]`	Visible characters	`[\x21-\x7E]`
`[:lower:]`	Lowercase letters	`[a-z]`
`[:print:]`	Visible characters and spaces	`[\x20-\x7E]`
`[:punct:]`	Punctuation characters	``[!"#$%&'()*+,-./:;?@[\\\]_`{|}~]``
`[:space:]`	Whitespace characters	`[ \t\r\n\v\f]`
`[:upper:]`	Uppercase letters	`[A-Z]`
`[:xdigit:]`	Hexadecimal digits	`[A-Fa-f0-9]`

Caution has to be taken if these expressions are used for non-english languages. They are *solely* concerned with the ASCII character set. For more sophisticated property processing it is advisable to use Unicode property expressions as explained in section <<formal/ucs-properties>>. In particular, it is advisable to use `\P{ID_Start}`, `\P{ID_Continue}`, `\P{Hex_Digit}`, `\P{White_Space}`, and `\G{Nd}`.

Note: If it is intended to use codings different from ASCII, e.g. UTF-8 or other Unicode character encodings, then the '–iconv' flag or '–icu' flag must be specified to enable the appropriate converter. See section *Character Encodings*.

In the same way as patterns character sets can be defined in a `define` section and replaced inside the `[: ... :]` brackets–provided that they are character sets and not complete state machines.

The use of Unicode character set potentially implies the handling of many different properties and character sets. For convinience, quex provides *operations on character sets* to combine and filter different character sets and create new adapted ones. The basic operations that quex allows are displayed in the following table:

Syntax	Example
`union(A0, A1, ...)`	`union([a-z], [A-Z]) = [a-zA-Z]`
`intersection(A0, A1, ...)`	`intersection([0-9], [4-5]) = [4-5]`
`difference(A, B0, B1, ...)`	`difference([0-9], [4-5]) = [0-36-9]`
`inverse(A0, A1, ...)`	`inverse([\x40-\5A]) = [\x00-\x3F\x5B-\U12FFFF]`

A `union` expression allows to create the union of all sets mentioned inside the brackets. The `intersection` expression results in the intersection of all sets mentioned. The difference between one set and another can be computed via the `difference` function. Note, that `difference(A, B)` is not equal to `difference(B, A)`. This function takes more than one set to be subtracted. In fact, it subtracts the union of all sets mentioned after the first one. This is for the sake of convenience, so that one has to build the union first and then subtract it. The `inverse` function builds the inverse set. This function also takes more than one set, so one does not have to build the union first.

Note, that the `difference` and `intersection` operation can be used convienently to filter different sets. For example

```
[: difference(\P{Script=Greek}, \G{Nd}, \G{Lowercase_Letter} :]
```

results in the set of greek characters except the digits and except the lowercase letters. To allow only the numbers from the arabic code block `intersection` can be used as follows:

```
[: intersection(\P{Block=Arabic}, \G{Nd}) :]
```

The subsequent section elaborates on the concept of Unicode properties. At this point, it is worth mentioning that quex provides a sophistacted query feature. This allows to determine the result of such set operations and view the result sets. For example, to see the results of the former set operation quex can be called the following way:

```
quex --set-by-expression 'difference(\P{Script=Greek}, \G{Nd}, \G{Lowercase_Letter}
```

In order to take full advantage of those set arithmetics the use should familiarize hiself with Unicode properties and quex's query mode.

3.2.3 Unicode Properties

Unicode is distinguished from other coding standards not only with respect to its level of completeness including all possibly used character sets. Moreover, much effort has been accomplished in categorizing characters, providing terminology for concepts related to letter systems and defining *character properties*. Unicode defines a fixed set of properties that a character can have. The most important property for a lexical analyzer is the 'code point', i.e. the integer value that represents a character. However, there are other interesting properties that simplify the description of regular expressions.

Quex supports Unicode Standard Properties through the `\P{..}` expressions, where the `P` stands for property. For a quex user properties can be devided into two categories:

- Binary Properties, i.e. properties that a character either has or has not. For example, a character is either a whitespace character or it is not. Sets of characters having a binary property `binary_property` can be accessed through `\P{binary_property}`.
- Non-Binary Properties, i.e. properties that require a particular value related to it. For example, each character belongs to a certain script. A character belonging to the greek script has the property 'Script=Greek'. Sets of characters that have a certain property setting can be accessed via `\P{property=value}`.

Note, that the result of a `\P{...}` expression is always a *set of characters*. Therefore, it cannot be used inside a quotes string expression. For convinience, the properties 'Name' and 'General_Category' are provided through the shortcuts `\N{...}` and `\G{...}`. Thus, `\N{MIDDLE DOT}` is a shorthand for `\P{Name=MIDDLE DOT}` and `\G{Uppercase_Letter}` is a shorthand for `\P{General_Category=Uppercase_Letter}`.

Unicode 5.0 provides more than 17000 character names, so please consult the standard literature for settings of the `Name` property footnote:[Alternatively, consider the file `UnicodeData.txt` that comes with the quex application]. Note also, that names as defined in Unicode 1.0 can be accessed through the 'Unicode_1_Name' property. This property also contains names of control functions according to ISO 6429 cite{}.

As for the General_Category property, the appendix provides the list of possible settings <<sec-appendix-property-general-category>>. At this place, more detailed information is specified about provided properties, their meaning, and settings of other likely-to-be-used properties.

When starting to write lexical analyzers using a wider range of unicode characters the reliance on properties becomes almost unavoidable. However, the huge volume of characters requires some sophisticated tool to browse through properties and related character sets. For this purpose quex provides a *query mode*. This mode allows the user to specify some queries on the command line and get the response immediately on the console. This subject is handled in section <<sec-query-intro>>.

As an option to facilitate the specification of property values, quex allows you to use wildcards `*`, `?` and simple character sets such as as `[AEIOUX-Z]` for the characters `A`, `E`, `I`, `O`, `U`, `X`, `Y`, and `Z`. If the first character is an `!` then the inverse character set is considered. This is conform with unix file name matching. set specifications such as `[a-z]` such to facilitate the search. It is advisable, though, to use quex's query functionality first <<sec-query-intro>> in order to get an impression to what value such a wild-card expression expands.

UCS Property Descriptions

This section focuses on the Unicode Character Properties as they are supported by quex. The current version of Quex is based on Unicode 5.0. It uses the databases as provided by the Unicode Consortium and it is likely that Quex integrates the property system of any later standard as soon as it is in a major state. In particular, Unicode Character Properties are used to define _sets of **characters_** to be matched during lexical analysis. This excludes some properties from consideration, namely the properties tagged as *string* property types and the 'quick-check' properties (see <<UCS#15>>). The following properties are explicitly supported by quex. The expressions in brackets are the aliases that can be used as a shorthand for the full name of the property.

`Binary Properties`

ASCII_Hex_Digit(AHex), Alphabetic(Alpha), Bidi_Control(Bidi_C), Bidi_Mirrored(Bidi_M), Composition_Exclusion(CE), Dash(Dash), Default_Ignorable_Code_Point(DI), Deprecated(Dep), Diacritic(Dia), Expands_On_NFC(XO_NFC), Expands_On_NFD(XO_NFD), Expands_On_NFKC(XO_NFKC), Expands_On_NFKD(XO_NFKD), Extender(Ext), Full_Composition_Exclusion(Comp_Ex), Grapheme_Base(Gr_Base), Grapheme_Extend(Gr_Ext), Grapheme_Link(Gr_Link), Hex_Digit(Hex), Hyphen(Hyphen), IDS_Binary_Operator(IDSB), IDS_Trinary_Operator(IDST), ID_Continue(IDC), ID_Start(IDS), Ideographic(Ideo), Join_Control(Join_C), Logical_Order_Exception(LOE), Lowercase(Lower), Math(Math), Noncharacter_Code_Point(NChar), Other_Alphabetic(OAlpha), Other_Default_Ignorable_Code_Point(ODI), Other_Grapheme_Extend(OGr_Ext), Other_ID_Continue(OIDC), Other_ID_Start(OIDS), Other_Lowercase(OLower), Other_Math(OMath), Other_Uppercase(OUpper), Pattern_Syntax(Pat_Syn), Pattern_White_Space(Pat_WS), Quotation_Mark(QMark), Radical(Radical), STerm(STerm), Soft_Dotted(SD), Terminal_Punctuation(Term), Unified_Ideograph(UIdeo), Uppercase(Upper), Variation_Selector(VS), White_Space(WSpace), XID_Continue(XIDC), XID_Start(XIDS),

`Non-Binary Properties`

Age(age), Bidi_Class(bc), Bidi_Mirroring_Glyph(bmg), Block(blk), Canonical_Combining_Class(ccc), Case_Folding(cf), Decomposition_Mapping(dm), Decomposition_Type(dt), East_Asian_Width(ea), FC_NFKC_Closure(FC_NFKC), General_Category(gc), Grapheme_Cluster_Break(GCB), Hangul_Syllable_Type(hst), ISO_Comment(isc), Joining_Group(jg), Joining_Type(jt), Line_Break(lb), Lowercase_Mapping(lc), NFC_Quick_Check(NFC_QC), NFD_Quick_Check(NFD_QC), NFKC_Quick_Check(NFKC_QC), NFKD_Quick_Check(NFKD_QC), Name(na), Numeric_Type(nt), Numeric_Value(nv), Script(sc), Sentence_Break(SB), Simple_Case_Folding(sfc), Simple_Lowercase_Mapping(slc), Simple_Titlecase_Mapping(stc), Simple_Uppercase_Mapping(suc), Special_Case_Condition(scc), Titlecase_Mapping(tc), Unicode_1_Name(na1), Unicode_Radical_Stroke(URS), Uppercase_Mapping(uc), Word_Break(WB),

Binary properties can simply be applied using an expression such as `\P{ID_Start}`, which results in the set of characters that can build the beginning of an identifier in usual programming languages. Non-binary properties require values and are specified in the form `\P{Property=Value}`. The supported values for each non-binary property are the following: Some settings can be provided using shorthand value aliases. Those aliases are specified in brackets.

`Age`
`1.1,2.0,2.1,3.0,3.1,3.2,4.0,4.1,5.0.`

`Bidi_Class`
`Arabic_Letter(AL), Arabic_Number(AN), Boundary_Neutral(BN), Common_Separator(CS), European_Number(EN), European_Separator(ES), European_Terminator(ET), Left_To_Right(L), Left_To_Right_Embedding(LRE), Left_To_Right_Override(LRO), Nonspacing_Mark(NSM), Other_Neutral(ON), Paragraph_Separator(B), Pop_Directional_Format(PDF), Right_To_Left(R), Right_To_Left_Embedding(RLE), Right_To_Left_Override(RLO), Segment_Separator(S),White_Space(WS).`

`Bidi_Mirroring_Glyph`
(not supported)

`Block`
`Aegean_Numbers, Alphabetic_Presentation_Forms, Ancient_Greek_Musical_Notation, Ancient_Greek_Numbers, Arabic, Arabic_Presentation_Forms-A, Arabic_Presentation_Forms-B, Arabic_Supplement, Armenian, Arrows, Balinese, Basic_Latin, Bengali, Block_Elements, Bopomofo, Bopomofo_Extended, Box_Drawing, Braille_Patterns, Buginese, Buhid, Byzantine_Musical_Symbols, CJK_Compatibility, CJK_Compatibility_Forms, CJK_Compatibility_Ideographs, CJK_Compatibility_Ideographs_Supplement,CJK_Radicals_Supplement, CJK_Strokes, CJK_Symbols_and_Punctuation, CJK_Unified_Ideographs, CJK_Unified_Ideographs_Extension_A, CJK_Unified_Ideographs_Extension_B, Cherokee,`

Combining_Diacritical_Marks, Combining_Diacritical_Marks_Supplement, Combining_Diacritical_Marks_for_Symbols, Combining_Half_Marks, Control_Pictures, Coptic, Counting_Rod_Numerals, Cuneiform, Cuneiform_Numbers_and_Punctuation, Currency_Symbols, Cypriot_Syllabary, Cyrillic, Cyrillic_Supplement, Deseret, Devanagari, Dingbats, Enclosed_Alphanumerics, Enclosed_CJK_Letters_and_Months, Ethiopic, Ethiopic_Extended, Ethiopic_Supplement, General_Punctuation, Geometric_Shapes, Georgian, Georgian_Supplement, Glagolitic, Gothic, Greek_Extended, Greek_and_Coptic, Gujarati, Gurmukhi, Halfwidth_and_Fullwidth_Forms, Hangul_Compatibility_Jamo, Hangul_Jamo, Hangul_Syllables, Hanunoo, Hebrew, High_Private_Use_Surrogates, High_Surrogates, Hiragana, IPA_Extensions, Ideographic_Description_Characters, Kanbun, Kangxi_Radicals, Kannada, Katakana, Katakana_Phonetic_Extensions, Kharoshthi, Khmer, Khmer_Symbols, Lao, Latin-1_Supplement, Latin_Extended-A, Latin_Extended-B, Latin_Extended-C, Latin_Extended-D, Latin_Extended_Additional, Letterlike_Symbols, Limbu, Linear_B_Ideograms, Linear_B_Syllabary, Low_Surrogates, Malayalam, Mathematical_Alphanumeric_Symbols, Mathematical_Operators, Miscellaneous_Mathematical_Symbols-A, Miscellaneous_Mathematical_Symbols-B, Miscellaneous_Symbols, Miscellaneous_Symbols_and_Arrows, Miscellaneous_Technical, Modifier_Tone_Letters, Mongolian, Musical_Symbols, Myanmar, NKo, New_Tai_Lue, Number_Forms, Ogham, Old_Italic, Old_Persian, Optical_Character_Recognition, Oriya, Osmanya, Phags-pa, Phoenician, Phonetic_Extensions, Phonetic_Extensions_Supplement, Private_Use_Area, Runic, Shavian, Sinhala, Small_Form_Variants, Spacing_Modifier_Letters, Specials, Superscripts_and_Subscripts, Supplemental_Arrows-A, Supplemental_Arrows-B, Supplemental_Mathematical_Operators, Supplemental_Punctuation, Supplementary_Private_Use_Area-A, Supplementary_Private_Use_Area-B, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tags, Tai_Le, Tai_Xuan_Jing_Symbols, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Unified_Canadian_Aboriginal_Syllabics, Variation_Selectors, Variation_Selectors_Supplement, Vertical_Forms, Yi_Radicals, Yi_Syllables, Yijing_Hexagram_Symbols (n/a).

Canonical_Combining_Class
0, 1, 10, 103, 107, 11, 118, 12, 122, 129, 13, 130, 132, 14, 15, 16, 17, 18, 19, 20, 202, 21, 216, 218, 22, 220, 222, 224, 226, 228, 23, 230, 232, 233, 234, 24, 240, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 7, 8, 84, 9, 91.

Case_Folding
(not supported)

Decomposition_Mapping
(not supported)

Decomposition_Type
Canonical(can), Circle(enc), Compat(com), Final(fin), Font(font), Fraction(fra), Initial(init), Isolated(iso), Medial(med), Narrow(nar), Nobreak(nb), Small(sml), Square(sqr), Sub(sub), Super(sup), Vertical(vert), Wide(wide).

East_Asian_Width
A, F, H, N, Na, W.

FC_NFKC_Closure
(not supported)

General_Category
Close_Punctuation(Pe), Connector_Punctuation(Pc), Control(Cc), Currency_Symbol(Sc), Dash_Punctuation(Pd), Decimal_Number(Nd), Enclosing_Mark(Me), Final_Punctuation(Pf), Format(Cf), Initial_Punctuation(Pi), Letter_Number(Nl), Line_Separator(Zl), Lowercase_Letter(Ll), Math_Symbol(Sm), Modifier_Letter(Lm), Modifier_Symbol(Sk), Nonspacing_Mark(Mn), Open_Punctuation(Ps), Other_Letter(Lo), Other_Number(No), Other_Punctuation(Po), Other_Symbol(So), Paragraph_Separator(Zp), Private_Use(Co), Space_Separator(Zs), Spacing_Mark(Mc), Surrogate(Cs), Titlecase_Letter(Lt), Uppercase_Letter(Lu).

Grapheme_Cluster_Break
CR(CR), Control(CN), Extend(EX), L(L), LF(LF), LV(LV), LVT(LVT), T(T), V(V).

Hangul_Syllable_Type
L, LV, LVT, T, V.

ISO_Comment
*, Abkhasian, Adrar_yaj, Aristeri_keraia, Assamese, Byelorussian, Dasia, Dexia_keraia, Dialytika, Enn, Enotikon, Erotimatiko, Faliscan, German, Greenlandic, Icelandic, Kaeriten, Kanbun_Tateten, Khutsuri, Maatham, Mandarin_Chinese_first_tone, Mandarin_Chinese_fourth_tone, Mandarin_Chinese_light_tone, Mandarin_Chinese_second_tone, Mandarin_Chinese_third_tone, Merpadi, Naal, Oscan, Oxia,_Tonos, Patru, Psili, Rupai, Sami, Serbocroatian, Tuareg_yab, Tuareg_yaw, Ukrainian, Umbrian, Varavu, Varia, Varudam, Vietnamese, Vrachy, a, aa, ae, ai, ang_kang_ye, ang_kang_yun, anusvara, ardhacandra, ash_*, au, b_*, bb_*, bha, break, bs_*, bub_chey, c_*, candrabindu, cha, chang_tyu, che_go, che_ta, che_tsa_chen, chu_chen, colon, d_*, danda, dd_*, dda, ddha, deka_chig, deka_dena, deka_nyi, deka_sum, dena_chig, dena_nyi, dena_sum, dha,

di_ren_⋆, dong_tsu, dorje, dorje_gya_dram, double_danda, drilbu, drul_shey, du_ta, dzu_ta_me_long_chen, dzu_ta_shi_mig_chen, e, escape, g_⋆, gg_⋆, gha, golden_number_17, golden_number_18, golden_number_19, gs_⋆, gug_ta_ye, gug_ta_yun, gup, gya_tram_shey, h_⋆, halfwidth_katakana-hiragana_semi-voiced_sound_mark, halfwidth_katakana-hiragana_voiced_sound_mark, harpoon_yaz, hdpe, hlak_ta, honorific_section, hwair, i, ii, independent, j_⋆, je_su_nga_ro, jha, ji_ta, jj_⋆, k_⋆, ka_sho_yik_go, ka_shog_gi_go_gyen, kha, kur_yik_go, kuruka, kuruka_shi_mik_chen, kyu_pa, l_⋆, lakkhang_yao, lazy_S, lb_⋆, ldpe, lg_⋆, lh_⋆, line-breaking_hyphen, lm_⋆, lp_⋆, ls_⋆, lt_⋆, m_⋆, mai_taikhu, mai_yamok, mar_tse, mathematical_use, n_⋆, nam_chey, nan_de, ng_⋆, nge_zung_gor_ta, nge_zung_nyi_da, nh_⋆, nikkhahit, nj_⋆, nna, norbu, norbu_nyi_khyi, norbu_shi_khyi, norbu_sum_khyi, not_independent, nukta, nyam_yig_gi_go_gyen, nyi_da_na_da, nyi_shey, nyi_tsek_shey, o, oe, or_shuruq, other, p_⋆, paiyan_noi, pause, pema_den, pete, pha, phurba, pp, ps, pug, punctuation_ring, pvc, r_⋆, ren_⋆, ren_di_⋆, ren_ren_⋆, ren_tian_⋆, repha, rinchen_pung_shey, s_⋆, sara_ai_mai_malai, sara_ai_mai_muan, sara_uue, section, sha, shey, ss_⋆, ssa, t_⋆, tamatart, ter_tsek, ter_yik_go_a_thung, ter_yik_go_wum_nam_chey_ma, ter_yik_go_wum_ter_tsek_ma, tha, tian_ren_⋆, trachen_char_ta, tru_chen_ging, tru_me_ging, tsa_tru, tsek, tsek_shey, tsek_tar, tta, ttha, u, uu, virama, visarga, vocalic_l, vocalic_ll, vocalic_r, vocalic_rr, yang_ta, yar_tse, yik_go_dun_ma, yik_go_kab_ma, yik_go_pur_shey_ma, yik_go_tsek_shey_ma.

Joining_Group
Ain, Alaph, Alef, Beh, Beth, Dal, Dalath_Rish, E, Fe, Feh, Final_Semkath, Gaf, Gamal, Hah, Hamza_On_Heh_Goal, He, Heh, Heh_Goal, Heth, Kaf, Kaph, Khaph, Knotted_Heh, Lam, Lamadh, Meem, Mim, Noon, Nun, Pe, Qaf, Qaph, Reh, Reversed_Pe, Sad, Sadhe, Seen, Semkath, Shin, Swash_Kaf, Syriac_Waw, Tah, Taw, Teh_Marbuta, Teth, Waw, Yeh, Yeh_Barree, Yeh_With_Tail, Yudh, Yudh_He, Zain, Zhain(n/a).

Joining_Type
C, D, R, T.

Line_Break
AI, AL, B2, BA, BB, BK, CB, CL, CM, CR, EX, GL, H2(H2), H3(H3), HY, ID, IN, IS, JL(JL), JT(JT), JV(JV), LF, NL, NS, NU, OP, PO, PR, QU, SA, SG, SP, SY, WJ, XX, ZW.

Lowercase_Mapping
(not supported)

NFC_Quick_Check
(not supported)

NFD_Quick_Check
(not supported)

NFKC_Quick_Check
(not supported)

NFKD_Quick_Check
(not supported)

Name
(see Unicode Standard Literature)

Numeric_Type
`Decimal(De)`, `Digit(Di)`, `Numeric(Nu)`.

Numeric_Value
`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`.

Script
`Arabic(Arab)`, `Armenian(Armn)`, `Balinese(Bali)`, `Bengali(Beng)`, `Bopomofo(Bopo)`, `Braille(Brai)`, `Buginese(Bugi)`, `Buhid(Buhd)`, `Canadian_Aboriginal(Cans)`, `Cherokee(Cher)`, `Common(Zyyy)`, `Coptic(Copt)`, `Cuneiform(Xsux)`, `Cypriot(Cprt)`, `Cyrillic(Cyrl)`, `Deseret(Dsrt)`, `Devanagari(Deva)`, `Ethiopic(Ethi)`, `Georgian(Geor)`, `Glagolitic(Glag)`, `Gothic(Goth)`, `Greek(Grek)`, `Gujarati(Gujr)`, `Gurmukhi(Guru)`, `Han(Hani)`, `Hangul(Hang)`, `Hanunoo(Hano)`, `Hebrew(Hebr)`, `Hiragana(Hira)`, `Inherited(Qaai)`, `Kannada(Knda)`, `Katakana(Kana)`, `Kharoshthi(Khar)`, `Khmer(Khmr)`, `Lao(Laoo)`, `Latin(Latn)`, `Limbu(Limb)`, `Linear_B(Linb)`, `Malayalam(Mlym)`, `Mongolian(Mong)`, `Myanmar(Mymr)`, `New_Tai_Lue(Talu)`, `Nko(Nkoo)`, `Ogham(Ogam)`, `Old_Italic(Ital)`, `Old_Persian(Xpeo)`, `Oriya(Orya)`, `Osmanya(Osma)`, `Phags_Pa(Phag)`, `Phoenician(Phnx)`, `Runic(Runr)`, `Shavian(Shaw)`, `Sinhala(Sinh)`, `Syloti_Nagri(Sylo)`, `Syriac(Syrc)`, `Tagalog(Tglg)`, `Tagbanwa(Tagb)`, `Tai_Le(Tale)`, `Tamil(Taml)`, `Telugu(Telu)`, `Thaana(Thaa)`, `Thai(Thai)`, `Tibetan(Tibt)`, `Tifinagh(Tfng)`, `Ugaritic(Ugar)`, `Yi(Yiii)`.

Sentence_Break
`ATerm(AT)`, `Close(CL)`, `Format(FO)`, `Lower(LO)`, `Numeric(NU)`, `OLetter(LE)`, `STerm(ST)`, `Sep(SE)`, `Sp(SP)`, `Upper(UP)`.

Simple_Case_Folding
(not supported)

Simple_Lowercase_Mapping
(not supported)

Simple_Titlecase_Mapping
(not supported)

`Simple_Uppercase_Mapping`
(not supported)

`Special_Case_Condition`
(not supported)

`Titlecase_Mapping`
(not supported)

`Unicode_1_Name`
(see Unicode Standard Literature)

`Unicode_Radical_Stroke`
(not supported)

`Uppercase_Mapping`
(not supported)

`Word_Break`
`ALetter(LE)`, `ExtendNumLet(EX)`, `Format(FO)`, `Katakana(KA)`, `MidLetter(ML)`, `MidNum(MN)`, `Numeric(NU)`.

3.2.4 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.

Another pitfall is related to character codes that the lexical analyser uses to indicate the *buffer-limit*. The values for those codes are chosen to be out of the range for sound regular expressions parsing human written text (0x0 for buffer-limit). If it is intended to parse binary files, and this value is supposed to occur in patterns, then its code need to be changed. Section <<sec-formal-command-line-options>> mentions how to specify the buffer limit code on the command line.

One more pitfall to be mentioned is actually a plain user error. But, since it resulted once in a bug-report [1] it is mentioned at this place. The issue comes into play when patterns span regions, such as HTML-tags.

```
define {
    ...
    P_XML   <\/?[A-Za-z!][^>]*>
    ...
}
```

Now, when parsing some larger files or database a perturbing buffer overflow might occur. The reason for this might be a '<' operator where it is not considered as a tag opener, as in the following text:

```
La funzione di probabilit data da ove "k" e "r" sono interi non
negativi e "p" una probabilit0<p<1) La funzione generatrice dei
momenti A confronto con le due ...
```

This occurence of the < in `(0<p<1)` opens the `P_XML` pattern and lets the analyzer search for the closing '>'. This might never occur. It is anyway inappropriate to consider this as an XML tag. Thus, patterns that span regions must be protected against unintentional region openers. One might use the `^`, i.e. begin of line to restrict the possible matches, e.g.

```
define {
    ...
    P_XML   ^[ \t]*<\/?[A-Za-z!][^>]*>
    ...
}
```

might restrict the set of possible matches reasonably.

3.2.5 Pre- and Post- Conditions

Additionally to the specification of the pattern to be matched quex 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

a regular expression `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

[1] See bug report 2272677. Thanks to Prof. G. Attardi for pinpointing this issue.

on how the particular operating system codes the newline. In this case, there is no harm coming from different conventions of newline.

R$

a regular expression R, but only at the end of a line and _not_ at 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 'rn' (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 (hexadecimal 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.

Note, that if the `$` shall trigger at the end of the file, it might be advantageous to add a newline at the end of the file by default.

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 [2].

Q/R/

matches `R` from the current position, but only if it is preceeded by a `Q`. Practically, this means that quex 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.

There is an important note to make about post contexts. A post context should never contain an empty path. Because, this would mean that if there is nothing specific following it is acceptable. Thus the remaining definition of the post context is redundant. However, an issue comes with End of File, which is not a character in the stream. Then a post context is required of the type:

```
X is either followed by nothing, but never followed by Y.
```

To achieve this, the problem needs to be translated into to pattern matches:

[2] 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.

```
X        => QUEX_TKN_X();
X/\A{Y} => QUEX_TKN_X();
```

That is both matches are associated with the same actions. The first matches the plain pattern X. The second matches an X which is followed by something that **cannot** match Y, i.e. the anti-pattern of Y.

3.2.6 Pitfalls with Pre- and Post-Conditions

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 sing "don't worry, be happy ... 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:

```
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:

```
...
something
...
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 ref{formal/patterns/context-free-pitfalls} for a detailed discussion on precedence pitfalls.

3.3 Actions

Once a pattern has been identified the caller of the lexical analyzer needs to be notified about what the token was and may be what the lexeme was that precisely matched. This type of *action* is the simplest kind of reaction to a pattern match. The next section will focus on tokens can be easily send to the caller of the lexical analyzer. More sophisticated actions may trigger the transition to other modes. The shortcut definition for mode transition triggering is explained in a dedicated section. Finally, actions may require more code to be programmed. For this reason, it is discussed how C/C++ code can be setup to perform actions.

Note, that actions not only occur as a companion of patterns. They may as well be applied for events. Then they take the role of an event handler. The syntax, though, for pattern-actions and actions for event handling is the same.

3.3.1 Token ID Definition

A token identifier is an integer that tells what type of lexeme has been identified in the input stream. Those identifiers can either be referred to a named constants that are prefixed with a so called token prefix, or directly with a number or a character code. The previous section already used the 'stamping' with token identifiers that were names numeric constants. All names of those token identifiers must have the same prefix. The default prefix is *QUEX_TKN_*, but it can be also adapted using the command line option *–token-prefix* followed by the desired prefix. Named constants do not have to be defined explicitly, but they can be defined in a *token* section, such as in the following example:

```
token {
   IDENTIFIER;
   STRUCT;
   TYPE_INT;
   TYPE_DOUBLE;
   SEND;
```

```
    EXPECT;
    SEMICOLON;
    BRACKET_OPEN;
    BRACKET_CLOSE;
    NUMBER;
}
```

Note, that the names of the token identifiers are specified without any prefix. This reduces typing efforts and facilitates the change from one token prefix to another. The explicit definition of token identifiers has an advantage. If a token identifier is mentioned in the *token* section, then quex will not report a warning if it hits on a token identifier that is not defined in the token section. Imagine a typo in the description of pattern-action pairs:

```
...
       "for"  => QUEX_TKN_KEYWORT_FOR;
...
```

In this case, the numeric constant for *QUEX_TKN_KEYWORT_FOR* is still automatically generated. But, the caller of the lexical analyzer might actually expect a *QUEX_TKN_KEYWORD_FOR* with a *D* in *KEYWORD*, not a *T*. Such a token identifier would never be reported by the lexical analyzer, because of the typo. In order to avoid such confusing situations, it is essential for quex to report warnings in case of the usage of token identifiers that have not explicitly been specified.

There is another way to define names of token identifiers. In frameworks with automatic parser generators, it is common that the parser generator provides a file with token identifier definitions. The foreign token id file can be specified by the *–foreign-token-id-file* command line option followed by the name of the file. For example, if bison or yacc create a token identifier file called *vbasic-token-ids.h* and use the token prefix *TOK_*, quex might be called in the following manner

```
> quex (...) --foreign-token-id-file vbasic-token-ids.h --token-prefix TOK_
```

Quex is still able to identify token identifiers which have not been mentioned in the file–in general. It is able to reflect on the content of the file to a certain extend, but it does not understand it in a way as a full fledged C-Compiler does. The worst thing that could happen is that it warns about an undefined token identifier which has actually been defined. At the time of this writing, it has not been reported to the author about any practical application that produces such an unexpected warning with the current version (0.36.3 or later) of quex. The numeric values of the token idenfier names are accessed directly from the foreign token identifier file and quex does not store or process them internally.

Quex also allows to specify specific numbers for the token ids. This facilitates, for example, the definition of token id groups by means of a signal bit, e.g.

```
token {
     TERMINATION   = 0b0000.0000;
     UNINITIALIZED = 0b1000.0000;
     DIV           = 0b0000.0001;
     MULTIPLY      = 0b0001.0001;
     PLUS           = 0b0011.0001;
     MINUS          = 0b0100.0001;
}
```

By ensuring that only operators DIV, MULTIPLY, PLUS and MINUS contain bit zero, the test for an operator token can happen by a simple binary ‘and’

```
if( token_id & 0x1 ) {
    /* 'token_id' is either DIV, MULTIPLY, PLUS, or MINUS */
    ...
}
```

3.3.2 Sending Tokens

The operator to send a token together with its token-id is =>. It has already appeared in some examples in preceeding sections. The meaning of a code fragment like

```
mode MINE {
  ...
      "while"  => QUEX_TKN_KEYWORD_WHILE;
  ...
}
```

is straight forward: When the pattern while matches, then return the token-id QUEX_TKN_KEYWORD_WHILE. Note, that quex takes the fuss of the user’s shoulders to define numerical values for the tokens. Technically, in the generated code the token constructor is called and the token-id is set to the specified value. Now, the token constructor may allow other arguments. Those additional arguments may be specified in brackets, such as

```
mode MINE {
  ...
      // More general, and required when generating 'C' code:
      [0-9]+   => QUEX_TKN_KEYWORD_NUMBER(number=atoi(Lexeme), text=4711);
      [a-z]+   => QUEX_TKN_KEYWORD_WORD(text=Lexeme);
  ...
}
```

The list of arguments contains assigments such as number = atoi(Lexeme). The assigned attributes are the members of the token class which are set. In the above example the lexeme’s numerical interpretation is assigned to the member variable ‘number’ inside the token, and the Lexeme itself is assigned directly to the text member variable. More than one of those assignments may occur.

Figure *Lexeme variables* shows the variables [3] which are available to access the matched lexeme.

[3] They are actually defined as C-preprocessor macros and they are only active arround the generated code segments. If they are not used, no computation time is consumed.

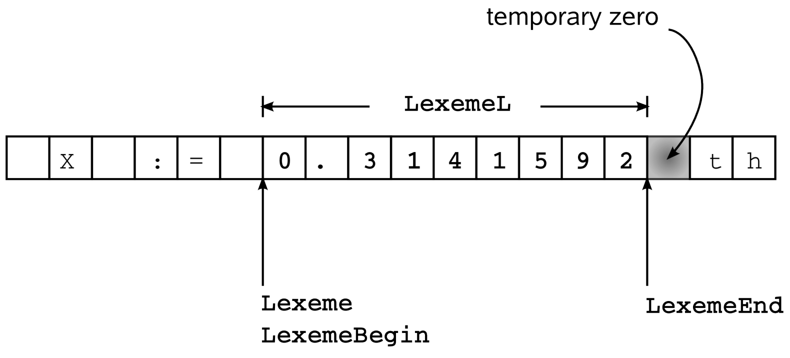


Figure 3.1: Lexeme description variables.

In detail they are:

`Lexeme`
: A pointer to the first character of the matched lexeme. The lexeme itself is temporarily zero-terminated for during the pattern action. This happens *only* if quex detects the identifier `Lexeme` inside the code fragment.

Note: If the lexeme is to be referred to longer outside the action or event handler, then the length or the end pointer has to be stored along with the lexeme. This is not necessary if the default token class is used because it copies the string into a `string` object.

`LexemeBegin`
: This is identical to `Lexeme`, except that no terminating zero is guaranteed.

`LexemeEnd`
: A pointer to the first character after the lexeme which matches the current pattern.

Warning:
If the the identifier `Lexeme` is not specified inside a code fragment, then quex does not necessarily set the terminating zero. Then you either have to set it by hand, i.e.

```
*LexemeEnd = (QUEX_TYPE_CHARACTER)0;
```

`LexemeL`
: The length of the lexeme.

`LexemeNull`
: This is a pseudo-lexeme of length zero. It is useful in cases where it is required to set some string inside a token[#f2]_.

Earlier, it was said that the argument list of brief token senders can only contain named token members. For the sake of simplicity, though, two shorthands are allowed that do not require named attribute assignments:

QUEX_TKN_XYZ (Lexeme)

If there is only one single unnamed parameter it must either be `Lexeme` or `LexemeNull`. No other identifier is allowed. This shorthand triggers a call to the token's 'take_text' function:

```
QUEX_NAME_TOKEN(take_text)(..., LexemeBegin, LexemeEnd);
```

which sets text content inside a token object. If `LexemeNull` is specified it designates the begin and end of the text to be passed the the take_text function. Example:

```
[a-z]+   => QUEX_TKN_IDENTIFIER(Lexeme);          // CORRECT!
```

is admissible, but not

```
"."[a-z]+   => QUEX_TKN_IDENTIFIER(Lexeme + 1);   // WRONG!
```

because the name of the argument is neither `Lexeme` nor `LexemeNull`.

QUEX_TKN_XYZ (Begin, End)

This special call requires `Begin` and `End` to be pointers to `QUEX_TYPE_CHARACTER`. Their name does not play a role. The shorthand triggers a call to

```
QUEX_NAME_TOKEN(take_text)(..., Begin, End);
```

Example:

```
"'"[a-z]+"'"  => QUEX_TKN_QUOTED_IDENTIFIER(LexemeBegin + 1, LexemeEnd - 1
```

Instead of relying on a named constant definition for a token-id, quex can directly use character codes as token-ids. This comes handy when used in conjunction with the parser generators like bison or yacc. The syntax is simply the character written in single quotes. Quex uses UTF-8 as input coding for the source files. Characters with codes beyond ASCII ranges can be specified in the same manner, if your editor is setup in UTF-8 mode. The following shows an example:

```
"="            => '=';
"+"            => '+';
"-"            => '-';
ϵ             => 'ϵ';
∞|infinity     => '∞';
```

As the last line points out, this type of token-id specification is not restricted to patterns of length one–they can be any other pattern. The character code of the token-id can also be specified numerically. Numeric specifications of token ids can be done in decimal (without any prefix), hexadecimal with a '0x' prefix, octal with a '0o' prefix, or binary with a '0b' prefix. This is shown in the following example:

```
Z      => 27;
honey  => 0x1000;            // decimal: 4069
butter => 0o456;             // decimal: 302 hex: 12E
bread  => 0b1000011010100101; // decimal: 34469 hex: 86A5
```

Finally, the token-id can be specified via the name of a character from the unicode character by using 'UC' plus whitespace as a prefix. The unicode character name must have the spaces inside replaced with underscores. An example is shown here:

```
X          => UC LATIN_CAPITAL_LETTER_X;
\U010455   => UC SHAVIAN_LETTER_MEASURE;
\x23       => UC NUMBER_SIGN;
```

Analyzis Continuation

If the token policy `users_token` is applied the analyzer returns after each sending of a token. When the token policy `queue` is used the analyzer continues its analyzis until it hits the safety border in the queue. An exception to this is the reaction to the end of file event, i.e. `on_end_of_stream` or `<<EOF>>`. By default, the analyzer returns. This is to prevent sending tokens after the `TERMINATION` token. Without an action defined for 'end of stream' or 'failure', the analyzer returns a `TERMINATION` token, by default.

Note: When using the token policies `queue` while the asserts are active, the engine **might** throw an exception if the user tries to send a token after a `TERMINATION` token. There is a scenario where it cannot detect it: if a `TERMINATION` is sent, then the queue is cleared, and then new tokens are sent. Then the engine has no reference to the last sent token. At the moment of token sending it cannot tell whether the last token was a `TERMINATION` token or not.

There are no worries when including other files. The include stack handler re-initializes the token queues as soon as the engine returns from an included file.

The behavior is there to help the user, not to bother him. It is to prevent subtle errors where the token queue contains tokens beyond the terminating token that signified the end of a file.

Preparing a Token Object

Sometimes, it might be necessary to perform some more complicated operations on a token object, before it can be sent. In this case, on must refer to the current token pointer. This can be achieved by accessing the current token directly using the 'write token pointer' as in

```
(many)+[letters] {
    self_write_token_p()->number = 4711;
    self_send(QUEX_TKN_SOMETHING);
}
```

This approach is safe to work with token policy queue and single. Actions that are applied on every token, may be accomplished in the `on_match` handler which is executed before the pattern action, e.g. when using a customized token class that stores end values of the column counters, then

```
    on_match {
         self_write_token_p()->my_end_column_n = self.column_number_at_end();
    }
```

does the job of 'stamping' the value in each and every token that is going to be sent.

3.3.3 Mode Transitions

The transition from one mode to another can be controlled via the commands `GOTO`, `GOSUB`, and `GOUP`. Consider an example of usage of `GOTO` for mode transitions:

```
mode NORMAL {
    "<" => GOTO(TAG);
    .   => QUEX_TKN_LETTER(Lexeme);

}
mode TAG {
    ">" => GOTO(NORMAL);
    .   => QUEX_TKN_LETTER(Lexeme);
}
```

There are two modes `NORMAL` and `TAG`. As soon as an opening < arrives the analyzer transits from `NORMAL` mode to `TAG` mode. As soon as the closing > arrives, the analyzer transits back to `NORMAL` mode.

Additionally to the mode transition, tokens can be sent if more arguments are added. The syntax is similar to the brief token senders mentioned in *Sending Tokens*. If it is required to send tokens for the tag regions, the above example may be extended as follows

```
mode NORMAL {
    "<"  => GOTO(TAG, QUEX_TKN_OPEN_TAG(Lexeme));
    "<<" => GOTO(TAG, QUEX_TKN_OPEN_TAG(text=Lexeme, number=2));
    .    => QUEX_TKN_LETTER(Lexeme);

}
```

When transitions to and from the `TAG` mode are triggered, tokens with the token-ids `QUEX_TKN_OPEN_TAG` and `QUEX_TKN_CLOSE_TAG` are also produced. Additional arguments, such as the `Lexeme` can be passed in the same manner as when sending normal tokens.

In case that one enters a mode from different other modes, the use of the mode stack comes handy. Imagine a mode `FORMAT_STRING` for parsing ANSI-C-like format strings. Now, those format strings may occur in a `PROGRAM` mode and in a `MATH` mode. If `GOSUB` is used instead of `GOTO` then the engine remembers the previous mode. The next `GOUP` triggers a transition to this previous mode. Theoretically, the depth of sub-modes is only restricted by the computer's memory. Practically, using a depth of more than two risks to end up in confusing mode transition behavior. In this cases the use of mode transition restrictions <<sec mode-options>> becomes a useful tool in order to detect undesired mode changes.

```
mode PROGRAM {
    ...
    \"          => GOSUB(FORMAT_STRING,  QUEX_TKN_OPEN_STRING);
    ...
}

mode MATH {
    ...
    \"          => GOSUB(FORMAT_STRING, QUEX_TKN_OPEN_STRING);
    ...
}

mode FORMAT_STRING {
    ...
    \"          => GOUP(QUEX_TKN_CLOSE_STRING);
    ...
}
```

As with `GOTO` additional arguments indicate a token sending. Those arguments are applied on the constructor for the resulting token.

Note: When a sub mode is entered the 'return mode' needs to be stored on a stack. This stack has a fixed size. It can be specified via the macro:

```
QUEX_SETTING_MODE_STACK_SIZE
```

A basis for using the sub mode feature is that there is a clear idea about the possible mode transitions. Thus, the number of possible 'gosubs' should be determined at the time of the design of the lexical analyzer. The default setting is eight.

3.3.4 C/C++ Code Segments

Alternatively to the convenient definition of actions in terms of sending tokens and mode transition, more sophisticated behavior can be specified by inserting code fragments directly as pattern-actions or event handlers. The syntax for such definitions is simply to enclose the code in curly brackets as in the following example

```
mode SOMETHING {
    ...
    "\"" {
        self << STRING_READER;
        self_send(QUEX_TKN_EVENT_MODE_CHANGE);
        RETURN;
    }

    ...

    [a-z]+ {
        self_send(QUEX_TKN_IDENTIFIER, Lexeme);
        self.allow_opening_indentation_f = true;
        RETURN;
    }
}
```

The C-code fragments may be ended with `RETURN` and `CONTINUE`. If `RETURN` is specified the analyzer returns from its analyzis. If `CONTINUE` is specified the analyzis may continue without returning to the caller. The reason for these two being defined as macros is that they behave differently depending of token passing policies (*Token Passing Policies*).

The functions used in this example are as follows. The lexical analyzer object can be accessed via

`self`
    self is a reference to the analyzer inside pattern actions and event handlers.

The following send macros are available:

`self_send` (TokenID)
    Sends one token to the caller of the lexical analyzer with the given token identifier `TokenID`.

`self_send_n` (N, TokenID)
    Implicit repetition of tokens can be achieved via the `self_send_n` macro. The token identifiers to be repeated must be defined in the `repeated_token` section, i.e.

```
repeated_token {
    BLOCK_CLOSE;
    BRACKET_OPEN;
    BRACKET_CLOSE;
}
```

    The `self_send_n` macro stores the repetition number inside the token itself. When the current token contains a repeated token identifier `TokenID`, then the call to the `.receive()` function returns the same token `N` times repeatedly before starting a new analysis. As long as the default token classes are used this is implemented automatically. Customized token classes need to prepare some information about how the repetition number is to be stored inside the token class (see :ref:‘ _sec-customized-token-class:‘).

`self_send1` (TokenID, Str)
    This allows to call the token's `take_text` function to set text content to what

is specified by the zero terminated string `Str`. `Str` must be of type 'pointer to `QUEX_TYPE_CHARACTER`'.

Warning: The token passed to 'self_send1(...)' and the like *must* be of the type `QUEX_TYPE_CHARACTER` and *must* have the same coding as the internal buffer. You might want to consider '–bet wchar_t' as buffer size when using converters. Then strings constants like `L"something"` could be conveniently passed.

self_send2 (TokenID, Begin, End)

This corresponds to a call to the current token's `take_text` function where `Begin` and `End` define the boundaries of the string to be taken. Both have to be of type 'pointer to `QUEX_TYPE_CHARACTER`'.

Warning: Relevant for token passing policy *users_token*. With this token policy no tokens can be sent inside event handlers.

The actual mechanism of sending consists of three steps:

1. Filling token content.
2. Setting the current token's identifier.
3. Incrementing setting the current token's pointer to the next token to be filled.

Depending on the particularities of the setup, the send macros adapt automatically. For example, they take care whether the token identifier is stored in a return value, in a token member variable, or in both. If plain send functions are not enough the for filling content into the token, the first step must be implemented by hand, followed by an appropriate send function call. The function `self_token_p()` respectively `self.token_p()` gives access to the current token via pointer. The pointer to the token may be used to prepare it *before* sending it out. The three mentioned steps above may, for example, be implemented like this

```
self.token_p()->set_number(...);
self.token_p()->take_text(LexemeBegin + 1, LexemeEnd -2);
self_send(QUEX_TKN_ID_SPECIAL);
```

When the token policy 'queue' is used, multiple such sequences can be performed without returning to the caller of the lexical analyzer. Modes can be switched with the <<-operator, as shown in the example, or `enter_mode`. For example

```
{P_STRING_DELIMITER} {
    self.enter_mode(STRING_READER);
    RETURN;
}
```

causes a mode transition to the `STRING_READER` mode as soon as a string delimiter arrives. A mode's id can be mapped to a mode object, and vice versa, via the two functions

`QuexMode& map_mode_id_to_mode(const int ModeID);`

`QuexMode&  map_mode_to_mode_id(const int ModeID);`

The current mode of the lexical analyzer can be queried using the functions

`QuexMode&           mode();`

`const std::string&  mode_name() const;`

`const int           mode_id() const;`

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 ModeID);`

`void set_mode_brutally(const QuexMode& 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.

In addition 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(queχ_mode& new_mode);

`void pop_mode();`

`void pop_drop_mode();`

The member function push_mode(new_mode) pushes the current mode on a last-in-first-out stack and sets 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.

Mode Objects

Modes themselves are implemented as objects of classes which are derived from the base class queχ_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 return 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 `assert``s 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 `assert` calls on mode transitions and provides convienient member functions in the lexical analyser class to access information about the current mode.

Warning: Relevant for token passing policies *users_token*, *users_queue*, and *users_mini_queue* when a customized token type is used.
If you use a customized token type that contains pointers, make sure that you read the section about token passing policies *Token Passing Policies*. The point is that the `send()` functions may override these pointers without being referred to elsewhere. It must be ensured that the pointers in received tokens are stored elsewhere, before the analyzer overwrites it.

3.4 Modes

A lexical analyzer mode referes to a behavior of the analyzer engine. More precisely, it defines a set of pattern-action pairs and event handlers that are exclusively active when the analyzer is in a particular mode. The reason for that might be syntactical. Imagine a nested mini-language in a 'mother' language that has interferences of its patterns with the pattern of the 'mother' language. For example, the mother language may contain floating pointer numbers defined as:

```
[0-9]+"."[0-9]*    => FLOAT_NUMBER(Lexeme);
```

In the mini-language there might only be integers and the 'dot' is considered a period of a sentence, such as in:

```
[0-9]+    => NUMBER(Lexeme);
"."       => TKN_PERIOD;
```

If both patterns were describe in a single mode, than interferences would occur. If a number occured at the end of a sentence, such as in:

```
The number of monkeys did not exceed 21. However, there was reason to ...
```

it would be eaten by the floating point number pattern (i.e. interpreted as `21.0`), since the engine follows the longest match. The period at the end of the sentence would not be detected. This is an example were multiple modes are required from a syntax point of view. Another reason for having more than one mode is computational performance. The C-pre-processor statements in the `#`-regions (e.g. `#ifdef`, `#define`, `#include`, etc.) rely on a reduced syntax. Since, not the whole C-language features need to be present in those regions it might make sense to have them parsed in something like a `C_PREPROCESSOR` mode.

The following sections elaborate on the characteristics of modes, event handlers in modes, mode inheritance, the features and options to define modes.

3.4.1 Mode Characteristics

The close relationship between a ‘mode’ and a ‘mood’ has been mentioned in literature <<cite Knuth>>. This section elaborates on how the character of a particular mode can be specified. Such a specification happens with the following constructs:

Options

Options have the form:

```
mode MINE :
    <key-word: argument_0 argument_1 ...> {
      ...
}
```

That is they are bracketed in ‘<’ ‘>’ brackets, start with a key-word for the option and possibly some optional arguments. Options follow the mode definition immediately. Options allow one to restrict mode transitions and inheritance relationships. Also, there are options that allow the implementation of optimized micro-engines to skip ranges.

Pattern-Action Pairs

See <<section ...>>.

Event Handlers

Event handlers allow one to express reactions to events immediately. Examples for events are *mode entrance*, *mode exit*, and *on indentation*. Event handlers appear together with pattern-action pairs in the body of the mode definition. They have the form:

```
mode MINE {
      on_indentation {
          ...
      }
}
```

Note, that the presence of event handlers enforces that patterns that have the same shape as an event handler need to be defined in quotes. A pattern `else` can be defined conveniently ‘as is’, but a pattern `on_indentation` would have to be defined in quotes, since otherwise it would be considered to be an event hander definition.:

```
mode MINE {
    ...
```

```
            else              => QUEX_TKN_KEYWORD_ELSE;
            "on_indentation" => QUEX_TKN_KEYWORD_ON_INDENTATION;
            ...
        }
```

The following sections elaborate on the concepts of options and event handlers. Pattern-action pairs have been discussed in previous sections.

Options

The possible options for modes to be specified are the following.

`<inheritable: arg>`
This option allows to restrict inheritance from this mode. Following values can be specified for `arg`:

- `yes` (which is the default value) explicitly allows to inherit from that mode.
- `no` means that no other mode may inherit from this mode.
- `only` prevents that the lexical analyzer ever enters this mode. Its sole purpose is to be a base mode for other modes. It then acts very much like an *abstract class* in C++ or an *interface* in Java.

`<exit: arg0 arg1 ... argN>`
As soon as this option is set, the mode cannot be left except towards the modes mentioned as arguments `arg0` to `arg1`. If no mode name is specified the mode cannot be left. By default, the allowance of modes mentioned in the list extends to all modes which are derived from them. This behavior can be influenced by the `restrict` option.

`<entry: arg0 arg1 ... argN>`
As soon as this option is set, the mode cannot be entered except from the modes mentioned as arguments `arg0` to `arg1`. If no mode name is specified the mode cannot be entered. The allowance for inherited modes follows the scheme for option `exit`.

`<restrict: arg>`
Restricts the entry/exit allowances to the listed modes. Following settings for `arg` are possible:

- `exit`: No mode derived from one of the modes in the list of an `entry` option is allowed automatically.
- `entry`: Same as `exit` for the modes in the `entry` option.

`<skip: [ character-set ]>`
By means of this option, it is possible to implement optimized skippers for regions of the input stream that are of no interest. Whitespace for example can be skipped by defining a `skip` option like:

```
  mode MINE :
  <skip:  [ \t\n]> {
      ...
  }

Any character set expression as mentioned in <<section>> can be defined
in the skip option. Skipper have the advantage that they are faster than
equivalent implementations with patterns. Further, they reduce the
requirements on the buffer size. Skipped regions can be larger than
the buffer size. Lexemes need be smaller or equal the buffer size.

What happens behind the scenes is the following: The skipper enters the
race as all patterns with a higher priority than any other pattern in the
mode. If it matches more characters than all other patterns, then it wins
the race and it enters the 'eating mode' where it eats everything until the
first character appears that does not fall into the specified skip character
set. Note, in particular that within a given mode

.. code-block:: cpp

   mode X : <skip: [ \t\n] {
       \\\n  => QUEX_TKN_BACKLASHED_NEWLINE;
   }

The token ``QUEX_TKN_BACKLASHED_NEWLINE`` will be sent as soon as the lexeme
matches a backslash and a newline. The newline is not going to be eaten. If
the skipper dominates a pattern definition inside the mode, then quex is
going to complain.
```

<skip_range: start-re end-string>

This option allows to define an optimized skipper for regions that are of no interest and which are determined by delimiters. In order to define a skipper for C/C++ comments one could write:

```
mode MINE :
<skip_range:  "/*" "*/">
<skip_range:  "//" "\n"> {
    ...
}
```

when the `skip_range` option is specified, there is an event handler available that can catch the event of a missing delimiter, i.e. if an end of file occurs while the range is not yet closed. The handler's name is `on_skip_range_open` as described in *_sec-usage-modes-characteristics-event-handlers*. The `start-re` can be an arbitrary regular expression. The `end-string` must be a linear string.

Warning:

For 'real' C++ comments the `skip_range` cannot produce a behavior that is conform to the standard. For this, the lexical analyzer must be able to consider the following as a single comment line

```
// Hello \ this \
   is \
   a comment
```

where the end of comment can be suppressed by a backslashed followed by whitespace. The `skip_range` option's efficiency is based on the delimiter being a linear character sequence. For the above case a regular expression is required.

For more complex cases, such as a standard conform C++ comment skipping must be replaced by a regular expression that triggers an empty action.

```
mode X {
    ...
    "//"([^\n]|(\\[ \t]*\r?\n))*\r?\n      { /* no action */ }
    ...
}
```

In a more general form, the following scheme might be able to skip most conceivable scenarios of range skipping:

```
mode X {
    ...
    {BEGIN}([:inverse({EOE}):]|({SUPPRESSOR}{WHITESPACE}*{END}))*{END}     { /
    ...
}
```

In the C++ case the following definitions are required

```
define {
    BEGIN         //
    END           \r?\n
    EOE           \n
    WHITESPACE    [ \t]
    SUPPRESSOR    \\
}
```

Where `EOE` stands for 'end of end', i.e. the last character of the `END` pattern.

`<skip_nested_range: start-string end-string>`

With this option nested ranges can be skipped. Many programming languages do not allow nested ranges. As a consequence it can become very inconvenient for the programmer to comment out larger regions of code. For example, the C-statements

```
/* Compare something_else */
if( something > something_else ) {
    /* Open new listener thread for reception */
    open_thread(my_listener, new_port_n);
} else {
    /* Close all listening threads. */
    while( 1 + 1 == 2 ) { /* Forever */
        const int next_listener_id = get_open_listener();
        if( next_listener_id == 0 ) break;
        com_send(next_listener_id, PLEASE_RETURN); /* Ask thread to exit. */
    }
}
```

Could only be commented out by `/* */` comments if all closing `*/` are replaced by something else, e.g. `*_/`. Thus

```
/*
/* Compare something_else *_/
if( something > something_else ) {
    /* Open new listener thread for reception *_/
    open_thread(my_listener, new_port_n);
} else {
    /* Close all listening threads. *_/
    while( 1 + 1 == 2 ) { /* Forever *_/
        const int next_listener_id = get_open_listener();
        if( next_listener_id == 0 ) break;
        com_send(next_listener_id, PLEASE_RETURN); /* Ask thread to exit. *_/
    }
}
*/
```

and the compiler might still print a warning for each `/*` that opens inside the outer comment. When the code fragment is uncommented, all `*_/` markers must be replaced again with `*/`.

All this fuss is not necessary, if the programming language supports nested comments. Quex supports this with nested range skippers. When a nested range skip option such as:

```
mode MINE :
<skip_nested_range:  "/*" "*/"> {
   ...
}
```

is specified, then the generated engine itself takes care of the 'commenting depth'. No comment range specifiers need to be replaced in order to include commented regions in greater outer commented regions.

Warning: Nested range skipping is a very nice feature for a programming language. However, when a lexical analyzer for an already existing language is to be developped, e.g. 'C' or 'C++', make sure that this feature is not used. Otherwise, the analyzer may produce undesired results.

Event Handlers

This section elaborates on the event handlers which can be provided for a mode. Event handlers are specified like:

```
event_handler_name {
    /* event handler code */
}
```

Some event handlers provide implicit arguments. Those arguments do not appear in the event handler definition. The list of event handlers is the following:

on_entry
: Implicit Argument: `FromMode`

  Event handler to be executed on entrance of the mode. This happens as a reaction to mode transitions. `FromMode` is the mode from which the current mode is entered.

on_exit
: Implicit Argument: `ToMode`

  Event handler to be executed on exit of the mode. This happens as a reaction to mode transitions. The variable `ToMode` contains the mode to which the mode is left.

on_match
: Implicit Arguments: `Lexeme`, `LexemeL`, `LexemeBegin`, `LexemeEnd`

  This event handler is executed on every match that every happens while this mode is active. It is executed *before* the pattern-action is executed that is related to the matching pattern. The implicit arguments allow access to the matched lexeme and correspond to what is passed to pattern-actions.

on_after_match
: Implicit Arguments: `Lexeme`, `LexemeL`, `LexemeBegin`, `LexemeEnd`

  The `on_after_match` handler is executed at every pattern match. It differs from `on_match` in that it is executed *after* the pattern-action. To make sure that the handler is executed, it is essential that `return` is never a used in any pattern action directly. If a forced return is required, `RETURN` must be used.

Warning: When using the token policy 'queue' and sending tokens from inside the `on_after_match` function, then it is highly advisable to set the safety margin of the queue to the maximum number of tokens which are expected to be sent from inside this handler. Define:

```
-DQUEX_SETTING_TOKEN_QUEUE_SAFETY_BORDER=...some number...
```

on the command line to your compiler. Alternatively, quex can be passed the command line option `--token-policy-queue-safety-margin` followed by the specific number.

Note: Since `on_after_match` is executed after pattern actions have been done. This includes a possible sending of the termination token. When asserts are enabled, any token sending after the termination token may trigger an error. This can be disabled by the definition of the macro:

```
QUEX_OPTION_SEND_AFTER_TERMINATION_ADMISSIBLE
```

on_failure

Event handler for the case that a character stream does not match any pattern in the mode. This is equivalent to the `<<FAIL>>` pattern.

Note: The definition of an `on_failure` section can be of great help whenever the analyzer shows an unexpected behavior. Before doing any in-depth analysis, or even bug reporting, the display of the mismatching lexeme may give a useful hint towards a lack in the specified pattern set.

In a broader sense, 'on_failure' implements the 'anti-pattern' of all occurring patterns in a mode. That is it matches the shortest lexeme that cannot match any lexeme in the mode. It is ensured, that the input is increased at least by one, so that the lexical analyzer is not stalled on the event of match failure. In general, the non-matching characters are overstepped. If the analyzer went too fare, the 'undo' and 'seek' function group allows for precise positioning of the next input (see section ''stream navigation''). That the philosophy of `on_failure` is to catch flaws in pattern definitions. If anti-patterns or exceptional patterns are to be caught, they are best defined explicitly.

The definition of anti-patterns is not as difficult as it seems on the first glance–the use of pattern precedence comes to help. If the interesting patterns are defined before the all-catching anti-pattern, then the anti-pattern can very well overlap with the interesting patterns. The anti-pattern will only match if all preceding patterns fail.

Note: A slightly unintuitive behavior may occur when the token policy 'queue' is used, as it is by default. As any other token which is sent, it goes through the token queue. It arrives at the user in a delayed manner after the queue has been filled up, or the stream ends. In this case, an immediate exceptional behavior cannot be implemented by the token passing and checking the token identifier.

To implement an immediate exception like behavior, an additional member variable may be used, e.g.

```
body {
    bool   on_failure_exception_f;
}
init {
    on_failure_exception_f = false;
}
...
mode MINE {
   ...
   on_failure { self.on_failure_exception_f = true; }
}
```

Then, in the code fragment that receives the tokens the flag could be checked, i.e.

```
...
my_lexer.receive(&token);
...
if( my_lexer.on_failure_exception_f ) abort();
...
```

Note: In cases where only parts of the mismatching lexeme are to be skipped, it is necessary to 'undo' manually. That is if one wants to skip only the first non-matching character all but one characters have to be undone as shown below

```
on_failure {
    QUEX_NAME(undo_n)(&self, LexemeL - 1);
    /* in C++: self.undo_n(LexemeL - 1) */
    self_send1(QUEX_TKN_FAILURE, Lexeme);
}
```

`on_end_of_stream`

Event handler for the case that the end of file, or end of stream is reached. By means of this handler the termination of lexical analysis, or the return to an including file can be handled. This is equivalent to the `<<EOF>>` pattern.

`on_skip_range_open`

Implicit Arguments: `Delimiter`

If a range skipper was specified, then it is possible that an end of stream occurs without that the skipped range is actually terminated by the closing delimiter, for example

```
mode X : <skip_range: "/*" "*/"> {
    ...
}
```

skips over anything in between `/*` and `*/`. However, if an analyzed file contains:

```
/* Some comment without a closing delimiter
```

where the closing `*/` is not present in the file, then the event handler is called on the event of end of file. The parameter `Delimiter` contains the string of the missing delimiter.

There are event handlers which are concerned with indentation detection, in case that the user wants to build indentation based languages. They are discussed in detail in section *_sec-advanced-indentation-blocks*. Here, there are listed only to provide in overview.

on_indentation

Note: Since version 0.51.1 this handler is very loosely supported since indentation management has been improved heavily. It is likely that it will be removed totally.

The occurence of the first non-whitespace in a line triggers the `on_indentation` event handler. Note, that it is only executed at the moment where a pattern matches that eats part (or all) of the concerned part of the stream. This event handler facilitates the definition of languages that rely on indentation. `Indentation` provides the number of whitespace since the beginning of the line. Please, refer to section *Indentation Based Blocks* for further information.

on_indent
If an opening indentation event occurs.

on_dedent
If an closing indentation event occurs. If a line closes multiple indentation blocks, the handler is called *multiple* times.

on_n_dedent
If an closing indentation event occurs. If a line closes multiple indentation blocks, the handler is called only *once* with the number of closed domains.

on_nodent
In case that the previous line had the same indentation as the current line.

on_indentation_error
In case that a indentation block was closed, but did not fit any open indentation domains.

on_indentation_bad
In case that a character occured in the indentation which was specified by the user as being *bad*.

As it has been mentioned in many places before, event handlers are specified in the same way like pattern-actions.

3.4.2 Event Handler Pitfalls

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` *may* 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 MODE_A {
    ...
    {P_SOMETHING} {
        self << MODE_B;
        self.send(TKN_EVENT_MODE_TRANSITION);
        return;
    }
    ...
}
```

meaning that one desires to enter 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 `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:

```
delta_t =   probability_of_mode_switch
          * time_for_dereferencing_empty_function_call / 2;
```

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.

3.4.3 Inheritance

In Object-Oriented Programming a derived class may use the capabilities of its base class via *inheritance*. Analogously, if a quex mode is derived from another mode it inherits the same way the characteristics of that other mode. This promotes the following ideas:

1. A mode has somehow the character of a set or a category. If a set *B* is a subset of a set *A* then every element of *B* is conform to the constraints for being *A*, but has some special constraints for being in *B*.

   A mode `B` is being derived from a mode `A`, then it contains all characteristics of `A` but also some special ones that are not implemented in `A`.

2. Characteristics that are shared between modes are programmed in one single place. For example, all modes might show the same reaction on *End of File*. Thus the pattern-action pair concerned with it might best be placed in a common base mode. This ensures that changes which effect multiple modes can be accomplished from one single place in the source code.

The mechanism of inheritance has certain rules. Those rules can be expressed in terms of mode characteristics. Let mode `B` derived from mode `A`, then the following holds:

Pattern-Action Pairs

- Mode `B` contains all pattern-action pairs of `A`.
- The pattern-action pairs of `A` have higher precedence than the pattern-action pairs of `B`. That is, if a lexeme matches a pattern in `A` and a pattern in `B` of the same length, the pattern action of mode `A` is executed.

  This ensures that `A` imposes its character on `B`. Conversely, any mode derived from `A` can be assumed to show a behavior described in `A`.
- For patterns that appear in `B` and `A` the pattern action for the pattern in `A` is executed. This is a direct consequence of the previous rule.

Note: As a general rule, it can be imagined that if `B` is derived from `A` it is as if the pattern-action pairs of `A` are pasted in front of `B`'s pattern-action pairs itself.

Alternatively, pattern-actions of `B` could have been executed after the pattern-actions of `A` in case of interference. However, the decision of a brutal outruling was done because the high probability of creating a mess. Imagine, a lexical analyzer engine sends multiple tokens as a reaction to a pattern match. Further, the places where those tokens are send are not in one place, but distributed over multiple classes. Also, multiple concatinated mode transitions are very much prone to end up in total confusion. This is why pattern actions in a base mode outrule pattern actions in the derived mode.

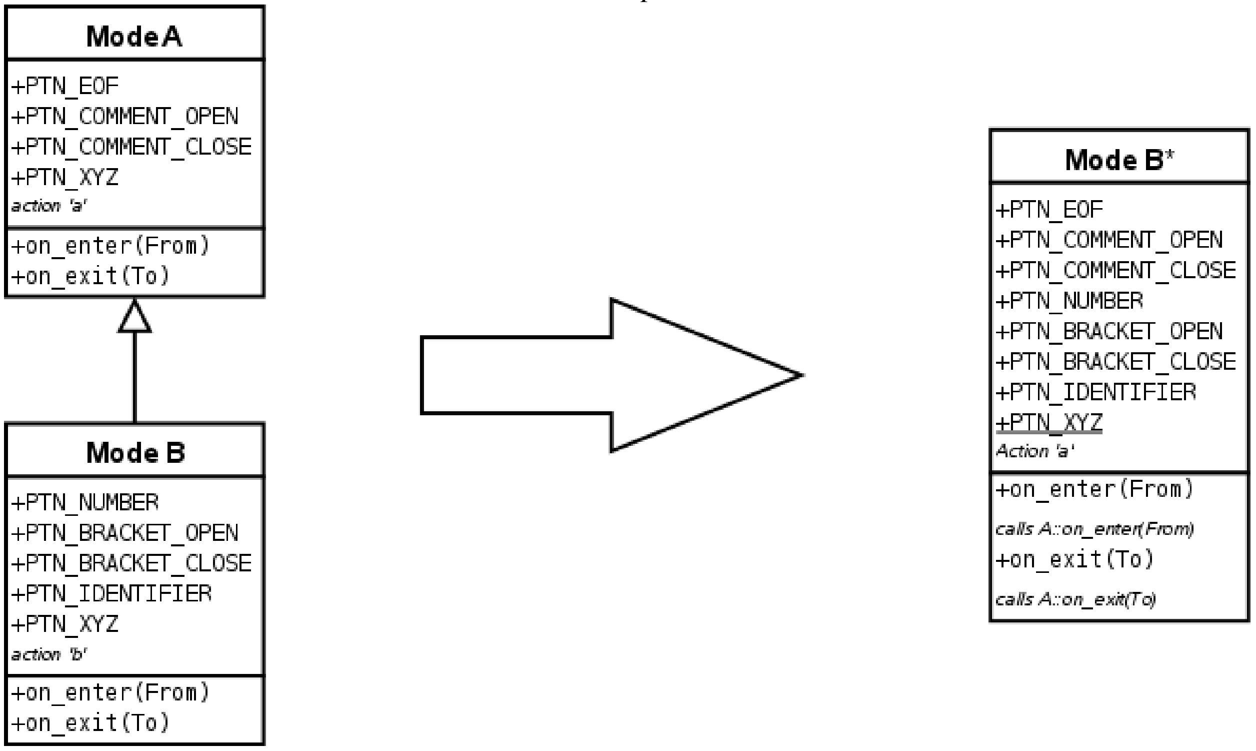
There are possibilities to influence this behavior to be discussed in <<section>>.

Event Handlers

- Event handlers in B are executed after event handlers of A.

  Note, that event handlers are not expected to perform mode transitions (see <<section>>) [4]. Also, tokens that are send from inside event handlers are *implicit* tokens, thus it is expected that those tokens are not necessarily tied to concrete lexemes. These assumptions makes the brutal outruling of the base mode over derived mode meaningless. Event handlers of base mode and derived mode may be executed both without an inherent source of confusion.

Figure *(this)* shows an example where a mode B is derived from mode A. The mode B* represents the internal result of the result of inheritance: patterns of the base mode *outrule* patterns of the derived mode, event handlers of the base mode *preceed* event handlers of the derived mode.



Multiple Inheritance

Quex allows multiple inheritance, i.e. a mode can inherit more than one base mode. Multiple inheritance is a potential cause of confusion when it comes to pattern match resolution or event handler sequence. The resolution sequence follows the 'depth-first approach'. As an example consider the following mode hierarchie:

```
mode A : B, C      { [a-z]{1,9} => T_A(Lexeme); }
mode C : E, F, G  { [a-z]{1,8} => T_C(Lexeme); }
mode G            { [a-z]{1,7} => T_G(Lexeme); }
```

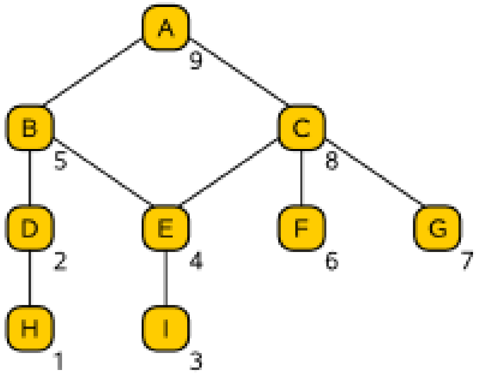
[4] An exception is on_indentation. However, this event handler is best placed in a common base mode, so that all modes use the same indentation mechanism.

```
mode F              { [a-z]{1,6} => T_F(Lexeme); }
mode B : D, E       { [a-z]{1,5} => T_B(Lexeme); }
mode E : I,         { [a-z]{1,4} => T_E(Lexeme); }
mode I              { [a-z]{1,3} => T_I(Lexeme); }
mode D : H          { [a-z]{1,2} => T_D(Lexeme); }
mode H              { [a-z]{1,1} => T_H(Lexeme); }
```

The according mode inheritance hierarchie is show in figure *Inheritance Resolution*. The letters in the boxes name the modes and the numbers at their right bottom edges indicate the mode's precedence. The applied depth-first approach implies two things:

1. Modes that mentioned first in the inheritance relationship have a higher precedence. In the above example `B` has a higher precedence then `C`, because it is mentioned before it in the definition `mode A : B, C { ... }`.
2. Modes located deeper in the inheritance structure precede modes which are located in derived classes. Thus, `I` has a higher precedence than `E` because `I` is derived from `E`.

The diamond problem, in our example classes `A`, `B`, `C`, and `E` is solved automatically. The derived mode `B` has a higher precedence than `C`, thus it is approached via the path of `B`. `B` treats the path of `E` after `C`. Thus, `E` has a precedence right after its closest base class `I`.



What sounds complicated is in practise very intuitive. Base mode patterns have high precedence, thus they have a high influence on the mode's behavior. This follows the idea that a derived mode has the character of its base mode plus some extra functionality. The fact that first mentioned base classes are considered first, is so intuitive that mentioning it is close to a tautology.

In any case, is is advisable when working with multiple inheritance, to have a look at the documentation string that quex produces about the pattern-action pairs in the generated engine source file. The aforementioned example caused quex to produce the following documentation:

```
/* MODE A:
 ...
 *     PATTERN-ACTION PAIRS:
 *       ( 75) H: [a-z]{1,1}
 *       ( 76) D: [a-z]{1,2}
 *       ( 77) I: [a-z]{1,3}
 *       ( 78) E: [a-z]{1,4}
 *       ( 79) B: [a-z]{1,5}
 *       ( 80) F: [a-z]{1,6}
 *       ( 81) G: [a-z]{1,7}
 *       ( 82) C: [a-z]{1,8}
 *       ( 83) A: [a-z]{1,9}
 ...
```

The first column of the table gives the index that was created for the pattern. It is an expression of precedence. The second column tells from what mode the pattern was inherited and the third column displays the pattern itself.

This means, that a word of one character is matched by mode `H`. All other modes can also match a word of one single character, but `H` has the highest precedence (it's the most 'base'-ic). A word of two characters is always matched by mode `D`. There are other modes which can also match a two character word, but `D` is the one with the highest precedence of all modes that can match.

Precedence Modifications

This section introduces indecent ways to violate the inheritance mechanisms. There might be situations, though, were a propper design of mode hierarchies is too labor-intensive. Then those features come handy–but they are to be used with care!

PRIORITY-MARK

Section <<section>> explained that base mode patterns outrule derived mode patterns.

For cases of real urgency, a keyword allows to struck the ruleset of pattern-action dispatching: `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. This requires, of course, for the pattern to be defined before. For example:

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

When the lexical analyser is in the `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 BASE {
  ...
  [a-z]+    { ... /* identifier */ }
  ...
}
mode DERIVED :
     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 differently on the pattern that `A` handles! 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 `A_base` and `A1` one containing desired patterns and the undesired patterns (see figure *Avoiding PRIORITY-MARK*). The original mode can be achieved by derivation from `A_base` and `A1`. The mode `B` can safely derive from `A_base`, other modes derive from `A` as if nothing happend. This is the clean way to avoid that undesired base class patterns have to high priority.

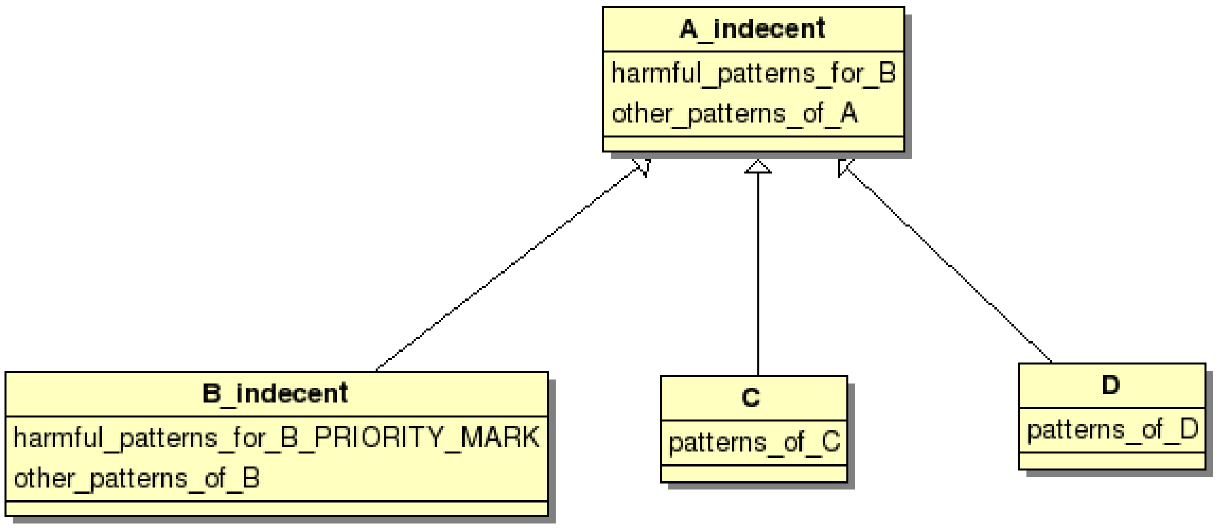


Figure 3.2: Solving conflicts with `PRIORITY-MARK`.

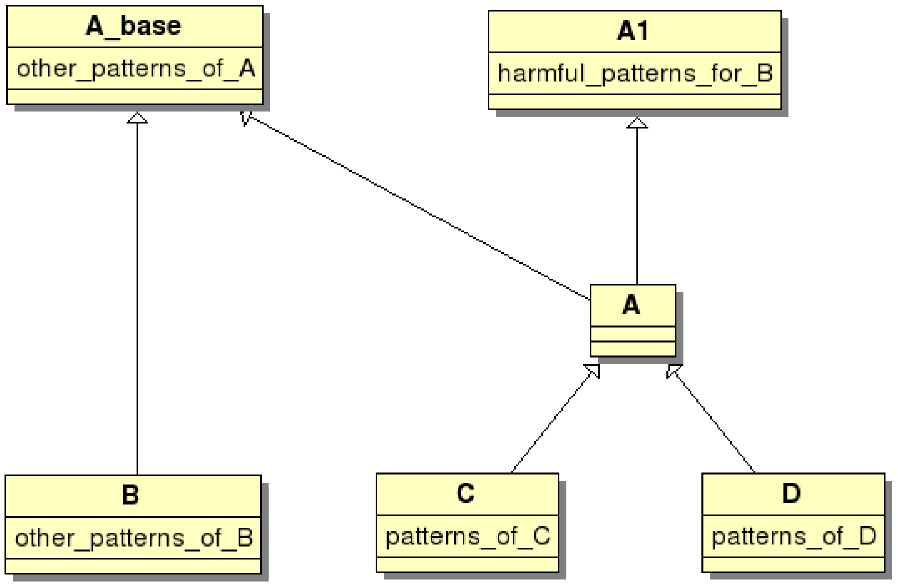


Figure 3.3: Avoiding the `PRIORITY-MARK` by re-design.

DELETION

Another indecent way of modifying the precedence behavior is by deleting a pattern that occurs in a base mode. The indecency lies in the fact that `B` claims to be derived from `A`, i.e. handles the patterns of `A`, but it does not. As in the case for priority, the appropriate re-design consist of splitting the base class: one base class containing the pattern to be deleted and another one without the pattern. The following example shows how a keyword pattern is deleted from the list of pattern that are inherited

```
mode A {
     ...
     [a-zA-Z_]+[a-zA-Z0-9_]   => TKN_IDENTIFIER(Lexeme);
     ...
}
mode B : A {
     /* IDENTIFIER would outrule all keywords. */
     [a-zA-Z_]+[a-zA-Z0-9_]   => DELETION;

     print                    => TKN_KW_PRINT;
     if                       => TKN_KW_IF;
```

```
    else                         => TKN_KW_ELSE;
}
```

On the first glance, the `DELETION` is an indispensable feature. In the above example the identifier pattern would indeed prevent any keyword pattern from matching. Any keyword matches also the identifier, but, since the identifier appears in the base mode it has preceedence. The use `DELETION` is a quick-fix, but a re-design by base mode splitting is the cleaner approach.

3.5 Show Time

The processing of a Quex-generated engine can be made visible if the compile macro:

```
QUEX_OPTION_DEBUG_SHOW
```

is defined (for example, by using `-DQUEX_OPTION_DEBUG_SHOW` as a compile option). Quex generates code that is instrumented with macros that are empty if this option is not set, and it if the compile option is defined the macros expand to debug output code. The output is printed to 'standard error' which is supposed to be immediately printed without buffering. Based on this output the inner function can be understood and analyzers can be debugged in detail. Such output looks like the following:

```
./tmp.cpp:152:  * init state ___________
./tmp.cpp:152:  input:            41 'A'
./tmp.cpp:152:  input position:   1
./tmp.cpp:195:  state 95 _______________
./tmp.cpp:195:  input:            42 'B'
./tmp.cpp:195:  input position:   2
./tmp.cpp:195:  state 95 _______________
./tmp.cpp:195:  input:            45 'E'
./tmp.cpp:195:  input position:   3
./tmp.cpp:195:  state 95 _______________
./tmp.cpp:195:  input:            52 'R'
...
./tmp.cpp:195:  state 95 _______________
./tmp.cpp:195:  input:            3A ':'
./tmp.cpp:195:  input position:   10
./tmp.cpp:536:  pre-terminal 7: [A-Z]+double-quote:double-quote
./tmp.cpp:539:  * terminal 7:   [A-Z]+double-quote:double-quote
```

This tells how the input triggers from one state to the next and finally ends up in the terminal for `[A-Z]:`. Especially, when reload is involved, it makes sense to set also the compile option:

```
QUEX_OPTION_DEBUG_SHOW_LOADS
```

If this option is set, additionally information about the buffer reload process is printed. For example:

```
FORWARD(entry)
```

```
   buffer front--------------------------->[0000] 0x0000
                                           [0001] 0x0041
                                           [0002] 0x0042
                                           [0003] 0x0045
                                           [0004] 0x0052
                                           ...
                                           [000A] 0x003A
                                           [000B] 0x0020
   lexeme start--------------------------->[000C] 0x0047
                                           [000D] 0x0055
   input, buffer back--------------------->[000E] 0x0000
```

is an output that tells that the start of the lexeme was at position '0xC' when the reload was triggered and the current input was at the end of the buffer content. The right-most numbers show the unicode value of the character that is the content of the buffer at the given position. This feature comes especially handy, if one tries to program a customizes character set converted. With the compile option:

```
QUEX_OPTION_DEBUG_SHOW_MODES
```

mode transitions can be made visible. With this option set, the analyzer documents each mode change by outputs such as:

```
| Mode change from GENERAL
|             to   MATH
...
| Mode change from MATH
|             to   FORMAT_STRING
...
| Mode change from FORMAT_STRING
|             to   MATH
...
```

CHAPTER

FOUR

CHARACTER ENCODINGS

Many cultures of the world in the history of mankind developped many types of scripts. A script in a more technical sense decribes the relationship between a *phonetic expression* or a *meaning* and its *graphical representation*[#f1]_. The atomic element of a writing is a character and its numerical representation is called a *code-point* or a *character code*. The formal relationship between characters and code-points is called an *encoding*. A lexical analyzer generator striving to be universal must therefore reflect the multitude of encodings. Much development in computer science at the end of the 2000's century has been made in a region of the world where the English was predominant. For this reason, the so called ASCII Standard encoding became widespread in the whole world, even though, it does only support English characters.

With the dramatic change of global interactions and communcations, the need for character encodings from other languages becomes indispensable. On of the major goals is to support the lexical analysis of any type of encoding. Quex supports the usage of character encodings in two ways:

1. The internal engine can be converted to run directly based on the desired encoding.
2. A converter may be plugged in which converts an incoming file stream into unicode. The lexical analyzer engine still runs on unicode characters.

The following section provides an overview about the lexical analysis process of a generated engine. The subsequent section elaborates on the adaption of the lexical analyzer engine for a particular codec. The remaining sections deal with the usage of libraries for the conversion of character encodings to unicode. Concluding on converters, a separate section explains how user generated converters can be plugged into the process.

4.1 The Process of Buffer Filling

This section discusses how a generated lexical analyzer reads data from a file or, more generally, from a stream. Figure *Process of loading data from a data stream* shows an overview over the process. It all starts with a character stream. The content is read via an input policy which basically abstracts the interface of the stream for the *buffer filler*. For the buffer filler, there is no difference between a stream coming from a `FILE` handle, and `ifstream` or an `istringstream`. The buffer filler has the following two tasks:

- Optionally convert incoming data appropriate for the character encoding of the lexical analyzer engine (Unicode).
- Fill the buffer with characters that are of *fixed* size.

Some character encodings, such as UTF-8, use a different number of bytes for the encoding of different characters. For the performance of the lexical analyzer engine it is essential that the characters have *constant* width. Iteration over elements of constant size is faster than over elements of dynamic size. Also the detection of content borders is very simple, fast, and not error-prone.

It becomes clear that the transfer of streaming data to the buffer memory can happen in different ways. This is reflected in the existence of multiple types of buffer fillers.

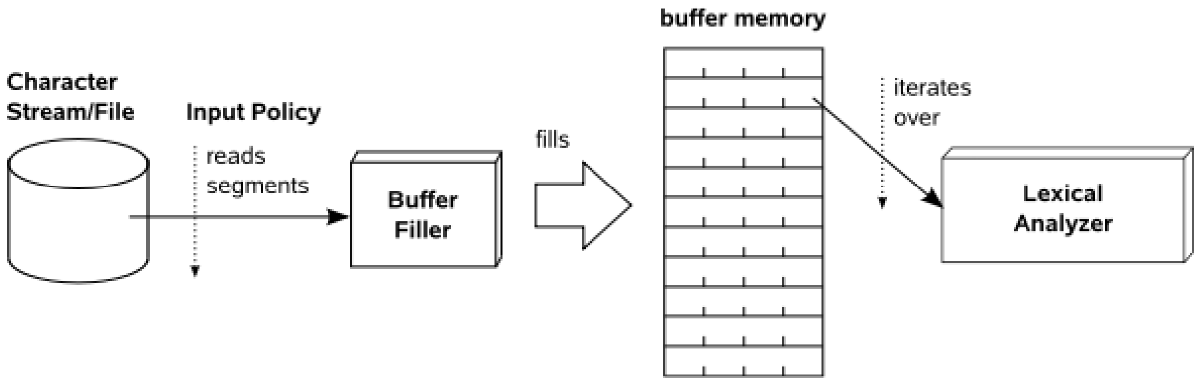


Figure 4.1: Loading data from a stream into the buffer memory.

As mentioned earlier, the usage of character encodings is supported by the plug-in of converter libraries, and by adaptions made to the analyzer engine. The following points should be kept in mind with respect to these two alternatives:

`Adapting the internal engine`

- Pro: Faster run-time performance, since no conversion needs to be done.
- **Contra: Freezing the design to a particular coding. If more than** one coding is to be dealt with, multiple engines need to be created.

`Using plugged-in converter libraries`

- Pro: Libraries provide a wider range of codecs to be used.
- **Contra: Using a converter implies a certain overhead with** respect to the memory footprint and run-time performance of an application.

Let the following terms be defined for the sake of clarity of the subsequent discussion:

- Character Width

  This determines the fixed number of bytes that each single character occupies in the buffer memory. It can be set via the command-line options `-b` or `--bytes-per-ucs-codepoint`. Directly related to it the the *character type*, i.e. the real C-type that is used to represent this entity. For example, `uint8_t` would be used to setup a buffer with one byte characters.

  Note: The character type is a C-type and does not make any assumptions about the encoding which is used. It is a means to specify the number of bytes that are used for a character in the buffer memory–not more.

- Data Stream Codec

  This is the character encoding of the input data stream. If non-unicode (or ASCII) streams are to be used, then a string must be passed to the buffer filler that tells from what encoding it has to convert.

- Lexical Analyzer Codec

  This is the character encoding of the analyzer engine. The codec determines what numeric value represents what character. For example, the greek letter α has a different numeric value in the codec 'UCS4' than it has in 'ISO8859-7'. By means of the command line flag `--codec` the engine's internal character encoding can be specified.

The next section focusses on the adaption of the internal analyzer engine to a particular codec. The following sections discuss the alternative usage of converters such as ICU and IConv to fill the analyzer buffer.

4.2 Analyzer Engine Codec

Apapting the internal codec of the generated lexical analyzer engine happens by means of the command line flag `--codec`. When a codec is specified the internal engine will no longer run on unicode, but rather on the specified codec. Consider the example of a lexical analyzer definition for some greek text to be encoded in ISO-8859-7:

```
define {
    CAPITAL   [-]
    LOWERCASE [-]
    NUMBER    [0-9][0-9.,]+
}

mode X :
<skip: [ \t\n]>
{
    {CAPITAL}{LOWERCASE}+  => QUEX_TKN_WORD(Lexeme);
    {NUMBER}                 => QUEX_TKN_NUMBER(Lexeme);
    km2|"%"|{LOWERCASE}+     => QUEX_TKN_UNIT(Lexeme);
}
```

The resulting state machine in Unicode is shown in Figure *Unicode Engine*. This engine could either fed by converted file content using a converting buffer filler. Alternatively, it can be converted so that it directly parses ISO8859-7 encoded characters. Specifying on the command line:

```
> quex (...) --codec iso8859_7 (...)
```

does the job. Note, that no character encoding name needs to be passed to the constructor, because the generated engine itself inhibits the codec. The character encoding name must be specified as '0x0' (if it is to be specified anyway).

```
quex::MyLexer    qlex(&my_stream);  // character encoding name = 0x0, by default
```

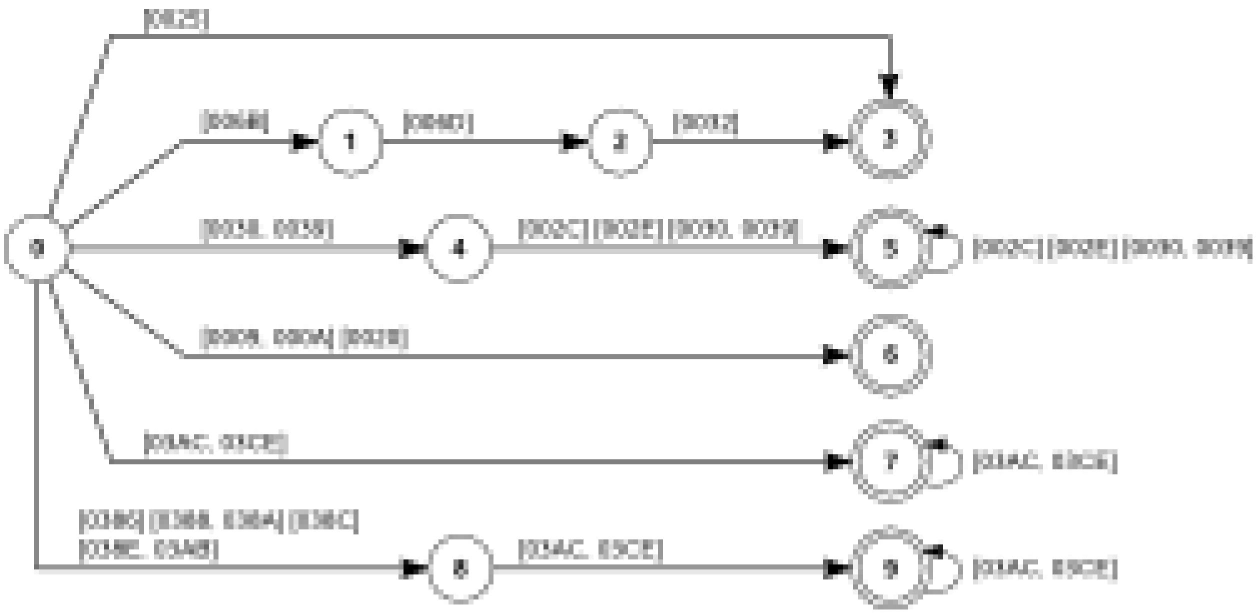


Figure 4.2: Sample lexical analyzer with an internal engine codec 'Unicode'.

The result is a converted state machine as shown in figure *ISO8859-7 Eninge*. that the state machine basically remains the same, only the transition rules have been adapted.

It is worth mentioning that the command line arguments `--codec-info` providing information about the list of codecs and `--codec-for-language` are usefule for making decisions about what codec to choose. Note, that in case that the engine is implemented for a specific codec, there is no 'coding name' to be passed to the constructor.

Note: When `--codec` is used, then the command line flag `--buffer-element-size` (respectively `-b`), it does not stand for the character's width. When quex generates an engine that actually understands the codec, this flag specifies the size of the code elements.

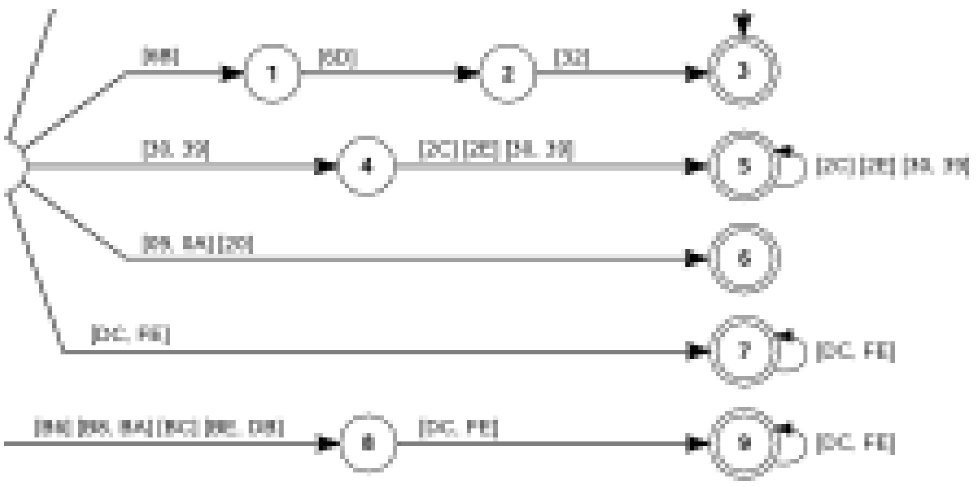


Figure 4.3: Sample lexical analyzer with an internal engine codec 'ISO8859-7' (greek).

For example, UTF8 covers the complete UCS code plane. So, its code points would require at least three bytes (0x0 - 0x10FFFF). However, the code elements of UTF8 are *bytes* and the internal engine triggers on bytes. Thus for codec `utf8 -b 1` must be specified. In the same way, UTF16 covers the whole plain, but its elements consist of *two bytes*, thus here `-b 2` must be specified.

4.2.1 The UTF8 Codec

The UTF8 Codec is different from all previously mentioned codecs in the sense that it encodes unicode characters in byte sequences of differing lengths. That means that the translation of a lexical analyzer to this codec cannot rely on an adaption of transition maps, but instead, must reconstruct a new state machine that triggers on the byte sequences.

Figure *UTF8 State Machine* shows the state machine that results from a utf8 state split transformation of the state machine displayed in figure *Sample lexical analyzer with an internal engine codec 'Unicode'*.. Where the codec adaptions happend on transition level, the main structure of the state machine remained in place. Now however, the state machine undergoes a complete metamorphosis.

Note: The skipper tag `<skip: ...>` cannot be used cautiously with the utf8 codec. Only those ranges can be skipped that lie undeneath the unicode value 0x7F. This is so since any higher value requires a multi-value sequence and the skipper is optimized for single trigger values.

Skipping traditional whitespace, i.e. `[ \t\n]` is still no problem. Skipping unicode whitespace

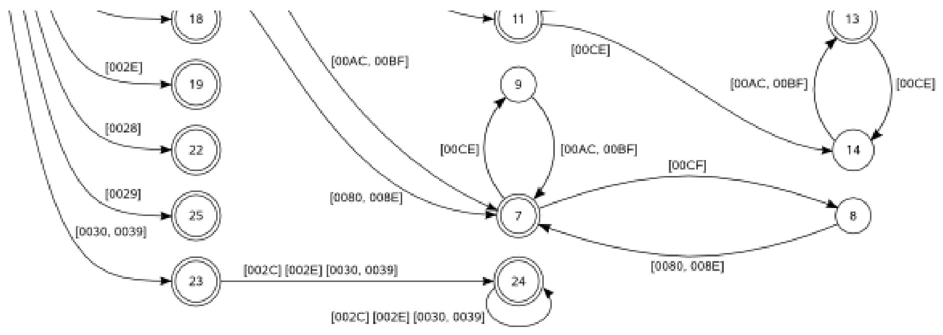


Figure 4.4: Sample lexical analyzer with an internal engine codec ‘UTF8’.

`[:\P{White_Space}:]` is a problem since the unicode property is carried by characters beyond 0x7F. In general, ranges above 0x7F need to be skipped by means of the 'null pattern action pair'.:

```
...
{MyIgnoredRange}    { /* Do nothing */ }
...
```

4.2.2 The UTF16 Codec

Similar to the UTF8 codec some elements of the unicode set of code points are encoded by two, others by four byte. To handle this type of codec, quex transforms the unicode state machine into a state machine that runs on triggers of a maximum range of 65536. The same notes and remarks made about UTF8 remain valid. However, they are less critical since only those code points are split into 4 bytes which are beyond 0xFFFF.

There is one important point about UTF16 which is not to be neglected: Byte Order, i.e. little endian or big endian. In order to work propperly the analyzer engine requires the buffer to be filled in the byte order which is understood by the CPU. UTF16 has three variants:

- UTF16-BE for big-endian encoded UTF16 streams.
- UTF16-LE for little endian encoded UTF16 streams.
- UTF16 which does not specify the byte order. Instead, a so call 'Byte Order Mark' (BOM) must be prepended to the stream. It consists of two bytes indicating the byte order:
  - `0xFE 0xFF` preceeds a big endian stream, and
  - `0xFF 0xFE` preceeds a little endian stream.

The analyzer generated by quex does not know about byte orders. It only knows the codec `utf16`. The provided stream needs to be provided in the byte order appropriate for particular CPU. This may mean that the byte order needs to be reversed during loading. Such a reversion can either passing the information to the constructor.

```
quex::MyLexer   qlex(fh, 0x0, /* ReverseByteOrderF */True);
```

Such a usage is appropriate if the codec is inverse to the machines codec. If, for example one tries to analyze a UTF16-BE (big endian stream) on an intel pentium (tm) machine, which is little endian, then the reverse byte order flag can be passed to the constructor. If a UTF16 stream is expected which specifies the byte order via a byte order mark (BOM), then the first bytes are to be read *before* constructor is called, or before a new stream is passed to the analyzer. In any case, the byte order reversion can be observed and adapted with the following member functions.

```
bool     byte_order_reversion();
void     byte_order_reversion_set(bool Value);
```

An engine created for codec `utf16` can be used for both, little endian and big endian data streams. The abovementioned flags allow to synchronize the byte order of the CPU with the data streams byte order by means of reversion, if necessary.

4.2.3 Summary

The command line flag `--codec` allows to specify the internal coding of the generated lexical analyzer. This enables lexical analyzers that run fast on codecs different from Unicode or ASCII. However, there are two drawbacks. First of all not all possible codecs are supported[#f1]_. Second, once an engine has been created for a particular codec, the codec is fixed and the engine can only run on this codec. Thus subsequent sections focuss on the 'converter approach' where the internal engine remains running on Unicode, but the buffer filler performs the conversion. It is not run time efficient as the internal engine codec, but more flexible, in case that the generated analyzer has to deal with a wide range of codecs.

4.3 Conversion

Quex provides support for character encodings by means of converters which are plugged into buffer fillers. The converters currently provided directly by quex are IBM's ICU library and the GNU IConv library. The latter is present by default on most Unix Machines, the former on Windows Systems and the like. The lexical analyzer engine is not aware of the conversion. It commands the buffer filler to filler to fill the buffer memory and then iterates over the content. It does not know what processes happen in the background. The buffer filler, however, is internally adapted to the library on which it relies. If you want to use character set conversion, for example to parse a file encoded in ISO-8859-3, then you need to have one of the supported libraries installed.

Quex can setup the buffer filler for the converter of your choice by command line arguments. Those are

`-icu`
: If you want to use IBM's ICU library for conversion.

`-iconv`
: If you want to use GNU IConv for conversion.

The decision which one to choose needs to be made should not have to do anything with the judgement about the clarity of the API of those libraries. The user is completely isolated from the details of that. Questions of concern might be

1. Make sure that the converter supports the encoding that you want to parse. Especially compressed formats might not be supported by every converter.
2. Computational speed: Compare both converters on some larger file and note down the differences in timing.
3. Memory Consumption: Compare the memory footprint of the binary for both converters.

In order to use the converter you need to pass the name of the encoding to the constructor of the lexical analyzer as shown below. For details consider the class definition of the generated lexical analyzer class.

```
quex::tiny_lexer  qlex("example.txt", "UTF-8");
```

4.4 Customized Converters

For 'normal' applications the provided converters provide enough performance and a sufficiently low memory footprint. There is no need for the user to implement special converters. However, it might become interesting when it comes to tuning. If someone knows that the UTF-8 encoding mostly reads characters from a specific code page it is possible to provide an optimized converter that is able to speed up a bit the total time for lexical analysis. Also, by means of this feature it is possible to scan personalized encodings.

Converters can be added to the quex engine very much in a plug-and-play manner. Nevertheless, this section is considered to be read by advanced users. Also, the effort required for implementing and testing a dedicated converter might exceed the time budgets of many projects. Lexical analyzers can be generated with quex, literally, in minutes. Implementing personalized converters together with the implementation of unit tests, however, can easily take up a whole week.

For everyone who is not yet scared off, here comes the explanation: A converter for a buffer filler is an object that contains the following function pointers described here in a simplified manner, without precise type definitions:

```
struct QuexConverter {

    void    (*open)(struct QuexConverter*,
                    const char* FromCodingName,
                    const char* ToCodingName);

    bool    (*convert)(struct QuexConverter*,
                       uint8_t**                  source,
                       const uint8_t*             SourceEnd,
                       QUEX_TYPE_CHARACTER**      drain,
                       const QUEX_TYPE_CHARACTER* DrainEnd);

    /* optional: can be set to 0x0 */
    void    (*on_conversion_discontinuity)(struct QuexConverter*);

    void    (*delete_self)(struct QuexConverter*);

    bool    dynamic_character_size_f;
};
```

Each function points to an implementation of a function that interacts the the library or the algorithm for character conversion. The role of each function is explained in the following list.

open (...)

This opens internally a conversion handle for the conversion from 'FromCodingName' to `ToCodingName`. Pass `0x0` as `ToCodingName` in order to indicate a conversion to unicode of size sizeof(QUEX_TYPE_CHARACTER).

It is the task of the particular implementation to provide the 'to coding' which is appropriate for sizeof(QUEX_TYPE_CHARACTER), i.e. ASCII, UCS2, UCS4.

Based on the `FromCodingName` this function needs to decide wether the character encoding is fixed size or dynamic size. According to this, the flag `dynamic_character_size_f` needs to be set. Setting it to `true` is safe, but may slow down the stream navigation.

convert (...)

Function `convert` tries to convert all characters given in `source` with the coding specified earlier to _open(...). `source` and `drain` are passed as pointers to pointers so that the pointers can be changed. This way the converter can inform the user about the state of conversion from source to drain buffer.:

```
START:
             *source              SourceEnd
             |                    |
      [      .....................]   source buffer

           *drain         DrainEnd
           |              |
      [....               ] drain buffer
```

At the beginning, 'source' points to the first character to be converted. 'drain' points to the place where the first converted character is to be written to.:

```
END:
                             *source
                             |
      [                      .....]   source buffer

                    *drain
                    |
      [.............      ] drain buffer
```

After convertsion, `source` points immediately behind the last character that was subject to conversion. `drain` points behind the last character that resulted from the conversion.

This function must provide the following return values

true

Drain buffer is filled as much as possible with converted characters.

false
More source bytes are needed to fill the drain buffer.

on_conversion_discontinuity(...)
The function `on_conversion_discontinuity` is called whenever a conversion discontinuity appears. Such cases appear only when the user navigates through the input stream (seek_character_index(...)), or with long pre-conditions when the buffer size is exceeded.

For 'normal' converters this function can be set to '0x0'. If a converter has an internal 'statefulness' that is difficult to be tamed, then use this function to reset the converter. Actually, the initial reason for introducing the function pointer was the strange behavior of the ICU Converters of IBM(R). Note, that if the function pointer is set, then the buffer filler does not use any hints on character index positions. This may slow down the seek procedure. If a pre-condition makes it necessary to load backwards, or the user navigates arbitrarily in the buffer stream there can be significant trade-offs.

delete_self(...)
This function closes the conversion handle produced with open(...) and deletes the object of the conversion object, in the same way as a virtual constructor does.

bool **dynamic_character_size_f**
This flag needs to be set to allow the buffer filler to allow its algorithms for character stream navigation.

For all functions mentioned above the user needs to implement to which those function pointer can point. In the next step, a user defined converter must be derived from `QuexConverter`. This should happen in the C-way-of-doing-it. That means, that `QuexConverter` becomes the first member of the derived class[f#1]_ . Consider the implementation for GNU IConv as an example

```
typedef struct {
    QuexConverter  base;

    iconv_t        handle;

} QuexConverter_IConv;
```

As another example consider the implementation of a converter class for IBM's ICU:

```
typedef struct {
    QuexConverter  base;

    UConverter*   from_handle;
    UConverter*   to_handle;
    UErrorCode    status;

    UChar         pivot_buffer[QUEX_SETTING_ICU_PIVOT_BUFFER_SIZE];
    UChar*        pivot_iterator_begin;
    UChar*        pivot_iterator_end;
} QuexConverter_ICU;
```

The role of the derived class (`struct`) is to contain data which is important for the conversion process. As an example, let the user defined converter functions be defined as

```
void
CryptoConverter_open(CryptoConverter*  me,
                     const char* FromCodingName,
                     const char* ToCodingName);

bool
CryptoConverter_convert(CryptoConverter*  me
                        uint8_t**       source,
                        const uint8_t*  SourceEnd,
                        QUEX_TYPE_CHARACTER**       drain,
                        const QUEX_TYPE_CHARACTER*  DrainEnd);
void
CryptoConverter_on_conversion_discontinuity(CryptoConverter*  me);

void
CryptoConverter_delete_self(CryptoConverter*  me);
```

Note, that the function signatures contain a pointer to `CryptoConverter` as a `me` pointer [1], where the required function pointers require a pointer to a `QuexConverter` converter object. This is no contradition. When the buffer filler creates an object of the derived type `CryptoConverter` is stores the pointer to it in as pointer to `QuexConverter`. The member functions, though, know that they work on an object of the derived class `CryptoConverter`.

Once, the access functions and the dedicated class have been defined a function needs to be implemented that creates a the converter. This needs to following:

1. Allocate space for the converter object. The allocation method for the converter object must correspond the deletion method in `delete_self`. A simple way to implent this is to rely on `malloc` like this

   ```
   ...
   me = (CryptoConverter*)malloc(sizeof(CryptoConverter));
   ...
   ```

   provided that the `delete_self` function is implemented like

   ```
   void
   CryptoConverter_delete_self(CryptoConverter*  me) {
       /* de-initialize resources */
       ...
       free(me);   /* corresponding call to 'malloc' */
   }
   ```

[1] A `me` pointer in C corresponds to the `this` pointer in C++. It gives access to the objects content.

Note, that quex provides you with sophisticated methods of memory management <<>>. This is the point where you can plug-in the memory allocation method for your converter object.

2. Set the function pointers for `open`, `convert`, `on_conversion_discontinuity`, and `delete_self`. The assignment of function pointers require a type cast, because the first argument differs. In the example above, the assignment of function pointers is

```
...
/* assume that 'me' has been allocated */
me->base.open         = (QUEX_NAME(QuexConverterFunctionP_open))CryptoConverter
me->base.convert      = (QUEX_NAME(QuexConverterFunctionP_convert))CryptoConver
me->base.delete_self = (QUEX_NAME(QuexConverterFunctionP_delete_self))CryptoCo
me->base.on_conversion_discontinuity = \
    (QUEX_NAME(QuexConverterFunctionP_on_conversion_discontinuity))\
    CryptoConverter_on_conversion_discontinuity;
...
```

   The macro `QUEX_NAME(...)` is used to identify the namespace, in case that the lexical analyzer is generated for 'C', where there are no namespaces–at the time of this writing.

3. Initialize the converter object. This is not the place to setup or allocate any conversion handle. The setup of conversion handles is the task of the `open` function. The iconv library, for example does not more than

```
...
me->handle = (iconv_t)-1;
...
```

   which assigns something useless to the conversion handle. This way it can be easily detected wether the `open` call was done propperly.

4. Return a pointer to the created object. That is the easiest part:

```
...
return me;
```

Now, the only thing that remains is to tell quex about the user's artwork. Using the command line option `--converter-new` (respectively `--cn`) the name of the converter creating function can be passed. If, for example, the this function is named `CryptoConverter_new`, then the call to quex needs to look like

```
> quex ... --converter-new CryptoConverter_new ...
```

Additionally, the compiler needs to know where to find the definition of you converter class, so you need to mention it in a `header` section.

```
header {
    ...
    #include "CryptoConverter.h"
    ...
}
```

The linker, also, has his rights and needs to be informed about the files of your converter and the files that implement the converter interface (e.g. `CryptoConverter.o`). This is all that is required to setup a user defined character converter.

4.5 Converter Helper Functions

The lexical analyzer engine generated by quex runs on a buffer that has potentially a different coding than what the user actually requires. When using a default token class, the two member functions

```
const std::string    pretty_char_text() const;
const std::wstring   pretty_wchar_text() const;
```

convert the member `text` inside the token into something appropriate for the types `char` and `wchar_t`. UTF8 is considered to be appropriate for `char`. Depending on the size of `wchar_t` the output may either be UCS4, i.e. UTF32, or UTF16 for systems where `sizeof(wchar_t) == 2`.

Converters, however, are provided for a much broader application. Thus, text can be converted to `char` and `wchar_t` whenever it is needed. Moreover, converters from the buffer's codec to UTF8, UTF16, UTF32 are provided for each generated lexical analyzer. For Unicode code based buffers the required functions are declared and implemented by including:

```
#include <quex/code_base/converter_helper/unicode>
```

and:

```
#include <quex/code_base/converter_helper/unicode.i>
```

When using subset of Unicode (e.g. ASCII, UCS4 or UTF32) as input encoding, then the buffer encoding is Unicode. Also, when using converters (see *Conversion*) the buffer is still encoded in Unicode. Then the Unicode converters in the files mentioned above can be used, as they are

```
QUEX_INLINE void   QUEX_NAME(unicode_to_utf8)(...);
QUEX_INLINE void   QUEX_NAME(unicode_to_utf16)(...);
QUEX_INLINE void   QUEX_NAME(unicode_to_utf32)(...);
QUEX_INLINE void   QUEX_NAME(unicode_to_char)(...);
QUEX_INLINE void   QUEX_NAME(unicode_to_wchar)(...);
```

The functions above convert a whole string. For single character conversions the same function group is present only ending with `_character` in the function name, i.e.

```
QUEX_INLINE void   QUEX_NAME(unicode_to_utf8_character)(...);
```

converts a single character from Unicode to UTF8.

The converters towards UTF8, UTF16, and UTF32 are the 'basis' and the converters towards `char` and `wchar_t` are mapped to one of them depending on what is appropriate for the size of `char` and `wchar_t`. Where the exact signature of each function follows the scheme of `converter` in the following code fragment

```
QUEX_INLINE void   converter(const SourceT** source_pp,
                             const SourceT*  SourceEnd,
                             DrainT**        drain_pp,
                             const DrainT*   DrainEnd);
```

The converter tries to convert as many characters from source to drain as possible; until it eventually reaches `SourceEnd` or the drain pointer reaches `DrainEnd`. The pointer to pointer arguments are required because the pointers need to be adapted. This facilitates a repeated call to the converter in case that either source or drain is fragmented.

Note: Depending on the drain's size the not all characters may be converted. A conversion for a character is not accomplished if the remaining drain size is less than the maximum character encoding. For UTF8 it is 8 bytes, for UTF16 4 bytes and for UTF32 for bytes.

The previous converter is present in C and C++. In C++ the following converters are available, which are possibly not as fast but more convenient

```
QUEX_INLINE string<uint8_t>   QUEX_NAME(unicode_to_utf8)(string<qtc>);
QUEX_INLINE string<uint16_t>  QUEX_NAME(unicode_to_utf16)(string<qtc>);
QUEX_INLINE string<uint32_t>  QUEX_NAME(unicode_to_utf32)(string<qtc>);
QUEX_INLINE string<char>      QUEX_NAME(unicode_to_char)(string<qtc>);
QUEX_INLINE string<wchar_t>   QUEX_NAME(unicode_to_wchar)(string<qtc>);
```

where `string<X>` is a shorthand for `std::basic_string<X>` and `string<qtc>` is a shorthand for `std::basic_string<QUEX_TYPE_CHARACTER>`. This means, that they can take a string of the type of the lexeme and return a string which is appropriate for the drain's codec.

When the internal engine is designed using `--codec` then the buffer codec is some dedicated character encoding. The `Lexeme` that is presented to the user has exactly the coding of the internal buffer. Precisely, it is a chain of `QUEX_TYPE_CHARACTER` objects that are encoded in the buffer's character encoding. Then quex has to generate the converters towards UTF8, UTF16, and UTF32. The converters follow the same scheme as for Unicode, only that 'unicode' is replaced by the codec's name, e.g.

```
QUEX_INLINE void   QUEX_NAME(iso8859_7_to_utf8)(...);
QUEX_INLINE void   QUEX_NAME(iso8859_7_to_utf16)(...);
QUEX_INLINE void   QUEX_NAME(iso8859_7_to_utf32)(...);
QUEX_INLINE void   QUEX_NAME(iso8859_7_to_char)(...);
QUEX_INLINE void   QUEX_NAME(iso8859_7_to_wchar)(...);
```

are the generated converters if `--codec iso8859-7` was specified. The converters can be included by

```
#include "MyLexer-converter-iso8859_7"   // Declarations
#include "MyLexer-converter-iso8859_7.i" // Implementations
```

Where `MyLexer` is the name of the generated lexical analyzer class and `iso8859_7` is the name of the engine's codec. Furthermore, there is a set of basic functions that are designed to support the abovementioned functions, but are still available for whom it may be useful. They are accessed by including

```
#include <quex/code_base/converter_helper/utf8>
#include <quex/code_base/converter_helper/utf16>
#include <quex/code_base/converter_helper/utf32>
```

and:

```
#include <quex/code_base/converter_helper/utf8.i>
#include <quex/code_base/converter_helper/utf16.i>
#include <quex/code_base/converter_helper/utf32.i>
```

They function exactly the same way as the dedicate converters for the `--codec` converters do. That is, their signatures are for example

```
QUEX_INLINE void   QUEX_NAME(utf8_to_utf8)(...);
QUEX_INLINE void   QUEX_NAME(utf8_to_utf16)(...);
QUEX_INLINE void   QUEX_NAME(utf8_to_utf32)(...);
QUEX_INLINE void   QUEX_NAME(utf8_to_char)(...);
QUEX_INLINE void   QUEX_NAME(utf8_to_wchar)(...);
```

in order to convert UTF8 strings to one of the target codecs. UTF16 and UTF32 work analogously.

4.6 Byte Order Mark

Unicode character stream may contain a byte order mark (BOM) at the beginning. Converters like IConv or ICU may possibly not eat the BOM. Also, if an engine runs in a genuine codec mode, i.e. without conversion, it does not know about BOMs. It is the user's task to cut the BOM–but Quex helps.

Quex provides functions to handle the recognition and cutting of the BOM from a character stream. To access the following files must be included:

```
#include <quex/code_base/bom>
#include <quex/code_base/bom.i>
```

These files declare and implement functions in in the main namespace (default `quex`). The function

```
QUEX_TYPE_BOM   QUEX_NAME(bom_snap)(InputHandleT* InputHandle);
```

checks whether the stream starts with a byte sequence that is identified as a byte order mark. If so, it steps over it so that the lexical analyzis can start immediately after it. An exception is the BOM for UTF7; this coding is identified, but the stream does not step over it. If UTF7 is detected, it has to be considered with care because it may actually a normal character sequence. The type `QUEX_TYPE_BOM` defines constants that identify BOMs. It is defined as follows:

Note: The BOM for UTF7 consists of the three bytes 0x2B, 0x2F, 0x76 plus one of 0x2B, 0x2F, 0x38, or 0x39. This corresponds to the caracters "+", "/", "v" and one of "+", "/", "8", or "9". All of them are normal Unicode Characters. Thus a normal Unicode stream could wrongly be identified as an UTF7 stream. Also, The last two bits are the beginning of a new character. Thus, the BOM cannot easily be snapped from the stream. Instead, the whole byte stream would have to be bit shifted.

For the aforementioned reasons, a UTF7 BOM is not cut from the byte stream.

Basic characteristics can be identified by binary 'and' operations. For example

```
using namespace quex;

bom_type = QUEX_NAME(bom_snap)(file_handle);

if( bom_type & (QUEX_BOM_UTF32 | QUEX_BOM_NONE) ) {
    ...
}
```

Checks whether a BOM of codec UTF32 was found, or if there was no BOM. The statement also holds if a UTF7 BOM is found, since the `QUEX_BOM_UTF7` has the `QUEX_BOM_NONE` bit raised. The exact information about the byte order can be detected by considering the whole value, e.g.

```
...
switch( bom_type ) {
/* Little ending BOM  => use the little endian converter. */
case QUEX_BOM_UTF_32_LE: codec_name = "UTF32LE"; break;
/* No BOM, or big endian bom => use big endian converter. */
case QUEX_BOM_UTF7:
case QUEX_BOM_NONE:
case QUEX_BOM_UTF_32_BE: codec_name = "UTF32BE"; break;
/* Unkown BOM => break                                    */
default:
     error_msg("Unknown BOM detected %s",
               QUEX_NAME(bom_name)(bom_type));
}
```

The example above, already, mentions another helper function that maps a BOM identifier to a human readable string

```
const char*      QUEX_NAME(bom_name)(QUEX_TYPE_BOM BOM);
```

If the user wishes to identify on some chunk of arbitrary memory the following function may be used

```
QUEX_TYPE_BOM    QUEX_NAME(bom_identify)(const uint8_t* const Buffer, size_t* n);
```

It receives a byte array in `Buffer` which must at least be of size four. It reports the found BOM as a return value and fills the number of bytes that the BOM occupies into what the second argument `n` points.

One important thing to notice is that the constructor does the first load from the data stream. Thus, if the BOM-cutting happens after the construction of the lexical analyzer object the 'cut' would not have any effect. Thus, the constructor call must be delayed after the call to `QUEX_NAME(BOM_snap)(...)`. If the initial call to the constructor cannot be avoided, then the call to the BOM snap function must be followed by a call to the `reset(...)` function. Also, an attempt to cut the BOM, after the constructor has done its initial load must fail.

Warning: Do not use the file or stream handle that is used for BOM cutting in the lexical analyzer constructor **before** the BOM cutting. If this is desired, then the constructor call **happen** after the BOM cut.

An example of how to cut the BOM can be found in `demo/*/003` in `example-bom.c`, respectively `example-bom.c`. The following code fragment shows an initialization in C language:

```
FILE*            fh = NULL;
EasyLexer        qlex;
QUEX_TYPE_BOM    bom_type = QUEX_BOM_NONE;

fh = fopen(file_name, "rb");

/* Either there is no BOM, or if there is one, then it must be UTF8 */
QUEX_TYPE_BOM    bom_type = QUEX_NAME(bom_snap)(fh);
if( (bom_type & (QUEX_BOM_UTF_8 | QUEX_BOM_NONE)) == 0 ) {
    printf("Found a non-UTF8 BOM. Exit\n");
    fclose(fh);
    return 0;
}

/* The lexer **must** be constructed after the BOM-cut */
QUEX_NAME(construct_FILE)(&qlex, fh, "UTF8", false);

/* Now, the qlex is ready for analyzis. */
...
```

CHAPTER

FIVE

DIRECT BUFFER ACCESS

A generated lexical analyzer runs its analysis on data located in a buffer. In general, this buffer is filled automatically by the underlying buffer management which relies on some type of input stream. It is, however, possible to access the buffer directly which, in some cases may be advantegous. In particular parsing the standard input, i.e. `cin` in C++ or `stdin` in C must rely on these mechanisms (see example in `demo/010/stdinlexer.cpp`). The following methods can be supplied to setup the analyzer's character buffer. The term 'framework' is used to refer to some kind of entity that delivers the data.

1. *Copying* content into the buffer.

   The framework provides its data in chunks and specifies itself the data's memory location. The location is possibly different for each received frame.

   Example: `demo/010/copy.cpp`.

2. *Immediate Filling* of the buffer.

   The framework writes its data in chunks into a memory location which is specified by the user.

   Example: `demo/010/fill.cpp`.

3. *Pointing* to a memory address where the buffer shall analyze data.

   The (hardware level) framework writes data into some pre-defined address space which is the same for each received frame.

   Example: `demo/010/point.cpp`.

Note: NOTE: Conversion on 'Copy' not implement in current version of quex.

With the methods *Copy* implicit character code converter may be applied. For *Immediate Fill* or *Pointing* the converters have to be called them explicitly. A so called 'buffer filler' can be created ouside the quex engine.

Note: In some frameworks, the buffer filling implies that a terminating zero character is set. This can cause an error:

```
exception: ... Buffer limit code character appeared as normal
           text content.
```

To avoid this, report one character less when using `buffer_fill_region_finish`, or make sure that the terminating zero is not copied.

In case of interrupted character streams, there is no direct way for the analyzer engine to determine wether a stream is terminated or not. Thus, either a 'end of analyzis' pattern must be introduced, or the analyzis is to be supervised by another thread which may end the analyzis based on time-out conditions. In the following description it is assumed that there exists a pattern that tells the analyzer that the stream is ended. It produces a `BYE` token. Direct buffer access can be performed by means of the following member functions:

```
QUEX_TYPE_CHARACTER*  buffer_fill_region_append(ContentBegin, ContentEnd);
void                  buffer_fill_region_prepare();
QUEX_TYPE_CHARACTER*  buffer_fill_region_begin();
QUEX_TYPE_CHARACTER*  buffer_fill_region_end();
size_t                buffer_fill_region_size();
void                  buffer_fill_region_finish(FilledCharacterN);
```

Analyzers that work directly on user managed memory should use the following constructor:

```
MyLexer(QUEX_TYPE_CHARACTER* MemoryBegin, size_t Size,
        const char*  CharacterEncodingName       = 0x0,
        const sizt_t TranslationBufferMemorySize = 0);
```

where `MyLexer` is the user specified engine name. The arguments `MemoryBegin` and `Size` may be set to zero, if the analyzer shall allocate the memory on its own. The last two arguments are only of interest if the incoming input is to be converted from a non-unicode character set to unicode/ASCII.

The input navigation when using direct memory access is fundamentally different from the navigation for file based input. For file based input the analyzer can navigate backwards in an arbitrary manner. This is not possible if the buffer is filled by the user. The maximum amount that can be navigated backwars [1] is determined by the fallback region. Its size is determined by the macro

QUEX_SETTING_BUFFER_MIN_FALLBACK_N

determines the maximum length of the pre-condition pattern. If no pre-condition pattern is used, this might be neglected.

Note: The presented methods are based on the token policy *User's Token*, i.e. the command line must contain `--token-policy users_token` when quex is called. Queue based policies

[1] Backward navigation may appear due to calls to `seek()`, but also when pre-conditions require a backward lexical analyzis (see *Pre- and Post- Conditions*).

might also be used, once the basic principles have been understood.

Warning: Is is highly recommdedable to define an `on_failure` handler for each lexical analyzer mode which sends something different from `TERMINATION`. The `TERMINATION` token is used in the strategies below to indicate the end of the currently present content. By default, a quex engine sends a `TERMINATION` token on failure, and thus the strategies below might hang up in an endless loop as soon as something is parsed which is not expected. A line such as

```
mode X {
    ...
    on_failure  => QUEX_TKN_MY_FAILURE_ID(Lexeme);
    ...
}
```

helps to avoid such subtle and confusing misbehavior.

5.1 Copying Content

The method of copying content into the analyzer's buffer can be used for the 'syntactically chunked input' (see *syntax-chunks*) and the 'arbitrarily chunked input' (see *arbitrary-chunks*). Copying of content implies two steps:

1. Copy 'used' content to the front of the buffer so that space becomes free for new content.
2. Copy the new content to the end of the current content of the buffer.

First, let us treat the case that the incoming frames are considered to be be *syntactically complete* entities–such as a command line, for example. This case is less complicated than the case where frame borders appear arbitrarily, because any trailing lexeme can be considered terminated and the analyzer does not need to wait for the next frame to possibly complete what started at the end of the last frame.

5.1.1 Syntactically Chunked Input Frames

The following paragraphs discuss the implementation of this use case. First, two pointers are required that keep track of the memory positions which are copied to the buffer.

```
typedef struct {
    QUEX_TYPE_CHARACTER* begin;
    QUEX_TYPE_CHARACTER* end;
} MemoryChunk;
```

A chunk of type MemoryChunk later contains information about the current content to be copied into the analyzer's buffer. .begin designates the beginning of the remaining content to be copied into the analyzer's buffer. .end points to the end of the currently available content as received from the messaging framework. The following segment shows which variables are required for the analyzis process.

```
int
main(int argc, char** argv)
{
    quex::tiny_lexer      qlex((QUEX_TYPE_CHARACTER*)0x0, 0);
    quex::Token*          token = 0x0;
    QUEX_TYPE_CHARACTER*  rx_buffer = 0x0; // receive buffer
    MemoryChunk           chunk;

    ...
```

The analyzis start with the following:

```
// -- trigger reload at loop start
chunk.end = chunk.begin;

// -- LOOP until 'bye' token arrives
token = qlex.token_p_switch(&token);
while( 1 + 1 == 2 ) {
    // -- Receive content from a messaging framework
    if( chunk.begin == chunk.end ) {
        // -- If the receive buffer has been read, it can be released.
        if( rx_buffer != 0x0 ) messaging_framework_release(rx_buffer);
        // -- Setup the pointers
        const size_t Size  = messaging_framework_receive_syntax_chunk(&rx_buffer);
        chunk.begin = rx_buffer;
        chunk.end   = chunk.begin + Size;
    }
    ...
```

At the beginning of the loop it is checked wether it is necessary to get new content from the messaging framework. If so, the previously received 'receive buffer' may be released for ulterior use. Then the messaging framework is called and it returns information about the memory position and the size where the received data has been stored. Now, the content needs to be copied into the analyzer's buffer.

```
chunk.begin = qlex.buffer_fill_region_append(chunk.begin, chunk.end);
```

This function call ensures that 'old content' is moved out of the buffer. Then, it tries to copy as much content as possible from chunk.begin to chunk.end. If there is not enough space to copy all of it, it returns the pointer to the end of the copied region. This value is stored in chunk.begin so that it triggers the copying of the remainder the next time of this function call. Now, the buffer is filled and the real analyzis can start.

```
// -- Loop until the 'termination' token arrives
while( 1 + 1 == 2 ) {
    const QUEX_TYPE_TOKEN_ID TokenID = qlex.receive();

    if( TokenID == QUEX_TKN_TERMINATION ) break;
    if( TokenID == QUEX_TKN_BYE )         return 0;

    cout << "Consider: " << string(*token) << endl;
}
```

When a `TERMINATION` token is detected a new frame must be loaded. The inner analyzis loop is left and the outer loop loads new content. If the `BYE` token appears the analyzis is done. Any token that is not one of the two abovementioned ones is a token to be considered by the parser. It follows the complete code of the analyzer for syntactically chunked input frames:

```
#include "tiny_lexer"
#include "messaging-framework.h"

typedef struct {
    QUEX_TYPE_CHARACTER* begin;
    QUEX_TYPE_CHARACTER* end;
} MemoryChunk;

int
main(int argc, char** argv)
{
    using namespace std;

    // Zero pointer to constructor --> memory managed by user
    quex::tiny_lexer       qlex((QUEX_TYPE_CHARACTER*)0x0, 0);
    quex::Token*           token = 0x0;
    QUEX_TYPE_CHARACTER*   rx_buffer = 0x0; // receive buffer
    MemoryChunk            chunk;

    // -- trigger reload of memory
    chunk.begin = chunk.end;

    // -- LOOP until 'bye' token arrives
    token = qlex.token_p();
    while( 1 + 1 == 2 ) {
        // -- Receive content from a messaging framework
        if( chunk.begin == chunk.end ) {
            // -- If the receive buffer has been read, it can be released.
            if( rx_buffer != 0x0 ) messaging_framework_release(rx_buffer);
            // -- Setup the pointers
            const size_t Size  = messaging_framework_receive_syntax_chunk(&rx_buffe
            chunk.begin = rx_buffer;
            chunk.end   = chunk.begin + Size;
        } else {
            // If chunk.begin != chunk.end, this means that there are still
            // some characters in the pipeline. Let us use them first.
        }

        // -- Copy buffer content into the analyzer's buffer
        chunk.begin = qlex.buffer_fill_region_append(chunk.begin, chunk.end);

        // -- Loop until the 'termination' token arrives
        while( 1 + 1 == 2 ) {
            const QUEX_TYPE_TOKEN_ID TokenID = qlex.receive();

            // TERMINATION => possible reload
            // BYE         => end of game
            if( TokenID == QUEX_TKN_TERMINATION ) break;
            if( TokenID == QUEX_TKN_BYE )         return 0;

            cout << "Consider: " << string(*token) << endl;
        }
    }
    return 0;
}
```

5.1.2 Arbitrarily Chunked Input Frames

In case that frames can be broken *in between* syntactical entities, more consideration is required. The fact that a pattern is matched does not necessarily mean, that it is the 'winning' pattern. For example, the frame at time '0':

```
frame[time=0]  [for name in print]
```

matches at the end `print` which might be a keyword. The lexical analyzer will return a KEYWORD token followed by a TERMINATION token. Let the above frame be continued as:

```
frame[time=0]  [for name in print]
frame[time=1]  [er_list: send file to name;]
```

which makes clear the actually the lexeme `printer_list` is to be matched. To deal with such cases one look-ahead token is required. A token is only to be considered, if the following token is not the TERMINATION token. If a TERMINATION token is returned by the `receive()` function, then the border of a frame has been reached. To match the last lexeme again after the appended content, the input pointer must be reset to the beginning of the previous lexeme. The procedure is demonstrated in detail in the following paragrpahs. The following code fragment shows all required variables and their initialization.

```
int
main(int argc, char**) {

    quex::tiny_lexer  qlex((QUEX_TYPE_CHARACTER*)0x0, 0);

    quex::Token    token_bank[2];      // Two tokens required, one for look-ahead
    quex::Token*   prev_token;         // Use pointers to swap quickly.

    QUEX_TYPE_CHARACTER*  rx_buffer = 0x0;  // A pointer to the receive buffer that
    //                                      // the messaging framework provides.

    MemoryChunk           chunk;       // Pointers to the memory positions under
    //                                 // consideration.

    QUEX_TYPE_CHARACTER*  prev_lexeme_start_p = 0x0; // Store the start of the
    //                                               // lexeme for possible
    //                                               // backup.

    // -- initialize the token pointers
    prev_token = &(token_bank[1]);
    token_bank[0].set(QUEX_TKN_TERMINATION);
    qlex.token_p_switch(&token_bank[0]);

    //
    // -- trigger reload of memory
    chunk.begin = chunk.end;
```

Two token pointers are used to play the role of look-ahead alternatingly. The tokens to which these pointers point are in the `token_array`. The current token id is set to `TERMINATION` to indicate that a reload happend. The loading of new frame content happens exactly the same way as for syntactically chunked input frames.

```
while( 1 + 1 == 2 ) {
    if( chunk.begin == chunk.end ) {
        if( rx_buffer != 0x0 ) messaging_framework_release(rx_buffer);
        const size_t  Size = messaging_framework_receive(&rx_buffer);
        chunk.begin = rx_buffer;
        chunk.end   = chunk.begin + Size;
    }
```

The inner analyzis loop, though, differs because a look-ahead token must be considered.

```
while( 1 + 1 == 2 ) {
    prev_lexeme_start_p = qlex.buffer_lexeme_start_pointer_get();

    // Let the previous token be the current token of the previous run.
    prev_token = qlex.token_p_switch(prev_token);
    const int TokenID = qlex.receive();

    // TERMINATION => possible reload
    // BYE         => end of game
    if( TokenID == QUEX_TKN_TERMINATION || TokenID == QUEX_TKN_BYE )
        break;

    // If the previous token was not a TERMINATION, it can be considered
    // by the syntactical analyzer (parser).
    if( prev_token->type_id() != QUEX_TKN_TERMINATION )
        cout << "Consider: " << string(*prev_token) << endl;
}
```

At the beginning of the loop the lexeme position is stored, because it might be needed to backup if a frame border is reached. The switch lets the current token become the look-ahead token and the previous token becomes the token to which the current token is to be stored. The end of the frame is detected with the `TERMINATION` token. The end of the analyzis is triggered by some `BYE` token which must appear in the stream. Both trigger a loop exit. If the current token (the 'look-ahead' token) is not a `TERMINATION` token, then the previous token can be considered by the parser.

The loop is exited either on 'end of frame' or 'end of analysis' as shown above. If the end of a frame was reached, the position of the last lexeme needs to be setup. The handling of the loop exit is shown below.

```
    // -- If the 'bye' token appeared, leave!
    if( current_token->type_id() == QUEX_TKN_BYE ) break;

    // -- Reset the input pointer, so that the last lexeme before TERMINATION
    //    enters the matching game again.
    qlex.buffer_input_pointer_set(prev_lexeme_start_p);
}
```

Warning: The procedure with one look-ahead token might fail in case that a pattern contains potentially a sequence of other patterns. Consider the mode

```
mode {
    "le"         => QUEX_TKN_ARTICLE;
    "monde"      => QUEX_TKN_WORLD;
    " "          => QUEX_TKN_SPACE;
    "le monde"   => QUEX_TKN_NEWSPAPER;
}
```

Where the begin of the `NEWSPAPER` pattern `le` can be made up of a sequence `le` (as `ARTICLE`) and ‘‘ ‘‘ (as `WHITESPACE`). Consider the frame sequence:

```
frame[time=0] [le ]
frame[time=1] [monde]
```

When the first frame border is reached now, the longest complete match holds, which is `le` (`ARTICLE`) and the analysis continues with the ‘‘ ‘‘ (`WHITESPACE`). Thus, `WHITESPACE` will be the last token before the TERMINATION token. The reconsideration triggered by the `TERMINATION` token is only concerned with the last token, i.e. `WHITESPACE`, but does not go back to the start of `le`. Eventually, the token sequence will be: `ARTICLE`, `SPACE`, `WORLD` instead of a single token `NEWSPAPER` which matches `le monde`.
A safe solution requires therefore *N* look-ahead tokens plus one, the current token. The *N* can be computed as the maximum number of sub-patterns into which a pattern in the analyzer might be broken down. The usual ‘keyword’-‘identifier’ race can be solved with one look-ahead token as explained above in this section.

The complete code to do the analyzis of arbitrarily chunked input frames is shown below.

```
#include "tiny_lexer"
#include "messaging-framework.h"

typedef struct {
    QUEX_TYPE_CHARACTER* begin;
    QUEX_TYPE_CHARACTER* end;
} MemoryChunk;

int
main(int argc, char** argv)
{
    using namespace std;

    quex::tiny_lexer  qlex((QUEX_TYPE_CHARACTER*)0x0, 0);
    quex::Token       token_bank[2];      // Two tokens required, one for look-aheac
    quex::Token*      prev_token;         // Use pointers to swap quickly.

    QUEX_TYPE_CHARACTER*  rx_buffer = 0x0;  // A pointer to the receive buffer that
    //                                        // the messaging framework provides.
    MemoryChunk           chunk;        // Pointers to the memory positions under
    //                                  // consideration.
    QUEX_TYPE_CHARACTER*  prev_lexeme_start_p = 0x0; // Store the start of the
    //                                               // lexeme for possible
    //                                               // backup.

    // -- initialize the token pointers
    prev_token = &(token_bank[1]);
    token_bank[0].set(QUEX_TKN_TERMINATION);
    qlex.token_p_switch(&token_bank[0]);

    // -- trigger reload of memory
    chunk.begin = chunk.end;

    // -- LOOP until 'bye' token arrives
    while( 1 + 1 == 2 ) {
        // -- Receive content from a messaging framework
        if( chunk.begin == chunk.end ) {
            // -- If the receive buffer has been read, it can be released.
            if( rx_buffer != 0x0 ) messaging_framework_release(rx_buffer);
            // -- Setup the pointers
            const size_t Size  = messaging_framework_receive(&rx_buffer);
            chunk.begin = rx_buffer;
            chunk.end   = chunk.begin + Size;
        }

        // -- Copy buffer content into the analyzer's buffer
        chunk.begin = qlex.buffer_fill_region_append(chunk.begin, chunk.end);

        // -- Loop until the 'termination' token arrives
        QUEX_TYPE_TOKEN_ID token_id = 0;
        while( 1 + 1 == 2 ) {
            prev_lexeme_start_p = qlex.buffer_lexeme_start_pointer_get();

            // Let the previous token be the current token of the previous run.
            prev_token = qlex.token_p_switch(prev_token);

            token_id = qlex.receive();

            // TERMINATION => possible reload
            // BYE         => end of game
            if( token_id == QUEX_TKN_TERMINATION ) break;
            if( token_id == QUEX_TKN_BYE )         return 0;

            // If the previous token was not a TERMINATION, it can be considered
            // by the syntactical analyzer (parser).
            if( prev_token->type_id() != QUEX_TKN_TERMINATION )
                cout << "Consider: " << string(*prev_token) << endl;
        }

        // -- If the 'bye' token appeared, leave!
        if( token_id == QUEX_TKN_BYE ) break;

        // -- Reset the input pointer, so that the last lexeme before TERMINATION
        //    enters the matching game again.
        qlex.buffer_input_pointer_set(prev_lexeme_start_p);
    }

    return 0;
}
```

5.2 Direct Filling

Instead of copying the input, the memory of the lexical analyzer can be filled directly. The address and size of the current region to be filled can be accessed via the member functions:

```
QUEX_TYPE_CHARACTER*  buffer_fill_region_begin();
QUEX_TYPE_CHARACTER*  buffer_fill_region_end();
size_t                buffer_fill_region_size();
```

In order to get rid of content that has already been treated the function

```
qlex.buffer_fill_region_prepare();
```

must be called before filling. As in the Copying case, it moves used content out of the buffer and, thus, creates space for new content. Finally, after new content has been filled in, the analyzer must be informed about the new 'end of memory'. This happens via a call to the function

```
qlex.buffer_fill_region_finish();
```

The core of an analyzis process based on direct filling looks like the following:

```
// -- Initialize the filling of the fill region
qlex.buffer_fill_region_prepare();

// -- Call the low lever driver to fill the fill region
size_t receive_n = receive_into_buffer(qlex.buffer_fill_region_begin(),
                                       qlex.buffer_fill_region_size());

// -- Inform the buffer about the number of loaded characters NOT NUMBER OF BYTES!
qlex.buffer_fill_region_finish(receive_n);
```

Note, that there is no `chunk.begin` information to be updated as in the case of Copying. The remaining framework for syntactically chunked and arbitrary chunked input is exactly the same as for the copying case. Source code examples can be reviewed in the `demo/010` directory.

5.3 Pointing

The 'Pointing' method implies that the user *owns* the piece of memory which is used by the lexical analyzer. A constructor call

```
quex::MyLexer   qlex((QUEX_TYPE_CHARACTER*)BeginOfMemory,
                     MemorySize,
                     (QUEX_TYPE_CHARACTER*)EndOfContent);
```

announces the memory to be used by the engine. Note, that the first position to be written to must be `BeginOfMemory + 1`, because the first element of the memory is filled with the buffer limit code. The buffer can, but does not have to, be filled initially. The third argument to the constructor must tell the end of the content. If the buffer is empty at construction time the end of content must point to `BeginOfMemory + 1`. The meaning of the arguments is again displayed in figure *User provided memory and its content.*.

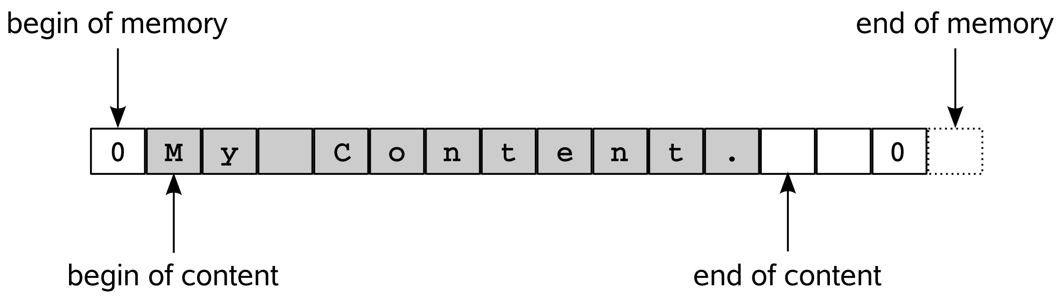


Figure 5.1: User provided memory and its content.

It is conceivable that the user fills this very same memory chunk with new content, so there must be a difference between the end of memory and the end of the content. The end of memory is communicated with the argument `MemorySize` and the end of content via `EndOfContent`. When the content of the buffer is filled a code fragment like

```
qlex.buffer_fill_region_finish(receive_n);
qlex.buffer_input_pointer_set(BeginOfMemory + 1);
```

tells the analyzer about the number of characters that make up the content. Also, it resets the input position to the start of the buffer. Now, the analyzis may start. The file `point.cpp` in the `demo/010` directory implements an example.

Note: The `Pointing` method is very seductive to be used in the context of hardware input buffers or shared memory. In such cases care is to be taken. The quex engine may put a terminating zero at the end of each a lexeme in order to facilitate the string processing. The definition of the macro

> QUEX_OPTION_TERMINATION_ZERO_DISABLED

prevents this, but the buffer limit code must still be set at the borders or the end of the content.

5.4 Character Conversions

It is very well possible to do character set conversions combined with direct buffer access. This enables the implementation command lines with UTF-8 encoding, for example. To enable character set conversion, the constructor must receive the name of the character set as the third argument, e.g.

```
quex::MyLexer  qlex((QUEX_TYPE_CHARACTER*)0x0, 0, "UTF-8");
```

And the engine must be created with a converter flag (`--iconv` or `--icu`) or one of the macros `-DQUEX_OPTION_CONVERTER_ICONV` or `-DQUEX_OPTION_CONVERTER_ICU` must be defined for compilation. Customized converters might also be used (see section

> *Customized Converters*). The

process of analyzis is the same, except for one single line in the code. Instead of appending plain content to the fill region it has to be converted. The interface functions take 'byte' pointers, since it is assumed that the input is raw. There are two possible cases:

1. The input is chunked arbitrarily and encoded characters might be cut at frame border. In this case, the function

   ```
   uint8_t*
   buffer_fill_region_append_conversion(uint8_t* Begin, uint8_t* End);
   ```

   ```
   has to be used. This is the *safe* way of doing character conversion.

   The return value is the pointer to the next content byte to be appended. If
   it is equal to ``End`` the whole content area from ``Begin`` to ``End``
   has beend appended.
   ```

2. The input chunks never cut in between an encoded character. In this case, the function

   ```
   uint8_t*
   buffer_fill_region_append_conversion_direct(uint8_t* Begin, uint8_t* End);
   ```

might be used. It does not use an intermediate buffer that stocks incoming data. Thus, it is faster and uses less memory. The raw buffer size of the converter can be set to zero, i.e. you can compile with `-DQUEX_SETTING_TRANSLATION_BUFFER_SIZE=0`.

The returned pointer corresponds to what has been said about the previous function.

This function is only to be used in case of 100% certainty that input frames only contain complete characters.

The two mentioned functions above are for the handling via 'copying'. On the other hand it is possible to use conversion with direct filling. Correspondent to the functions introduced in *Filling*, the following function group allows to fill the conversion buffer directly and perform the conversions.

```
void        buffer_conversion_fill_region_prepare();
uint8_t*    buffer_conversion_fill_region_begin();
uint8_t*    buffer_conversion_fill_region_end();
size_t      buffer_conversion_fill_region_size();
void        buffer_conversion_fill_region_finish(const size_t ByteN);
```

The functions work on `uint8_t` data, i.e. 'bytes' rather than `QUEX_TYPE_CHARACTER`. The interact directly with the 'raw' buffer on which the converter works.

For all three methods, there a sample applications in the `demo/010` directory.

CHAPTER

SIX

ADVANCED USAGE

The previous chapter was concerned with the implementation of lexical analyzer. This chapter focussed on advanced topics concerned with memory management and computation time. It not only sheds light on some internal mechanisms of the engine that quex generates, but also gives a toolset at hand that allows for the optimization of memory consumption and analysis speed.

6.1 Token Passing Policies

This section discusses how the generated analyzer communicates its results to its user. The primary result result of a lexical analysis step is a so called 'token identifier', that tells how the current input can be catagorized, e.g. as a 'KEYWORD', a 'NUMBER', or an 'IP_ADDRESS'. Additionally some information about the actual stream content might have to be stored. For example when a 'STRING' is identified the content of the string might be interesting for the caller of the lexical analyzer. This basic information needs to be passed to the user of the lexical analyzer. Token passing in quex can be characterized by

`Memory managements`

Memory management of the token or token queue can be either be accomplished by the analyzer engine or by the user.

By default the analyzer engine takes over memory management for the token or token queue. Also, it calls constructors and destructors as required.

`Token passing policy`

Possible policies are 'queue' and 'single'. Tokens can be stored in a *queue* to be able to produce more than one token per analyzis step. If this is not required a *single* token instance may be used to communicate analyzis results.

The default policy is 'queue' since it is the easiest and safest to handle during analyzis and event handlers.

The following two sections discuss automatic and user controlled token memory management separately. For each memory management type the two token passing policies 'single' and 'queue' are described.

6.1.1 Automatic Token Memory Management

Automatic token memory management is active by default. It relieves the user from anything related to memory management of the internal token or token queues. It is the seemingless nature which makes it possible to introduce the concepts of token passing policies without much distractive comments.

Queue

The default token policy is 'queue'. Explicitly it can be activated via the command line option `--token-policy`, i.e.:

```
> quex ... --token-policy queue ...
```

This policy is the safe choice and requires a minimum of programming effort. It has the following advantages:

- Using a token queue enables the sending of multiple tokens as response to a single pattern match.
- The events `on_entry`, `on_exit`, and `on_indentation` can be used without much consideration of what happens to the tokens.
- Multiple pattern matches can be performed without returning from the lexical analyzer function.

However, there is a significant drawback:

- The state of the analyzer is not syncronized with the currently reported token, but with the token currently on top of the queue.

A direct consequence of this is that parameters like line number, stream position etc. must be stored inside the token at the time it is created, i.e. inside the pattern action (see *Stamping Tokens*). For the above reason, the line number reported by the analyzer is not necessarily the line number of the token which is currently taken from the queue.

Figure *fig-token-queue* shows the mechanism of the token queue policy. The lexical analyzer fills the tokens in the queue and is only limited by the queue's size. The user maintains a pointer to the currently considered token `token_p`. As a result to the call to 'receive()' this pointer bent so that it points to the next token to be considered. The receive function itself does the following:

- if there are tokens left inside the queue, it returns a pointer to the next token.
- if not, then a call to the analyzer function is performed. The analyzer function fills some tokens in the queue, and the first token from the queue is returned to the user.

An application that applies a 'queue' token policy needs to implement a code fragment like the following

```
QUEX_TYPE_TOKEN*  token_p = 0x0;
...
while( analyzis ) {
    lexer.receive(&token_p);
    ...
    if( content_is_important_f ) {
        safe_token = new QUEX_TYPE_TOKEN(*token_p);
    }
}
```

All tokens primarily live inside the analyzer's token queue and the user only owns a reference to a token object `token_p`. The next call to 'receive()' may potentially overwrite the content to which `token_p` points. Thus, if the content of the token is of interest for a longer term, then it must be stored safely away.

Note: The usage of `malloc` or `new` to allocate memory for each token may have a major influence on performance. Consider a memory pool instead, which is allocated at once and which is able to provide token objects quickly without interaction with the operating system.

The size of the token queue is constant during run-time. Quex generates the size based on the the command line option `--token-queue-size`. An overflow of the queue is prevented since the analyser returns as soon as the queue is full.

Then the `--token-queue-safety-border` command line flag allows to define a safety region. Now, the analyzer returns as soon as the remaining free places in the queue are less then the specified number. This ensures that for each analyzis step there is a minimum of places in the queue to which it can write tokens. In other words, the safety-border corresponds to the maximum number of tokens send as reaction to a pattern match, including indentation events and mode transitions.

For low level interaction with the token queue the following functions are provided

```
bool   token_queue_is_empty();
void   token_queue_remainder_get(QUEX_TYPE_TOKEN**  begin_p,
                                 QUEX_TYPE_TOKEN**  end_p);
```

The first function informs tells whether there are tokens in the queue. The second function empties the queue. Two pointers as first and second argument must be provided. At function return, the pointers will point to the begin and end of the list of remaining tokens. Again, end means the first token after the last token. Note, that the content from `begin` to `end` may be overwritten at the next call to receive(). The following code fragment displays the usage of this function:

QUEX_TYPE_TOKEN* iterator = 0x0; QUEX_TYPE_TOKEN* begin_p = 0x0; QUEX_TYPE_TOKEN* end_p = 0; ... while(analyzis) {

qlex.receive(&iterator); if(content_is_important_f) {

store_away(iterator);

```
        }

        token_queue_take_remainder(&iterator; &end_p); while( iterator != end_p
        ) {

            ... if( content_is_important_f ) {

                store_away(iterator);

            } ++iterator;

        }

    }
```

Such a setup may be helpful if the lexical analyzer and the parse run in two different threads. Then the token tokens that are communicated between the threads could be copied in greater chunks.

Single

The policy 'single' is an alternative to the 'queue' policy. Explicitly it can be activated via the command line option `--token-policy single`, i.e.:

```
> quex ... --token-policy single ...
```

The advantages of a token queue are all lost, i.e.

– No more than one token can be sent per pattern match.

—Event handlers better not send any token or it must be sure that the surrounding patterns do not send token.

– Only one pattern match can be performed per call to the analyzer function.
The usage of `CONTINUE` in a pattern action is generally not advisable.

On the other hand, there is a major advantage:

– The state of the analyzer is conform with the currently reported token.

Thus, line numbers stream positions and even the current lexeme can be determined from outside the analyzer function. The analyzer function does not have to send a whole token. Actually, it is only required to send a token id. This token id is the functions return value and, with most compilers, stored in a CPU register. Since only one token object is stored inside the analyzer the data locality is improved and cache misses are less probable. The token passing policy 'single' is designed to support a minimal setup, which may improve the performance of an analyzer.

Note: Even with the token passing policy 'single' the ability to send multiple repeated tokens (using `self_send_n()`) at the same time remains intact. The repetition is implemented via a repetition counter not by duplication of tokens.

In contrast to the policy 'queue' a with 'single' the call to `receive()` does not bend any token pointer to a current token. Instead, the one token inside the analyzer can be referred to once. The receive function returns only the token id as a basis for later analyzis. If necessary, the application may rely on the token's content by dereferencing the pointer at any time. A code fragment for policy 'single' is shown below:

```
QUEX_TYPE_TOKEN*  token_p = 0x0;
...
token_p = qlex.token_p();
...
while( analyzis ) {
    token_id = qlex.receive();
    ...
    if( content_is_important_f ) {
        safe_token = new QUEX_TYPE_TOKEN(*token_p);
    }
}
```

Again, the next call to `receive()` may overwrite the content of the token. If it is needed on a longer term, then it must be copied to a safe place.

Note, that the function signature for the receive functions in 'queue' and 'single' are incompatible. The receive function for 'queue' has the following signature

```
void  receive(QUEX_TYPE_TOKEN**);  // 'queue' bends a pointer
```

where the signature in case of token passing policy 'single' is

```
QUEX_TYPE_TOKEN_ID  receive();     // 'single' only reports token id
```

This choice has been made so that user code breaks if the token policy is switched. Both policies require a different handling and a compile error forces the user to rethink his strategy[#f1]_. It is expected that the compiler reacts to a mismatching function call by pointing to a location where a matching candidate can be found. At this location, the user will find a hint that the token policy is not appropriate.

6.1.2 User Controlled Token Memory Management

The previous discussion has revealed a major drawback in automatic token memory management: Whenever the content of a token is of interest for a longer term, it must be copied. This could be spared, if the lexical analyzer is told were to write the token information. When an analyzis step is done the user takes the pointer to the finished token, and provides the analyzer with a pointer to an empty token. This token-switching reduces the need for memory allocation and disposes the need of copying. As a disadvantage, the user is now responsible for allocating and freeing of memory, as well as constructing and destructing of the involved tokens. User controlled memory management is activated via the command line option `--token-memory-management-by-user`, or `--tmmbu`, i.e. quex has to be called with:

```
> quex ... --token-memory-management-by-user ...
```

The following paragraphs will first discuss the 'single' token passing policy and then 'queue', because it is straight forward for the former and a little more involved for the later. The function signatures for both policies remain the same as with automatic token memory management. Depending whether the received token is of interest, the token inside the analyzer can be switched with the new token. This can be done by the functions

```
void                token_p_set(QUEX_TYPE_TOKEN*);
```

The first function only sets a new token. The currently set token pointer is overwritten. This is dangerous, since one might loose the reference to an allocated object. To avoid this the current token pointer can be read out using

```
QUEX_TYPE_TOKEN*   token_p();
```

To do the read-out and set operation in one step the function

```
QUEX_TYPE_TOKEN*   token_p_switch(QUEX_TYPE_TOKEN*);
```

is provided. It returns the a pointer to the currently set token inside the analyzer and sets the token pointer to what is provided as a second argument. It must point to a constructed, token object. A typical code fragment for this scenerio looks like

```
QUEX_TYPE_TOKEN  token_bank[2];
QUEX_TYPE_TOKEN* token_p = &token_bank[1];
...
lexer.set_token_p(&token_bank[0]);
...
while( analyzis ) {
    /* call to receive(...) switches token pointer */
    token_id = lexer.receive();
    ...
    if( content_is_important_f ) {
        store_away(lexer.token_p_switch(get_new_token());
    }
}
```

The idea of 'switching the thing on which the analyzer writes' can also be applied to the 'queue' token passing policy. The user provides a chunk of allocated and constructed tokens. The receive function fills this token queue during the call to `receive`. If the token queue is of interest, it can be taken out and the lexer gets a new, ready-to-rumble token queue. This actions can be take by means of the functions

```
void  token_queue_get(QUEX_TYPE_TOKEN** begin, size_t* size);
void  token_queue_set(QUEX_TYPE_TOKEN* Begin, size_t Size);
void  token_queue_switch(QUEX_TYPE_TOKEN** queue,
                         size_t*           size);
```

The following code fragment displays a sample application of this approach:

```
QUEX_TYPE_TOKEN*  iterator   = 0x0;
QUEX_TYPE_TOKEN*  queue      = 0x0;
size_t            queue_size = (size_t)16;
...
queue = get_new_token_queue(queue_size);
qlex.token_queue_set(queue, queue_size);
...
while( analyzis ) {
    iterator    = qlex.receive();
    /* consider the first token, the remainder is in the queue.
    ...
    qlex.token_queue_switch(&queue);

    /* Manual iteration over received token queue */
    for(iterator = queue; iterator != queue_watermark ; ++iterator) {
        ...
        if( content_is_important_f ) {
            store_away(iterator);
        }
    }
}
```

Token Construction and Destruction

As mentioned for user token memory management, the user owns the token's memory and is responsible for construction and destruction. In C++ construction and destruction happen implicitly, when on calls the `new` and `delete` operator, or if one defines a local variable and this variable runs out of scope, e.g.

```
QUEX_TYPE_TOKEN*   queue = QUEX_TYPE_TOKEN[16];
```

allocates space for 16 tokens, and calls their constructors. A call to delete, e.g.

```
delete [] queue;
```

invoques the token destructor and deletes all related memory. In C, however, construction and destruction must happen by hand, i.e.

```
QUEX_TYPE_TOKEN*   queue = (QUEX_TYPE_TOKEN*)malloc(sizeof(QUEX_TYPE_TOKEN) * 16);

for(iterator = queue; iterator != queue QueueSize; ++iterator)
    QUEX_NAME_TOKEN(construct)(iterator);
```

and

```
for(iterator = queue; iterator != queue QueueSize; ++iterator)
    QUEX_NAME_TOKEN(destruct)(iterator);

free(queue)
```

6.1.3 Remark on Token Passing

The user initiates an analyzis step by his call to `.receive()`. The analyzer tries to make sense out of the following sequence of characters, i.e. characterizes it as being a number, a keyword, etc. The character sequence that constitutes the pattern, i.e. the lexeme, has a begin and an end. Now, comes the point where this information has to be passed to the caller of `.receive()` through a token.

Note: The concept of a 'token' is that of a container that carries some information from the analyzer to the user. No tokens shall be allocated in a pattern action. During analyzis it is only filled, not created, and when `receive()` returns the user reads it what is in it.

Token allocation, construction, destruction, and deallocation is either done by the analyzer itself (default automatic token memory management), or by the caller of `.receive()` (user token memory management). It is not element of an analyzer step.

Token passing is something with happens notoriously often, and thus it is crucial for the performance of the lexical analyzer. During the pattern-match action the lexeme is referred to by means of a character pointer. This pointer is safe, as long as no buffer reload happens. Buffer reloads are triggered by analyzis steps. Depending on the token passing policy, the lexeme pointer is safe to be used without copying:

`Single`
: The pointer to the lexeme is safe from the moment that it is provided in the matching pattern action and while the function returns from `receive()`. When `receive()` is called the next time, a buffer reload may happen. If string to which the lexeme pointer points is important, then it must be copied before this next call to receive.

  Note: When `CONTINUE` is used, this initiates an analyzis step during which a buffer reload may happen.

`Queue`
: As with 'single', a lexeme pointer must be copied away before the next call to `receive()`. However, potentially multipl tokens are sent, and analyzis continues after a pattern has matched, then the lexeme pointer may be invalid after each step.

The default token implementation ensures, that the lexeme is stored away in safe place, during the pattern match action. However, for things that can be interpreted, such as numbers it may be advantegeous to interpret them directly and store only the result inside the token.

The above mentioned copy operations are done, because the buffer content *might* change while the lexeme content is of interest. A very sophisticated management of lexemes triggers on the actual buffer content change and safes all trailing lexemes at on single beat into a safe place. This is possible by registering a callback on buffer content change. The function to do this is

```
void  set_callback_on_buffer_content_change(
            void (*callback)(QUEX_TYPE_CHARACTER* ContentBegin,
                             QUEX_TYPE_CHARACTER* ContentEnd));
```

As long as the provided callback is not called, all lexeme pointers are safe. If the callback is called, then the current buffer content is overwritten and thus all related lexeme pointers will be invalid. To help with the decision which lexemes need to be stored away, the callback is called with pointers to the begin and end of the related content.

6.1.4 Summary

Token policies enable a fine grain adaption of the lexical analyzer. They differ in their efficiency in terms of computation speed and memory consumption. Which particular policy is preferable depends on the particular application. Token queues are easier to handle since no care has to be taken about tokens being sent from inside event handlers and multiple tokens can be sent from inside a single action without much worries. Token queues require a little more memory and a little more computation time than single tokens. Token queue can reduce the number of function calls since the analyzis can continue without returning until the queue is filled. Nevertheless, only benchmark tests can produce quantitative statements about which policy is preferable for a particular application.

Note: The implicit repetition of tokens is available for both policies. That is, `self_send_n()` is always available. The requirement that a multiple same tokens may be sent repeatedly does *not* imply that a token queue policy must be used.

6.2 User defined Token Classes

For some tokens the token identifier is sufficient information for the parse. This is true for tokens of the kind `OPERATOR_PLUS` which tells that a plus sign was detected. Other token types might require some information about the actually found lexeme. A token containing an identifier might have to carry the lexeme that actually represents the identifier. A number token might carry the actual number that was found. Queχ provides a default token class that allows the storage of a string object, an integer value. It is, however, conceivable that there is more complex information to be stored in the token, or that the information can be stored more efficiently. For this case, quex allows the definition of a customized token class. The first subsection introduces a convienent feature of quex that allows to specify a token class without worries. The second subsection explains the detailed requirements for a customized, user written token class.

Before continuing the reader should be aware, though, that there are two basic ways to treat token information:

1. Interpreting the lexeme at the time of lexical analysis.

   This requires in a sophisticated token class which can carry all related information.

2. Interpreting the lexeme at parsing time, when a syntax tree is build.

   This requires only a very basic token class that carries only the lexeme itself.

The interpretation of the lexeme needs to be done anyway. The first approach puts the weight on the sholders of the lexical analyzer, the second approach places the responsibility on the parser. For fine tuning both approaches should be studies with respect to their memory print and cache locality. It might not be the first approach which is always preferable.

The remaining framework of quex does not any adaptions to a customized token class. If the token class is designed according certain rules, then it fits smoothly in any generated engine.

6.2.1 Customized Token Classes

Quex has the ability to generate a customized token class that satisfies all formal requirements automatically: In the code section `token_type` a dedicated token type can be specified with a minimum amount of information. The underlying model of a general token class is displayed in figure *Token Class Model <fig-token-class-model>*. The memory of token object consists of three regions:

1. A region that contains *mandatory* information that each token requires, such as a token id, and (optionally) line and column numbers[#f1]_.

   Quex provides a means to specify the concrete type of those mandatory members.

2. A region that contains *distinct* members, i.e. members that appear in each token object, but which are not mandatory. Each place in the memory is associated with a specific type.

   For distinct members, both, type and member name can be specified.

3. A region of *union* members which is a chunk of memory which can be viewed differently, depending on the token id. This way, the *same* piece of memory can be associated with multiple types.

Note: All type names used in the `token_type` section must be available! This means, that definitions or the header files which define them must be either built-in, or mentioned in the `header` code section. If, for example, `std::string` and `std::cout` are to be used, the code should look like

```
header {
#include <string>
#include <iostream>
}
...
token_type {
    ...
     distinct {
         my_name   std::basic_string<QUEX_TYPE_CHARACTER>;
     }
     constructor {
         std::cout << "Hello Constructor\n";
     }
     ...
}
```

The following is a list of all possible fields in a `token_type` section. All fields are of the form `keyword`, followed by `{`, followed by content, followed by `}`.

standard

```
standard {
    id            : unsigned;
    line_number   : unsigned;
    column_number : unsigned;
}
```

The standard members are implemented in the actual token class with a preceeding underscore. That means, `id` becomes `_id`, `line_number` becomes `_line_number` and `column_number` becomes `_column_number`. Depending on the setting of the macros:

```
QUEX_OPTION_COLUMN_NUMBER_COUNTING
QUEX_OPTION_LINE_NUMBER_COUNTING
```

the members `_line_number` and `_column_number` are are enabled or disabled.

distinct

```
distinct {
    name        :  std::basic_string<QUEX_TYPE_CHARACTER>;
    number_list :  std::vector<int>;
}
```

union

```
union {
    {
       mini_x : int8_t;
       mini_y : int8_t;
    }
    {
       big_x  : int16_t;
       big_y  : int16_t;
    }
    position  : uint16_t;
}
```

The variable definitions inside these regions create automatically a framework that is able to deal with the token senders *Sending Tokens*. These token senders work like overloaded functions in C++. This means that the particularly used setters are resolved via the type of the passed arguments. For the three sections above the following setters are defined in the token class

```
void set(const QUEX_TYPE_TOKEN_ID ID);
void set_mini_x(const int8_t Value);
void set_mini_y(const int8_t Value);
void set_big_x(const int16_t Value);
void set_big_y(const int16_t Value);
void set_position(const int16_t Value);
```

Those are then implicitly used in the token senders. Note, that it is particularly useful to have at least one member that can carry a QUEX_TYPE_CHARACTER pointer so that it can catch the lexeme as a plain argument. As mentioned, the setter must be identified via the type. The above setters would allow token senders inside a mode to be defined as

```
mode TEST {
    fred|otto|karl => QUEX_TKN_NAME(Lexeme);
    mini_1         => QUEX_TKN_N1b(mini_x=1, position=1);
    big_1          => QUEX_TKN_N1c(big_x=1,  position=2);

}
```

The brief token senders may be rely on with *named* arguments, except for the two convenience pattern TKN_X(Lexeme) and TKN_X(Begin, End) –as mentioned in *usage-sending-tokens*.

If more flexibility is required explicit C-code fragments *C/C++ Code Segments* may be implemented relying on the current token pointer, i.e. the member function

```
QUEX_TYPE_TOKEN*  token_p()
```

and then explicitly call the named setters such as

```
...
self.token_p()->set_name(Lexeme);
self.token_p()->set_mini_1(LexemeL);
...
```

Standard operations of the token class can be specified via three code sections. The variable `self` is a reference to the token object itself.

Note: The assigment operator '=' is provided along the token class. However, there is a potential a major performance loss due to its passing of a return value. When copying tokens, better rely on the `__copy` member function.

constructor

This code section determines the behavior at construction time. The token class provides a default constructor and a copy constructor. In case of the copy constructor, the code segment is executed *after* the copy operation (see below). Example:

```
constructor {
    self.pointer_to_something = 0x0;    /* default */
    std::cout << "Constructor\n";
}
```

destructor

The destructor code segment is executed at the time of the destruction of the token object. Here, all resources owned by the token need to be released. Example:

```
destructor {
    if( self.pointer_to_something != 0x0 ) delete pointer_to_something;
    std::cout << "Destructor\n";
}
```

copy

This code segment allows for the definition of customized copy operations. It is executed when the member function `__copy` is called or when the assigment operator is used.

Implicit Argument: `Other` which is a reference to the token to be copied.

```
copy {
    std::cout << "Copy\n";
    /* Copy core elements: id, line, and column number */
    self._id = Other._id;
#       ifdef      QUEX_OPTION_TOKEN_STAMPING_WITH_LINE_AND_COLUMN
#       ifdef QUEX_OPTION_LINE_NUMBER_COUNTING
            self._line_n = Other._line_n;
#       endif
#       ifdef  QUEX_OPTION_COLUMN_NUMBER_COUNTING
            self._column_n = Other._column_n;
#       endif
#       endif

    /* copy all members */
    self.name        = Other.name;
    self.number_list = Other.number_list;
    /* plain content copy of the union content */
    self.content     = Other.content;
}
```

Alternative to copying each member one-by-one, it may be advantegous to rely on the optimized standard `memcpy` of the operating system. The default copy operation does exactly that, but is not aware of related data structures. If there are non-trivial related data structures, they need to be dealt with 'by hand'. This is shown in the following example:

```
copy {
    /* Explicit Deletion of non-trivial members */
    self.name.~std::basic_string<QUEX_TYPE_CHARACTER>();
    self.number_list.~std::vector<int>();

    /* Copy the plain memory chunk of the token object. */
    __STD_QUEX_memcpy((void*)&self, (void*)&Other), sizeof(QUEX_TYPE_TOKEN));

    /* Call placement new for non-trivial types: */
    new(&self.name)        std::basic_string<QUEX_TYPE_CHARACTER>(Other.name);
    new(&self.number_list) std::vector<int>(Other.number_list);
}
```

take_text
Optional, only a must if string accumulator is activated.

Whenever the accumulator flushes accumulated text it accesses the current token and passes its content to the function `take_text`. Inside the section the following variables can be accessed:

self
which is a reference to the token under concern.

analyzer
which gives access to the lexical analyzer so that decisions can be made according to the current mode, line numbers etc.

Begin, End
wich are both pointers of `QUEX_TYPE_CHARACTER`. `Begin` points to the first character in the text to be received by the token. `End` points to the first character *after* the string to be received.

LexemeNull
This is a pointer to the empty lexeme. The `take_that` section might consider not to allocate memory if the LexemeNull is specified, i.e. `Begin == LexemeNull`. This

happens, for example, in the C default token implementation.

The `take_text` section receives the raw memory chunk and is free to do what it wants with it. However, it must return a boolean value[#f2]_.

``true``

If `true` is returned the token took over the ownership over the memory chunk and claims responsibility to delete it. This is the case if the token maintains a reference to the text and does not want it to be deleted.

> **Warning:** It is highly dangerous to apply this strategy if `take_text` is called from inside the lexical analyzer. To claim ownership would mean that the token owns a piece of memory from the analyzer's buffer. This is impossible!
> There is a way to check whether the memory chunk is from the analyzer's buffer. A check like the following is sufficient
>
> ```
> if(     Begin >= analyzer.buffer._memory._front
>      && End   <= analyzer.buffer._memory._back ) {
>      /* Never claim ownership on analyzer's buffer ... */
>      /* Need one more space to store the terminating zero */
>      if( self.text != 0x0 ) {
>          QUEX_NAME(MemoryManager_Text_free(self.text);
>      }
>      self.text = QUEX_NAME(MemoryManager_Text_allocate)(sizeof(QUEX_TYPE_C
> } else {
>      /* Maybe, claim ownership. */
> }
> ```

``false``

If `false` is returned the caller shall remain responsible. This can be used if the token does not need the text, or has copied it in a 'private' copy.

If the token takes over the responsibility for the memory chunk it must be freed in a way that correponds the memory management of the string accumulator. The safe way to accomplish this is by adding something like

```
constructor {
     self.text = 0x0;
}

destructor {
     if( self.text ) {
          QUEX_NAME(MemoryManager_Text_free)((void*)self.text);
     }
}

take_text {
    self.text = Begin;
```

```
        return true;
    }
```

The `take_text` section is vital for all analyzers that rely on lexeme content. If it is not rock-solid, then the analyzer is on jeopardy. The default implementations in `$QUEX_PATH/code_base/token` contain disabled debug sections that may be copy-pasted into customized classes. For example for C, the default implementation in `CDefault.qx` contains the sections

```
    #   if 0
        {
            /* Hint for debug: To check take_text change "#if 0" to "#if 1" */
            const QUEX_TYPE_CHARACTER* it = 0x0;
            printf("previous:  '");
            if( self.text != 0x0 ) for(it = self.text; *it ; ++it) printf("%04X.",
            printf("'\n");
            printf("take_text: '");
            for(it = Begin; it != End; ++it) printf("%04X.", (int)*it);
            printf("'\n");
        }
    #   endif

        ... code of take_text ...

    #   if 0
        {
            /* Hint for debug: To check take_text change "#if 0" to "#if 1" */
            const QUEX_TYPE_CHARACTER* it = 0x0;
            printf("after:     '");
            if( self.text != 0x0 ) for(it = self.text; *it ; ++it) printf("%04X.",
            printf("'\n");
        }
    #   endif
```

As mentioned in the comment, these section can be activated by switching the `0` to `1` in the pre-processor conditionals.

Additional content may be added to the class' body using the following code section:

body

This allows to add constructors, member functions, friend declarations, internal type definitions etc. being added the token class. The content of this section is pasted as-is into the class body. Example:

```
body {
    typedef std::basic_string<QUEX_TYPE_CHARACTER> __string;

    void    register();
```

```
        void    de_register();
    private:
        friend  class MyParser;
    }
```

Note: The token class generator does not support an automatic generation of all possible constructors. If this was to be done in a sound and safe manner the formal language to describe this would add significant complexity. Instead, defining the constructors in the `body` section is very ease and intuitive.

When token repetition (see *Token Repetition*) is to be used, then the two following code fragments need to be defined

`repetition_get`
: The only implicit argument is `self`. The return value shall be the stored repetition number.

`repetition_set`
: Implicit arguments are `self` for the current token and `N` for the repetition number to be stored.

The code inside these fragments specifies where and how inside the token the repetition number is to be stored and restored. In most cases the setting of an integer member will do, e.g.

```
repetition_set {
    self.number = N;
}

repetition_get {
    return self.number;
}
```

In order to paste code fragments before and after the definition of the token class, the following two sections may be used.

`header`
: In this section header files may be mentioned or `typedef``s may be made which are required for the token class definition.  For example, the definition of token class members of type ``std::complex` and `std::string` requires the following header section.

```
header {
   #include <string>
   #include <complex>
}
```

`footer`

This section contains code to be pasted after the token class definition. This is useful for the definition of closely related functions which require the complete type definition. The code fragment below shows the example of an output operator for the token type.

```
footer {
     inline std::ostream&
     operator<<(std::ostream& ostr, const QUEX_TYPE_TOKEN_XXX& Tok)
     { ostr << std::string(Tok); return ostr; }
}
```

The token class is written into a file that can be specified via

`file_name = name ';'`

If no file name is specified the name is generated as engine name + `"-token-class"`. The name of the token class and its namespace can be specified via

`name = [namespace ... ::] token class name ';'`

Where the term after the = sign can be either

- Solely, the name of the token class. In this case the class is placed in namespace `quex`.
- A list of identifiers separated by `::`. Then all but the last identifier is considered a name space name. The last identifier is considered to be the token class name. For example,

  ```
  name = europa::deutschland::baden_wuertemberg::ispringen::MeinToken;
  ```

  causes the token class `MeinToken` to be created in the namespace `ispringen` which is a subspace of `baden_wuertemberg`, which is a subspace of `deutschland`, which is a subspace of `europa`.

In C++, when classes can be inherited they better provide a virtual destructor. If this is required the flag

`inheritable ';'`

can be specified. The following shows a sample definition of a `token_type` section.

```
token_type {
   name = europa::deutschland::baden_wuertemberg::ispringen::MeinToken;

   standard {
        id            :    unsigned;
        line_number   :    unsigned;
        column_number :    unsigned;
   }
   distinct {
       name        :  std::basic_string<QUEX_TYPE_CHARACTER>;
       number_list :  std::vector<int>;
   }
   union {
```

```
        {
           mini_x        : int8_t;
           mini_y        : int8_t;
        }
        {
           big_x         : int16_t;
           big_y         : int16_t;
        }
        who_is_that      : uint16_t;
    }
    inheritable;
    constructor { std::cout << "Constructor\n"; }
    destructor  { std::cout << "Destructor\n"; }
    body        { int __nonsense__; }
    copy        {
        std::cout << "Copy\n";
        /* Copy core elements: id, line, and column number */
        _id          = Other._id;
#       ifdef QUEX_OPTION_TOKEN_STAMPING_WITH_LINE_AND_COLUMN
#       ifdef QUEX_OPTION_LINE_NUMBER_COUNTING
               _line_n = Other._line_n;
#       endif
#       ifdef  QUEX_OPTION_COLUMN_NUMBER_COUNTING
               _column_n = Other._column_n;
#       endif
#       endif
        /* copy all members */
        name         = Other.name;
        number_list = Other.number_list;
        /* plain content copy of the union content */
        content      = Other.content;
    }
}
```

which results in a generated token class in C++:

```
class MeinToken {
public:
    MeinToken();
    MeinToken(const MeinToken& That);
    void __copy(const MeinToken& That);
    /* operator=(..): USE WITH CAUTION--POSSIBLE MAJOR PERFORMANCE DECREASE!
     *                BETTER USE __copy(That)                                */
    MeinToken operator=(const MeinToken& That)
    { __copy(That); return *this; }
    virtual ~MeinToken();
    std::vector<int>                      number_list;
    std::basic_string<QUEX_TYPE_CHARACTER> name;

    union {
        struct {
            int16_t                               big_x;
            int16_t                               big_y;
        } data_1;
        struct {
            int8_t                                mini_x;
            int8_t                                mini_y;
        } data_0;
        uint16_t                              who_is_that;
    } content;

public:
    std::basic_string<QUEX_TYPE_CHARACTER> get_name() const
    { return name; }
    void                                   set_name(std::basic_string<QUEX_TYPE_CHA
    { name = Value; }
    std::vector<int>                       get_number_list() const
    { return number_list; }
    void                                   set_number_list(std::vector<int>& Value)
    { number_list = Value; }
    int8_t                                 get_mini_x() const
    { return content.data_0.mini_x; }
    void                                   set_mini_x(int8_t& Value)
    { content.data_0.mini_x = Value; }
    int8_t                                 get_mini_y() const
    { return content.data_0.mini_y; }
    void                                   set_mini_y(int8_t& Value)
    { content.data_0.mini_y = Value; }
    uint16_t                               get_who_is_that() const
    { return content.who_is_that; }
    void                                   set_who_is_that(uint16_t& Value)
    { content.who_is_that = Value; }
    int16_t                                get_big_x() const
    { return content.data_1.big_x; }
    void                                   set_big_x(int16_t& Value)
    { content.data_1.big_x = Value; }
    int16_t                                get_big_y() const
    { return content.data_1.big_y; }
    void                                   set_big_y(int16_t& Value)
    { content.data_1.big_y = Value; }


    void set(const QUEX_TYPE_TOKEN_ID ID)
    { _id = ID; }
    void set(const QUEX_TYPE_TOKEN_ID ID, const std::basic_string<QUEX_TYPE_CHARACT
    { _id = ID; name = Value0; }
    void set(const QUEX_TYPE_TOKEN_ID ID, const std::vector<int>& Value0)
    { _id = ID; number_list = Value0; }
    void set(const QUEX_TYPE_TOKEN_ID ID, const std::basic_string<QUEX_TYPE_CHARACT
    { _id = ID; name = Value0; number_list = Value1; }
    void set(const QUEX_TYPE_TOKEN_ID ID, const int16_t& Value0, const int16_t& Val
    { _id = ID; content.data_1.big_x = Value0; content.data_1.big_y = Value1; }
    void set(const QUEX_TYPE_TOKEN_ID ID, const int8_t& Value0, const int8_t& Value
    { _id = ID; content.data_0.mini_x = Value0; content.data_0.mini_y = Value1; }
    void set(const QUEX_TYPE_TOKEN_ID ID, const uint16_t& Value0)
    { _id = ID; content.who_is_that = Value0; }


        QUEX_TYPE_TOKEN_ID    _id;
    public:
        QUEX_TYPE_TOKEN_ID    type_id() const      { return _id; }
        static const char*    map_id_to_name(QUEX_TYPE_TOKEN_ID);
        const std::string     type_id_name() const { return map_id_to_name(_id); }

#   ifdef     QUEX_OPTION_TOKEN_STAMPING_WITH_LINE_AND_COLUMN
#       ifdef QUEX_OPTION_LINE_NUMBER_COUNTING
    private:
        QUEX_TYPE_TOKEN_LINE_N  _line_n;
    public:
        QUEX_TYPE_TOKEN_LINE_N    line_number() const
        void                      set_line_number(const QUEX_TYPE_TOKEN_LINE_N Valu
#       endif
#       ifdef  QUEX_OPTION_COLUMN_NUMBER_COUNTING
    private:
        QUEX_TYPE_TOKEN_COLUMN_N  _column_n;
    public:
        QUEX_TYPE_TOKEN_COLUMN_N  column_number() const
        void                      set_column_number(const QUEX_TYPE_TOKEN_COLUMN_N
#       endif
#   endif
    public:

   int __nonsense__;
};
```

6.2.2 Formal Requirements on Token Classes.

The previous section introduced a convienent feature to specify customized token classes. If this is for some reason not sufficient, a manually written token class can be provided.

Note: It is always a good idea to take a token class generated by quex as a basis for a manually written class. This is a safe path to avoid spurious errors.

The user's artwork is communicated to quex via the command line argument `--token-class-file` which names the file where the token class definition is done. Additionally, the name and namespace of the token class must be specified using the option `--token-class`. For example:

```
> quex ... --token-class MySpace::MySubSpace::MyToken
```

specifies that the name of the token class is `MyToken` which is located in the namespace `MySubSpace` which is located in the global namespace `MySpace`. This sets automatically the following macros in the configuration file:

QUEX_TYPE_TOKEN
The name of the token class defined in this file together with its namespace.

```
#define QUEX_TYPE_TOKEN   my_space::lexer::MyToken
```

QUEX_TYPE0_TOKEN
The token class without the namespace prefix, e.g.

```
#define QUEX_TYPE0_TOKEN   MyToken
```

A hand written token class must comply to the following constraints:

- The following macro needs to be defined outside the class:

  QUEX_TYPE_TOKEN_ID
  Defines the C-type to be used to store token-ids. It should at least be large enough to carry the largest token id number.

  It is essential to use macro functionality rather than a typedef, since later general definition files need to verify its definition. A good way to do the definition is shown below:

  ```
  #ifndef    QUEX_TYPE_TOKEN_ID
  #   define QUEX_TYPE_TOKEN_ID               uint32_t
  #endif
  ```

  Note, that the header file might be tolerant with respect to external definitions of the token id type. However, since it defines the token class, it must assume that it has not been defined yet.

- A member function that maps token-ids to token-names inside the token's namespace

  const char* QUEX_NAME_TOKEN **(map_id_to_name)** (QUEX_TYPE_TOKEN_ID *Id*) 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`. The macro `QUEX_NAME_TOKEN` adapts the mapping function to the appropriate naming. Relying on the above function signature allows to define the appropriate function.

- 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:

  ```
  // map lexeme to my_type-object
  my_type tmp(split(Lexeme, ":"), LexemeL);
  self_send2(TKN_SOMETHING, LexemeL, tmp);
  return;
  ```

- It must provide a member `_id` token's identifier

  QUEX_TYPE_TOKEN_ID **_id**()

- The following function must be defined. Even an empty definition will do.

  void QUEX_NAME_TOKEN **(copy)** (Token**me*, const Token**Other*)

  void QUEX_NAME_TOKEN **(construct)** (Token*__*this*)

  void QUEX_NAME_TOKEN **(destruct)** (Token*__*this*)

inline void QUEX_NAME_TOKEN(destruct)($$TOKEN_CLASS$$* __this)

in the token's namespace which copies the content of token `Other` to the content of token `me`.

- If a text accumulator is to be used, i.e. QUEX_OPTION_STRING_ACCUMULATOR is defined, then there must be a function

  bool QUEX_NAME_TOKEN(take_text)(QUEX_TYPE_TOKEN* me, QUEX_TYPE_ANALYZER* analyzer, const QUEX_TYPE_CHARACTER* Begin, const QUEX_TYPE_CHARACTER* End)

  The meaning and requirements of this functions are the same as for the take_text section above.

- There must be member and member _line_n and _column_n for line and column numbers which are dependent on compilation macros. The user must provide the functionality of the example code segment below.

  ```
  #   ifdef     QUEX_OPTION_TOKEN_STAMPING_WITH_LINE_AND_COLUMN
  #       ifdef QUEX_OPTION_LINE_NUMBER_COUNTING
          public:
              size_t  _line_n;
              size_t  line_number() const                   { return _line_n; }
              void    set_line_number(const size_t Value) { _line_n = Value;
  #       endif
  #       ifdef  QUEX_OPTION_COLUMN_NUMBER_COUNTING
          public:
              size_t  _column_n;
              size_t  column_number() const                   { return _column_
              void    set_column_number(const size_t Value) { _column_n = Val
  #       endif
  #   endif
  ```

  The conditional compilation must also be implemented for the __copy operation which copies those values.

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.

6.3 Token Repetition

There are cases where a single step in the lexical analyzis produces multiple lexical tokens with the same token identifier. A classical example is the ‘block-close’ token that appears in indentation based languages, such as Python. Consider, for example the following python code fragment:

```
for i in range(10):
    if i in my_list:
```

```
            print "found"
    print "<end>"
```

When the lexical analyzer passes `"found"` and transits to the last `print` statement, it needs to close *three* indentation levels. Thus, three 'block-close' tokens need to be sent as result of one single analyzis step.

The obvious solution is to have a field inside the token that tells how often it is to be repeated. Indeed, this is what the token send macro `self_send_n()` does. For it to function, the policy of how to set and get the repetition number must be defined inside the token class (see `repetition_set` and `repetition_get` in *sec-token-class*).

Tokens that should carry the potential to be sent repeatedly must be mentioned in a `repeated_token` section inside the quex input sources, e.g.

```
repeated_token {
    ABC;
    XYZ;
    CLOSE;
}
```

where `QUEX_TKN_XYZ` is the token identifier that can be repeated. Now, the generated engine supports token repetition for the above three token ids. This means that `self_send_n()` can be used for them and the token receive functions consider their possible repetition. That means that if, for example,

```
self_send_n(5, QUEX_TKN_XYZ);
```

is called from inside the analyzer, then

```
token_id = my_lexer.receive();
```

will return five times the token identifier `QUEX_TKN_XYZ` before it does the next analyzis step. Note, implicit token repetition may have a minor impact on performance since for each analyzis step an extra comparison is necessary. In practical, though, the reduction of function calls for repetition largely outweighs this impact.

6.4 Line and Column Number Counting

Any compiler or lexical analyzer imposes rules on the text that it has to apply. Most of those texts are written by humans, and humans occasionally make errors and disrespect rules. A gentle compiler tells its user about his errors and also tells it about the place where the error occured. Using quex's ability to count lines and columns facilitates the task of pointing into the user's code. Line and column numbers are activated by means of the command line options

`-line-count`

-column-count

Both options can be set independently. The analyzer's member functions to access the current line and column number are:

int **line_number**()

int **line_number_at_begin**()

int **line_number_at_end**()

int **column_number**()

int **column_number_at_begin**()

int **column_number_at_end**()

The return the line and column number at the begin and the end of the currently matched lexeme. `line_number()` is a synonyme for `line_number_at_begin` and `column_number()` is a synonyme for `column_number_at_begin`.

Queχ tries to analyze patterns so that the column and line number counting can be reduced at run time, in many cases, to a plain constant addition. There is a general default algorithm to count lines and columns that 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 [1]. If the pattern contains, for example, a fixed number of newlines, then 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[#f2]_. For reasonable applications as known from popular programming languages the mechanisms of counting should be optimal.

For some purposes, it might be necessary to set the line and column number actively. Then the following member functions may be used:

```
void        line_number_set(size_t Y);
void        column_number_set(size_t X);
```

Line and column counting can be turned off individually by pre-processor switches.

QUEX_OPTION_COLUMN_NUMBER_COUNTING_DISABLED

QUEX_OPTION_LINE_NUMBER_COUNTING_DISABLED

These switches turn the related counting mechanisms off. It is possible that it runs a little faster[#f3]_. For serious applications, though, at least line number counting should be in place for error reporting.

[1] Even the indentation count algorithm is adapted to profit from knowledge about the patterns internal structure.

Warning: The member functions for reporting line and column numbers always report the *current* state. If the token policy `queue` (default) or `users_queue` (see *Token Passing Policies*) is used, then a these function only report correct values inside pattern actions!
From ouside, i.e. after a call to `.receive(...)` the line and column numbers represent the values for the last token in the queue. If precise numbers are required they are better stored inside the token at the time of the pattern match.

6.5 Stamping Tokens

Tokens can be stamped at the time that they are sent with the current line and/or column number. Indeed, this is what happens by default. If line or column counting is disabled, then also the stamping of the disabled value is disabled (see *sec-line-column-count*). The line and column numbers of a token can be accessed via the member functions

size_t **`line_number`**`()`

size_t **`column_number`**`()`

of each token object. The stamping happens inside the 'send()' functions. If the stamping procedure needs to be enabled or disabled against the abovementioned default settings the macro

`QUEX_OPTION_TOKEN_STAMPING_WITH_LINE_AND_COLUMN_DISABLED`

may be used to adapt to disable stamping mechanism. If line or column numbering is disabled, also the stamping of the corresponding value is disabled. Further, no member in the tokens is reserved to carry that value.

6.6 Stream Navigation

Independent of the underlying character encoding *Character Encodings* quex's generated lexical analyzers are equiped with functions for stream navigation on character basis. All required mechanisms of buffer loading and stream positioning is taking care of in the background. The current character index can be accessed by the member function:

```
size_t  tell();
```

The plain C versions of all mentioned functions are mentioned at the end of this section. The function `tell()` reports the current character index where the lexical analyzer is going to continue its next step. The lexical analyzer can be set to a certain position by means of the following member functions:

```
void     seek(const size_t CharacterIndex);
void     seek_forward(const size_t CharacterIndex);
void     seek_backward(const size_t CharacterIndex);
```

The first function moves the input pointer to an absolute position. The remaining two functions move the input pointer relative to its current position.

Warning: The usage of the above functions is similar to the usage of 'goto' in many popular programming languages. In particular it is possible to stall the lexical analyzer by uncoordinatedly `seek`-ing backwards.
Also, the `seek` functions *do not* take care of the line and column number adaption. This must be done manually, if desired. That is somehow the new line and column numbers must be determined and then set explicitly:

```
here = self.tell();
self.seek(Somewhere);

my_computation_of_line_and_column(here, Somewhere,
                                  &line_n, &column_n);

self.column_number_set(column_n);
self.line_number_set(line_n);
```

Note, that `column_number_set(...)` and `line_number_set(...)` define the column and line numbers of the next pattern to match.

The reading of the current lexeme can be undone by

```
void    undo();
void    undo_n(size_t DeltaN);
```

The function `undo()` sets the input pointer to where it was before the current lexeme was matched. With `undo_n(...)` it is possible to go only a specified number of characters backwards. However, it is not possible to go back more then the length of the current lexeme.

The undo functions have several advantages over the seek functions. First of all they are very fast. Since the lexical analyzer knows that no buffer reloading is involved, it can do the operation very quickly. Second, it can take care of the line and column numbers. The user does not have to compute anything manually. Nevertheless, they should be used with care. For example, if `undo()` is not combined with a mode change, it is possible to stall the analyzer.

In plain C, the stream navigation functions are available as

```
size_t  QUEX_NAME(tell)(QUEX_TYPE_ANALYZER* me);
void    QUEX_NAME(seek)(QUEX_TYPE_ANALYZER* me, const size_t);
void    QUEX_NAME(seek_forward)(QUEX_TYPE_ANALYZER*  me, const size_t);
void    QUEX_NAME(seek_backward)(QUEX_TYPE_ANALYZER* me, const size_t);
void    QUEX_NAME(undo)(QUEX_TYPE_ANALYZER* me);
void    QUEX_NAME(undo_n)(QUEX_TYPE_ANALYZER* me, size_t DeltaN_Backward);
```

where the `me` pointer is a pointer to the analyzer, e.g. `&self`. This follows the general scheme, that a member function `self.xxx(...)` in C++ is available in plain C as `QUEX_NAME(xxx)(me,`

...).

6.7 Include Stack

A useful feature of many programming languages is the feature to *include* files into a file. It has the effect that the content of the included file is treated as if it is pasted into the including file. For example, let `my_header.h` be a 'C' file with the content

```
#define TEXT_WIDTH 80
typedef short      my_size_t;
```

Then, a C-file `main.c` containing

```
/* (C) 2009 Someone */
#include "my_header.h"
int main(int argc, char** argv)
{
    ...
}
```

produces the same token sequence as the following code fragment where `my_header.h` is pasted into `main.c`

```
/* (C) 2009 Someone */
#define WIDTH 80
typedef short my_size_t;

int main(int argc, char** argv)
{
    ...
}
```

What happens internally is that the following:

1. An `#include` statement is found.
2. The analyzer switches to the included file
3. Analyzis continues in the included file until 'End of File'.
4. The analyzer switches back to the including file and continues the analyzes after the `#include` statement.

This simple procedure prevents users from writing the same code multiple times. Moreover, it supports centralized organization of the code. Scripts or configurations that are referred to in many files can be put into a central place that is maintained by trustworthy personal ensuring robustness of the overall configuration. The advantages of the simple `include` feature are many. In this section it is described how quex supports the inclusion of files.

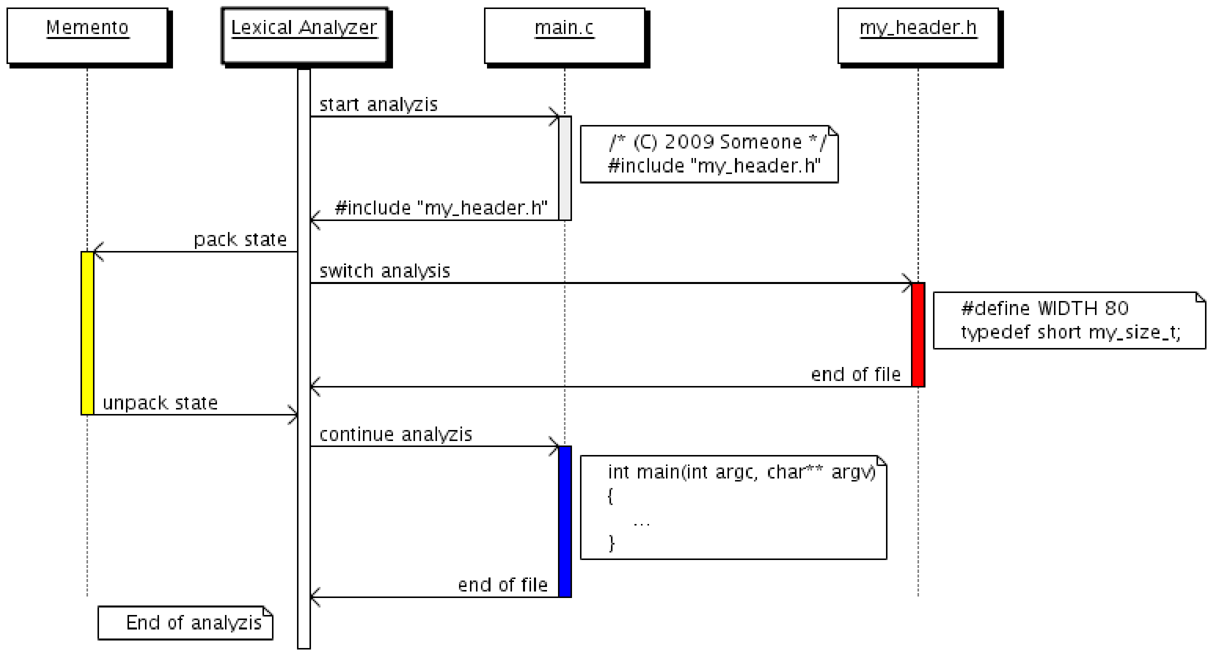


Figure 6.1: Inclusion of content from other files.

Figure *Inclusion of content from other files.* shows what happens behind the scenes. The lexical analyzer first reads the file `main.c` until it hits on an include statement. This statement tells it to read the content of file `my_header.h` and then continue again with the remainder of `main.c`. In order to do this the lexical analyzer needs to store his state-relevant [2] in a so called *memento* [3]. When the analysis of `my_header.h` terminates on 'end of file' the memento can be unpacked and the old state revived. The analyzis can continue in the file `main.c`.

If an included file includes another file, then the memento needs to know that there is some 'parent'. Thus the chain of mementos acts like a stack where the last memento put onto it is the first to be popped from it. An example is shown in figure *Recursive inclusion of files.* where a file A includes a file B which includes a file C which includes a file D. Whenever a file returns the correct memento is unpacked and the lexical analyzer can continue in the including file. The mementos are lined up as a list linked by 'parent' pointers as shown in *List of Mementos*. The parent pointers are required to find the state of the including file on return from the included file.

Note: The memento class always carries the name of the lexical analyzer with the suffix `Memento`. If for example an analyzer is called `Tester`, then the memento is called `TesterMemento`. The name of this class might be needed when iterating over mementos, see *Infinite Recursion Protection*.

[2] There are also variables that describe structure and which are not concerned with the current file being analyzed. For example the set of lexer modes does not change from file to file. Thus, it makes sense to pack relevant state data into some smaller object.

[3] The name memento shall pinpoint that what is implemented here is the so called 'Memento Pattern'. See also <<cite: DesignPatterns>>.

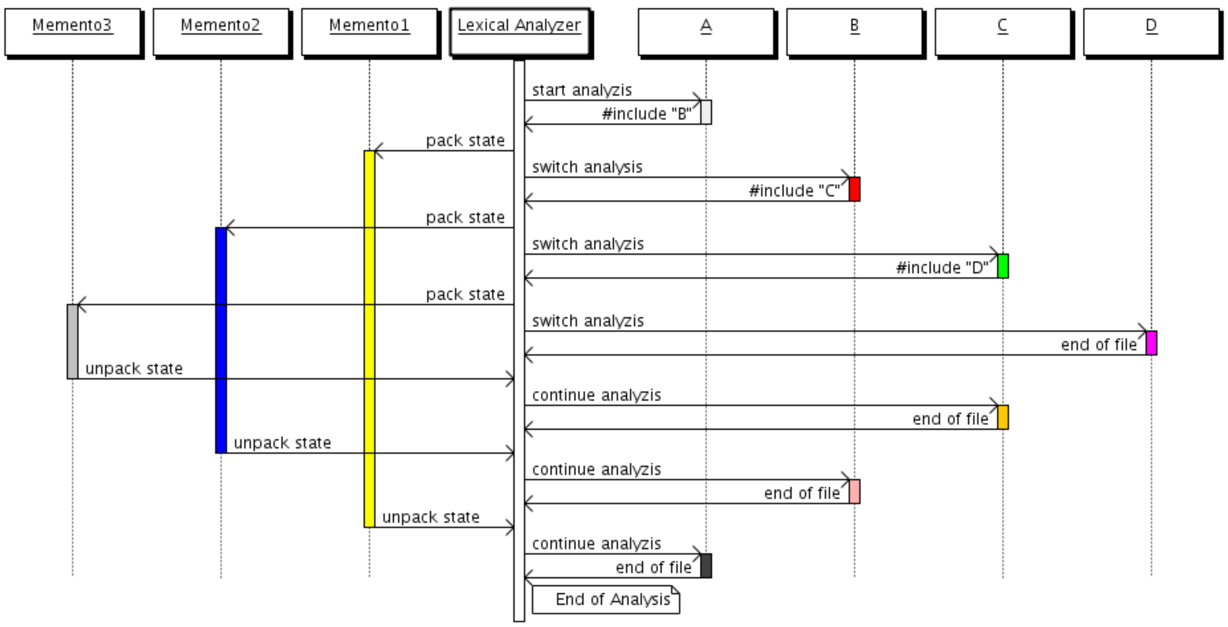


Figure 6.2: Recursive inclusion of files.

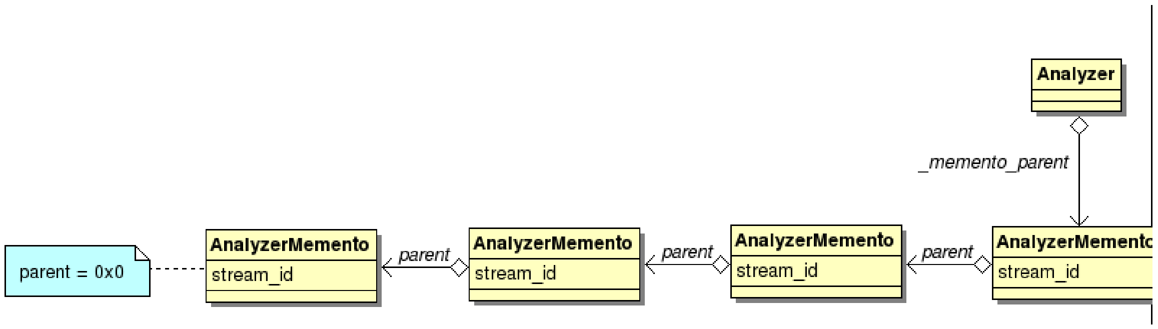


Figure 6.3: List of Mementos implementing the Include Stack.

Using the include stack handler revolves around the following member functions of the lexical analyzer:

void include_push<T>(QUEX_TYPE_CHARACTER* InputName, ...)

The 'push' function needs to be called when an `include`-like token is found. By means of this function the current lexical analyzer state is packed into a memento and the analysis of the new input is initialized. The function, actually, exists as two overloaded versions:

```
template <class InputHandleT> void
include_push<InputHandleT>(QUEX_TYPE_CHARACTER*   InputName,
                           const QUEX_NAME(Mode)* Mode                /* = 0x0
                           const char*            CharacterCodecName /* = 0x0

template <class InputHandleT> void
include_push<InputHandleT>(InputHandleT*          input_handle,
                           const QUEX_NAME(Mode)* Mode                /* = 0x0
                           const char*            CharacterCodecName /* = 0x0
```

The first function allows to include input streams, and the second function passed the input name to the memento pack function. The function must be called with template parameter specification, so that it knows what stream handle to deal with internally, e.g.

```
self.include_push<T>("myfile.dot", 0x0, 0x0);
```

will expect `T*` handle to be generated by `memento_pack`. The arguments to this function are the following:

InputName (or) input_handle

As shown above, in the first function version, the first argument can be a pointer to a character string `InputName` (of type `QUEX_TYPE_CHARACTER*`). If this is defined, the section `memento_pack` will receive an argument `input_handle` where:

```
*input_handle == 0x0
```

This tells the `memento_pack` section that the input_handle has to be provided based on the `InputName`, e.g. by opening a file or opening a TCP connection.

In the second function version, the first argument is the type of the template parameter, i.e. a pointer to an input handle. Inside the `memento_pack` section it can be referred to as `*input_handle`, where this argument is better unequal zero, otherwise the `memento_pack` section might assume that there is an `InputName`. Since no such name is given it will be clueless about how to include anything.

mode (*optional*)

Start mode in which the include file shall be analyzed. Defaultwise the initial mode is used.

BufferFillerType (*optional*)

Must be a constant defined in the enum `QuexBufferFillerTypeEnum`, e.g.

`QUEX_PLAIN`, `QUEX_CONVERTER`, etc.. By default the same filler type is used as in the current file.

`CharacterCodingName` (*optional*)
: Character encoding name for the converter. By default the same encoding is used as in the current file.

bool **`include_pop`**()
: This function unpacks a memento from the stack and puts the analyzer in the state in which it was before the inclusion. This function must be called when an 'end of file' is reached. Return values are

  `true`
  : if there was a memento and the old state was restored.

  `false`
  : if there was no memento. The analyzer is in the root of all files. The most appropriate reaction to this return value is to stop analyzis–at least for the given file.

Note: There are two state elements of the analyzer which are *not* subject to storage on inclusion. They are the

- token queue and the
- mode stack.

Both are not stored away or re-initialized when a new file is entered. If this is intended it must be done by hand.

6.7.1 Memento Extensions

As shown in *Sections* the `body` section allows for the definition of new members in the analyzer class. Imagine a scenario where new variables are defined in the analyzer class and those variables are file specific and are state relevant. In order to avoid to distort the data with the results of an included file, the variables have to be packed with the memento. Vice versa, the variable need to be restored when the memento gets unpacked.

In the quex source files the memento behavior can be influenced by the sections `memento`, `memento_pack`, and `memento_unpack`. The following implicit variables are available in `memento_pack` and `memento_unpack`:

`self`
: Reference to the lexical analyzer.

`memento`
: Pointer to the memento object.

InputName

Is the 'name' of the input that is to be included, such as a file name. The type of InputName is zero terminated string of type `QUEX_TYPE_CHARACTER*`. If this is different from `char`, then it might have to be converted to something that is compatible to what file handler functions require.

input_handle

Only for `memento_pack`.

Input handle that was passed to `include_push(...)`. It may be stored in an memento in order to be able to close it as soon as the included file is analyzed. There are two cases that might have to be distinguished:

1. `*input_handle` != 0x0

   This is the case, if the input handle has already been specified by the caller that triggered the include. It does not have to be provided.

2. `*input_handle` == 0x0

   Which indicates that the input handle has to be provided inside the `memento_pack` function based on the `InputName` argument.

This is explained in an example: For some purpose the user wants to count the number of whitespace characters and the occurencies of the word 'bug' in each of the analyzed files. For this purpose he adds some members to the analyzer class:

```
body {
     /* Code to be added to the class' body */
     size_t     whitespace_count;
     size_t     bug_count;
}
...
init {
    /* Code to be added to the constructor */
    whitespace_count = 0;
    bug_count        = 0;
}
...
mode A : {
     ...
     [ \t\n]+  { self.whitespace_count += strlen(Lexeme); }
     bug|Bug   { self.bug_count        += 1;              }
     ...
}
```

Since these variables are file specific, they need to be stored away on file inclusion. The `memento`-section allows extend the memento class. The class needs to contain variables that can store the saved information from the analyzer:

```
memento {
    size_t  __whitespace_count;
    size_t  __bug_count;
}
```

The content of the `memento_pack`-section extends the actions to be taken when a lexical analyzer is packed into a memento. We use the section to store away the variables for whitespace and bug counting:

```
memento_pack {
    memento->__whitespace_count = self.whitespace_count;
    memento->__bug_count        = self.bug_count;
    /* initialize variables */
    self.whitespace_count = 0;
    self.bug_count        = 0;
}
```

Note, that the variable `memento` provides access to the memento object. With the `memento_unpack`-section the actions of unpacking may be extended. Before the new file is being analyzed the member variables need to be re-initialized. When the file analyzis is terminated we need to ensure that the saved variables are restored:

```
memento_unpack {
     self.whitespace_count = memento->__whitespace_count;
     self.bug_count        = memento->__bug_count;
}
```

This is all that needs to be specified for including other files. The directory `demo/005` contains an example handling include stacks. Providing a language with such the 'include' feature is a key to propper code organization. By means of the above mechanisms quex tries to facilitate this task as much as possible. There is, however, a pitfall that needs to be considered. This will be the subject of the following section.

6.7.2 HOWTO

There are two basic ways to handle the inclusion of files during analyzis. First, files can be included from within analyzer actions, i.e. as consequence of a pattern match or an event. Second, they can be included from outside when the `.receive(...)` function returns some user define `INCLUDE` token. If the token policy `users_token` is used there is no problem with the second solution. Nevertheless, the first solution is more straightforward, causes less code fragmentation and involves less complexity. This section explains how to do include handling by means of analyzer actions, i.e. from 'inside'. The second solution is mentioned in the *Caveats* section at the end of this chapter.

The 'usual' case in a programming language is that there is some keyword triggering a file inclusion, plus a string that identifies the stream to be included, e.g.

```
\input{math.tex}
```

or:

```
include header.txt
```

The include pattern, i.e. `\input` or `include` triggers the inclusion. But, when it triggers the file name is not yet present. One cannot trigger a file inclusion whenever a string matches, since it may also occur in other expressions. This is a case for a dedicated mode to be entered when the include pattern triggers. This dedicated mode triggers an inclusion as soon as a string came in. In practical this looks like this:

```
mode MAIN : BASE
{
    "include"       => GOSUB(OPEN_INCLUDED_FILE);
    [_a-zA-Z0-9.]+  => QUEX_TKN_IDENTIFIER(Lexeme);
    [ \t\r\n]+      {}
}
```

When the trigger `include` matches in mode `MAIN`, then it transits into mode `OPEN_INCLUDED_FILE`. It handles strings differently from the `MAIN` mode. Its string handling includes an `include_push` when the string has matched. Notice, that mode `MAIN` is derived from `BASE` which is to be discussed later on. The mode `OPEN_INCLUDED_FILE` is defined as

```
mode OPEN_INCLUDED_FILE : BASE
{
    [a-zA-Z0-9_.]+ {
        /* We want to be revived in 'MAIN' mode, so pop it up before freezing. */
        self.pop_mode();
        /* Freeze the lexer state */
        self.include_push<std::ifstream>(Lexeme);
    }

    . {
        printf("Missing file name after 'include'.");
        exit(-1);
    }
}
```

As soon as a filename is matched the previous mode is popped from the mode stack, and then the analyzer state is packed into a memento using the function `include_push`. The memento will provide an object of class `ifstream`, so it has to be told via the template parameter. The default match of this mode simply tells that no file name has been found. When the included file hits the end-of-file, one needs to return to the including file. This is done using the `include_pop` function. And, here comes the `BASE` mode that all modes implement:

```
mode BASE {
    <<EOF>> {
        if( self.include_pop() ) return;
        /* Send an empty lexeme to overwrite token content. */
        self_send1(QUEX_TKN_TERMINATION, LexemeNull);
        return;
    }

    [ \t\r\n]+  { }
}
```

The `include_pop()` function returns `true` if there was actually a file from which one had to return. It returns `false`, if not. In the latter case we reached the 'end-of-file' of the root file. So, the lexical analyzis is over and the `TERMINATION` token can be sent. This is all to say about the framework. We can now step on to defining the actions for packing an unpacking mementos. First, let the memento be extended to carry a stream handle:

```
memento {
    std::ifstream*  included_sh;
}
```

When the analyzer state is frozen and a new input stream is initialized, the `memento_pack` section is executed. It must provide an input handle in the variable `input_handle` and receives the name of the input as a `QUEX_TYPE_CHARACTER` string. The memento packer takes responsibility over the memory management of the stream handle, so it stores it in `included_sh`.

```
memento_pack {
    *input_handle = new std::ifstream((const char*)InputName, std::ios::binary);

    if( (*input_handle)->fail() ) {
        delete *input_handle;
        return 0x0;
    }
    memento->included_sh = *input_handle;
}
```

Note: If `*input_handle` points to something different from `0x0` this means that the `include_push` has already provided the input handle and it must not be made available by the `memento_pack` section.

At the time that the analyzer state is restored, the input stream must be closed and the stream object must be deleted. This happens in the `memento_unpack` section

```
memento_unpack {
    memento->included_sh->close();
    delete (memento->included_sh);
}
```

The closing of the stream needs to happen in the `memento_unpack` section. The analyzer cannot do it on its own for a very simple reason: not every input handle provides a 'close' functionality. Symetrically to the `memento_pack` section where the input handle is created, it is deleted in the `memento_unpack` section, when the inclusion is terminated and the analyzer state is restored.

6.7.3 Infinite Recursion Protection

When a file is included, this happens from the beginning of the file. But, what happens if a file includes itself? The answer is that the lexical analyzer keeps including this file over and over again, i.e. in hangs in an *infinite recursion*. If there is no terminating condition specified by the implementer, then at some point in time the system on which it executes runs out of resources and terminates after its fancy.

The case that a file includes itself is easily detectable. But the same mentioned scenario evolves if some file in the include chain is included twice, e.g. file A includes B which includes C which includes D which includes E which includes F which includes G which includes C. In this case the analyzer would loop over the files C, D, E, F, G over and over again.

Quex does not make any assumptions about what is actually included. It may be a file in the file system accessed by a `FILE` pointer or `ifstream` object, or it may be a stream coming from a specific port. Nevertheless, the solution to the above problem is fairly simple: Detect whether the current thing to be included is in the chain that includes it. This can be done by iteration over the memento chain. The member `stream_id` in figure *List of Mementos implementing the Include Stack.* is a placeholder for something that identifies an input stream. For example let it be the name of the included file. Then, the memento class extension must contain its definition

```
memento {
    ...
    std::string   file_name; // ... is more intuitive than 'stream_id'
    ...
}
```

The lexical analyzer needs to contain the filename of the root file, thus the analyzer's class body must be extended.

```
body {
    ...
    std::string   file_name;
    ...
}
```

Then, at each inclusion it must be iterated over all including files, i.e. the preceeding mementos. The first memento, i.e. the root file has a parent pointer of `0x0` which provides the loop termination condition.

```
...
MyLexer  my_lexer("example.txt");
```

```
my_lexer.file_name = "example.txt";
...

memento_pack {
    /* Detect infinite recursion, i.e. re-occurence of file name           */
    for(MyLexerMemento* iterator = my_analyzer._memento_parent;
        iterator != 0x0; iterator = iterator->parent ) {
        /* Check wether the current file name already appears in the chain */
        if( iterator->file_name == (const char*)InputName ) {
            REACTION_ON_INFINITE_RECURSION(Filename);
        }
    }
    /* Open the file and include */
    FILE*  fh = open((const char*)InputName, "rb");
    if( fh == NULL ) MY_REACTION_ON_FILE_NOT_FOUNT(Filename);

    /* Set the filename, so that it is available, in case that further
     * inclusion is triggered.                                             */

    memento->file_name = self.file_name;
    self.file_name     = (const char*)InputName;
}
```

All that remains is to reset the filename on return from the included file. Here is the correspondent `memento_unpack` section:

```
memento_unpack {
    ...
    self.file_name = memento->file_name;
    ...
}
```

Note: Do not be surprised if the `memento_unpack` handler is called upon deletion of the lexical analyzer or upon reset. This is required in order to give the user a chance to clean up his memory propperly.

6.7.4 Caveats

Section *HOWTO* explained a safe and sound way to do the inclusion of other files. It does so by handling the inclusion from inside the pattern actions. This has the advantage that, independent of the token policy, the token stream looks as if the tokens appear in one single file.

The alternative method is to handle the inclusion from outside the analyzer, i.e. as a reaction to the return value of the `.receive(...)` functions. The 'trick' is to check for a token sequence consisting of the token trigger and and an input stream name. This method, together, with a queue token policy requires some precaution to be taken. The 'outer' code fragment to handle inclusion looks like

```
do {
    qlex.receive(&Token);

    if( Token.type_id() == QUEX_TKN_INCLUDE ) {
         qlex.receive(&Token);
         if( Token.type_id() != QUEX_TKN_FILE_NAME ) break;
         qlex.include_push((const char*)Token.get_text().c_str());
    }

} while( Token.type_id() != QUEX_TKN_TERMINATION );
```

The important thing to keep in mind is:

Warning: The state of the lexical analyzer corresponds to the last token in the token queue! The `.receive()` functions only take one token from the queue which is not necessarily the last one.

In particular, the token queue might already be filled with many tokens after the input name token. If this is desired, quex provides functions to save away the token queue remainder and restore it. They are discussed later on in this chapter. The problem with the remaining token queue, however, can be avoided if it is ensured that the `FILE_NAME` token comes at the end of a token sequence. This can be done similarly to what was shown in section *HOWTO*. The analyzer needs to transit into a dedicated mode for reading the file name. On the event of matching the filename, the lexical analyzer needs to explicitly `return`.

An explicit `return` stops the analyzis and the `.receive(...)` functions return only tokens from the stack until the stack is empty. Since the last token on the stack is the `FILE_NAME` token, it is safe to assume that the token stack is empty when the file name comes in. Thus, no token queue needs to be stored away.

If, for some reason, this solution is not practical, then the remainder of the token queue needs to be stored away on inclusion, and re-inserted after the inclusion finished. Quex supports such a management and provides two functions that store the token queue in a back-memory and restore it. To use them, the memento needs to contain a `QuexTokenQueueRemainder` member, i.e.

```
memento {
    ...
    QuexTokenQueueRemainder  token_list;
    ...
}
```

`void QuexTokenQueueRemainder_save(...);`

This function allocates a chunk of memory and stores the remaining tokens of a token queue in it. The remaining tokens of the token queue are detached from their content. Any reference to related object exists only inside the saved away chunk. The remaining tokens are initialized with the placement new operator, so that the queue can be deleted as usual.

Arguments:

`QuexTokenQueueRemainder* me`
Pointer to the `QuexTokenQueueRemainder` struct inside the memento.

`QuexTokenQueue* token_queue`
Token queue to be saved away. This should be `self._token_queue` which is the token queue of the analyzer.

`void QuexTokenQueueRemainder_restore(...);`
Restores the content of a token queue remainder into a token queue. The virtual destructor is called for all overwritten tokens in the token queue. The tokens copied in from the remainder are copied via plain memory copy. The place where the remainder is stored is plain memory and, thus, is not subject to destructor and constructor calls. The references to the related objects now resist only in the restored tokens.

Arguments: Same as for `QuexTokenQueueRemainder_save`.

The two functions for saving and restoring a token queue remainder are designed for one sole purpose: Handling include mechanisms. This means, in particular, that the function `QuexTokenQueueRemainder_restore` is to be called *only* when the analyzer state is restored from a memento. This happens at the end of file of an included file. It is essential that the analyzer returns at this point, i.e. the `<<EOF>>` action ends with a `return` statement. Then, when the user detects the `END_OF_FILE` token, it is safe to assume that the token queue is empty. The restore function only works on empty token queues and throws an exception if it is called in a different condition.

The handling from outside the analyzer never brings an advantage in terms of computational speed or memory consumption with respect to the solution presented in *HOWTO*. The only scenario where the 'outside' solution might make sense is when the inclusion is to be handled by the parser. Since the straightforward solution is trivial, the demo/005 directory contains an example of the 'outer' solution. The code displayed there is a good starting point for this dangerous path.

6.8 Indentation Based Blocks

With the rise of the Python programming language, the use of indentation as the block delimiter has become popular. Classical programming languages such as C, Java, and Pascal, rely on brackets, e.g. `{` and `}` for opening and opening blocks. Indentation based languages rely on the indentation space of a code fragment. For example, in the C-code statements list:

```
while( i < L )
{
    if( check_this(i) )
    {
        do_something();
    }
}
print("done")
```

blocks are delimited { and }. An equivalent statement list in an indentation based language, such as Python looks like the following:

```
while i < L:
    if check_this(i):
        do_something()
print("done")
```

The code in the second example, obviously looks much more dense and contains lesser visual noise. For readability, code is best indented according to the block it belongs to. In this sense the brackets are redundant–if one is able to detect the indentation. Quex generated engines can.

For the parser to group statements into blocks, it requires that tokens are sent that indicate the opening and closing of a block. When relying on explicit delimiters, such as brackets, then this does not require any additional effort. For clarity, let 'indentation' be defined as follows:

Note: *Indentation* is considered the amount of whitespace between the beginning of a line and the first non-whitespace character in a line.

For indentation based languages the lexical analyzer has some work to do behind the scenes. It must count indentation, detect wether blocks are opening, remain the same, or are closing. When closing blocks, is possible that with one step multiple tokens may be sent. In the example above, after the function call `do_something()`, two scopes close the `if` and the `while` block. Quex generated code takes care of this.

If indentation counting is enabled, Quex generated engines send the following tokens:

- `INDENT` if an indentation block opens.
- `DEDENT` if an indentation block closes.
- `NODENT` if the indentation remains the same.

An indentation counting framework is implemented for a mode as soon as a `indentation` option is specified–even an empty option will do:

```
.. code-block:: cpp

   mode X :
       <indentation: /* all default */>
   {
      ...
   }
```

There are different philosophies around indentation based parsing with respect to spaces, tabulators, and newline suppressors. Even the definition of the newline pattern may be subject to discussion (e.g. 0xA or 0xD, 0xA, or both). Inside the `indentation` option the character of those can be specified, e.g.

```
<indentation:
   [ ]        => space 1;
   [\t]       => grid 4;
   (\r\n)|\n  => newline;
   \\[ \t]*   => suppressor;
>
```

specifies that a normal space is counted as 1 `space` and tabulators span a `grid` of width 4. Both, the Unix (`\\n`) and the DOS Version of newline (`\\r\\n`) are accepted as `newline`. A newline suppressor is defined as a backslash. Since whitespace between the backslash and newline is hardly identified, possible whitespace is packed into the definition of the suppressor. The above setup is a *safe* setup to work on many environments and it helps to avoid confusion. It is indeed the default setup which is in place, if nothing is specified. The following section explains in detail the setting of the above parameters.

6.9 Indentation Parameters

The syntax of parameter settings in the `indentation` option follows the scheme:

```
pattern '=>' parameter-name [argument] ';'
```

The allowed parameter names are `space`, `grid`, `bad`, `newline`, and `suppressor`. The first three parameter names allow only character sets as pattern. For `newline` and `suppressor` any regular expression can by given. The following list explains the different parameters.

`space [number|variable]`

This defines what characters are accepted as a 'space'. A space is something that always increments the column counter by a distinct number. The argument following `space` can either be a number or a variable name is specified, it will become a member of the lexical analyzer with the type 'size_t' as defined in 'stddef.h'. Then the increment value can be changed at runtime by setting the member variable.

Multiple definitions of `space` are possible in order to define different space counts. Note, that in Unicode the following code points exist to represent different forms of whitespace:

- `0x0020`: Normal Space.
- `0x00A0`: Normal space, no line break allowed after it.
- `0x1680`: Ogham (Irish) Space Mark.

- **`0x2002`: Space of the width of the letter 'n': 'En Space'.** This is half the size of an Em Space.
- `0x2003`: Space of the widht of the letter 'm': 'Em Space'.
- `0x2004`: 1/3 of the width of an 'm': 'Three-Per-Em Space'.
- `0x2005`: 1/4 of the width of an 'm': 'Four-Per-Em Space'.
- `0x2006`: 1/6 of the width of an 'm': 'Six-Per-Em Space'.
- `0x2007`: Size of a digit (in fonts with fixed digit size).
- `0x2008`: Punctuation Space that follows a Comma.
- `0x2009`: 1/5 of the width of an 'm': 'Thin Space'.
- `0x200A`: Something thinner than 0x2009: 'Hair Space'.
- `0x200B`: Zero-Width Space.
- `0x202F`: Narrow No-Break Space, no line break allowed after it.
- `0x205F`: Medium Mathematical Space.
- `0x2060`: Word Joiner (similar to 0x200B)
- `0x2422`: Blank Symbol ().
- `0x2423`: Open Box Symbol ().
- **`0x3000`: Ideographic Space, size of a Chinese, Japanese,** or Korean letter.

Provided that the editor supports it the 'm' based spaces could for example be parameterized as:

```
<indentation:
    [\X2003] => space 60; /* Em Space           */
    [\X2002] => space 30; /* En Space           */
    [\X2004] => space 20; /* Three-Per-Em Space */
    [\X2005] => space 15; /* Four-Per-Em Space  */
    [\X2009] => space 12; /* Thin Space         */
    [\X2006] => space 10; /* Six-Per-Em Space   */
>
```

`grid [number|variable]`

Characters associated with a 'grid' set the column number according to a grid of a certain width. Tabulators are modelled by grids. For example, if the grid width is four and the current indentation count is 5, then a tabulator will set the column count to 8, because 8 is the closest grid value ahead.

As with 'space', a run-time modification of the grid value is possible by specifying a variable name instead of a number. For example,

```
[\t]  => grid  tabulator_width;
```

results in a member variable `tabulator_width` inside the analyzer that can be changed at run-time, e.g.

```
...
MyLexer   qlex(...);
...
if( file_format == MSVC ) qlex.tabulator_width = 8;
else                      qlex.tabulator_width = 4;
...
```

bad

By this specifier characters can be defined which are explicitly 'bad'. There are very rational arguments for 'spaces are bad' and so there are arguments for 'tabulators are bad'. The latter philosophy can be expressed by

```
[\t]  =>  bad;
```

newline

Indentation count is triggered by 'newline'. By this specifier it can be determined what character or character sequence triggers the indentation count. For example,

```
(\r\n)|\n  => newline;
```

matches newlines under DOS (0x0D, 0x0A) and under Unix (0x0A). All specifiers before only accept character sets as input. Clearly, the newline specifier accepts a full regular expression.

The newline pattern will be used to trigger the indentation counting. Actually, the newline pattern is automatically extended to the pattern:

```
newline [[ ispace ]* newline]*
```

and inserted into state machine. Here, `ispace` is any kind of indentation counter mentioned in `space` or `grid`. By means of this construction empty lines are eaten silently. Thus, it is avoided that empty lines cause a DEDENT or NODENT events.

suppressor

The newline event can be suppressed by a subsequent suppressor. When it is suppressed the subsequent line is not going to be subject to indentation count. Famous suppressors are the backslash, as in Python, C, and Makefiles, or the underline '_' as in some Basic dialects. For example, the backslash in

```
if    db.out_of_date() \
   or db.disconnected():
       ...
```

prevents the python interpreter to consider indentation before the 'or' which is now grouped into the if-condition.

Many times interpreters are sensitive to whitespace that follows these. Quex allows to be less sensitive by defining the suppressor as a regular expression, e.g.

```
\\[ \t]*    => suppressor;
```

eats any amount of non-newline whitespace after the suppressor '\'.

When an indention option is specified, the generated lexical analyzer starts sending tokens carrying indentation information. As mentioned earlier, those are `QUEX_TKN_INDENT` if a new indentation block opens, `QUEX_TKN_DEDENT` if an indentation block closes, and `QUEX_TKN_NODENT` if the indentation is the same as in the previous line. Note, that the newline character is eaten by the indentation counter. If it is a statement delimiter, then it might make sense to define in the `token` section something like:

```
token {
    ...
    NODENT = STATEMENT_END;
    ...
}
```

which ensures that the token id of `NODENT` is the same as the id for `STATEMENT_END` and no special treatment is required. If more then one token is to be sent on indentation events, or if some sophisticated customized handling is required the indentation events can be specified, as shown in the next section.

6.10 Customized Indentation Event Handlers

By default, a quex generated engine sends tokens on the event of indentation and aborts on the event of a bad character or indentation error. If this behavior is not enough, the correspondent actions may be customized by means of event handlers, as they are:

- `on_indent` on the event that an opening indentation occurs.

  For example in the code fragment

  ```
  while 1 + 1 == 2:
      print "Hello"
      print "World"
  ```

  The `print` commands are more indented than the `while` key word. This indicates that the prints are in a nested block. On the event of the first indented `print` key word the `on_indent` handler is called. The user might want to send an `INDENT` token to indicate the opening of a block. For example:

```
on_indent {
    self_send(QUEX_TKN_INDENT);
}
```

- on_dedent and on_n_dedent on the event that one ore mode indentation blocks are closed.

  Note, that by means of a single line multiple indentation blocks may be closed. For example in

  ```
  while time() < 3600:
      if time() % 10:
          print "Tick"
  print "End"
  ```

  The line containing `print "End"` closes the `if` block and the `while` block. It is appropriate that the lexical analyzer sends two `DEDENT` tokens. There are basically two ways to do this. Either by sending a `DEDENT` token each time an indentation block closes, or by counting the indentation blocks which close and the start the sending of the `DEDENT` tokens. Accordingly there are two de-dentation handlers.

  on_dedent
  Argument: `First`

  The `on_dedent` handler is called repeatedly for each closing indentation. Each time the handler is called one `DEDENT` token should be sent. If there are things to be done only once for whole a de-dentation sequence, then the flag `First` can be used. It is `true` for the first de-dedentation event of a sequence and `false` for any other. A typical usage would be

  ```
  on_dedent {
      ...
      if( First ) self_send(QUEX_TKN_NEWLINE);
      self_send(QUEX_TKN_DEDENT);
      ...
  }
  ```

  on_n_dedent
  Argument: `ClosedN`

  The `on_n_dedent` is called after the number of closing indentations has been counted and it receives the argument `ClosedN`. This argument indicates the number of closed indentations. A typical handler will then call `self_send_n(...)` somewhere down the lines as shown below.

  ```
  on_n_dedent {
      ...
      /* provided that token repetition support is enabled! */
  ```

```
        self_send_n(ClosedN, QUEX_TKN_DEDENT);
        ...
    }
```

It is advisable to activate token repetition support :ref:`, since otherwise the token queue might be flooded with ``DEDENT` tokens.

- **`on_nodent` on the event that the current line has the same indentation** as the previous.
- **`on_indentation_error` on the event that a lesser indentation occured**

  which does not fit the indentation borders of previous indentation blocks.

  `Indentation`
  The indentation that has occured.

  `IndentationStackSize`
  The number of currently open indentation blocks.

  `IndentationStack(i)`
  Delivers the indentation number 'i' from the current indentation blocks.

  `IndentationUpper`
  Delivers the smallest indentation level that is greater than the current.

  `IndentationLower`
  Delivers the greatest indentation level that is smaller than the current.

  `ClosedN`
  Number of closed indentation levels.

- **`on_indentation_bad` on the event that a *bad* indentation character**

  occured. The argument to this handler is

  `BadCharacter`
  A constant that contains the bad indentation character. It is of type `QUEX_TYPE_CHARACTER`.

  Quex does not forbid the definition of a pattern that contains the bad character. The contrary, it is essential to define such a pattern in case that only a warning is intended and not a break up of the lexical analyzis. A skipper will also do. For example,

```
mode X : <indentation: [\t] => bad;>
         <skip: [\t]>
{
     ...
     on_indentation_bad {
         std::cout << "Warning: Bad indentation character!\n";
     }
```

```
        ...
    }
```

is a reasonable setup in a lexical analyzer that forbids tabulators in indentation. Alternatively, a 'bad character token' might be defined and sent.

The following code fragment shows an example application that implements the default behavior.

Note: The current indentation level can be accessed via the macro `self_indentation()`. For this, it holds the same as for line and column counting: The value is the current indentation level and not the level of indentation of the current token. For token policy `token-queue` this value might be stored inside the token itself.

6.11 Remarks on Hand-Written Indentation Management

It is not trivial to express indentation in terms of pattern action pairs based solely 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 back 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:

Note: A design without indentation events, forces the pattern actions to know about each other. Otherwise, they might not function propperly together! In an environment of many different modes which are additionally related by inheritance, it is potentially difficult to guarantee that all pattern actions avoid interferences with some overal concepts.

Similarly, catching indentation with pre-condition newline plus whitespace, i.e. `^[ \t]*` is fragile, in the 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 quex's ability to detect indentation blocks ends up in a much clearer and safer design.

6.11.1 Caveat

If a pattern contains more than one newline then only the indentation event concerning the last newline is triggered! Imagine a pattern such as in the following example:

```
mode INDENTICUS {
    " "*"hello"[\n]+" "*"world"[\n]+" "*"how are you?" => TKN_STRANGE;
}
```

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. If the line after the concatinated line does not end with a backslash the event handler is automatically active and indentation handling is in place. Lets turn this into a warning.

Warning: Avoid having multiple non-whitespace sub patterns (such as keywords or identifiers) concatinated by newline-containing sub-patterns in *one single pattern*. Otherwise only the last transition from whitespace to non-whitespace inside the pattern triggers an indentation event.

The author of this text hopes that this caveat is only of theoretical interest. It is hard to imagine a case where such a construct would actually make sense. In any case, before implementing an indentation based analyzer it is advisable to have a look at the demo/002 directory for a functioning example.

6.12 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. The `body` section allows to add new members to the analyzer class which helps to customize functionality. It is, however, conceivable that the generated lexical analyzer needs to remain as it is to ensure the analyzis being stable. At the same time, derivates of such a 'frozen' analyzer might be needed for testing or research purposes. Adding new features to frozen code is best done using C++ inheritance, i.e. one derives his own class from the generated lexer class.

The name and definition of the derived 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 DerivedLexer
   : public quex::GeneratedLexer {
     small_lexer(const std::string& filename,
                 std::ostream*      output_stream = 0);

     // ...
};
```

The derived class has to call a base class' constructor from inside its constructor initialization list in order to properly initialize the lexical analyser object.

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

The destructor must be virtual in order to ensure that the base class' destruction happens propperly.

6.13 Multiple Lexical Analyzers

This section discusses how to use multiple lexical analyzers generated by quex in a single application without causing clashes. Since version 0.47.1 this task has become very easy. The only requirement from the user's prespective is that the lexical analyzers and the token classes need to live in separate namespaces. The command line option `-o`, `--engine`, `--analyzer-class` allow to specify a name of the analyzer class as well as its namepsace by means of preceeding `::` sequences, e.g.:

```
> quex ... -o world::germany::baden::Fritzle ...
```

defines that the analyzer class is called `Fritzle` and is located in namespace `baden` which is inside `germany` which is inside `world`. If no namespace is specified, the class is located in namespace `quex`. Similarly, the token class can be specified by means of `--token-class`, `--tc`:

```
> quex ... -o castle::room::jewel_case::MyToken ...
```

specifies that the generated token class `MyToken` is inside the namespace `castle`, which contains `room`, which contains `jewel_case`. If no namespace is specified, then the token class is placed in the namespace of the lexical analyzer class. In any case, when migrating from a single to multiple lexical analyzers, then your code must reflect this. For example, a code fragment as

```
int main(int argc, char** argv) {
     using namespace quex;
     ALexer     qlex("example.txt");
     ...
}
```

should be adapted to something similar to

```
int main(int argc, char** argv) {
     namespace1::ALexer     a_lexer("example.a");
     namespace2::BLexer     b_lexer("example.b");
     ...
}
```

In conclusion, the only thing that has to be made sure is that the generated analyzers live in separate namespaces. All related entities of the analyzers are then propperly distinguished for compiling and linking. Note, that directory `demo/012` contains an example implementation of an application containing three different analyzers with different numbers of bytes per character and different decoders (ICU, IConv and engine codec).

6.14 Reset and Change of Input Source

In many practical applications the source of input is changed in the form of included files that are 'pasted' at the point where the according statement occurs. The handling of this scenerio was mentioned in section *Include Stack*. In this section a more liberal approach is described where the input source can be switched without keeping track of previous content. This is done by the `reset()` function group which has the following signatures

```
template <class InputHandleT> void
reset(InputHandleT*   input_handle,
      const char*     CharacterEncodingName = 0x0);

void
reset_buffer(QUEX_TYPE_ANALYZER*   me,
             QUEX_TYPE_CHARACTER* BufferMemoryBegin,
             size_t               BufferMemorySize,
             QUEX_TYPE_CHARACTER* BufferEndOfContentP,
             const char*          CharacterEncodingName = 0x0);
```

`reset(...)`

The first argument is an input handle of the user's like, i.e. either a `FILE*`, `istream*`, or `wistream*` pointer, or a pointer to a derived class' object. The second argument is character encoding name which is used for the input. If *the same* converter is used, the `CharacterEncodingName` must be set to zero. If the analyzer works only on plain memory, the following reset function may be used

```
void  reset(const char* CharacterEncodingName = 0x0);
```

which allows to specify a character encoding name, but the input stream is not specified.

The reset function clears and resets all components of the lexical analyzer as they are:

1. The core analyzer including buffer management and buffer fill management. The buffer memory remains intact, but is cleaned. When `reset(...)` is called an initial load to fill the buffer is accomplished. Thus, the input handle must be placed at the point where the lexical analyzer is supposed to start.
2. The current mode is reset to the start mode and the mode stack is cleared.
3. The include stack is cleared and all mementos are deleted.
4. The accumulator is cleared.
5. The line and column number counter is reset.
6. All entries of the post categorizer dictionary are deleted.

Note: The byte order reversion flag is *not* set to `false`. It remains as it was before the reset.

---

reset_buffer(...)

The reset buffer function allows to (re-)start lexical analyzis on a user provided buffer. It does basically the same as the `reset` function. However, it distinguishes two cases as indicated by its return value:

== 0x0
: If `reset_buffer` returns 0x0, then there was no user provided buffer and nothing further is to be done with respect to memory management.

!= 0x0
: If the return value is non-zero, then there was a buffer memory which as previouly been provided by the user. The return value points to the beginning of the old memory chunk. The engine itselft does not de-allocate the memory. It is the task of the user.

If the function `reset_buffer` is called with zero as the first argument `BufferMemoryBegin`, then no memory is initialized. However, the current memory pointer is returned if it was provided by the user. Otherwise, as described above, zero is returned. This is helpful at the end of the analyzis in order to de-allocate any memory that is still contained inside the analyzer. An application of `reset_buffer` can be observed in file `re-point.cpp` and `re-point.c` in the `010` subdirectory of the demos.

6.15 The Accumulator

The accumulator is a member of the lexical analyzer that allows stock strings to communicate between pattern-actions[#f1]_. In the practical example in section [sec-practical-intro] the string contained in string delimiter marks was accumulated until the on_exit handler was activated, i.e. the‘ STRING_READER‘ mode is left. Inside the handler, the string is flushed into a token with a specific id TKN_STRING. The accumulator provides the following functions:

```
void   self_accumulator_add(const QUEX_TYPE_CHARACTER* Begin, const QUEX_TYPE_CHARA
void   self_accumulator_add_chararacter(const QUEX_TYPE_CHARACTER);
void   self_accumulator_flush(const token::id_type TokenID);
void   self_accumulator_clear();
```

The add-functions add a string or a character to the accumulated string. `Begin` must point to the first character of the string, and `End` must point right after the last character of the string. `Lexeme` can be passed as `Begin`, and `LexemeEnd` can be passed as `End`. 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.

Warning:
If a dynamic length encoding is used (such as `--codec utf8` or `--codec utf16`),
then one *must not* use the function

```
void   self_accumulator_add_chararacter(const QUEX_TYPE_CHARACTER);
```

Even if one really wants to add only a single character. since it expects a fixed size character object. Instead, please use

```
void   self_accumulator_add(const QUEX_TYPE_CHARACTER* Begin,
                            const QUEX_TYPE_CHARACTER* End);
```

even if the added element is only one letter.

6.16 The Post Categorizer

A quex generated analyzer may contain an entity to do post-categorization. The post- categorizer is activated via the command line option:

```
--post-categorizer
```

This feature allows the categorization of a lexeme after it has matched a pattern. It performs the mapping:

```
lexeme ---> token identifier
```

This comes handy if the meaning of lexemes change at run time of the analysis. For example, an interpreter may allow function names, operator names and keywords to be defined during analysis and requires from the lexical analyzer to return a token FUNCTION_NAME, OPERATOR_XY, or KEYWORD when such a lexeme occurs. However assume that those names may follow the same pattern as identifiers, so one needs to post-categorize the pattern. The caller of the analyzer may somewhere enter the meaning of a lexeme into the post- categorizer using the function enter(...) where the first argument is the name of the lexeme and the second argument is the token id that is to be sent as soon as the lexeme matches.

```
...
my_lexer.post_categorizer.enter(Name, QUEX_TKN_FUNCTION_NAME);
...
if( strcmp(setup.language, "german") == 0 ) {
    my_lexer.post_categorizer.enter("und",   QUEX_TKN_OPERATOR_AND);
    my_lexer.post_categorizer.enter("oder",  QUEX_TKN_OPERATOR_OR);
    my_lexer.post_categorizer.enter("nicht", QUEX_TKN_OPERATOR_NOT);
}
...
my_lexer.post_categorizer.enter(Name, QUEX_TKN_FUNCTION_NAME);
...
```

The following is a quex code fragment that uses the post categorizer relying on the function get_token_id(...)

```
mode POST_CAT {
    ...
    [a-z]+ {
        QUEX_TYPE_TOKEN_ID* token_id = self.post_categorizer.get_token_id(Lexeme);
        if( token_id != QUEX_TKN_UNINITIALIZED ) {
            self_send1(QUEX_TKN_IDENTIFIER, Lexeme);
        }
        else {
            self_send1(token_id, Lexeme);
        }
    }
    ...
}
```

It sends the IDENTIFIER token as long as the post-categorization on default. This is determined by a return vale being QUEX_TKN_UNINITIALIZED. If the post-categorizer has found an entry that fits, the appropriate token-id is send.

CHAPTER

SEVEN

TUNING

This chapter is dedicated to the subject of how to improve a lexical analyzer in terms of computational speed and memory footprint. It is not intended as a 'silver bullet' that guides ultimately to the optimal configuration. However, it introduces some basic thoughts which are crucial for optimization. This chapter starts by the introduction of two mechanisms that reduce the code size of the lexical analyzer: template and path compression. Then the influence of the token queue is discussed. Finally, the involvement of converters via engine based codecs is investigated. Were possible the findings are supported by data accessed by some sample applications which are available in the quex distribution package directory 'demo/tuning/'.

Before any measurements can be made the following points need to be assumed:

.# Asserts are disabled, i.e the compile option `QUEX_OPTION_ASSERTS_DISABLED` is in place.

.# Avoid printing through system calls, i.e avoid `fprintf`, "cout << " and their likes. If such a system call is performed for each token, the analyzer's overhead is neglectable and the results only measure the performance of the input/output stream library and its implementation on the operating system.

.# The buffer size must be set to something reasonable. Define `QUEX_SETTING_BUFFER_SIZE=X` where `X` should be something in the range of at least 32Kb.

.# Avoid any compiler flag that produces additional code such as `-ggdb` for debugging, `--coverage` for test coverage, or `--fprofile-arcs` for profiling.

A tuning setup is mostly the contrary of a debug scenario. For debugging all asserts should be active, print-outs prosper, and the buffer size is as small as possible to check out reload scenarios. Using a a debug setup for tuning, though, safely drives the development into nonsensical directions.

7.1 Template Compression

Quex introduces a powerful feature to reduce the code size of a lexical analyzer of a lexical analyzer: Template Compression. Template compression combines multiple states into a single state that changes slightly dependent on the particular state that is represents. It can be activated with the command line option:

```
> quex ... --template-compression
```

The aggressivity of the compression can be controlled by the template compresison 'min-gain' to be set by the command line argument:

```
> quex ... --template-compression-min-gain [number]
```

It represents the minimum of estimated bytes that could be spared before two states are considered for combination into a template state. If only states with the same entry and drop out section should be considered then the following option can be specified:

```
> quex ... --template-compression-uniform
```

Especially unicode lexical analyzers large transition maps can profit tremendously from template compression. If your compiler supports computed gotos try to set `QUEX_OPTION_COMPUTED_GOTOS` on the compilers' command line, e.g. with GNU C++:

```
> g++ -DQUEX_OPTION_COMPUTED_GOTOS ... MyLexer.cpp -c -o MyLexer.o
```

7.1.1 Principle of Template Compression

This is displayed in the figures *Three states to be compressed by Template compression.* and *Template state representing states 0, 1, and 2 from figure fig-template-compression-before.*. The first figure displays three states which have a somehow similar transition map, that is they trigger on similar states to similar target states. A so called 'template state' must be able to detect all character ranges involved. An additional matrix helps to find the particular target state for the represented state.

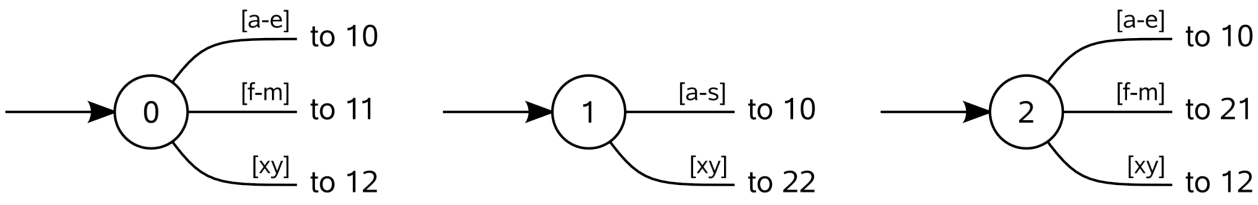


Figure 7.1: Three states to be compressed by Template compression.

Figure *Template state representing states 0, 1, and 2 from figure fig-template-compression-before.* displays a template state that shall represent state 0, 1, and 2 from figure *Three states to be com-*

pressed by Template compression.. Every state is replaced by a small stub that defines a 'state key'. The role of the state key is to define the behavior of the template state. In particular, it defines the target states that belong to certain trigger sets. State 0, 1, and 2 differ with respect to the character ranges `[f-m]` and `[o-s]`. For these ranges the transitions differ. The target states for these ranges are stored in the arrays `X0` and `X1`. `X0[i]` represents the target state for `[f-m]` if the template needs to represent the state `i`. Analogously, `X1[i]` helps to mimik the transitions for `[o-s]`.

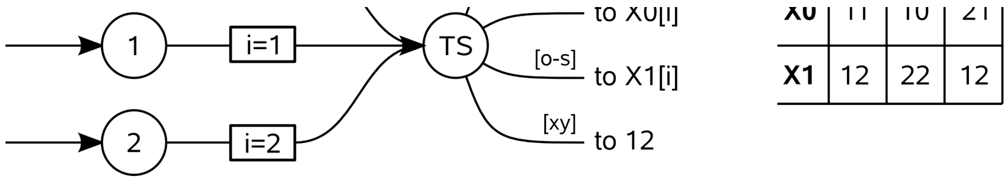


Figure 7.2: Template state representing states 0, 1, and 2 from figure *Three states to be compressed by Template compression.*.

Template compression can combine any set of states but its efficiency depends on the similarity. For this reason the template compression coefficient (see option `--template-compression-coefficient`) has been introduced that defines how aggressively states shall be combined. That means how much lack of similarity can is tolerated and states can still be combined into sets.

7.2 Path Compression

Quex introduces another way of compression that profits from the sequential order of similar states. It identifies paths of single character sequences in the state machine. The command line argument:

```
--path-compression
```

activates the analyzis and compression. With this compression all states are considered to be combined into a path. As a result some special handling is implemented to distinguish the particularities of each state. If only uniform states shall be considered, the command line flag:

```
--path-compression-uniform
```

may be provided. Then the overhead of particularities is avoided, but less states may be combined. Path compression requires a path-end character. By default it is set to the character code 0x16 (38 decimal, 'SYNCHRONOUS IDLE') assuming that this never occurs in the data stream. If this character occurs in the stream to be analyzed, then the path termination character must be defined explicitly by:

```
--path-termination [number]
```

Where the specified number must be different from the buffer limit code.

It applies if a sequence of single characters can be identified that guide along a path of states with matching transition maps. This requirement seems very rigid and thus the chance of hitting a state machine that contains such states may appear low. However, in practical applications this exactly the case where keywords, such as `for`, `while`, etc. intersect with the pattern of identifiers, e.g. `[a-z]+`. In other words, languages with many keywords may profit from this approach.

Instead of implementing for each state of the path a full state, only one state is implemented a so called 'pathwalker'. A pathwalker consists of the transition map which is common to all states of the path and the path itself.

As with template compression using the computed feature of your compiler might improve performance and code size.

7.2.1 Principle of Path Compression

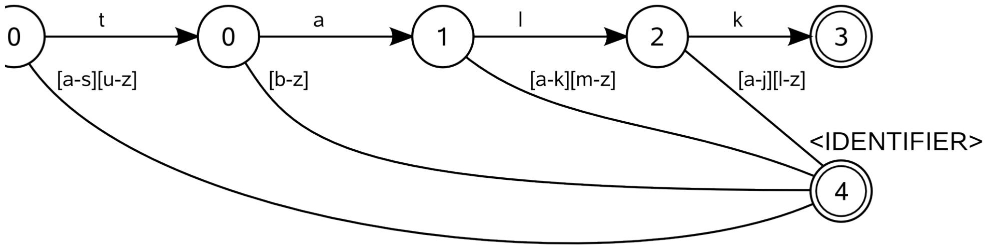


Figure 7.3: State sequence to be compressed by path compression.

7.3 Combining Multiple Compression Types

It is possible to combine multiple compression types simply by defining multiple of them on the command line. The result of applying path compression before or after template compression may be significantly different. The sequence of analysis corresponds to the sequence that the activating command line options appear, i.e.:

```
> quex ... --template-compression ... --path-compression ...
```

determines that template compression is performed before path compression. Uniform and non-uniform compression can be treated as separate procedures. Thus, it is possible to say for example:

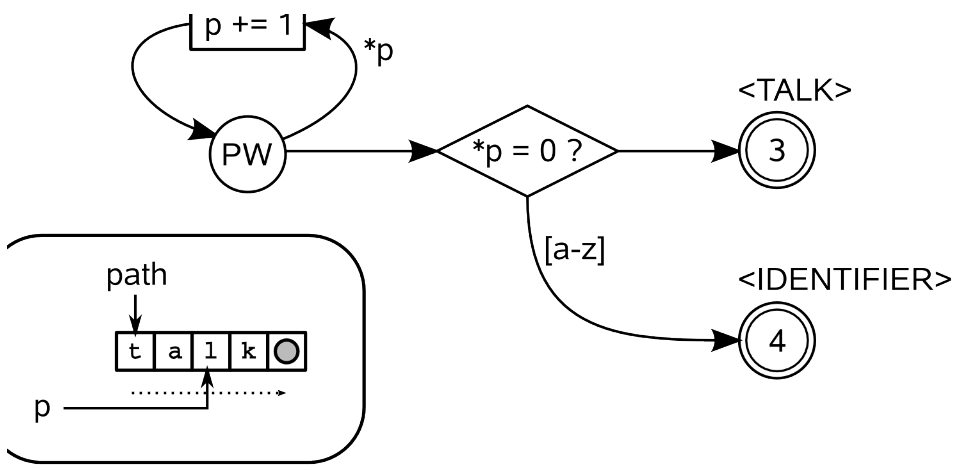


Figure 7.4: State sequence from figure *State sequence to be compressed by path compression.* implemented by pathwalker.

```
> quex ... --template-compression-uniform \
           --path-compression \
           --template-compression
```

which first does a template compression of very similar states, then a general path compression of the remainder. Then whatever remains of states is tried to be combined by aggressive template compression.

7.4 Token Queues

7.5 Memory Management

7.6 Additional Hints

It follows a list of general hints for performance tuning:

Note: The three most important things to consider when improving performance are the CPU's cache, the CPU's cache, and the CPU's cache. At the time of this writing (2010 C.E.) a cache-miss is by factors slower then a normal cache read. A fast program can be slowed down to 'snail speed' simply by excessive cache miss scenarios.

Practically, this means that the data that is access frequently is best kept close together, so that cache misses are less probable.

Note: The forth important thing about improving performance is to avoid frequent system calls. For example, allocate memory in a chunk and then cut from it when needed, instead of calling `new`, `delete`, `malloc` or `free` all the time. You might also consider to implement containers yourself instead of relying in STL or similar libraries, if this allows you to control memory placement.

Note: The fifth important thing is to use `memcpy` and `memmove` for copying of content. Especially for larger datasets. Few people can compete with the inside that is put into this functions. Simply compare a `memcpy` operation with a `for` loop doing the same thing. It is not seldom a factor of 40 between the two. Use `memmove` when source and destination may overlap.

[to be continued]

CHAPTER

EIGHT

INVOCATION AND COMPILATION

This chapter focusses on the invocation of quex and the specification of compile options. Previous sections already mentioned how quex can be executed to create a functioning analyzer. The subsequent sections are concerned with the control of more special features and fine-tuning.

8.1 Command Line Options

This section lists the command line options to control the behavior of the generated lexical analyzer. Strings following these options must be either without whitespaces or in quotes. Numbers are specified in C-like format as described in *sec-number-format*.

-i, -mode-files `file-list`
: `file-list` = list of files of the file containing mode definitions (see sections <<sec-practical-modes>>, <<sec-practical-pattern-action-pairs>>,

  and <<sec-formal-generated-class-mode-handling>>).

  DEFAULT = `<empty>`

-o`, '-engine', '-analyzer-class' name`
: `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 generated by quex. At the same time, the namespace of the analyzer class can be specified by means of a sequence separated by '::' specifiers, e.g.:

```
> quex ... --analyzer-class MySpace::MySubSpace::MySubSubSpace::Lexer
```

  specifies that the lexical analyzer class is `Lexer` and that it is located in the namespace `MySubSubSpace` which in turn is located `MySubSpace` which it located in `MySpace`.

  DEFAULT = `lexer`

-output-dir, -odir `directory`
: `directory` = name of the output directory where generated files are to be written. This

does more than merely copying the sources to another place in the file system. It also changes the include file references inside the code to refer to the specified `directory` as a base.

`-file-extension-scheme``, ``-fes`` `ext`

Specifies the filestem and extensions of the output files. The provided argument identifies the naming scheme.

``DEFAULT``

If the option is not provided, then the naming scheme depends on the `--language` command line option. That is:

C++

- No extension for header files that contain only declarations.
- `.i` for header files containing inline function implementation.
- `.cpp` for source files.

C

- `.h` for header files.
- `.c` for source files.

``++``

- `.h++` for header files.
- `.c++` for source files.

``pp``

- `.hpp` for header files.
- `.cpp` for source files.

``cc``

- `.hh` for header files.
- `.cc` for source files.

``xx``

- `.hxx` for header files.
- `.cxx` for source files.

For `C` there is currently no different naming scheme supported.

`-language` `name`

Defines the programming language of the output. `name` can be

- `C` for plain C code.
- `C++` for C++ code.
- `dot` for plotting information in graphviz format.

DEFAULT = `C++`

`-character-display` `['hex', 'utf8']`

Specifies how the character of the state transition are to be displayed when *–language dot* is used. If `hex` is specified then character will be displayed with the Unicode code point in hexadecimal notation. If `utf8` is specified the character will be displayed 'as is' in UTF8 notation.

DEFAULT="'utf8"'

`-debug``

If provided, then code fragments are created to activate the output of every pattern match. Then defining the macro *QUEX_OPTION_DEBUG_QUEX_PATTERN_MATCHES* activates those printouts in the standard error output. Note, that this options produces considerable code overhead.

DEFAULT = <*disabled*>

`-buffer-based``, `` `-bb` ``

Turns on buffer based analyzis. If this option is not set, buffer based analyzis can still be activated with the compile option `QUEX_OPTION_BUFFER_BASED_ANALYZIS`.

`-version-id` `name`

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

DEFAULT = *"0.0.0-pre-release"*

`-no-mode-transition-check``

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.

By default, the mode transition check is enabled.

`-single-mode-analyzer``, `` `-sma` ``

In case that there is only one mode, this flag can be used to inform quex that it is not intended to refer to the mode at all. In that case no instance of the mode is going to be implemented. This reduces memory consumption a little and may possibly increase performance slightly.

`-no-string-accumulator``, `` `-nsacc` ``

Turns the string accumulator option off. This disables the use of the string accumulator to accumulate lexemes. See class 'quex::Accumulator'.

By default, the string-accumulator is implemented.

`-no-include-stack``, `` `-nois` ``

Disables the support of include stacks where the state of the lexical analyzer can be saved and restored before diving into included files. Setting this flag may speed up a bit compile time

By default, the include stack handler is implemented.

`-post-categorizer``

Turns the post categorizer option on. This allows a 'secondary' mapping from lexemes to token ids based on their name. See class 'quex::PostCategorizer'.

-no-count-lines`
: Lets quex generate an analyzer without internal line counting.

-no-count-columns`
: Lets quex generate an analyzer without internal column counting.

If an independent source package is required that can be compiled without an installation of quex, the following option may be used

**-source-package`** `directory`
: Creates all source code that is required to compile the produced lexical analyzer. Only those packages are included which are actually required. Thus, when creating a source package the same command line 'as usual' must be used with the added *–source-package* option.

  The string that follows is the directory where the source package is to be located.

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

-derived-class`, ``-dc` name`

`name` = If specified, the name of the derived class that the user intends to provide

(see section <<sec-formal-derivation>>). 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. The namespace of the derived analyzer class is specified analgously to the specification for *–analyzer-class*, as mentioned above.

DEFAULT = <empty>

**-derived-class-file`** `filename`
: `filename` = If specified, the name of the file where the derived class is defined. This option only makes sense in the context of optioin –derived-class'.

  DEFAULT = <empty>

-token-prefix `name`
: `name` = Name prefix to prepend to the name given in the token-id files. For example, if a token section contains the name `COMPLEX` and the token-prefix is `TOKEN\_PRE_` then the token-id inside the code will be `TOKEN_PRE_COMPLEX`.

  The token prefix can contain namespace delimiters, i.e. `::`. In the brief token senders the namespace specifier can be left out.

  DEFAULT = `QUEX_TKN_`

-token-policy, -tp `[queue, single]`
: Determines the policy for passing tokens from the analyzer to the user.

  DEFAULT = `queue`

-token-memory-management-by-user, -tmmbu
Enables the token memory management by the user. This command line option is equivalent to the compile option:

```
QUEX_OPTION_USER_MANAGED_TOKEN_MEMORY
```

It provides the functions `token_queue_memory_switch(...)` for token policy 'queue' and `token_p_switch(...)` for token policy 'single' (section *Token Passing Policies*).

-token-queue-size `number`
In conjunction with token passing policy 'queue', `number` specifies the number of tokens in the token queue. This determines the maximum number of tokens that can be send without returning from the analyzer function.

DEFAULT = `64`.

-token-queue-safety-border `number`
Specifies the number of tokens that can be sent at maximum as reaction to one single pattern match. More precisely, it determines the number of token slots that are left empty when the token queue is detected to be full.

DEFAULT = `16`

-no-warning-on-no-token-queue
If set, this option disables the warning in case that an event handler is specified without having set `--token-policy queue`. Without a token queue no tokens can be propperly sent from inside event handlers. There might be other things the user decides to do without receiving any warning.

-token-id-offset `number`
`number` = Number where the numeric values for the token ids start to count. Note, that this does not include the standard token ids for termination, unitialized, and indentation error.

DEFAULT = `10000`

Certain token ids are standard, in a sense that they are required for a functioning lexical analyzer. Namely they are `TERMINATION` and `UNINITIALIZED` The default values of those do not follow the token id offset, but are 0, 1, and 2. If they need to be different, they must be defined in the `token { ... }` section, e.g.:

```
token {
    TERMINATION   = 10001;
    UNINITIALIZED = 10002;
    ...
}
```

A file with token ids can be provided by the option

-foreign-token-id-file `filename`
`filename` = Name of the file that contains an alternative definition of the numerical values

for the token-ids (see also section <<sec-formal-macro>>).

DEFAULT = <empty>

The following options support the definition of a independently customized token class:

**-token-class-file`** `filename`
: `filename` = Name of file that contains the definition of the token class. Note, that the setting provided here is possibly overwritten if the `token_type` section defines a file name explicity (see *Customized Token Classes*).

  DEFAULT = `$(QUEX_PATH)/code_base/Token`

-token-class`, `` `-tc` `` `name`
: `name` is the name of the token class. Using '::'-separators it is possible to defined the exact namespace as mentioned for the *–analyzer-class* command line option.

**-token-id-type`** `type-name`
: `type-name` defines the type of the token id. This defines internally the macro `QUEX_TYPE_TOKEN_ID`. This macro is to be used when a customized token class is defined. The types of Standard C99 'stdint.h' are encouraged.

  DEFAULT = `uint32_t`

-token-type-no-stringless-check`, `` `-ttnsc` ``
: Disable the 'stringless check' for customized token types. If the user defines a token type that cannot take a `QUEX_TYPE_CHARACTER*` then quex posts a warning. By means of this flag the warning is disabled.

There may be cases where the characters used to indicate buffer limit needs to be redefined, because the default value appear in a pattern footnote:[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.]. The following option allows modification of the buffer limit code:

-buffer-limit `number`
: DEFAULT = `0x0`

If the trivial end-of-line pre-condition (i.e. the '$' at the end of a regular expression) is used, by default quex produces code that runs on both Unix and DOS-like systems. Practically, this means that it matches against 'newline' 0x0A and 'carriage return/newline' 0x0D 0x0A. For the case that the resulting analyzer only runs on a Unix machine some tiny performance improvements might be achieved by disabling the 0x0D 0x0A sequence and only triggering on 0x0A. In this case, the following flag may be specified:

-no-DOS

For unicode support it is essential to allow character conversion. Currently quex can interact with GNU IConv and IBM's ICU library. For this the correspondent library must be installed on your system. On Unix systems, the iconv library is usually present. If a coding other than ASCII is required, specify the following options:

-iconv

Enable the use of the iconv library for character stream decoding. This is equivalent to defining '-DQUEX_OPTION_CONVERTER_ICONV' as a compiler flag. Depending on your compiler setup, you might have to set the '-liconv' flag explicitly in order to link against the IConv library. DEFAULT = <*disabled*>

-icu

Enable the use of IBM's ICU library for character stream decoding. This is equivalent to defining '-DQUEX_OPTION_CONVERTER_ICU' as a compiler flag. There are a couple of libraries that are required for ICU. You can query those using the ICU tool 'icu-config'. A command line call to this tool with '–ldflags' delivers all libraries that need to be linked. A typical list is '-lpthread -lm -L/usr/lib -licui18n -licuuc -licudata'. DEFAULT = <*disabled*>

-b, -bes, -buffer-element-size `[1, 2, 4]`

With this option the number of bytes are specified that a buffer element occupies. The buffer element is the 'trigger' on which the analyzer's state machine triggers. Usually, a buffer element carries a character. This is true for fixed size character encodings, or when converters are used (see options `--icu` or `--iconv`). In these cases all characters are internally processed in Unicode. Thus, the size of a buffer element should be large enough so that it can carry the unicode value of any character of the desired input coding space. When using Unicode, to be safe '-b 4' should be used except that it is unconceivable that any code point beyond 0xFFFF ever appears. In this case '-b 2' is enough.

Dynamic size character encodings are available with the '–codec' option, for example 'utf8' and 'utf16'. In this case, the buffer element carries a 'character chunk'. UTF8 runs on chunks of one byte. UTF16 runs on two byte chunks. Respectively, UTF8 requires '-b 1' and UTF16 requires '-b 2'.

You can only specify `1` byte, `2` byte or `4` byte per character.

DEFAULT = `1`

Warning: If a character size different from one byte is used, the `.get_text()` member of the token class does contain an array that particular type. This means, that `.text().c_str()` does not result in a nicely printable UTF8 string. Use the member `.utf8_text()` instead.

-buffer-element-size-irrelevant

If this flag is specified, the regular expressions are not constructed and not checked to fit a particular character range. Note, that this option is implicitly set when a '–codec' is specified.

DEFFAULT: not set.

-bet, -buffer-element-type `name`

A flexible approach to specify the buffer element size and type is by specifying the name of the buffer element's type, which is the purpose of this option. Note, that there are some 'well-known' types such as `uint*_t` (C99 Standard), `u*` (Linux Kernel), `unsigned*` (OSAL) where the `*` stands for 8, 16, or 32. Quex can derive its size automatically.

Note, that if a converter is specified (`--iconv`, `--icu`, or `--converter-new`) and the buffer element type does not allow to deduce the unicode coding name that the converter requires, then it must be explicitly specified by '–converter-ucs-coding-name'.

DEFAULT: Determined by buffer element size.

-endian `[little, big, <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 quex runs then this option is not required, since quex 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`.

According to the setting of this option one of the three macros is defined in the header files:

- `__QUEX_OPTION_SYSTEM_ENDIAN`
- `__QUEX_OPTION_LITTLE_ENDIAN`
- `__QUEX_OPTION_BIG_ENDIAN`

Those macros are of primary use for character code converters. The converters need to know what the analyser engines number representation is. However, the user might want to use them for his own special purposes (using `#ifdef __QUEX_OPTION_BIG_ENDIAN ... #endif`).

DEFAULT="”<system>”‘

-converter-new `String`
Section *Customized Converters* explains how to implement customized converters. With the command line option above the user may specify his own converter. The string that follows the option is the name of the converter's `_New` function. When this option is set, automatically customized user conversion is turned on.

-converter-ucs-coding-name, -cucn
Determines what string is passed to the converter so that it converters a codec into unicode. In general, this is not necessary. But, if a unknown user defined type is specified via '–buffer-element-type' then this option must be specified.

DEFAULT: Determined by buffer element type.

-codec `String`
Specifies a codec for the generated engine. By default the internal engine runs on unicode code points, i.e. ASCII for characters below 0x7F. When `--codec` specifies a codec 'X', for example, is specified, the internal engine triggers on code elements of 'X'. It does not need character conversion (neither `--iconv` nor `--icu`). Codec based analyzers are explained in section *Analyzer Engine Codec*.

When `--codec` is specified the command line flag `-b` or `--buffer-element-size` does not represent the number of bytes per character, but *the number of bytes per code element*. The codec UTF8, for example, is of dynamic length and its code elements are bytes, thus only `-b 1` makes sense. UTF16 triggers on elements of two bytes, while the length of an encoding for a character varries. For UTF16, only `-b 2` makes sense.

When `--codec` is specified, the range check for characters is disabled. That means, the option `--buffer-element-size-irrelevant` is set automatically.

-codec-file `filename.dat`

By means of this option a freely customized codec can be defined. The follower `filename.dat` determines at the same time the file where the codec mapping is described and the codec's name. The codec's name is the directory-stripped and extension-less part of the given follower. Each line of such a file must consist of three numbers, that specify 'source interval begin', 'source interval length', and 'target interval end. Such a line specifies how a cohesive Unicode character range is mapped to the number range of the customized codec. For example, the mapping for codec iso8859-6 looks like the following:

```
0x000 0xA1 0x00
0x0A4 0x1  0xA4
0x0AD 0x1  0xAD
0x60C 0x1  0xAC
0x61B 0x1  0xBB
0x61F 0x1  0xBF
0x621 0x1A 0xC1
0x640 0x13 0xE0
```

Here, the Unicode range from 0 to 0xA1 is mapped one to one from Unicode to the codec. 0xA4 and 0xAD are also the same as in Unicode. The remaining lines describe how Unicode characters from the 0x600-er page are mapped inside the range somewhere from 0xAC to 0xFF.

Note, that this option is only to be used, if quex does not support the codec directly. The options `--codec-info` and `--codec-for-language` help to find out whether Quex directly supports a specific codec. If a `--codec-file` is required, it is advisable to use `--codec-file-info filename.dat` to see if the mapping is in fact as desired.

Template and Path Compression can be controlled with the following command line options:

-template-compression

If this option is set, then template compression is activated.

-template-compression-min-gain `'number'`

The number following this option specifies the template compression coefficient. It indicates the relative cost of routing to a target state compared to a simple 'goto' statement. The optimal value may vary from processor platform to processor platform, and from compiler to compiler.

DEFAULT = 1

-path-compression

This flag activates path compression. By default, it compresses any sequence of states that allow to be lined up as a 'path'. This includes states of different acceptance values, store input positions, etc.

-path-compression-uniform
This flag enables path compression. In contrast to the previous flag it compresses such states into a path which are uniform. This simplifies the structure of the correspondent pathwalkers. In some cases this might result in smaller code size and faster execution speed.

-path-termination `'number'`
Path compression requires a 'pathwalker' to determine quickly the end of a path. For this, each path internally ends with a signal character, the 'path termination code'. It must be different from the buffer limit code in order to avoid ambiguities.

Modification of the 'path termination code' makes only sense if the input stream to be analyzed contains the default value.

DEFAULT = `0x1`.

For version information pass option *–version*' or *-v*'. The options *–help* and *-h*' are reserved for requesting a help text. Those are the options for using quex in the 'normal' mode where it creates lexical analyzers. However, quex provides some services to query and test character sets. If one of those options is called, then quex does not create a lexical analyzer but responds with some information requested by the user. Those options are the following.

-codec-info `[name]`
Displays the characters that are covered by the given codec's name. If the name is omitted, a list of all supported codecs is printed. Engine internal character encoding is discussed in section *sec-engine-internal-coding*.

-codec-file-info `filenname.dat`
Displays the characters that are covered by the codec provided in the given file. This makes sense in conjunction with `--codec-file` where customized codecs can be defined.

-codec-for-language `[language]`
Displays the codecs that quex supports for the given human language. If the language argument is omitted, all available languages are listed.

-property `name`
If `name` is specified, then information about the property with the given name is displayed. Note, that `name` can also be a property alias. If `name` is not specified, then brief information about all available unicode properties is displayed.

-set-by-property `setting`
For binary properties only the property name has to be specified. All other properties require a term of the form `property-name = value`. Quex then displays the set of character that has this particular property.

-set-by-expression `expression`
Character set expressions that are ususally specified in `[: ... :]` brackets can be specified as expression. Quex then displays the set of characters that results from it.

-property-match `wildcard-expression`
: Quex allows the use of wildcards in property values. Using this option allows display of the list of values to which the given wildcard expression expands. Example: The wildcard-expression `Name=*LATIN*` gives all settings of property `Name` that contain the string `LATIN`.

-numeric
: If this option is specified the numeric character codes are displayed rather then the utf8 characters.

-intervals
: This option disables the display of single character or single character codes. In this case sets of adjacent characters are displayed as intervals. This provides a somewhat more abbreviated display.

The following options control the comment which is added to the generated code:

-comment-state-machine
: With this option set a comment is generated that shows all state transitions of the analyzer in a comment at the begin of the analyzer function. The format follows the scheme presented in the following example

```
/* BEGIN: STATE MACHINE
 ...
 * 02353(A, S) <~ (117, 398, A, S)
 *       <no epsilon>
 * 02369(A, S) <~ (394, 1354, A, S), (384, 1329)
 *       == '=' ==> 02400
 *       <no epsilon>
 ...
 * END: STATE MACHINE
 */
```

It means that state 2369 is an acceptance state (flag 'A') and it should store the input position ('S'), if no backtrack elimination is applied. It originates from pattern '394' which is also an acceptance state and '384'. It transits to state 2400 on the event of a '=' character.

-comment-transitions
: Adds to each transition in a transition map information about the characters which trigger the transition, e.g. in a transition segment implemented in a C-switch case construct

```
...
case 0x67:
case 0x68: goto _2292;/* ['g', 'h'] */
case 0x69: goto _2295;/* 'i' */
case 0x6A:
case 0x6B: goto _2292;/* ['j', 'k'] */
case 0x6C: goto _2302;/* 'l' */
case 0x6D:
...
```

The output of the characters happens in UTF8 format.

-comment-mode-patterns

If this option is set a comment is printed that shows what pattern is present in a mode and from what mode it is inherited. The comment follows the following scheme:

```
/* BEGIN: MODE PATTERNS
 ...
 * MODE: PROGRAM
 *
 *     PATTERN-ACTION PAIRS:
 *       (117) ALL:      [ \r\n\t]
 *       (119) CALC_OP:  "+"|"-"|"*"|"/"
 *       (121) PROGRAM:  "//"
 ...
 * END: MODE PATTERNS
 */
```

This means, that there is a mode `PROGRAM`. The first three pattern are related to the terminal states '117', '119', and '121'. The whitespace pattern of 117 was inherited from mode *ALL*. The math operator pattern was inherited from mode `CALC_OP` and the comment start pattern "//" was implemented in `PROGRAM` itself.

The comment output is framed by `BEGIN:` and `END:` markers. This facilitates the extraction of this information for further processing. For example, the Unix command 'awk' can be used:

```
awk 'BEGIN {w=0} /BEGIN:/ {w=1;} // {if(w) print;} /END:/ {w=0;}' MyLexer.c
```

The following option influences the analysis process on the very lowest level.

-state-entry-analysis-complexity-limit N

Note: Never use this option until quex proposes in a warning message that you may use it to control the speed of code generation. The warning message proposing the usage of this option should only appear in engines with thousands of very similar patterns including some repetitions.

For state entry analysis an algorithm is applied that is quadratic with the number of different cases to be considered. In extremely strange setups, this may blow the computation time beyond of what is acceptable. When more than 'N' different cases are detected, Quex only considers the 'N' best candidates in the search of an optimal solution. This includes a certain risk of not finding the absolute optimum.

DEFAULT = 1000

8.2 Compile Options

Quex offers a set of compile options that can be set by specifying them via the `-D` flag of the C-compiler. No explicit setting is necessary to create a working analyzer. The options, though, allow to configure and fine tune the lexical analyzer engine. The compile options split into the following groups

QUEX_OPTION_ . . .

These macros can either be defined or not. If they are defined they trigger a certain behavior. All macros of this group have a sibling of the shape:

```
QUEX_OPTION_ ... _DISABLED
```

If this option is set than any setting is outruled, even those that have been set by quex. To disable asserts for example the compiler is called like this:

```
$CC ... -DQUEX_OPTION_ASSERTS_DISABLED
```

The disablement of asserts is, by the way, essential to high performance lexical analysis.

QUEX_SETTING_ . . .

These macros contain a value that is expanded in the code of quex. This can for example be used to set the buffer size with:

```
$CC ... -DQUEX_SETTING_BUFFER_SIZE=4096
```

QUEX_TYPE_ . . .

By means of those macros types are defined. The following overides quex's definition for the internal character type:

```
$CC ... -DQUEX_TYPE_CHARACTER=UChar
```

None of the following compile options is necessary to get a working analyser. Indeed, many of them are already controlled by quex. Nevertheless, they provide toolset for fine tuning or adaptions to less regular system constraints.

8.2.1 Options

For the sake of simplicity only the positive options are mentioned. For each (or most) of the options below, there exists a sibling with a `_DISABLED` suffix. The option without the suffix enables something, the option with the suffix disables the 'something'.

QUEX_OPTION_ASSERTS

Explicitly enables asserts which are enabled by default anyway. More of use is the sibling `QUEX_OPTION_ASSERTS_DISABLED`. Disabling asserts is key for getting reasonable performance.

QUEX_OPTION_AUTOMATIC_ANALYSIS_CONTINUATION_ON_MODE_CHANGE
If defined (as by default) mode changes may happen without a token being returned. Then, the analysis process continues until the first valid token is returned. However, if:

```
QUEX_OPTION_AUTOMATIC_ANALYSIS_CONTINUATION_ON_MODE_CHANGE_DISABLED
```

is defined, then the `.receive()` function returns always without checking that a token has actually been sent. This option is only effective for the token policy 'single'. With the 'queue' token policy, the analyzer never returns with an empty token queue.

QUEX_OPTION_BUFFER_BASED_ANALYZIS
Turn on buffer based analyzis. In particular automatic buffer reload is disabled. Also some consistency checks are disabled if asserts were enabled.

QUEX_OPTION_COLUMN_NUMBER_COUNTING
Enables column number counting. Respectively, `QUEX_OPTION_COLUMN_NUMBER_COUNTING_DISA` disables it.

QUEX_OPTION_COMPUTED_GOTOS
Enables/disables the usage of computed gotos. Use this option only if your compiler supports it. For example, GNU's gcc does support it from version 2.3.

QUEX_OPTION_CONVERTER_ICONV
Enables/disables the use of the GNU IConv library for conversion. This can also be set with the command line option `--iconv`.

QUEX_OPTION_CONVERTER_ICU
Enables/disables the use of the IBM's ICU library for conversion. This can also be set with the command line option `--icu`. Note, that IConv and ICU are best used mutually exclusive.

QUEX_OPTION_DEBUG_SHOW
Enables/disables the output of details about the lexical analyzis process.

QUEX_OPTION_DEBUG_SHOW_LOADS
Enables/disables the output of details about reloads of the buffer.

QUEX_OPTION_ENABLE_PINARD_RECODE
Enables/disables Francois Pinards Recode library. At the time of this writing the converter is not yet implemented.

QUEX_OPTION_ERROR_OUTPUT_ON_MODE_CHANGE_ERROR
Enables/disables error messages on disallows mode transitions.

QUEX_OPTION_INCLUDE_STACK
Enables/disables the feature of an intelligent include stack.

QUEX_OPTION_INFORMATIVE_BUFFER_OVERFLOW_MESSAGE
Enables/disables informative buffer overflow messages. If set it prints the lexeme on which the buffer overflow happend.

QUEX_OPTION_LINE_NUMBER_COUNTING
Enables/disables line number counting. Line and column number counting are implemented very efficiently. Most probably no performance decrease will be measurable by deactivating this feature.

QUEX_OPTION_POST_CATEGORIZER
Enables/disables an optional post categorizer that maps from a lexeme to a token-id.

QUEX_OPTION_RUNTIME_MODE_TRANSITION_CHECK
Enables/disables mode transition checks during run-time.

QUEX_OPTION_STRANGE_ISTREAM_IMPLEMENTATION
Some input streams behave rather strange. When receiving `N` characters from a stream, their stream position might increase by a number `M` which is different from `N`. To handle those streams, set this option.

Alternatively, one might consider opening the stream in binary mode.

QUEX_OPTION_TERMINATION_ZERO_DISABLED
If this macro is defined, the setting of the terminating zero at the end of a lexeme is disabled. This may cause some speed-up, but it is necessary in order to run the lexical analyzer on read-only memory.

QUEX_OPTION_STRING_ACCUMULATOR
Enables/disables an optional accumulator which can collect some longer string.

QUEX_OPTION_TOKEN_POLICY_QUEUE
Enables/disables the token sending via a token queue. With the current version of quex (0.36.4) some performance increase can be achieved if the token queue is disabled.

QUEX_OPTION_TOKEN_POLICY_SINGLE
Enables/disables the token sending via a token object owned by the user.

QUEX_OPTION_TOKEN_STAMPING_WITH_LINE_AND_COLUMN
Enables/disables the stamping of tokens with the line and column number. The stamping happens by default. If it is desired to disable this stamping the `..._DISABLED` version of this macro must be defined. If column or line counting is disabled the corresponding stamping is also disabled, anyway.

QUEX_OPTION_WARNING_ON_PLAIN_FILLER_DISABLED
Disable the warning message that is printed if a analyzer is used without a character encoding while it was created for a converter interface such as IConv or ICU.

QUEX_OPTION_SEND_AFTER_TERMINATION_ADMISSIBLE
If asserts are enabled, then quex triggers an error message whenever a token is sent after the 'TERMINATION' token. In some cases, this might just be convenient, so that one might want to allow it by defining this macro.

8.2.2 Settings

Following settings can be made on the command line:

QUEX_SETTING_BUFFER_FILLER_SEEK_TEMP_BUFFER_SIZE
: For seeking in character streams a temporary buffer is required. By means of this macro its size can be specified.

QUEX_SETTING_BUFFER_LIMIT_CODE
: The buffer limit code is by default 0x0. If it is intended that this character code appears inside patterns, then it need to be reset by this setting.

QUEX_SETTING_BUFFER_MIN_FALLBACK_N
: Buffers in quex keep a certain fallback region when loading new content into the buffer. This prevents the lexer from reloading backwards if the input pointer is navigated below the current reload position. Note, that this has only effect in case of manual navigation of the input pointer, or in scanners with pre-conditions.

QUEX_SETTING_BUFFER_SIZE
: The size of a buffer on which the quex engine does its analysis. The larger the buffer, the less reloads are required. But at a certain buffer size (usually about 32Kb) an increase in buffer size does not show measurable performance increase.

  The buffer size *must* be greater or equal the largest conceivable lexeme that the lexer can match.

QUEX_SETTING_ICU_PIVOT_BUFFER_SIZE
: When using IBM's ICU library for character conversion (command line option `--icu`), the intermediate buffer size is determined by this setting.

QUEX_SETTING_MODE_STACK_SIZE
: Defines the size of the mode stack. It defines the number of 'sub mode calls' that can be done via the `.push_mode` and `.pop_mode` member functions. See also section *Mode Transitions*.

QUEX_SETTING_TOKEN_QUEUE_SIZE
: Defines the initial number of elements of the token queue–if it is enable (see also command line option `--no-token-queue`)

QUEX_SETTING_TOKEN_QUEUE_SAFETY_BORDER
: Defines the size of the safety border. In other words, this corresponds to the maximum number of tokens to be sent caused by a pattern match '-1'. If each pattern sends only one token, it can be set safely to zero.

QUEX_SETTING_TRANSLATION_BUFFER_SIZE
: When using a converter (`--icu` or `--iconv`) this is the size of the buffer where the original stream data is read into.

QUEX_SETTING_OUTPUT_TEMPORARY_UTF8_STRING_BUFFER_SIZE
: When printing the `text` content of the default token type, it can be converted to UTF8. The

conversion requires a temporary buffer whose size is defined by means of this macro.

8.2.3 Types

QUEX_TYPE_ANALYZER_FUNCTION
Function pointer type of the analyzer function. Best not to be changed.

QUEX_TYPE_CHARACTER
Internal carrier type for characters. This is set automatically by quex if the command line option `-b` or `--buffer-element-size` is specified.

QUEX_TYPE_TOKEN_ID
Type to be used to store the numeric value of a token identifier. On small systems this might be set to `short` or `unsigned char`. On larger systems, if very many different token-ids exits (e.g. > 2e32) then it might be set to `long`. It might be more appropriate to use standard types from `stdint.h` such as `uint8_t`, `uint16_t` etc.

The character type can also be specified via the command line option `-b`, or `--buffer-element-size`.

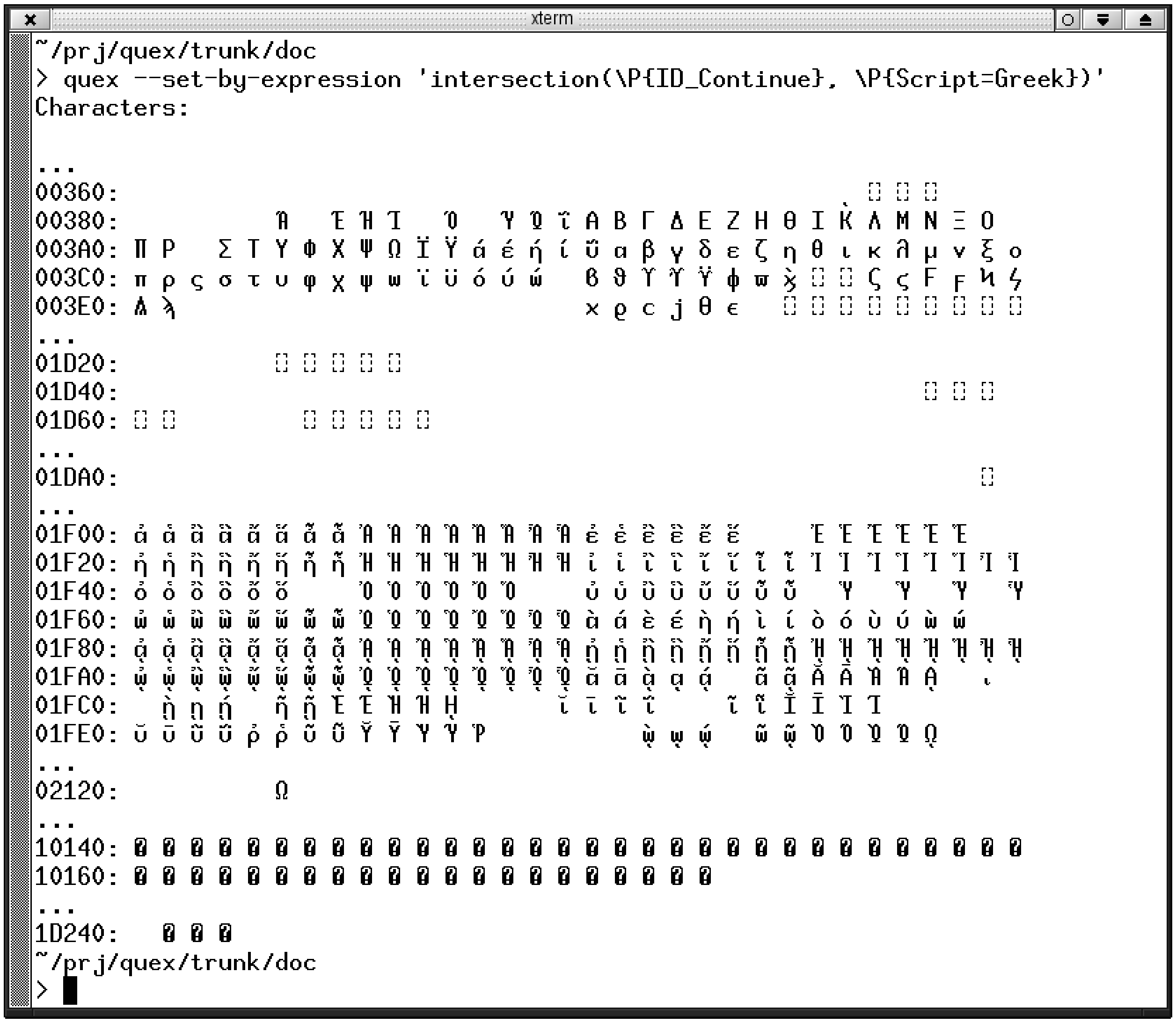
8.3 Character Set Queries

The specification of character sets based on properties and character set operations requires in general some closer inspection in order to avoid the inclusion of unwanted characters or to span a character set that is wider than actually intended. Consider for example that one might want to provide a programming language that allows Latin and Greek letters in identifiers. The direct approach would be as follows:

```
define {
    LATIN_ID_START   [: intersection(\P{ID_Start},    \P{Script=Latin}) :]
    GREEK_ID_START   [: intersection(\P{ID_Start},    \P{Script=Greek}) :]
    LATIN_ID_CONT    [: intersection(\P{ID_Continue}, \P{Script=Latin}) :]
    GREEK_ID_CONT    [: intersection(\P{ID_Continue}, \P{Script=Greek}) :]

    LATIN_ID   {LATIN_ID_START}({LATIN_ID_CONT})*
    GREEK_ID   {GREEK_ID_START}({GREEK_ID_CONT})*
}
```

This specification is totally rational. However, it might include more characters than one actually intended. If one mentions Greek and Latin characters one usually thinks about lower and upper case letters from a to z and α to ω. However, the set of characters that are included in `Script=Latin` is much larger than that–and so is the set of characters in `Script=Greek`. Figure *Greek identifier* displays the set of characters for the character set specified by `GREEK_ID_CONT`.



```
~/prj/quex/trunk/doc
> quex --set-by-expression 'intersection(\P{ID_Continue}, \P{Script=Greek})'
Characters:

...
00360:                                                   ͺ □ □ □
00380:           Ά   Έ Ή Ί   Ό   Ύ Ώ ΐ Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο
003A0: Π Ρ   Σ Τ Υ Φ Χ Ψ Ω Ϊ Ϋ ά έ ή ί ΰ α β γ δ ε ζ η θ ι κ λ μ ν ξ ο
003C0: π ρ ς σ τ υ φ χ ψ ω ϊ ϋ ό ύ ώ   ϐ ϑ ϒ ϓ ϔ ϕ ϖ ϗ □ □ Ϛ ϛ Ϝ ϝ Ϟ ϟ
003E0: Ϡ ϡ                             ϰ ϱ ϲ ϳ ϴ ϵ   □ □ □ □ □ □ □ □ □
...
01D20:           □ □ □ □ □
01D40:                                                         □ □ □
01D60: □ □         □ □ □ □ □
...
01DA0:                                                             □
...
01F00: ἀ ἁ ἂ ἃ ἄ ἅ ἆ ἇ Ἀ Ἁ Ἂ Ἃ Ἄ Ἅ Ἆ Ἇ ἐ ἑ ἒ ἓ ἔ ἕ     Ἐ Ἑ Ἒ Ἓ Ἔ Ἕ
01F20: ἠ ἡ ἢ ἣ ἤ ἥ ἦ ἧ Ἠ Ἡ Ἢ Ἣ Ἤ Ἥ Ἦ Ἧ ἰ ἱ ἲ ἳ ἴ ἵ ἶ ἷ Ἰ Ἱ Ἲ Ἳ Ἴ Ἵ Ἶ Ἷ
01F40: ὀ ὁ ὂ ὃ ὄ ὅ     Ὀ Ὁ Ὂ Ὃ Ὄ Ὅ     ὐ ὑ ὒ ὓ ὔ ὕ ὖ ὗ   Ὑ   Ὓ   Ὕ   Ὗ
01F60: ὠ ὡ ὢ ὣ ὤ ὥ ὦ ὧ Ὠ Ὡ Ὢ Ὣ Ὤ Ὥ Ὦ Ὧ ὰ ά ὲ έ ὴ ή ὶ ί ὸ ό ὺ ύ ὼ ώ
01F80: ᾀ ᾁ ᾂ ᾃ ᾄ ᾅ ᾆ ᾇ ᾈ ᾉ ᾊ ᾋ ᾌ ᾍ ᾎ ᾏ ᾐ ᾑ ᾒ ᾓ ᾔ ᾕ ᾖ ᾗ ᾘ ᾙ ᾚ ᾛ ᾜ ᾝ ᾞ ᾟ
01FA0: ᾠ ᾡ ᾢ ᾣ ᾤ ᾥ ᾦ ᾧ ᾨ ᾩ ᾪ ᾫ ᾬ ᾭ ᾮ ᾯ ᾰ ᾱ ᾲ ᾳ ᾴ   ᾶ ᾷ Ᾰ Ᾱ Ὰ Ά ᾼ   ι
01FC0:   ῂ ῃ ῄ   ῆ ῇ Ὲ Έ Ὴ Ή ῌ       ῐ ῑ ῒ ΐ     ῖ ῗ Ῐ Ῑ Ὶ Ί
01FE0: ῠ ῡ ῢ ΰ ῤ ῥ ῦ ῧ Ῠ Ῡ Ὺ Ύ Ῥ           ῲ ῳ ῴ   ῶ ῷ Ὸ Ό Ὼ Ώ ῼ
...
02120:           Ω
...
10140: ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
10160: ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
...
1D240:   ■ ■ ■
~/prj/quex/trunk/doc
>
```

Figure 8.1: Example query for greek identifier characters.

As can be seen in the screenshot, the character set that is actually spanned by the expression is rather huge and exceeds the set of characters that can be displayed by the console of the user. At this point in time, one might have to use further `intersection` and `difference` operations in order to cut out the set that is actually desired. Also, consider not using the powerful tool of unicode properties, if things are expressed elegantly with traditions character set expressions. In the above case, a solution that likely fulfills the user's purpose might look like this:

```
define {
    LATIN_ID    [_a-zA-Z]([_a-zA-Z0-9])*
    GREEK_ID    [_α-ωΑ-Ω]([_α-ωΑ-Ω0-9])*
}
```

Including other scripts, or enabling other features of a used language requires closer investigation of the unicode properties. Quex provides some powerful services to investigate the effect of a character set expression on the command line. Sophisticated programming language design on a non-latin language, though, requires some review of the literature of UCS based character sets and their characteristics. This chapter shows how quex can be used to query the unicode database and see the effect of character set expressions immediately, without having to compile a whole lexical analyzer.

8.3.1 Unicode Properties

The first doubt that arises when properties are to be applied is whether the properties actually exists. The second doubt, then, is what values these properties can actually have. To help out with the first question, simply call quex with the command line option `--property` and it prints a list of properties that it is able to extract from the unicode database. This leads to an output as follows:

```
# Abbreviation, Name, Type
AHex,    ASCII_Hex_Digit,                    Binary
Alpha,   Alphabetic,                         Binary
Bidi_C,  Bidi_Control,                       Binary
Bidi_M,  Bidi_Mirrored,                      Binary
CE,      Composition_Exclusion,              Binary
Comp_Ex, Full_Composition_Exclusion,         Binary
DI,      Default_Ignorable_Code_Point,       Binary
Dash,    Dash,                               Binary
Dep,     Deprecated,                         Binary

...

na,      Name,                               Miscellaneous
na1,     Unicode_1_Name,                     Miscellaneous
nt,      Numeric_Type,                       Enumerated
nv,      Numeric_Value,                      Numeric
sc,      Script,                             Catalog
scc,     Special_Case_Condition,             String,         <unsupported>
```

```
sfc,      Simple_Case_Folding,                String,       <unsupported>
slc,      Simple_Lowercase_Mapping,           String,       <unsupported>
stc,      Simple_Titlecase_Mapping,           String,       <unsupported>
suc,      Simple_Uppercase_Mapping,           String,       <unsupported>
tc,       Titlecase_Mapping,                  String,       <unsupported>
uc,       Uppercase_Mapping,                  String,       <unsupported>
```

Each line contains three fields separated by commas. The first field contains the *alias* for the property name, the second field contains the *property name*, and the last column contains the *type* of property. If a field containing `<unsuported>` is appended, this means that this property is not supported by quex. In most cases this so because these properties support character operations rather then the definition of character sets.

To help out with the second question call quex with the command line option `--property` followed by the name of the property that you want to know more about. The following displays the query about the property `Numeric_Type`:

```
> quex --property Numeric_Type
(please, wait for database parsing to complete)

NAME          = 'Numeric_Type'
ALIAS         = 'nt'
TYPE          = 'Enumerated'
VALUE_ALIASES = {
        Decimal(De),
        Digit(Di),
        Numeric(Nu).
}
```

This tells, that `Numeric_Type` is a property of type `Enumerated`, i.e. its values are taken from a fixed list of values. The *alias* `nt` can be used as a placeholder for `Numeric_Type`, and the possible value settings are `Decimal`, `Digit`, and `Numeric`. The strings mentioned in brackets are the *value aliases* that can be used as placeholders for the values, in case one does not want to type the whole name of the value. From this output one knows that expressions such as `\P{Numeric_Type=Decimal}`, `\P{nt=Di}`, and `\P{Numeric_Type=Nu}` are valid for this property. The next doubt that arises is about the character set that is actually spanned by such expressions. This is discussed in the subsequent section.

8.3.2 Character Set Expressions

The specification of character sets using unit code properties requires caution and a precise vision of the results of character set operations. In order to allow to user to see results of character set operations, quex provides a query mode where the character set corresponding to a specific property setting can be investigated. Additionally, whole character set expressions can be specified and quex displays the set of characters that results from it.

In order to display the character set related to a unicode property setting, call quex with the option `--set-by-property` followed by a string that specifies the setting in the same way as this is done in regular expressions with the `\P{...}` operation. Note, that binary properties have no value in the sense that they are specified in a manner as `property-name=value`. Instead, it is sufficient to give the name of the binary property and quex displays all characters that have that property, e.g.

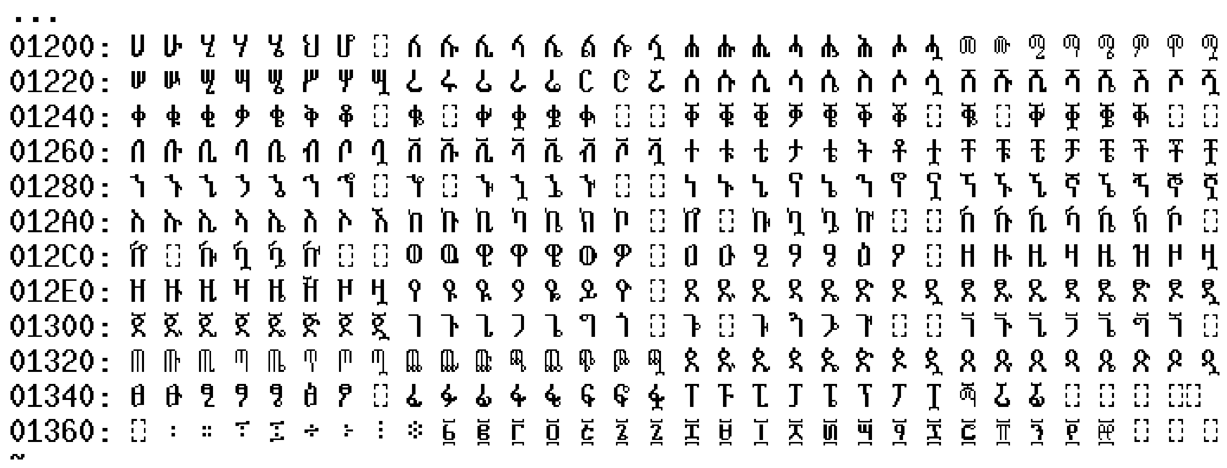
```
> quex --set-by-property ASCII_Hex_Digit
```

displays all characters that have the property `ASCII_Hex_Digit`, i.e.

```
00020:                              0 1 2 3 4 5 6 7 8 9
00040: A B C D E F
00060: a b c d e f
```

For non-binary properties the value of the property has to be specified. For example, the set of characters where the property `Block` is equal to `Ethiopic` can be displayed by calling

```
> quex --set-by-property Block=Ethiopic
```



```
...
01200: ሀ ሁ ሂ ሃ ሄ ህ ሆ □ ለ ሉ ሊ ላ ሌ ል ሎ ሏ ሐ ሑ ሒ ሓ ሔ ሕ ሖ ሗ መ ሙ ሚ ማ ሜ ም ሞ ሟ
01220: ሠ ሡ ሢ ሣ ሤ ሥ ሦ ሧ ረ ሩ ሪ ራ ሬ ር ሮ ሯ ሰ ሱ ሲ ሳ ሴ ስ ሶ ሷ ሸ ሹ ሺ ሻ ሼ ሽ ሾ ሿ
01240: ቀ ቁ ቂ ቃ ቄ ቅ ቆ □ ቈ □ ቊ ቋ ቌ ቍ □ □ ቐ ቑ ቒ ቓ ቔ ቕ ቖ □ ቘ □ ቚ ቛ ቜ ቝ □ □
01260: በ ቡ ቢ ባ ቤ ብ ቦ ቧ ቨ ቩ ቪ ቫ ቬ ቭ ቮ ቯ ተ ቱ ቲ ታ ቴ ት ቶ ቷ ቸ ቹ ቺ ቻ ቼ ች ቾ ቿ
01280: ኀ ኁ ኂ ኃ ኄ ኅ ኆ □ ኈ □ ኊ ኋ ኌ ኍ □ □ ነ ኑ ኒ ና ኔ ን ኖ ኗ ኘ ኙ ኚ ኛ ኜ ኝ ኞ ኟ
012A0: አ ኡ ኢ ኣ ኤ እ ኦ ኧ ከ ኩ ኪ ካ ኬ ክ ኮ □ ኰ □ ኲ ኳ ኴ ኵ □ □ ኸ ኹ ኺ ኻ ኼ ኽ ኾ □
012C0: ዀ □ ዂ ዃ ዄ ዅ □ □ ወ ዉ ዊ ዋ ዌ ው ዎ □ ዐ ዑ ዒ ዓ ዔ ዕ ዖ □ ዘ ዙ ዚ ዛ ዜ ዝ ዞ ዟ
012E0: ዠ ዡ ዢ ዣ ዤ ዥ ዦ ዧ የ ዩ ዪ ያ ዬ ይ ዮ □ ደ ዱ ዲ ዳ ዴ ድ ዶ ዷ ዸ ዹ ዺ ዻ ዼ ዽ ዾ ዿ
01300: ጀ ጁ ጂ ጃ ጄ ጅ ጆ ጇ ገ ጉ ጊ ጋ ጌ ግ ጎ □ ጐ □ ጒ ጓ ጔ ጕ □ □ ጘ ጙ ጚ ጛ ጜ ጝ ጞ □
01320: ጠ ጡ ጢ ጣ ጤ ጥ ጦ ጧ ጨ ጩ ጪ ጫ ጬ ጭ ጮ ጯ ጰ ጱ ጲ ጳ ጴ ጵ ጶ ጷ ጸ ጹ ጺ ጻ ጼ ጽ ጾ ጿ
01340: ፀ ፁ ፂ ፃ ፄ ፅ ፆ □ ፈ ፉ ፊ ፋ ፌ ፍ ፎ ፏ ፐ ፑ ፒ ፓ ፔ ፕ ፖ ፗ ፘ ፙ ፚ □ □ □ □□
01360: □ ፡ ። ፣ ፤ ፥ ፦ ፧ ፨ ፩ ፪ ፫ ፬ ፭ ፮ ፯ ፰ ፱ ፲ ፳ ፴ ፵ ፶ ፷ ፸ ፹ ፺ ፻ ፼ □ □ □
~
```

Characters for unicode property `Block=Ethiopic`.

and the result is displayed in figure <<fig-screenshot-block-ethiopic.png>>. Naturally, sets specified with a simple property setting are not precisely what the user desires. To apply complex operations of character set, quex provides character set operations <<sec-formal/patterns/operations>>. Again, it is essential to know which character sets these expressions expand. Thus, quex provides a function to investigate set expressions with the command line option `--set-by-expression`. The following call

```
> quex --set-by-expression 'intersection(\P{Script=Arabic}, \G{Nd})'
```

Displays all characters that are numeric digits in the arabic script. Note that the display of characters on the screen is not always meaningful, so you might specify the one of the options:

-numeric
: in order to display numeric values of the characters.

-intervals
: prints out character intervals instead of each isolated character.

-names
: which prints out the character names.

Depending on the specific information required, these options allow to display the results appropriately.

8.3.3 Property Wildcard Expressions

Property value settings can be specified using wildcards as explained in <<sec-formal/patterns/character-set>>. Using the command line option `--property-match` allows the user to see to which values these wildcard expressions actually expand. For example, the call

```
> quex --property-match Name=MATH*FIVE*
```

delivers:

```
MATHEMATICAL_BOLD_DIGIT_FIVE
MATHEMATICAL_DOUBLE-STRUCK_DIGIT_FIVE
MATHEMATICAL_MONOSPACE_DIGIT_FIVE
MATHEMATICAL_SANS-SERIF_BOLD_DIGIT_FIVE
MATHEMATICAL_SANS-SERIF_DIGIT_FIVE
```

Note, that the final character sets spanned by these settings can be viewed using `--set-by-property Name=MATH*FIVE*` and `--set-by-expression \N{MATH*FIVE*}` in the manner previously explained. For example,

```
> quex --set-by-property 'Name=MATH*FIVE*'
```

delivers:

```
1D7D0:             1D7D3
1D7D8:                         1D7DD
1D7E0:                                     1D7E7
...
1D7F0: 1D7F1
1D7F8:             1D7FB
```

8.3.4 Codec Queries

The usage of codecs is facilitated with the following query options:

-codec-info` [codec-name]
: Displays the characters which are supported by the given codec name. If no codec name is omitted, the list of all supported codecs is printed on the screen.

-codec-file-info` filename.dat
: Displays the characters that are covered by a user written codec mapping file.

-codec-for-language` [language-name]
: Displays all supported codecs which are designed for the specified (human) language. If no language's name is omitted, the list of all supported codecs is printed on the screen.

For example, if it is intended to design an input language for Chinese, quex can be called to show the directly supported codecs (other than Unicode):

```
> quex --codec-for-language
```

shows the list of supported languages:

```
...
command line: English, Traditional Chinese, Hebrew, Western Europe,
command line: Greek, Baltic languages, Central and Eastern Europe,
command line: Bulgarian, Byelorussian, Macedonian, Russian, Serbian,
command line: Turkish, Portuguese, Icelandic, Canadian, Arabic, Danish,
command line: Norwegian, Thai, Japanese, Korean, Urdu, Vietnamese,
command line: Simplified Chinese, Unified Chinese, West Europe, Esperanto,
command line: Maltese, Nordic languages, Celtic languages, Ukrainian,
command line: Kazakh,
```

Now, asking for the codecs supporting the Celtic language:

```
> quex --codec-for-language 'Celtic languages'
```

delivers `iso8859_14` as possible codec. Again a call to quex allows to verify if all desired characters are supported.

> quex –codec-info iso8859_14

By means of those queries it can be decided quickly which character encoding is the most appropriate for ones needs. For some script and languages, though, problems may arise from the fact that multiple code points carry the same 'character'. Systems that are able to render according to arabic letter linking rules might rely on 28 unicode code points starting from 600 (hex). The same 28 letters are represented in about 128 code points starting from FE80 (hex)-for systems that might not be able to do the rendering automatically.

The decision which encoding to choose is very specific to the the particular application. In any case, if `iconv` is installed on a system, the validity of an encoding can be checked easily. One starts with some representive samples of the text to be analyzed coded in UTF-8 (or any other 'complete' encoding). A call to `iconv` of the type:

```
> iconv -f utf-8 -t CodecX   sample.txt
```

shows the result on the screen. If there are characters which could not be translated, then `CodecX` is not suited as an encoding to handle the file `sample.txt`.

8.4 State Machine Plots

Since version 0.21.11, quex provides the ability to produce transition graphs for the state machines that result from the regular expressions of its modes. This happens with help of the tool 'graphviz' from AT&T Labs http://www.graphviz.org. Please install this package before any attempt to produce transition graphs.

The plot of transition graphs is initiated with the command line argument `--language dot` followed by a string that indicates the graphic format, e.g.

```
> quex -i core.qx --language dot
```

will produce transition graphs of all state machines that would result from descriptions in `core.qx`. As a result of this operation a set of files is created that carry the extension of the graphic format. The basenames of the files are related to the mode from which its content results. The characters on the state transitions can be either displayed as UTF8 characters or in hexadecimal notation, if the command line option `--character-display` is provided, e.f.

```
> quex -i core.qx --language dot --character-display hex
```

displays the characters as hexadecimal numbers of their Unicode Code Points. Once, those files have been generated, the dot utility can be called to produce state machine plots, e.g.

```
> dot -Tsvg -oMyfile.svg MY_MODE.dot
```

generates an SVG file from the file that quex produced 'MY_MODE.dot'. An example of a state machine plot can be viewed in *Sample plot*.

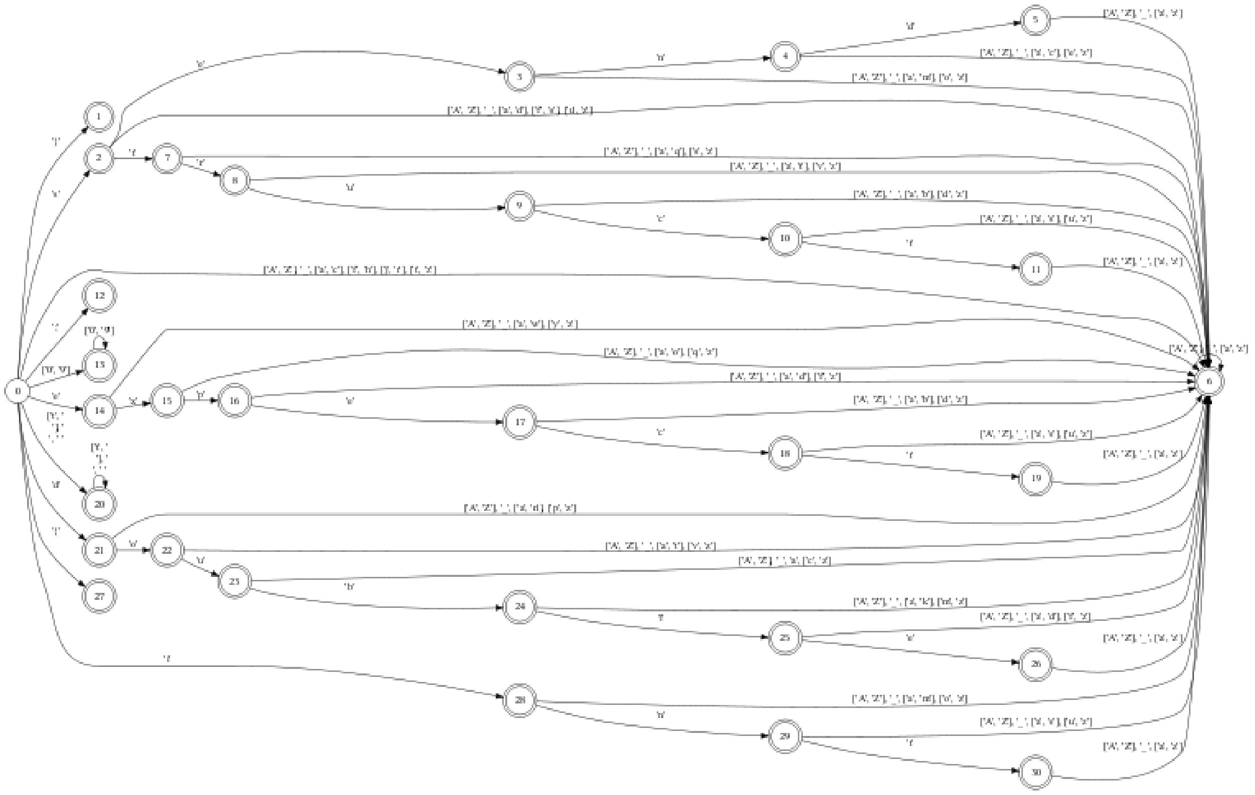


Figure 8.2: Sample plot of a simple state machine.

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*