Contents

Chapter 1

The Buffer

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

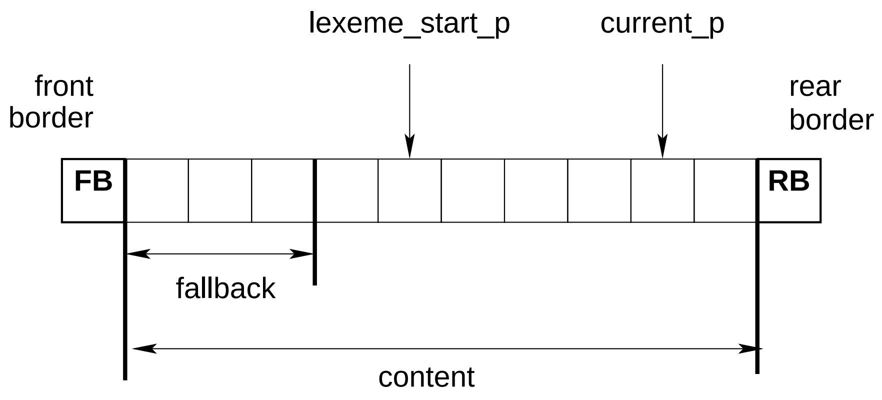

Figure 1.1: Structure of queχ's buffer.

The structure of queχs buffer is shown in figure 1.1. The buffer itself is not more than a continous chunk of memory. It has to places to store border characters. One is located at the front to store 'begin of buffer' or 'begin of file'. The other is located at the end to store 'end of buffer' or 'end of file'. However, the 'end of file' character may come anywhere from the start of the buffer to the end. Details about these limit characters are explained below. Thus the actual buffer size is two elements greater than the actual number of content elements. A 'current' pointer iterates from left to right[1] to go forward. The element to which it points designates the current input to the lexical analyser state machine. If it reaches a border new content has to be loaded. The fallback area allows to prevent immediate loads in the opposite directions if one requests to go back immediately after loading.

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

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

1.1 Programming Interface

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

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

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

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

1.2 Mechanisms

The buffer implementation in queχ takes advantage of the mechanism of lexical analysis, and manages to keep the overhead of buffering extremely low. This works as shown in figure 1.2. First of all, some content of the information stream in stored in a fixed chunk of memory, the buffer. During the lexical analysis one basically iterates through this chunk. Whenever a new input character is required to determine a state transition, the pointer to the current character, short the 'current' pointer, is increased and the input is assigned the content to what this pointer points. The buffer limits, begin of file, and end of file is determined through special characters. Thus, if the input that is received equals those, then one touched either a buffer limit, the begin of the file, or the end of the file. The trick here is that the transition checks of any state will drop if those characters occur. In any other case the business can continue as usual.

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

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

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

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

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

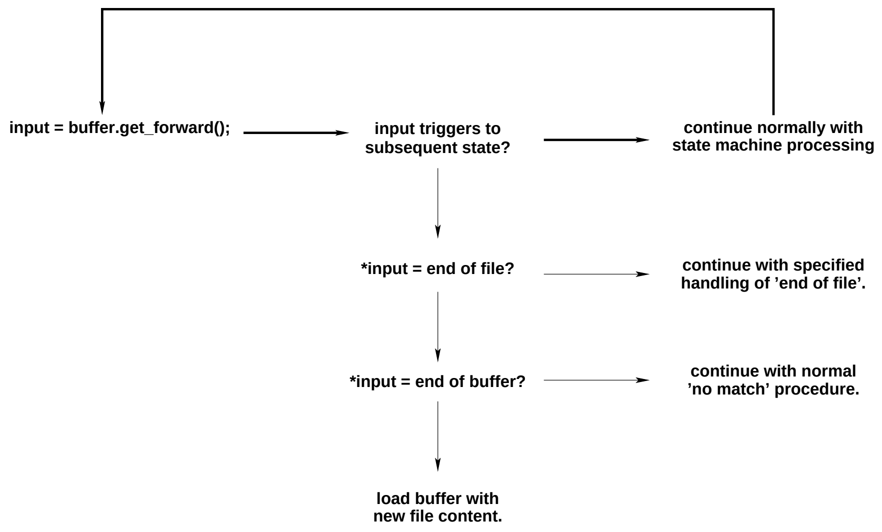
input = buffer.get_forward();
input triggers to subsequent state?
continue normally with state machine processing
*input = end of file?
continue with specified handling of 'end of file'.
*input = end of buffer?
continue with normal 'no match' procedure.
load buffer with new file content.

Figure 1.2: Interaction of buffer and the lexical analyser state machine.

1.3 Loading from File

Figure 1.3 shows the process of loading the buffer with fresh content in forward direction. Loading data from a device into a buffer is very costy. So, it is a good idea to introduce some fallback area where the end of the current buffer content is stored at the beginning. This prevents an immediate load backwards as soon as a character is required from before the current position. Also, since the current lexeme may be post-treated by some pattern action, the pointer to the lexeme start also has to lie inside the buffer. The maximum of both values defines the fallback border.

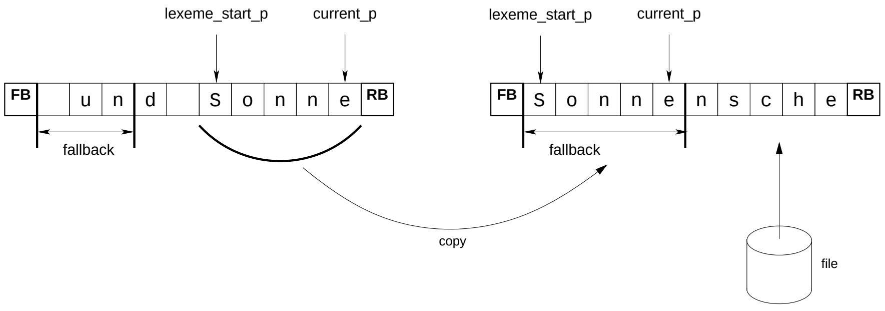

Figure 1.3: Loading new contents 'forwards' from the stream into the buffer.

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

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

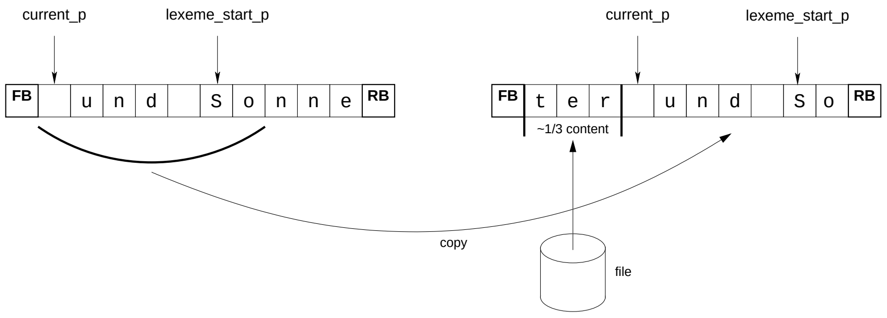

Figure 1.4: Loading new contents 'backwards' from the stream into the buffer.

Figure 1.4 shows the process of backward loading. Unlike loading forward not such a large amount of memory is newly loaded. Note, that backward loading is only necessary if the fallback buffer was not enough. Thus, it implies some 'abuse of design' that is caught and treated seeminglessly. However, a basic assumption about lexical analysis is that the main direction is `forward`. Instead of loading a large bunch of memory before the current position, loading backwards only loads about the first third of the buffer with backwards content from the file. The two thirds of buffer remain intact. Thus, when lexical analysis finally goes forward again, there is still much room before a forward loading is necessary.

1.4 Creation of A Buffer Object

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

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

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

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

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