Building Unicode Scanners with GPLEX

(*Version 1.0.0 November 2008*)

John Gough QUT

November 11, 2008

1 Overview

Gardens Point *LEX* (*gplex*) is a scanner generator which accepts a "*LEX*-like" specification, and produces a *C#* output file. The scanners produced by *gplex* can operate in two modes —

* *Byte Mode*, in which patterns of seven or eight-bit bytes are specified, and the input source is read byte-by-byte. This mode corresponds to the traditional semantics of *LEX*-like scanner generators.

* *Unicode Mode*. In this mode the patterns are specified as regular expressions over the unicode alphabet. The generated scanner matches sequences of codepoints. Traditional *LEX* has no equivalent semantics.

The choice between byte-mode and unicode-mode is made at scanner generation time, either by a command-line option to *gplex*, or an option marker in the specification file.

For unicode mode scanners, the input to the generated scanner must be decoded according to some known encoding scheme. This choice is made at scanner-runtime. Unicode text files with a valid unicode prefix (sometimes called a *Byte-Order-Mark*, "*BOM*") are decoded according to the scheme specified by the prefix. Files without a prefix are interpreted according to a "*fallback codepage*" option. This option may be specified at scanner generation time. The scanner infrastructure also provides methods to allow scanner applications to override the default at scanner runtime, or even to defer choice until after scanning the entire file.

1.1 Gplex Options for Unicode Scanners

The following options of *gplex* are relevant to the unicode features of the tool.

/codepage:*Number*

In the event that an input file does not have a unicode prefix, the scanner will map the bytes of the input file according to the codepage with the specified number. If there is no such codepage, or the codepage is unsuitable, an exception is thrown and processing terminates. For version 1.0 of *gplex* the specified codepage must have the single-byte property[1], or must be one of 1200 (*utf-16*), 1201 (*unicodeFFFE*) or 65001 (*utf-8*).

[1]An encoding has the single byte property if each byte of the input file delivers a unicode codepoint to the scanner. For example, all of the iso-8859 encodings have this property. For this version of *gplex* input in multi-byte encodings must use one of the *UTF* formats.

/codepage:*Name*

In the event that an input file does not have a unicode prefix, the scanner will map the bytes of the input file according to the codepage with the specified name. If there is no such codepage, or the codepage is unsuitable, an exception is thrown and processing terminates. For version 1.0 of *gplex* the specified codepage must have the single-byte property, or must be one of "*utf-16*" (Little-Endian Unicode), "*unicodeFFFE*" (Big-Endian Unicode) or "*utf-8*".

/codepage:default

In the event that an input file does not have a unicode prefix, the scanner will map the bytes of the input file according to the default codepage of the host machine. This codepage must have the single-byte property. This option is the default for unicode scanners.

/codepage:guess

In the event that an input file does not have a unicode prefix, the scanner will rapidly scan the file to see if it contains any byte sequences that suggest that the file is either *utf-8* or that it uses some kind of single-byte codepage. On the basis of this scan result the scanner will use either the default codepage on the host machine, or interpret the input as a *utf-8* file. See Section 2.5 for more detail.

/codepage:raw

In the event that an input file does not have a unicode prefix, the scanner will use the uninterpreted bytes of the input file. In effect, only codepoints from 0 to u+00ff will be delivered to the scanner.

/unicode

By default *gplex* generates byte-mode scanners that use 8-bit characters, and read input files byte-by-byte. This option allows for unicode-capable scanners to be created. Using this option implicitly uses character classes.

/nounicode

This negated form of the /unicode option is the default for *gplex*.

/UTF8default

This option is deprecated. It will continue to be supported in version 1.0. However, the same effect can be obtained by using "`/codepage:utf-8`".

/noUTF8default

This option is deprecated. It will continue to be supported in version 1.0. However, the same effect can be obtained by using "`/codepage:raw`".

1.2 Unicode Options for Byte-Mode Scanners

Most of the unicode options for *gplex* have no effect when a byte-mode scanner is being generated. However, the codepage options have a special rôle in the special case of character set predicates.

The available character set predicates in *gplex* are those supplied by the *.NET* base class libraries. These predicates are specified over the unicode character set. On a machine with that uses a single-byte codepage *gplex* must know what that codepage is, in order to correctly construct character sets such as "`[:IsPunctuation:]`".

The available options are —

/codepage:*Number*

If a character set predicate is used, the set will include all the byte values which correspond in the codepage mapping to unicode characters for which the predicate is true. The nominated codepage must have the single-byte property.

/codepage:*Name*

If a character set predicate is used, the set will include all the byte values which correspond in the codepage mapping to unicode characters for which the predicate is true. The nominated codepage must have the single-byte property.

/codepage:default

If a character set predicate is used, the set will include all the byte values which correspond to unicode characters for which the predicate is true. In this case the mapping from byte values to unicode characters is performed according to the default code page of the *gplex* host machine. The default codepage must have the single-byte property.

/codepage:raw

If a character set predicate is used, the set will include all the byte values which numerically correspond to unicode codepoints for which the predicate is true.

Caution

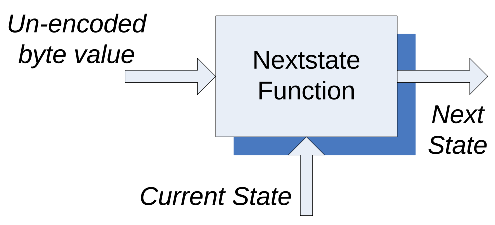
Character set predicates should be used with caution in byte-mode scanners. The potential issue is that the byte-mode character sets are computed at scanner generation time. Thus, unlike the case of unicode scanners, the codepage of the scanner host machine must be known at scanner generation time rather than at scanner runtime (see also section 2.2).

2 Specifying Scanners

The scanning engine that *gplex* produces is a finite state automaton (*FSA*)[2] This *FSA* deals with codepoints from either the *ASCII* or *Unicode* alphabet. Byte-mode scanners have the conceptual form shown in Figure 1. The un-encoded byte values of the input

[2](*Note for the picky reader*) Well, the scanner is *usually* an *FSA*. However, the use of the "*/stack*" option allows state information to be stacked so that in practice such *gplex*-generated recognizers can have the power of a push-down automaton.

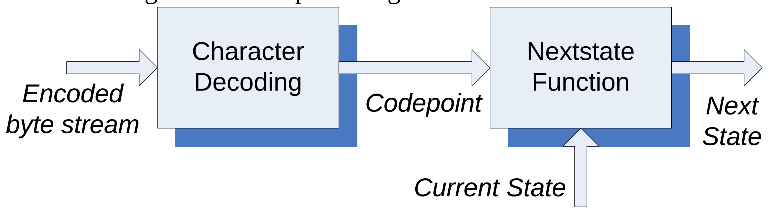
Figure 1: Conceptual diagram of byte-mode scanner



are used by the “next-state” function to compute the next state of the automaton.

In the unicode case the sequence of input values may come from a string of *System.Char* values, or from a file. Unicode codepoints need 21-bits to encode, so some interpretation of the input is required for either input form. The conceptual form of the scanner is shown in Figure 2 for file input. The corresponding diagram for *string* input

Figure 2: Conceptual diagram of unicode scanner



differs only in that the input is a sequence of *System.Char* values rather than bytes.

For *gplex* version 1.0 the scanner that *gplex* uses to read its own input (the “`*.lex`” file) operates in byte-mode. Nevertheless, the input byte-mode text may specify either a byte-mode scanner as *gplex*-output, or a unicode-mode scanner as output.

Because of the choice of byte-mode for *gplex* input, literal characters in specifications denote precisely the codepoint that represents that character in the input file. Characters that cannot denote themselves in character literals must be specified by character escapes of various kinds.

In this section we consider the way in which byte-mode scanners and unicode scanners respectively are specified while complying with this constraint. Issues of portability of specifications and generated scanners across globalization boundaries are also discussed.

2.1 Byte Mode Scanners

In byte-mode scanners, the only valid codepoints are in the range from ‘\0’ to ‘\xff’. When *gplex* input specifies a byte-mode scanner, character literals in regular expression patterns may be: literals such as ‘a’, one of the traditional control code escapes such as ‘\0’ or ‘\n’, or any other of the allowed numeric escapes.

The allowed numeric escapes are octal escapes ‘*ddd*’, where the *d* are octal digits; hexadecimal escapes ‘\x*hh*’, where the *h* are hexadecimal digits; unicode escapes ‘\u*hhhh*’ and ‘\U*hhhhhhhh*’, where the *h* are hexadecimal digits. If the specification is for a byte-mode scanner the numerical value of any character literal must be less than 256, or an error occurs.

It is important to see that even for byte-mode scanners, these choices lead to certain kinds of portability issues across cultures. Let us examine an example.

Suppose that a specification file is being prepared with an editing system that uses the Western European (Windows) code page 1252. In this case the user can enter a literal character ‘ß’, the *sharp s* character. This character will be represented by a byte 0xdf in the specification file. The byte-mode scanner which is generated will treat any 0xdf byte as corresponding to this character. To be perfectly clear: when the specification is viewed in an editor it may display a *sharp s* but *gplex* neither knows nor cares about how characters are displayed on the screen. When *gplex* reads its input it will find a 0xdf byte, and will interpret it as meaning “a byte with value 0xdf”.

Suppose now that the same specification is viewed on a machine which uses the Greek (Windows) code page 1253. In this case the same character literal will be displayed as the character ί, *small letter iota with tonos*. Nevertheless, the scanner that *gplex* generates on the second machine will be identical to the scanner generated on the first machine.

Thus the choice of a byte-mode scanner for *gplex*-input achieves portability in the sense that any specification that does not use character predicates will generate a precisely identical scanner on every host machine. However, it is unclear whether, in general, the *meaning* of the patterns will be preserved across such boundaries.

In summary, byte-mode scanners handle the full 8-bit character set, but different codepages may ascribe different meanings to character literals for the upper 128 characters. Byte-mode scanners are inherently non-portable across cultures.

2.2 Character Predicates in Byte-Mode Scanners

Scanner specifications may use character set literals in the familiar form, the archetypical example of which is “`[a-zA-Z]`”. In *gplex* character set definitions may also use character predicates, such as “`[[:IsLetter:]]`”. In traditional *LEX*, the names of the character predicates are those available in “`libc`”. In *gplex* the available predicates are from the *.NET* base class library, and apply to unicode codepoints.

Consider the following example: a byte-mode specification declares a character set

```
PunctuationChars [[:IsPunctuation:]]
```

Now, the base class library function allows us to easily generate a set of *unicode* codepoints p such that the static predicate

```
Char.IsPunctuation(p);
```

returns true. Sadly, this is not quite what we need for a byte-mode scanner. Recall that byte-mode scanners operate on uninterpreted byte-values, as shown in figure 1. What we need is a set of byte-values v such that

```
Char.IsPunctuation(Map(v));
```

returns true, for the mapping *Map* defined by some codepage.

For example, in the Western European (Windows) character set the ellipsis character ‘. . .’ is byte 0x85. The ellipsis is a perfectly good punctuation character, however

```
Char.IsPunctuation((char)0x85);
```

is false! The problem is that the ellipsis character is unicode codepoint u+2026, while unicode codepoint u+0085 is the “newline” control character *NEL*. All of the characters of the iso-8859 encodings that occupy the byte-values from 0x80 to 0x9f correspond to unicode characters from elsewhere in the space.

The character set “`[:IsLetter:]`” provides another example. For a byte-mode scanner using the Western European codepage 1252, this set will contain 126 members. The same set has only 123 members in codepage 1253. In the uninterpreted, raw case the set has only 121 members.

Nevertheless, it is permissible to generate character sets using character predicates in the byte-mode case. When this is done, the user may specify the codepage that maps between the byte-values that the generated scanner reads, and the unicode codepoints to which they correspond.

If no codepage is specified, the mapping is taken from the default codepage of the *machine on which gplex is running*. This poses no problem if the machine on which the generated scanner will run has the same culture settings as the generating machine, or if the codepage of the scanner host is known with certainty at scanner generation time. Other cases may lack portability.

2.3 Unicode Mode Scanners

The unicode standard ascribes unique 21-bit *codepoints* for every defined character[3]. Thus, if we want to recognize *both* the ‘ß’ character *and* the ‘ί’ character then we must use a unicode scanner. In unicode ß has codepoint u+00df, while ί has codepoint u+03af.

In unicode-mode scanners, the valid codepoints are in the range from u+0000 to u+10ffff. As was the case for byte-mode, character literals in the specification file may be literals such as ‘a’, one of the traditional control code escapes such as ‘\0’, or ‘\n’, or any other of the allowed numeric escapes.

The allowed numeric escapes are just as for the byte-mode case: octal escapes ‘*ddd*’, where the *d* are octal digits; hexadecimal escapes ‘\x*hh*’, where the *h* are hexadecimal digits; unicode escapes ‘\u*hhhh*’ and ‘\U*hhhhhhhh*’, where the *h* are hexadecimal digits. However, in this case the unicode escapes may evaluate to a codepoint up to the limit of 0x10ffff.

Since unicode scanners deal with unicode codepoints, it is best practise to always use unicode escapes to denote characters beyond the (7-bit) *ASCII* boundary. Thus our two example characters should be denoted ‘\u00df’ and ‘\u03af’ respectively.

Reading Scanner Input

The automata of unicode scanners deal only with unicode codepoints. Thus the scanners that *gplex* produces must generate the functionality inside the left-hand box in figure 2. This *Character Decoding* function maps the bytes of the input file (or the characters of a string) into the codepoints that the scanner automaton consumes.

In the best of all worlds, the problem is simple. If the scanner’s input file is encoded using “little-endian” utf-16 our two example characters will each take two bytes. The ß character will be denoted by two bytes {0xdf, 0x00}, while the ί character will be denoted by the two bytes {0xaf, 0x03}.

If the scanner’s input file is encoded using utf-8 our two example characters will again take two bytes each. The ß character will be denoted by two bytes {0xc3, 0x9f}, while the ί character will be denoted by the two bytes {0xce, 0x9f}.

[3]This is not the same as saying that every character has an unambiguous meaning. For example, in the *CJK compatability* region of unicode ideograms with different meanings in Chinese, Japanese and Korean may share the same codepoint provided they share the same graphical representation.

In both of these cases, the files should begin with a prefix which unambiguously indicates the format of the file. If a file is opened which does not start with a prefix then there is a problem.

Consider the case of a byte file prepared using either codepage 1252 or codepage 1253. Of course, such a file cannot contain both ß and ί characters, since both of these are denoted by the same byte value 0xdf. The question is — if such a file is being scanned and a 0xdf byte is found — what codepoint should be delivered to the automaton[4]? Note that unlike the "utf-with-prefix" cases there is no certain way to know what codepage a file was encoded with, and hence no certain way to know what decoding to use.

At the time that *gplex* generates a scanner, either a command line "`/codepage:`" option or a "`%option`" declaration in the specification may specify the fall-back codepage that should be used if an input file has no unicode prefix. A common choice is "`/codepage:default`", which treats files without prefix as 8-bit byte files encode according to the default codepage on the host machine. This is a logical choice when the input files are prepared in the same culture as the scanner host machine. In fact, this is the fallback that *gplex* uses in the event that no codepage option is specified.

The other common choice is "`/codepage:utf-8`", which treats files without prefix as utf-8 files anyway.

If it is known for certain that input files will have been generated using a codepage that is different to the host machine, then that known codepage may be explicitly specified as the fallback. Note however, that this fallback will be applied to *every* file that the scanner encounters that does not have a prefix. In such cases it is more useful to allow the fallback to be specified to the scanner application on a file-by-file basis. How to do this is the subject of the next section.

What may we conclude from this discussion?

* Use unicode scanners for global portability whenever possible.

* Input files to unicode scanners should always be in one of the utf formats, whenever that is possible. Always place a prefix on such files.

* Consider using the default fallback to the host-machine codepage unless it is known at scanner generation time that input files will originate from another culture.

* Applications that use *gplex* scanners should allow users to override the codepage fallback when it is known that a particular input file originates from another culture.

2.4 Overriding the Codepage Fallback at Application Runtime

The fallback codepage that is specified at scanner generation time is hardwired into the code of the generated scanner. However, an application that uses a *gplex* scanner may need to have its fallback codepage changed for a particular input file when the encoding of that file is known.

Scanners generated by *gplex* implement a static method of the stream buffer class with the following signature —

```
public static int GetCodePage(string command);
```

[4]We have discussed only two possibilities here. Other codepages will give many additional meanings to the same 0xdf byte value.

This method takes a string argument, which is a codepage-setting command from the calling application. If the command begins with the string “*codepage:*” this prefix is removed, and the remaining string is converted to a codepage index. The command may specify either a codepage name or a number, or the special values “raw”, “default” or “guess”. Raw denotes no interpretation of the raw byte values, while “default” decodes according to the default codepage of the host machine. Finally, “guess” attempts to determine the codepage from the byte-patterns in the file. These semantics are the same as the */codepage:* option of *gplex*, which indeed invokes this same method.

There are two constructors for the scanner objects in the scanner that *gplex* generates. One takes a stream object as its sole argument, while the other takes a stream object and a command string denoting the fallback codepage. The second constructor passes the string argument to *GetCodePage*, and then sends the resulting integer to the appropriate call of *SetSource*[5]. Alternatively, the application may directly call *SetSource* itself, as shown below.

An application program that wishes to set the fallback codepage of its scanner on a file-by-file basis should follow the example of the schema in Figure 3. If the

Figure 3: Using the *GetCodePage* method

```
string codePageArg = null;
    ...
    // Process the codepage argument
    if (arg.StartsWith("codepage:"))
        codePageArg = arg;
    ...
    // Instantiate a scanner
    FileStream file = new FileStream(...);
    Scanner scnr = new Scanner();
    if (codePageArg != null) {
        int cp = Scanner.StreamBuff.GetCodePage(codePageArg);
        scnr.SetSource(file, cp);
    }
    else // Use machine default codepage, arg1 = 0
        scnr.SetSource(file, 0);
    ...
```

application passes multiple input files to the same scanner instance, then an appropriate value for the fallback codepage should be passed to each subsequent call of *SetSource* in the same way as shown in the figure.

2.5 Adaptively Setting the Codepage

There are occasions in which it is not possible to predict the codepage of input files that do not have a unicode prefix. This is the case, for example, with programming language scanners that deal with input that has been generated by a variety of different text editing systems.

[5]In the case of byte-mode scanners there is no fallback codepage, so the two-arg constructor simply ignores its second argument.

In such cases, if an input file has no prefix, a last resort is to scan the input file to see if it contains some byte value sequences that unambiguously indicate the codepage. In principle the problem has no exact solution, so we may only hope to make a correct choice in the majority of cases.

Version 1.0.0 of *gplex* contains code to automate this decision process. In this first release the decision is only made between the *utf-8* codepage and the default codepage of the host machine. The option is activated either by using the command line option "`/codepage:guess`", or by arranging for the host application to pass this command to the *GetCodePage* method.

The code that implements the decision procedure scans the whole file. The "*guesser*" is a very lean example of a *gplex*-generated byte-mode *FSA*. This *FSA* searches for byte sequences that correspond to well-formed two, three and four-byte utf-8 codepoints. The automaton forms a weighted sum of such occurrences. The automaton also counts bytes with values greater than 128 ("*high-bytes*") which do not form part of any legal utf-8 codepoint.

If a file has an encoding with the single-byte property there should be many more high-bytes than legal utf-8 sequences, since the probability of random high-bytes forming legal utf-8 sequences is very low. In this event the host machine codepage is chosen.

Conversely if a file is encoded in utf-8 then there should be many multi-byte utf-8 patterns, and a zero high-byte count. In this event a utf-8 decoder is chosen for the scanner.

Note that it is possible to deliberately construct an input that tricks the guesser into a wrong call. Nevertheless, the statistical likelyhood of this occurring without deliberation is very small.

There is also a processing cost involved in scanning the input file twice. However, the auxiliary scanner is very simple, so the extra processing time will generally be significantly less than the runtime of the final scanner.

3 Input Buffers

Whenever a scanner object is created, an input buffer holds the current input text. There are seven concrete implementations of the abstract *ScanBuff* class. Two are used for string input, and five for file input.

The *ScanBuff* class in Figure 4 is the abstract base class of the stream and string

Figure 4: Features of the *ScanBuff* Class

```
public abstract class ScanBuff {
    ...
    public abstract int  Pos { get; set; }
    public abstract int  ReadPos { get; }
    public abstract int  Read();
    public abstract int  Peek();
    public abstract string  GetString(int begin, int end);
}
```

buffers of the generated scanners. The important public features of this class are the property that allows setting and querying of the buffer position, and the creation of strings corresponding to all the text between given buffer positions. The *Pos* property returns the current position of the underlying input stream. The *ReadPos* property returns the stream position of the (start of the) "*current character*". For some kinds of text streams this is not simply related to the current *Pos* value.

The method *Read* returns an integer corresponding to the codepoint of the next character, and advances the input position by one or more input elements. The method *Peek* returns an integer corresponding to the codepoint of the next character, but does not advance the input position.

New buffers are created by calling one of the *SetSource* methods of the scanner class. The signatures of these methods are shown in Figure 5.

Figure 5: Signatures of *SetSource* methods

```
// Create a string buffer and attach to the scanner. Start reading from offset ofst
public void SetSource(string source, int ofst);

// Create a line buffer from a list of strings, and attach to the scanner
public void SetSource(IList<string> source);

// Create a stream buffer for a byte-file, and attach to the scanner
public void SetSource(Stream src);

// Create a text buffer for an encoded file, with the specified encoding fallback
public void SetSource(Stream src, int fallbackCodepage);
```

3.1 String Input Buffers

There are two classes for string input: *StringBuff* which holds a single string of input, and *LineBuff* which holds a list of lines.

Scanners that accept string input should always be generated with the */unicode* option. This is because non-unicode scanners will throw an exception if they are passed a codepoint greater than 255. Unless it is possible to guarantee that no input string will contain such a character, the scanner will be unsafe.

The *StringBuff* Class

If the scanner is to receive its input as a single string, the user code passes the input to the first of the *SetSource* methods, together with a starting offset value —

```
public void  SetSource(string s, int ofst);
```

This method will create a buffer object of the *StringBuff* type. Colorizing scanners for *Visual Studio* always use this method.

Buffers of this class consume either one or two characters for each call of *Read*, unless the end of string has been found, in which case the *EOF* value -1 is returned. Two characters are consumed if they form a surrogate pair, and the caller receives a single codepoint which in this case will be greater than u+ffff. Calls directly or indirectly to *GetString* that contain surrogate pairs will leave the pair as two characters.

The *LineBuff* Class

An alternative string interface uses a data structure that implements the *IList*<`string`> interface —

```
        public void  SetSource(IList<string> list);
```

This method will create a buffer object of the *LineBuff* type. It is assumed that each string in the list has been extracted by a method like *ReadLine* that will remove the end-of-line marker. When the end of each string is reached the buffer *Read* method will report a '`\n`' character, for consistency with the other buffer classes. In the case that tokens extend over multiple strings in the list *buffer.GetString* will return a string with embedded end of line characters.

3.2 File Input Buffers

There are five flavors of file buffers —

* *StreamBuff*. The buffer for a byte file, which reads one byte at a time. It is used for non-unicode scanners, and by unicode scanners for files that have no prefix when the fall-back "`/codepage:raw`" is specified.

* *CodePageBuff*. The buffer for a text file which is encoded according to some specified codepage. This is used by unicode scanners for files that have no prefix, and a single-byte fallback codepage has been specified.

* *TextBuff*. The buffer for a text file encoded according to the *UTF-8* form. This is used by unicode scanners for files with a utf-8 prefix, of for files without a prefix if "`/codepage:utf-8`" has been specified.

* *BigEndTextBuff*. The buffer for a text file encoded according to the "big-endian" *UTF-16* form. This is used by unicode scanners for files with a utf-16 prefix, or for files without a prefix if "`/codepage:unicodeFFFE`" has been specified.

* *LittleEndTextBuff*. The buffer for a text file encoded according to the "little-endian" *UTF-16* form. This is used by unicode scanners for files with a utf-16 prefix, of for files without a prefix if "`/codepage:utf-16`" has been specified.

For all forms of file input, the scanner opens a file stream with code equivalent to the following —

```
   FileStream file = new FileStream(name, FileMode.Open);
   Scanner scnr = new Scanner();
   scnr.SetSource(file, ...);
```

The constructor code of the *Scanner* object that is emitted by *gplex* is customized according to the */unicode* option. If the unicode option is not in force a scanner is generated with a *StreamBuff* buffer object. In this case the single-argument version of *SetSource* (third method in figure 5) will be called. This buffer reads input byte-by-byte, and the resulting scanner will match patterns of 8-bit bytes.

If the unicode option is in force, the two-argument overload of *SetSource* (last method in figure 5) will be called. This version of *SetSource* reads the first few bytes of the stream in an attempt to find a valid unicode prefix.

If a valid prefix is found corresponding to a *UTF-8* file, or to one or other *UTF-16* file formats, then a corresponding unicode text buffer object is created. If no prefix is found, then the form of buffer is determined by the "`/codepage:`" option. In the event that no codepage option is in force a *CodePageBuff* will be created, and loaded up with the default codepage for the host machine.