The GPLEX Scanner Generator

(*Version 1.2.0 May 2012*)

John Gough QUT

May 11, 2012

About GPLEX

Gardens Point *LEX* (*gplex*) generates scanners based on finite state automata. The generated automata have the number of states minimized by default, and have a large number of options for table compression. The default compression scheme is chosen depending on the input alphabet cardinality, and almost always gives a reasonable result. However a large number of options are available for the user to tune the behavior if necessary.

The tool implements many of the *FLEX* extensions, including such things as start-state stacks.

The generated scanners are designed to interface cleanly with bottom-up parsers generated by Gardens Point Parser Generator (*gppg*). However, *gplex*-generated scanners have been successfully used with both handwritten parsers and with parsers generated by *COCO/R*.

This Version at a Glance

Error messages are now in *MSBuild*-friendly format to interwork better with *Visual Studio*. This is the default behavior, with a new option allowing legacy format messages to the console.

This version changes the default file encoding to Unicode, with a fallback to raw (uninterpreted bytes) if there is no *BOM* prefix.

New optional code may be used to allow *gppg* parsers to push back wrapped symbols to a *gplex* scanner.

Generated scanners suppress several *CSC* warning messages.

Contents

List of Figures

Part I
Introduction to GPLEX

1 Overview

This paper is the documentation for the *gplex* scanner generator.

Gardens Point *LEX* (*gplex*) is a scanner generator which accepts a "*LEX*-like" specification, and produces a *C#* output file. The implementation shares neither code nor algorithms with previous similar programs. The tool does not attempt to implement the whole of the *POSIX* specification for *LEX*, however the program moves beyond *LEX* in some areas, such as support for unicode.

The scanners produce by *gplex* are thread safe, in that all scanner state is carried within the scanner instance. The variables that are global in traditional *LEX* are instance variables of the scanner object. Most are accessed through properties which expose only a getter.

The implementation of *gplex* makes heavy use of the facilities of the 2.0 version of the Common Language Runtime (*CLR*). There is no prospect of making it run on earlier versions of the framework.

There are two main ways in which *gplex* is used. In the most common case the scanner implements or extends certain types that are defined by the parser on whose behalf it works. Scanners may also be produced that are independent of any parser, and perform pattern matching on character streams. In this "*stand-alone*" case the *gplex* tool inserts the required supertype definitions into the scanner source file.

The code of the scanner derives from three sources. There is invariant code which defines the class structure of the scanner, the machinery of the pattern recognition engine, and the decoding and buffering of the input stream. These parts are defined in a "*frame*" file and a "*buffers*" file each of which is an embedded resource of the *gplex* executable.

The tables which define the finite state machine that performs pattern recognition, and the semantic actions that are invoked when each pattern is recognized are interleaved with the code of the frame file. These tables are created by *gplex* from the user-specified "`*.lex`" input file.

Finally, user-specified code may be embedded in the input file. All such code is inserted in the main scanner class definition, as is explained in more detail in section 3.2. Since the generated scanner class is declared `partial` it is also possible for the user to specify code for the scanner class in a *C#* file separate from the *LEX* specification.

If you would prefer to begin by reviewing the input file format, then go directly to Part II of this document.

1.1 Typical Usage

A simple, typical application using a *gplex* scanner consists of two parts. A parser is constructed using *gppg* invoked with the /*gplex* option, and a scanner is constructed using *gplex*. The parser object always has a property "*Scanner*" of *AbstractScanner* type imported from the *QUT.Gppg* namespace (see figure 3). The scanner specification file will include the line —

```
%using ParserNamespace
```

where *ParserNamespace* is the namespace of the parser module defined in the parser specification. The *Main* method of the application will open an input stream, construct a scanner and a parser object using code similar to the snippet in Figure 1.

Figure 1: Typical Main Program Structure

```
static void  Main(string[] args)
{
    Stream file;
    // parse input args, and open input file
    Scanner scanner = new Scanner(file);
    Parser parser = new Parser(scanner);
    parser.Parse();
    // and so on ...
}
```

For simple applications the parser and scanner may interleave their respective error messages on the console stream. However when error messages need to be buffered for later reporting and listing-generation the scanner and parser need to each hold a reference to some shared error handler object. If we assume that the scanner has a field named "`yyhdlr`" to hold this reference, the body of the main method could resemble Figure 2.

Figure 2: Main with Error Handler

```
    ErrorHandler handler = new ErrorHandler();
    Scanner scanner = new Scanner(file);
    Parser parser = new Parser(scanner, handler);
    scanner.yyhdlr = parser.handler; // share handler ref.
    parser.Parse();
    // and so on ...
```

1.2 The Interfaces

All of the code of the scanner is defined within a single class "*Scanner*" inside the user-specified namespace. All user-specified code in the input specification is copied into the body of this class. The invariant buffering code defines string and file buffering classes, and allows characters to be decoded by any of the encodings supported by the *.NET* framework. For more detail on the buffering options, see section 3.3.

For the user of *gplex* there are several separate views of the facilities provided by the scanner module. First, there are the facilities that are visible to the parser and the rest of the application program. These include calls that create new scanner instances, attach input texts to the scanner, invoke token recognition, and retrieve position and token-kind information.

Next, there are the facilities that are visible to the semantic action code and other user-specified code embedded in the specification file. These include properties of the current token, and facilities for accessing the input buffer.

Finally, there are facilities that are accessible to the error reporting mechanisms that are shared between the scanner and parser.

Each of these views of the scanner interface are described in turn. The special case of stand-alone scanners is treated in section 3.6.

The Parser Interface

The parser "interface" is that required by the *YACC*-like parsers generated by the Gardens Point Parser Generator (*gppg*) tool. Figure 3 shows the signatures. This abstract

Figure 3: Scanner Interface of *GPPG*

```
public abstract class AbstractScanner<TValue, TSpan>
    where TSpan :  IMerge<TSpan>
{
    public TValue yylval;
    public virtual TSpan yylloc {
        get { return default(TSpan); }
        set { /* skip */ }
    }
    public abstract int yylex();
    public virtual void yyerror(string msg,
                                params object[] args) {}
}
```

base class defines the *API* required by the runtime component of *gppg*, the library *ShiftReduceParser.dll*. The semantic actions of the generated parser may use the richer *API* of the concrete *Scanner* class (Figure 5), but the parsing engine needs only *AbstractScanner*.

AbstractScanner is a generic class with two type parameters. The first of these, *TValue* is the "*SemanticValueType*" of the tokens of the scanner. If the grammar specification does not define a semantic value type then the type defaults to `int`.

The second generic type parameter, *TSpan*, is the location type that is used to track source locations in the text being parsed. Most applications will either use the parser's default type *QUT.Gppg.LexLocation*, shown in Figure 21, or will not perform location tracking and ignore the field. Section 4.1 has more information on the default location type.

The abstract base class defines two variables through which the scanner passes semantic and location values to the parser. The first, the field "*yylval*", is of whatever "*SemanticValueType*" the parser defines. The second, the property "*yylloc*", is of the chosen location-type.

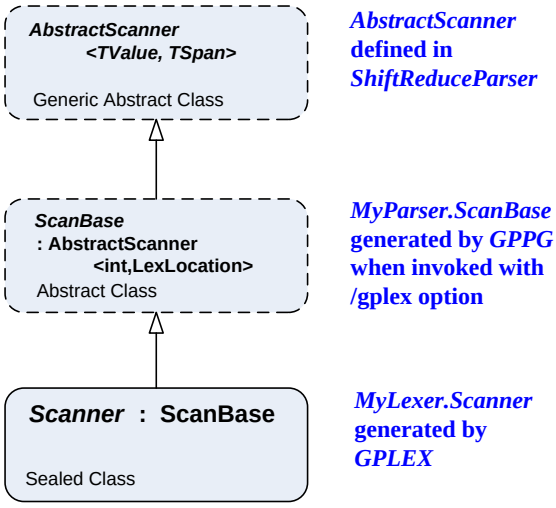
The first method of *AbstractScanner*, *yylex*, returns the ordinal number corresponding to the next token. This is an abstract method, which the code of the frame file overrides.

The second method, the low-level error reporting routine *yyerror*, is called by the parsing engine during error recovery. This method is provided for backward compatability. The default method in the base class is empty. User code in the scanner is able to override the empty *yyerror*. If it does so the default error messages of the shift-reduce parser may be used. Alternatively the low level *yyerror* method may be ignored

completely, and error messages explicitly created by the semantic actions of the parser and scanner. In this case the actions use the *ErrorHandler* class, the *TSpan* location objects, and numeric error codes. This is almost always the preferred approach, since this allows for localization of error messages.

All *gppg*-produced parsers define an abstract "wrapper" class that instantiates the generic *AbstractScanner* class with whatever type arguments are implied by the "`*.y`" file. This wrapper class is named *ScanBase*. The inheritance hierarchy for the case of *gppg* and *gplex* used together is shown in figure 4. For this example it is assumed that

Figure 4: Inheritance hierarchy of the Scanner class



the parser specification has declared "`%namespace MyParser`" and the scanner specification has declared "`%namespace MyLexer`".

Class *ScanBase* always defines a default predicate method *yywrap* which is called whenever an end-of-file is detected in the input. The default method always returns `true`, and may be overridden by the user to support multiple input sources (see Section 3.5).

The scanner class extends *ScanBase* and declares a public buffer field of the *ScanBuff* type, as seen in Figure 5. *ScanBuff* is the abstract base class of the stream and string buffers of the 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 input buffer. The *Read* method of *ScanBuff* returns the next buffer element, but is never called by user code. The method is called by the scanner object's *GetCode* method, which finalizes the character decoding.

Every *gplex*-constructed scanner is either a *byte-mode scanner* or a *unicode-mode scanner*. Byte-mode scanners define two public constructors, while unicode-mode scanners define three. The default "no-arg" constructor creates a scanner instance that initially has no buffer. The buffer may be added later using one of the *SetSource* methods. The other constructors take a *System.IO.Stream* argument, with an optional *code page fallback* argument.

There is a group of four overloaded methods named *SetSource* that attach new buffers to the current scanner instance. The first of these attaches a string buffer to the

Figure 5: Features of the *Scanner* Class

```
// This class defined by gplex
public sealed partial class Scanner :  ScanBase {
    public  ScanBuff buffer;
    public void  SetSource(string s, int ofst);
    ...
}

// This class defined by gppg, when run with the /gplex option
public abstract class ScanBuff {
    public abstract int Read();
    ...
    public abstract int  Pos { get; set; }
    public abstract string  GetString(int begin, int end);
}
```

scanner, and is part of the *IColorScan* interface (see Figure 8). This method provides the only way to pass a string to the scanner.

Scanners that take file input usually have a file attached by the scanner constructor, as shown in Figure 1. However, when the input source is changed *SetSource* will be used. The signatures of the *SetSource* method group are shown in Figure 6.

Figure 6: 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 source);

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

The Internal Scanner *API*

The semantic actions and user-code of the scanner can access all of the features of the *AbstractScanner* and *ScanBase* super types. The frame file provides additional methods shown in Figure 7. The first few of these are *YACC* commonplaces, and report information about the current token. *yyleng*, *yypos* and *yytext* return the length of the current token, the position in the current buffer, and the text of the token. The text is created lazily, avoiding the overhead of an object creation when not required. *yytext* returns an immutable string, unlike the usual array or pointer implementations.

Figure 7: Additional Methods for Scanner Actions

```
public string yytext { get; }  // text of the current token
int yyleng { get; }  // length of the current token
int yypos { get; }  // buffer position at start of token
int yyline { get; }  // line number at start of token
int yycol { get; }  // column number at start of token
void yyless(int n);  // move input position to yypos + n

internal void BEGIN(int next);
internal void ECHO(); // writes yytext to StdOut
internal int YY_START { get; set; } // get and set start condition
```

yyless moves the input pointer backward so that all but the first n characters of the current token are rescanned by the next call of *yylex*.

There is no implementation, in this version, of *yymore*. Instead there is a general facility which allows the buffer position to be read or set within the input stream or string, as the case may be. *ScanBuff.GetString* returns a string holding all text between the two given buffer positions. This is useful for capturing all of the text between the *beginning* of one token and *end* of some later token[1].

The final three methods are only useful within the semantic actions of scanners. The traditional *BEGIN* sets the start condition of the scanner. The start condition is an integer variable held in the scanner instance variable named *currentScOrd*. Because the names of start conditions are visible in the context of the scanner, the *BEGIN* method may be called using the names known from the lex source file, as in "*BEGIN*(*INITIAL*)"[2]. Start conditions are discussed further in Section 8.3.3.

1.2.1 The IColorScan Interface

If the scanner is to be used with the *Visual Studio SDK* as a colorizing scanner for a new language service, then *gppg* is invoked with the */babel* option. In this case, as well as defining the scanner base class, *gppg* also defines the *IColorScan* interface. Figure 8 is this "colorizing scanner" interface. *Visual Studio* passes the source to be scanned to

Figure 8: Interface to the colorizing scanner

```
public interface IColorScan
{
    void SetSource(string source, int offset);
    int GetNext(ref int state, out int start, out int end);
}
```

[1] Note carefully however, that the default buffering implementation only guarantees that the text of the current token will be available. If arbitrary strings from the input are required the */persistBuffer* option must be used.

[2] Note however that these names denote constant **int** values of the scanner class, and must have names that are valid *C#* identifiers, which do not clash with *C#* keywords. This is different to the *POSIX LEX* specification, where such names live in the macro namespace, and may have spellings that include hyphens.

the *SetSource* method, one line at a time. An offset into the string defines the logical starting point of the scan. The *GetNext* method returns an integer representing the recognized token. The set of valid return values for *GetNext* may contain values that the parser will never see. Some token kinds are displayed and colored in an editor that are just whitespace to the parser.

The three arguments returned from the *GetNext* method define the bounds of the recognized token in the source string, and update the state held by the client. In most cases the state will be just the start-condition of the underlying finite state automaton (*FSA*), however there are other possibilities, discussed below.

2 Running the Program

From the command line *gplex* may be executed by the command —

`gplex [`*options*`]`*filename*

If no filename extension is given, the program appends the string "`.lex`" to the given name.

2.1 Gplex Options

This section lists all of the command line options recognized by *gplex*. Options may be preceded by a '–' character instead of the '/' character. All of the following options are recognized by a case-insensitive character matching algorithm.

/babel

With this option the produced scanner class implements the additional interfaces that are required by the *Managed Babel* framework of the *Visual Studio SDK*. This option may also be used with /*noparser*. Note that the Babel scanners may be unsafe unless the /*unicode* option is also used (see section 3.7).

/caseinsensitive

With this option the produced scanner is insensitive to character case. The scanner does not transform the input character sequences so that the *yytext* value for a token will reflect the actual case of the input characters. There are some important limitations in the use of this option in the unicode case. These are discussed Section 3.8.

/check

With this option the automaton is computed, but no output is produced. A listing will still be produced in the case of errors, or if */listing* is specified. This option allows syntactic checks on the input to be performed without producing an output file.

/classes

For almost every *LEX* specification there are groups of characters that always share the same next-state entry. We refer to these groups as "character equivalence classes", or *classes* for short. The number of equivalence classes is typically very much less that the cardinality of the symbol alphabet, so next-state tables indexed on the class are

much smaller than those indexed on the raw character value. There is a small speed penalty for using classes since every character must be mapped to its class before every next-state lookup. This option produces scanners that use classes. Unicode scanners implicitly use this option.

/codePageHelp

The code page option list is sent to the console. Any option that contains the strings "codepage" and either "help" or "?" is equivalent.

/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 code page with the specified number. If there is no such code page known to the runtime library an exception is thrown and processing terminates. Commonly used code pages are 1200 (*utf-16*), 1201 (*unicodeFFFE*) and 65001 (*utf-8*).

/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 code page with the specified name. If there is no such code page an exception is thrown and processing terminates.

/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 code page of the host machine. This option is the default for unicode scanners, if no code page option is specified.

/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 code page. On the basis of this scan result the scanner will use either the default code page on the host machine, or interpret the input as a *utf-8* file. See Section 6.4 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 code points from 0 to u+00ff will be delivered to the scanner.

/errorsToConsole

By default *gplex* generates error messages that are interpreted by Visual Studio. This command generates error messages in the legacy format, in which error messages are sent to the console preceeded by the text of the source line to which they refer.

/frame:*frame-file-path*

Normally *gplex* uses an embedded resource as the frame file. This option allows a nominated file to be used instead of the resource. Using an alternative frame file is likely to be only of interest to *gplex*-developers.

/help

In this case the usage message is produced. "`/?`" is a synonym for "`/help`".

/listing

In this case a listing file is produced, even if there are no errors or warnings issued. If there are errors, the error messages are interleaved in the listing output.

/noCompress

gplex compresses its scanner next-state tables by default. In the case of scanners that use character equivalence classes (see above) it compresses the character class-map by default in the /*unicode* case. This option turns off both compressions. (See Section 4.6 for more detail of compression options.)

/noCompressMap

This option turns off compression of the character equivalence-class map, independent of the compression option in effect for the next-state tables.

/noCompressNext

This option turns off compression of the next-state tables, independent of the compression option in effect for the character equivalence-class map table.

/noEmbedBuffers

By default the code for the buffer classes is enclosed within the scanner namespace in the *gplex* output file. With this option the buffer code is emitted within namespace *QUT.GplexBuffers*, in a file named "`GplexBuffers.cs`". This is useful for applications with multiple scanners which may then share the common buffer definitions.

/noFiles

This option declares that the scanner does not require file input, but reads its input from a string. For suitable cases this reduces the memory footprint of the scanner by omitting all of the file IO classes.

/noMinimize

By default *gplex* performs state minimization on the *DFSA* that it computes. This option disables minimization.

/noParser

By default *gplex* defines a scanner class that conforms to an interface defined in an imported parser module. With this option *gplex* produces a stand-alone scanner that does not rely on any externally defined scanner super-classes.

/noPersistBuffer

By default file-based buffering in *gplex* scanners uses double buffering but does not reclaim buffer space during the scanning of large files. This option turns on reclaiming of buffer space. The option reduces the memory footprint of the scanner on very large input files, but cannot be used for those applications which require *ScanBuff.GetString* to extract strings from the input buffer at arbitrary positions.

/**out:***out-file-path*

Normally *gplex* writes an output *C#* file with the same base-name as the input file. With this option the name and location of the output file may be specified.

/out:–

With this option the generated output is sent to *Console.Out*. If this option is used together with /*verbose* the usual progress information is sent to *Console.Error*.

/parseOnly

With this option the *LEX* file is checked for correctness, but no automaton is computed.

/squeeze

This option specifies that the *gplex* should attempt to produce the smallest possible scanner, even at the expense of runtime speed.

/stack

This option specifies that the scanner should provide for the stacking of start conditions. This option makes available all of the methods described in Section 4.3.

/summary

With this option a summary of information is written to the listing file. This gives statistics of the automaton produced, including information on the number of backtrack states. For each backtrack state a sample character is given that may lead to a backtracking episode. It is the case that if there is even a single backtrack state in the automaton the scanner will run slower, since extra information must be stored during the scan. These diagnostics are discussed further in section 4.5.

/unicode

By default *gplex* produces scanners that use 8-bit characters, and which read input files byte-by-byte. This option allows for unicode-capable scanners to be created. Using this option implicitly uses character equivalence classes. (See Section 3.7 for more detail.)

/utf8default

This option is deprecated. Use "`/codePage:utf-8`" instead. The deprecated "`/noUtf8default`" option is equivalent to "`/codePage:raw`".

/verbose

In this case the program chatters on to the console about progress, detailing the various steps in the execution. It also annotates each table entry in the *C#* automaton file with a shortest string that leads to that state from the associated start state.

/version

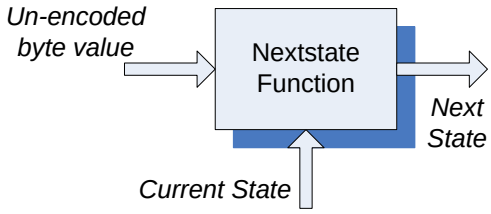
The program sends its characteristic version string to the console.

3 The Generated Scanner

3.1 Byte-Mode and Unicode-Mode

Every scanner generated by *gplex* operates either in *byte-mode*, or in *unicode-mode*. The conceptual form of a byte-mode scanner is shown in Figure 9. In this mode,

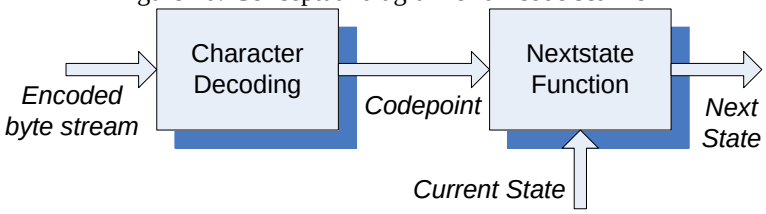
Figure 9: Conceptual diagram of byte-mode scanner



the next state of the scanner automaton is determined by the next-state function from the current input byte and the current state. The bytes of the input stream are used uninterpreted.

In unicode mode the next state of the scanner automaton is determined by the next-state function from the current *unicode code point* and the current state. The sequence of code points may come from a string of *System.Char* values, or from a file. Unicode code-points have 21 significant bits, so some interpretation of the input is required for either string or file input. The conceptual form of the scanner is shown in Figure 10 for file input. The corresponding diagram for *string* input differs only in that the input is a sequence of *System.Char*, rather than a stream of bytes.

Figure 10: Conceptual diagram of unicode scanner



3.2 The Scanner File

The program creates a scanner file which by default is named *filename*.cs where *filename* is the base name of the given source file name.

The file defines a class *Scanner*, belonging to a namespace specified in the lex input file. This class defines the implementation of the interfaces previously described.

The format of the output file is defined by a template file named *gplexx.frame*. User defined and tool generated code is interleaved with this file to produce the final *C#* output file. Since Version 1.1.0 of *gplex* the frame file is an embedded resource in the tool.

The overall structure of the *C#* output file is shown in Figure 11. There are seven

Figure 11: Overall Output File Structure

```
using System;
using System.IO;
using ... ;
user defined using declarations
user defined namespace declaration
{
   public sealed partial class Scanner : ScanBase
   {
      generated constants go here
      user code from definitions goes here
      int state;
      ...              // lots more declarations
      generated tables go here
         ...           // all the other invariant code
      // The scanning engine starts here
      int Scan() {  // Scan is the core of yylex
            optional user supplied prolog
            ...        // invariant code of scanning automaton
            user specified semantic actions
            optional user supplied epilog
      }
      user-supplied body code from "usercode" section
   }
}
Scanners with embedded buffers place buffer code here
```

places where user code may be inserted. These are shown in red in the figure. They

are —

* Optional additional “using” declarations that other user code may require for its proper operation.
* A namespace declaration. This is not optional.
* Arbitrary code from within the definitions section of the lex file. This code typically defines utility methods that the semantic actions will call.
* Optional prolog code in the body of the *Scan* method. This is the main engine of the automaton, so this is the place to declare local variables needed by your semantic actions.
* User-specified semantic actions from the rules section.
* Optional epilog code. This actually sits inside a *finally* clause, so that all exits from the *Scan* method will execute this cleanup code. It might be important to remember that this code executes *after* the semantic action has said “`return`”.
* Finally, the “user code” section of the lex file is copied into the tail of the scanner class. In the case of stand-alone applications this is the place where “`public static void Main`” will appear.

As well as these, there is also all of the generated code inserted into the file. This may include some tens or even hundreds of kilobytes of table initialization. There are actually several different implementations of *Scan* in the frame file. The fastest one is used in the case of lexical specifications that do not require backtracking, and do not have anchored patterns. Other versions are used for every one of the eight possible combinations of backtracking, left-anchored and right-anchored patterns. *gplex* statically determines which version to “`#define`” out.

Note however that the *Scanner* class is marked `partial`. Much of the user code that traditionally clutters up the lex specification can thus be moved into a separate scan-helper file containing a separate part of the class definition.

3.3 Choosing the Input Buffer Class

Scanner code interacts with a buffer object of the *ScanBuff* class. *ScanBuff* is an abstract, public class. The concrete classes derived from *ScanBuff* are all private. Buffers of the derived classes are instantiated by calling a static factory method *ScanBuff.GetBuffer*. There are four overloads of this method, as shown in Figure 12

There are three concrete implementations of the abstract *ScanBuff* class in *gplex*. There are two string input buffer classes and the *BuildBuff* class that handles all file input. The buffer code is invariant, and is either emitted as the separate source file *GplexBuffers.cs* or is embedded in the scanner source file. This behavior is controlled by the */noEmbedBuffers* option flag. The default is that buffer code is embedded.

The File Input Buffers

The left-most function box in figure 10 expands for file input as shown in Figure 13. The tranformation from the input byte stream to the sequence of unicode code points is performed in two steps.

Figure 12: Signatures of *ScanBuff.GetBuffer* methods

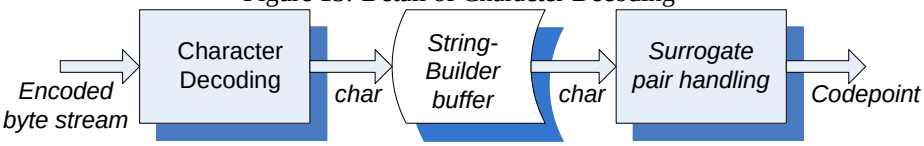
```
// Create a string buffer.
public static ScanBuff GetBuffer(string source);

// Create a line buffer from a list of strings
public static ScanBuff GetBuffer(IList<string> source);

// Create a BuildBuffer for a byte-file
public static ScanBuff GetBuffer(Stream source);

// Create a BuildBuffer for an encoded file, with the specified default encoding
public static ScanBuff GetBuffer(Stream source,
                                 int fallbackCodePage);
```

Figure 13: Detail of Character Decoding



First, the byte sequence in the file is decoded into a sequence of values of the `char` type. The decoding is performed by *System.Globalization* methods from the *.NET* base class libraries.

The sequence of `char` values are held in a buffer of *StringBuilder* class. The character index in this buffer is the value which is used as the abstract "input position" attribute of the recognized tokens.

Finally, the unicode code points are extracted from the buffer by the scanning engine's *GetCode* method. This method interprets any surrogate pairs, and returns an integer value to the automaton.

The structure, as shown in the figure, is invariant for all file input. However the semantics of the two processing blocks are variable. 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 code of the *Scanner* class that is emitted by *gplex* is customized according to the */unicode* option. If the unicode option is not in force the scanner's *ScanBuff* object is instantiated by calling the stream-argument version of *SetSource* (third method in figure 6). In this case the buffer will have an empty character decoder that simply reads single bytes and returns the corresponding `char` value. For the byte-mode case surrogate pairs cannot arise, so the second processing block is empty also.

If the unicode option is in force, the two-argument overload of *SetSource* (last method in figure 6) will be called. This version of *SetSource* reads the first few bytes of the stream in an attempt to find a valid unicode prefix (*BOM*).

If a valid prefix is found corresponding to a *UTF-8* file, or to one or other *UTF-16* file formats, then a corresponding *StreamReader* object is created. If no prefix is

found, then the encoding of the character decoder will be determined from the *gplex* "`/codePage:`" option. In the event that no code page option is in force the default code page for the host machine is chosen.

Note that the choice of alphabet cardinality for the scanner tables is determined at scanner *construction* time, based on the value of the */unicode* option. The choice of buffer implementation, on the other hand, is determined at *runtime*, when the input file is opened. It is thus possible as a corner case that a unicode scanner will open an input file as a byte-file containing only 8-bit characters. The scanner will work correctly, and will also work correctly with input files that contain unicode data in any of the supported formats.

String Input Buffers

If the scanner is to receive its input as one or more string, the user code passes the input to one of the *SetSource* methods. In the case of a single string the input is passed to the method, 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.

An alternative 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.

3.4 How Buffering Works

The scanning engine that *gplex* produces is a finite state automaton (*FSA*)[3] This *FSA* deals with code-points from either the *Byte* or *Unicode* alphabets, as described in section 3.1.

Files containing character data may require as little as one byte to encode a unicode code-point, or as many as four bytes in the worst case of a legal unicode code-point in an *utf-8* file. The *StreamReader* object that decodes the bytes of the file supplies `char` values to the *StringBuilder* buffer structure. Some instances of stream readers encapsulate state, and do not provide a mapping from code point index to file byte-position. As a consequence the index in the *buffer* must be used as a proxy for file position. It follows that encoded input streams are only seekable within the *StringBuilder* buffer structure. For those applications which need to call *GetString* on arbitrary buffer locations, the (default) */persistBuffer* option must be used to prevent reclaiming of buffer space.

Strings containing character data from the full unicode alphabet may require two `char` values to encode a single code-point. Decoders based on `char`-buffers detect surrogate characters and read a second value when needed.

[3](*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.

Finally, it should be noted that textual data exported from the scanner, such as *yytext*, are necessarily of *System.String* type. This means that if the sequence of code-points contains points beyond the 64k boundary (that is, not from the *Basic Multilingual Plane*) those points must be folded back into surrogate pairs in *yytext* and *buffer.GetSource*.

An example

Suppose an input text begins with a character sequence consisting of four unicode characters: '`\u0061`', '`\u00DF`', '`\u03C0`', '`\U000100AA`'. These characters are: lower case letter '*a*', Latin lower case *sharp s* as used in German, Greek lower case *pi*, and the Linear-B ideogram for "*garment*". For all four characters the predicate *IsLetter* is true so the four characters might form a programming language identifier in a suitably permissive language.

Figure 14 shows what this data looks like as a UTF-8 encoded file. Figure 15 shows what the data looks like as a big-endian UTF-16 file. In both cases the file begins with a

Figure 14: Encoding of the example as UTF-8 file

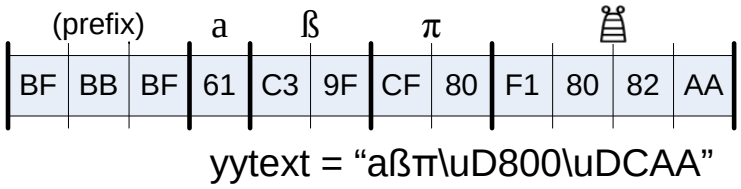
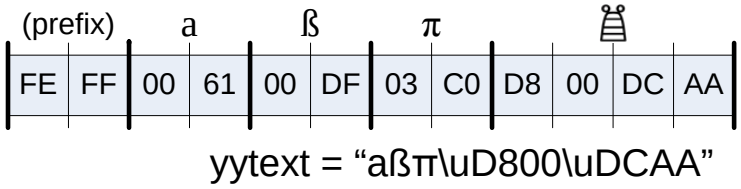


Figure 15: Encoding of the example as big-endian UTF-16 file



representation of the file prefix character `u+feff`. The encoded form of this character occupies three bytes in a UTF-8 file, and two in a UTF-16 file. Reading this prefix allows the scanner to discover in which format the following data is encoded.

The UTF-8 file directly encodes the code-points using a variable-length representation. This example shows all encoded lengths from one to four. The UTF-16 file consists of a sequence of `ushort` values, and thus requires the use of a surrogate pair for the final code-point of the example, since this has more than sixteen significant bits.

In every case the sequence of code-points delivered to the *FSA* will be: `0x61, 0xdf, 0x3c0, 0x100aa`. The *yytext* value returned by the scanner is the same in each case, using the same surrogate pair as in the UTF-16 file. For string input, the input string would be exactly the same as for the big-endian UTF-16 case, but without the prefix code.

Files Without Prefix

The case of text files that do not have a prefix is problematic. What should a unicode scanner do in the case that no prefix is found? In version 1.1.0 of *gplex* the decision is made according to the *fallback code page* setting.

The default setting for the fallback code page of *gplex*-generated scanners is to read the input byte-by-byte, and map the byte-values to unicode using the default code page of the host machine. Other possible fallbacks are to use a specified code page, to use the byte-value uninterpreted ("raw"), or to rapidly scan the input file looking for any characteristic patterns that indicate the encoding.

At scanner generation time the user may specify the required fallback behavior. Generated scanners also contain infrastructure that allows the scanner's host application to override the generation-time default. This overriding may be done on a file-by-file basis.

A complete treatment of the unicode option, including the treatment of fallback code pages is detailed in Part III of this document.

3.5 Multiple Input Sources

There are two common scenarios in which multiple input sources are needed. The first occurs when multiple input sources are treated as though concatenated. Typically, when one input source is exhausted input is taken from the next source in the sequence.

The second scenario occurs in the implementation of "include files" in which a special marker in the current source causes input to be read from an alternative source. At some later stage input may again be read from the remaining text of the original source.

gplex includes facilities to enable the encoding of both of these behaviors, and examples of both are included in Section 6.

Whenever an end-of-input event is found by the scanner, *EOF* processing is invoked. If there is an explicit user action attached to the *EOF*-event for the current start-state then that specified action is executed. If there is no such action, or if the specified action completes without returning a token value, then the default *EOF* action is executed. The default action calls the predicate *yywrap*(). If *yywrap* returns `true` the call to *yylex* will return *Tokens.EOF* thus causing the parser to terminate. If, on the other hand, the predicate returns `false` then scanning continues.

The *ScanBase* class contains a default implementation of *yywrap*, which always returns `true`. Users may override this method in their *Scanner* class. The user-supplied *yywrap* method will determine whether there is further input to process. If so, the method will switch input source and return `false`[4]. If there is no further input, the user-supplied *yywrap* method will simply return `true`.

Chaining Input Texts

When input texts are chained together, the *yywrap* method may be used to manage the buffering of the sequence of sources. A structured way to do this is to place the texts (filenames, or perhaps strings) in a collection, and fetch the enumerator for that collection. Figure 16 is a template for the *yywrap* method. The code for creation and

[4]Beware that returning false *without* replacing the input source is yet another way of making a scanner hang in a loop.

Figure 16: Chaining input texts with *yywrap*

```
protected override bool yywrap() {
    if (enumerator.MoveNext()) { // Is there more input to process?
        SetSource(...) // Choice of four overloads here
        return false
    } else
        return true; // And cause yylex to return EOF
}
```

initialization of the new input buffer depends on the buffer class that is appropriate for the next input text. In the case of a *StringBuff* a call to the first *SetSource* method —

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

does everything that is required.

The case of a file buffer is slightly more complicated. The file stream must be created, and a new buffer allocated and attached to the scanner. For a byte-stream the following code is *almost* sufficient.

```
SetSource(new FileStream(filename, FileMode.Open));
```

Of course, sensible code would open the file within a `try` block to catch any exceptions.

In the unicode case, a call to the fourth method in Figure 6 will create a buffer for an encoded text file.

The BufferContext Class

Switching input sources requires replacement of the *buffer* object of the executing scanner. When a new input source is attached, some associated scanner state variables need to be initialized. The buffer and associated state values form the *BufferContext*. It is values of this type that need to be saved and restored for include-file handling.

There are predefined methods for creating values of *BufferContext* type from the current scanner state, and for setting the scanner state from a supplied *BufferContext* value. The signatures are shown in Figure 17. In cases where include files may be

Figure 17: BufferContext handling methods

```
// Create context from current buffer and scanner state
BufferContext MkBuffCtx() { ...  }

// Restore buffer value and associated state from context
void RestoreBuffCtx(BufferContext value) { ...  }
```

nested, context values are created by *MkBuffCtx* and are then pushed on a stack. Conversely, when a context is to be resumed *RestoreBuffCtx* is called with the popped value as argument.

The *BufferContext* type is used in the same way for *all* types of buffer. Thus it is possible to switch from byte-files to unicode files to string-input in an arbitrary fashion. However, the creation and initialization of objects of the correct buffer types is determined by user code choosing the appropriate overload of *SetSource* to invoke.

Include File Processing

If a program allows arbitrary nesting of include file inclusion then it is necessary to implement a stack of saved *BufferContext* records. Figure 18 is a template for the user code in such a scanner. In this case it is assumed that the pattern matching rules of

Figure 18: Nested include file handling

```
Stack<BufferContext> bStack = new Stack<BufferContext>();

private void TryInclude(string filename) {
    try {
        BufferContext savedCtx = MkBuffCtx();
        SetSource(new FileStream(filename, FileMode.Open));
        bStack.Push(savedCtx);
    } catch { ... }; // Handle any IO exceptions
}

protected override bool yywrap() {
    if (bStack.Count == 0) return true;
    RestoreBuffCtx(bStack.Pop());
    return false;
}
```

the scanner will detect the file-include command and parse the filename. The semantic action of the pattern matcher will then call *TryInclude*.

This template leaves out some of the error checking detail. The complete code of a scanner based around this template is shown in the distributed examples.

3.6 Class Hierarchy

The scanner file produced by *gplex* defines a scanner class that extends an inherited *ScanBase* class. Normally this super class is defined in the parser namespace, as seen in Figure 4. As well as this base class, the scanner relies on several other types from the parser namespace.

The enumeration for the token ordinal values is defined in the *Tokens* enumeration in the parser namespace. Typical scanners also rely on the presence of an *ErrorHandler* class from the parser namespace.

Stand-Alone Scanners

gplex may be used to create stand-alone scanners that operate without an attached parser. There are some examples of such use in the *Examples* section.

The question is: if there is no parser, then where does the code of *gplex* find the definitions of *ScanBase* and the *Tokens* enumeration?

The simple answer is that the *gplexx.frame* file contains minimal definitions of the types required, which are activated by the */noparser* option on the command line or in the lex specification. The user need never see these definitions but, just for the record, Figure 19 shows the code.

Figure 19: Standalone Parser Dummy Code

```
public enum Tokens {
    EOF = 0, maxParseToken = int.MaxValue
    // must have just these two, values are arbitrary
}

public abstract class  ScanBase {
    public abstract int  yylex();
    protected virtual bool yywrap() { return true; }
}
```

Note that mention of *AbstractScanner* is unecessary, and does not appear. If a standalone, colorizing scanner is required, then *gplex* will supply dummy definitions of the required features.

Colorizing Scanners and *maxParseToken*

The scanners produced by *gplex* recognize a distinguished value of the *Tokens* enumeration named "*maxParseToken*". If this value is defined, usually in the *gppg*-input specification, then *yylex* will only return values less than this constant.

This facility is used in colorizing scanners when the scanner has two callers: the token colorizer, which is informed of *all* tokens, and the parser which may choose to ignore such things as comments, line endings and so on.

gplex uses reflection to check if the special value of the enumeration is defined. If no such value is defined the limit is set to `int.MaxValue`.

Colorizing Scanners and *Managed Babel*

Colorizing scanners intended for use by the *Managed Babel* framework of the *Visual Studio SDK* are created by invoking *gplex* with the /*babel* option. In this case the *Scanner* class implements the *IColorScan* interface (see figure 8), and *gplex* supplies an implementation of the interface. The *ScanBase* class also defines two properties for persisting the scanner state at line-ends, so that lines may be colored in arbitrary order.

ScanBase defines the default implementation of a scanner property, *EolState*, that encapsulates the scanner state in an *int32*. The default implementation is to identify *EolState* as the scanner start state, described below. Figure 20 shows the definition in *ScanBase*. *gplex* will supply a final implementation of *CurrentSc* backed by the scanner state field *currentScOrd*, the start state ordinal.

EolState is a virtual property. In a majority of applications the automatically generated implementation of the base class suffices. For example, in the case of multi-line, non-nesting comments it is sufficient for the line-scanner to know that a line starts or ends inside such a comment.

However, for those cases where something more expressive is required the user must override *EolState* so as to specify a mapping between the internal state of the scanner and the *int32* value persisted by *Visual Studio*. For example, in the case of multi-line, possibly nested comments a line-scanner must know how *deep* the comment

Figure 20: The *EolState* property

```
public abstract class ScanBase {
    ...    // Other (non-babel related) ScanBase features
    protected abstract int CurrentSc { get; set; }
    // The currentScOrd value of the scanner will be the backing field for CurrentSc

    public virtual int EolState {
        get { return CurrentSc; }
        set { CurrentSc = value; } }
}
```

nesting is at the start and end of each line. The user-supplied override of *EolState* must thus encode both the *CurrentSc* value *and* a nesting-depth ordinal.

3.7 Unicode Scanners

gplex is able to produce scanners that operate over the whole unicode alphabet. The *LEX* specification itself may be either in a Unicode file in one of the *UTF* formats, or an an 8-bit file.

Specifying a Unicode Scanner

A unicode scanner may be specified either on the command line, or with an option marker in the *LEX* file. Putting the option in the file is always the preferred choice, since the need for the option is a fixed property of the specification. It is an error to include character literals outside the 8-bit range without specifying the /*unicode* option.

Furthermore, the use of the unicode option implies the /*classes* option. It is an error to specify *unicode* and then to attempt to specify /*noClasses*.

Unicode characters are specified by using the usual unicode escape formats \u*xxxx* and \U*xxxxxxxx* where *x* is a hexadecimal digit. Unicode escapes may appear in literal strings, as primitive operands in regular expressions, or in bracket-delimited character class definitions.

Unicode Scanners and the Babel Option

Scanners generated with the *babel* option should always use the *unicode* option also. The reason is that although the *LEX* specification might not use any unicode literals, a non-unicode scanner will throw an exception if it scans a string that contains a character beyond the latin-8 boundary.

Thus it is unsafe to use the babel option without the unicode option unless you can absolutely guarantee that the scanner will never meet a character that is out of bounds. *gplex* will issue a warning if this dangerous combination of options is chosen.

Unicode Scanners and the Input File

Unicode scanners that read from strings use the same *StringBuff* class as do non-unicode scanners. However, unicode scanners that read from filestreams must use

a buffer implementation that reads unicode characters from the underlying byte-file. The current version supports any file encoding for which the *.NET* library supplies a *StreamReader*.

When an scanner object is created with a filestream as argument, and the */unicode* option is in force, the scanner tries to read an encoding prefix from the stream. An appropriate *StreamReader* object is created, and attached to a buffer of the *BuildBuffer* class. If no prefix is found the input stream position is reset to the start of the file and the encoding setting of the stream reader will depend on the *fallback code page* setting.

3.8 Case-Insensitive Scanners

The use of the */caseInsensitive* option causes *gplex* to generate a case-insensitive scanner. In effect, the option ensures that the same accept state will be reached by every case-permuted version of each input that reaches that state.

indexItalyytext When a case-insensitive scanner reads input, it does not transform the input characters. This means that the *yytext* strings will preserve the original casing in the input.

Scanners that rely on a user-supplied helper method for keyword recognition will need to ensure that the helper method performs its own case-normalization.

3.8.1 Limitations

There are a few things to consider if you use the case-insensitive option for a unicode scanner. *gplex* transforms the input specification on a character by character basis using the *.NET ToUpper* and *ToLower* methods. These functions are necessarily culture-sensitive, and *gplex* uses the culture setting of the machine on which it is running. If this is different to the culture setting on which the generated scanner runs then there may be slightly different results. As well, there are examples where case transformation is inherently inaccurate because, for example, a given lower case character transforms into *two* upper case characters.

Characters outside the *basic multilingual plane*, that is, code points that require the use of surrogate pairs of `char` values, do not even get checked for case.

Finally, it should be noted that the construction of character equivalence classes for specifications that include large unicode character sets is computationally intensive. Thus specifications that include sets such as `[[:IdentifierStartCharacter:]]`, with its 90 000+ elements may add several seconds to the scanner generation time. However, the *generated scanner* will run at the same speed as the corresponding case-sensitive version.

3.9 Using *GPLEX* Scanners with Other Parsers

When *gplex*-scanners are used with parsers that offer a different interface to that of *gppg*, some kind of adapter classes may need to be manually generated. For example if a parser is used that is generated by *gppg* but not using the "*/gplex*" command line option, then adaptation is required. In this case the adaptation required is between the raw *AbstractScanner* class provided by *ShiftReduceParser* and the *ScanBase* class expected by *gplex*.

A common design pattern is to have a tool-generated parser that creates a *partial* parser class. In this way most of the user code can be placed in a separate "parse helper" file rather than having to be embedded in the parser specification. The parse

helper part of the partial class may also provide definitions for the expected *ScanBase* class, and mediate between the calls made by the parser and the *API* offered by the scanner.

4 Advanced Topics

4.1 Location Information

Parsers created by *gppg* have default actions to track location information in the input text. Parsers define a class *LexLocation*, that is the default instantiation of the *TSpan* generic type parameter. The default type is simply mapped to the text span format used by *Visual Studio*.

The parsers call the merge method at each reduction, expecting to create a location object that represents an input text span from the start of the first symbol of the production to the end of the last symbol of the production. *gppg* users may substitute other types for the default, provided that they implement a suitable *Merge* method. Section 4.4.2 discusses the non-default alternatives. Figure 21 is the definition of the default class. If a *gplex* scanner ignores the existence of the location type, the parser

Figure 21: Default Location-Information Class

```
public class LexLocation :  IMerge<LexLocation>
{
    public int sLin; // Start line
    public int sCol; // Start column
    public int eLin; // End line
    public int eCol; // End column
    public LexLocation() {};

    public LexLocation(int sl; int sc; int el; int ec)
      { sLin=sl; sCol=sc; eLin=el; eCol=ec; }

    public LexLocation Merge(Lexlocation end) {
      return new LexLocation(sLin,sCol,end.eLin,end.eCol);
    }
}
```

will still be able to access some location information using the *yyline*, *yycol* properties, but the default text span tracking will do nothing[5].

If a *gplex* scanner needs to create location objects for the parser, then it must do it for *all* tokens, otherwise the automatic text-span merging of the parser will not work. The logical place to create the location objects is in the epilog of the scan method. Code after the final rule in the rules section of a lex specification will appear in a `finally` clause in the *Scan* method. For the default location type, the code would simply say —

```
yylloc = new LexLocation(tokLin,tokCol,tokELin,tokECol)
```

[5]The parser will not crash by trying to call *Merge* on a null reference, because the default code is guarded by a null test.

4.2 Applications with Multiple Scanners

Applications that use multiple *gplex*-generated scanners have a variety of possible structures. First of all, there is the option of placing each of the scanners in a separate *.NET* assembly, perhaps shared with the associated parser. It is also possible to place all of the scanners (and parsers) in the same assembly.

There are two mechanisms that may be used to avoid name-clashes between the tool-generated types. The code of each scanner may be placed within a distinct namespace so that the fully qualified names of the types are distinct. Alternatively, the default names of the token, scanner and scanner base classes may be overridden to make the names distinct, even within the same namespace. The declarations that override the default type names are detailed in Section 8.3.2.

A further consideration is the placement of the buffer code. The scanner base class and the generated scanner are specialized by the choice of type-arguments and input grammar. By contrast the buffer code is invariant for all *gplex*-generated scanners[6]. For applications with a single scanner it seems harmless to embed the buffer code in the scanner namespace, and this is the *gplex* default. For applications with multiple scanners it is possible to embed a separate copy of the buffer code within each scanner, at the cost of some code duplication. However, it is probably better to use the *noEmbedBuffers* option and access a single copy of the buffer code from the *QUT.GplexBuffers* namespace.

Scanners in Separate Assemblies

If each scanner is placed in a separate assembly then the issue of name-clashes may be removed from consideration by limiting the visibility of the scanner's *API* classes. A possible structure would be to have the external footprint of each assembly limited to a thin wrapper which initializes and invokes an otherwise inaccessible parser and scanner. In this case the buffer code may be shared from within some other assembly. If the buffer code is embedded, the scanner namespaces must be distinct, since the buffer types are public.

Scanners in the Host Assembly

If all the scanners are placed in the same assembly, assumed to be the same assembly as the host, then the visibility of the scanner classes should be `internal`. As before, the scanner classes are dis-ambiguated either by declaring them within differing namespaces, or by overriding the default naming of types.

If the buffer class definitions are embedded then the scanners *must* reside in different name spaces. Even so, some unnecessary code duplication will occur. This may be eliminated by using the (non-default) *noEmbedBuffers* option.

In summary: to place all scanners in the main application assembly, generate each scanner with the (non-default) `internal` visibility option. Each scanner should be generated with the (non-default) *noEmbedBuffers* option.

An Example: Multiple Scanners in *GPPG*

There are two scanners in the *gppg* code base. One is the main scanner which works on behalf of a *gppg*-generated parser. The other is a specialized "action scanner" which

[6] File buffering is specialized according to the file encoding, but this specialization happens at *scanner runtime*, not at scanner generation time.

is used to process and error-check text spans that contain semantic actions. The action scanner has no associated parser.

Both of the scanners are placed in the same namespace, *QUT.GPGen.Lexers*. The main scanner declares internal visibility but retains the default type-names for the scanner class and the scanner base class. The token enumeration is renamed "*Token*" in both "`gppg.y`" and "`gppg.lex`".

The action scanner renames the scanner class as "*ActionScanner*", the scanner base class as "*ActionBase*", and the token enumeration as "*ActionToken*".

Both scanner specifications use the *noEmbedBuffers* option, with the shared buffer code placed in the *GplexBuffers.cs* source file.

4.3 Stacking Start Conditions

For some applications the use of the standard start condition mechanism is either impossible or inconvenient. The lex definition language itself forms such an example, if you wish to recognize the *C#* tokens as well as the lex tokens. We must have start conditions for the main sections, for the code inside the sections, and for comments inside (and outside) the code.

One approach to handling the start conditions in such cases is to use a *stack* of start conditions, and to push and pop these in semantic actions. *gplex* supports the stacking of start conditions when the "`stack`" command is given, either on the command line, or as an option in the definitions section. This option provides the methods shown in Figure 22. These are normally used together with the standard *BEGIN* method. The

Figure 22: Methods for Manipulating the Start Condition Stack

```
// Clear the start condition stack
internal void yy_clear_stack();

// Push currentScOrd, and set currentScOrd to "state"
internal void yy_push_state(int state);

// Pop start condition stack into currentScOrd
internal int yy_pop_state();

// Fetch top of stack without changing top of stack value
internal int yy_top_state();
```

first method clears the stack. This is useful for initialization, and also for error recovery in the start condition automaton.

The next two methods push and pop the start condition values, while the final method examines the top of stack without affecting the stack pointer. This last is useful for conditional code in semantic actions, which may perform tests such as —

```
if (yy_top_state() == INITIAL) ...
```

Note carefully that the top-of-stack state is not the current start condition, but is the value that will *become* the start condition if "pop" is called.

4.4 Setting *yylval* and *yylloc*

Parsers constructed by tools like *gppg* have built-in mechanisms that allow the semantic values and location values to be evaluated as the input text is parsed. The types of these values are the *TValue* and *TSpan* types that parameterize the generic *ShiftReduceParser* class. These same types are the type parameters of the generic *AbstractScanner* class, from which all *gplex* scanner classes are derived.

The built-in mechanisms of the parser facilitate the computation of *synthesized attributes* of the (virtual) derivation tree that such parsers trace out during parsing. That is to say, the values at each interior node of the tree are computed from the values of that node's immediate children. The starting points of all such calculations are the values of the leaf nodes, which represent the tokens supplied by the scanner.

When the scanner's *yylex* method is called it recognizes a pattern, and returns an integer value corresponding to one of the values of the *Tokens* enumeration. For those applications that need more information than the bare integer the additional information must be passed in the two scanner "variables" *yylval* of type *TValue* and *yylloc* of type *TSpan*.

4.4.1 The *TValue* Type Parameter

Not all parsers need to define a semantic value type. And even for those applications that do need semantic values from the scanner, not all tokens have meaningful attribute information.

Consider the *RealCalc* example distributed with the *gppg* tool. This is a grammar which recognizes infix arithmetic arithmetic. The tokens are *digit*, *letter*, left and right parentheses and the four operators. The operators and the parentheses have no attributes, and do not set *yylval*. Only the lexical categories *digit* and *letter* have semantic values of `int` and `char` type respectively. The parser wants to use the semantic value type to compute the expression value in, so the final semantic value type for this example is a "union" with an integer, character, and floating point double variant.

As described in section 4.1, if an application uses location information it should be produced for *all* tokens. The *yylloc*-setting code is thus naturally placed in the epilog of the scanner's *Scan* method. However, since only a sub-set of tokens have semantic information associated with them the *yylval*-setting code is placed in the semantic actions of those patterns of the lexical specification that need it.

4.4.2 The *TSpan* Type Parameter

The *TSpan* type parameter is used to hold location information, and must implement the *IMerge* interface of Figure 23. In the absence of an explicit declaration of a location

Figure 23: Location types must implement *IMerge*

```
public interface IMerge<YYLTYPE> {
    YYLTYPE Merge(YYLTYPE last);
}
```

type, the default type *LexLocation* is used.

If an application needs a more elaborate location type than the default, then the name of the new type is declared in the parser specification. For example, the parsers in both *gplex* and *gppg* rely on a different location type, *LexSpan*, which includes buffer position values as well as line and column information. The *LexSpan* type has a method which is able to extract all of the input text of a span as a string value. It makes no sense to do this with a *yylloc* value (it would just be a roundabout way of getting *yytext*), but the merged location value of a production right-hand-side will extract *all* of the text of that pattern.

4.5 Scanner Backtracking Information

When the "`/summary`" option is sent to *gplex* the program produces a listing file with information about the produced automaton. This includes the number of start conditions, the number of patterns applying to each condition, the number of *NFSA* states, *DFSA* states, accept states and states that require backup.

Because an automaton that requires backup runs somewhat more slowly, some users may wish to modify the specification to avoid backup. A backup state is a state that is an accept state that contains at least one *out*-transition that leads to a non-accept state. The point is that if the automaton leaves a perfectly good accept state in the hope of finding an even longer match it may fail. When this happens, the automaton must return to the last accept state that it encountered, pushing back the input that was fruitlessly read.

It is sometimes difficult to determine from where in the grammar the backup case arises. When invoked with the "`/summary`" option *gplex* helps by giving an example of a shortest possible string leading to the backup state, and gives an example of the character that leads to a transition to a non-accept state. In many cases there may be many strings of the same length leading to the backup state. In such cases *gplex* tries to find a string that can be represented without the use of character escapes.

Consider the grammar —

```
foo             |
foobar          |
bar             { Console.WriteLine("keyword " + yytext); }
```

If this is processed with the summary option the listing file notes that the automaton has one backup state, and contains the diagnostic —

> After `<INITIAL>"foo"` automaton could accept "`foo`" in state 1
> — after '`b`' automaton is in a non-accept state and might need to backup

This case is straightforward, since after reading "foo" and seeing a '`b`' as the next character the possibility arises that the next characters might not be "ar"[7].

In other circumstances the diagnostic is more necessary. Consider a definition of words that allows hyphens and apostrophes, but not at the ends of the word, and not adjacent to each other. Here is one possible grammar —

```
alpha  [a-zA-Z]
middle ([a-zA-Z][\-']|[a-zA-Z])
%%
{middle}+{alpha}                    { ...
```

For this automaton there is just one backup state. The diagnostic is —

[7] But note that the backup is removed by adding an extra production with pattern "`{ident}*`" to ensure that all intermediate states accept *something*.

After <INITIAL>"AA" automaton could accept "{middle}+{alpha}" in state 1
— after '' automaton is in a non-accept state and might need to backup

The shortest path to the accept state requires two alphabetic characters, with "AA" a simple example. When an apostrophe (or a hyphen) is the next character, there is always the possibility that the word will end before another alphabetic character restores the automaton to the accept state.

4.6 Choosing Compression Options

Depending on the options, *gplex* scanners have either one or two lookup tables. The program attempts to choose sensible compression defaults, but in cases where a user wishes to directly control the behavior the compression of the tables may be controlled independently.

In order to use this flexibility, it is necessary to understand a little of how the internal tables of *gplex* are organized. Those readers who are uninterested in the technical details can safely skip this section and confidently rely on the program defaults.

Scanners Without Equivalence Classes

If a scanner does not use either the /*classes* or the /*unicode* options, the scanner has only a next-state table. There is a one-dimensional array, one element for each state, which specifies for each input character what the next state shall be. In the simple, uncompressed case each next-state element is simply an array of length equal to the cardinality of the alphabet. States with the same next-state table share entries, so the total number of next state entries is $(|N| - R) \times |S|$ where $|N|$ is the number of states, R is the number of states that reference another state's next-state array, and $|S|$ is the number of symbols in the alphabet. In the case of the *Component Pascal LEX* grammar there are 62 states and the 8-bit alphabet has 256 characters. Without row-sharing there would be 15872 next-state entries, however 34 rows are repeats so the actual space used is 7168 entries.

It turns out that these next-state arrays are very sparse, in the sense that there are long runs of repeated elements. The default compression is to treat the $|S|$ entries as being arranged in a circular buffer and to exclude the longest run of repeated elements. The entry in the array for each state then has a data structure which specifies: the lowest character value for which the table is consulted, the number of *non*-default entries in the table, the default next-state value, and finally the *non*-default array itself. The length of the *non*-default array is different for different states, but on average is quite short. For the *Component Pascal* grammar the total number of entries in all the tables is just 922.

Note that compression of the next-state table comes at a small price at runtime. Each next-state lookup must inspect the next-state data for the current state, check the bounds of the array, then either index into the shortened array or return the default value.

Non-Unicode Scanners With Equivalence Classes

If a scanner uses character equivalence classes, then conceptually there are two tables. The first, the *Character Map*, is indexed on character value and returns the number of the equivalence class to which that character belongs. This table thus has as many entries as there are symbols in the alphabet, $|S|$. Figure 24 shows the conceptual form of a scanner with character equivalence classes. This figure should be compared with

Figure 24: Conceptual diagram of scanner with character equivalence classes

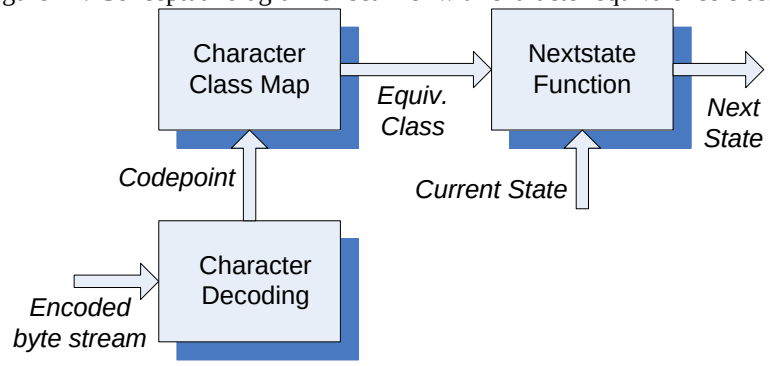


Figure 10.

The "alphabet" on which the next-state tables operate has only as many entries as there are equivalence classes, $|E|$. Because the number of classes is always very much smaller than the size of the alphabet, using classes provides a useful compression on its own. The runtime cost of this compression is the time taken to perform the mapping from character to class. In the case of uncompressed maps, the mapping cost is a single array lookup.

In the case of the *Component Pascal* scanner there are only 38 character equivalence classes, so that the size of the uncompressed next-state tables, $(|N| - R) \times |E|$, is just $(62 - 34)$ states by 38 entries, or 1064 entries. Clearly, in this case the total table size is not much larger than the case with compression but no mapping. For typical 8-bit scanners the *no-compression but character class* version is similar in size and slightly faster in execution than the default settings.

Note that although the class map often has a high degree of redundancy it is seldom worth compressing the map in the non-unicode case. The map takes up only 256 bytes, so the default for non-unicode scanners with character equivalence classes is to *not* compress the map.

Tables in Unicode Scanners

For scanners that use the unicode character set, the considerations are somewhat different. Certainly, the option of using uncompressed next-state tables indexed on character value seems unattractive, since in the unicode case the alphabet cardinality is 1114112 if all planes are considered. For the *Component Pascal* grammar this would lead to uncompressed tables of almost seventy mega-bytes. In grammars which contain unicode character literals spread throughout the character space the simple compression of the next-state tables is ineffective, so unicode scanners *always* use character equivalence classes.

With unicode scanners the use of character equivalence classes provides good compaction of the next-state tables, since the number of classes in unicode scanners is generally as small as is the case for non-unicode scanners. However the class map itself, if uncompressed, takes up more than a megabyte on its own. This often would dominate the memory footprint of the scanner, so the default for unicode scanners is to compress the character map.

When *gplex* compresses the character map of a unicode scanner it considers two

strategies, and sometimes uses a combination of both. The first strategy is to use an algorithm somewhat related to the Fraser and Hansen algorithm for compressing sparse switch statement dispatch tables. The second is to use a "two-level" table lookup.

Compression of a sparse character map involves dividing the map into dense regions which contain different values, which are separated by long runs of repeated values. The dense regions are kept as short arrays in the tables. The *Map*( ) function implements a binary decision tree of depth $\lceil \log_2 R \rceil$, where R is the number of regions in the map. After at most a number of decisions equal to the tree-depth, if the character value has fallen in a dense region the return value is found by indexing into the appropriate short array, while if a long repeated region has been selected the repeated value is returned.

A two-level table lookup divides the map function index into high and low bits. For a 64k map it is usual to use the most significant eight bits to select a sub-map of 256 entries, and use the least significant eight bits to index into the selected sub-map. In a typical case not all the sub-maps are different, so that if N is the number of bytes in the pointer type, and U is the number of unique sub-maps the total space required is $(256 \times N)$ bytes for the upper level map and $(256 \times U)$ bytes of sub-maps. Two level maps are fast, since they take only two array lookups to find a value, but for the sparse case may take more space than the alternative method.

When generating a unicode scanner *gplex* always computes a decision tree data structure. The program tries to limit the decision-tree depth in order to safeguard performance. In the case that the decision tree is too deep the program switches to two-level lookup table for the *Basic Multilingual Plane* (that is for the first 64k characters) and recursively considers a decision tree for the region beyond the 64k boundary. This is a good strategy since 14 of the remaining 16 planes are unallocated and the other two are almost always infrequently accessed.

For the common case where a *LEX* specification has no literals beyond the *ASCII* boundary the character space collapses into just two regions: a dense region covering the 7 or 8-bit range, and a repeated region that repeats all the way out to the 21-bit boundary. In this case the "decision tree" collapses into the obvious bounds-check —

```
sbyte MapC(intchr) {
    if (chr < 127) return mapC0[chr];
    else return (sbyte) 29;
}
```

where *mapC0* is the map for the dense region from '`\0`' to '`~`', and equivalence class 29 encodes the "no transition" class.

It is possible to force *gplex* to use the decision-tree algorithm over the whole alphabet by using the */squeeze* option. This almost always leads to the smallest scanner tables, but sometimes leads to very deep decision trees and poor performance.

Statistics

If the *summary* option is used, statistics related to the table compression are emitted to the listing file. This section has data for two different scanners. One is a relatively simple specification for a *Component Pascal*, and contains no unicode literal characters. The other is an extremely complicated specification for a *C#* scanner. This specification uses character equivalence classes that range through the whole of the unicode alphabet.

Figure 25 contains the statistics for the lexical grammar for the *Component Pascal Visual Studio* language service, with various options enabled. This grammar is for a

Babel scanner, and will normally get input from a string buffer. Note particularly that

Figure 25: Statistics for *Component Pascal* scanners

Options	nextstate entries	char-classes	map-entries	tree-depth
compress #	902	–	–	–
nocompress	7168	–	–	–
classes, nocompressmap, nocompressnext	1064	38	256	–
classes, nocompressmap, compressnext #	249	38	256	–
classes, compressmap, compressnext	249	38	127	1
classes, compressmap, nocompressnext	1064	38	127	1
unicode, nocompressmap, nocompressnext	1064	38	1.1e6	–
unicode, nocompressmap, compressnext	249	38	1.1e6	–
unicode, compressmap, compressnext #	249	38	127	1
unicode, compressmap, nocompressnext	1064	38	127	1

Default compression option

since the *LEX* file has no unicode character literals a unicode scanner will take up no more space nor run any slower than a non-unicode scanner using character equivalence classes. In return, the scanner will not throw an exception if it is passed a string containing a unicode character beyond the Latin-8 boundary. The default compression case is indicated in the table. Thus if no option is given the default is */compress*. With option */classes* the default is */nocompressmap /compressnext*. Finally, with option */unicode* the default is */compressmap /compressnext*.

For the unicode scanners that compress the map the compression used is: a table for the single dense region covering the first 127 entries, a default *don't care* value for the rest of the alphabet, and a decision tree that has degenerated into a simple bounds check.

An example more typical of unicode scanners is the scanner for *C#*. This scanner implements the *ECMA-334* standard, which among other things allows identifiers to contain characters that are located throughout the whole unicode alphabet. In this

Figure 26: Statistics for *C#* scanner

Options	nextstate entries	char-classes	map-entries	tree-depth
unicode	1360	55	13568	5
unicode, squeeze	1360	55	9744	7
unicode, nocompressmap, nocompressnext	4675	55	1.1e6	–
unicode, nocompressmap, compressnext	1360	55	1.1e6	–
unicode, compressmap, compressnext #	1360	55	13568	5
unicode, compressmap, nocompressnext	4675	55	13568	5

Default compression option

case, the default compression if only the */unicode* option is given is */compressmap /compressnext*. The compressed map in this case consists of: a two level lookup table for the basic multilingual plane with a 256-entry upper map pointing to 47 unique submaps. The rest of the map is implemented by a decision-tree of depth 5, with a total of only 1280 entries in the dense arrays.

The use of the */squeeze* option generates a scanner with a map that is compressed by a single decision-tree. The tree has depth 7, and the dense arrays contain a total of 9744 elements. Given that the decision tree itself uses up memory space, it is not clear that in this case the overall compression is significantly better than the default.

When to use Non-Default Settings

If a non-unicode scanner is particularly time critical, it may be worth considering using character equivalence classes and not compressing either tables. This is usually slightly faster than the default settings, with very comparable space requirements. In even more critical cases it may be worth considering simply leaving the next-state table uncompressed. Without character equivalence classes this will cause some increase in the memory footprint, but leads to the fastest scanners.

For unicode scanners, there is no option but to use character equivalence classes, in the current release. In this case, a moderate speedup is obtained by leaving the next-states uncompressed. Compressing the next-state table has roughly the same overhead as one or two extra levels in the decision tree.

The depth of the decision tree in the compressed maps depends on the spread of unicode character literals in the specification. Some pathological specifications are known to have caused the tree to reach a depth of seven or eight.

Using the *summary* option and inspecting the listing file is the best way to see if there is a problem, although it may also be seen by inspecting the source of the produced scanner *C#* file.

5 Errors and Warnings

There are a number of errors and warnings that *gplex* detects. Errors are fatal, and no scanner source file is produced in that case. Warnings are intended to be informative, and draw attention to suspicious constructs that may need manual checking by the user.

5.1 Errors

Errors are displayed in the listing file, with the location of the error highlighted. In some cases the error message includes a variable text indicating the erroneous token or the text that was expected. In the following the variable text is denoted <...>.

“%%” marker must start at beginning of line —
An out-of-place marker was found, possibly during error recovery from an earlier error.

Cannot set `/unicode` option inconsistently <...> —
Normally options are processed in order and may undo other option’s effect. However, options that explicitly set the alphabet size such as */unicode* or */nounicode* cannot be contradicted by later options.

Class <...> not found in assembly —
The class specified for a user-defined character class predicate could not be found in the nominated assembly.

Context must have fixed right length or fixed left length —
gplex has a limitation on the implementation of patterns with right context. Either the right context or the body of the pattern must recognize fixed length strings.

Context operator cannot be used with a right anchor "$" —
The regular expression (possibly after expanding named categories) has both a context operator and a right anchor symbol.

Empty semantic action, must be at least a comment —
No semantic action was found. This error also occurs due to incorrect syntax in the *preceeding* rule.

Expected character <...> —
During the scanning of a regular expression an expected character was not found. This most commonly arises from missing right hand bracketing symbols, or closing quote characters.

Expected space here —
The *gplex* parser was expecting whitespace. This can arise when a lexical category definition is empty or when the pattern of a rule is followed by an end-of-line rather than a semantic action.

Expected end-of-line here —
Unexpected non-whitespace characters have been found at the end of a construct when an end of line is the only legal continuation.

Extra characters at end of regular expression —
The regular expression is incorrectly terminated.

Illegal escape sequence <...> —
An illegal escape sequence was embedded in a literal string.

Illegal name for start condition <...> —
Names of start conditions must be identifiers. As a special case the number zero may be used as a shortcut for a used occurrence of the initial start state. Any other numeric reference is illegal.

Illegal octal character escape <...> —
Denotation of character values by escaped octal sequences must contain exactly three octal digits, except for the special case of '`\0`'.

Illegal hexadecimal character escape <...> —
Denotation of character values by escaped hexadecimal sequences must contain exactly two hexadecimal digits.

Illegal unicode character escape <...> —
Denotation of character values by unicode escapes must have exactly four hexadecimal digits, following a '`\u`' prefix, or exactly eight hexadecimal digits, following a '`\U`' prefix.

Illegal character in this context —
The indicated character is not the start of any possible *gplex* token in the current scanner state.

Inconsistent "`%option`" command <...> —
The message argument is an option that is inconsistent with already processed options. In particular, it is not possible to declare */noClasses* for a unicode scanner.

Invalid action —
There is a syntax error in the multi-line semantic action for this pattern.

Invalid or empty namelist —
There is a syntax error in the namelist currently being parsed.

Invalid production rule —
There is a syntax error in the rule currently being parsed.

Invalid character range: lower bound > upper bound —
In a character range within a character class definition the character on the left of the '–' must have a numerically smaller code point than the character on the right.

Invalid single-line action —
gplex found a syntax error in the parsing of a single-line semantic action.

Invalid class character: '–' must be escaped —
A '–' character at the start or end of a character set definition is taken as a literal, single character. Everywhere else in a set definition this character must be escaped unless it is part of a range declaration.

Lexical category <...> already defined —
The lexical category in this definition is already defined in the symbol table.

Lexical category must be a character class <...> —
In this version of *gplex* character set membership predicates can only be generated for lexical categories that are character classes "[...]".

Method <...> not found in class —
The method specified for a user-defined character class predicate could not be found in the nominated class, or the method does not have the correct signature.

Missing matching construct <...> —
The parser has failed to find a matching right hand bracketing character. This may mean that brackets (either '(', '[' or '{') are improperly nested.

"namespace" is illegal, use "`%namespace`" instead —
C# code in the lex specification is inserted *inside* the generated scanner class. The namespace of the scanner can only be set using the non-standard `%namespace` command.

"next" action '|' cannot be used on last pattern —
The '`|`' character used as a semantic action has the meaning "*use the same action as the following pattern*". This action cannot be applied to the last pattern in a rules section.

No namespace has been defined —
The end of the definitions section of the specification was reached without finding a valid namespace declaration.

Non unicode scanner cannot use /codePage:guess —
For byte-mode scanners the code page setting is used at scanner generation time to determine the meaning of character predicates. The code page guesser works at scanner runtime.

Only "public" and "internal" allowed here —
The "`%visibility`" marker can only declare the scanner class to be public or internal.

Parser error <...> —
The *gplex* parser has encountered a syntax error in the input *LEX* file. The nature of the error needs to be found from the information in the <...> placeholder.

Start state <...> already defined —
All start state names must be unique. The indicated name is already defined.

Start state <...> undefined —
An apparent use of a start state name does not refer to any defined start state name.

Symbols '^' and '$' can only occur at ends of patterns —
The two anchor symbols can only occur at the end of regular expressions. This error can arise when an anchor symbol is part of a lexical category which is then used as a term in another expression. Using anchor symbols in lexical categories should be deprecated.

This assembly could not be found —
The assembly specified for a user-defined character class predicate could not be found. The *PE*-file must be in the current working directory.

This assembly could not be loaded —
The assembly specified for a user-defined character class predicate could not be loaded. The assembly must be a valid *.NET* managed code *PE*-file, and *gplex* must have sufficient privilege to load the assembly.

This token unexpected —
The parser is expecting to find indented text, which can only be part of a *C#* code-snippet. The current text does not appear to be legal *C#*.

Type declarations impossible in this context —
gplex allows type declarations (`class, struct, enum`) in the definitions section of the specification, and in the user code section. Type declarations are not permitted in the rules section.

"using" is illegal, use "%using" instead —
C# code in the lex specification is inserted *inside* the generated scanner class. The using list of the scanner module can only have additional namespaces added by using the non-standard `%using` command.

Unknown lexical category <...> —
This name is not the name of any defined lexical category. This could be a character case error: lexical category names are case-sensitive.

Unexpected symbol, skipping to <...> —
gplex has found a syntax error in the current section. It will discard input until it reaches the stated symbol.

Unrecognized "`%option`" command <...> —
The given option is unknown.

Unknown character predicate <...> —
The character predicate name in the `[: ... :]` construct is not known to *gplex*.

Unicode literal too large <...> —
The unicode escape denotes a character with a code point that exceeds the limit of the unicode definition, `0x10ffff`.

Unterminated block comment start here —
A end of this block comment `/*` ... `*/` was not found before the end of file was reached. The position of the *start* of the unterminated comment is marked.

Unknown lex tag name —
Tags in *gplex* are all those commands that start with a %.... The current tag is not known. Remember that tag names are case-sensitive.

Version of gplexx.frame is not recent enough —
The version of gplexx.frame that *gplex* found does not match the *gplex* version.

5.2 Warnings

A number of characteristics of the input specification may be dangerous, or require some additional checking by the user. In such cases *gplex* issues one of the following warnings. In some cases the detected constructs are intended, and are safe.

`/babel` **option is unsafe without** `/unicode` **option** —
Scanners generated with the *babel* option read their input from strings. It is unsafe to generate such a scanner without declaring */unicode* since the input string might contain a character beyond the Latin-8 boundary, which will cause the scanner to throw an exception.

Code between rules, ignored —
Code *between* rules in the rules section of a specification cannot be assigned to any meaningful location in the generated scanner class. It has been ignored.

No upper bound to range, <...> included as set class members —
It is legal for the last character in a character set definition to be the '–' character. However, check that this was not intended to be part of a range definition.

Special case: <...> included as set class member —
It is legal for the first character in a character set definition to be the '–' character. However, check that this was not intended to be part of a range definition.

This pattern is never matched —
gplex has detected that this pattern cannot ever be matched. This might be an error, caused by incorrect ordering of rules. (See the next two messages for diagnostic help).

This pattern always overridden by <...> —
In the case that a pattern is unreachable, this warning is attached to the unreachable pattern. The variable text of the message indicates (one of) the patterns that will be matched instead. If this is not the intended behavior, move the unreachable pattern earlier in the rule list.

This pattern always overrides pattern <...> —
This warning message is attached to the pattern that makes some other pattern unreachable. The variable text of the message indicates the pattern that is obscured.

This pattern matches the empty string, and might loop —
One of the input texts that this pattern matches is the empty string. This may be an error, and might cause the scanner to fail to terminate. The following section describes the circumstances under which such a construct is *NOT* an error.

Matching the Empty String

There are a number of circumstances under which a pattern can match the empty string. For example, the regular expression may consist of a `*`-closure or may consist of a concatenation of symbols each of which is optional. It is also possible for a pattern with fixed-length right context to have a pattern body (variable-length left context) which matches the empty string. All such patterns are detected by *gplex*.

Another way in which a pattern recognition might consume no input is for the semantic action of a pattern to contain the command `yyless(0)`. If this is the case the semantic action will reset the input position back to the *start* of the recognised pattern.

In all cases where the pattern recognition does not consume any input, if the start state of the scanner is not changed by the semantic action the scanner will become stuck in a loop and never terminate.

Nevertheless, it is common and useful to include patterns that consume no input. Consider the case where some characteristic pattern indicates a “phase change” in the input. Suppose X denotes that pattern, S_1 is the previous start condition and the new phase is handled by start condition S_2. The following specification-pattern is a sensible way to implement this semantic —

```
<S1>X { BEGIN(S2); yyless(0); }
<S2>...
```

Using this specification-pattern allows the regular expression patterns that belong to the S_2 start state to include patterns that begin by matching the X that logically begins the new input phase. The lexical specification for *gplex* uses this construct no less than three times. For scanners that use the */stack* option, calling *yy_pop_state* or *yy_push_state* also constitute a change of start state for purposes of avoiding looping.

6 Examples

This section describes the stand-alone application examples that are part of the *gplex* distribution. In practice the user code sections of such applications might need a bit more user interface handling.

The text for all these examples is in the "`Examples`" subdirectory of the distribution.

6.1 Word Counting

This application scans the list of files on the argument list, counting words, lines, integers and floating point variables. The numbers for each file are emitted, followed by the totals if there was more than one file.

The next section describes the input, line by line.

The file *WordCount.lex* begins as follows.

```
%namespace LexScanner
%option noparser, verbose
%{
    static int lineTot = 0;
    static int wordTot = 0;
    static int intTot  = 0;
    static int fltTot  = 0;
%}
```

the definitions section begins with the namespace definition, as it must. We do not need any "`using`" declarations, since *System* and *System.IO* are needed by the invariant code of the scanner and are imported by default. Next, four class fields are defined. These will be the counters for the totals over all files. Since we will create a new scanner object for each new input file, we make these counter variables `static`.

Next we define three character classes —

```
alpha [a-zA-Z]
alphaplus [a-zA-Z\-']
digits [0-9]+
%%
```

Alphaplus is the alphabetic characters plus hyphens (note the escape) and the apostrophe. *Digits* is one or more numeric characters. The final line ends the definitions section and begins the rules.

First in the rules section, we define some local variables for the *Scan* routine. Recall that code *before* the first rule becomes part of the prolog.

```
int lineNum = 0;
int wordNum = 0;
int intNum  = 0;
int fltNum  = 0;
```

These locals will accumulate the numbers within a single file. Now come the rules —

```
\n|\r\n?                    lineNum++; lineTot++;
{alpha}{alphaplus}*{alpha} wordNum++; wordTot++;
{digits}                    intNum++;  intTot++;
{digits}\.{digits}          fltNum++;  fltTot++;
```

The first rule recognizes all common forms of line endings. The second defines a word as an alpha followed by more alphabetics or hyphens or apostrophes. The third and fourth recognize simple forms of integer and floating point expressions. Note

especially that the second rule allows words to contain hyphens and apostrophes, but only in the *interior* of the word. The word must start and finish with a plain alphabetic character.

The fifth and final rule is a special one, using the special marker denoting the end of file. This allows a semantic action to be attached to the recognition of the file end. In this case the action is to write out the per-file numbers.

```
<<EOF>>  {
          Console.Write("Lines:  " + lineNum);
          Console.Write(", Words:  " + wordNum);
          Console.Write(", Ints:  " + intNum);
          Console.WriteLine(", Floats:  " + fltNum);
        }
%%
```

Note that we could also have placed these actions as code in the epilog, to catch termination of the scanning loop. These two are equivalent in this particular case, but only since no action performs a return. We could also have placed the per-file counters as instance variables of the scanner object, since we construct a fresh scanner per input file.

The final line of the last snippet marks the end of the rules and beginning of the user code section.

The user code section is shown if Figure 27. The code opens the input files one by one, creates a scanner instance and calls *yylex*.

Figure 27: User Code for Wordcount Example

```
public static void Main(string[] argp) {
  for (int i = 0; i < argp.Length; i++) {
    string name = argp[i];
    try {
      int tok;
      FileStream file = new FileStream(name, FileMode.Open);
      Scanner scnr = new Scanner(file);
      Console.WriteLine("File:  " + name);
      do {
        tok = scnr.yylex();
      } while (tok > (int)Tokens.EOF);
    } catch (IOException) {
      Console.WriteLine("File " + name + " not found");
    }
  }
  if (argp.Length > 1) {
    Console.Write("Total Lines:  " + lineTot);
    Console.Write(", Words:  " + wordTot);
    Console.Write(", Ints:  " + intTot);
    Console.WriteLine(", Floats:  " + fltTot);
  }
}
```

Building the Application

The file *WordCount.cs* is created by invoking —

```
D:\gplex\test> gplex /summary WordCount.lex
```

This also creates *WordCount.lst* with summary information.

This particular example, generates 26 *NFSA* states which reduce to just 12 *DFSA* states. Nine of these states are *accept* states[8] and there are two backup states. Both backup states occur on a "." input character. In essence when the lookahead character is dot, *gplex* requires an extra character of lookahead to before it knows if this is a full-stop or a decimal point. Because *gplex* performs state minimization by default, two backup states are merged and the final automaton has just nine states.

Since this is a stand-alone application, the parser type definitions are taken from the *gplexx.frame* file, as described in Figure 19. In non stand-alone applications these definitions would be accessed by "`%using`" the parser namespace in the lex file. By default *gplex* embeds the buffer code in the *WordCount.cs* output file. Thus we only need to compile a single file —

```
D:\gplex\test> csc WordCount.cs
```

producing *WordCount.exe*. Run the executable over its own source files —

```
D:\gplex\test> WordCount WordCount.cs WordCount.lex
File: WordCount.cs
Lines: 590, Words: 1464, Ints: 404, Floats: 3
File: WordCount.lex
Lines: 64, Words: 151, Ints: 13, Floats: 0
Total Lines: 654, Words: 1615, Ints: 417, Floats: 3
D:\gplex\test>
```

The text in plain typewriter font is console output, the slanting, bold font is user input.

Where do the three "floats" come from? Good question! The text of *WordCount.cs* quotes some version number strings in a header comment. The scanner thinks that these look like floats. As well, one of the table entries of the automaton has a comment that the shortest string reaching the corresponding state is "`0.0`".

6.2 *ASCII* Strings in Binary Files

A very minor variation of the word-count grammar produces a version of the *UNIX* "strings" utility, which searches for ascii strings in binary files. This example uses the same user code section as the word-count example, Figure 27, with the following definitions and rules section —

```
alpha [a-zA-Z]
alphaplus [a-zA-Z\-']
%%
{alpha}{alphaplus}*{alpha}  Console.WriteLine(yytext);
%%
```

This example is in file "`strings.lex`".

[8] These are always the lowest numbered states, so as to keep the dispatch table for the semantic action **switch** statement as dense as possible.

6.3 Keyword Matching

The third example demonstrates scanning of *strings* instead of files, and the way that *gplex* chooses the lowest numbered pattern when there is more than one match. Here is the start of the file "`foobar.lex`".

```
%namespace LexScanner
%option noparser nofiles
alpha [a-zA-Z]
%%
foo         |
bar         Console.WriteLine("keyword " + yytext);
{alpha}{3}  Console.WriteLine("TLA " + yytext);
{alpha}+    Console.WriteLine("ident " + yytext);
%%
```

The point is that the input text "foo" actually matches three of the four patterns. It matches the "*TLA*" pattern and the general ident pattern as well as the exact match. Altering the order of these rules will exercise the "unreachable pattern" warning messages. Try this!

Figure 28 is the string-scanning version of the user code section. This example

Figure 28: User Code for keyword matching example

```
public static void Main(string[] argp) {
    Scanner scnr = new Scanner();
    for (int i = 0; i < argp.Length; i++) {
        Console.WriteLine("Scanning \"" + argp[i] + "\"");
        scnr.SetSource(argp[i], 0);
        scnr.yylex();
    }
}
```

takes the input arguments and passes them to the *SetSource* method. Try the program out on input strings such as "`foo bar foobar blah`" to make sure that it behaves as expected.

After playing with this example, try generating a scanner with the *caseInsensitive* option. The scanner will recognize all of "foo", "FOO", "fOo" and so on as keywords, but will display the actual text of the input in the output. Notice that in this case the character class "alpha" could just as well have been defined as "`[a-z]`".

One of the purposes of this example is to demonstrate one of the two usual ways of dealing with reserved words in languages. One may specify each of the reserved words as a pattern, with a catch-all identifier pattern at the end. For languages with large numbers of keywords this leads to automata with very large state numbers, and correspondingly large next-state tables.

When there are a large number of keywords it is sensible to define a single identifier pattern, and have the semantic action delegate to a method call —

```
return GetIdToken(yytext);
```

The *GetIdToken* method should check if the string of the text matches a keyword, and return the appropriate token. If there really are many keywords the method should perform a switch on the first character of the string to avoid sequential search. Finally,

for scanners generated with the */caseInsensitive* switch remember that the *yytext* value will retain the case of the original input. For such applications the *GetIdToken* method should do a *String.ToUpper* call to canonicalize the case before testing for string equality.

6.4 The Code Page Guesser

The "code page guesser" is invoked by unicode scanners generated with the *codePage:guess* option if an input file is opened which has no *UTF* prefix. The guesser scans the input file byte-by-byte, trying to choose between treating the file as a utf-8 file, or presuming it to be an 8-bit byte-file encoded using the default code page of the host machine.

The example file "`GuesserTest.lex`" is a wrapped example of the code page guesser. It scans the files specified in the command line, and reports the number of significant patterns of each kind that it finds in each file.

The basic idea is to look for sequences of bytes that correspond to well-formed utf-8 character encodings that require two or more bytes. The code also looks for bytes in the upper-128 byte-values that are not part of any valid utf-8 character encoding. We want to create an automaton to accumulate counts of each of these events. Furthermore, we want the code to run as quickly as possible, since the real scanner cannot start until the guesser delivers its verdict.

The following character sets are defined —

```
Utf8pfx2  [\xc0-\xdf] // Bytes with pattern 110x xxxx
Utf8pfx3  [\xe0-\xef] // Bytes with pattern 1110 xxxx
Utf8pfx4  [\xf0-\xf7] // Bytes with pattern 1111 0xxx
Utf8cont  [\x80-\xbf] // Bytes with pattern 10xx xxxx
Upper128  [\x80-\xff] // Bytes with pattern 1xxx xxxx
```

These sets are: all those values that are the first byte of a two, three or four-byte utf-8 character encoding respectively; all those values that are valid continuation bytes for multi-byte utf-8 characters; and all bytes that are in the upper-128 region of the 8-bit range.

Counts are accumulated for occurrences of two-byte, three-byte and four-byte utf-8 character patterns in the file, and bytes in the upper 128 byte-value positions that are not part of any legal utf-8 character. The patterns are —

```
{Utf8pfx2}{Utf8cont}      utf2++; // Increment 2-byte utf counter
{Utf8pfx3}{Utf8cont}{2}   utf3++; // Increment 3-byte utf counter
{Utf8pfx4}{Utf8cont}{3}   utf4++; // Increment 4-byte utf counter
{Upper128}                uppr++; // Increment upper non-utf count
```

It should be clear from the character set definitions that this pattern matcher is defined in a natural way in terms of symbol equivalence classes. This suggests using *gplex* with the *classes* option. The resulting automaton has six equivalence classes, and just twelve states. Unfortunately, it also has two backup states. The first of these occurs when a *Utf8pfx3* byte has been read, and the next byte is a member of the *Utf8cont* class. The issue is that the first byte is a perfectly good match for the *uppr* pattern, so if the byte *two ahead* is not a second *Utf8cont* then we will need to back up and accept the *uppr* pattern. The second backup state is the cognate situation for the four-byte *utf4* pattern.

Having backup states makes the automaton run slower, and speed here is at a premium. Some reflection shows that the backup states may be eliminated by defining three extra patterns —

```
{Utf8pfx3}{Utf8cont}      uppr += 2; // Increment uppr by two
{Utf8pfx4}{Utf8cont}      uppr += 2; // Increment uppr by two
{Utf8pfx4}{Utf8cont}{2}   uppr += 3; // Increment uppr by three
```

With these additional patterns, when the first two bytes of the *utf3* or *utf4* patterns match, but the third byte does not, rather than back up, we add *two* to the *uppr* count. Similarly, if the first three bytes of the *utf4* pattern match but the fourth byte does not match we add *three* to the *uppr* count.

The new automaton has the same number of equivalence classes, and the same number of states, but has no backup states. This automaton can run very fast indeed.

6.5 Include File Example

The example program *IncludeTest* is a simple harness for exercising the include file facilities of *gplex*. The complete source of the example is the file "`IncludeTest.lex`" in the distribution.

The program is really a variant of the "strings" program of a previous example, but has special semantic actions when it reads the string "`#include`" at the start of an input line. As expected, the file declares a *BufferContext* stack.

```
Stack<BufferContext> bStack = new Stack<BufferContext>();
```

Compared to the strings example there are some additional declarations.

```
%x INCL             // Start state while parsing include command
dotchr [^\r\n]      // EOL-agnostic version of traditional LEX '.'
eol    (\r\n?|\n)   // Any old end of line pattern
 ...                // And so on ...
```

The rules section recognizes strings of length two or more, the include pattern, and also processes the filenames of included files.

```
{alpha}{alphaplus}*{alpha}  { Console.WriteLine(
      "{0}{1} {2}:{3}", Indent(), yytext, yyline, yycol); }
^"#include"                 BEGIN(INCL);
<INCL>{eol}                 BEGIN(0); TryInclude(null);
<INCL>[ \t]                 /* skip whitespace */
<INCL>[^ \t]{dotchr}*       BEGIN(0); TryInclude(yytext);
```

The *Indent* method returns a blank string of length depending on the depth of the buffer context stack. This "pretty prints" the output of this test program.

The user code in Figure 29 supplies *Main*, *TryInclude* and *yywrap* for the example. In this example the command line arguments are passed into a *LineBuff* buffer. Since the buffers that result from file inclusion will be of *BuildBuff* type, this demonstrates the ability to mix buffers of different types using file inclusion.

Most of the error checking has been left out of the figure, but the example in the distribution has all the missing detail.

7 Notes

7.1 Moving From v1.0 to v1.1.0

Version 1.1.0 of *gplex* is a relatively major change to the tool, and involves a number of changes that are potentially breaking for some existing applications. Breaking changes are a matter of regret, so this section attempts to explain the nature of the changes, and the reasons.

Figure 29: User code for *IncludeTest* example

```
public static void Main(string[] argp) {
    if (argp.Length == 0)
        Console.WriteLine("Usage:  IncludeTest args");
    else {
        int tok;
        Scanner scnr = new Scanner();
        scnr.SetSource(argp); // Create LineBuff object from args
        do {
            tok = scnr.yylex();
        } while (tok > (int)Tokens.EOF);
    }
}

private void TryInclude(string fName) {
    if (fName == null)
        Console.Error.WriteLine("#include, no filename");
    else {
        BufferContext savedCtx = MkBuffCtx();
        SetSource(new FileStream(fName, FileMode.Open));
        Console.WriteLine("Included file {0} opened", fName);
        bStack.Push(savedCtx); // Don't push until after file open succeeds!
    }
}

protected override bool yywrap() {
    if (bStack.Count == 0) return true;
    RestoreBuffCtx(bStack.Pop());
    Console.WriteLine("Popped include stack");
    return false;
}
```

7.1.1 Performance Issues

Earlier versions of *gplex* produced scanners with poor performance on large input files. All file buffers were built on top of a buffered byte-stream, and the byte-stream position was used for the buffer's *Pos* property. As a consequence calls to *yyless* and *buffer.GetString* caused IO seeks, with a large performance hit.

Version 1.1.0 uses an object of the *StreamReader* class to read the input stream, and then buffers the resulting `char` values in a double-buffer based on the *StringBuilder* class. "Seek" within this buffer causes no IO activity, but simply indexes within the builder. This solves the performance problem.

However, the ability to perform an arbitrary seek within the input file has been lost, since the string builder tries to keep no more than two file-system pages in the buffer. The default behavior of the buffers in version 1.1.0 is to *not* reclaim buffer space. The */noPersistBuffer* option reduces the memory footprint for those application where the buffers do not need to be persisted.

7.1.2 Removing Unicode Encoding Limitations

Earlier versions of *gplex* used character encodings which were built on top of a byte stream. However, the available encodings were limited to the *utf* formats, or any of the library encodings that have the "single byte property". The current version may use any of the encoders from the *System.Globalization* namespace of the base class libraries.

Scanners consume unicode code points, represented as integer values. However, for all input sources the code point "position" is represented by the ordinal number of the first *System.Char* from which the code point is derived. See figure 13. There is some small inefficiency involved for *utf-8* encodings where characters from outside the *basic multilingual plane* are decoded to an integer value and then split into a surrogate pair in the buffer. The *GetCode* method will then merge the pair back into a single code point to deliver to the scanning engine. This is a small price to pay for the convenience of having a uniform representation for input position[9].

7.1.3 Avoiding Name-Clashes with Multiple Scanners

For those applications that use multiple scanners, problems arose with name-clashes in duplicated code. The new version moves all of the invariant, buffer code into the separate resource "*GplexBuffers*". This resource may either be included in the project as a single file which may be shared between multiple scanners, or may be embedded in each of the separate scanner namespaces. The default behavior is to embed the code in the scanner namespace. The default is appropriate and simple for single-scanner applications, particularly stand-alone scanner-based tools. See section 4.2 for more detail.

An additional resource in version 1.1.0 is the possibility to limit the visibility of the generated types, and to override the default naming of the scanner, token and scanner base types.

7.1.4 Compliance with *FxCop*

Applications which embed *gplex* scanners trigger a large number of design-rule warnings in *FxCop*. Some of these warnings relate to naming guidelines, while others impact on code safety.

Version 1.1.0 generates scanners which are *FxCop*-friendly. Those guidelines which *gplex* cannot honor, such as the naming of legacy *API* elements with names beginning with "yy" are explicitly marked with a message suppression attribute. In most cases the reason for the message suppression is noted in a source comment.

The major changes resulting from this exercise with the potential to break existing applications fall into two categories. Some of the non-legacy members of the scanner classes have been renamed. This will cause issues for *user-written* code that accesses internal scanner class members. This may require some renaming of identifiers. For example, the abstract base class of scanners, defined in *gppg*, has been changed from *IScanner* to *AbstractScanner*. User code probably never refers to this class, but if an existing application happens to do this, the code will need changing. Similarly, user-written semantic actions normally have no need to directly call the "get next codepoint" function of the scanner class. However, if existing scanners do this, then it is relevant that the name has changed from *GetChr* to *GetCode*.

[9] Several attempts were made to create a buffer class that directly buffered code points, but none performed as well as the *StringBuilder* class.

More serious is the restructuring and renaming of classes in the buffer code. All of the concrete buffer classes are now private, and scanners access buffers *only* via the facilities presented by the abstract *ScanBuff* class. User code can only create buffer objects using the static factory method-group *ScanBuff.GetBuffer*, or more sensibly, using the scanner's *SetSource* method-group. For a tabular summary of potentially breaking changes see Appendix **??**.

7.2 Implementation Notes

Versions since 0.4.0 parse their input files using a parser constructed by Gardens Point Parser Generator (*gppg*). Because it is intended to be used with a colorizing scanner the grammar contains rules for both the *LEX* syntax and also many rules for *C#*. The parser will match braces and other bracketing constructs within the code sections of the *LEX* specification. *gplex* will detect a number of syntax errors in the code parts of the specification prior to compilation of the resulting scanner output file.

Compatibility

The current version of *gplex* is not completely compatible with either *POSIX LEX* or with *Flex*. However, for those features that *are* implemented the behaviour follows *Flex* rather than *POSIX* when there is a difference.

Thus *gplex* implements the "`<<EOF>>`" marker, and both the "`%x`" and "`%s`" markers for start states. The semantics of pattern expansion also follows the *Flex* model. In particular, operators applied to named lexical categories behave as though the named pattern were surrounded by parentheses. Forthcoming versions will continue this preference.

Error Reporting

The default error-reporting behavior of *gppg*-constructed parsers is relatively primitive. By default the calls of *yyerror* do not pass any location information. This means that there is no flexibility in attaching messages to particular positions in the input text. In contexts where the *ErrorHandler* class supplies facilities that go beyond those of *yyerror* it is simple to disable the default behaviour. The scanner base class created by the parser defines an empty *yyerror* method, so that if the concrete scanner class does not override *yyerror* no error messages will be produced automatically, and the system will rely on explicit error messages in the parser's semantic actions.

In such cases the semantic actions of the parser will direct errors to the real error handler, without having these interleaved with the default messages from the shift-reduce parsing engine.

7.3 Limitations for Version 1.1.0

Version 1.1.0 supports anchored strings but does not support variable right context. More precisely, in $\mathbf{R}_1/\mathbf{R}_2$ at least one of the regular expressions $\mathbf{R}_2$ and $\mathbf{R}_1$ must define strings of fixed length. Either regular expression may be of arbitrary form, provided all accepted strings are the same constant length. As well, the standard lex character set definitions such as "`[:isalpha:]`" are not supported. Instead, the character predicates from the base class libraries, such as *IsLetter* are permitted.

The default action of *LEX*, echoing *unmatched* input to standard output, is not implemented. If you really need this it is easy enough to do, but if you don't want it, you don't have to turn it off.

7.4 Installing *GPLEX*

gplex is distributed as a zip archive. The archive should be extracted into any convenient folder. The distribution contains four subdirectories. The "`binaries`" directory contains the file: *gplex.exe*. In environments that have both *gplex* and Gardens Point Parser Generator (*gppg*), it is convenient to put the executables for both applications in the same directory.

The "`project`" directory contains the *Visual Studio* project from which the current version of *gplex* was built. The "`documentation`" directory contains the files —

"`Gplex.pdf`",
"`Gplex-Changelog.pdf`", and the file
"`GplexCopyright.rtf`".

The "`examples`" directory contains the examples described in this documentation.

The application requires version 2.0 of the *Microsoft .NET* runtime.

7.5 Copyright

Gardens Point *LEX* (*gplex*) is copyright © 2006–2010, John Gough, Queensland University of Technology. See the accompanying document "`GPLEXcopyright.rtf`". Code that you generate with *gplex* is not covered by the *gplex* licence, it is your own. In particular, the inclusion of *gplex* library code in the generated code does not make the generated code a "derived work" of *gplex*.

7.6 Bug Reports

Gardens Point *LEX* (*gplex*) is currently being maintained and extended by John Gough. Bug reports and feature requests for *gplex* should be posted to the issues tab of the *gplex* page on CodePlex.

Part II
The Input Language

8 The Input File

8.1 Lexical Considerations

Every *gplex*-generated scanner operates either in byte-mode or in unicode-mode. Since Version 1.2.0 *gplex* expects its own input to be a Unicode file in one of the *UTF* formats, with a valid byte order mark (*BOM*). If *gplex* reads an input file without a *BOM* it falls back to the "raw" codepage and treats its input file as a stream of 8-bit bytes.

Newer versions of *gplex* will therefore correctly interpret byte-mode lex files prepared for previous versions.

8.1.1 Character Denotations

Input files may define unicode scanners, whether or not the input comes from a Unicode or byte-mode file. In either case specfications may denote character literals throughout the entire unicode range. Denotations of characters in *gplex* may be uninterpreted occurrences of plain characters, or may be one of the conventional character escapes, such as '`\n`' or '`\0`'. As well, characters may be denoted by octal, hexadecimal or unicode escapes.

In different contexts within a *LEX* specification different sets of characters have special meaning. For example, within regular expressions parentheses "`(, )`" are used to denote grouping of sub-expressions. In all such cases the ordinary character is denoted by an *escaped* occurrence of the character, by being prefixed by a backslash '`\`' character. In the regular expression section 9 of this document the characters that need to be escaped in each context are listed.

8.1.2 Names and Numbers

There are several places in the input syntax where names and name-lists occur. Names in version 1.0 are simple, *ASCII*, alphanumeric identifiers, possibly containing the low-line character '_'. This choice, while restrictive, makes input files independent of host code page setting. Name-lists are comma-separated sequences of names.

Numbers are unformatted sequences of decimal digits. *gplex* does not range-check these values. If a value is too large for the `int` type an exception will be thrown.

8.2 Overall Syntax

A lex file consists of three parts: the *definitions* section, the *rules* section, and the *user-code* section[10].

LexInput
: *DefinitionSequence* "%%" *RulesSection* $UserCodeSection_{opt}$
;

UserCodeSection
: "%%" $UserCode_{opt}$
;

[10] Grammar fragments in this documentation will follow the meta-syntax used for *gppg* and other bottom-up parsers.

The *UserCode* section may be left out, and if is absent the dividing mark "%%" may be left out as well.

8.3 The Definitions Section

The definitions section contains several different kinds of declarations and definitions. Each definition begins with a characteristic keyword marker beginning with "%", and must be left-anchored.

DefinitionSequence
: *DefinitionSequence*$_{opt}$ *Definition*
;

Definition
: *NamespaceDeclaration*
| *UsingDeclaration*
| *VisibilityDeclaration*
| *NamingDeclaration*
| *StartConditionsDeclaration*
| *LexicalCategoryDefintion*
| *CharacterClassPredicatesDeclaration*
| *UserCharacterPredicateDeclaration*
| *UserCode*
| *OptionsDeclaration*
;

8.3.1 Using and Namespace Declarations

Two non-standard markers in the input file are used to generate `using` and `namespace` declarations in the scanner file.

The definitions section must declare the namespace in which the scanner code will be placed. A sensible choice is something like *AppName*`.Lexer`. The syntax is —

NamespaceDeclaration
: "`%namespace`" *DottedName*
;

where *DottedName* is a possibly qualified *C#* identifier.

The following namespaces are imported by default into the file that contains the scanner class —

```
using System;
using System.IO;
using System.Text;
using System.Globalization;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.Diagnostics.CodeAnalysis;
```

If buffer code is *not* embedded in the scanner file, then *QUT.GplexBuffers* is imported also.

Any other namespaces that are needed by user code or semantic actions must be specified in a "`%using`" declaration.

UsingDeclaration
: "`%using`" *DottedName* '`;`'
;

For scanners that work on behalf of *gppg*-generated parsers it would be necessary to import the namespace of the parser. A typical declaration would be —

```
%using myParserNamespace;
```

Note that the convention for the use of semicolons follows that of *C#*. Using declarations need a semicolon, namespace declarations do not.

Every input file must have exactly one namespace declaration. There may be as many, or few, using declarations as are needed by the user.

8.3.2 Visibility and Naming Declarations

Four non-standard declarations are used to control the visibility and naming of the types used in the *gplex API*. The visibility declaration has the following syntax —

VisibilityDeclaration
: "`%visibility`" *Keyword*
;

where *Keyword* may be either `public` or `internal`. The declaration sets the visibility of the types *Tokens, ScanBase, IColorScan, Scanner*. The default is public.

Naming declarations have the following syntax —

NamingDeclaration
: "`%scanbasetype`" *Identifier*
| "`%scannertype`" *Identifier*
| "`%tokentype`" *Identifier*
;

where *Identifier* is a simple *C#* identifier.

These declarations declare the name of the corresponding type within the generated scanner. In the absence of naming declarations *gplex* generates a scanner as though it had seen the declarations —

```
%scannertype Scanner
%scanbasetype ScanBase
%tokentype Tokens
```

It is important to remember that the code of the scanner **defines** the scanner class name. The scanner base class and the token enumeration name are defined in the parser, so the corresponding naming declarations really are **declarations**. These declarations must synchronize with the definitions in the parser specification. The naming declaration syntax is identical in the *gplex* and *gppg* tools.

In the case of stand-alone scanners, which have no parser, all three naming declarations **define** the type names.

8.3.3 Start Condition Declarations

Start condition declarations define names for various *start conditions*. The declarations consist of a marker: "`%x`" for exclusive conditions, and "`%s`" for inclusive conditions, followed by one or more start condition names. If more than one name follows a marker, the names are comma-separated. The markers, as usual, must occur on a line starting in column zero.

Here is the full grammar for start condition declarations —

```
StartConditionsDeclaration
    : Marker   NameList
    ;
Marker
    : "%x"  |  "%s"
    ;
NameList
    : ident
    | NameList  ','  ident
    ;
```

Such declarations are used in the rules section, where they predicate the application of various patterns. At any time the scanner is in exactly one start condition, with each start condition name corresponding to a unique integer value. On initialization a scanner is in the pre-defined start condition "*INITIAL*" which always has value 0.

When the scanner is set to an *exclusive* start condition *only* patterns predicated on that exclusive condition are "active". Conversely, when the scanner is set to an *inclusive* start condition patterns predicated on that inclusive condition are active, and so are all of the patterns that are unconditional[11].

8.3.4 Lexical Category Definitions

Lexical category code defines named regular expressions that may be used as sub-expressions in the patterns of the rules section.

```
LexicalCategoryDefinition
    : ident   RegularExpression
    ;
```

The syntax of regular expressions is treated in detail in Section 9 A typical example might be —

```
digits [0-9]+
```

which defines *digits* as being a sequence of one or more characters from the character class '0' to '9'. The name being defined must start in column zero, and the regular expression defined is included for used occurrences in patterns. Note that for *gplex* this substitution is performed by tree-grafting in the *AST*, not by textual substitution, so each defined pattern must be a well formed regular expression.

8.3.5 Character Class Membership Predicates

Sometimes user code of the scanner needs to test if some computed value corresponds to a code-point that belongs to a particular character class.

```
CharacterClassPredicatesDeclaration
    : "%charClassPredicate"  NameList
;
```

NameList is a comma-separated list of lexical category names, which must denote character classes.

For example, suppose that some support code in the scanner needs to test if the value of some unicode escape sequence denotes a code point from some complicated character class, for example —

```
ExpandsOnNFC [ ...  ] // Normalization length not 1
```

[11] *gplex* follows the *Flex* semantics by **not** adding rules explicitly marked *INITIAL* to inclusive start states.

This is the set of all those unicode characters which do not have additive length in normalization form C. The actual definition of the set has been abstracted away.

Now *gplex* will generate the set from the definition (probably using the unicode database) at scanner generation time. We want to be able to look up membership of this set at scanner *runtime* from the data in the automaton tables. The following declaration —

```
%charClassPredicate ExpandsOnNFC
```

causes *gplex* to generate a public instance method of the scanner class, with the following signature —

```
public bool Is_ExpandsOnNFC(int codepoint);
```

This method will test the given code-point for membership of the given set.

In general, for every name N in the *NameList* a predicate function will be emitted with the name `Is_`N, with the signature —

```
public bool Is_N(int codepoint);
```

8.3.6 User Character Predicate Declaration

Character classes in *gplex* may be generated from any of the built-in character predicate methods of the *.NET* runtime, or any of the three other built-in functions that *gplex* itself defines (see Section 9.2.5).

If a user needs to make use of additional character class predicates, then the user may supply a *PE*-file containing a class which implements the *QUT.Gplex.ICharTestFactory* interface shown in Figure 30. The *GetDelegate* method of the interface should

Figure 30: Interface for user character predicates

```
namespace QUT.Gplex
{
    public delegate bool CharTest(int codePoint);

    public interface ICharTestFactory {
        CharTest GetDelegate(string name);
    }
}
```

return delegates which implement the predicate functions. These might either be user-written code, or existing library methods with matching signatures.

User character predicates are declared in the *LEX* specification with the following syntax.

UserCharacterPredicateDeclaration
: "`%userCharPredicate`" *ident* '`[`' *DottedName* '`]`' *DottedName*
;

This declaration associates the simple name of the *ident* with the method specified in the rest of the command. The first dotted name is the filename of the library in which the interface implementation is found. The second dotted name is the name of the class which implements the interface with the last component of the name being the argument which is sent to *GetDelegate*.

A use-example might be a *LEX* file containing the following —

```
%userCharPredicate Favorites [MyAssembly.dll]MyClass.Test
```

This states that the identifier *Favorites* is associated with the name *Test* in the named assembly. If, later in the specification, a character class is defined using the usual syntax —

```
FavoritesSet      [[:Favorites:]]
```

then the following will happen —

* *gplex* will look for the *PE*-file "`MyAssembly.dll`" in the current directory and, if successful, load it.
* *gplex* will use reflection to find the class *MyClass* in the loaded assembly.
* *gplex* will create an instance of the class, and cast it to the *ICharTestFactory* type.
* *gplex* will invoke *GetDelegate* with argument "`Test`".
* *gplex* will invoke the returned delegate for every codepoint in the unicode alphabet, to evaluate *FavoritesSet*.

If the *PE*-file cannot be found, or the assembly cannot be loaded, or the named class cannot be found, or the class does not implement the interface, or the returned delegate value is null, then an error occurs.

8.3.7 User Code in the Definitions Section

Any indented code, or code enclosed in "`%{`" ... "`%}`" delimiters is copied to the output file.

UserCode
: "`%{`" *CodeBlock* "`%}`"
| *IndentedCodeBlock*
;

As usual, the `%`-markers must start at the left margin.

CodeBlock is arbitrary *C#* code that can correctly be textually included inside a class definition. This may include constants, member definitions, sub-type definitions, and so on.

IndentedCodeBlock is arbitrary *C#* code that can correctly be textually included inside a class definition. It is distinguished from other declaratory matter by the fact that each line starts with whitespcace.

It is considered good form to always use the "`%{`" ... "`%}`" delimited form, so that printed listings are easier to understand for human readers.

8.3.8 Comments in the Definitions Section

Block comments, "`/* ... */`", in the definition section that begin in column zero, that is *unindented* comments, are copied to the output file. Any indented comments are taken as user code, and are also copied to the output file. Note that this is different behaviour to comments in the rules section.

Single line "`//`" comments may be included anywhere in the input file. Unless they are embedded in user code they are treated as whitespace and are never copied to the output. Consider the following user code fragment —

```
%{
        // This is whitespace
        void Foo()   // This gets copied
        {   // This gets copied
        }   // This is whitespace
%}
```

The text-span of the code block reaches from "`void`" through to the final right brace. Single line comments within this text span will be copied to the scanner source file. Single line comments outside this text span are treated as whitespace.

8.3.9 Option Declarations

The definitions section may include option markers with the same meanings as the command line options described in Section 2.1. Option declarations have the format —

OptionsDeclaration
: "`%option`" *OptionsList*
;
OptionsList
: *Option*
| *OptionsList* '`,`'$_{opt}$ *Option*
;

Options within the definitions section begin with the "`%option`" marker followed by one or more option specifiers. The options may be comma or white-space separated.

The options correspond to the command line options. Options within the definitions section take precedence over the command line options. A full list of options is in Section 15.

Some options make more sense on the command line than as hard-wired definitions, but all commands are available in both modalities.

8.4 The Rules Section

8.4.1 Overview of Pattern Matching

The rules section specifies the regular expression patterns that the generated scanner will recognize. Rules may be predicated on one or more of the start states from the definitions section.

Each regular expression declaration may have an associated *Semantic Action*. The semantic action is executed whenever an input sequence matches the regular expression. *gplex* always returns the *longest* input sequence that matches any of the applicable rules of the scanner specification. In the case of a tie, that is, when two or more patterns of the same length might be matched, the pattern which appears first in the specification is recognized.

The longest match rule means that *gplex*-created scanners sometimes have to "back up". This can occur if one pattern recognizes strings that are proper prefixes of some strings recognized by a second pattern. In this case, if some input has been scanned that matches the first pattern, and the next character could belong to the longer, second pattern, then scanning continues. If it should happen that the attempt to match the longer pattern eventually fails, then the scanner must back up the input and recognize the first pattern after all.

The main engine of pattern matching is a method named *Scan*. This method is an instance method of the scanner class. It uses the tables of the generated automaton

to update its state, invoke sematic actions whenever a pattern is matched, and return integer values to its caller denoting the pattern that has been recognized.

8.4.2 Overall Syntax of Rules Section

The marker "%%" delimits the boundary between the definitions and rules sections.

RulesSection
: *PrologCode*$_{opt}$ *RuleList* *EpilogCode*$_{opt}$
;
RuleList
: *RuleList*$_{opt}$ *Rule*
| *RuleList*$_{opt}$ *RuleGroup*
;
PrologCode
: *UserCode*
;
EpilogCode
: *UserCode*
;

The user code in the prolog and epilog may be placed in "%{" ... "%}" delimiters or may be an indented code block.

The *CodeBlock* of the optional prolog *UserCode* is placed at the start of the *Scan* method. It can contain arbitrary code that is legal to place inside a method body[12]. This is the place where local variables that are needed for the semantic actions should be declared.

The *CodeBlock* of the optional epilog *UserCode* is placed in a catch block at the end of the *Scan* method. This code is therefore guaranteed to be executed for every termination of *Scan*. This code block may contain arbitrary code that is legal to place inside a catch block. In particular, it may access local variables of the prolog code block.

Code interleaved *between* rules, whether indented or within the special delimiters, has no sensible meaning, attracts a warning, and is ignored.

8.4.3 Rule Syntax

The rules have the syntax —

Rule
: *StartConditionList*$_{opt}$ *RegularExpression* *Action*
;
StartConditionList
: '<' *NameList* '>'
| '<' '*' '>'
;
Action
: '|'
| *CodeLine*
| '{' *CodeBlock* "}"
;

[12] And therefore cannot contain method definitions, for example.

Start condition lists are optional, and are only needed if the specification requires more than one start state. Rules that are predicated with such a list are only active when (one of) the specified condition(s) applies. Rules without an explicit start condition list are implicitly predicated on the *INITIAL* start condition.

The names that appear within start condition lists must exactly match names declared in the definitions section, with just two exceptions. Start condition values correspond to integers in the scanner, and the default start condition *INITIAL* always has number zero. Thus in start condition lists "`0`" may be used as an abbreviation for *INITIAL*. All other numeric values are illegal in this context. Finally, the start condition list may be "`<*>`". This asserts that the following rule should apply in every start state.

The *Action* code is executed whenever a matching pattern is detected. There are three forms of the actions. An action may be a single line of *C#* code, on the same line as the pattern. An action may be a block of code, enclosed in braces. The left brace must occur on the same line as the pattern, and the code block is terminated when the matching right brace is found. Finally, the special vertical bar character, on its own, means "the same action as the next pattern". This is a convenient rule to use if multiple patterns take the same action[13].

Semantic action code typically loads up the *yylval* semantic value structure, and may also manipulate the start condition by calls to *BEGIN*`(NEWSTATE)`, for example. Note that *Scan* loops forever reading input and matching patterns. *Scan* exits only when an end of file is detected, or when a semantic action executes a "`return` *token*" statement, returning the integer token-kind value.

The syntax of regular expressions is treated in detail in Section 9

8.4.4 Rule Group Scopes

Sometimes a number of patterns are predicated on the same list of start conditions. In such cases it may be convenient to use *rule group scopes* to structure the rules section. Rule group scopes have the following syntax —

```
RuleGroup
    : StartConditionList  '{'  RuleList  '}'
    ;
StartConditionList
    : '<'  NameList  '>'
    | '<'  '*'  '>'
    ;
```

The rules that appear within the scope are all conditional on the start condition list which begins the scope. The opening brace of the scope must immediately follow the start condition list, and the opening and closing braces of the scope must each be the last non-whitespace element on their respective lines.

As before, the start condition list is a comma-separated list of known start condition names between '<' and '>' characters. The rule list is one or more rules, in the usual format, each starting on a separate line. It is common for the embedded rules within the scope to be unconditional, but it is perfectly legal to nest either conditional rules or rule group scopes. In nested scopes the effect of the start condition lists is cumulative. Thus —

[13] And this is not just a matter of saving on typing. When *gplex* performs state minimization two accept states are only able to be considered for merging if the semantic actions are the same. In this context "the same" means using the same text span in the lex file.

```
<one>{
    <two>{
        foo   { FooAction(); }
        bar   { BarAction(); }
    }
}
```

has exactly the same effect as —

```
<one,two>{
    foo   { FooAction(); }
    bar   { BarAction(); }
}
```

or indeed as the plain, old-fashioned sequence —

```
<one,two>foo   { FooAction(); }
<one,two>bar   { BarAction(); }
```

It is sensible to use indentation to denote the extent of the scope. So this syntax necessarily relaxes the constraint that rules must start at the beginning of the line.

Note that almost any non-whitespace characters following the left brace at the start of a scope would be mistaken for a pattern. Thus the left brace must be the last character on the line, except for whitespace. As usual, "whitespace" includes the case of a *C#*-style single-line comment.

8.4.5 Comments in the Rules Section

Comments in the rules section that begin in column zero, that is *unindented* comments, are not copied to the output file, and do not provoke a warning about "code between rules". They may thus be used to annotate the lex file itself.

Any *indented* comments *are* taken as user code. If they occur before the first rule they become part of the prolog of the *Scan* method. If they occur after the last rule they become part of the epilog of the *Scan* method.

Single line "`//`" comments may be included anywhere in the input file. Unless they are embedded in user code they are treated as whitespace and are never copied to the output.

8.5 The User Code Section

The user code section contains nothing but user code. Because of this, it is generally unnecessary to use the "`%{ ... %}`" markers to separate this code from declarative matter. All of the text in this section is copied verbatim into the definition for the scanner class.

Since *gplex* produces *C#* partial classes, it is often convenient to move all of the user code into a "scan-helper" file to make the lex input files easier to read.

9 Regular Expressions

9.1 Concatenation, Alternation and Repetition

Regular expressions are patterns that define languages of strings over some alphabet. They may define languages of finite or infinite cardinality. Regular expressions in *gplex* must fit on a single line, and are terminated by any un-escaped white space such as a blank character not in a character class.

9.1.1 Definitions

Regular expressions are made up of primitive atoms which are combined together by means of concatenation, alternation and repetition. Concatenation is a binary operation, but has an implicit application in the same way as some algebraic notations denote ab to mean "a multiplied by b".

If $\mathbf{R}_1$ and $\mathbf{R}_2$ are regular expressions defining languages $\mathbf{L}_1$ and $\mathbf{L}_2$ respectively, then $\mathbf{R}_1\mathbf{R}_2$ defines the language which consists of any string from $\mathbf{L}_1$ concatentated with any string from $\mathbf{L}_2$.

Alternation is a binary infix operation. It is denoted by the vertical bar character '`|`'. If $\mathbf{R}_1$ and $\mathbf{R}_2$ are regular expressions defining languages $\mathbf{L}_1$ and $\mathbf{L}_2$ respectively, then $\mathbf{R}_1$`|`$\mathbf{R}_2$ defines the language which consists all the strings from either $\mathbf{L}_1$ or $\mathbf{L}_2$.

Repetition is a unary operation. There are several forms of repetition with different markers. The plus sign '`+`' is used as a suffix, and denotes one or more repetitions of its operand. If **R** is a regular expressions defining language **L** then **R+** defines the language which consists one or more strings from **L** concatenated together. Note that the use of the word "repetition" in this context is sometimes misunderstood. The defined language is not repetitions of the *same* string from **L** but concatenations of any members of **L**.

9.1.2 Operator Precedence

The repetition markers have the highest precedence, concatenation next highest, with alternation lowest. Sub-expressions of regular expressions are grouped using parentheses in the usual way.

If '`a`', '`b`' and '`c`' are atoms denoting themselves, then the following regular expressions define the given languages.

`a`	defines the language with just one string { "`a`" }.
`a+`	defines the infinite language { "`a`", "`aa`", "`aaa`", ... }.
`ab`	defines the language with just one string { "`ab`" }.
`a|b`	defines the language with two strings { "`a`", "`b`" }.
`ab|c`	defines the language with two strings { "`ab`", "`c`" }.
`a(b|c)`	defines the language with two strings { "`ab`", "`ac`" }.
`ab+`	defines the infinite language { "`ab`", "`abb`", "`abbb`", ... }.
`(ab)+`	defines the infinite language { "`ab`", "`abab`", "`ababab`", ... }.

and so on.

9.1.3 Repetition Markers

There are three single-character repetition markers. These are —

* The suffix operator '`+`' defines a language which contains all the strings formed by concatentating one or more strings from the language defined by its operand on the left.

* The suffix operator '`*`' defines a language which contains all the strings formed by concatentating zero or more strings from the language defined by its operand on the left. If **R** is some regular expression, **R*** defines almost the same language as **R+**. The language defined using the "star-closure" contains just one extra element, the empty string "".

* The suffix operator '`?`' defines a language which concatentates zero or one string from the language defined by its operand on the left. If **R** is some regular expression, **R**`?` defines almost the same language as **R**. The language defined using the "optionality" operator contains just one extra element, the empty string "".

The most general repetition marker allows for arbitrary upper and lower bounds on the number of repetitions. The general repetition operator $\{N,M\}$, where N and M are integer constants, is is a unary suffix operator. When it is applied to a regular expression it defines a language which concatenates between N and M strings from the language defined by the operand on its left. It is an error if N is greater than M. If there is no upper bound, then the second numerical argument is left out, but the comma remains. Note however that the $\{N,\}$ marker must not have whitespace after the comma. In *gplex* un-escaped whitespace terminates the candidate regular expression.

If both the second numerical argument *and* the comma are taken out then the operator defines the language that contains all of the strings formed by concatenating exactly N (possibly different) strings from the language defined by the operand on the left.

We have the following identities for any regular expression **R** —

$$\begin{aligned}
\mathbf{R}\texttt{+} &= \mathbf{R}\{\texttt{1,}\} && \text{// \textit{One or more repetitions}}\\
\mathbf{R}\texttt{*} &= \mathbf{R}\{\texttt{0,}\} && \text{// \textit{Zero or more repetitions}}\\
\mathbf{R}\texttt{?} &= \mathbf{R}\{\texttt{0,1}\} && \text{// \textit{Zero or one repetition}}\\
\mathbf{R}\{N\} &= \mathbf{R}\{N\texttt{,}N\} && \text{// \textit{Exactly } } N \text{ \textit{repetitions}}
\end{aligned}$$

As may be seen, all of the simple repetition operators can be thought of as special cases of the general $\{N,M\}$ form.

It is an interesting but not very useful fact that, conversely, every instance of the general repetition form can be written in terms of concatenation, alternation, the '`*`' operator and the *empty language* which we denote ϵ. Here is a hint of the proof. First we have two shift rules that allow us to reduce the lower repetition count by one at each application, so long as the count remains non-negative —

$$\begin{aligned}
\mathbf{R}\{N\texttt{,}\} &= \mathbf{RR}\{N-1\texttt{,}\} && \text{// \textit{Start-index shift rule}}\\
\mathbf{R}\{N\texttt{,}M\} &= \mathbf{RR}\{N-1\texttt{,}M-1\} && \text{// \textit{Finite-index shift rule}}
\end{aligned}$$

After we have reduced the lower bound to zero, we can do an inductive step —

$$\begin{aligned}
\mathbf{R}\{\texttt{0,1}\} &= (\epsilon\texttt{|}\mathbf{R}) && \text{// \textit{Zero or one repetition}}\\
\mathbf{R}\{\texttt{0,2}\} &= (\epsilon\texttt{|}\mathbf{R}\texttt{|}\mathbf{RR}) && \text{// \textit{Zero, one or two repetitions}}\\
\ldots & && \text{// \textit{And so on ... with limit case} —}\\
\mathbf{R}\{\texttt{0,}\} &= \mathbf{R}\texttt{*} && \text{// \textit{Zero or more repetitions}}
\end{aligned}$$

Using this result we could, for example, write —

$$\begin{aligned}
\mathbf{R}\{\texttt{3,}\} &= \mathbf{RRRR}\texttt{*}\\
\mathbf{R}\{\texttt{3,5}\} &= \mathbf{RRR}(\epsilon\texttt{|}\mathbf{R}\texttt{ |}\mathbf{RR})
\end{aligned}$$

9.2 Regular Expression Atoms

9.2.1 Character Denotations

Characters that do not have a special meaning in a particular context, and which are represented in the *gplex* input alphabet are used to represent themselves. Thus the regular expression `foo` defines a language that has just one string: "foo".

Characters that have some format affect on the input must be escaped, so the usual control characters in *C#* are denoted as \\, \a, \b, \f, \n, \r, \t, \v, \0, exactly as in *C#*[14].

In contexts in which a particular character has some special meaning, that character must be escaped in the same way, by prefixing the character by a '\'.

To denote characters that cannot be represented by a single byte in the input file, various numerical escapes must be used. These are —

* *Octal escapes* '*ddd*' where the d are octal digits.
* *Hexadecimal escapes* '\x*hh*' where the h are hexadecimal digits.
* *Unicode escapes* '\u*hhhh*' where the h are hexadecimal digits.
* *Unicode escapes* '\U*hhhhhhhh*' where the h are hexadecimal digits.

In the final case the hexadecimal value of the codepoint must not exceed 0x10ffff.

Within a regular expressions the following characters have special meaning and must be escaped to denote their uninterpreted meaning —

'.', '"', '(', ')', '{', '}', '[', ']', '+', '*', '/', '|', ' '

This list is in addition to the usual escapes for control characters and characters that require numerical escapes.

The last character in the list is the space character. It appears here because a space signals the end of the regular expression in *gplex*.

9.2.2 Lexical Categories – Named Expressions

Lexical categories are named regular expressions that may be used as atoms in other regular expressions. Expressions may be named in the definitions section of the input file. Used occurrences of these definitions may occur in other named regular expressions, or in the patterns in the rules section. *gplex* implements a simple "*declaration before use*" scoping rule for such uses.

Used occurrences of lexical categories are denoted by the name of the expression enclosed in braces "{*name*}".

As an example, if we have named regular expressions for octal, hex and unicode escape characters earlier in the input file, we may define all the numerical escapes as a new named expression —

```
NumericalEscape  {OctalEscape}|{HexEscape}|{UnicodeEscape}
```

Roughly speaking, the *meaning* of a used occurrence of a named expression is obtained by substituting the named expression into the host expression at the location of the used occurrence. In the case of *gplex* the effect is as if the named expression is surrounded by parentheses. This is different to the earliest implementations of *LEX*, which performed a textual substitution, but is equivalent to the semantics of *Flex*.

This particular choice of semantics means that if we have an expression named as "keyword" say —

```
keyword     foo|bar
```

and then use this lexical category in another expression —

```
the{keyword}   // Expands as the(foo|bar), not as thefoo|bar
```

[14]Note however that the regular expression `\n` matches the *ASCII LF* character, while `\\n` matches the length-2 literal string which could be written either as `@"\n"` or as `"\\n"` in a *C#* source file.

The language defined by this expression contains two strings, { “thefoo”, “thebar”}. With the original *LEX* semantics the defined language would have contained the two strings { “thefoo”, “bar”}.

A consequence of this choice is that every named pattern must be a well-formed regular expression.

9.2.3 Literal Strings

Literal strings in the usual *C#* format are atoms in a regular expression.

The meaning of a literal string is exactly the same as the meaning of the regular expression formed by concatenating the individual characters of the string. For simple cases, enclosing a character sequence in quotes has no effect. Thus the regular expression `foo` matches the same pattern as the regular expression `"foo"`.

However there are two reasons for using the string form: first, a string is an atom, so the regular expression `"foo"+` defines the language { “foo”, “foofoo”, ...}, while the regular expression `foo+` defines the language { “foo”, “fooo”, “foooo” ...}. Secondly, the only ordinary character that must be escaped within a literal string is ‘"’, together of course with the control characters and those requiring numerical escapes. This may make the patterns much more readable for humans.

Literal strings must be terminated by a closing double-quote before the end of line, or an error is reported. The verbatim literal string format of *C#* may be used in embedded *C#* code, but cannot form part of a regular expression.

9.2.4 Character Classes

Character classes are sets of characters. When used as atoms in a regular expression they match any character from the set. Such sets are defined by a bracket-enclosed list of characters, character-ranges and character predicates. There is no punctuation in the list of characters, so the definition of of a named expression for the set of the decimal digits could be written —

```
digits      [0246813579]
```

The digits have deliberately been scrambled to emphasise that character classes are unordered collections, and the members may be added in any order.

For sets where *ranges* of contiguous characters are members, we may use the character range mechanism. This consists of the first character in the range, the dash character ‘-’, and the last character in the range. The same set as the last example then could have been written as —

```
digits      [0-9] // Decimal digits
```

It is an error if the ordinal number of the first character is greater than the ordinal number of the last character.

We can also define *negated* sets, where the members of the set are all those characters *except* those that are listed as individual characters or character ranges. A negated set is denoted by the caret character ‘^’ as the first character in the set. Thus, all of the characters *except* the decimal digits would be defined by —

```
notDigit    [^0-9] // Everything but digits
```

Within a character class definition the following characters have special meaning: ‘]’, marking the end of the set; ‘-’, denotes the range operator, except as the first or last character of the set; ‘^’, denotes set inverse, but only as the first character in the set. In

all locations where these characters have their special meaning they must be escaped in order to denote their literal character value.

The dash character - does not need escaping if it is the first or last character in the set, but *gplex* will issue a warning to make sure that the literal meaning was intended.

The usual control characters are denoted by their escaped forms, and all of the numerical escapes may be used within a character class.

9.2.5 Character Class Predicates

Some of the character classes that occur with unicode scanners are too large to easily define explicitly. For example, the set of all those unicode codepoints which (according to *ECMA-334*) are possible first characters of a *C#* identifier contains 92707 characters which appear in 362 ranges.

Within a character class, the special syntax "`[:`*PredicateMethod*`:]`" denotes all of the characters from the selected alphabet[15] for which the corresponding *.NET* base class library method returns the true value. The implemented methods are —

* *IsControl, IsDigit, IsLetter, IsLetterOrDigit, IsLower, IsNumber, IsPunctuation, IsSeparator, IsSymbol, IsUpper, IsWhiteSpace*

There are three additional predicates built into *gplex* —

* *IsFormatCharacter* — Characters with unicode category Cf
* *IdentifierStartCharacter* — Valid identifier start characters for *C#*
* *IdentifierPartCharacter* — Valid continuation characters for *C#* identifiers, excluding category Cf

Note that the bracketing markers "`[:`" and "`:]`" appear within the brackets that delimit the character class. For example, the following two character classes are equal.

```
alphanum1 [[:IsLetterOrDigit:]]
alphanum2 [[:IsLetter:][:IsDigit:]]
```

These classes are *not* equivalent to the set —

```
alphanum3 [a-zA-Z0-9]
```

even in the 8-bit case, since this last class does not include all of the alphabetic characters from the latin alphabet that have diacritical marks, such as ä and ñ.

These character predicates are intended for use with unicode scanners. Their use with byte-mode scanners is complicated by the code page setting of the host machine. For futher information on this, see the section "*Character Predicates in Byte-Mode Scanners*" in the Part III

New in version 1.0.2 of *gplex* is the ability for users to define their own character predicate functions. This feature is specified in Section 8.3.6.

[15] In the non-unicode case, the sets will include only those byte values that correspond to unicode characters for which the predicate functions return true. In the case of the /unicode option, the full sets are returned.

9.2.6 The Dot Metacharacter

The "dot" character, '`.`', has special meaning in regular expressions. It means *any character except* '`\n`'. This traditional meaning is retained for *gplex*.

The "dot" is often used to cause a pattern matcher to match everything up to the end-of-line. It works perfectly for files that use the *UNIX* end-of-line conventions. However, for maximum portability in unicode scanners it is better for the user to define a character class which is *any character except* any *of the unicode end-of-line characters*. This set can be defined by —

```
any     [^\r\n\u0085\u2028\u2029]
```

Given this definition, the character class {`any`} can be used any place where the traditional dot would have been used.

9.2.7 Context Markers

The context operators of *gplex* are used to declare that particular patterns should match only if the input immediately preceeding the pattern (the *left context*) or the input immediately following the pattern (the *right context*) are as requested.

There are three context markers: *left-anchor* '`^`', *right-anchor* '`$`', and the *right context* operator "`/`".

A left-anchored pattern `^`**R**, where **R** is some regular expression, matches any input that matches **R**, but only if the input starts at the beginning of a line. Similarly, a right-anchored pattern **R**`$`, where **R** is some regular expression, matches any input that matches **R**, but only if the input finishes at the end of a line. Traditional implementations of *LEX* define "end of the line" as whatever the *ANSI C* compiler defines as end of line. *gplex* accepts any of the standard line-end markers. For byte-mode scanners, either '`\n`' or '`\r`' will match the right-anchor condition. For unicode-mode scanners the right-anchor character set is "`[\n\r\x85\u2085\u2086]`".

The expression $\mathbf{R}_1$`/`$\mathbf{R}_2$ matches text that matches $\mathbf{R}_1$ with right context matching the regular expression $\mathbf{R}_2$. The entire string matching $\mathbf{R}_1\mathbf{R}_2$ participates in finding the longest matching string, but only the text corresponding to $\mathbf{R}_1$ is consumed. Similarly for right anchored patterns, the end of line character(s) participate in the longest match calculation, but are not consumed.

It is a limitation of the current gplex *implementation that when the right-context operator is used, as in* $\mathbf{R}_1$`/`$\mathbf{R}_2$ *at least one of* $\mathbf{R}_1$ *or* $\mathbf{R}_2$ *must define a language of constant length strings.*

9.2.8 End-Of-File Marker

Finally, there is one more special marker that *gplex* recognizes. The character sequence "`<<EOF>>`" denotes a pattern that matches the end-of-file. The marker may be conditional on some starting condition in the usual way, but cannot appear as a component of any other pattern. Beware that pattern `"<<EOF>>"` (with the quotes) exactly matches the seven-character-long pattern "`<<EOF>>`", while the pattern `<<EOF>>` (without the quotes) matches the end-of-file.

10 Special Symbols in Semantic Actions

10.1 Properties of the Matching Text

10.1.1 The yytext Property

Within the semantic action of a pattern **R**, this read-only property returns a `string` containing the input text that matches **R**.

If a semantic action calls the *yyless* method, it will modify *yytext*. In the case of a pattern with right-context, the string has already had the right context trimmed.

10.1.2 The yyleng Property

The *yyleng* property returns the length of the input text that matched the pattern. It is a read-only property.

The length is given in codepoints, that is, logical characters. For many text file encodings *yyleng* is less than the number of bytes read. Even in the case of string input the number of codepoints will be less than the number of `char` values, if the string contains surrogate pairs.

10.1.3 The yypos Property

The *yypos* property returns the position of the input file buffer at the start of the input text that matched the pattern. It is a read-only property.

Although *yypos* returns an integer value, it should be treated as opaque. In particular, arithmetic using *yypos* and *yyleng* will not behave as expected.

10.1.4 The yyline Property

The *yyline* property returns the line-number at the start of the input text that matched the pattern. It is a read-only property. Line numbers count from one.

10.1.5 The yycol Property

The *yycol* property returns the column-number at the start of the input text that matched the pattern. It is a read-only property. Column numbers count from zero at the start of each line.

10.2 Looking at the Input Buffer

Every *gplex*-generated scanner has an accessible buffer object as a field of the scanner object. There are many different buffer implementations, all of which derive from the abstract *ScanBuff* class.

The last character to be read from the buffer is stored within the scanner in the field *code*.

10.2.1 Current and Lookahead Character

When a pattern has been matched, the scanner field *code* holds the codepoint of the last character to be read. This is an integer value. The value is not part of the current pattern, but will be the first character of the input text that the scanner matches *next*.

In every case *code* is the input character that follows the last character of *yytext*. Thus for patterns with right context *code* is the first character of the context, and calls to *yyless* that discard characters will change the value.

Buffer implementations in version 1.1.0 of *gplex* do not contain a buffer lookahead *Peek* method. This method now exists as a private method in the scanner class. The new method always returns a valid unicode code point, or the special end-of-file value.

10.2.2 The yyless Method

After a scanner has matched a pattern, the *yyless* method allows some or all of the input text to be pushed back to the buffer.

```
void yyless(int len); // Discard all but the first len characters
```

Following this call, *yytext* will be *len* characters long, and *buffer.Pos*, *yyleng* and *code* will have been updated consistently.

This method can either trim *yytext* to some fixed length, or can cut a fixed length suffix from the text. For example, to push back the last character of the text *yyless*(*yyleng*−1) should be called.

A useful idiom when changing from one start condition to another is to recognize the pattern that starts the new phase, change the start condition, and call *yyless*(0). In that way the starting pattern is scanned again in the new condition. Here is an example for scanning block comments. The scanner has a *CMNT* start condition, and the relevant rules look like this —

```
\/\*  BEGIN(CMNT); yyless(0); // No return!
<CMNT>...
```

Note that both the slash *and* the star characters must be escaped in the regular expression.

In this way, the *CMNT* "mini-scanner" will get to see *all* of the comment, including the first two characters. It is then possible for the comment scanner to return with a *yytext* string that contains the whole of the comment.

10.2.3 The yymore Method

This method is not implemented in the current version of *gplex*.

10.3 Changing the Start Condition

10.3.1 The *BEGIN* Method

The *BEGIN* method sets a new start condition. Start conditions correspond to constant integer values in the scanner. The initial condition always has value one, but the values assigned by *gplex* to other start conditions is unpredictable. Therefore the argument passed to the call of *BEGIN* should always be the *name* of the start condition, as shown in the example in the discussion of *yyless*.

10.3.2 The *YY_START* Property

YY_START is a read-write property that gets or sets the start condition. Setting *YY_START* to some value X is precisely equivalent to calling *BEGIN*(*X*).

Reading the value of *YY_START* is useful for those complicated scenarios in which a pattern applies to multiple start conditions, but the semantic action needs to vary

depending on the actual start condition. Code of the following form allows this behavior —

```
SomePattern  { if (YY_START == INITIAL)
                    ... else ...
             }
```

Another scenario in which *YY_START* is used is those applications where the parser needs to manipulate the start condition of the scanner. The *YY_START* property has `internal` accessibility, and hence may be set by a parser in the same *PE*-file as the scanner.

10.4 Stacking Start Conditions

For some applications the use of the standard start conditions mechanism is either impossible or inconvenient. The lex definition language itself forms such an example, if you wish to recognize the *C#* tokens as well as the lex tokens. We must have start conditions for the main sections, for the code inside the sections, and for comments inside (and outside) the code.

One approach to handling the start conditions in such cases is to use a *stack* of start conditions, and to push and pop these in semantic actions. *gplex* supports the stacking of start conditions when the "`stack`" command is given, either on the command line, or as an option in the definitions section. This option provides the methods shown in Figure 31. These are normally used together with the standard *BEGIN* method. The

Figure 31: Methods for Manipulating the Start Condition Stack

```
// Clear the start condition stack
internal void yy_clear_stack();

// Push currentScOrd, and set currentScOrd to "state"
internal void yy_push_state(int state);

// Pop start condition stack into currentScOrd
internal int yy_pop_state();

// Fetch top of stack without changing top of stack value
internal int yy_top_state();
```

first method clears the stack. This is useful for initialization, and also for error recovery in the start condition automaton.

The next two methods push and pop the start condition values, while the final method examines the top of stack without affecting the stack pointer. This last is useful for conditional code in semantic actions, which may perform tests such as —

```
if (yy_top_state() == INITIAL) ...
```

Note carefully that the top-of-stack state is not the current start condition, but is the value that will *become* the start condition if "pop" is called.

10.5 Miscellaneous Methods

10.5.1 The *ECHO* Method

This method echos the recognized text to the standard output stream. It is equivalent to

```
System.Console.Write(yytext);
```

Part III
Using Unicode

11 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 code-points. 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 code page*" 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.

11.1 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 code page with the specified number. If there is no such code page an exception is thrown and processing terminates.

/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 code page with the specified name. If there is no such code page an exception is thrown and processing terminates.

/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 code page of the host machine. 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 code page. On the basis of this scan result the scanner will use either the default code page on the host machine, or interpret the input as a *utf-8* file. See Section 12.5 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 equivalence 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`".

11.2 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 code page 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 code page *gplex* must know what that code page 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 code page mapping to unicode characters for which the predicate is true. The nominated code page 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 code page mapping to unicode characters for which the predicate is true. The nominated code page 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 code page 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.

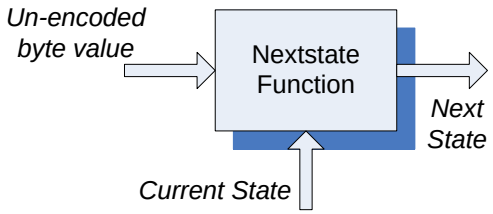
Caution

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 code page of the scanner host machine must be known at scanner generation time rather than at scanner runtime (see also section 12.2).

12 Specifying Scanners

The scanning engine that *gplex* produces is a finite state automaton (*FSA*)[16] This *FSA* deals with codepoints from either the *ASCII* or *Unicode* alphabet. Byte-mode scanners have the conceptual form shown in Figure 32 (repeated from Figure 9 in Part I). The

Figure 32: Conceptual diagram of byte-mode scanner



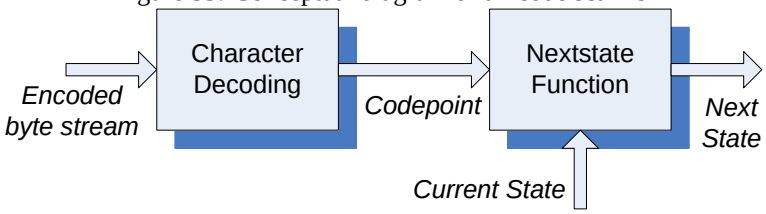
un-encoded byte values of the input are used by the "next-state" function to compute the next state of the automaton.

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

[16](*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.

scanner is shown in Figure 33 for the case of file input. (This figure is repeated here from Figure 10 in Part I). The corresponding diagram for *string* input differs only in

Figure 33: Conceptual diagram of unicode scanner



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.

12.1 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 code pages may ascribe different meanings to character literals for the upper 128 characters. Byte-mode scanners are inherently non-portable across cultures.

12.2 Character Class 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 32. 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 code page.

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 code page 1252, this set will contain 126 members. The same set has only 123 members in code page 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 code page that maps between the byte-values that the generated scanner reads, and the unicode codepoints to which they correspond.

If no code page is specified, the mapping is taken from the default code page 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 code page of the scanner host is known with certainty at scanner generation time. Other cases may lack portability.

12.3 Unicode Mode Scanners

The unicode standard ascribes unique 21-bit *codepoints* for every defined character[17]. Thus, if we want to recognize *both* the 'ß' character *and* the '$\acute{\imath}$' character then we must use a unicode scanner. In unicode ß has codepoint u+00df, while $\acute{\imath}$ 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 33. 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 $\acute{\imath}$ 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 $\acute{\imath}$ character will be denoted by the two bytes {0xce, 0x9f}.

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 code page 1252 or code page 1253. Of course, such a file cannot contain both ß and $\acute{\imath}$ 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[18]? Note that unlike the "utf-with-prefix" cases there is no certain way to know what code page 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 code page 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

[17] 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.

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

according to the default code page 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 code page 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 code page that is different to the host machine, then that known code page 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 code page 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 code page fallback when it is known that a particular input file originates from another culture.

12.4 Overriding the Codepage Fallback at Application Runtime

The fallback code page 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 code page changed for a particular input file when the encoding of that file is known.

Scanners generated by *gplex* implement a static method with the following signature —

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

This method takes a string argument, which is a code page-setting command from the calling application. If the command begins with the string "*code page:*" this prefix is removed, and the remaining string is converted to a code page index. The command may specify either a code page 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 code page of the host machine. Finally, "guess" attempts to determine the code page from the byte-patterns in the file. These semantics are the same as the */codePage:* option of *gplex*, which indeed invokes this same method.

The method is found in the buffer code of the generated scanner. If the */noEmbedBuffers* option is in force the method will be in the class *QUT.GplexBuffers.CodePageHandling*. For the default, embedded buffer case, the class *CodePage-Handling* is directly nested in the same namespace as the *Scanner* class.

There are two constructors for the scanner objects in each unicode 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 code page. The second constructor passes the string argument to *GetCodePage*, and then sends the resulting

integer to the appropriate call of *SetSource*[19]. Alternatively, the application may directly call *SetSource* itself, as shown below.

An application program that wishes to set the fallback code page of its scanner on a file-by-file basis should follow the example of the schema in Figure 34. If the

Figure 34: Using the *GetCodePage* method

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

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

12.5 Adaptively Setting the Codepage

There are occasions in which it is not possible to predict the code page 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.

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 code page. 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* code page and the default code page 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

[19] In the case of byte-mode scanners there is no fallback code page, so only the first constructor is generated.

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 code page 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.

13 Input Buffers

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

The *ScanBuff* class in Figure 35 is the abstract base class of the stream and string

Figure 35: Features of the *ScanBuff* Class

```
public abstract class ScanBuff {
    ...
    public abstract int  Pos { get; set; }
    public abstract int  Read();
    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 character index in the underlying input stream.

The method *Read* returns an integer corresponding to the ordinal value of the next character, and advances the input position by one or more input elements. *Read* returns –1 for end of file.

New buffers are created by calling one of the *SetSource* methods of the scanner class. The signatures of these methods are shown in Figure 36, repeated here from Figure 6.

13.1 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.

Figure 36: 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);
```

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 —

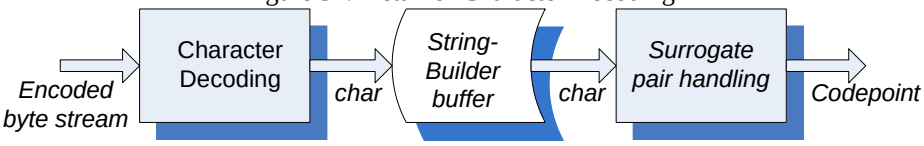
```
public void  SetSource(IList<string> list);
```

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.

13.2 File Input Buffers

All file input to *gplex* is held in a buffer of the *BuildBuffer* class. In every case the sequence of bytes in the file is transformed into a sequence of code points supplied

Figure 37: Detail of Character Decoding



to the scanner by the scheme shown in Figure 37, repeated here from Figure 13. The various generation-time options and scanner-runtime code page settings simply modify the processing in the rectangular boxes of the figure.

The various possibilities are —

* *GetBuffer* is called with a single, *Stream* argument. This is the only possibility in the case of a byte-mode scanner. In this case the character decoding is trivial, with the bytes of the stream added unmodified to the buffer. In this case surrogate pairs cannot arise, so the right-hand box in the figure is empty also.

* *GetBuffer* is called with the *Stream* and a fallback code page argument. In all such cases the scanner checks if the stream begins with a valid *utf*-prefix. If a prefix is detected an appropriate *StreamReader* object is created, and transforms the bytes of the stream to the characters in the buffer. The buffer is filled with block-read operations rather than character by character.

* If the two-argument version of *GetBuffer* is called but no prefix is found then there are three special cases, and a general case. The general case is to create a *StreamReader* using whatever encoding is specified by the fallback code page. The three special cases are the distinguished fallback values “raw”, “default” and “guess”. The raw value reverts to byte-mode decoding. The default value uses the default code page of the runtime host machine. Finally, the guess value causes the entire file to be scanned before a choice is made between *utf-8* and the default code page of the host machine. See also the discussion in section 12.5.

Part IV
Appendices

14 Appendix A: Tables

14.1 Keyword Commands

Keyword	Meaning
`%x`	This marker declares that the following list of comma-separated names denote exclusive start conditions.
`%s`	This marker declares that the following list of comma-separated names denote inclusive start conditions.
`%using`	The dotted name following the keyword will be added to the namespace imports of the scanner module.
`%namespace`	This marker defines the namespace in which the scanner class will be defined. The namespace argument is a dotted name. This marker must occur exactly once in the definition section of every input specification.
`%option`	This marker is followed by a list of option-names, as detailed in section 15. The list elements may be comma or white-space separated.
`%charClassPredicate`	This marker is followed by a comma-separated list of character class names. The class names must have been defined earlier in the text. A membership predicate function will be generated for each character class on the list. The names of the predicate functions are generated algorithmically by prefixing "*Is_*" to the name of each character class.
`%userCharPredicate`	This marker is followed by a simple identifier and the designator of a user-supplied *CharTest* delegate. When the identifier is used in a character class definition *gplex* will call the user delegate to evaluate the character class at scanner creation time. See section 8.3.6 for usage rules.
`%visibility`	This marker controls the visibility of the *Scanner* class. The permitted arguments are `public` and `internal`. The default is `public`.
`%scannertype`	The identifier argument **defines** the scanner class name, overriding the default *Scanner* name.
`%scanbasetype`	The identifier argument **declares** the name of the scanner base class defined by the parser. This overrides the *ScanBase* default.
`%tokentype`	The identifier argument **declares** the name of the token enumeration type defined by the parser. This overrides the *Tokens* default.

14.2 Semantic Action Symbols

Certain symbols have particular meanings in the semantic actions of *gplex* parsers. As well as the symbols listed here, methods defined in user code of the specification or its helper files will be accessible.

Symbol	Meaning
`yytext`	A read-only property which lazily constructs the text of the currently recognized token. This text may be invalidated by subsequent calls of *yyless*.
`yyleng`	A read-only property returning the number of symbols of the current token. In the unicode case this is not necessarily the same as the number of characters or bytes read from the input.
`yypos`	A read-only property returning the buffer position at the start of the current token.
`yyline`	A read-only property returning the line number at the start of the current token.
`yycol`	A read-only property returning the column number at the start of the current token.
`yyless`	A method that truncates the current token to the length given as the `int` argument to the call.
`BEGIN`	Set the scanner start condition to the value nominated in the argument. The formal parameter to the call is of type `int`, but the method is always called using the symbolic name of the start state.
`ECHO`	A no-arg method that writes the current value of *yytext* to the standard output stream.
`YY_START`	A read-write property that gets or sets the current start ordinal value. As with *BEGIN*, the symbolic name of the start condition in normally used.
`yy_clear_stack` ‡	This no-arg method empties the start condition stack.
`yy_push_state` ‡	This method takes a start condition argument. The current start condition is pushed and the argument value becomes the new start condition.
`yy_pop_state` ‡	This method pops the start condition stack. The previous top of stack becomes the new start state.
`yy_top_of_stack` ‡	This function returns the value at the top of the start condition stack. This is the value that would become current if the stack were to be popped.

‡ This method only applies with the */stack* option.

15 Appendix B: *GPLEX* Options

15.1 Informative Options

The following options are informative, and cannot be negated —

`help`	Send the usage message to the console
`codePageHelp`	Send help for the code page options to the console
`out:`*out-file-path*	Generate a scanner output file with the prescribed name
`frame:`*frame-file-path*	Use the specified frame file instead of seeking "gplexx.frame" on the built-in search path
`codePage:`*code-page-arg*	For unicode scanners: deal with input files that have no *UTF* prefix in the nominated way. For byte-mode scanners: interpret the meaning of character class predicates according to the encoding of the nominated code page.
`errorsToConsole`	By default *gplex* generates error messages that are interpreted by Visual Studio. This command generates error messages in the legacy format, in which error messages are sent to the console preceeded by the text of the source line to which they refer.

15.2 Boolean Options

The following options correspond to Boolean state flags within *gplex*. They can each be negated by prefixing "`no`" to the command name —

Option	Meaning	Default
`babel`	Include interfaces for Managed Babel framework	*default is noBabel*
`caseInsensitive`	Create a case-insensitive scanner	*default is noCaseInsensitive*
`check`	Compute the automaton, but do not create an output file	*default is noCheck*
`classes`	Use character equivalence classes in the automaton	*unicode default is classes*
`compress`	Compress all tables of the scanner automaton	*default is compress*
`compressMap`	Compress the equivalence class map	*unicode default is compressMap*
`compressNext`	Compress the next-state table of the scanner	*default is compressNext*
`embedBuffers`	Embed buffer code in the scanner namespace	*default is embedBuffers*
`files`	Provide file-handling code in scanners	*default is files*

Table continues on next page...

Boolean Options Continued ...

Option	Meaning	Default
`listing`	Generate a listing, even when there are no errors	*default is noListing*
`minimize`	Minimize the number of states of the automaton	*default is minimize*
`parseOnly`	Check the input, but do not construct an automaton	*default is noParseOnly*
`parser`	Expect type definitions from a host parser	*default is parser*
`persistBuffer`	Do not reclaim buffer space during scanning	*default is persistBuffer*
`stack`	Allow for start conditions to be stacked	*default is noStack*
`squeeze`	Generate the automaton with the smallest tables	*default is noSqueeze*
`summary`	Write out automaton statistics to the listing file	*default is noSummary*
`unicode`	Generate a unicode-mode (not byte-mode) scanner	*default is noUnicode*
`verbose`	Send *gplex*' progress information to the console	*default is noVerbose*
`version`	Send *gplex* version details to the console	*default is noVersion*

16 Appendix C: Pushing Back Input Symbols

When the client of a *gplex*-generated scanner is a backtracking parser[20] the scanner may be required to push back symbols from the input stream. Since version 1.1.8 *gplex* is distributed with helper code to facilitate this.

The code consists of three parts —

* *gppg* defines a *ScanObj* class that wraps a token integer value, along with its associated semantic value and position information.
* An *API* for enqueueing and dequeueing lookahead symbols, and manipulating the queueing data structure.
* Prolog code for the *Scan* method, to dequeue pushed-back symbols before any further symbols are read from the input buffer.

Whenever a parser requires to look ahead in the sequence of input symbols it needs a mechanism to return to the sequence position prior to that lookahead. One way to do this would be to save the scanner state prior to the lookahead, and afterward restore that state. However, this operation is complicated by the possibility that the scanner may have changed start state or even input source, and the fact that the input byte stream must be precisely restored in the case of multi-byte character encodings. As well, any creation or processing of semantic value objects will need to be repeated when the input is read a second time.

An alternative is to save the lookahead tokens, and push them back to a suitable data structure within the scanner. Whenever the scanner method *yylex*() is called the scanner will return pushed back symbols before switching back to reading new symbols from the input source.

16.1 The *ScanObj* Class

When tokens are pushed back to the scanner, it is necessary to also preserve any associated semantic and location values. The *ScanObj* class has three fields: the token integer value, the semantic value and the location value. The type of the semantic value is the "`TValue`" type argument of the generic *AbstractScanner*<*TValue*,*TSpan*> class from which all scanners are derived. The type of the location value is the "`TSpan`" type argument of *AbstractScanner*. The definition of this class is created by *gppg* which, of course, knows what the type arguments are in any particular case. For the default type arguments, the definition of *ScanObj* is given in Figure 38.

16.2 Prolog for the Scan Method

The only difference between a *gplex* scanner which allows token pushback and one that does not, is a prolog to the scanner's *Scan* method. The prolog tests if the pushback queue is empty. If so, the normal *Scan* method body is executed, fetching a symbol from the current input source buffer. If the pushback queue is not empty, *Scan* dequeues a *ScanObj*, and sets the return token value, *yylval* and *yylloc*. The code of the prolog is declared in the scanner specification, as shown in Figure 39.

The position of this code in the generated scanner may be seen from Figure 11.

[20] Section 4.5 of this documentation dealt with backtracking states of *gplex scanners*. For scanners backtracking is commonplace and is automatically generated by *gplex*. For *parsers* backtracking is avoided whenever possible, but is sometimes necessary.

Figure 38: Default *ScanObj* Definition

```
internal class ScanObj {
    public int token;
    public int yylval;
    public LexLocation yylloc;

    public ScanObj(int t, int v, QUT.Gppg.LexLocation l) {
        this.token = t; this.yylval = v; this.yylloc = l;
    }
}
```

Figure 39: "lex" Specification with *Scan* prolog

```
// Start of definition section
    Definitions go here.

%% // Start of pattern section

%{
    // Scan prolog: Code to take tokens from non-empty queue
    if (this.pushback.QueueLength > 0) {
        ScanObj obj = this.pushback.DequeueCurrentToken();
        this.yylval = obj.yylval;
        this.yylloc = obj.yylloc;
        return this.token;
    }
%}

Regular expressions define patterns here.

%% // Start of user code section

User code, if any, goes here.
```

The prolog code appears in the position of the line "optional user supplied prolog". It may be usefule to know that in the case of a specification that defines a scan method epilog, typically to construct the location text span, the prolog return does *not* traverse the epilog code. Thus the setting of the *yylval*, *yylloc* values in Figure 39 is safe[21].

16.3 The Pushback Queue API

The *API* of the pushback queue is defined by the *PushbackQueue* class. The *API* is shown in Figure 40. The class is a generic class of one type parameter. The class is instantiated for the particular *ScanObj* that is required. The class is declared in the *QUT.Gppg* namespace, and the source is found in the "`LookaheadHelper.cs`" file in

[21] If the user has defined an epilog the tool-generated body of *Scan* lies within a try block. The finally block of this try is the user-defined epilog. However, the user-defined prolog is *outside* the try block so a return in the prolog does not execute the epilog.

Figure 40: Lookahead Helper API

```
namespace QUT.Gppg {
    internal class PushbackQueue<Obj> {
        public delegate Obj GetObj(); // Fetch ScanObj from host scanner
        public delegate Obj TokWrap(int tok); // Wrap given token value

        public static PushbackQueue<Obj>
            NewPushbackQueue(GetObj getObj, TokWrap tokWrap);
        public int QueueLength { get; } // Number of queued ScanObj
        public Obj EnqueueAndReturnInitialSymbol(int token);
        public Obj GetAndEnqueue(); // Enqueue new ScanObj from scanner
        public void AddPushbackBufferToQueue();
        public void AbandonPushback();
        public Obj DequeueCurrentToken(); // Helper for Scan prolog
    }
}
```

the *gppg* distribution.

The class defines two delegate types. One value of each of these types is passed as argument to the factory method *NewPushbackQueue* that creates a pushback queue. This factory method is called just once for each parser instance, during scanner initialization.

The first delegate, *getObj*, invokes *yylex*() on the host scanner, and constructs a *ScanObj* from scanner state. The second delegate, *tokWrap*, takes an integer argument and constructs a *ScanObj* with that token value. Both are necessary, since the pushback queue may begin with an object corresponding to the usual one-symbol lookahead, and then continue with objects arising from fresh invocations of *yylex*().

Figure 41 is a typical initialization of the pushback data structure. The two argu-

Figure 41: Typical *PushbackQueue* Initialization

```
using QUT.Gppg;

internal sealed partial class Scanner {
  ... // Other feature declarations
  public PushbackQueue<ScanObj> pushback;

  public void Init( ...  )  {
    ... // Other initializations
    this.pushback = PushbackQueue<ScanObj>.NewPushbackQueue(
      () => new ScanObj(this.yylex(),this.yylval,this.yylloc),
      (int t) => new ScanObj(t, this.yylval, this.yylloc));
  }

  ... // Rest of scanner partial class
}
```

ments to *NewPushbackQueue* are anonymous methods, each of which calls the *ScanObj*

constructor. These lambda methods encapsulate the necessary reference to the host scanner object.

There are two methods in the *API* which return *ScanObj* objects. These are invoked by user-specified semantic actions of the parser that trigger lookahead actions. The first method, *EnqueueAndReturnInitialSymbol*, initializes a new, empty pushback buffer to store all of the symbols read during the lookahead process. The method is passed the current parser *NextToken* value. This value may be zero, indicating that the parsing engine has not yet fetched the lookahead symbol, or may be the token value of the lookahead symbol. The method fetches the next symbol, if necessary, and enqueues the object in the pushback buffer. The second method, *GetAndEnqueue* fetches further input symbols after the initial lookahead symbol, adding them to the pushback buffer.

A lookahead semantic action sequence is terminated by calling the method *AddPushbackBufferToQueue*. Because a lookahead action may be terminated without exhausting the pushed back symbols of a previous lookahead, the newly created pushback buffer is added to front of the existing queue.

For the special case where a single symbol of lookahead was sufficient it is possible to adopt a slightly more lightweight termination strategy. The scanner action may write the lookahead token to the parser *NextToken* variable, and invoke the method *AbandonPushback*. This avoids all queue manipulation.

16.4 Starting and Stopping Lookahead

The ways in which these facilities are used to program around non-*LALR(1)* constructs in a grammar are beyond the scope of these notes. However, here is a simple example of the use of these facilities to perform one extra symbol of lookahead.

Suppose some special action needs to be taken following some particular reduction, but only if the following two symbols are x followed by y. We assume that the parser has a property *QueueRef* that returns a reference to the *PushbackQueue* object. Figure 42 is the outline of a possible semantic action.

Figure 42: Semantic action with length-2 lookahead

```
ScanObj next = QueueRef.EnqueueAndReturnSymbol(NextToken);
if (next.token = (int)LexEnum.x) {
   next = QueueRef.GetAndEnqueue()
      if (next.token = (int)LexEnum.y)
         do_something_special();
}
QueueRef.AddPushbackBufferToQueue();
```

16.5 Summary: How to use Symbol Pushback

In order create a *gppg* parser that uses symbol pushback to program around non-*LALR*(1) grammar features the necessary steps are —

* Create a *gplex* scanner that supports symbol pushback.
* Modify the grammar so that the *gppg* parser performs symbol lookahead to resolve conflicts.

16.5.1 Creating a Scanner Supporting Symbol Pushback

* Include the file *LookaheadHelper.cs* in your project. This file is in folder `project\SpecFiles` in the distribution.
* Declare a scanner field *pushback* in the scanner class, and initialize the field at scanner instantiation. (Cut and paste template from helper file.)
* Add the prolog code to the lex file from which the scanner is generated. (Cut and paste template from helper file.)

16.5.2 Modify the Grammar to Perform *ad hoc* Lookahead

At each point in the grammar at which it is necessary to perform an *ad hoc* lookahead choose a reduction that marks arrival of the parse at that particular decision point.

In the following it is assumed that the *PushbackQueue* object is accessible through some parser property named *QueueRef*.

Begin the lookahead action as follows —

TwoLookahead Semantic action with length-2 lookahead `ScanObj next =`

```
QueueRef.EnqueueAndReturnSymbol(NextToken);
if (next.token = (int)LexEnum.x) {
    next = QueueRef.GetAndEnqueue()
        if (next.token = (int)LexEnum.y)
            do_something_special();
}
QueueRef.AddPushbackBufferToQueue();
```

Index