The GPLEX Scanner Generator

(*Version 1.0.0 November 2008*)

John Gough QUT

November 11, 2008

New in this release

Compared to the v0.9 (August 2008) release this version has the following significant changes and new features —

* New options for unicode scanners allow the user to specify the fallback codepage to use if an input file does not have a valid *UTF* prefix.
* New facilities for unicode scanners allow the host application to set the fallback codepage at scanner runtime.
* New facilities for unicode scanners allow the scanner to scan the input file to determine the probable encoding used.
* New options for byte-mode scanners allow the user to specify the codepage mapping that is used to define the meaning of character set predicates.
* A separate documentation file "`Codepage.pdf`" collects the details of the unicode-specific features in one place.
* The change log information has been separated out into a separate documentation file "`ChangeLog.pdf`".
* Multiple input sources are now allowed, using user-supplied overrides of the default *yywrap* predicate. Use-examples are included.
* Start condition scopes have been introduced, so that pattern rules may share the same start state conditionals. These scopes may be nested.
* *C#*-style single line comments may be used anywhere in the specification file, and are treated as white space.

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 *C#*. 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 an invariant part which defines the class structure of the scanner, and the machinery of the pattern recognition engine. This part is defined in a "*frame*" file. The second part contains the tables which define the finite state machine that performs the pattern recognition, and the semantic actions that are invoked when each pattern is recognized. This part is created by *gplex* from the user-specified "`*.lex`" input file. Finally, there is user-specified code that 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 5.2. 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 like to begin by reviewing the input file format, then go to section 3.

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 field "*scanner*" of an abstract *IScanner* type (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 parser and a scanner 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
    parser = new Parser();
    parser.scanner = new Scanner(file);
    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

```
parser = new Parser();
parser.handler = new ErrorHandler();
parser.scanner = new Scanner(file);
parser.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 is inserted into this class. The invariant code supplied by the frame file specifies several buffer classes nested within the scanner class. One, *Scanner.StreamBuff*, deals with byte-stream inputs of type *System.IO.Stream*, while others deal with text files with various encodings. Finally, *Scanner.StringBuff* and *Scanner.LineBuff* deal with inputs of type *System.String*. For more detail on the available options, see section 5.3.

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 5.6.

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. Despite its name, *IScanner* is an abstract base class, rather than an interface. This abstract 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 4), but the parsing engine needs only *IScanner*.

Figure 3: Scanner Interface of *GPPG*

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

IScanner is a generic class with two type parameters. The first of these, *YYSTYPE* 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, *YYLTYPE*, 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 *gppg.LexLocation*, shown in Figure 10, or will not perform location tracking and ignore the field.

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 *IScanner*, *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 *YYLTYPE* 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 *IScanner* 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 5. For this example it is assumed that the parser specification has declared "`%namespace MyParser`" and the scanner specification has declared "`%namespace MyLexer`".

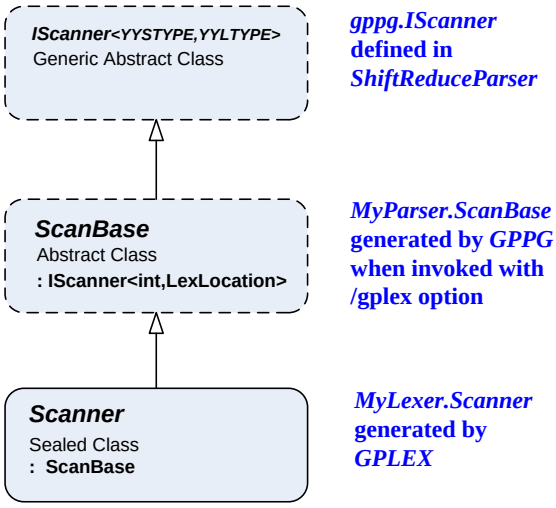
Class *ScanBase* always defines a default predicate method *yywrap* which is called whenever an end-of-file is detected in the input. The default method always returns `true`, and may be overridden by the user to support multiple input sources (see Section 5.5).

Figure 4: Features of the *Scanner* Class

```
public sealed partial class Scanner :  Parser.ScanBase {
    public  ScanBuff buffer;
    public void  SetSource(string s, int ofst);
    ...
}

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

Figure 5: Inheritance hierarchy of the Scanner class



The scanner class extends *ScanBase* and declares a public buffer field of the *ScanBuff* type. *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 underlying input stream. The *ReadPos* property, new for version 0.6.0, returns the stream position of the "*current character*". For some kinds of text streams this is not simply related to the current *Pos* value.

There are two public constructors defined in the frame file, and user code may specify others if required. 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. Another constructor takes a *System.IO.Stream* argument, and creates a stream buffer initialized with the given stream.

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 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 *IScanner* 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

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
```

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. *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.

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*)"[1].

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);
}
```

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.

[1]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.

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.

/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 5.7).

/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 codepage 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 codepage with the specified number. If there is no such codepage, or the codepage is unsuitable, an exception is thrown and processing terminates. For version 1.0 of *gplex* the specified codepage must have the single-byte property[2], or must be one of 1200 (*utf-16*), 1201 (*unicodeFFFE*) or 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 codepage with the specified name. If there is no such codepage, or the codepage is unsuitable, an exception is thrown and processing terminates. For version 1.0 of *gplex* the specified codepage must have the single-byte

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

property, or must be one of "*utf-16*" (Little-Endian Unicode), "*unicodeFFFE*" (Big-Endian Unicode) or "*utf-8*".

/codepage:default

In the event that an input file does not have a unicode prefix, the scanner will map the bytes of the input file according to the default codepage of the host machine. This codepage must have the single-byte property. This option is the default for unicode scanners, if no codepage 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 codepage. On the basis of this scan result the scanner will use either the default codepage on the host machine, or interpret the input as a *utf-8* file. See Section 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 codepoints from 0 to u+00ff will be delivered to the scanner.

/frame:*frame-file-path*

Normally *gplex* looks for a template ("frame") file named *gplexx.frame* in the current working directory, and if not found there then in the directory from which the executable was invoked. This option allows the user to override this strategy by looking for the named file first. If the nominated file is not found, then *gplex* still looks for the usual file in the executable directory. Using an alternative frame file is only likely to be 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 5.8 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.

/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.

/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 3.5.

/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 3.4.

/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 classes. (See Section 5.7 for more detail.)

/UTF8default

This option is deprecated. Use "`/codepage:utf-8`" instead. The deprecated "`/no-UTF8default`" 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 Input File

An overview of the input file specification is given in this section. The most important information is the relationship between *C#* source code locations in the input file and the place in the scanner file that the code ends up. It is important to note that the specification file for the current version of *gplex* is always an 8-bit file. The specification may specify literal unicode characters using the usual unicode escapes `\u`*xxxx* and `\U`*xxxxxxxx* where *x* denotes a hexadecimal character.

A lex file consists of three parts: the *definitions* section, the *rules* section, and the *user-code* section[3].

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

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

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

3.1 The Definitions Section

The definitions section contains "using" and "namespace" declarations, option markers, start condition declarations, lexical category definitions, character class predicate definitions and user code.

The namespaces *System, System.IO, System.Collections.Generic* are included by default. Other namespaces that are needed must be specified in the specification file. Two non-standard markers in the input file are used to generate `using` and `namespace` declarations in the scanner file. The syntax is —

"`%using`" *DottedName* "`;`"
"`%namespace`" *DottedName*

where *DottedName* is a possibly qualified *C#* identifier. As usual, for syntactic markers starting with "`%`" the keywords must be at the start of the line.

Option Markers

The definitions section may include option markers with the same meanings as the command line options described in Section 2.1. Lines of option markers have the format –

"`%option`" *OptionList*

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. The following options cannot be negated —

`help`
`codepagehelp`
`out:`*out-file-path*
`frame:`*frame-file-path*
`codepage:`*codepage-arg*

The following options can all be negated by prefixing "`no`" to the command name.

```
babel          // default is nobabel
check          // default is nocheck
classes        // default is classes for unicode
compress       // default is compress
compressmap    // default is compressmap for unicode
compressnext   // default is compressnext
files          // default is files
listing        // default is nolisting
minimize       // default is minimize
parseonly      // default is noparseonly
parser         // default is parser
stack          // default is nostack
squeeze        // default is nosqueeze
summary        // default is nosummary
unicode        // default is nounicode
verbose        // default is noverbose
version        // default is noversion
```

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

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 —

StartConditions
: *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[4].

Lexical Category Definitions

Lexical category code defines named patterns that may be used in patterns in the rules section. 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.

Character Class Membership Predicates

Sometimes user code of the scanner needs to test if a code-point corresponding to the value of some variable belongs to a particular character class. If the character class is a named lexical category, named *SetX* for example, then the following declaration "`%charClassPredicate SetX`" will cause *gplex* to generate a public method of the Scanner class —

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

This method will test the given code-point for membership of the named character class. In general, the syntax of the *charClassPredicate* declaration allows for a list of character class names.

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

User Code in the Definitions Section

Any indented code, or code enclosed in "`%{`" ... "`%}`" delimiters is copied to the output file. The "`%{`" ... "`%}`" delimited form *must* be used to include code that syntactically must start in column zero, such as "`#define`" declarations. It is considered good form to always use the delimiters for included code, so that printed listings are easier to understand for human readers.

Comments in the Definitions Section

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.

3.2 The Rules Section

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 example in Section 6.3 illustrates this rule.

As explained in Section 3.4, the attempt to find the longest match means that *gplex*-created scanners sometimes have to "back up". This occurs when a match has been found and an attempt to find an even longer match then fails.

Rule Syntax

The marker "`%%`" delimits the boundary between the definitions and rules sections. As in the definitions section, indented text and text within the special delimiters is included in the output file. All code appearing before the first rule becomes part of the prolog of the *Scan* method. Code appearing after the last rule becomes part of the epilog of the of the *Scan* method. Code *between* rules has no sensible meaning, attracts a warning, and is ignored.

The rules have the syntax —

```
Rule
    : StartConditionList_opt  pattern  Action  ;
StartConditionList
    : '<'  NameList  '>'  |  '<'  '*'  '>'  ;
Action
    : '|'
    | CodeLine
    | '{'  CodeBlock  "}"
    ;
```

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, such as *ECHO*`()`, for example[5].

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.

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.

[5] 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 "same" means using the same text span in the lex file.

Patterns

The patterns are regular expressions. Patterns must start in column zero, or immediately following a start condition list. Patterns are terminated by whitespace. The primitive elements of the expressions are single characters, the metacharacter "." (meaning any character *except* '\n'), literal strings (enclosed in double quote characters """), character classes and used occurrences of lexical categories from the definitions section.

Character classes are defined between (square) brackets. A character class defines a set of characters, and matches any character from the set. The members of the class are specified by any one of the following mechanisms: (i) individual literal characters appearing in the definition, (ii) sequences of characters that are contiguous in the `char` collating sequence, denoted by the first and last member of the sequence separated by a dash character '-', and (iii) characters corresponding to the character predicates from the *System.Char* library (see the next section for the syntax). Because of their special meaning in this context literal right bracket characters must be backslash escaped. For the same reason, literal dash characters must be backslash escaped except if the dash occurs as the first or last member of the set[6].

If the caret symbol "^" is the first character of the class the set of matching characters is inverted, that is, all characters *except* those in the class are matched. Beyond the first position the caret has no special meaning and denotes itself.

Used occurences of lexical categories are denoted by the name of the category within (curly) braces. Used occurences may occur in patterns in the rules section or within the definitions of other lexical categories. However, the defining occurence of each category must textually precede all the used occurences of that category.

The operators of the expressions are concatenation (implicit), alternation (the vertical bar), and various forms of repetition. There are also the context operators: *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 "`(\r\n|\r|\n)`".

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.

The repetition markers are: "`*`" — meaning zero or more repetitions; "`+`" — meaning one or more repetitions; "`?`" — meaning zero or one repetition; "`{n,m}`" where n and m are integers — meaning between n and m repetitions; "`{n,}`" where n is an integer — meaning n or more repetitions; "`{n}`" where n is an integer — meaning exactly n repetitions. Note carefully that the "`{n,}`" marker must not have whitespace after the comma. In the current *gplex* scanner un-escaped white space terminates the candidate regular expression.

Finally, there is one special marker that *gplex* recognizes. The character sequence "`<<EOF>>`" denotes a pattern that matches the end-of-file. The marker may be condi-

[6]The case of un-escaped dashes provokes a warning, just in case the literal interpretation is not the intended meaning.

tional 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.

Character Predicates

Within a character class, the special syntax "`[:`*PredicateMethod*`:]`" denotes all of the characters from the selected alphabet[7] 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 —

* *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 equivalent.

```
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.

Character Predicates in Byte-Mode Scanners

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. If these predicates are used in byte-mode scanners some care must be taken.

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 12. What we need is a set of byte-values v such that

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

[7] 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.

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

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

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

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

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

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

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

3.3 Start-Condition Scopes

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

StartConditionScope
: *StartConditionList* '{' *RuleList* '}' ;
StartConditionList
: '<' *NameList* '>' | '<' '*' '>' ;
RuleList
: $\mathit{RuleList}_{opt}$ *Rule*
| $\mathit{RuleList}_{opt}$ *StartConditionScope*
;

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 start condition scopes. In nested scopes the effect of the start condition lists is cumulative. Thus —

```
<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.

3.4 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"[8].

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 —

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

```
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 —

> 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.

3.5 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 9. These are normally used together with the standard *BEGIN* method. The

Figure 9: 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.

3.6 Location Information

Parsers created by *gppg* have default actions to track location information in the input text. The parsers define a class *LexLocation*, that is the default instantiation of the *YYLTYPE* generic type parameter. 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. Figure 10 is the definition of the default class. If a *gplex*

Figure 10: 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);
    }
}
```

scanner ignores the existence of the location type, the parser will still be able to access some location information using the *yyline*, *yycol* properties, but the default text span tracking will do nothing[9].

If a *gplex* scanner needs to create location objects for the parser, the logical place to do this 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)
```

where the arguments are internal variables of the scanner defined in *gplexx.frame*.

The *IMerge* interface is shown in Figure 11.

Figure 11: Location Types Must Implement *IMerge*

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

[9]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 Errors, Warnings and Gotchas

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.

"*Gotchas*" are an informal category of potential malfunctions. These are situations that users should treat with caution.

4.1 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.

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.

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

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 codepoint 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 "[...]".

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 scanners allow only single-byte codepages —
For byte-mode scanners the codepage option modifies the behavior of character set predicates. Only codepages with the single byte property make sense for this purpose.

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

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 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 codepoint 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.

4.2 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>...
```

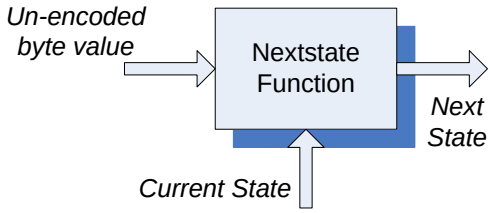
Using this specification-pattern allows the regular expression patterns that belong to the S_2 start state to include patterns that begin by matching the X that logically begins the new input phase. The lexical specification for *gplex* uses this construct no less than three times. For scanners that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

5 The Generated Scanner

5.1 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 12. In this mode, the next state of the scanner automaton is determined by the next-state function from

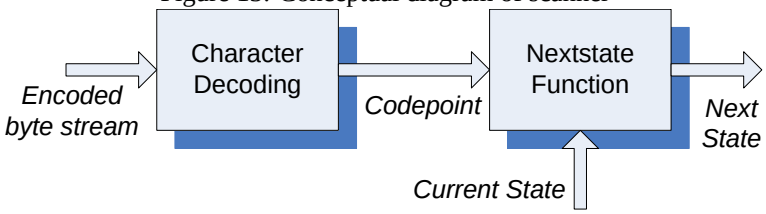
Figure 12: Conceptual diagram of byte-mode scanner



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 codepoint* and the current state. The sequence of codepoints 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 input form. The conceptual form of the scanner is shown in Figure 13 for file input. The corresponding diagram for *string* input differs only in that the input is a

Figure 13: Conceptual diagram of scanner



sequence of *System.Char*.

5.2 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. There are a number of nested classes in this class, as well as the implementations of the interfaces previously described.

The format of the 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[10].

The overall structure of the file is shown in Figure 14. There are seven 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.

[10]Later versions may hide this file away in the executable, but it is convenient to have the file explicitly available during development of *gplex*.

Figure 14: Overall Output File Structure

```
using System;
using System.IO;
using System.Collections.Generic;
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
  }
}
unicode scanners include codepage "guesser" code here
```

* 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.

5.3 Choosing the Input Buffer Class

There are a total of seven concrete implementations of the abstract *ScanBuff* class in *gplex*. There are five flavors of file input buffer, and two string input buffers.

The File Input Buffers

There are five flavors of file buffers —

* *StreamBuff*. The buffer for a byte file, which reads one byte at a time. It is used for non-unicode scanners, and by unicode scanners for files that have no prefix when the fall-back "`/codepage:raw`" is specified.
* *CodePageBuff*. The buffer for a text file which is encoded according to some specified codepage. This is used by unicode scanners for files that have no prefix, and a single-byte fallback codepage has been specified.
* *TextBuff*. The buffer for a text file encoded according to the *UTF-8* form. This is used by unicode scanners for files with a utf-8 prefix, of for files without a prefix if "`/codepage:utf-8`" has been specified.
* *BigEndTextBuff*. The buffer for a text file encoded according to the "big-endian" *UTF-16* form. This is used by unicode scanners for files with a utf-16 prefix, or for files without a prefix if "`/codepage:unicodeFFFE`" has been specified.
* *LittleEndTextBuff*. The buffer for a text file encoded according to the "little-endian" *UTF-16* form. This is used by unicode scanners for files with a utf-16 prefix, of for files without a prefix if "`/codepage:utf-16`" has been specified.

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

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

The constructor code of the *Scanner* object that is emitted by *gplex* is customized according to the */unicode* option. If the unicode option is not in force a scanner is generated with a *StreamBuff* buffer object. In this case the single-argument version of *SetSource* (third method in figure 6) will be called. This buffer reads input byte-by-byte, and the resulting scanner will match patterns of 8-bit bytes.

If the unicode option is in force, the two-argument overload of *SetSource* (last method in figure 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.

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

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 an 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.

5.4 How Buffering Works

The scanning engine that *gplex* produces is a finite state automaton (*FSA*)[11] This *FSA* deals with code-points from either the *Byte* or *Unicode* alphabets, as described in section 5.1.

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. *gplex* scanners always treat their input as byte-files, and layer any decoding on top of the *readbyte* method. This is so that arbitrary file-position seeks may be made to the first byte of any character. When a *gplex*-generated scanner opens a file at runtime it will instantiate an appropriate file buffer object for the file-encoding that it has detected.

Strings containing character data from the full unicode alphabet may require two characters to encode a single code-point. The *StringBuffer* object detects surrogate characters and reads a second character when needed.

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 *StringBuff.GetSource*.

[11] (*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.

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

Figure 15: Encoding of the example as UTF-8 file

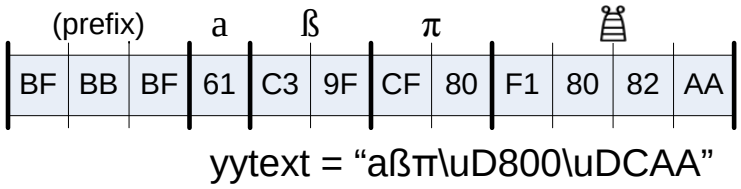
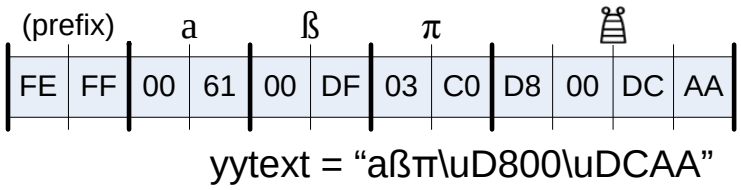


Figure 16: Encoding of the example as big-endian UTF-16 file



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 1.0 of *gplex* the decision is made according to the *fallback codepage* setting.

The default setting for the fallback codepage of *gplex*-generated scanners is to read the input byte-by-byte, and map the byte-values to unicode using the default codepage

of the host machine. Other possible fallbacks are to use a specified codepage, 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.

The treatment of codepages is detailed in the separate document "`Codepage.pdf`".

5.5 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`[12]. 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 17 is a template for the *yywrap* method. The code for creation and 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));
```

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

Figure 17: 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
}
```

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 18. In cases where include files may be

Figure 18: 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 19 is a template for the user code in such a scanner. In this case it is assumed that the pattern matching rules of the scanner detect the file-include command and parse the filename. The semantic action of the pattern matcher will then call *TryInclude*.

Figure 19: 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;
}
```

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.

5.6 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 5. 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 20 shows the code.

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

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

Figure 20: 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; }
}
```

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 *IScanner* 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.

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 21 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,

Figure 21: 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; } }
}
```

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 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.

5.7 Unicode Scanners

gplex is able to produce scanners that operate over the whole unicode alphabet. However, the *LEX* specification itself is always 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 excapes 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 three kinds of text file encodings— *UTF-8*, and 16-bit Unicode in both big-endian and little-endian variants.

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. If the prefix indicates any of the supported encodings an appropriate buffer object is created, derived from the *TextBuff* class. If no prefix is found the input stream position is reset to the start of the file and the type of buffer that is created depends on the *fallback codepage* setting.

5.8 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 Character 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 22 shows the conceptual form of a scanner with character equivalence classes. This figure should be compared with

Figure 22: Conceptual diagram of scanner with character equivalence classes

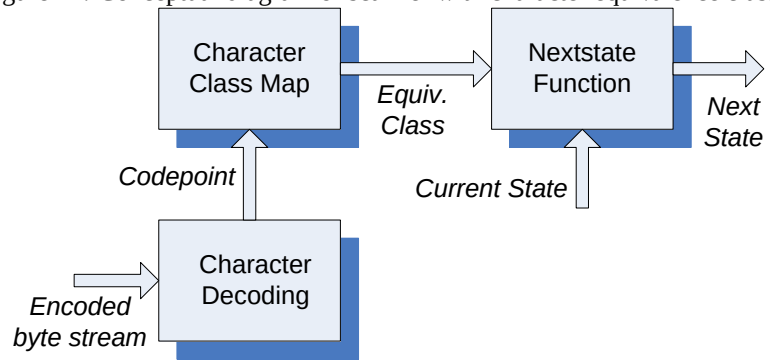


Figure 13.

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 specification there are only 38 character 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 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 classes.

With unicode scanners the use of character 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 classes that range through the whole of the unicode alphabet.

Figure 23 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 23: 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 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 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 sub-maps. 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.

Figure 24: 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

When to use Non-Default Settings

If a non-unicode scanner is particularly time critical, it may be worth considering using character 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 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 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.

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 25. The code opens the input files one by one, creates a scanner instance and calls *yylex*.

Figure 25: 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 /minimize /summary WordCount.lex
```

This also creates *WordCount.lst* with summary information. The frame file *gplexx.frame* should be in the same folder as the *gplex* executable.

This particular example, generates 26 *NFSA* states which reduces to just 12 *DFSA* states. Nine of these states are *accept* states[13] 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

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

full-stop or a decimal point. If the "`/minimize`" command line option is used the 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 20. In non stand-alone applications these definitions would be accessed by "`%using`" the parser namespace in the lex file. The application is compiled by —

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

producing *WordCount.exe*. Run it 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 25, 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`".

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 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);
%%
```

Figure 26: 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();
    }
}
```

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

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 languages for which keywords are not case sensitive the *GetIdToken* method can do a *String.ToLower* call to canonicalize the case before matching.

6.4 The Codepage Guesser

The "codepage 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 codepage of the host machine.

The example file "`GuesserTest.lex`" is a wrapped example of the codepage 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-\xrf] // 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 27 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 *StreamBuff* 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 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.

Figure 27: 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;
}
```

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.

7.2 Limitations for Version 1.0.0

Version 1.0.0 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.

7.3 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 four files: *gplex.exe*, *ShiftReduceParser.dll*, *gplexx.frame* and *Guesser.incl*. All four of these must be on the executable path, and in the same directory. 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`", "`Codepage.pdf`", "`ChangeLog.pdf`" and the file "`GPPGcopyright-.rtf`". The "`examples`" directory contains the examples described in this documentation.

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

7.4 Copyright

Gardens Point *LEX* (*gplex*) is copyright © 2006–2008, John Gough, Queensland University of Technology. See the accompanying document "`GPLEXcopyright.rtf`".

7.5 Bug Reports

Gardens Point *LEX* (*gplex*) is currently being maintained and extended by John Gough. Bug reports and feature requests for *gplex* should be sent to John at "j.gough *at-sign* qut.edu.au".

8 Appendix A: *GPLEX* Special Symbols

8.1 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 on page 12. 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.

8.2 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.

Semantic Action Symbols (continued)

Symbol	Meaning
`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.