# The GPLEX Scanner Generator

John Gough QUT

November 13, 2007

**This document applies to GPLEX version 0.6.2.***

## 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. Early releases of the tool do not attempt to implement the whole of the *POSIX* specification for *LEX*, later versions will move toward complete feature coverage. 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 only expose 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 appears within the main scanner class definition, as is explained in more detail in section 3.

If you would like to begin by reviewing the input file format, then go to section 4.

### 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 3.1.

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

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

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

Each of these views of the scanner interface are described in turn. The special case of stand-alone scanners is treated in section 3.2.

### 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

Figure 3: Scanner Interface of *GPPG*

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

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 *Shift-ReduceParser.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*.

*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 11, 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 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

Figure 4: Features of the *Scanner* Class

```
public abstract 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);
}
```

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.

The *SetSource* method attaches a new string buffer to the current scanner instance, with text “*s*” and scanning to begin at offset ”*ofst*”.

The final public features of the scanners are the constructors. There are two 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. Another constructor takes a *System.IO.Stream* argument, and creates a stream buffer initialized with the given stream.

### 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 5. The first few of these are *YACC* commonplaces, and report information about the current token. *yyleng, yypos* and *yytext* return the length of the current token, the position in the current buffer, and the text of the token. The text is created lazily, avoiding the overhead of an object creation when not required. *yytext* returns an immutable string, unlike the usual array or pointer implementations. *yyless* moves

Figure 5: 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
```

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 *yypos* of one token and (*yypos* + *yyleng*) on 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 6 is this "colorizing scanner" interface. *Visual Studio* passes the source to be scanned to

Figure 6: 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

[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.

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. All of the options may be preceded by a '–' instead of the '/' character. Available options in the current version are —

**/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 6.1).

**/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 grammar 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.

**/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 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 6.2 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.

**/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 external 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.

**/stack**

This option specifies that the scanner should provide for the stacking of start conditions. This option makes available all of the methods described in Section 4.4.

**/summary**

With this option a summary of information is written to the listing file. This gives statistics of the automaton produced, including information on the number of backtrack states. For each backtrack state a sample character is given that may lead to a backtracking episode. It is the case that if there is even a single backtrack state in the automaton the scanner will run slower, since extra information must be stored during the scan. These diagnostics are discussed further in section 4.3.

**/unicode**

By default *gplex* produces scanners that use 8-bit characters, and read input files byte-by-byte. This option allows for unicode-capable scanners to be created. Using this option implicitly uses character classes. (See Section 6.1 for more detail.)

**/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 file from the associated start state.

**/version**

The program sends its characteristic version string to the console.

## 3 The Produced 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[2].

The overall structure of the file is shown in Figure 7. There are six places where user code may be inserted. These are —

* Optional additional “using” declarations that other user code may require for its proper operation.

* A namespace declaration. Currently this is not optional.

* Arbitrary code from within the definitions section of the lex file. This code typically defines utility methods that the semantic actions will call.

[2]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 7: Overall Output File Structure

```
using System;
using System.IO;
using System.Collections.Generic;
user defined using declarations
user defined namespace declaration
{
    public 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
  }
}
```

* 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.

* 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 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.

## 3.1 Choosing the Input Buffer Class

There are a total of six concrete implementations of the abstract *ScanBuff* class in *gplex* from version 0.6.2. There are four flavors of file input buffer, and two string input buffers.

### The File Input Buffers

For all forms of file input, the user opens a file stream with the code

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

The code of the constructor for *Scanner* objects that is emitted by *gplex* is customized according to the */unicode* option. If the unicode option is not set a scanner is generated with a *StreamBuff* buffer object. This buffer reads input byte-by-byte, and the resulting scanner will match *ASCII* patterns.

If the unicode option is set, the constructor attempts to read a file preamble from the file stream. If a valid preamble is found corresponding to a *UTF-8* or one or other *UTF-16* files one of the three unicode stream buffers is created. If no preamble is found, the buffer defaults to *UTF-8*.

### String Input Buffers

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

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

This method will create a buffer object of the *StringBuff* type. This method is always used to produce colorizing scanners for *Visual Studio*.

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

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

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

## 3.2 Class Hierarchy

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

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

### Stand-alone Scanners

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

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

The simple answer is that the *gplexx.frame* file contains minimal definitions of the types required, which are activated by the */noparser* option on the command line or in the lex specification. The user need never see these definitions but, just for the record, Figure 8 shows the code.

Figure 8: 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();
}
```

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 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.

Versions of *gplex* from version 0.4.1 use 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 6), 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 9 shows the definition in *ScanBase*. *gplex* will supply a final implementation of *CurrentSc* backed by the

Figure 9: 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; } }
}
```

scanner state field *currentScOrd*, the start state ordinal.

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

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

## 4 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 *gplex* is always an 8-bit file. The specification may specify literal unicode characters using the usual unicode escape `\u`*xxxx* where *x* denotes a hexadecimal character.

A lex file consists of three parts.

## 4.1 The Definitions Section

The definitions section contains start condition declarations, lexical category definitions, and user code. 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.

The definitions section may include option markers with the same meanings as the command line options described in Section 2. Lines of option markers have the format –

"`%option`" *OptionList*

A complete list of options is given in the following section.

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.

A typical start condition declaration looks like this —

```
%x IN_COMMENT
```

These declarations are used in the rules section, where they predicate the application of various patterns. They may be exclusive "`%x`" or inclusive, "`%s`". The markers, as usual, must occur alone on a line starting in column zero.

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[3].

Lexical category code defines useful 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 inclusion, so each defined pattern must be a well formed regular expression.

### 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.

[3]Warning: the semantics of inclusive start conditions were incorrect in *gplex* for versions before 0.4.2. Furthermore, note that from version 0.5.1.126 *gplex* follows the *Flex* semantics by **not** adding rules explicitly marked *INITIAL* to inclusive start states.

**Options**

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 —

```
frame:frame-file-path
help
out:out-file-path
```

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

|  |  |
|---|---|
| `babel` | // *default is nobabel* |
| `check` | // *default is nocheck* |
| `listing` | // *default is nolisting* |
| `minimize` | // *default is minimize* |
| `compress` | // *default is compress* |
| `compressnext` | // *default is compressnext* |
| `compressmap` | // *default is compressmap for unicode* |
| `parseonly` | // *default is noparseonly* |
| `parser` | // *default is parser* |
| `stack` | // *default is nostack* |
| `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.

## 4.2 The Rules Section

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 EBNF Syntax —

|  |  |  |
|---|---|---|
| Rule | $\rightarrow$ | [ StartConditionList ] pattern Action . |
| StartConditionList | $\rightarrow$ | "<" *name* { "`,`" *name* } ">" . |
|  | \| | "<" "`*`" ">" . |
| Pattern | $\rightarrow$ | *regular expression* . |
| Action | $\rightarrow$ | *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.

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[4].

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.

## 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`’), 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[5].

If the caret symbol “ˆ” 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

[4]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.

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

the first position the caret has no special meaning. Used occurences of lexical category names appear within braces.

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 "`/`".

Left-anchored patterns only match at the start of a line, while right-anchored patterns only match 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.

#### Character Set Predicates

Within a character class, the special syntax "`[:`*PredicateMethod*`:]`" denotes all of the characters from the selected alphabet[6] for which the corresponding base class library method returns the true value. The implemented methods are —

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

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.

## 4.3 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

[6]In the non-unicode case, the sets will include only those characters from the first 256 unicode code points for which the predicate functions return true. In the case of the /unicode option, the full sets are returned.

that is an accept state that contains at least one *out*-transition that leads to a non-accept state. The point is that if the automaton leaves a perfectly good accept state in the hope of finding an even longer match it may fail. When this happens, the automaton must return to the last accept state that it encountered, pushing back the input that was fruitlessly read.

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

Consider the grammar —

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

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

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

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

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

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

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

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 returns the automaton to the accept state.

## 4.4 Stacking Start Conditions

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

One approach to handling the start conditions in such cases is to use a *stack* of start conditions, and to push and pop these in semantic actions. *gplex* supports the stacking of start conditions when the "`stack`" command is given, either on the command line, or as an %option in the definitions section. This option provides the methods shown in

Figure 10: 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();
```

Figure 10. These are normally used together with the standard *BEGIN* method. The first method clears the stack. This is useful for initialization, and also for error recovery in the start condition automaton.

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

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

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

## 4.5 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 11 is the definition of the default class. If a *gplex* 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[8].

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*.

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

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

Figure 11: 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);
    }
}
```

The *IMerge* interface is shown in Figure 12.

Figure 12: Location Types Must Implement *IMerge*

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

## 5 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.

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

Figure 13: 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 —

```
> 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[9] and there are two backup states. Both backup states occur on a "." input character. In essence when the lookahead character

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

is dot, *gplex* requires an extra character of lookahead to before it knows if this is a 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 8. In non stand-alone applications these definitions would be accessed by "`%using`" the parser namespace in the lex file. The application is compiled by —

```
> 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>
```

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`".

## 5.2 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 13, 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`".

## 5.3 Keyword Matching

The final 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
alpha [a-zA-Z]
%%
foo         |
bar         Console.WriteLine("keyword " + yytext);
{alpha}{3}  Console.WriteLine("TLA " + yytext);
{alpha}+    Console.WriteLine("ident " + yytext);
%%
```

The point is that the input text “foo” actually matches three of the four patterns. It matches the “*TLA*” pattern and the general ident pattern as well as the exact match. Here is the string-scanning version of the user code section.

```
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();
    }
}
```

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 exercise 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 read —

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

The *GetIdToken* method should check if the string of the text matches a keyword, and returns 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 Notes

## 6.1 Unicode Scanners

From version 0.6.0 *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 format `\u`*xxxx* where *x* is a hexadecimal digit. These may appear in literal strings, as primitive operands in regular expressions, or in bracket-delimited character classes.

#### 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 *UTF-8* decoder is created and the stream position reset to the start of the file.

## 6.2 Choosing the 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.

#### 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 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.

### Scanners With Character Classes

If a scanner uses character classes, then 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, $|S|$.

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 *Component Pascal* 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 65536. For the *Component Pascal* grammar this would lead to uncompressed tables of over four mega-bytes. In grammars which actually contain unicode character literals the simple compression of the next-state tables is often 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 64k bytes 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.

Compression of the 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.

### Statistics

If the *summary* option is used, statistics related to the table compression are emitted to the listing file.

Figure 14 are the statistics for the 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 since the *LEX*

Figure 14: 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 | 65536 | |
| unicode, nocompressmap, compressnext | 249 | 38 | 65536 | |
| unicode, compressmap, compressnext | 249 | 38 | 127 | 1 |
| unicode, compressmap, nocompressnext | 1064 | 38 | 127 | 1 |

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.

### 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. In such cases, if speed is a consideration, then it is possible to accept the memory footprint penalty of 64kB, use an uncompressed map, and reclaim all of this overhead.

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.

## 6.3 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* nor with *Flex*. However, for those features that *are* implemented the behaviour follows *Flex* rather than *POSIX* when there is a difference.

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

#### Error Reporting

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

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

## 6.4 Limitations for Version 0.6.0

Version 0.6.0 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.

## 6.5 Installing *GPLEX*

*gplex* is distributed as a zip archive. The archive should be extracted into any convenient folder. The distribution contains four subdirectories. The "`bin`" directory contains three files: *gplex.exe*, *ShiftReduceParser.dll*, and *gplexx.frame*. All three 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 "`source`" directory contains all of the source code for *gplex*. The "`doc`" directory contains the files "`gplex.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.

## 6.6 Copyright

Gardens Point *LEX* (*gplex*) is copyright © 2006–2007, John Gough, Queensland University of Technology. See the accompanying document "`GPLEXcopyright.rtf`".

## 6.7 Change Log

This section tracks the updates and bug fixes from the intial release candidate version RC0 of October 2006. RC0 was the first release candidate bootstrapped using a self-generated scanner and *gppg*-generated parser. Versions prior to RC0 used a handwritten scanner.

#### Changes in version 0.6.2 (gplex, November 2007)

* New buffer implementation for lists of strings contributed by Nigel Horspool.
* The scanner class is marked `partial`, so that much of the user code of the scanner can be separated out into a separate file.
* A number of small bugs in the unicode support have been fixed.

#### Changes in version 0.6.1 (gplex, August 2007)

* Dead code warnings in semantic action dispatch of the automaton have been suppressed.
* Sharing of redundant rows is implemented for uncompressed next-state tables
* Treatment of un-escaped dash characters as the first or last character in a character class definition follows the *LEX* semantics, rather than being an error.
* Incorrect behavior with empty strings in semantic actions has been fixed.
* Bug in V0.6.0 text buffer code lost the last character of *yytext* if an unexpected end of file was encountered. Fixed.

### Changes in version 0.6.0 (gplex, July 2007)

* Table compression using character equivalence classes is invoked by the /*classes* option.

* New option /*unicode* allows unicode scanners to be generated. *UTF-8*, and both byte-orders of *UTF-16* are supported.

* Option /*minimize* is now on by default.

* The frame file has numerous small changes - the encoding of *EOF* has changed, the abstract buffer class has a property *ReadPos* for the file position of the current character (not the same as *Pos*–1 in the unicode case), and new buffer classes have been added for encoded text files.

* A bug in the parsing of regular expressions, if a character escape appears immediately after a range operator '–', is fixed.

### Changes in version 0.5.1 (gplex, March 2007)

* Rules explicitly marked with the *INITIAL* start condition are **not** added to inclusive start states. This aligns *gplex* with the *Flex* semantics.

* The /*babel* option produces incremental scanners compatible with the *Managed Babel* framework.

* The "frame" file, *gplexx.frame*, has been changed to allow for the /*babel* option.

* To use a *gplex* scanner and a *gppg* parser with *Managed Babel* both tools must be invoked with the /*babel* option.

* When working with string buffer input, *gplex* returned an *EOL* character if the scanner attempted to read past the end of the string. This behavior is now restricted to the /*babel* option, where it is necessary for compatibility with the managed package framework. In previous versions attempts to fetch *yytext* for this virtual token threw an exception. The code is now guarded.

### Changes in version 0.5.0 (gplex-RTM, February 2007)

* Parser is now generated by version 1.0.2.* of *gppg*, using a modified version of *IScanner*. *Existing* gplex *scanners will require recompilation to work with parsers produced by the new version of* gppg.

* Version strings are now generated from an assembly attribute, and accessed via reflection.

* New facilities allow output to be sent to the standard output stream. This may involve redirecting verbose progress output to standard error.

### Changes in version 0.4.2 (gplex-RC2, January 2007)

* Semantics of inclusive start states were incorrect. Now fixed.

* There was a bug in the analysis of right context patterns involving concatentation, leading to some cases being incorrectly analysed as having fixed length. Fixed.

**Changes in version 0.4.1 (gplex-RC1, November 2006)**

* Program failed on empty user code containing not even a comment. Fixed.

* Reflection is used in *gplexx.frame* to determine if the *maxParseToken* value has been defined in the parser specification.

* Must allocate a default error handler if *OpenSource* fails.

* Table compression algorithm failed for some corner cases where the longest run of equal next-state entries wraps around character 255.

* Within the constructed scanners token start and end position is now represented by private variables *tokPos*, *tokEPos*, *tokLin*, *tokELin*, *tokCol*, *tokECol* to allow single tokens to correspond to multi-line *LexLocation* values. The previous *tokLen* has disappeared, with *yyleng* now computed from *tokEPos* and *tokPos*. *This change may break user-defined YYLTYPE computations*.