The MPLEX Scanner Generator

John Gough QUT

August 21, 2006

This document applies to MPLEX version 0.21

1 Overview

This paper is a preliminary version of the documentation for the *mplex* scanner generator.

Managed Package *LEX* (*mplex*) 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.

A particular objective of the design is to provide good support for the production of colorizing scanners, that is, scanners which may be called on a line by line basis in arbitrary line-order. In order to do this, the produced scanners implement two interfaces. One is the traditional "interface to *YACC*" while the other is aimed at *Visual Studio* integration. Internally the scanner holds a buffer object. The buffer type is abstract: the concrete object will hold either a stream buffer of a string buffer, as required.

The scanners produce by *mplex* 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 instance. Most are accessed through properties which only expose a getter.

The implementation of *mplex* makes heavy use of the facilities of the 2.0 version of *C#*. There is no prospect of making it run on earlier version of the framework.

1.1 The Interfaces

The produced scanners implement the interfaces shown in the following figures. Figure 1 is the "colorizing scanner" interface. In this case the text to be scanner is passed as a string in the *SetSource* method. An offset into the string defines the logical starting point of the line. The *GetNext* method returns an integer representing the recognized token. The set of valid values for this interface 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 to 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

Figure 1: 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);
}
```

state will be just the start-condition of the underlying finite state automaton (*FSA*), however there are other possibilities, discussed below.

The second "interface" is that required by variants of *YACC*-generated parsers. This interface is aimed at the *Managed Package Parser Generator* (*MPPG*) tool. Figure 2 shows the signatures. This abstract base class defines the interface required by the

Figure 2: Scanner Interface of *MPPG*

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

runtime component of *mppg*, the library *MppgRuntime.dll*. The semantic actions of the generated parser may use the richer *API* that *mplex* supports, but the parsing engine needs only this subset.

The class 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 *LexLocation*, shown in Figure 10 or will not perform location tracking and ignore the field.

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

The first method, *yylex*, returns the ordinal number corresponding to the next token. This is an abstract method, the actual scanner *must* supply a method to override this.

The second method, the low-level error reporting routine *yyerror*, is called by the parsing engine during error recovery. The default method in the base class is empty. The scanner has the choice of overriding *yyerror* or not. If the scanner overrides *yyerror* it may use that method to emit error messages. Alternatively the semantic actions of

the parser may explicitly generate error messages, possibly using the location tracking facilities of the parser, and leave *yyerror* empty.

In the case that there is no parser, a dummy declaration may supply any value type as a type argument. The low-level error reporting routine *yyerror* is called by the parsing engine during error recovery. There is an empty implementation in the base class. Parsers that implement error reporting in their semantic actions can simply not override the empty implementation.

The scanners support a number of other utility routines, discussed later. Each scanner also defines an internal data structure for an error handler that implements the *IErrorHandler* interface shown in Figure 3. Typical implementations of *IErrorHandler*

Figure 3: Interface to the Error Handler

```
public interface IErrorHandler
{
    int ErrNum { get; }
    int WrnNum { get; }
    void AddError(string msg,
                  int lin, int col, int len, int severity);
}
```

provide for buffering of errors, sorting according to line number, and the merging of source text and error messages in a listing file. The scanner and parser of *mplex* itself use such an example.

The *AScanner* interface provides all of the calls that the invariant part of the *Parser* will want to make on the scanner. However there is another set that is useful within the *semantic actions* of the scanner and parser. These include the methods in Figure 4. The

Figure 4: Additional methods for scanner actions

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

public int BuffPos { get; set; }
public string GetString(int start, int end);
public IErrorHandler Handler { get; }

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

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 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 part of the input of the current token is rescanned.

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. *GetString* returns a string with the first and last buffer positions specified. 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*. The definition of this variable, and its connection with the *IColorScan* interface is described further in Section 3.1. 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].

2 Running the Program

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

`mplex [`*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 —

/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.

/frame:*frame-file-path*

Normally *mplex* looks for a template ("frame") file named *mplex.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 *mplex* still looks for the usual file in the executable directory. Using an alternative frame file is only likely to be of interest to *mplex*-developers.

/help

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

[1] Note 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* specifications, where such names live in the macro namespace, and may have spellings that include hyphens.

/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.

/minimize

By default *mplex* does not perform state minimization on the *DFSA* that it computes. This option forces state minimization to take place. As a general rule more complex specifications benefit most from minimization. For example, the lexical specification of *mplex* itself requires about 350 *DFSA* states, which reduce down to 110 with minimization. This will probably become the default for future releases.

/nocompress

mplex compresses its scanner tables by default. It is possible to force emission of uncompressed tables for a slight speedup. However, for pathological patterns, such as "`{alpha}*x{alpha}{8}`" the compressed tables are 90% smaller.

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

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

/parseonly

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

/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.

/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, 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 relationship between the base class *ScanBase* and the *AScanner* and *IColorScan* types is discussed further in Section 3.1.

The format of the file is defined by a template file named *mplex.frame*. User defined, or 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 5. There are six places where

Figure 5: 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
      public abstract class ScanBuffer
        ...            // all the other invariant code
      // The scanning engine starts here
      int Scan() {  // Scan is the core of yylex and GetNext
            optional user supplied prolog
            ...        // invariant code of scanning automaton
            user specified semantic actions
            optional user supplied epilog
      }
      user-supplied body code from "usercode" section
  }
}
```

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.

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

* 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.

* 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. *mplex* statically determines which version to "`#define`" out.

3.1 Where Does ScanBase Come From?

The runtime component of parsers generated by Managed Package Parser Generator (*mppg*) expect scanners to conform to the generic class *AScanner*, as shown in Figure 2. The template file for *mplex* scanner, on the other hand, invariably extends a class named *ScanBase*.

The *ScanBase* class forms an adapter that instantiates the generic *AScanner* class. *mppg* computes the types of the two generic arguments, and emits the definition of the *ScanBase* class to its output parser file. Figure 6 is a typical definition. It is emphasised

Figure 6: Example ScanBase class

```
public abstract class
        ScanBase :  AScanner<int,LexLocation>, IColorScan
{
   protected int currentScOrd;
   public virtual int
      GetEolState() { return currentScOrd; }
   public virtual void
      SetEolState(int value) { currentScOrd = value; }
   public abstract void
      SetSource(string s, int o);
   public abstract int
      GetNext(ref int state, out int start, out int end);
}
```

that this definition is automatically created by *mppg*. The class wraps together the

obligations that the scanner has to the two interfaces and provides some default helper methods.

The *currentScOrd* variable is the place where the start condition value is held. As well, the wrapper introduces abstract methods for the *IColorScan* interface. The two virtual methods provide helpers for the default behavior of the *GetNext* method used by colorizing scanners, such as the *LineScanner* of the *Visual Studio Babel* package. Normally, the only state that needs to be persisted between lines is the start condition value. However there are circumstances, such as when start condition values are stacked or if a brace depth counter is needed, when some sort of aggregate value needs to be passed. Such effects are easily achieved by overriding the virtual *GetEolState, SetEolState* methods in the user code of the lex specification.

SetSource is the method that passes strings to the scanner as input. The integer parameter is a starting offset into the string so that the scanner need not start at the first character.

3.2 Stand-alone Scanners

mplex may also be used to create stand-alone scanners that operate without an attached parser. There are some examples of such use in a later section.

The question is: if there is no parser, then where does the *ScanBase* abstract base class come from in such cases? Furthermore, the definition of the *Tokens* enumeration and the *IErrorHandler* interface are normally also imported from the parser.

The answer is that a small *C#* dummy definition is supplied. Figure 7 is a typical example.

The *ParserDummy* file shown in the example defines three classes. A dummy *Tokens* class defines two constants, *EOF* and *maxParseToken*. The scanner template expects these two, even if there are no others. Since there is no parser, we can claim that the parser will accept any token at all and set *maxParseToken* to “maxint”.

The *ScanBase* class is the heart of the matter. Stand-alone scanners probably do not need the *IColorScan* functionality, but since the template file defines some overriding methods, the base class must define the corresponding abstract methods. The body of this class is similar to that of the same class as produced by *mppg*, except that all reference to the generic parameters *YYSTYPE, YYLTYPE* can be removed. An important addition is the *yylex()* method. Since we do not derive from the generic *AScanner* class in this case, we must declare *yylex* so the definition in the frame file has something to override.

Finally, the *IErrorHandler* class is exactly the standard definition as shown in Figure 3. The only important functionality is the ability for the user code of the scanner to ask if there were any errors.

4 The Input File

The detailed specification of the input file syntax is not given here, except in overview. The 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. A lex file consists of three parts.

Figure 7: Dummy Parser Definitions File

```
using System;
namespace LexScanner; // Or whatever namespace is needed
{
    public class Tokens {
        public constant int EOF = 0;
        public constant int maxParseToken = int.MaxValue;
    }

    public abstract class ScanBase { // Different from figure 6
      protected int currentScOrd;
      public virtual int
        GetEolState() { return currentScOrd; }
      public virtual void
        SetEolState(int value) { currentScOrd = value; }
      public abstract void
        SetSource(string s, int o);
      public abstract int
        GetNext(ref int state, out int start, out int end);
      public abstract int yylex();
    }

    public interface IErrorHandler { // Same as figure 3
      int ErrNum { get; }
      int WrnNum { get; }
      void AddError(string msg,
                    int lin, int col, int len, int severity);
    }
}
```

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 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.

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 *mplex* 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.

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	→	[StartConditionList] pattern Action .
StartConditionList	→	"`<`" *name* { "`,`" *name* } "`>`" .
	|	"`<`" "`*`" "`>`" .
Pattern	→	*regular expression* .
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. If any of the names is an inclusive start condition, then the same rule may apply to the distinguished, default start condition *INITIAL*.

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[3].

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 definition 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. Character sequences adjacent in the encoding sequence are separated by the dash character. Thus literal dash and right bracket characters must be backslash escaped. 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. 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.

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.

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[4] — meaning n or more repetitions; "`{n}`" where n is an integer — meaning exactly n repetitions.

[3] And this is not just a matter of saving on typing. When *mplex* 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.

[4] Beware of including space after the comma in this marker. White space will terminate the regular expression making it ill-formed.

4.3 Backtracking Information

When the "`/summary`" option is sent to *mplex* 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 *mplex* 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.

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"[5].

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. *mplex* will find a shortest possible string that leads from the associated start condition to the backup state. Among all such shortest strings it tries to find a string that does not need character escapes.

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

4.4 Remarks on Colorizing Scanners

Colorizing scanners pose a small number of unique problems for the designer of a lex grammar. It is necessary, at the end of each line, to return enough information to allow the scanner to restart on the next line in the correct context. This means, for example, that there must be a state for "in the middle of a block comment" which the parser would usually not see, and the scanner would never have as a *start* state. The *StartState* facility of *LEX* provides a sufficient mechanism for this. Our recommendation is to create an augmented token numbering with states only of interest to the colorizer beyond the limit of interest for the parser. Thus inside *mplex* the code for *yylex* and *GetNext* behave differently —

```
do { tok = Scan(); } while (tok > Tokens.maxParseToken);
```

and

```
tok = Scan();
```

Tokens such as comment fragments that are only of interest to the colorizer are skipped over in *yylex* while still being visible to the colorizer. The user must define *maxParseToken*, usually with a `%token maxParseToken` declaration in the "`*.y`" file.

4.5 The IColorScan.GetNext Method

IColorScan.GetNext is the heart the colorizing scanner. The body of this method, in the frame file, is shown in Figure 8. The method begins by setting up the scanner state. It

Figure 8: Body of *GetNext* method

```
public override int
       GetNext(ref int state, out int start, out int end);
{
    int next; // Next state variable
    SetEolState(state); // Load up currentScOrd
    currentStart = startState[currentScOrd];
    next = Scan(); // Call main scanner method
    start = tokPos;
    end = tokPos + tokLen - 1;
    state = GetEolState();
    return next; // Return token ordinal
}
```

does this by calling the virtual *SetEolState* method inherited from the *ScanBase* class. It then sets up the initial state of the *DFSA*, and invokes the *Scan* method.

When the *Scan* method returns, the token value is passed back to the caller. However, before the return the code must send the persistent state back to the caller through the by-ref *state* argument, and deliver the start and end indices to the caller.

In the usual case, the inherited methods for handling the *EolState* simply get and set the start condition variable *currentScOrd*. A more challenging problem arises in the case that the scanner needs to store information normally only known to the parser. *LEX* itself has such a feature, since it must match braces to know where block code

ends in an action. It is possible to solve this problem by encoding the "state" so that it stores both the *DFSA* state and a brace depth count. This may be done by overriding the virtual *GetEolState, SetEolState* methods that are called in *GetNext*.

4.6 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 case is to use a stack of start conditions, and to push and pop these in semantic actions. *mplex* supports the stacking of start conditions and supplies the methods shown in Figure 9. These are normally used together with standard *BEGIN* method. The first method clears the stack. This is

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();
```

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) ...
```

Remember however, that if the start condition stack facility is used in a colorizing scanner the state values returned by *GetNext* must encode the whole of the stack state.

4.7 Location Information

Parsers created by *mppg* 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. *mppg* 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 *mplex*

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[6].

If an *mplex* 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(yyline, yycol, yyline, yycol + yyleng)
```

The *IMerge* interface is shown in Figure 11.

Figure 11: 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 *mplex* 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]The parser will not crash by trying to call *Merge* on a null reference, because the default code is guarded by a null test.

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

Figure 12: 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 > 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 —

```
> mplex /summary WordCount.lex
```

This also creates *WordCount.lst* with summary information. The frame file "`mplex-.frame`" should be in the same folder as the *mplex* executable.

This particular example, generates 26 *NFSA* states which reduces to just 12 *DFSA* states. Nine of these states are *accept* states[7] and there are two backup states. Both backup states occur on a "." input character. In essence when the lookahead character is dot, *mplex* 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 *ParserDummy* file described in Figure 7. 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 ParserDummy.cs
```

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

```
D:\mplex\test> WordCount WordCount.cs ParserDummy.cs
File: WordCount.cs
Lines: 609, Words: 1435, Ints: 525, Floats: 2
File: ParserDummy.cs
Lines: 32, Words: 107, Ints: 1, Floats: 0
Total Lines: 641, Words: 1542, Ints: 526, Floats: 2
D:\mplex\test>
```

Where do the two "floats" come from? Good question! The text of *WordCount.cs* quotes some version number strings in a comment. The scanner thinks that these look like floats.

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 12, 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 *mplex* chooses the lowest numbered pattern when there is more than one match. Here is the file "`foobar.lex`".

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

```
%namespace LexScanner
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 Implementation Notes

Version 0.21 parses its input files using a parser constructed by Managed Package Parser Generator (*mppg*). 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. *mplex* will detect a number of syntax errors in the code parts of the specification prior to compilation of the resulting scanner output file.

The scanner for the *LEX* input files is still hand-written for this version. When the scanner is being used in colorizing mode it must be able to separate the *LEX* tokens from the *C#* tokens without the assistance of the parser state. The current design requires about 12 start conditions to achieve this.

Compatibility

The current version of *gplex* is compatible with neither *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 both the "`%x`" and "`%s`" markers for start states, and the semantics of pattern expansion follow the *Flex* model. In particular, operators applied to named lexical categories behave as though the named pattern were surrounded by parentheses.

Error Reporting

The default error-reporting behavior of *mppg*-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 the *IErrorHandler* 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.

6.2 Limitations for Version 0.21

Version 0.21 supports anchored strings but does not support the right context operator. As well, the standard lex character set definitions such as "`[:isalpha:]`" are not supported. 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.

The current release does only simple table compression. Furthermore, the facilities for changing input stream during scanning have not yet been added.

The current version supports only 8-bit characters. The extension to unicode is a more difficult matter.