The GPLEX Input Language

(*Version 1.1.0 March 2009*)

John Gough QUT

March 3, 2009

Documentation Map

This paper is the documentation of the input format for the *gplex* scanner generator. The complete documentation for *gplex* consists of the following files —

* Gplex.pdf – the main documentation file
* Gplex-Input.pdf – description of the input language of *gplex*, (this file)
* Gplex-Unicode.pdf – documentation of the unicode-specific features
* Gplex-Changelog.pdf – change log for *gplex*

1 The Input File

1.1 Lexical Considerations

Every *gplex*-generated scanner operates either in byte-mode or in unicode-mode. *gplex* scans its own input using a byte-mode scanner. It follows that the "`*.lex`" files that *gplex* reads are treated as streams of 8-bit bytes.

1.1.1 Character Denotations

The *gplex* scanner operates in byte-mode. Nevertheless, the input files can define unicode scanners, and can denote character literals throughout the entire unicode range. Denotations of characters in *gplex* may be uninterpreted occurrences of plain characters, or may be one of the conventional character escapes, such as '`\n`' or '`\0`'. As well, characters may be denoted by octal, hexadecimal or unicode escapes.

In different contexts within a *LEX* specification different sets of characters have special meaning. For example, within regular expressions parentheses "`(, )`" are used to denote grouping of sub-expressions. In all such cases the ordinary character is denoted by an *escaped* occurrence of the character, by being prefixed by a backslash '`\`' character. In the regular expression section 2 of this document the characters that need to be escaped in each context are listed.

1.1.2 Names and Numbers

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

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

1.2 Overall Syntax

A lex file consists of three parts: the *definitions* section, the *rules* section, and the *user-code* section[1].

```
LexInput
      : DefinitionSequence  "%%"  RulesSection  UserCodeSection_opt
      ;
UserCodeSection
      : "%%"  UserCode_opt
      ;
```

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

1.3 The Definitions Section

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

```
DefinitionSequence
      : DefinitionSequence_opt  Definition
      ;
Definition
      : NamespaceDeclaration
      | UsingDeclaration
      | VisibilityDeclaration
      | NamingDeclaration
      | StartConditionsDeclaration
      | LexicalCategoryDefintion
      | CharacterClassPredicatesDeclaration
      | UserCharacterPredicateDeclaration
      | UserCode
      | OptionsDeclaration
      ;
```

1.3.1 Using and Namespace Declarations

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

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

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

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

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

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

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

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

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

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

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

```
%using myParserNamespace;
```

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

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

1.3.2 Visibility and Naming Declarations

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

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

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

Naming declarations have the following syntax —

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

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

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

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

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

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

1.3.3 Start Condition Declarations

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

Here is the full grammar for start condition declarations —

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

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

When the scanner is set to an *exclusive* start condition *only* patterns predicated on that exclusive condition are "active". Conversely, when the scanner is set to an *inclusive* start condition patterns predicated on that inclusive condition are active, and so are all of the patterns that are unconditional[2].

1.3.4 Lexical Category Definitions

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

```
LexicalCategoryDefinition
      : ident   RegularExpression
      ;
```

The syntax of regular expressions is treated in detail in Section 2 A typical example might be —

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

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

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

1.3.5 Character Class Membership Predicates

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

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

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

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

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

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

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

```
%charClassPredicate ExpandsOnNFC
```

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

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

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

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

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

1.3.6 User Character Predicate Declaration

Character classes in *gplex* may be generated from any of the built-in character predicate methods of the *.NET* runtime, or any of the three other built-in functions that *gplex* itself defines (see Section 2.2.5).

If a user needs to make use of additional character class predicates, then the user may supply a *PE*-file containing a class which implements the *QUT.Gplex.ICharTestFactory* interface shown in Figure 1. The *GetDelegate* method of the interface should return delegates which implement the predicate functions. These might either be user-written code, or existing library methods with matching signatures.

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

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

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

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

Figure 1: Interface for user character predicates

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

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

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

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

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

then the following will happen —

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

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

1.3.7 User Code in the Definitions Section

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

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

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

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

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

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

1.3.8 Comments in the Definitions Section

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

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

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

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

1.3.9 Option Declarations

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

```
OptionsDeclaration
      : "%option"   OptionsList
      ;
OptionsList
      : Option
      | OptionsList   ','opt   Option
      ;
```

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

The options correspond to the command line options. Options within the definitions section take precedence over the command line options. A full list of options is in Section 4.3.

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

1.4 The Rules Section

1.4.1 Overview of Pattern Matching

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

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

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

The main engine of pattern matching is a method named *Scan*. This method is an instance method of the scanner class. It uses the tables of the generated automaton to update its state, invoke sematic actions whenever a pattern is matched, and return integer values to its caller denoting the pattern that has been recognized.

1.4.2 Overall Syntax of Rules Section

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

```
RulesSection
    : PrologCode_opt   RuleList   EpilogCode_opt
    ;
RuleList
    : RuleList_opt   Rule
    | RuleList_opt   RuleGroup
    ;
PrologCode
    : UserCode
    ;
EpilogCode
    : UserCode
    ;
```

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

The *CodeBlock* of the optional prolog *UserCode* is placed at the start of the *Scan* method. It can contain arbitrary code that is legal to place inside a method body[3]. This is the place where local variables that are needed for the semantic actions should be declared.

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

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

1.4.3 Rule Syntax

The rules have the syntax —

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

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

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

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

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

The syntax of regular expressions is treated in detail in Section 2

1.4.4 Rule Group Scopes

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

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

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

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

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

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

has exactly the same effect as —

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

or indeed as the plain, old-fashioned sequence —

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

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

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

1.4.5 Comments in the Rules Section

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

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

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

1.5 The User Code Section

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

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

2 Regular Expressions

2.1 Concatenation, Alternation and Repetition

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

2.1.1 Definitions

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

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

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

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

2.1.2 Operator Precedence

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

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

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

and so on.

2.1.3 Repetition Markers

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Regular Expression Atoms

2.2.1 Character Denotations

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

Characters that have some format affect on the input must be escaped, so the usual control characters in *C#* are denoted as `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\0`, exactly as in *C#*[5].

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

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

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

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

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

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

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

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

2.2.2 Lexical Categories – Named Expressions

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

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

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

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

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

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

```
keyword     foo|bar
```

and then use this lexical category in another expression —

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

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

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

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

2.2.3 Literal Strings

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

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

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

2.2.4 Character Classes

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

```
digits      [0246813579]
```

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

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

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

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

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

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

Within a character class definition the following characters have special meaning: ‘`]`’, marking the end of the set; ‘`-`’, denotes the range operator, except as the first or last character of the set; ‘`^`’, denotes set inverse, but only as the first character in the set. All these characters must be escaped in locations where they have special meaning.

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

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

2.2.5 Character Class Predicates

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

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

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

There are three additional predicates built into *gplex* —

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

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

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

New in version 1.0.2 of *gplex* is the ability for users to define their own character predicate functions. This feature is specified in Section 1.3.6.

2.2.6 The Dot Metacharacter

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

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

```
any     [ˆ\r\n\u0085\u2028\2029]
```

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

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

2.2.7 Context Markers

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

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

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

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

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

2.2.8 End-Of-File Marker

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

3 Special Symbols in Semantic Actions

3.1 Properties of the Matching Text

3.1.1 The yytext Property

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

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

3.1.2 The yyleng Property

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

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

3.1.3 The yypos Property

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

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

3.1.4 The yyline Property

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

3.1.5 The yycol Property

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

3.2 Looking at the Input Buffer

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

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

3.2.1 Current and Lookahead Character

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

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

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

3.2.2 The yyless Method

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

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

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

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

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

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

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

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

3.2.3 The yymore Method

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

3.3 Changing the Start Condition

3.3.1 The *BEGIN* Method

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

3.3.2 The *YY_START* Property

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

Reading the value of *YY_START* is useful for those complicated scenarios in which a pattern applies to multiple start conditions, but the semantic action needs to vary depending on the actual start condition. Code of the following form allows this behavior —

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

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

3.4 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 2. These are normally used together with the standard *BEGIN* method. The

Figure 2: Methods for Manipulating the Start Condition Stack

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

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

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

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

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

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

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

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

3.5 Miscellaneous Methods

3.5.1 The *ECHO* Method

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

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

4 Appendix A: Tables

4.1 Keyword Commands

Keyword	Meaning
`%x`	This marker declares that the following list of comma-separated names denote exclusive start conditions.
`%s`	This marker declares that the following list of comma-separated names denote inclusive start conditions.
`%using`	The dotted name following the keyword will be added to the namespace imports of the scanner module.
`%namespace`	This marker defines the namespace in which the scanner class will be defined. The namespace argument is a dotted name. This marker must occur exactly once in the definition section of every input specification.
`%option`	This marker is followed by a list of option-names, as detailed in section 4.3. The list elements may be comma or white-space separated.
`%charClassPredicate`	This marker is followed by a comma-separated list of character class names. The class names must have been defined earlier in the text. A membership predicate function will be generated for each character class on the list. The names of the predicate functions are generated algorithmically by prefixing "*Is_*" to the name of each character class.
`%userCharPredicate`	This marker is followed by a simple identifier and the designator of a user-supplied *CharTest* delegate. When the identifier is used in a character class definition *gplex* will call the user delegate to evaluate the character class at scanner creation time. See section 1.3.6 for usage rules.
`%visibility`	This marker controls the visibility of the *Scanner* class. The permitted arguments are `public` and `internal`.
`%scannertype`	The identifier argument **defines** the scanner class name, overriding the default *Scanner* name.
`%scanbasetype`	The identifier argument **declares** the name of the scanner base class defined by the parser. This overrides the *ScanBase* default.
`%tokentype`	The identifier argument **declares** the name of the token enumeration type defined by the parser. This overrides the *Tokens* default.

4.2 Semantic Action Symbols

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

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

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

4.3 *GPLEX* Options

4.3.1 Informative Options

The following options are informative, and cannot be negated —

`help`	Send the usage message to the console
`codePageHelp`	Send help for the code page options to the console
`out:`*out-file-path*	Generate a scanner output file with the prescribed name
`frame:`*frame-file-path*	Use the specified frame file instead of seeking "gplexx.frame" on the built-in search path
`codePage:`*code-page-arg*	For unicode scanners: deal with input files that have no *UTF* prefix in the nominated way. For byte-mode scanners: interpret the meaning of character class predicates according to the encoding of the nominated code page.

4.3.2 Boolean Options

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

Option	**Meaning**	**Default**
`babel`	Include interfaces for Managed Babel framework	*default is noBabel*
`check`	Compute the automaton, but do not create an output file	*default is noCheck*
`classes`	Use character equivalence classes in the automaton	*unicode default is classes*
`compress`	Compress all tables of the scanner automaton	*default is compress*
`compressMap`	Compress the equivalence class map	*unicode default is compressMap*
`compressNext`	Compress the next-state table of the scanner	*default is compressNext*
`embedBuffers`	Embed buffer code in the scanner namespace	*default is embedBuffers*
`files`	Provide file-handling code in scanners	*default is files*
`listing`	Generate a listing, even when there are no errors	*default is noListing*
`minimize`	Minimize the number of states of the automaton	*default is minimize*
`parseOnly`	Check the input, but do not construct an automaton	*default is noParseOnly*
`parser`	Expect type definitions from a host parser	*default is parser*

Table continues on next page...

Boolean Options Continued ...

Option	**Meaning**	**Default**
`persistBuffer`	Do not reclaim buffer space during scanning	*default is persistBuffer*
`stack`	Allow for start conditions to be stacked	*default is noStack*
`squeeze`	Generate the automaton with the smallest tables	*default is noSqueeze*
`summary`	Write out automaton statistics to the listing file	*default is noSummary*
`unicode`	Generate a unicode-mode (not byte-mode) scanner	*default is noUnicode*
`verbose`	Send *gplex*’ progress information to the console	*default is noVerbose*
`version`	Send *gplex* version details to the console	*default is noVersion*