The GPPG Parser Generator

John Gough, Wayne Kelly QUT

November 13, 2007

This document applies to GPPG version 1.2.0.*

1 Overview

These notes are brief documentation for the Gardens Point Parser Generator (*gppg*).

gppg is a parser generator which accepts a "*YACC*-like" specification, and produces a *C#* output file. Both the parser generator and the runtime components are implemented entirely in *C#*. They make extensive use of the generic collection classes, and so require **version 2.0** of the *.NET* framework.

Gardens Point Parser Generator (*gppg*) is normally distributed with the scanner generator Gardens Point *LEX* (*gplex*). The two are designed to work together, although each may be used separately.

If you want to begin by reviewing the input grammar accepted by *gppg*, then go directly to section 2.

1.1 Installing *GPPG*

gppg is distributed as a zip archive. The archive should be extracted into any convenient folder. The distribution contains three subdirectories. The "`bin`" directory contains two *PE*-files: *gppg.exe* and *ShiftReduceParser.dll*. Both of these must be on the executable path. The "`source`" directory contains all of the source code for *gppg*. The "`doc`" directory contains the files "`gppg.pdf`" and the file "`GPPGcopyright.rtf`".

Application programs that use parsers generated by *gppg* rely on the presence of the runtime component *ShiftReduceParser.dll*. This *PE*-file should be in the same directory as the assembly that contains the parser.

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

1.2 Running *GPPG*

gppg is invoked by the command —

`gppg` [*options*] *inputFile* > *outputFile*

the available options are —

* `/help` — displays a help message

* `/version` — displays version information
* `/report` — displays *LALR(1)* state information
* `/no-lines` — suppresses emission of output `#line` directives
* `/defines` — writes a "tokens" file with one token name per line
* `/verbose` — creates more detailed information on parser conflicts.
* `/gplex` — makes *gppg* customize its output for the Gardens Point *LEX* (*gplex*) scanner generator.
* `/conflicts` — writes a "conflicts" file with detailed information about any parser conflicts (see section 2.4).
* `/babel` — makes *gppg* emit the additional interface required by the *Managed Babel* package of the *Visual Studio SDK*, (see "Colorizing Scanners and *Managed Babel*" in section 2.3).

1.3 Using *GPPG* Parsers

Parsers constructed by *gppg* expose a simple interface to the user. Instances of the parser may be created by calling any of the constructor methods defined in the user code. The name of the parser class is *Parser*, unless the default is overridden (see 2.3). Typically other code will associate a scanner and error handler object with the parser instance. The scanner, in turn, will have been associated with some input text.

The parser instance is invoked by calling the *Parse* method, inherited from the abstract base class *ShiftReduceParser*. *Parse* has the following signature —

```
public bool Parse() { ... }
```

This method returns false if the parse is unsuccessful, and true for a successful parse. Note that the success or otherwise of the parse is distinct to the issue as to whether errors were detected. False implies that the parse terminated abnormally.

In general the parser is expected to do more than just return true or false. In many cases the parser will be expected to construct some kind of abstract syntax tree and/or symbol tables as a side effect of a successful parse. When this is the case, the parser result is normally attached to some accessible field of the parser instance from where it may be retrieved by the invoking process.

1.4 Outputs

The parser generator reads a grammar specification input file and produces a *C#* output file containing —

* an enumeration type declaring symbolic tokens

  ```
  public enum  Tokens {error=127, EOF=128, ... }
  ```

  The ordinal sequence of the tokens in the enumeration will start above the ordinal numbers of any literal characters appearing in the grammar specification.

* a type definition for the "semantic value" type specified in the grammar. In the case of a union type, *gppg* will emit —

```
public partial struct ValType { ... }
```

The semantic value type is the type that is returned by the scanner in the instance field *yylval*. This type argument thus corresponds to the *YYSTYPE* of traditional implementations of *YACC*-like tools. The struct is partial if the marker "`%partial`" appears in the definitions part of the parser specification "`*.y`" file.

* a definition for the class that implements the parser

```
public partial class
        Parser :  ShiftReduceParser<ValType, LocType> {
    ...
}
```

The class is partial if the marker "`%partial`" appears in the definitions part of the parser specification "`*.y`" file. This class definition provides an instantiation for the generic class *ShiftReduceParser* with the actual type arguments *ValType* and *LocType*, inferred from the grammar specification, substituted for the type parameters *YYSTYPE* and *YYLTYPE* respectively.

The generated *C#* source file, as well as defining the above types, also contains the parsing tables for the parser and the code for the user-specified semantic actions. The parser implements a "bottom-up *LALR*(1)" shift-reduce algorithm, and relies for its operation on an invariant runtime component "*ShiftReduceParser.dll*". The main class of the runtime is a generic class of two parameters which is instantiated with the two type arguments determined by the grammar specification.

If the command line option "/defines" is used, or the input file contains the "%defines" marker then an additional output file is created. This file will have the name "*basename*.tokens" where *basename* is the name of the input file, without a filename extension. This file contains a list of all of the symbolic (that is, *non-character-literal*) tokens, one per output line. The names are syntactically correct references to the underlying enumeration constants.

1.5 Scanner Interface

Parser instances contain a public field named *scanner*. The parser expects this field to be assigned a reference to a scanner that implements the class shown in Figure 1. Despite its name, *IScanner* is the abstract base class of the scanners, rather than an interface. The base class provides the *API* required by the runtime component of *gppg*, the library *ShiftReduceParser.dll*. Of course scanners will usually implement other facilities that are required by the scanner semantic actions. These actions will use the richer *API* that the concrete scanner class supports, but the shift-reduce parsing engine itself needs only the subset defined in the base class.

User code of the parser may also access the richer *API* of the concrete scanner class by casting the scanner reference from the abstract type to the actual concrete type.

The abstract scanner 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 (see section 2.3) then the type defaults to `int`. From version 1.2 of *gppg* the semantic value type can be any *CLR* type. Previous versions required a value-type.

The second generic type parameter, *YYLTYPE*, is the location type that is used to track source locations in the text being parsed. In almost all applications it is sufficient

Figure 1: Scanner Interface of *GPPG*

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

to use the default location type, *LexLocation*, shown in Figure 5. Location-tracking is discussed further in section 4.2

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

The first method, *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. Error handling in the parser is treated in more detail in section 3.

If *gppg* is used with the */gplex* option the parser file defines a wrapper class *ScanBase* which instantiates the generic *IScanner* class, and several other features. Full details of this and other convenience features of this option are given in the *gplex* documentation.

Using *GPPG* Parsers with Non-*LEX* Scanners

gppg has been successfully used with both hand-written scanners, and with scanners produced by tools such as *COCO/R* that are not at all *LEX*-like. In the case of newly hand-written scanners the code is written to conform to the *IScanner* interface. In the case of existing scanners, or scanners produced by other tools it is usually necessary to write adapter code to wrap the scanner *API* to conform to the expected interface.

2 Input Grammar

The input grammar for *gppg* is based on the traditional *YACC* language. There are a number of unimplemented constructs in the current version, and a small number of extensions for the *C#* programming environment.

gppg performs a small number of checks on the validity of the grammar that it is given. If a particular symbol does not appear in a token declaration, and does not appear

as the left-hand-side of at least one production, then the grammar is non-terminating. *gppg* issues a error message naming the symbol that is involved. This is a fatal error, as parser production fails under such circumstances.

As well as the terminating test, *gppg* checks that every non-terminal symbol is reachable from the start symbol. Any such symbols attract a warning, but their presence is not fatal to parser production.

Errors of both type most commonly arise because of typographical errors in the grammar. Remember that symbol names are case sensitive in *gppg*.

2.1 Declarations

gppg implements some of the declarations familiar from other parser generators, as well as a number of extensions that specifically have to do with the *.NET* platform.

The following symbols are recognized, with the standard meanings —

```
%union       // see section 2.3
%prec        // usual meaning
%token       // usual meaning
%type        // usual meaning
%nonassoc    // usual meaning
%left        // usual meaning
%right       // usual meaning
%start       // usual meaning
%locations   // usual meaning
```

The command "`%output=`*filepath*" redirects the output parser to the nominated file. In the absence of this declaration the output is sent to standard output. The filepath is terminated by whitespace, so if the filename includes spaces the filepath must be enclosed in quotes " and ".

Finally, the command "`%definitions`" creates a "tokens" file with a list of the symbolic tokens, one per line. The names are written in fully qualified form, with the enumeration typename prepended.

2.2 Unimplemented Constructs

gppg does not currently support the *Bison* "`%expect` *N*" construct.

Versions of *gppg* prior to version 1.0.4 only allowed semantic actions to refer to the first nine symbols in the right-hand-side of a production. This restriction is now removed.

2.3 Extensions to the Grammar

Naming Types

The name and visibility of the parser class may be defined by the "`%parsertype`" and "`%visibility`" constructs. In the absence of these *gppg* acts as though it had seen the declarations —

```
%parsertype Parser
%visibility public
```

Similarly, the name of the token enumeration may be set by the "`%tokentype`" declaration. In the absence of such a declaration *gppg* acts as though it had seen the declaration —

```
%tokentype Tokens
```

The visibility of the token type is the same as that declared for the parser class type.

Defining a Semantic Value Type

According to tradition, the semantic value type expected from the scanner, *YYSTYPE*, is defined by a "`union`" construct in the grammar specification file. Of course, *C#* does not have a union type construct, achieving roughly the same intent by subclassing.

Nevertheless, *gppg* recognizes the "`%union`" construct, emitting a corresponding *struct* definition to the output file. The structure will have a field corresponding to every member of the "union", with members selected using exactly the expected "dot" notation. The effect is to substitute a *product* type for the usual *union*, with the loss of some storage efficiency.

The name of the semantic value type may be explicitly declared as described below. The default name for the "union" type, in the absence of an explicit declaration will be "*ValueType*"[1].

If the grammar does not declare a "union" type, but does declare a semantic value type name, then the semantic value stack of the parser will expect to hold values of the named type. Thus in new grammars it is probably better to *define* a semantic value type in the *C#*, and declare the type's name to *gppg*, thus avoiding the slightly misleading union word. In some applications it is convenient to define the semantic value type to be the abstract base class of an abstract syntax tree construct. This allows the semantic actions of the parser conveniently to build the *AST*.

Partial Types

The "`%partial`" marker, at the beginning of a line in the .y file, declares that the generated parser class will be a partial class. This is a convenient mechanism to use, so that the bulk of the (non semantic action) code required by the parser may be defined in a separate file. By default *gppg* produces a complete class.

In the case that the grammar declares the semantic value type using the "`%union`" mechanism the generated parser file will declare a struct that is also *partial*.

The use of this partial marker is a very great convenience, allowing the grammar file to hold little but the grammar syntax, with all of the other code appearing in separate files. This is also a big gain with the definition of the semantic value type. Typically this type contains data and many instance methods for manipulation of the type. Without the partial marker all of these method bodies would need to be defined inside the dummy "`%union`" construct in the .y file.

`%namespace` *NameSpaceName*

The whole of the output of *gppg* will be enclosed in a namespace declaration with the given name. The name is used verbatim, and may be a dotted name.

[1] That is, the type name will be "*MyNamespace*`.ValueType`" which should not be confused with the super type of every value type `System.ValueType`.

%YYSTYPE *ValueTypeName*

If a grammar contains this marker the semantic value type will have the prescribed name in every used occurrence. The specified name should be a simple type identifier. "`%valuetype`" is a synonym for this marker.

If a grammar contains neither a valuetype declaration *nor* a "union" declaration, then the semantic value type will be `int`.

%YYLTYPE *LocationTypeName*

This marker overrides the default location type name, *LexLocation*. The default type is sufficient for most applications, but when additional functionality is required it is possible to define a new type, and declare its name with this marker.

`%using` *UsingName*

The given name is inserted into the output file, immediately before the namespace marker. There may be as many of these directives as is necessary, and the names may be either simple or dotted names.

`%using` *UsingName*

Colorizing Scanners and *maxParseToken*

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

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

If for some reason you wish to define token values that are not meaningful to the *gppg*-grammar, then define *maxParseToken* and place all the token values that the parser will ignore after this value.

Versions of *gplex* from version 0.4.1 use reflection to check if the special value of the enumeration is defined. It is always safe to leave the special value out, if it is not needed.

Colorizing Scanners and *Managed Babel*

The *Visual Studio SDK* includes tools to allow for easy contruction of language services based on the *Managed Package Framework (MPF)*. The *SDK* ships with the Managed Package Parser Generator (*mppg*) tool, but it is also possible to use *gppg* to construct a compatible parser.

MPF-compatible parsers do not require any changes to the grammar specification, other than possibly defining a *maxParseToken* enumeration value. The changes are all in the scanner base class definition that *gppg* emits when run with the /*gplex* option.

If *gppg* is run with the /*babel* option (which implies the /*gplex* option), then the emitted parser source file will define the *IColorScan* interface. Some additional features of the scanner base class, *ScanBase*, are also emitted. These allow the scanners to operate incrementally by providing end-of-line scanner state to be persisted.

2.4 Parser Conflict Messages

By default *gppg* sends a brief message to the error stream noting any shift/reduce or reduce/reduce errors detected during parser construction. More detailed messages are written to the error stream if the */verbose* command line option is used. Even more detailed information is generated in the case that the */conflicts* command line option is used. In that case the information is written to a file with the name derived from the input file name, but with filename extension "`.conflicts`".

Reduce/Reduce Conflicts

If a reduce/reduce conflict is detected, the conflicts file will contain information similar to that in figure 2. In this example there are two productions both of which can be

Figure 2: Reduce/Reduce Conflict Information

```
Reduce/Reduce conflict on symbol "error",
                     parser will reduce production 41
  Reduce 41: TheRules -> RuleList
  Reduce 52: ListInit -> /* empty */
```

reduced when the lookahead symbol is the error token. In such cases the parser will always choose the lower numbered production. Reduce/Reduce conflicts are generally a more serious matter than shift/reduce conflicts, so any instances of these need to be considered carefully. In this particular example, the conflict only affects the error-recovery behavior of the parser.

Shift/Reduce Conflicts

Shift/Reduce conflicts tend to be more common, and are often but not always benign. The conflicts file for a typical case will contain information similar to that in figure 3. In this example, with a current symbol of "`rCond`", the reduce action is to accept

Figure 3: Shift/Reduce Conflict Information

```
Shift/Reduce conflict on symbol "rCond",
                                    parser will shift
  Reduce 24: NameList -> error
  Shift "rCond":  State-79 -> State-80
    Items for From-state for State 79
      55 StartCondition: lCond error . rCond
      24 NameList: error .
              -lookahead: [ rCond, ]
    Items for Next-state for State 80
      55 StartCondition: lCond error rCond .
              -lookahead: [ pattern, ]
```

production 24. The alternative, shift action is to shift the token and move from state 79

to state 80. The current state, 79, has two "items" in its kernel set. The first item is production 55, after shifting an error, and expecting to next see the *rCond* symbol. The current position in the recognition of the production right-hand-side is marked by the dot. The second item is production 24, with the dot at the end. Since the dot is at the end, the action for this item is to reduce production 24. The default resolution of such conflicts is to shift, trying to munch the maximum number of tokens for each reduction. For this example, that is clearly the correct behavior.

For items which are complete, that is, those that have the dot at the end, the diagnostic file also shows the lookahead symbols that can validly appear at that point.

3 Error Handling

3.1 Parser Action

The default action of the parser, when neither a shift nor a reduce is possible, is to call the *yyerror* method of the scanner interface (see figure 1). The parser runtime then discards values from the parser state, value and location stacks until a state is found that can shift the synthetic "error" token. After the error token has been shifted the parser checks to see if an ordinary shift or reduce action is then possible given the existing lookahead symbol. If no such action is possible, the parser discards input tokens until an acceptable token is found or the input ends.

In the event that no state on the parser stack can shift an error token and the stack becomes empty, or if the input ends while discarding tokens, the *Parse* method returns false.

Syntactic error recovery sets a boolean flag which prevents cascading calls to *yyerror*. This flag is not cleared until three input tokens have been shifted without further syntactic errors resulting. This constraint does not apply to the reporting of any *semantic* error messages that are explicit in semantic actions.

In cases where it is certain that error recovery has succeeded a semantic action may clear the flag explicitly by a call to the built-in parser method *yyerrok*(). As well, the lookahead token may be explicitly discarded in a semantic action by calling the built-in parser method *yyclearin*().

3.2 Error Reporting

As noted, the parser will call *yyerror* in case of errors. If the scanner overrides the empty implementation in *IScanner* then that method may construct a suitable error message. It is useful to note that error recovery is attempted because the next input symbol is not a possible lookahead for either a shift or a reduce action. It is always the case that the input symbol that blocked progress is the symbol corresponding to the scanner's current *yylval* and *yylloc* at the moment that *yyerror* was called.

The default mechanism suffices for simple applications, but there are options for improved functionality. For example in many applications it is desired that a *list* of errors be constructed with associated text spans pointing into the input text.

The alternative strategy for constructing error messages is to leave *yyerror* empty, and place explicit calls to an error handler in the semantic actions of productions that mention the error token. Such calls to the error handler will be able to make good use of the automatic location tracking mechanisms of the parser to provide information for the error handler. For example, in the case of a missing member of some kind of

paired construct the semantic action should have access to the location information of the current lookahead symbol *and* the symbols whose pair was expected.

4 Notes

4.1 Semantic Actions

Commonly, the semantic action that is invoked at a reduction will perform some kind of computation on the semantic values of the symbols on the right of the selected production. The destination of the computed semantic value is denoted "`$$`", while the previously computed semantic values of the first, second and subsequent symbols on the right-hand-side are denoted `$1`, `$2`, ... `$`*n*, where *n* is any decimal number less than or equal to the length of the right-hand-side of the chosen production. The index *n* undergoes an index bounds check at parser construction time.

In case the semantic action needs to refer to a particular component of a semantic value of aggregate type, the notation `$<`*member*`>`N refers to the named member of the aggregate.

Default Semantic Action

For production right-hand-sides of zero length, the default semantic value of the production is a default value of the *YYSTYPE* type. For production right-hand-sides of all other lengths, the default action is equivalent to "`$$=$1`".

4.2 Location Tracking

The second generic type parameter of the scanner interface in figure 1, *YYLTYPE*, is the location type. Instances of the location type contain information that mark the start and end of the relevant phrase in the input text, that is, the type is a representation of a text span. The actual type that is substituted for the *YYLTYPE* parameter must implement the *IMerge* interface shown in Figure 4. The location type supplies a method that

Figure 4: Location types must implement *IMerge*

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

produces a value that spans locations from the start of the "this" value to the end of the "*last*" argument. The parser, during every reduction, calls the *Merge* method to create a location object representing the complete production right-hand-side phrase.

Location Actions

The semantic actions of the parser may refer to the location values as well as to the semantic values. This is most commonly done so as to pass location information to an error handler.

In a production, the location value of the left-hand-side symbol is referred to as `@$`, while the location values of the first, second and subsequent symbols on the right-hand-side are denoted `@1`, `@2`, ... *@n*, where *n* is any decimal number less than or equal to the length of the right-hand-side of the chosen production.

The default action at every reduction is equivalent to the code –

`@$ = @1.Merge(@`N`)`

where *N* is the number of symbols in the production right-hand-side. The default action is carried out *before* any user-specified semantic action. Thus it is possible for a user action to override the default location-merging action by explicitly attaching a different location object to "`@$`".

If a scanner does not contain code to generate location objects, then the scanner's *yylloc* field will always be null. This does not cause exceptions in the default location action, as the code is guarded by a null reference test. Location processing may thus be safely ignored in those cases that it is not needed.

Default Location Type

Parser specifications may declare the name of a type that is to be used as the location type. This type must implement the *IMerge* interface. In the event that no such declaration is made, the default location tracking type is the *LexLocation* type shown in Figure 5. This type implements a simple text-span representation.

Figure 5: 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);
    }
}
```

Supplying a Different Location Type

Sometimes if may be necessary to use a different location type. This is the case with *gplex* itself, which needs to track not only line and column numbers but also file-buffer positions.

To override the default location type, the parser specification needs to include the command —

`%YYLTYPE` *TypeIdent*

where *TypeIdent* is the simple name of the desired type[2]. The type must implement the *IMerge* interface, but may also provide whatever other methods are required.

In the case of the *LexSpan* type of *gplex* the type contains the same line and column fields as *LexLocation*. These are used by the error reporting in the usual way. The new type has additional fields for the start and end file position pointers into the input buffer, and a reference to the buffer itself. The additional methods of the type write out text spans from the buffer to specified output streams, and extract strings from the buffer corresponding to particular location spans.

Special Behavior for Empty Productions

Special care must be taken when generating location information for productions with empty right hand sides. The issue is not so much with the empty production, but when a location span from such an empty production is used further up a derivation tree.

Consider the production $A \rightarrow BCD$. The default location processing action when this production is reduced is to create a location span that begins at the start of the B phrase, and finishes at the end of the D phrase. Now, suppose that B and D are *nullable* symbols, and each has been produced by reduction by an empty production. A moment's consideration will show that the correct behavior is produced if the location span for each empty production *begins* with the start of the lookahead token, and *ends* with the finish of the last token shifted. Such a location value makes no sense on it own, it has negative length for example, but merges correctly with other spans. The 1.0.1 version of *gppg* uses this strategy to deal with location information for empty productions[3].

4.3 Reporting Errors Using the ErrorHandler Interface

There are a number of limitations to the low level reporting of errors using the *yyerror* method. In many applications it is preferable to define an error handler class that provides for the buffering and sorting of errors. Such error handlers need to specify the error location using a text-span of the *YYLTYPE* type. They should also select the error message by an ordinal number to allow for easy localization of the message text. Finally, the error handler needs to be callable from the semantic actions of the parser (and other semantic checking code) and by the scanner.

In use, the application will create an instance of its *ErrorHandler* class. A reference to the error handler object is either directly visible to the scanner or is copied to a field in the scanner. The scanner and parser will then be able to interleave error messages in the error handler buffer.

4.4 Copyright

[2] Since this is a single identifier, in the current version, the type must be accessible by its unqualified name.

[3] There are other ways of getting correct behavior, such as leaving the location value *null* and using conditional code for the default action that searches up and down the location stack to find non-*null* values to operate on.

5 Appendix A: *GPPG* Special Symbols

5.1 Keyword Commands

Keyword	Meaning
`%defines`	*gppg* will create a "*basename*.tokens" file defining the token enumeration that the scanner will use. The scanner does not need this text file, but it is useful for other tools.
`%left`	this marker declares that the following token or tokens will have *left associativity*, that is, $a \bullet b \bullet c$ is interpreted as $(a \bullet b) \bullet c$.
`%locations`	this marker is ignored in this version: location tracking is always turned on in *gppg*.
`%namespace`	this marker defines the namespace in which the parser class will be defined. The namespace argument is a dotted name.
`%nonassoc`	this marker declares that the following token or tokens are not associative. This means that $a \bullet b \bullet c$ is a syntax error.
`%output`	allows the output stream to be redirected to a specified, named file. See section 2.1.
`%partial`	this marker causes *gppg* to define a *C#* partial class, so that the body of the parser code may be placed in a separate *parse-helper* file.
`%parsertype`	this marker allows for the default parser class name, "*Parser*", to be overridden. The argument must be a valid *C#* simple identifier.
`%prec`	this marker is used to attach context-dependent precedence to an occurrence of a token in a particular rule. This is necessary if the same token has more than one precedence.
`%right`	this marker declares that the following token or tokens will have *right associativity*, that is, $a \bullet b \bullet c$ is interpreted as $a \bullet (b \bullet c)$.
`%start`	this marker allows the goal, (start) symbol of the grammar to be specified, instead of being taken from the left-hand-symbol of the first production rule.
`%token`	declares that the following names are tokens of the lexicon.
`%tokentype`	this marker allows for the default token enumeration class name, "*Tokens*", to be overridden. The argument must be a valid *C#* simple identifier.
`%type`	the form "`%type` < *member* > *non-terminal list*", where *member* is the name of a member in a union declaration, declares that the following non-terminal symbols have the stated type.
`%union`	marks the start of a semantic value-type declaration. See section 2.3.
`%using`	this marker adds the given namespace to the parser's using list. The argument is a dotted name, in general.

Keyword	Meaning
`%visibility`	this marker sets the visibility keyword of the token enumeration and the semantic value-type struct. The argument must be a valid *C#* visibility keyword. The default is public.
`%valuetype`	a synonym for *YYSTYPE*, deprecated.
`%YYSTYPE`	this marker declares the name of the semantic value type. The default is *int*.
`%YYLTYPE`	this marker declares the name of the location type. The default is *LexLocation*.

5.2 Semantic Action Symbols

Certain symbols have particular meanings in the semantic actions of *gppg* parsers. As well as the symbols listed here, the scanner will also define accessible symbols. Those for *gplex*-generated scanners are given in figure 1. figure 1.

Symbol	Meaning
`$$`	the symbolic location holding the semantic value of the left-hand-side of the current reduction.
`$`N	the value of the Nth symbol on the right-hand-side of the current reduction.
`@$`	the symbolic location holding the location span of the left-hand-side of the current reduction.
`@`N	the location span of the Nth symbol on the right-hand-side of the current reduction.
`YYABORT`	placing this symbol in a semantic action causes the parse method to return false.
`YYACCEPT`	placing this symbol in a semantic action causes the parse method to return true.
`YYERROR`	placing this symbol in a semantic action causes the parser to attempt error recovery. No error message is generated.
`YYRECOVERING`	this Boolean property denotes whether or not the parser is currently recovering from an error.
`yyclearin()`	placing this method call in a semantic action causes the parser to discard the current lookahead symbol.
`yyerrok()`	placing this method call in a semantic action asserts that error recovery is complete.

6 Appendix B: Change Log

This section tracks the updates and bug fixes from version "`0.9`" of July 2006. Changes prior to that version were the addition of additional markers in the grammar files, and removing some runtime failures.

Changes in version 1.2.0 (November 2007)

* *YYSTYPE* may be a reference type *or* a value type.

* a number of minor bug fixes, particulary making the parsing of grammar files with errors more robust.

Changes in version 1.0.4 (August 2007)

* New semantic markers *YYACCEPT, YYABORT, YYERROR, YYRECOVERING* have been added.
* The indices of semantic value and location value references in semantic actions can now be arbitrary decimal numbers, and are bounds-checked at parser construction time.
* The default semantic action for productions of length greater than one has been changed for greater compatibility with *Bison*.
* The parsing code of *gppg* has been made more robust, eliminating several cases where syntactically incorrect input crashed the program.

Changes in version 1.0.3 (March 2007)

* New command line option /*babel* added.
* Output with the /*gplex* option expects to interface with version 0.5.1 or greater of *gplex*.

Changes in version 1.0.2 (February 2007)

* Assembly attributes in *main.cs* now set the version string.
* A bug in the string representation of productions with literal terminal symbols involving backslash-escapes has been fixed.
* A bug in the parsing of semantic actions involving terms such "`$<kind>`*n*" has been fixed.
* Scanner variable *yylloc* defined in *IScanner* has become a property. This leads to better code in handwritten scanners.
* When invoked with the */gplex* option, *gppg* emits a *ScanBase* file that defines a private backing field for the *yylloc* property, and the accessor methods. *Existing* gplex *scanners will require recompilation to use the new wrapper semantics.*
* New command line option /*conflicts* creates a file "*basename*.`conflicts`" that has complete production and item information for any conflicts in the grammar (see section 2.4).

Changes in version 1.0.1 (January 2007)

* New command line option /*verbose* generates more informative warnings for shift/reduce and reduce/reduce conflicts.
* Default location-tracking behaviour for empty productions.
* Behavior of location-tracking when production starts with a nullable non-terminal symbol has been corrected.

* *Static* bounds checks for semantic action indices introduced so that erroneous usages are trapped at build time.

* Output stream was not correctly flushed when "`%output=`" marker used to redirect output from the specification file.

* Unmatched left braces in semantic actions caused *gppg* to loop. This now generates a "block ended by EOF" error.

Changes in version 1.0 (October 2006)

* Static grammar checking was added to the tool. *gppg* now checks that all non-terminals are reachable from the start symbol, and that all non-terminals possess at least one terminating production.

* Strange case of the shift-reduce parser looping during error recovery was fixed.

* New command line option /*defines* added.

* New command line option /*gplex* customizes parser output as expected by *gplex*-generated scanners.