The GPPG Parser Generator

(*Version 1.5.0 May 2012*)

John Gough, Wayne Kelly QUT

May 11, 2012

About GPPG

Gardens Point Parser Generator (*gppg*) is a parser generator that produces parsers written in the *C#* V2.0 language. *gppg* generates bottom-up parsers.

The generated parsers recognize languages that are *LALR*(1), with the traditional yacc disambiguations. There are a number of extensions of the traditional input language that are necessary for correctness of the generated *C#* output files.

The generated parsers are designed to interface cleanly with scanners generated by Gardens Point *LEX* (*gplex*). However, *gppg*-generated parsers have been successfully used with both handwritten scanners and with scanners generated by *COCO/R*.

A particular feature of the tool is the optional generation of an html report file that allows easy navigation of the finite state automaton that recognizes the viable prefixes of the specified language. The report shows the production items, lookahead symbols and actions for each state of the automaton. It also optionally shows an example of a shortest input, and shortest *FSA*-path reaching each state. This report file considerably simplifies the diagnosis and removal of grammar conflicts.

This Version at a Glance

Error messages are now in *MSBuild*-friendly format to interwork better with *Visual Studio*. This is the default behavior, with a new option allowing legacy messages to the console.

The format of the html output with the /report option has been enhanced to make tracing of the automaton easier. The format of the tracing output for parsers generated with the *TRACE_ ACTIONS* symbol has also been improved.

This version changes the default file encoding to Unicode, with a fallback to raw (uninterpreted bytes) if there is no *BOM* prefix.

When *gppg* is invoked with the /gplex option a *ScanObj* class is defined that encapsulates symbol information.

Additional features have been added to *gppg* and to *gplex* to support symbol pushback for cases where a backtracking automaton is required.

Some new options allow two parsers to share the same token definition, including token associativity and precedence. One parser specification defines the token enumeration and serializes the token dictionary. Client parser specifications deserialize the dictionary rather than defining their own token enumeration. This feature allows multiple parsers to interleave consumption of tokens from a shared scanner.

Contents

List of Figures

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 at least **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 wish to begin by reviewing the input grammar accepted by *gppg*, then go directly to section 2. Should you wish for a quick refresher on shift-reduce parsing, or definitions for the terms used in this documentation, go to Appendix B, section 9.

1.1 Installing *GPPG*

gppg is distributed as a zip archive. The archive should be extracted into any convenient folder. The distribution contains four subdirectories. The "`binaries`" directory contains two *PE*-files: *gppg.exe* and *ShiftReduceParser.dll*. The "`project`" directory contains the source code for *gppg*. The "`documentation`" directory contains the files "`gppg.pdf`" (this file), "`gppg-changelog.pdf`", the compiled html help file "`ShiftReduceParser.chm`" and the file "`GPPGcopyright.rtf`". Finally, the "`testfiles`" directory contains three test files "`Calc.y`", "`RealCalc.y`" and "`RealTree.y`". These simple tests are described in Sections 7.1, 7.2. and 7.3.

Application programs that use parsers generated by *gppg* may embed the invariant code of the runtime component *ShiftReduceParser* in the application assembly, or may access the separate assembly *QUT.ShiftReduceParser.dll*. The *DLL* is a strongly named component and may be installed in the *.NET* fusion cache, or may be placed in the same directory as the assembly that contains the parser.

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

1.2 Running *GPPG*

gppg is invoked by the command —

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

Options are case-insensitive, with the available options —

* `/babel` — causes *gppg* to emit the additional interface required by the *Managed Babel* package of the *Visual Studio SDK*, (see "Colorizing Scanners and *Managed Babel*" in section 2.6.9).
* `/conflicts` — writes a file "*basename*.`conflicts`" with detailed information about any parser conflicts (see section 3.4).
* `/defines` — writes a file "*basename*.`tokens`" with one token name per line.
* `errorsToConsole` — by default *gplex* generates error messages that are interpreted by Visual Studio. This command generates error messages in the legacy format, in which error messages are sent to the console preceeded by the text of the source line to which they refer.

* `/gplex` — makes *gppg* customize its output for the Gardens Point *LEX* (*gplex*) scanner generator.
* `/help` — displays the usage message.
* `/line-filename:`*name* — the `# line` markers in the output file reference the file given in the argument *name*. `/linefilename:`*name* is equivalent.
* `/listing` — cause *gppg* to always produce a listing file, "*basename*.`lst`". Without this option *gppg* produces a listing only if there are errors or warnings.
* `/no-info` — suppresses emission of the default header information in the output file that identifies the host machine and user. `/noinfo` is equivalent.
* `/no-filename` — suppresses emission of source filename in the output file header comment. `/nofilename` is equivalent.
* `/no-lines` — suppresses emission of output `#line` directives. The hyphen is optional, with "`/nolines`" having the same effect. `/nolines` is equivalent.
* `/noThrowOnError` — ensures that no internal exception escapes being reported to the error log.
* `/out:`*name* — the output is redirected to the file given in the argument *name*. `/o:`*name* and `/output:`*name* both equivalent commands.
* `/report` — generates a file "*basename*.`report.html`" with *LALR(1)* state information, as well as producing a parser.
* `/verbose` — sends more detailed information to the console, and more detailed information to the conflict and *LALR* reports.
* `/version` — displays version information for *gppg*.

The behavior of *gppg* when the */report* option is used with and without the */verbose* option is described in Section 3.5.

Return Codes

When *gppg* terminates it returns the following result codes —

0 `MC_OK` – *gppg* completed normally, possibly with errors.

1 `MC_FILEERROR` – the specified input file for *gppg* either could not be found or was found but could not be opened.

2 `MC_TOOMANYERRORS` – *gppg* terminated with the *Too Many Errors* error. The first fifty errors will be written to the list file.

3 `MC_EXCEPTION` – *gppg* terminated with an unexpected exception. If the *noThrowOnException* flag was set, the identity of the exception will be written to the standard error stream.

These return codes are available to be used by *msbuild* or other build engines.

No matter what the return code, any buffered error messages will be sent to the listing file before termination.

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 Section 2.4). User code typically attaches a scanner and error handler object to the parser instance (see Section 1.6). The scanner, in turn, will have been given some input text to read from.

The parser instance is invoked by calling the *Parse* method, inherited from an abstract base class. The class *ShiftReduceParser* is a generic abstract class with two type-parameters: a semantic value type *TValue* and a location text-span type *TSpan*. The user specification chooses appropriate type-arguments for the concrete parser class.

The *Parse* method cannot be overridden, and 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 from 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. Be aware that the use of unicode escapes for character literals may push this boundary very high.

* 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, as shown in Figure 1. The parser 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 *TValue* and *TSpan* respectively.

Figure 1: Concrete parser class

```
public class Parser:ShiftReduceParser<ValType, LocType>
{
    // Property inherited from ShiftReduceParser
    //protected AbstractScanner<ValType, LocType> Scanner
    //{ get; set; }
    ...
}
```

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 the invariant code of the generic *ShiftReduceParser* class. This code may be imported as a source file of the application, or accessed from the strongly named assembly "*ShiftReduceParser.dll*". Section 5 discusses various architectural options for the parser runtime.

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].

1.5 Scanner Interface

Parser instances contain a protected property named *Scanner*. The parser expects this field to be assigned a reference to a scanner that implements the class shown in Figure 2. *AbstractScanner* is the abstract base class of the scanners. The base class provides the *API* required by the runtime component of *gppg*, the library *ShiftReduceParser.dll*. Of

Figure 2: Scanner Interface of *GPPG*

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

course scanners will usually implement other facilities that are required by the scanner

[1] In version 1.4.7 an additional mechanism is provided for directly sharing token definitions. This is described in Section 5.4.

semantic actions. These actions may 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 concrete type.

The abstract scanner class is a generic class with two type parameters. The first of these, *TValue* is the "*SemanticValueType*" of the tokens of the scanner. If the grammar specification does not define a semantic value type (see section 2.6.3) 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, *TSpan*, is the location type that is used to track source locations in the text being parsed. In almost all applications it is sufficient to use the default location type, *LexLocation*, shown in Figure 4. Location-tracking is discussed further in section 2.9.

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 4.

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

1.6 Instantiating the Parser Object

The parser code generated by the *gppg* tool does not declare any constructor methods at all. Furthermore, the generic base class declares a single constructor only.

```
protected ShiftReduceParser
                (AbstractScanner<TValue, TSpan> scanner) {
    this.scanner = scanner;
}
```

This constructor is the *preferred* method for attaching a scanner object to the parser. However, in the interests of compatibility with previous versions, a no-arg constructor for the parser might pass `null` to the base class constructor, and then use the setter of the parser's *Scanner* property to add the scanner reference later.

It is up to the user to define how the concrete parser object is to be created. Either the parser specification must define one or more constructor methods for the parser class, or constructors may be defined in a *parse helper* file which contains more of the partial parser class.

For a typical application the parser class will have other members that may conveniently be initialized in the constructor. For example, in *gppg* itself the parser class defines a single constructor with the following header text —

```
internal Parser(
    string filename,
    Scanner scanner,
    ErrorHandler handler) :  base(scanner) { ...  }
```

For this example the inheritance relations of the *Scanner* class are as follows. *Scanner* is defined by *gplex*, and derives from the *ScanBase* class that *gppg* emits when it is run with the */gplex* option. *ScanBase* is a wrapper for the *AbstractScanner* class, instantiated with type arguments *ValueType* and *LexSpan*.

1.7 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 may be written to conform to the *AbstractScanner* 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 adapt 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.

The rules of the grammar are specified in terms of *terminal symbols*. and *non-terminal symbols*. "Terminal" symbols are so named because they appear at the *leaves* of derivation trees, thus terminating the substitution process. They may correspond to a single input source sequence, such as a semicolon character '`;`', or may denote an unbounded *lexical category* such as "identifier" in most programming language lexicons. The terminal symbols correspond to the various lexemes recognized by the scanner. When each lexeme is recognized the scanner passes the parser a *token* and optionally a semantic value and a location object. Tokens are integer values that correspond either to members of a parser-defined enumeration, or are the ordinal values of single characters. The single character tokens do not need to be declared in the parser specification, but the enumeration names must be declared. It is also possible to declare a correspondence between a particular token-enumeration value and some fixed literal string, the *display string*. In such cases it is allowed to use the display string to denote the terminal symbol in the grammar rules.

Non-terminal symbols denote the *syntactic categories* of the phrase-structured grammar. They are implicitly defined by their appearance in a production rule of the grammar.

gppg performs 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. *Unreachable 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 Input File Encoding

Version 1.5.0 of *gppg* expects its input file to be a Unicode file with a valid byte order mark (*BOM*). By default, text files produced by *Visual Studio* will be in *UTF-8* format. Versions of *gppg* prior to 1.5.0 scanned their input using a byte-mode scanner.

If *gppg* reads an input file that does not have a valid *BOM* the input scanner reverts back to the "raw" codepage and reads input byte-by-byte. Thus input files prepared for previous versions of *gppg* will be read and interpreted as before.

2.2 Input Grammar Structure

The overall structure of the grammar is described by the following production rules

Grammar
: *DefinitionSequence*$_{opt}$ "%%" *RulesSection* *UserSection*$_{opt}$
;
DefinitionSequence
: *DefinitionSequence*$_{opt}$ *Declaration*
| *DefinitionSequence*$_{opt}$ "%{" *CodeBlock* "%}"
;
UserSection
: "%%" *CodeBlock*
;

All of the tokens begining with the "percent" character must occur alone at the start of a line. *CodeBlock* is any fragment of well formed *C#* code.

2.3 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. Further details are summarized in Appendix A, Section 8.1 —

```
%union          // usual meaning, but see section 2.6.3
%prec           // usual meaning, see section 2.7.2
%token          // usual meaning
%type           // usual meaning
%nonassoc       // usual meaning
%left           // usual meaning
%right          // usual meaning
%start          // usual meaning
%locations      // usual meaning
```

The following are extensions to the syntax, or have modified semantics —

```
%output          // sets the output filepath
%definitions     // creates a token declaration file
%sharetokens     // emit the token type for import by other parsers
%importtokens    // import the emitted token type from another parser
%namespace       // declares the namespace for the parser
%parsertype      // names the parser class within namespace
%scanbasetype    // names the scanner base class
%tokentype       // names the token enumeration
%visibility      // declares the visibility of the parser class
%YYSTYPE         // names the semantic value type
%YYLTYPE         // names the location value type
%partial         // declares the parser class to be partial
%using           // inserts a "using" clause in parser prolog
```

All of these extensions to the declaration syntax are described in Section 2.4.

2.3.1 Declaring Tokens

The `%token`, `%left`, `%right` and `%nonassoc` keywords may all be used to declare token names. Although the tokens have different semantics according to how they are declared, the syntax of all of these declaration forms are the same. Here are two examples.

```
Declaration : ... // Productions for other declarations
    | "%left"  Kind_opt  TokenList
    | "%token" Kind_opt  TokenList
    ;
Kind
    : '<' ident '>'
    ;
TokenList
    : TokenDecl
    | TokenList ','_opt TokenDecl
    ;
TokenDecl
    : litchar
    | ident number_opt litstring_opt
    ;
```

In this syntax *ident*, *number*, *litchar* and *litstring* are lexical categories recognized by the *gppg*-scanner.

The optional *Kind* clause declares that the semantic values of the following token-list elements are accessed by using the nominated identifier as a field-selector on the *yylval* variable.

Elements of a *TokenList* may be either whitespace-separated or comma-separated. They consist of either a literal character (enclosed in single quotes) or an identifier. Literal character tokens do not *need* to be declared, unless they require a kind declaration.

In the case of named tokens the identifier must be a legal *C#* identifier, and may be followed by an optional number and optional literal string. The optional number is for compatability with other tools, but the value is ignored, with a warning to the user[2]. The literal string associates a *display string* with the token. The display string is used in all diagnostic messages from the generated parser. This is particularly helpful so

[2] Future versions may honor the numeric assignment.

that, for example, a user error message could say “expected `"=>"` symbol” instead of using whatever cryptic identifier name that symbol has in the *Tokens* enumeration.

Both defining and used occurrences of literal character tokens may use the character that they denote, or any of the “usual”, octal, hexadecimal or unicode escape forms that denote the same value. All such occurrences are canonicalized so that, for example, the same lexical value may be referred to as ‘`\n`’, ‘`\012`’, ‘`\x0a`’, or even ‘`\u000a`’.

2.3.2 Token Precedence

For expression grammars there are two ways of controlling the precedence of operators, so as to implement the desired grouping of sub-expressions. One way is to invent a hierarchy of syntactic categories (*expression, simple-expression, term, factor, primary* and so on) to control the order in which derivation steps are invoked. This is the method that must be used for *predictive* or *top-down* parsers.

The “multiple sub-expression categories” method works perfectly well for bottom up parsers such as those generated by *gppg*, but it is traditional to use the second method. In this case, the application of a particular production rule is determined by attributes of the lookahead token.

Tokens may be declared as having *left* or *right* associativity, or being non-associative. Furthermore, the relative precedence of all such tokens is determined by the order in which they are declared. Tokens declared in the same list have the same precedence, while those declared in later lists have higher precedence than those on all earlier lists.

There is a special mechanism that can be used for those unusual cases of tokens that have more than one precedence. The familiar example of this occurs for the “minus” sign of conventional arithmetic grammars, where the same token may denote subtraction (which has low precedence), and unary negation (which has very high precedence). The special mechanism is described in Section 2.7.2.

2.3.3 Declaring Non-Terminal Symbol Types

Just as different tokens may pass different semantic values to the parser, so the recognition of different non-terminals may create different semantic values on the parser’s semantic value stack.

Semantic actions in the rules section can push a value onto the semantic value stack by using the symbolic code `$$ =` *Expression*. If the semantic value type is some named aggregate type, then the assignment will need to target one of the members of that type.

The code to achieve this is automatically generated by *gppg*, after the following declaration —

```
Declaration : ... // Productions for other declarations
    | "%type" Kind NonTerminalList
    ;
Kind
    : '<' ident '>'
    ;
```

The identifier in the *Kind* clause is the name of the member of the aggregate which will hold the semantic value for specified symbols. Similarly, if a semantic value is referenced using the symbolic name `$`*N*, where *N* is an index, then the appropriate member selection code will automatically be generated by *gppg*.

The *NonTerminalList* is a list of non-terminal symbol names. Elements of the list may be either comma-separated or whitespace-separated.

2.4 Extensions to the Declaration Grammar

2.4.1 Declaring an Output Filepath

The command —

`%output=`*filepath*

redirects *gppg* output to the nominated file. In the absence of this declaration the output is sent to standard output.

It is necessary for *gppg* to be able to send its output to an arbitrarily named file, including filenames that cannot be expressed in an 8-bit text file. The scanner accepts three different forms for the filepath —

* An ordinary unquoted filename which does not contain any whitespace or escaped characters.
* A normal literal string using *C#* conventions. This string may include whitespace, and any of the usual escape character forms.
* A verbatim literal string using the *C#* "`@"..."`" convention. This form is particularly convenient if the path contains backslash escapes as path-component separators.

For the last two forms the filepath string expands any escape characters before use. However, *gppg* does not check the legality of the resulting filepath string. Thus illegal filenames will only be detected when use is attempted.

2.4.2 Creating a Token Definitions File

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. This file is not used by the parser or scanner, but is useful for other tools.

2.5 Sharing the Token Definitions

The command —

```
%sharetokens
```

causes the parser to serialize its terminal-symbol dictionary to data file named *basename*.dat. This data may be imported by other parser specifications. Note especially that the data not only lists those symbols that appear in the tokens enumeration, but also preserves the definitions of the literal single character symbols, their associativty and precedence.

See Section 5.4 for more on this option.

2.6 Importing the Token Definitions

The command —

`%importtokens=`*filepath*

causes *gppg* to construct the terminal symbol dictionary by deserializing the data file produced by another parser specification. As with the case with the `%output` command, the filepath may be any of —

* An ordinary unquoted filename which does not contain any whitespace or escaped characters.
* A normal literal string using *C#* conventions. This string may include whitespace, and any of the usual escape character forms.
* A verbatim literal string using the *C#* "`@"..."`" convention. This form is particularly convenient if the path contains backslash escapes as path-component separators.

For the last two forms the filepath string expands any escape characters before use. However, *gppg* does not check the legality of the resulting filepath string. Thus illegal filenames will only be detected when use is attempted.

See Section 5.4 for more on this option.

2.6.1 Choosing the Namespace — `%namespace` *NameSpaceName*

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

2.6.2 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
```

When run with the */gplex* option, *gppg* defines a scanner base class instantiated from the generic *AbstractScanner* class. The "`%scanbasetype`" declaration allows the name of the base type to be set. In the absence of of such a declaration *gppg* acts as though it had seen the declaration —

```
%scanbasetype ScanBase
```

Note carefully that names of the token type and the scanner base class must be agreed between the parser and scanner specifications. The command syntax for this is the same in the two tools.

The visibility of the token type is the same as that declared for the parser class. Similarly, the visibility of the *ScanBase* abstract class that *gppg* defines when given the */gplex* option is the same as that of the parser class.

2.6.3 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 type declared by the union construct may be an arbitrary type. For example, the declaration in *gppg*'s own parser specification is

```
%union { public int iVal;
         public List<string> stringList;
         public List<TokenInfo> tokenList;
         public TokenInfo tokenInfo;
         public Production prod;
         public ActionProxy proxy;
       }
```

Note the use of types from *System.Collections.Generic* here.

The default name for the "union" type, in the absence of an explicit declaration will be "*ValueType*"[3]. For another example of use of the "`%union`" construct see Section 7.2.

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*, using the `%YYSTYPE` declaration, 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*.

2.6.4 Choosing the Semantic Value Type Name —

`%YYSTYPE` *ValueTypeName*
`%YYSTYPE` *TypeConstructor*

This declaration defines the type that will be used as the semantic value type. "`%value-type`" is a deprecated synonym for the `%YYSTYPE` marker. The first form simply declares the *name* of the type. If there is a `%union` declaration, then the name should be a simple identifier, and will be the name given to the struct that implements the "union". If there is no `%union` declaration, then the name may be a qualified ("dotted") name that references a named type defined elsewhere.

The second form of the declaration allows an arbitrary type constructor to define the semantic values. Using this form is the only way to declare a semantic value type that is an array type, for example, since in *C#* array types do not have identifier names. The type-constructor form cannot be used if there is a `%union` declaration.

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

2.6.5 Choosing the Location Type Name —

`%YYLTYPE` *LocationTypeName*

This marker overrides the default location type name, *QUT.Gppg.LexLocation*. The location type name may be a dotted name. 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. Location tracking is discussed in detail in Section 2.9

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

2.6.6 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.

2.6.7 Using Declarations — `%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[4]. The following namespaces are included by default, and need not be explicitly imported —

* *System*
* *System.Text*
* *System.Globalization*
* *System.Collections.Generic*
* *QUT.Gppg;* *// To access ShiftReduceParser*

2.6.8 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 *yylex* will only return values less than this constant.

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

If for any 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.

Scanners produced by current versions of *gplex* use runtime reflection to check if the special value of the enumeration is defined. If the value is not defined, it is set to *Int32.MaxValue*. It is always safe to leave the special value out, if it is not needed.

[4] The semicolon following the using name is new in version 1.3.5, and is the preferred usage. This follows the usage in *gplex* and in *C#*, where using declarations need a semicolon, and namespace declarations do not. The form without the semicolon is deprecated, but still recognized.

2.6.9 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.

2.7 Production Rules

The production rules for each non-terminal consist of the symbol name, starting on a new line in the first column, followed by a colon character, and zero or more right-hand-sides. Right-hand-sides are separated by the vertical bar character '|', and the sequence is terminated by a semicolon.

Rule
: *NonTermSymbol* ':' $RhsSequence_{opt}$ ';'
;
RhsSequence
: *RightHandSide*
| *RhsSequence* '|' *RightHandSide*
;

With the exception of a few possible special cases discussed later, a production right-hand-side consists of a sequence of zero or more symbols, followed by an optional semantic action. The symbols may be terminal or non-terminal symbols, including those terminal symbols that are denoted by a character literal. At runtime in the *gppg*-generated parser, when a token sequence corresponding to that production right-hand-side has been recognized, the semantic action, if there is one, is executed. The special case of a production right-hand-side with zero symbols is called an *empty production* and derives exactly one string, the *empty* string "". Reduction by an empty production is also called an *erasure* since the non-terminal symbol disappears, and is replaced by nothing.

All of the productions for a given non-terminal symbol may occur together in the specification, with separate right-hand-sides separated by the vertical bar. Alternatively, the productions for a symbol may be spread throughout the grammar in multiple production groups each beginning with the non-terminal name.

2.7.1 Semantic Action Syntax

Semantic actions consist of arbitrary *C#* statements enclosed in braces. The semantic actions are not checked or interpreted in any way by *gppg*[5]. The semantic action ends when the right brace is located that matches the left brace that began the action. Malformed actions that do not have matching braces lead to syntactic errors from which it is difficult for the *gppg* parser to recover.

[5]Except of course for recognizing literal strings and comments, so as to safeguard the matching of left and right braces.

As well as regular *C#* code, the semantic actions may contain a number of special symbols that refer to attributes of the rule just matched. A summary of these special symbols is given in Section 8.2, and their use is discussed in Section 2.8.

2.7.2 Controlling Precedence

The ordinary rules of relative precedence, and associativity for operator-like symbols are sufficient for grammars where such symbols have an unique precedence. However, for those rare cases where symbols have different precedence in differing contexts a special feature of *YACC*-like grammars must be used.

As an example, we consider a simplified version of the expression grammer in the *Calc* example of Section 7.1. The simplified version has only three operators, and the following relevant productions.

```
expr
    : '(' expr ')'
    | '-' expr %prec UMINUS
    | expr '-' expr
    | expr '+' expr
    | expr '*' expr
    ;
```

The token declarations for this grammar give '`-`' and '`+`' a lower precedence than '`*`', and give the highest priority to the dummy "token" *UMINUS*. All of these tokens are declared as having "`%left`" associativity. The second right-hand-side has special markers that say that that production should have the precedence of the *UMINUS* dummy token.

If we generate a parser from this grammar, and another from the same grammar but without the precedence marker we may compare them. Using the */report* option of *gppg* and examining the generated html files shows that only one state of the parser is different between the two versions. The "kernel items" for that state are identical —

Kernel Items

'`-`' *expr* •
expr • '`-`' *expr*
expr • '`+`' *expr*
expr • '`*`' *expr*

In words, the kernel items show the position within the recognition of various right-hand-sides that cause the automaton to be in this particular state. The "dot" marks the current position. Clearly we are either about to reduce (that is, finalize) recognition of the first production (since the dot is at the end), or we are in the middle of one of the other productions and about to shift a binary operator.

In situations such as this, where there are both shift and reduce possibilities *gppg* determines, for each possible lookahead token, whether the generated parser will shift the next token or reduce a completed production. It makes this decision by comparing the precedence of the completed right-hand-side with the precedence of each possible lookahead symbol. Since in this case we have forced the second production to have the highest possible priority, we will always reduce by that production when in this state.

In the absence of the "`%prec`" marker the situation is rather different. If the lookahead is '`*`' we shift the operator and continue parsing, since '`*`' has a higher precedence than '`-`'. For all other lookahead symbols, the precedences of the lookahead and the production are equal, and the parser reduces, since the '`-`' operator is declared to be left-associative.

This is but one example, so we must generalize this by stating the general rules by which precedence is determined. When both shift and reduction rules apply to a state the precedence of the *production* and the precedence of the *lookahead token* are compared. Here are the rules for determining precedence —

* The precedence of a *token* is determined by the position of the declaration group in which it occurs. Groups declared later in the definitions section have higher precedence (see also Section 2.3.2).
* The precedence of a *production* is that given by the "`%prec` *TokenName*" declaration, if there is one.
* Otherwise, the precedence of a production is that of the rightmost terminal symbol in the right-hand-side, if there are any terminal symbols in the right-hand-side.
* Otherwise the production has zero precedence.

And here are the rules for comparing precedence —

* If the precedence of the production is higher than the precedence of the lookahead token, then reduce.
* Otherwise, if the precedence of the lookahead token is higher than the precedence of the production, then shift.
* If the precedences are equal and the associativity of the lookahead token is *left* then reduce.
* If the precedences are equal and the associativity of the lookahead token is *right* then shift.

It is important to note that these rules are applied during the generation of the parsing tables, and not at runtime for the generated parser.

Finally, here are the rules that *gppg* uses for deciding when to issue conflict diagnostics during the generation of the parsing tables.

* If an automaton state has two or more productions that can be reduced, that is, two or more items with the "dot" at the end, then issue a reduce/reduce conflict warning.
* If an automaton state has a reduction and also possible shift actions, then the conflicts are resolved as detailed above. However, if the conflict is resolved in favor of shifting because the production has zero precedence, then issue a shift/reduce conflict warning.

2.7.3 Mid-Rule Actions

It is uncommon, but nevertheless legal, to place semantic actions in the middle, or even the beginning of production rules. In effect, the parser generator performs a transformation of the production as described below.

Suppose that we have a production —

A : *B* { *MRA* } *C* ;

where *MRA* is some mid-rule action.

This production is treated as if transformed by replacing the mid-rule action by a new, anonymous non-terminal symbol *Anon*, say. The new symbol has a single, empty production, and takes the code of the mid-rule action as a normal, end-of-rule action.

```
A    : B  Anon  C  ;
Anon : /* empty */ { MRA }  ;
```

The use of mid-rule actions sometimes leads to parser conflicts that would not occur without the action. This may be understood by considering the example above. Consider two productions —

```
A: B  C  ;
A: B  D  ;
```

We shall assume that the non-terminal symbols *C* and *D* have overlapping first terminal symbol sets. To be specific, let us assume that either *C* or *D* can start with terminal symbol x.

The fact that these two non-terminals have overlapping first sets does not cause a conflict between the two productions. The parser does not have to choose between the two productions until it has seen *all* of the symbols that make up a complete *C* or a complete *D*.

Suppose however that we now introduce a mid-rule action in the first of these productions. After the transformation described above, we consider the state with the following two items —

Kernel Items

A • *Anon* *C*
A • *D*

Now here is the problem: a lookahead token of x in this state will be consistent with the reduction *Anon*→*empty*, but is also consistent with shifting the first token of an expected *D* symbol.

Thus, introducing the mid-rule action can cause a shift/reduce conflict that was not there before. In effect, putting in a mid-rule action sometimes forces the parser to choose between two productions before it has seen enough of the input to make that decision.

If introducing a mid-rule action causes a damaging shift/reduce conflict the correct strategy is to take the action out. The idea is to perform the action *after* the whole production has been recognized. In order to do this it may be necessary to store away some additional information in the semantic values of the intermediate symbols to use in the later action.

A final, important point to remember is that if a mid-rule action is introduced the counting of symbols for the `$`*N* and `@`*N* terms in semantic actions must count one for each mid-term action. This is to account for the anonymous non-terminal that stands proxy for the action in the transformed production.

2.7.4 Right-Hand-Side Syntax

We are now in a position to reveal the complete syntax of production right-hand-sides. This looks a little silly, since it acknowledges that a production right-hand-side may have an action at either end, and between any two symbols. Furthermore, an optional precedence-setting clause may occur anywhere preceding a point at which an action may be placed.

2.8 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.

2.8.1 Default Semantic Action

Whenever a reduction is performed a default semantic action is performed, whether or not there is an explicit user-specified action also[6].

For production right-hand-sides of zero length, that is, for an *erasure*, the default semantic value of the production is a default value of the *TValue* type. If the semantic value type is a reference type, the value will be `null`. For scalar types the value will be "0". For structured value types the default value is the value created by the no-arg constructor. For production right-hand-sides of all non-zero lengths, the default action is equivalent to "$$=$1".

2.9 Location Tracking

The second generic type parameter of the scanner interface in figure 2, *TSpan*, is the location type. Instances of the location type typcially 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 *TSpan* parameter must implement the *IMerge* interface shown in Figure 3. The location type supplies a method that

Figure 3: 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.

2.9.1 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.

[6] Note that the unconditional nature of the default action implies that if the default action is part of the desired action there is no need to specify the default action explicitly.

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.

2.9.2 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 4. This type is defined in *ShiftReduceParser* and implements a simple text-span

Figure 4: Default location-information class

```
public class LexLocation :  IMerge<LexLocation>
{
    public int startLine; // Start line
    public int startColumn; // Start column
    public int endLine; // End line
    public int endColumn; // End column
    public LexLocation() {};

    public LexLocation(int sl; int sc; int el; int ec) {
      startLine = sl; startColumn = sc;
      endLine = el; endColumn = ec;
    }

    public LexLocation Merge(Lexlocation last) {
      return new LexLocation(
        startLine,startColumn,last.endLine,last.endColumn);
    }
}
```

representation.

2.9.3 Supplying a Different Location Type

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

positions. *gppg* uses these text-span values to access verbatim representations of user semantic actions.

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[7]. The type must implement the *IMerge* interface, but may provide any additional methods that are required.

In the case of the *LexSpan* type of *gppg* the type contains the same line and column fields as the default *LexLocation* type. 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 extract strings from the buffer corresponding to a given location span, and write out text spans from the buffer to the output streams.

2.9.4 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 a 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[8].

3 Errors, Diagnostics and Warnings

When *gppg* processes an input grammar it checks for a number of different conditions that might make the grammar invalid. If the grammar is well-formed it proceeds to construct an automaton to recognize the language specified by the grammar. If the grammar has conflict states *gppg* reports this.

In the case that there are errors or conflicts in the grammar *gppg* can give several levels of diagnostic help to the user. This section describes all of these errors, warnings and diagnostic messages.

3.1 Error Messages

gppg error messages From version 1.3, the parser generator uses a *gppg*-generated parser, and attempts error recovery from syntax errors. Error messages are buffered, and a listing file is produced if any errors or warnings are emitted, or if the */listing*

[7] This type identifier may be a qualified, "dotted" name.

[8] 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.

command line option is in force. In the listing, the location of the error is highlighted. In some cases the error message includes a variable text indicating the erroneous token or the text that was expected. In the following the variable text is denoted <...>.

Bad format for decimal number —
The *gppg* scanner has failed to compute the value of the apparent decimal number.

Bad separator character in list —
Lists may either be comma-separated or whitespace-separated.

Code block has unbalanced braces '{', '}' —
A code block has been terminated (by *EOF* or "%}") before finding a balancing number of right braces.

Duplicate definition of Semantic Value Type name —
There are duplicate definitions of the semantic value type name. Both occurrences are flagged.

Invalid string escape <...> —
The escape sequence in the placeholder in invalid in this literal string.

Keyword "%}" is out of place here —
This keyword is invalid in this context.

Keyword must start in column-0 —
All of the %-keywords must be left justified.

Literal string terminated by *EOL* —
The literal string reached end of line without finding a terminating quote character. Linebreaks are permitted in verbatim literal strings.

NonTerminal symbol "<...>" has no productions —
This is a fatal error. Carefully check to see if a rule has been left out, or whether a symbol has simply been misspelled.

Only whitespace is permitted here —
Many of the formatting keywords must occur alone on a line and can only be followed by whitespace or comment.

Premature termination of code block —
A %% separator has terminated a code block while still inside one or more nested braces.

Semantic action index is out of bounds —
The index into the production right-hand-side is out of bounds. Indices start from 1, and cannot exceed the number of symbols in the rule, counting mid-rule actions as an additional symbol.

Syntax error, unexpected <...>, expecting <...> —
This is the general, *ShiftReduceParser*-generated syntax error message. The second place holder is a list of the expected lookahead symbols at the error site.

Source file <...> not found —
The specified source file was not found.

There are <...> non-terminating NonTerminalSymbols {**<...>**} —
The second place-holder lists the non-terminating non-terminal symbols.

This character is invalid in this context —
In the current scanner state, this character does not form part of any legal token.

This name already defined as a terminal symbol —
A duplicate definition of this terminal symbol has been declared.

Unknown %keyword in this context —
The selected keyword is either unknown, or is invalid in this context.

Unknown special marker in semantic action —
This symbolic marker in the semantic action is unknown.

Unterminated comment starts here —
The input file ended while inside a comment. The text span in the error message is the start of the unterminated comment.

With *%union*, *%YYSTYPE* can only be a simple name —
If the specification defines a "union" type, then any declaration of *%YYSTYPE* can only give a simple name to the type. Without the union declaration *%YYSTYPE* can define an arbitrary type-constructor, including dotted names, arrays, instantiated generic types and so on.

3.1.1 Non-Terminating Diagnostics

After the grammar has been parsed *gppg* checks that every non-terminal symbol of the grammar is *reachable*, and that there is at least one production for each non-terminal. There is a separate check that every non-terminal is *terminating*.

A non-terminal symbol is reachable if it is the goal symbol, or if it occurs on the right-hand-side of a production with a reachable left-hand-side. If a non-terminal symbol is unreachable this means that there is no sequence of derivations starting from the goal symbol that produces a sentential form containing that symbol.

A non-terminal symbols is non-terminating if there is no sequence of productions that starts from the given symbol and derives a sequence of terminal symbols.

If a grammar has an unreachable symbol a warning is issued, but *gppg* can continue. However if a grammar contains a reachable symbol with no productions, or a non-terminating non-terminal then the error is fatal.

When a grammar symbol is unreachable it is almost always a simple typographical error in the input grammar. Often a whole sub-grammar may become unreachable because a single production has been omitted from the input. Similarly, when a non-terminal symbol is mis-spelled the resulting grammar will often have both an unreachable non-terminal *and* a non-terminal with no produtions. This is often the result of different occurrences of what was meant to be the same symbol being spelled with different case characters.

3.2 Warning Messages

If warnings are issued, but there are no errors detected, then an automaton is created. The user should note these warnings however, since some of them indicate possible errors in the grammar.

`%locations` **is the default in *gppg*** —
This keyword is included for compatability, but is unnecessary, as it is the default for *gppg*.

Highest char literal token <...> is very large —
The use of unicode escapes for literal character tokens is permitted, but the use of characters with high codepoints pushes the start of the token enumeration up to unusually high values.

Mid-rule `%prec` **has no effect** —
gppg allows a precedence marker to be attached to any action, including those that occur mid-rule. However, such a mid-rule precedence marker has no effect since mid-rule actions match a notional empty string, and are executed for all valid lookahead symbols.

NonTerminal symbol "<...>" is unreachable —
An unreachable NonTerminal is not fatal to parser generation, but usually indicates an error, or at least misunderstanding, in the specification file.

Optional numeric code ignored in this version —
Optional numeric values for tokens are allowed, for compatability with other tools. However, the values are ignored.

Terminating <...> fixes the following NonTerminal set <...> —
The second placeholder is a list of the non-terminating symbols that are fixed by creating a terminating production for the NonTerminal in the first placeholder position. This is a useful diagnostic in cases where a single missing production triggers a whole cascade of non-termination of dependent NonTerminals.

The following <...> symbols form a non-terminating cycle <...> —
The second placeholder is a list of the non-terminating symbols in the dependency cycle.

3.3 Non-Terminating Grammars

A grammar is non-terminating if it has one or more non-terminating symbols. This may occur for a number of reasons. Some of these are simple typographical errors in the input grammar. Figure 5 is a typical example. This specification has two errors

Figure 5: Grammar With Errors

```
%token  blip skip
%%
Goal    : ListOpt | skip ;
ListOpt : Element ListOpt ;
Element : Blah ;
Blah    : '(' Element ')' |  Blip ;
```

in it. The terminal symbol "blip" is mis-spelled on the final line, and the *ListOpt* non-terminal seems from its name to be intended to be an optional grammatical element, but has no nullable production. When run through *gppg* the following diagnostic is produced –

```
There are 4 non-terminating NonTerminal Symbols {ListOpt, Element, Blah, Blip}
The following 2 symbols form a non-terminating cycle {Blah, Element}
Terminating Blah fixes the following size-2 NonTerminal set {Element, Blah}
Terminating Element fixes the following size-2 NonTerminal set {Element, Blah}
Terminating Blip fixes the following size-3 NonTerminal set {Element, Blah, Blip}
FATAL: NonTerminal symbol "Blip" has no productions
```

gppg analyses the dependencies between the non-terminating symbols, and looks for leaf symbols in the dependency graph. It reports any instances where modifying the grammar to terminate a single symbol would fix multiple symbols.

In this example the diagnostics show that there is a circular dependency with the symbols *Element* and *Blah*. Making either of these terminating will fix the other symbol as well. However, the diagnostic also shows that "*Blip*" has no productions, and further, if fixed would fix *Element* and *Blah* as well. Fixing symbols with no productions is always the first step in cases like this.

After the final production is changed to —

```
Blah    : '(' Element ')' |  blip ;
```

the *gppg* diagnostic then reads —

```
There are 1 non-terminating NonTerminal Symbols {ListOpt}
Terminating ListOpt fixes the following size-1 NonTerminal set {ListOpt}
Unexpected Error: Non-terminating grammar
```

Now *ListOpt* alone is non-terminating, and changing the productions of the other symbols will not help. It is not difficult to see that a symbol with one production cannot recursively depend on itself. If the apparently intended null production is added to the symbol —

```
ListOpt : /* empty */ | Element ListOpt ;
```

Then the grammar is well-formed and a parser is created.

3.4 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`".

3.4.1 Reduce/Reduce Conflicts

If a reduce/reduce conflict is detected, the conflicts file will contain information similar to that in figure 6. In this example there are two productions both of which can be 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.

Figure 6: Reduce/Reduce Conflict Information

```
Reduce/Reduce conflict on symbol "error",
                     parser will reduce production 51
  Reduce 51: TheRules -> RuleList
  Reduce 64: ListInit -> /* empty */
```

Figure 7: Shift/Reduce Conflict Information

```
Shift/Reduce conflict on symbol "rCond",
                                    parser will shift
  Reduce 29: NameList -> error
  Shift "rCond":  State-87 -> State-88
    Items for From-state for State 87
      67 StartCondition: lCond error . rCond
      29 NameList: error .
              -lookahead: [ rCond, ]
    Items for Next-state for State 88
      67 StartCondition: lCond error rCond .
              -lookahead: [ pattern, ]
```

3.4.2 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 7. In this example, with a current symbol of "`rCond`", the reduce action is to accept production 29. The alternative, shift action is to shift the token and move from state 87 to state 88. The current state, 87, has two "items" in its kernel set. The first item is production 67, 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 29, with the dot at the end. Since the dot is at the end, the action for this item is to reduce production 29. 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 conflicts file also shows the lookahead symbols that can validly appear at that point.

3.5 Conflict Diagnostics

It is sometimes quite difficult to discover the underlying reason for a conflict in a grammar. Sometimes it may be necessary to trace the path by which the automaton entered the state with the conflict in order to understand how the conflict is caused.

A */report* option *gppg* gives additional diagnostic information so as to make this task a little easier. In this case *gppg* produces a file named *basename*`.report.html`. This file is hyperlinked to assist in navigation around the sometimes large data set.

3.5.1 The Report Option

The */report* option generates a file with a formatted version of the productions, together with information about each state in the *LALR*(1) automaton.

The information provided for each state of the automaton is —

* All the "kernel items" for that state. This is a list of all of the productions that lead to that state, with a dot '.' indicating the position in the production that the pattern is matched up to.
* For each completed kernel item (that is, for all items where the dot is at the right-hand end) the list of lookahead tokens that predicate reduction by that production.
* The parser actions. This is a list of tokens and the associated actions that the parser will take. The actions may be "*shift token and go to state* `N`", or "*reduce using* `rule M`". In each case the output is hyperlinked to the destination state or production.
* Non-terminal transitions. This is a list of state transitions to be taken when a reduction recognizes a non-terminal symbol starting from the current state. The reduction may start from the current state or from a successor state.

Figure 8 shows the information generated by the option, for state 4 of the automaton for the fixed version of the tiny grammer in Figure 5. The state information shows

Figure 8: State information with */report* option

```
State 4
  Kernel Items
    5 ListOpt:  Element .  ListOpt

  Parser Actions
    '('       shift, and go to state 7
    blip      shift, and go to state 10
    EOF       reduce using rule 4 (Erasing ListOpt)

  Transitions
    ListOpt   go to state 5
    Element   go to state 4
    Blah      go to state 6
```

that this state has a single item. There are two shift actions and one reduce action. The report draws attention to the fact that the reduction in this case is an *erasure*, that is, a reduction that derives the empty string.

There are three non-terminal transitions from the state.

When trying to understand the origins of a parser conflict it is sometimes helpful to know two things about the conflicted state: the path through the states of the automaton by which the conflicted state has been reached, and a typical prefix that spells out that path. This is additional information that is provided by the */report* option if */verbose* is also specified.

Of course, there may be more than one path leading to any particular state, and there may be many prefixes that spell out the path. *gppg* computes an example of a shortest path that leads to the state, and a shortest prefix.

For our example state, the information is shown in Figure 9. In this state the

Figure 9: State information with */report* **and** */verbose* options

```
State 4
  Shortest prefix: Element
  State path: 0->3
  Predecessor states: 0

  Kernel Items
    5 ListOpt:  Element .  ListOpt

  Parser Actions
    '('       shift, and go to state 7
    blip      shift, and go to state 10
    EOF       reduce using rule 4 (Erasing ListOpt)

  Transitions
    ListOpt   go to state 5
    Element   go to state 4
    Blah      go to state 6
```

shortest prefix is the non-terminal symbol *Element*. The state path is only of length 1. State 0 is the start state. Each state on the state path is hyperlinked so that a browser can navigate to each of the states to gather more information. Each state also has a list of possible predecessor states to allow the user to navigate backward through the state trace.

3.5.2 Tracing the Shift-Reduce Actions

There is a facility in *gppg* which allows a tracing parser to be constructed. A tracing parser uses a specially compiled version of ShiftReduceParser to announce, step-by-step, the actions that the parser is taking. Such a parser should never be used in production since it generates a very large volume of output that is sent to the standard error stream. However, it is a useful way of following the action of a parser on experimental input, or to understand the behavior of some new production rule. The tracing output is always used in conjunction with the html report generated by the */report* option.

In order to construct a tracing parser the ShiftReduceParserCode.cs file must be compiled with the symbol *TRACE_ACTIONS* defined. In the case where shift-reduce parser is embedded in the same assembly as the parser the enclosing assembly should be compiled with the symbol defined.

For the Integer Calculator example (Section 7.1), Figure 10 is the output to standard error when the input is “3 + 5”.

Figure 10: Trace output from integer calculator parsing “3+5\n”

```
 Entering state 0
 State stack is now: 0
 Reducing stack by rule 2, /* empty */ -> list
 Entering state 1
 State stack is now: 0 1
 Reading: Next token is DIGIT
 Shifting token DIGIT, Entering state 30
 State stack is now: 0 1 30
 Reducing stack by rule 18, DIGIT -> number
 Entering state 28
 State stack is now: 0 1 28
 Reading: Next token is '+'
 Reducing stack by rule 17, number -> expr
 Entering state 7
 State stack is now: 0 1 7
 Next token is still '+'
 Shifting token '+', Entering state 14
 State stack is now: 0 1 7 14
 Reading: Next token is DIGIT
 Shifting token DIGIT, Entering state 30
 State stack is now: 0 1 7 14 30
 Reducing stack by rule 18, DIGIT -> number
 Entering state 28
 State stack is now: 0 1 7 14 28
 Reading: Next token is '\n'
 Reducing stack by rule 17, number -> expr
 Entering state 15
 State stack is now: 0 1 7 14 15
 Next token is still '\n'
 Reducing stack by rule 11, expr '+' expr -> expr
 Entering state 7
 State stack is now: 0 1 7
 Next token is still '\n'
 Reducing stack by rule 5, expr -> stat
 Entering state 3
 State stack is now: 0 1 3
 Next token is still '\n'
 Shifting token '\n', Entering state 4
 State stack is now: 0 1 3 4
 Reducing stack by rule 3, list stat '\n' -> list
 Entering state 1
 State stack is now: 0 1
 Reading: Next token is EOF
 Shifting token EOF, Entering state 2
 State stack is now: 0 1 2
 Reducing stack by rule 1, list EOF -> $accept
State stack is now: 0
```

4 Error Handling in *GPPG* Parsers

4.1 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 2). 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*().

4.2 Overriding the Default Error Handling

As noted, the parser will call *yyerror* in case of errors. If the scanner overrides the empty implementation in *AbstractScanner* 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.

Error reporting based around an error handler object 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.

5 Advanced Topics

5.1 Runtime Shift-Reduce Engine

All *gppg*-generated parsers use precisely the same generic code to implement the shift-reduce parsing algorithm. The concrete parser class that *gppg* generates supplies type arguments to the generic object constructor, and also allocates and initializes the parsing tables.

There are two ways that a parser may access this invariant code. The shift-reduce engine is distributed as the strongly named component "*QUT.ShiftReduceParser.dll*", or the source file "*ShiftReduceParserCode.cs*" may be added to the project that hosts the parser.

5.1.1 Using the Runtime *DLL*

"*QUT.ShiftReduceParser.dll*" defines the following public types – the *AbstractScanner* class shown in figure 2, the *IMerge* interface of figure 3, the *LexLocation* type of figure 4 and the main *ShiftReduceParser* class.

The only public member of the main class is the no-arg *Parse* method, but the component exposes a number of protected methods that the generated parser class uses to initialize the parsing tables. There are a couple of other classes that are exposed to the outside, so that the semantic actions of the parser may access the interior of the push-down stacks using the "`$`n" and "`@`n" forms.

The runtime component is strongly named, and is both signed and versioned. This allows the assembly to be placed in the *.NET* fusion cache, avoiding a potential versioning problem. Applications built using a reference to this assembly will only link to precisely this version.

Since the source of the component is distributed with *gppg*, users who need to modify the behavior of the component can do so, but will have to leave the assembly unsigned. In order to create a version of the component with public types the code must be compiled with the *EXPORT_GPPG* conditional compilation symbol defined. Without the symbol, the types are private.

5.1.2 Using the Source Code File

Applications that wish to have the simplest possible deployment strategy may choose to incorporate the code of the shift-reduce engine into the main application assembly.

All of the code of the runtime engine is distributed in the single source file "*ShiftReduceParserCode.cs*". If this code is included in a project the classes of the engine will, by default, have `internal` accessibility.

Here are two distinct, but sensible, use scenarios —

* The parser and the shift-reduce engine code are placed in the main application assembly. The parser specification should declare "`%visibility internal`", so neither the parser nor the runtime engine are publicly visible.

* The parser and the shift-reduce engine code are placed in the same assembly, but are separate from the host application assembly. If the parser specification declares "`%visibility public`", then the base class must be public also. This requires compilation with the *EXPORT_GPPG* symbol defined. Alternatively, both classes have internal visibility, with the application accessing the parser object via a public wrapper class.

5.2 Applications with Multiple Parsers

Applications that use multiple parsers have an added dimension of choice in their configuration. The most obvious configurations might be —

* Each parser in a separate assembly, possibly shared with the associated scanner.
* Multiple parsers in the same assembly, but separate from the assembly of the host application.
* Multiple parsers in the same assembly, shared with the host application.

Parser specifications define the enclosing namespace of the parser, and are able to override the default names of the parser class, the token enumeration, the scanner base class, the semantic value type and the location type. The default public visibility of the "exported" types may also be changed.

5.2.1 Parsers in Separate Assemblies

If each parser is placed in a separate assembly, each with its own scanner, then the design is relatively unconstrained.

For example, each of the assemblies could declare a different namespace, with all of the scanner and parser code encapsulated inside. The visibility and all type names could be left at the default values, and no name-clashes would occur. Each of the assemblies might access the same shift-reduce library assembly, avoiding code duplication.

However, a better design would remove unnecessary visibility of the internal parser and scanner types. In this case, the visibility of all of the *gppg*-defined types should be restricted to `internal`. Each assembly would need a handwritten, public wrapper so the application could initialize and call the parser, and receive results in return. With this configuration the default type-names could be retained, and the namespace could even be shared across the multiple assemblies (although code clarity might be aided by judicious renaming of types).

5.2.2 All Parsers in a Shared Assembly

If all of the parsers and their associated scanners are to be placed in the same assembly, then there are two ways of avoiding name-clashes.

If all of the scanners and parsers reside in the same namespace, then the distinct visible types must uniquely named. For the parsers, the following must have distinct names: the parser class, the scanner base class and the token enumeration. The semantic value types and location types must be uniquely named, unless they are indeed the same type. For the scanners, the token and scanner base types will be distinctly named by the parsers, and the scanner classes must be distinctly named. With this configuration the scanners must *not* embed their buffer code, but must share a single, invariant copy.

If the scanners and parsers reside in separate namespaces then the default naming of the types will not cause any problem, although the types might still be differently named in the interests of code comprehensibility.

With a shared assembly it is possible to embed the *QUT.Gppg* namespace and share the source code of the shift-reduce parser thus avoiding code duplication. Of course, using the shift-reduce *DLL* also avoids code duplication.

5.2.3 Single-Assembly Applications

If multiple parsers and scanners share a single assembly with their host application, then all the considerations of the previous section apply. However, in this case all the parser and scanner types should have internal visibility only.

If it is a design goal to deploy the application as a single assembly and minimize the memory footprint, then the shift-reduce parser source code should be included, with its default internal visibility.

5.3 Multiple Parser Instances

Some applications require multiple instances of the same parser, perhaps running on separate threads. *gppg*-generated parsers are suitable for such applications as the parsers are thread-safe. All parser state is held within its own instance object.

There is no facility to reset the parser state, since this would require some kind of facility to reset the user-defined error-handler and scanner objects at the same time. If an application needs to reset a parser it should create a new parser instance.

From version 1.4.0, the creation of parser instances is very much faster, as only a single parsing table is constructed, and is shared between all instances.

5.3.1 Sharing Parser Tables

When the parser class is first loaded, a single set of tables is constructed and stored in a static variable of the parser class. Whenever *ShiftReduceParser.Parse* is called a reference to the static tables is copied into the *ShiftReduceParser* base-class object.

Note that the tables must be held in a static variable of the *Parser* class. Every instance of a particular parser class requires an identical parsing table. However *different* parser classes, with different tables, may share the same base class if they instantiate the same type parameters for the underlying generic *ShiftReduceParser* class. Thus each parser instance must pass a copy of the shared table to its own base-class instance.

5.4 Parsers Sharing a Common Token Definition

It is sometimes necessary to construct a ”satellite parser” that uses the token stream from another parser. This occurs, for example, in cases where it is necessary to simulate a backtracking parsing machine.

Typically the sub-parser will read and enqueue tokens from the shared parser. After the sub-parser returns, signalling success or failure, the main parser resumes, consuming the enqueued tokens.

In such cases it is necessary to ensure that the two parsers use exactly the same encoding for the token type. The `%sharetokens` and `%importtokens` commands allow this to happen.

There are a few issues to care about when using this facility. Firstly the users of the shared token dictionary cannot add further values to the enumeration, since that would potentially disturb the enumeration order. It *is* possible to add extra character literal symbols, but only if they have ordinal numbering **not greater** than the maximum character literal in the exporting definition. *gppg* enforces these restrictions.

6 Notes

6.1 Copyright

Gardens Point Parser Generator (*gppg*) is copyright © 2005–2010, Wayne Kelly, Queensland University of Technology. See the accompanying file "`GPPGcopyright.rtf`".

6.2 Bug Reports

Gardens Point Parser Generator (*gppg*) is currently being maintained and extended by John Gough. The best way to report bugs or make feature requests is to use the issues tab on the *gppg* page on *CodePlex*. John also occasionally writes about *gppg* and *gplex*-issues on his blog — `http://softwareautomata.blogspot.com`

7 Examples

The distribution contains three simple, related examples. First is a simple integer calculator, the second calculates real numbers and illustrates several additional grammar features. A final example demonstrates how to build an Abstract Syntax Tree (*AST*) using an abstract class as the semantic value type of the parser.

It should be noted that both *gppg* and *gplex* use *gppg*-generated parsers, so these applications provide two additional, complex examples.

7.1 Integer Calculator

The file *Calc.y* contains the specification for a simple integer calculator. The calculator can run with a file as input or, if run without arguments, reads standard input.

The specification contains a simple scanner method *yylex* in the user code section. Notice that the parser detects the first digit of a number and sets the number base to octal if the first digit is zero. There is a predefined array of 26 integers, which are used to store the values for variables named by a single alphabetic character. When there is a used occurence of a variable name in an expression the value is retrieved by indexing into the array.

The specification is very simple, and uses the default semantic value type, integer. The default is sufficient to hold character values as well as the result of intermediate computations when expressions are evaluated. The second example uses a richer structure. Note the use of the synthetic token "`UMINUS`" so that the '–' operator may have a different precedence when used as a unary operator.

7.1.1 Running the Program

The parser is generated by the command —

```
D:\work> gppg /nolines calc.y > calc.cs
```

In this and subsequent examples, user input is set in a bold, slanted, mono-spaced font. Program generated output is shown in plain typewriter font.

There are no errors or warnings and the generated parser, *calc.cs* may be compiled with the command line compiler using the command —

```
D:\work> csc /r:QUT.ShiftReduceParser.dll calc.cs
```

The parser references the base classes in the runtime component *ShiftReduceParser*. For the above command this is presumed to be in the working directory.

Alternatively, a single-assembly version of the application may be built using the *ShiftReduceParser* source code, using the command —

```
D:\work> csc ShiftReduceParserCode.cs calc.cs
```

The application may be run from the command line. Here is a typical input session —

```
D:\work> Calc
c = 34
s = 13
26 * c / s
68
s = 013
26 * c / s
80
^C
```

Notice that the second value that is assigned to the variable *s* has been interpreted as octal, because it starts with a zero digit.

The program continues to evaluate expressions until it is forcibly terminated by an input of "`^C`".

7.2 Real Number Calculator

The real number calculator is based on the integer version, but illustrates the use of a more complicated semantic value type. The source file for this example is included in the distribution as *RealCalc.y*

As with the first example, there is a 26-long array that stores the values of alphabetically named variables. In this case the values are real numbers stored as floating point `double` data. The semantic values of expressions are also floating point values. Nevertheless, *yylex* still passes its semantic values to the parser character by character. The file *RealCalc.y* declares the semantic value type using the "`%union`" construct, as seen in Figure 11. As described in Section 2.6.3, this semantic value type will be

Figure 11: Start of *RealCalc* specification

```
%union { public double dVal;
         public char cVal;
         public int iVal; }

%token <iVal> LETTER
%token <cVal> DIGIT

%type <dVal> expr
...
```

implemented by *gppg* as a *C#* struct.

The figure also illustrates the use of the "`%type`" keyword so that the semantic actions do not have to explicitly select the appropriate field of the struct. We also illustrate the use of the optional *Kind* construct in the "`%token`" declaration to declare

that *DIGIT* token has a `char` semantic value returned in the *cVal* member, *LETTER* token has an `int` semantic value returned in the *iVal* member.

The semantic action for the first production of the symbol *number* is called when the first digit of a number is recognized. Figure 12 shows the relevant production rules. The action creates a new string-builder object and appends the first digit. Each

Figure 12: Extract from *RealCalc* semantic actions

```
number :  digit {
              buffer = new StringBuilder();
              buffer.Append($1);
          }
       | number digit {
              buffer.Append($2);
          }
       | number '.'  digit {
              buffer.Append('.');
              buffer.Append($3);
          }
       ;

expr   : ...    // Other productions for expr
       | number
            {
                try {
                    $$ = double.Parse(buffer.ToString());
                } catch (FormatException) {
                    Lexer.yyerror(
                        "Illegal number \"{0}\"", buffer);
                }
            }
       ;
```

subsequent digit is appended to the buffer, as are any decimal points that are discovered along the way. The scanner does not try to check on the legality of any input numbers, although that would be simple enough to do with a *gplex*-generated scanner. Instead, the semantic action attached to the completion of number recognition takes the string from the string-builder and submits it to the *System.Double.Parse* method. In the event that an illegal number is entered as input, *Parse* throws an exception which is caught by its caller and converted into a call of *yyerror*.

7.2.1 Running the Program

The parser is generated by the command —

```
D:\work> gppg /nolines RealCalc.y > RealCalc.cs
```

As before, user input is set in a bold, slanted, mono-spaced font. Program generated output is shown in plain typewriter font.

There are no errors or warnings and the generated parser, *RealCalc.cs* may be compiled with the command line compiler using the command —

```
D:\work> csc /r:QUT.ShiftReduceParser.dll RealCalc.cs
```

Alternatively, the application may embed the shift-reduce library code —

```
D:\work> csc ShiftReduceParserCode.cs RealCalc.cs
```

The application may be run from the command line. Here is a typical input session —

```
D:\work> RealCalc
RealCalc expression evaluator, type ^C to exit
c = 34 ; s = 13
26.2 * c / s
68.5230769230769
s = 13.0.0
Illegal number "13.0.0"
^C
```

7.3 Tree-Building Calculator

The Tree-Building Calculator example is yet another variant on the calculator theme. The source code of the example is also available for download from the *gppg* page on CodePlex.

The purpose of this example is to demonstrate how to use a reference type as the semantic value type. This is the preferred method of using a *gppg* parser to build an Abstract Syntax Tree (*AST*).

The idea is that the caluclator will use its parser to build a tree-representation of the input expressions. In this case, the 26 variables of the calculator hold expression trees, which may include simple literal values and references to other variables.

The data types of the *AST* all derive from an abstract class named *Node*. An outline of the classes is shown in Figure 13. There are three subtypes. These are used to

Figure 13: Skeleton of Node Definitions

```
internal enum NodeTag {error, name, literal, plus,
                           minus, mul, div, rem, negate }

internal abstract class Node {
   readonly NodeTag tag;
   public NodeTag Tag {get {return this.tag; }}

   protected Node(NodeTag tag ) {this.tag = tag; }
   public abstract double Eval( Parser p );
   public abstract string Unparse();

}

internal class Leaf :  Node {...}

internal class Unary :  Node {...}

internal class Binary :  Node {...}
```

represent leaf nodes of the tree, unary nodes and binary nodes. Every node instance carries a tag, which distinguishes between nodes of the same arity that represent different expression operators.

The *Node* base-type declares two abstract methods. *Eval* evaluates the tree rooted at that particular node, returning a *double* value. *Unparse* generates a fully parenthesized textual representation of the tree at a particular node. The two methods behave differently when the traversal reaches a leaf node that is a reference to a named variable. In the *Unparse* case the name of the variable is returned. In the *Eval* case the traversal recursively evaluates the tree held by the named variable[9].

The grammar specification is shown in Figure 14. Compared to the earlier examples the input language is enhanced by the addition of several imperative commands. These are —

* **eval** — Evaluate the given expression.
* **exit** — Exit the program.
* **help** — Print the usage message.
* **print** — Display the variable expression trees.
* **reset** — Set all variables to the empty tree.

Essentially all of the work of the semantic actions is performed in helper functions. These reside in the hand-written part of the partial *Parser* class, in file "`RealTree-Helper.cs`".

The various *Make** methods are passed the root node(s) of the sub-tree(s), together with a tag value. The default action in the case of leaf expressions is to copy the reference contained in the scanner's *yylval* field. Thus the scanner is responsible for allocating an appropriate leaf node for a literal numeric value, or for a variable reference.

The result of a tree evaluation is a floating point double value. Evaluations are invoked in two different places in the semantic actions. The *command* "`eval` *expr*" evaluates the expression and then displays the result to the user.

On the other hand, the *expression* "`eval(` *expr*`)`" performs an immediate evaluation during tree-building. Thus the command sequence —

```
> a = 34 ; b = 13
> c = a + b
> d = eval(a + b)
> print
register 1 = a = 34
register 2 = b = 13
register 3 = c = (a + b)
register 4 = d = 47
```

demonstrates that the assignment to variable 'c' assigns the expression tree value ***a + b***, while the assignment to variable 'd' assigns the result of *evaluating* the current value of ***a + b***.

7.3.1 Running the Program

The parser is generated by the command —

```
> gppg /nolines RealTree.y
```

As before, user input is set in a bold, slanted, mono-spaced font. Program generated output is shown in plain typewriter font.

There are no errors or warnings and the generated parser, *RealTreeParser.cs* may be compiled with the command line compiler using the command —

[9] Of course, a test for circularity is required to prevent stack overflows!

Figure 14: Grammar for RealTree Example

```
%namespace RealTree
%output=RealTreeParser.cs
%partial
%start list
%visibility internal
%YYSTYPE RealTree.Node

%token LITERAL LETTER PRINT EVAL RESET EXIT HELP EOL

%left '+' '-'
%left '*' '/' '%'
%left UMINUS

%%

list
    :   /*empty */
    |   list stat EOL
    |   list error EOL  { yyerrok(); }
    ;

stat
    : /* empty */
    | HELP              { this.PrintHelp(); }
    | RESET             { this.ClearRegisters(); }
    | PRINT             { this.PrintRegisters(); }
    | EXIT              { this.CallExit(); }
    | EVAL expr         { this.Display($2); }
    | LETTER '=' expr   { this.AssignExpression($1, $3); }
    ;

expr
    : '(' expr ')'      { $$ = $2; }
    |  EVAL'(' expr ')' { $$ = MakeConstLeaf(Eval($3)); }
    |  expr '*' expr    { $$ = MakeBinary(NodeTag.mul,$1,$3);}
    |  expr '/' expr    { $$ = MakeBinary(NodeTag.div,$1,$3);}
    |  expr '%' expr    { $$ = MakeBinary(NodeTag.rem,$1,$3);}
    |  expr '+' expr    { $$ = MakeBinary(NodeTag.plus,$1,$3);}
    |  expr '-' expr    { $$ = MakeBinary(NodeTag.minus,$1,$3);}
    |  LETTER           // $$ is automatically lexer.yylval
    |  LITERAL          // $$ is automatically lexer.yylval
    |  '-' expr %prec UMINUS  {
                          $$ = MakeUnary(NodeTag.negate,$2);
                        }
    ;
%%
```

```
> csc /out:RealTree.exe RealTreeParser.cs RealTreeHelper.cs
ShiftReduceParserCode.cs
```

The application may be run from the command line.

```
> RealTree
RealCalc expression evaluator, type help for help
>
```

Here is a typical input session —

```
> s = 5 ; g = 1; n = g+(s-g*g)/(2*g)
> g = eval(n); eval (1+g)/2
result: 2
```

In this “program” s (s for *square*) is a number that we wish to find the square root of, using Newton’s method; g (g for *guess*) is an initial guess for the root; n (n for *next*) is the next iteration of the guess.

If the second command line is repeated the value g converges quadratically toward the square root of 5 and the result converges toward 1.618 033 988 749 89, the “golden mean”.

8 Appendix A: *GPPG* Special Symbols

8.1 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.
`%importtokens`	*gppg* will deserialize the terminal symbol dictionary from the nominated data file. The specification file cannot define any additional symbolic tokens. See Section 5.4.
`%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 2.3.
`%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. See section 2.7.2 for further detail.
`%right`	this marker declares that the following token or tokens will have *right associativity*, that is, $a \bullet b \bullet c$ is interpreted as $a \bullet (b \bullet c)$.
`%scanbasetype`	this marker defines the name of the scanner base class, overriding the *ScanBase* default. The argument must be a valid *C#* identifier. (Only applies to the */gplex* option.)
`%sharetokens`	the terminal symbol dictionary will be serialized to the file "*basename*.dat" so that the same tokens may be imported into another parser. See Section 5.4.
`%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.

Table continues on next page...

Keyword Commands Continued ...

Keyword	Meaning
`%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 set the value of the nominated member.
`%union`	marks the start of a semantic value-type declaration. See section 2.6.3 for more detail.
`%using`	this marker adds the given namespace to the parser's using list. The argument is a dotted name, in general.
`%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*.

8.2 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 2.

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.

9 Appendix B: Shift-Reduce Parsing Refresher

9.1 Some Definitions

A *grammar* is a set of rules that, for a particular *symbol alphabet*, determines which ordered sequences of symbols form valid *sentences* in the language that the grammar specifies.

Parsers are automata that recognize the valid sentences of a given grammar. Given an arbitrary symbol sequence as input, a parser must answer yes or no according to the legality or otherwise of that sequence.

Phrase-structured grammars declare a set of syntactic rules that recognize fragments of a sentence as belonging to particular *syntactic categories*. In phrase structured grammars for natural languges the syntactic categories correspond to concepts such as "noun", "verb" and "adjectival clause of reason". In a programming language syntactic categories might be "assignment statement", "variable declaration" and so forth.

Among all classes of phrase-structured grammars the most useful for formal languages are the *context-free grammars*. The syntactic rules of a context free grammar have the special form N→*S*, where N is a syntactic category, and *S* is a sequence of zero or more symbols each of which may denote either an alphabet symbol or a syntactic category. For this example rule we say "N *derives S*" or equivalently "*S* may be *reduced* to N". We call such rules *productions*. Such derivation rules are context-free because *any* occurrence of an N may be expanded to the sequence *S*, without regard for the *context*, that is, without regard for what comes before or after the N from which the *S* is being derived[10].

There is a special member of the set of syntactic categories that we designate as the *goal symbol* G. We may generate all the sentences of a language by starting with the goal symbol and progressively replacing any syntactic categories with the right-hand-side of a corresponding production rule. When we have a sequence of only alphabet symbols, that sequence is a sentence in the language. We say the particular choice of substitution steps has *derived* the sentence in the language. Each intermediate state of the sequence as the substitution steps are carried out is called a *sentential form*. Until the final substitution is made the sequence is not a sentence, but at all stages it has the *form* of a sentence subject only to the expansion of any remaining syntactic placeholders.

Because of this concept of a derivation terminating when there are only alphabet symbols remaining we call the alphabet symbols *terminal symbols* of the grammar. Conversely, the syntactic categories are *non-terminal symbols*.

There are two parsing strategies for context-free grammars. One is to start with the goal symbol and to try to find a sequence of derivation steps that matches up with the input sequence. Parsers that work this way are called *top-down parsers*. "Recursive descent" is the most widely used top-down parsing technique.

The second parsing strategy, *bottom-up parsing*, attempts to match fragments of the input sequence with the right-hand-sides of production rules. When a fragment is matched it is replaced by the single, non-terminal symbol of the rule's left-hand side. This substitution step is called *reduction*.

Shift-reduce parsers, as generated by *gppg*, implement the matching of production right-hand sides with the help of a "push-down" stack structure. The input is read

[10]This is not quite the same as saying that any *S* may be *reduced* to an N. There may be other non-terminal symbols to which the same *S* may be reduced, and the legality of any particular reduction *may* depend on the context.

symbol by symbol, and at each step a decision is made whether to push the (terminal) symbol on the stack, or to recognize the top n symbols on the stack as matching the right-hand-side of a particular production. If a symbol is pushed on the stack, we have performed a *shift*. If a right-hand-side of length n is recognized we *reduce* by popping the topmost n symbols from the stack and push the corresponding left-hand-side non-terminal symbol. If this process leads to the stack holding just the goal symbol a sentence has been recognized.

It is one of the classic results of computer science that the set of sequences of symbols that indicate the applicability of each reduction rule belong to a *regular language*. That is: the *viable prefixes* of each reduction may be recognized by a finite state machine (*FSA*).

The specification for a *gppg* parser declares the production rules of the grammar. Specifications usually also declare computations that are performed step by step as each pattern is recognized and the reduction performed. These *semantic actions* compute attributes of the production left-hand-side symbol in terms of the attribute values of the production right-hand-side symbols.

A *gppg*-generated parser does not actually maintain a stack of terminal and non-terminal symbols. Instead is has three stacks to guide its operation. The first stack is a stack of *parser states*, which are integer values. These values are the states of the *FSA* that decides whether to shift or reduce, and if reducing, by which production. The second stack is the stack of attribute values for the values that would be on the symbol stack, if there was one. The third stack is a stack of text spans corresponding to the symbols that would be on the symbol stack, if there was one. This last stack is useful for semantic actions that require the retrieval of position information or text strings.

9.2 How Shift-Reduce Parsing Works

At each step in the parsing process a shift-reduce parser examines the state on the top of the state stack, and the identity of the next input symbol. The next input symbol, fetched from the associated *lexical scanner*, is called the *lookahead symbol*.

Each state of the *FSA* corresponds to a particular finite set of partially recognized production right-hand-sides. These partially recognized rules are called *production items* of the grammar. Production items are just right-hand-sides annotated with a "dot" to show how many of the symbols of the rule have been seen so far. The */report* option of *gppg* generates an html file that shows all of the items for each state of the automaton. Many state have a single kernel item, but others may have several. For an example of kernel items see the discussion in section 2.7.2.

The runtime representation of each parser state stores a list of actions to be taken in that state for each possible lookahead symbol. There is also a list showing the next state value after pushing all the possible terminal or non-terminal symbols.

For any particular combination of top-of-stack state and lookahead symbol the parsing engine may decide to shift, reduce or signal an error.

Shift Action

If a shift is chosen the *FSA* makes a transition from the current state to a new state. The new state is pushed on the state stack. If the scanner has placed any symbol attribute information in the *yylval* field this is pushed onto the semantic value stack. Finally, if the scanner has placed location information in the *yylloc* field this is pushed onto the location stack.

Reduce Action

If the parser chooses a reduction by a particular production, then the following steps are taken. First, if the production specifies any semantic action then this computation is performed. If the chosen production has n symbols in its right-hand-side, then the semantic action may use the topmost n values of the semantic value and location stacks.

Following the execution of the semantic action, the topmost n elements are popped from all three stacks. The left-hand-side of the chosen reduction must now be "pushed" onto the stack. The preceeding popping of the stack will have exposed a new top-of-stack state, and the information of that state detemines what the new state will be following the push of the production left-hand-side non-terminal symbol. At the same time any semantic value and location information computed by the semantic action are pushed onto the semantic value and location stacks.

Error Action

If there is no possible reduction or shift action for a given state, lookahead symbol combination then the *yyerror* method is called.

Loop Forever

After a shift or reduce action, if the new top of stack state corresponds to the goal symbol, and the lookahead symbol is "end of input" the parser returns true. Otherwise, if the lookahead symbol has been consumed by a shift a new input symbol is fetched.

The parser continues by making a new shift-reduce decision based on the new top-of-state stack state value and the (possibly new) lookahead symbol.

9.3 What Can Go Wrong

Not every context-free grammar can be successfully recognized by a *gppg* parser. This final section considers some of the things that can prevent successful generation of a parser.

First, it must be said that some grammars cannot be recognized by *any* parser that does not back-track. Furthermore, there are grammars that define sentences that have multiple derivations, that is, sentences that are ambiguous. Finally, there are grammars which inherently require more than one symbol of lookahead to make correct parsing decisions.

In practice, when a grammar is submitted to *gppg*, or any other shift-reduce parser-generator, the tool may notify the user of various *conflicts*. These arise when the states of the parser have more than one possible action for some state-lookahead combination. There are two kind of conflict: a *shift-reduce conflict* is notified if a particular state has both a possible shift action *and* a possible reduction. The occurrence of a shift-reduce conflict is not fatal, and by default *gppg* parsers always shift in preference to reducing. Shift-reduce conflicts are notified to the user so that it may be checked that the default, shift action does lead to a correct result.

The second kind of conflict is a *reduce-reduce conflict*. Such a conflict arises if there are two or more production items in the state that signify a possible reduction action. Reduce-reduce conflicts tend to be more serious, and it is often necessary to modify a grammar to remove such conflicts to obtain correct behavior.

gppg has advanced facilities for conflict reporting, and special help for diagnosing the cause of such conflicts. These are aimed at helping the user to successfully remove such conflicts. See section 3.4, and section 3.5.

There is also a facility to produce parsers that emit a trace of the parser actions for each input. This facility is discussed in section3.5.2.

10 Appendix C: Pushing Back Input Symbols

When the client of a *gplex*-generated scanner is a backtracking parser[11] the scanner may be required to push back symbols from the input stream. Since *gppg* version 1.4.7 and *gplex* version 1.1.8 both tools support pushing back input.

The code consists of three parts —

* *gppg* defines a *ScanObj* class that wraps a token integer value, along with its associated semantic value and position information.
* An *API* for enqueueing and dequeueing lookahead symbols, and manipulating the queueing data structure.
* Prolog code for the *Scan* method, to dequeue pushed-back symbols before any further symbols are read from the input buffer.

Whenever a parser requires to look ahead in the sequence of input symbols it needs a mechanism to return to the sequence position prior to that lookahead. One way to do this would be to save the scanner state prior to the lookahead, and afterward restore that state. However, this operation is complicated by the possibility that the scanner may have changed start state or even input source, and the fact that the input byte stream must be precisely restored in the case of multi-byte character encodings. As well, any creation or processing of semantic value objects will need to be repeated when the input is read a second time.

An alternative is to save the lookahead tokens, and push them back to a suitable data structure within the scanner. Whenever the scanner method *yylex*() is called the scanner will return pushed back symbols before switching back to reading new symbols from the input source.

10.1 The *ScanObj* Class

When tokens are pushed back to the scanner, it is necessary to also preserve any associated semantic and location values. The *ScanObj* class has three fields: the token integer value, the semantic value and the location value. The type of the semantic value is the "`TValue`" type argument of the generic *AbstractScanner*<*TValue*,*TSpan*> class from which all scanners are derived. The type of the location value is the "`TSpan`" type argument of *AbstractScanner*. The definition of this class is created by *gppg* which, of course, knows what the type arguments are in any particular case. For the default type arguments, the definition of *ScanObj* is given in Figure 15.

10.2 Prolog for the Scan Method

The only difference between a *gplex* scanner which allows token pushback and one that does not, is a prolog to the scanner's *Scan* method. The prolog tests if the pushback queue is empty. If so, the normal *Scan* method body is executed, fetching a symbol from the current input source buffer. If the pushback queue is not empty, *Scan* dequeues a *ScanObj*, and sets the return token value, *yylval* and *yylloc*. The code of the prolog is declared in the scanner specification, as shown in Figure 16.

It may be useful to know that in the case of a specification that defines a scan method epilog, typically to construct the location text span, the prolog return does *not*

[11] This appendix also appears in the *gplex* documentation.

Figure 15: Default *ScanObj* Definition

```
internal class ScanObj {
    public int token;
    public int yylval;
    public LexLocation yylloc;

    public ScanObj(int t, int v, QUT.Gppg.LexLocation l) {
        this.token = t; this.yylval = v; this.yylloc = l;
    }
}
```

Figure 16: "lex" Specification with *Scan* prolog

```
// Start of definition section
    Definitions go here.

%% // Start of pattern section

%{
    // Scan prolog: Code to take tokens from non-empty queue
    if (this.pushback.QueueLength > 0) {
        ScanObj obj = this.pushback.DequeueCurrentToken();
        this.yylval = obj.yylval;
        this.yylloc = obj.yylloc;
        return this.token;
    }
%}

Regular expressions define patterns here.

%% // Start of user code section

User code, if any, goes here.
```

traverse the epilog code. Thus the setting of the *yylval*, *yylloc* values in Figure 16 is safe[12].

10.3 The Pushback Queue API

The *API* of the pushback queue is defined by the *PushbackQueue* class. The *API* is shown in Figure 17. The class is a generic class of one type parameter. The class is instantiated for the particular *ScanObj* that is required. The class is declared in the *QUT.Gppg* namespace, and the source is found in the "`LookaheadHelper.cs`" file in the *gppg* distribution.

The class defines two delegate types. One value of each of these types is passed

[12]If the user has defined an epilog the tool-generated body of *Scan* lies within a try block. The finally block of this try is the user-defined epilog. However, the user-defined prolog is *outside* the try block so a return in the prolog does not execute the epilog.

Figure 17: Lookahead Helper API

```
namespace QUT.Gppg {
    internal class PushbackQueue<Obj> {
        public delegate Obj GetObj(); // Fetch ScanObj from host scanner
        public delegate Obj TokWrap(int tok); // Wrap given token value

        public static PushbackQueue<Obj>
            NewPushbackQueue(GetObj getObj, TokWrap tokWrap);
        public int QueueLength { get; } // Number of queued ScanObj
        public Obj EnqueueAndReturnInitialSymbol(int token);
        public Obj GetAndEnqueue(); // Enqueue new ScanObj from scanner
        public void AddPushbackBufferToQueue();
        public void AbandonPushback();
        public Obj DequeueCurrentToken(); // Helper for Scan prolog
    }
}
```

as argument to the factory method *NewPushbackQueue* that creates a pushback queue. This factory method is called just once for each parser instance, during scanner initialization.

The first delegate, *getObj*, invokes *yylex*() on the host scanner, and constructs a *ScanObj* from scanner state. The second delegate, *tokWrap*, takes an integer argument and constructs a *ScanObj* with that token value. Both are necessary, since the pushback queue may begin with an object corresponding to the usual one-symbol lookahead, and then continue with objects arising from fresh invocations of *yylex*().

Figure 18 is a typical initialization of the pushback data structure. The two arguments to *NewPushbackQueue* are anonymous methods, each of which calls the *ScanObj* constructor. These lambda methods encapsulate the necessary reference to the host scanner object.

Figure 18: Typical *PushbackQueue* Initialization

```
using QUT.Gppg;

internal sealed partial class Scanner {
  ... // Other feature declarations
  public PushbackQueue<ScanObj> pushback;

  public void Init( ...  )  {
    ... // Other initializations
    this.pushback = PushbackQueue<ScanObj>.NewPushbackQueue(
      () => new ScanObj(this.yylex(),this.yylval,this.yylloc),
      (int t) => new ScanObj(t, this.yylval, this.yylloc));
  }

  ... // Rest of scanner partial class
}
```

There are two methods in the *API* which return *ScanObj* objects. These are invoked by user-specified semantic actions of the parser that trigger lookahead actions. The first method, *EnqueueAndReturnInitialSymbol*, initializes a new, empty pushback buffer to store all of the symbols read during the lookahead process. The method is passed the current parser *NextToken* value. This value may be zero, indicating that the parsing engine has not yet fetched the lookahead symbol, or may be the token value of the lookahead symbol. The method fetches the next symbol, if necessary, and enqueues the object in the pushback buffer. The second method, *GetAndEnqueue* fetches further input symbols after the initial lookahead symbol, adding them to the pushback buffer.

A lookahead semantic action sequence is terminated by calling the method *AddPushbackBufferToQueue*. Because a lookahead action may be terminated without exhausting the pushed back symbols of a previous lookahead, the newly created pushback buffer is added to the front of the existing queue.

For the special case where a single symbol of lookahead was sufficient it is possible to adopt a slightly more lightweight termination strategy. The scanner action may write the lookahead token to the parser *NextToken* variable, and invoke the method *AbandonPushback*. This avoids all queue manipulation.

10.4 Summary: How to use Symbol Pushback

In order create a *gppg* parser that uses symbol pushback to program around non-*LALR*(1) grammar features the necessary steps are —

* Create a *gplex* scanner that supports symbol pushback.
* Modify the grammar so that the *gppg* parser performs symbol lookahead to resolve conflicts.

10.4.1 Creating a Scanner Supporting Symbol Pushback

All of the steps that are required to make a *gplex* scanner support symbol pushback depend on code in the file *LookaheadHelper.cs*, in the folder *SpecFiles* folder of the *gplex* distribution.

* Include the file *LookaheadHelper.cs* in your project.
* Declare a scanner field *pushback* in the scanner class, and initialize the field at scanner instantiation. (Cut and paste template from helper file.)
* Add the prolog code to the lex file from which the scanner is generated. (Cut and paste template from helper file.)

10.4.2 Modify the Grammar to Perform *ad hoc* Lookahead

At each point in the grammar at which it is necessary to perform an *ad hoc* lookahead choose a reduction that marks arrival of the parse at that particular decision point. This is the *triggering reduction*.

In the following it is assumed that the *PushbackQueue* object is accessible through some parser property named *QueueRef*.

Begin the semantic action that performs the lookahead action. The code will have the following outline —

```
ScanObj next =
        QueueRef.EnqueueAndReturnInitialSymbol(NextToken);
...   // Code that determines the nature of the following symbols. Further symbols
...   // are fetched using the method — QueueRef.GetAndEnqueue()
...   // Signal result to the parser.
QueueRef.AddPushbackBufferToQueue();
```

The code that determines the right context of the triggering reduction varies in complexity. For example, if the ambiguity is resolved by some known lookahead depth, then that number of symbols are read to make the decision. For example, in the *C#* grammar, the language specification Section 7.7.6 implies that the parser must look one symbol past a closing right parenthesis to determine if the construct "`(x)`" is a type-cast, a parenthesized expression, or a single-arg version of an implicit anonymous parameter list.

There are other cases where the length of the lookahead is not bounded, but a decision may be made by simply reading ahead to find some characteristic "sentinel" symbol.

Finally, there are cases where it is necessary to use a context free parser to recognize the right context.

Once a decision has been made on the basis of the lookahead it is necessary to use that information to modify the future trajectory of the parse. One way of doing this while leaving the shift-reduce engine unmodified is to inject a *dummy token* into the input sequence.

A more detailed discussion of these aspects of the subject are part of a continuing series on the blog `http://softwareautomata.blogspot.com`

Index