The GPPG Parser Generator

(*Version 1.3.0 October 2008*)

John Gough, Wayne Kelly QUT

November 11, 2008

New in this release

Compared to the v1.2.1 (August 2008) release this version has the following significant changes and new features —

* The new release has a completely new frontend. The scanner and parser are constructed by *gplex* and *gppg* respectively.
* The documentation has been expanded in an attempt to make it more self-contained. It now has a more complete description of the input syntax, and explains (for example) how production precedence *really* works.
* Error handling is based around an error buffer and a listing generator, and the parser attempts error recovery. Most error messages have changed, and a number of syntactic errors that were previously undetected are now reported.
* Type declarations for *YYSTYPE* and the `%union` declaration now allow arbitrary syntax including (even nested) generic types, and arrays. When both `%union` and *YYSTYPE* are defined *YYSTYPE* must be a simple identifier, naming the "union" type. Otherwise *YYSTYPE* may define be an arbitrary type declaration.
* An extended syntax for token list declarations has been implemented. Every token has the option of declaring a "display form" that is used in all diagnostic messages. All token declaration constructs allow for optional "kind" markers.
* Hexadecimal and unicode escapes are allowed in strings and character literals. Character literals are canonicalized before insertion in the dictionary.
* Literal strings for filenames may use either the verbatim or normal form, and escape characters are interpretted in filenames. However, the `*.y` input file is still an 8-bit byte-file.

1 Overview

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

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

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

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

1.1 Installing *GPPG*

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

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

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

1.2 Running *GPPG*

gppg is invoked by the command —

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

the available options are —

* `/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 2.3.1).
* `/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.
* `/gplex` — makes *gppg* customize its output for the Gardens Point *LEX* (*gplex*) scanner generator.
* `/help` — displays the usage message.
* `/listing` — cause *gppg* to always produce a listing file. Without this option *gppg* produces a listing only if there are errors or warnings.
* `/no-lines` — suppresses emission of output `#line` directives.
* `/report` — generates a file "*basename*.`report.html`" with *LALR(1)* state information.

* `/verbose` — sends more detailed information to the concole, and to the conflict and *LALR* reports.

* `/version` — displays version information.

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

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 2.3). Typically other code will attach a scanner and error handler object to the parser instance. The scanner, in turn, will have been provided with some input text to read from.

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

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

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

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

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

1.5 Scanner Interface

Parser instances contain a public field named *scanner*. The parser expects this field to be assigned a reference to a scanner that implements the class shown in Figure 1. Despite its name, *IScanner* is the abstract base class of the scanners, rather than an interface. The base class provides the *API* required by the runtime component of *gppg*, the library *ShiftReduceParser.dll*. Of course scanners will usually implement other

Figure 1: Scanner Interface of *GPPG*

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

facilities that are required by the scanner semantic actions. These actions will use the richer *API* that the concrete scanner class supports, but the shift-reduce parsing engine itself needs only the subset defined in the base class.

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

The abstract scanner class is a generic class with two type parameters. The first of these, *YYSTYPE* is the "*SemanticValueType*" of the tokens of the scanner. If the grammar specification does not define a semantic value type (see section 2.3.1) then the type defaults to `int`. From version 1.2 of *gppg* the semantic value type can be any *CLR* type. Previous versions required a value-type.

The second generic type parameter, *YYLTYPE*, is the location type that is used to track source locations in the text being parsed. In almost all applications it is sufficient to use the default location type, *LexLocation*, shown in Figure 3. Location-tracking is discussed further in section 2.6

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 *IScanner* class, and several other features. Full details of this and other convenience features of this option are given in the *gplex* documentation.

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

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

2 Input Grammar

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

The rules of the grammar are specified in terms of *terminal symbols*[1] and *non-terminal symbols*. 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

[1]"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.

of single characters. The single character tokens do not need to be declared in the parser specification, but the enumeration names must be declared. In cases where enumeration tokens correspond to fixed strings in the scanner input it is possible use the fixed 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 a small number of checks on the validity of the grammar that it is given. If a particular symbol does not appear in a token declaration, and does not appear as the left-hand-side of at least one production, then the grammar is non-terminating. *gppg* issues a error message naming the symbol that is involved. This is a fatal error, as parser production fails under such circumstances.

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

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

2.1 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.

2.2 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 7.1 —

```
%union       // usual meaning, but see section 2.3.1
%prec        // usual meaning, see section 2.4.1
%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
%namespace     // declares the namespace for the parser
%parsertype    // names the parser class within namespace
%visibility    // declares the visibility of the parser class
%tokentype     // names the token enumeration
%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 2.3.

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. The literal string associates a *display string* with the token. This display string is used in all diagnostic messages from the generated parser. This is particularly helpful so 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`'.

2.2.1 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 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 2.4.1.

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.

2.3 Extensions to the Declaration Grammar

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, escape characters, or even unicode escapes.
* 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 has any escape characters expanded before use. However, *gppg* does not check the legality of the resulting filepath string.

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.

`%namespace` *NameSpaceName*

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

Naming Types

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

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

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

```
%tokentype Tokens
```

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

2.3.1 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> sLst;
         public List<TokenInfo> tLst;
         public TokenInfo info;
         public Production prod;
         public ActionProxy prxy;
       }
```

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*"[2]. For an example of use of the "`%union`" construct see Section 6.2.

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*.

`%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, since in *C#* arrays 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]That is, the type name will be "*MyNamespace*`.ValueType`" which should not be confused with the super type of every value type `System.ValueType`.

`%YYLTYPE` *LocationTypeName*

This marker overrides the default location type name, *LexLocation*. The default type is sufficient for most applications, but when additional functionality is required it is possible to define a new type, and declare its name with this marker. Location tracking is discussed in detail in Section 2.6

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.

`%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.

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

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.

Colorizing Scanners and *Managed Babel*

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

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

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

2.4 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 an token sequence corresponding to that production right-hand-side has been recognized, the semantic action, if there is one, is executed.

All of the productions for a given non-terminal symbol may occur together in the specificatin, 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.

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*[3]. 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.

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 7.2, and their use is discussed in Section 2.5.

2.4.1 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,

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

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 6.1. 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 html files generated 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 2.2.1).

* 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.

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.

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.

2.5 Semantic Actions

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

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

Default Semantic Action

The default semantic action is invoked for every reduction by a production that has no user-supplied semantic action.

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 *YYSTYPE* 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`".

2.6 Location Tracking

The second generic type parameter of the scanner interface in figure 1, *YYLTYPE*, is the location type. Instances of the location type contain information that mark the start and end of the relevant phrase in the input text, that is, the type is a representation of a text span. The actual type that is substituted for the *YYLTYPE* parameter must implement the *IMerge* interface shown in Figure 2. The location type supplies a method that

Figure 2: Location types must implement *IMerge*

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

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

Location Actions

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

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

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

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

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

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

Default Location Type

Parser specifications may declare the name of a type that is to be used as the location type. This type must implement the *IMerge* interface. In the event that no such declaration is made, the default location tracking type is the *LexLocation* type shown in Figure 3. This type implements a simple text-span representation.

Figure 3: Default location-information class

```
public class LexLocation :  IMerge<LexLocation>
{
    public int sLin; // Start line
    public int sCol; // Start column
    public int eLin; // End line
    public int eCol; // End column
    public LexLocation() {};
    public LexLocation(int sl; int sc; int el; int ec)
      { sLin=sl; sCol=sc; eLin=el; eCol=ec; }

    public LexLocation Merge(Lexlocation end) {
      return new LexLocation(sLin,sCol,end.eLin,end.eCol);
    }
}
```

Supplying a Different Location Type

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

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

`%YYLTYPE` *TypeIdent*

where *TypeIdent* is the simple name of the desired type[4]. The type must implement the *IMerge* interface, but may also provide whatever other methods are required.

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

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

Special Behavior for Empty Productions

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

Consider the production $A \rightarrow BCD$. The default location processing action when this production is reduced is to create a location span that begins at the start of the B phrase, and finishes at the end of the D phrase. Now, suppose that B and D are *nullable* symbols, and each has been produced by reduction by an empty production. A moment's consideration will show that the correct behavior is produced if the location span for each empty production *begins* with the start of the lookahead token, and *ends* with the finish of the last token shifted. Such a location value makes no sense on it own, it has negative length for example, but merges correctly with other spans. The 1.0.1 version of *gppg* uses this strategy to deal with location information for empty productions[5].

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

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* 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.

[5]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.

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.

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 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 possible 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

Grammars may be non-terminating for a number of reasons. Some of these are simple typographical errors in the input grammar. Figure 4 is a typical example. This

Figure 4: Grammar With Errors

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

specification has two errors 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`".

Reduce/Reduce Conflicts

If a reduce/reduce conflict is detected, the conflicts file will contain information similar to that in figure 5. In this example there are two productions both of which can be

Figure 5: Reduce/Reduce Conflict Information

```
Reduce/Reduce conflict on symbol "error",
                     parser will reduce production 41
  Reduce 51: TheRules -> RuleList
  Reduce 64: ListInit -> /* empty */
```

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

Shift/Reduce Conflicts

Shift/Reduce conflicts tend to be more common, and are often but not always benign. The conflicts file for a typical case will contain information similar to that in figure 6. 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

Figure 6: 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, ]
```

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.

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

Figure 7: 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 null 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 automaton by which the 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 8. In this state the 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.

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 1). The parser runtime then discards values from the parser state, value and location stacks until a state is found that can shift the synthetic “error” token. After the error token has been shifted the parser checks to see if an ordinary shift or reduce action is then possible given the existing lookahead symbol. If no such action is possible, the parser discards input tokens until an acceptable token is found or the input ends.

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

Figure 8: State information with */report* **and** */verbose* options

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

  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
```

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 *IScanner* then that method may construct a suitable error message. It is useful to note that error recovery is attempted because the next input symbol is not a possible lookahead for either a shift or a reduce action. It is always the case that the input symbol that blocked progress is the symbol corresponding to the scanner's current *yylval* and *yylloc* at the moment that *yyerror* was called.

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

The alternative strategy for constructing error messages is to leave *yyerror* empty, and place explicit calls to an error handler in the semantic actions of productions that mention the error token. Such calls to the error handler will be able to make good use of the automatic location tracking mechanisms of the parser to provide information for the error handler. For example, in the case of a missing member of some kind of paired construct the semantic action should have access to the location information of the current lookahead symbol *and* the symbols whose pair was expected.

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 Notes

5.1 Copyright

Gardens Point Parser Generator (*gppg*) is copyright © 2005–2008, Wayne Kelly, Queensland University of Technology. See the accompanying file “`GPPGcopyright.rtf`”.

5.2 Bug Reports

Gardens Point Parser Generator (*gppg*) is currently being maintained and extended by John Gough. Bug reports and feature requests for *gppg* should be sent to John at “j.gough *at-sign* qut.edu.au”.

6 Examples

The distribution contains two simple, related examples. One is a simple integer calculator, the other calculates real numbers and illustrates several additional grammar features.

6.1 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.

Running the Program

The parser is generated by the command —

```
D:\work> gppg /no-lines 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 /nologo /r: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.

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`".

6.2 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 an 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 9. As described in Section 2.3.1, this semantic value type will be

Figure 9: 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 10 shows the relevant production rules. The action creates a new string-builder object and appends the first digit. Each

Figure 10: 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) {
                     scanner.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*.

Running the Program

The parser is generated by the command —

```
D:\work> gppg /no-lines 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 /nologo /r:ShiftReduceParser.dll realcalc.cs
```

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

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

7 Appendix A: *GPPG* Special Symbols

7.1 Keyword Commands

Keyword	Meaning
`%defines`	*gppg* will create a "*basename*.tokens" file defining the token enumeration that the scanner will use. The scanner does not need this text file, but it is useful for other tools.
`%left`	this marker declares that the following token or tokens will have *left associativity*, that is, $a \bullet b \bullet c$ is interpreted as $(a \bullet b) \bullet c$.
`%locations`	this marker is ignored in this version: location tracking is always turned on in *gppg*.
`%namespace`	this marker defines the namespace in which the parser class will be defined. The namespace argument is a dotted name.
`%nonassoc`	this marker declares that the following token or tokens are not associative. This means that $a \bullet b \bullet c$ is a syntax error.
`%output`	allows the output stream to be redirected to a specified, named file. See section 2.2.
`%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 2.4.1 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 $a \bullet (b \bullet c)$.
`%start`	this marker allows the goal, (start) symbol of the grammar to be specified, instead of being taken from the left-hand-symbol of the first production rule.
`%token`	declares that the following names are tokens of the lexicon.
`%tokentype`	this marker allows for the default token enumeration class name, "*Tokens*", to be overridden. The argument must be a valid *C#* simple identifier.
`%type`	the form "`%type` < *member* > *non-terminal list*", where *member* is the name of a member in a union declaration, declares that the following non-terminal symbols set the value of the nominated member.
`%union`	marks the start of a semantic value-type declaration. See section 2.3.1 for more detail.
`%using`	this marker adds the given namespace to the parser's using list. The argument is a dotted name, in general.

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

7.2 Semantic Action Symbols

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

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.