TP Lex and Yacc – The Compiler Writer's Tools for Turbo Pascal Version 4.1 User Manual

Albert Gräf
Department of Musicinformatics
Johannes Gutenberg-University Mainz

ag@muwiinfa.geschichte.uni-mainz.de

April 1998

Contents

1 Introduction

This document describes the TP Lex and Yacc compiler generator toolset. These tools are designed especially to help you prepare compilers and similar programs like text processing utilities and command language interpreters with the Turbo Pascal (TM) programming language.

TP Lex and Yacc are Turbo Pascal adaptions of the well-known UNIX (TM) utilities Lex and Yacc, which were written by M.E. Lesk and S.C. Johnson at Bell Laboratories, and are used with the C programming language. TP Lex and Yacc are intended to be approximately "compatible" with these programs. However, they are an independent development of the author, based on the techniques described in the famous "dragon book" of Aho, Sethi and Ullman (Aho, Sethi, Ullman: *Compilers : principles, techniques and tools,* Reading (Mass.), Addison-Wesley, 1986).

Version 4.1 of TP Lex and Yacc works with all recent flavours of Turbo/Borland Pascal, including Delphi, and with the Free Pascal Compiler, a free Turbo Pascal-compatible compiler which currently runs on DOS and Linux (other ports are under development). Recent information about TP Lex/Yacc, and the sources are available from the TPLY homepage:

`http://www.musikwissenschaft.uni-mainz.de/~ag/tply`

For information about the Free Pascal Compiler, please refer to:

`http://www.freepascal.org`

TP Lex and Yacc, like any other tools of this kind, are not intended for novices or casual programmers; they require extensive programming experience as well as a thorough understanding of the principles of parser design and implementation to be put to work successfully. But if you are a seasoned Turbo Pascal programmer with some background in compiler design and formal language theory, you will almost certainly find TP Lex and Yacc to be a powerful extension of your Turbo Pascal toolset.

This manual tells you how to get started with the TP Lex and Yacc programs and provides a short description of these programs. Some knowledge about the C versions of Lex and Yacc will be useful, although not strictly necessary. For further reading, you may also refer to:

- Aho, Sethi and Ullman: *Compilers : principles, techniques and tools.* Reading (Mass.), Addison-Wesley, 1986.
- Johnson, S.C.: *Yacc – yet another compiler-compiler.* CSTR-32, Bell Telephone Laboratories, 1974.
- Lesk, M.E.: *Lex – a lexical analyser generator.* CSTR-39, Bell Telephone Laboratories, 1975.
- Schreiner, Friedman: *Introduction to compiler construction with UNIX.* Prentice-Hall, 1985.
- The Unix Programmer's Manual, Sections 'Lex' and 'Yacc'.

Credits

I would like to thank Berend de Boer (berend@pobox.com), who adapted TP Lex and Yacc to take advantage of the large memory models in Borland Pascal 7.0 and Delphi, and Michael Van Canneyt (Michael.VanCanneyt@fys.kuleuven.ac.be), the maintainer of the Linux version of the Free Pascal compiler, who is responsible for the Free Pascal port. And of course thanks are due to the many TP Lex/Yacc users all over the world for their support and comments which helped to improve these programs.

Getting Started

Instructions on how to compile and install TP Lex and Yacc on all supported platforms can be found in the `README` file contained in the distribution.

Once you have installed TP Lex and Yacc on your system, you can compile your first TP Lex and Yacc program `expr`. `Expr` is a simple desktop calculator program contained in the distribution, which consists of a lexical analyzer in the TP Lex source file `exprlex.l` and the parser and main program in the TP Yacc source file `expr.y`. To compile these programs, issue the commands

```
lex exprlex
yacc expr
```

That's it! You now have the Turbo Pascal sources (`exprlex.pas` and `expr.pas`) for the `expr` program. Use the Turbo Pascal compiler to compile these programs as follows:

```
tpc expr
```

(Of course, the precise compilation command depends on the type of compiler you are using. Thus you may have to replace `tpc` with `bpc`, `dcc` or `dcc32`, depending on the version of the Turbo/Borland/Delphi compiler you have, and with `ppc386` for the Free Pascal compiler. If you are using TP Lex and Yacc with Free Pascal under Linux, the corresponding commands are:

```
plex exprlex
pyacc expr
ppc386 expr
```

Note that in the Linux version, the programs are named `plex` and `pyacc` to avoid name clashes with the corresponding UNIX utilities.)

Having compiled `expr.pas`, you can execute the `expr` program and type some expressions to see it work (terminate the program with an empty line). There is a number of other sample TP Lex and Yacc programs (`.l` and `.y` files) in the distribution, including a TP Yacc cross reference utility and a complete parser for Standard Pascal.

The TP Lex and Yacc programs recognize some options which may be specified anywhere on the command line. E.g.,

```
lex -o exprlex
```

runs TP Lex with "DFA optimization" and

```
yacc -v expr
```

runs TP Yacc in "verbose" mode (TP Yacc generates a readable description of the generated parser).

The TP Lex and Yacc programs use the following default filename extensions:

- `.l`: TP Lex input files
- `.y`: TP Yacc input files
- `.pas`: TP Lex and Yacc output files

As usual, you may overwrite default filename extensions by explicitly specifying suffixes.

If you ever forget how to run TP Lex and Yacc, you can issue the command `lex` or `yacc` (resp. `plex` or `pyacc`) without arguments to get a short summary of the command line syntax.

2 TP Lex

This section describes the TP Lex lexical analyzer generator.

2.1 Usage

```
lex [options] lex-file[.l] [output-file[.pas]]
```

2.2 Options

`-v` "Verbose:" Lex generates a readable description of the generated lexical analyzer, written to lex-file with new extension `.lst`.

`-o` "Optimize:" Lex optimizes DFA tables to produce a minimal DFA.

2.3 Description

TP Lex is a program generator that is used to generate the Turbo Pascal source code for a lexical analyzer subroutine from the specification of an input language by a regular expression grammar.

TP Lex parses the source grammar contained in `lex-file` (with default suffix `.l`) and writes the constructed lexical analyzer subroutine to the specified `output-file` (with default suffix `.pas`); if no output file is specified, output goes to `lex-file` with new suffix `.pas`. If any errors are found during compilation, error messages are written to the list file (`lex-file` with new suffix `.lst`).

The generated output file contains a lexical analyzer routine, `yylex`, implemented as:

```
function yylex : Integer;
```

This routine has to be called by your main program to execute the lexical analyzer. The return value of the `yylex` routine usually denotes the number of a token recognized by the lexical analyzer (see the `return` routine in the `LexLib` unit). At end-of-file the `yylex` routine normally returns `0`.

The code template for the `yylex` routine may be found in the `yylex.cod` file. This file is needed by TP Lex when it constructs the output file. It must be present either in the current directory or in the directory from which TP Lex was executed (TP Lex searches these directories in the indicated order). (NB: For the Linux/Free Pascal version, the code template is searched in some directory defined at compile-time instead of the execution path, usually /usr/lib/fpc/lexyacc.)

The TP Lex library (`LexLib`) unit is required by programs using Lex-generated lexical analyzers; you will therefore have to put an appropriate `uses` clause into your program or unit that contains the lexical analyzer routine. The `LexLib` unit also provides various useful utility routines; see the file `lexlib.pas` for further information.

2.4 Lex Source

A TP Lex program consists of three sections separated with the `%%` delimiter:

```
definitions
%%
rules
%%
auxiliary procedures
```

All sections may be empty. The TP Lex language is line-oriented; definitions and rules are separated by line breaks. There is no special notation for comments, but (Turbo Pascal style) comments may be included as Turbo Pascal fragments (see below).

The definitions section may contain the following elements:

- regular definitions in the format:

  ```
  name   substitution
  ```

  which serve to abbreviate common subexpressions. The `{name}` notation causes the corresponding substitution from the definitions section to be inserted into a regular expression. The name must be a legal identifier (letter followed by a sequence of letters and digits; the underscore counts as a letter; upper- and lowercase are distinct). Regular definitions must be non-recursive.

- start state definitions in the format:

  ```
  %start name ...
  ```

  which are used in specifying start conditions on rules (described below). The `%start` keyword may also be abbreviated as `%s` or `%S`.

- Turbo Pascal declarations enclosed between `%{` and `%}`. These will be inserted into the output file (at global scope). Also, any line that does not look like a Lex definition (e.g., starts with blank or tab) will be treated as Turbo Pascal code. (In particular, this also allows you to include Turbo Pascal comments in your Lex program.)

The rules section of a TP Lex program contains the actual specification of the lexical analyzer routine. It may be thought of as a big `CASE` statement discriminating over the different patterns to be matched and listing the corresponding statements (actions) to be executed. Each rule consists of a regular expression describing the strings to be matched in the input, and a corresponding action, a Turbo Pascal statement to be executed when the expression matches. Expression and statement are delimited with whitespace (blanks and/or tabs). Thus the format of a Lex grammar rule is:

```
expression      statement;
```

Note that the action must be a single Turbo Pascal statement terminated with a semicolon (use `begin ... end` for compound statements). The statement may span multiple lines if the successor lines are indented with at least one blank or tab. The action may also be replaced by the `|` character, indicating that the action for this rule is the same as that for the next one.

The TP Lex library unit provides various variables and routines which are useful in the programming of actions. In particular, the `yytext` string variable holds the text of the matched string, and the `yyleng` Byte variable its length.

Regular expressions are used to describe the strings to be matched in a grammar rule. They are built from the usual constructs describing character classes and sequences, and operators specifying repetitions and alternatives. The precise format of regular expressions is described in the next section.

The rules section may also start with some Turbo Pascal declarations (enclosed in `%{ %}`) which are treated as local declarations of the actions routine.

Finally, the auxiliary procedures section may contain arbitrary Turbo Pascal code (such as supporting routines or a main program) which is simply tacked on to the end of the output file. The auxiliary procedures section is optional.

2.5 Regular Expressions

Table 1 summarizes the format of the regular expressions recognized by TP Lex (also compare Aho, Sethi, Ullman 1986, fig. 3.48). c stands for a single character, s for a string, r for a regular expression, and n, m for nonnegative integers.

The operators `*`, `+`, `?` and `{}` have highest precedence, followed by concatenation. The `|` operator has lowest precedence. Parentheses `()` may be used to group expressions and overwrite default precedences. The `<>` and `/` operators may only occur once in an expression.

The usual C-like escapes are recognized:

- `\n` denotes newline

<table>
<tr><th>EXPRESSION</th><th>MATCHES</th><th>EXAMPLE</th></tr>
<tr><td>c</td><td>any non-operator character c</td><td>a</td></tr>
<tr><td>\ c</td><td>character c literally</td><td>*</td></tr>
<tr><td>"s"</td><td>string s literally</td><td>"**"</td></tr>
<tr><td>.</td><td>any character but newline</td><td>a.*b</td></tr>
<tr><td>^</td><td>beginning of line</td><td>^abc</td></tr>
<tr><td>$</td><td>end of line</td><td>abc$</td></tr>
<tr><td>[s]</td><td>any character in s</td><td>[abc]</td></tr>
<tr><td>[^s]</td><td>any character not in s</td><td>[^abc]</td></tr>
<tr><td>r*</td><td>zero or more r's</td><td>a*</td></tr>
<tr><td>r+</td><td>one or more r's</td><td>a+</td></tr>
<tr><td>r?</td><td>zero or one r</td><td>a?</td></tr>
<tr><td>r{m,n}</td><td>m to n occurrences of r</td><td>a{1,5}</td></tr>
<tr><td>r{m}</td><td>m occurrences of r</td><td>a{5}</td></tr>
<tr><td>r_1r_2</td><td>r_1 then r_2</td><td>ab</td></tr>
<tr><td>r_1|r_2</td><td>r_1 or r_2</td><td>a|b</td></tr>
<tr><td>(r)</td><td>r</td><td>(a|b)</td></tr>
<tr><td>r_1/r_2</td><td>r_1 when followed by r_2</td><td>a/b</td></tr>
<tr><td><x>r</td><td>r when in start condition x</td><td><x>abc</td></tr>
</table>

Table 1: Regular expressions.

- `\r` denotes carriage return
- `\t` denotes tab
- `\b` denotes backspace
- `\f` denotes form feed
- `\`*nnn* denotes character no. *nnn* in octal base

You can also use the `\` character to quote characters which would otherwise be interpreted as operator symbols. In character classes, you may use the `-` character to denote ranges of characters. For instance, `[a-z]` denotes the class of all lowercase letters.

The expressions in a TP Lex program may be ambigious, i.e. there may be inputs which match more than one rule. In such a case, the lexical analyzer prefers the longest match and, if it still has the choice between different rules, it picks the first of these. If no rule matches, the lexical analyzer executes a default action which consists of copying the input character to the output unchanged. Thus, if the purpose of a lexical analyzer is to translate some parts of the input, and leave the rest unchanged, you only have to specify the patterns which have to be treated specially. If, however, the lexical analyzer has to absorb its whole input, you will have to provide rules that match everything. E.g., you might use the rules

```
.     |
\n    ;
```

which match "any other character" (and ignore it).

Sometimes certain patterns have to be analyzed differently depending on some amount of context in which the pattern appears. In such a case the `/` operator is useful. For instance, the expression `a/b` matches `a`, but only if followed by `b`. Note that the `b` does not belong to the match; rather, the lexical analyzer, when matching an `a`, will look ahead in the input to see whether it is followed by a `b`, before it declares that it has matched an `a`. Such lookahead may be arbitrarily complex (up to the size of the `LexLib` input buffer). E.g., the pattern `a/.*b` matches an `a` which is followed by a `b` somewhere on the same input line. TP Lex also has a means to specify left context which is described in the next section.

2.6 Start Conditions

TP Lex provides some features which make it possible to handle left context. The `^` character at the beginning of a regular expression may be used to denote the beginning of the line. More distant left context can be described conveniently by using start conditions on rules.

Any rule which is prefixed with the `<>` construct is only valid if the lexical analyzer is in the denoted start state. For instance, the expression `<x>a` can only be matched if the lexical analyzer is in start state `x`. You can have multiple start states in a rule; e.g., `<x,y>a` can be matched in start states `x` or `y`.

Start states have to be declared in the definitions section by means of one or more start state definitions (see above). The lexical analyzer enters a start state through a call to the `LexLib` routine `start`. E.g., you may write:

```
%start x y
%%
<x>a    start(y);
<y>b    start(x);
%%
begin
  start(x); if yylex=0 then ;
end.
```

Upon initialization, the lexical analyzer is put into state `x`. It then proceeds in state `x` until it matches an `a` which puts it into state `y`. In state `y` it may match a `b` which puts it into state `x` again, etc.

Start conditions are useful when certain constructs have to be analyzed differently depending on some left context (such as a special character at the beginning of the line), and if multiple lexical analyzers have to work in concert. If a rule is not prefixed with a start condition, it is valid in all user-defined start states, as well as in the lexical analyzer's default start state.

2.7 Lex Library

The TP Lex library (`LexLib`) unit provides various variables and routines which are used by Lex-generated lexical analyzers and application programs. It provides the input and output streams and other internal data structures used by the lexical analyzer routine, and supplies some variables and utility routines which may be used by actions and application programs. Refer to the file `lexlib.pas` for a closer description.

You can also modify the Lex library unit (and/or the code template in the `yylex.cod` file) to customize TP Lex to your target applications. E.g., you might wish to optimize the code of the lexical analyzer for some special application, make the analyzer read from/write to memory instead of files, etc.

2.8 Implementation Restrictions

Internal table sizes and the main memory available limit the complexity of source grammars that TP Lex can handle. There is currently no possibility to change internal table sizes (apart from modifying the sources of TP Lex itself), but the maximum table sizes provided by TP Lex seem to be large enough to handle most realistic applications. The actual table sizes depend on the particular implementation (they are much larger than the defaults if TP Lex has been compiled with one of the 32 bit compilers such as Delphi 2 or Free Pascal), and are shown in the statistics printed by TP Lex when a compilation is finished. The units given there are "p" (positions, i.e. items in the position table used to construct the DFA), "s" (DFA states) and "t" (transitions of the generated DFA).

As implemented, the generated DFA table is stored as a typed array constant which is inserted into the `yylex.cod` code template. The transitions in each state are stored in order. Of course it would have been more efficient to generate a big `CASE` statement instead, but I found that this may cause problems with the encoding of large DFA tables because Turbo Pascal has a quite rigid limit on the code size of individual procedures. I decided to use a scheme in which transitions on different symbols to the same state are merged into one single transition (specifying a character set and the corresponding next state). This keeps the number of transitions in each state quite small and still allows a fairly efficient access to the transition table.

The TP Lex program has an option (`-o`) to optimize DFA tables. This causes a minimal DFA to be generated, using the algorithm described in Aho, Sethi, Ullman (1986). Although the absolute limit on the number of DFA states that TP Lex can handle is at least 300, TP Lex poses an additional restriction (100) on the number of states in the initial partition of the DFA optimization algorithm. Thus, you may get a fatal `integer set overflow` message when using the `-o` option even when TP Lex is able to generate an unoptimized DFA. In such cases you will just have to be content with the unoptimized DFA. (Hopefully, this will be fixed in a future version. Anyhow, using the merged transitions scheme described above, TP Lex usually constructs unoptimized DFA's which are not far from being optimal, and thus in most cases DFA optimization won't have a great impact on DFA table sizes.)

2.9 Differences from UNIX Lex

Major differences between TP Lex and UNIX Lex are listed below.

- TP Lex produces output code for Turbo Pascal, rather than for C.
- Character tables (`%T`) are not supported; neither are any directives to determine internal table sizes (`%p`, `%n`, etc.).
- Library routines are named differently from the UNIX version (e.g., the `start` routine takes the place of the `BEGIN` macro of UNIX Lex), and, of course, all macros of UNIX Lex (`ECHO`, `REJECT`, etc.) had to be implemented as procedures.

- The TP Lex library unit starts counting line numbers at 0, incrementing the count *before* a line is read (in contrast, UNIX Lex initializes `yylineno` to 1 and increments it *after* the line end has been read). This is motivated by the way in which TP Lex maintains the current line, and will not affect your programs unless you explicitly reset the `yylineno` value (e.g., when opening a new input file). In such a case you should set `yylineno` to 0 rather than 1.

3 TP Yacc

This section describes the TP Yacc compiler compiler.

3.1 Usage

```
yacc [options] yacc-file[.y] [output-file[.pas]]
```

3.2 Options

`-v` "Verbose:" TP Yacc generates a readable description of the generated parser, written to `yacc-file` with new extension `.lst`.

`-d` "Debug:" TP Yacc generates parser with debugging output.

3.3 Description

TP Yacc is a program that lets you prepare parsers from the description of input languages by BNF-like grammars. You simply specify the grammar for your target language, augmented with the Turbo Pascal code necessary to process the syntactic constructs, and TP Yacc translates your grammar into the Turbo Pascal code for a corresponding parser subroutine named `yyparse`.

TP Yacc parses the source grammar contained in `yacc-file` (with default suffix `.y`) and writes the constructed parser subroutine to the specified `output-file` (with default suffix `.pas`); if no output file is specified, output goes to `yacc-file` with new suffix `.pas`. If any errors are found during compilation, error messages are written to the list file (`yacc-file` with new suffix `.lst`).

The generated parser routine, `yyparse`, is declared as:

```
function yyparse : Integer;
```

This routine may be called by your main program to execute the parser. The return value of the `yyparse` routine denotes success or failure of the parser (possible return values: 0 = success, 1 = unrecoverable syntax error or parse stack overflow).

Similar to TP Lex, the code template for the `yyparse` routine may be found in the `yyparse.cod` file. The rules for locating this file are analogous to those of TP Lex (see Section *TP Lex*).

The TP Yacc library (`YaccLib`) unit is required by programs using Yacc- generated parsers; you will therefore have to put an appropriate `uses` clause into your program or unit that contains the parser routine. The `YaccLib` unit also provides some routines which may be used to control the actions of the parser. See the file `yacclib.pas` for further information.

3.4 Yacc Source

A TP Yacc program consists of three sections separated with the `%%` delimiter:

```
definitions
%%
rules
%%
auxiliary procedures
```

The TP Yacc language is free-format: whitespace (blanks, tabs and newlines) is ignored, except if it serves as a delimiter. Comments have the C-like format `/* ... */`. They are treated as whitespace. Grammar symbols are denoted by identifiers which have the usual form (letter, including underscore, followed by a sequence of letters and digits; upper- and lowercase is distinct). The TP Yacc language also has some keywords which always start with the `%` character. Literals are denoted by characters enclosed in single quotes. The usual C-like escapes are recognized:

- `\n` denotes newline
- `\r` denotes carriage return
- `\t` denotes tab
- `\b` denotes backspace
- `\f` denotes form feed
- `\`*nnn* denotes character no. *nnn* in octal base

3.5 Definitions

The first section of a TP Yacc grammar serves to define the symbols used in the grammar. It may contain the following types of definitions:

- start symbol definition: A definition of the form

```
%start symbol
```

  declares the start nonterminal of the grammar (if this definition is omitted, TP Yacc assumes the left-hand side nonterminal of the first grammar rule as the start symbol of the grammar).

- terminal definitions: Definitions of the form

```
%token symbol ...
```

  are used to declare the terminal symbols ("tokens") of the target language. Any identifier not introduced in a `%token` definition will be treated as a nonterminal symbol.

  As far as TP Yacc is concerned, tokens are atomic symbols which do not have an innert structure. A lexical analyzer must be provided which takes on the task of tokenizing the input stream and return the individual tokens and literals to the parser (see Section *Lexical Analysis*).

- precedence definitions: Operator symbols (terminals) may be associated with a precedence by means of a precedence definition which may have one of the following forms

```
%left symbol ...
%right symbol ...
%nonassoc symbol ...
```

  which are used to declare left-, right- and nonassociative operators, respectively. Each precedence definition introduces a new precedence level, lowest precedence first. E.g., you may write:

```
%nonassoc '<' '>' '=' GEQ LEQ NEQ
   /* relational operators */
%left     '+' '-'  OR
   /* addition operators */
%left     '*' '/' AND
  /* multiplication operators */
%right    NOT UMINUS
  /* unary operators */
```

  A terminal identifier introduced in a precedence definition may, but need not, appear in a `%token` definition as well.

- type definitions: Any (terminal or nonterminal) grammar symbol may be associated with a type identifier which is used in the processing of semantic values. Type tags of the form `<name>` may be used in token and precedence definitions to declare the type of a terminal symbol, e.g.:

  ```
  %token <Real>  NUM
  %left  <AddOp> '+' '-'
  ```

  To declare the type of a nonterminal symbol, use a type definition of the form:

  ```
  %type <name> symbol ...
  ```

  e.g.:

  ```
  %type <Real> expr
  ```

  In a `%type` definition, you may also omit the nonterminals, i.e. you may write:

  ```
  %type <name>
  ```

  This is useful when a given type is only used with type casts (see Section *Grammar Rules and Actions*), and is not associated with a specific nonterminal.

- Turbo Pascal declarations: You may also include arbitrary Turbo Pascal code in the definitions section, enclosed in `%{ %}`. This code will be inserted as global declarations into the output file, unchanged.

3.6 Grammar Rules and Actions

The second part of a TP Yacc grammar contains the grammar rules for the target language. Grammar rules have the format

```
name : symbol ... ;
```

The left-hand side of a rule must be an identifier (which denotes a nonterminal symbol). The right-hand side may be an arbitrary (possibly empty) sequence of nonterminal and terminal symbols (including literals enclosed in single quotes). The terminating semicolon may also be omitted. Different rules for the same left-hand side symbols may be written using the `|` character to separate the different alternatives:

```
name : symbol ...
     | symbol ...
     ...
     ;
```

For instance, to specify a simple grammar for arithmetic expressions, you may write:

```
%left '+' '-'
%left '*' '/'
%token NUM
%%
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '(' expr ')'
     | NUM
     ;
```

(The `%left` definitions at the beginning of the grammar are needed to specify the precedence and associativity of the operator symbols. This will be discussed in more detail in Section *Ambigious Grammars*.)

Grammar rules may contain actions – Turbo Pascal statements enclosed in `{ }` – to be executed as the corresponding rules are recognized. Furthermore, rules may return values, and access values returned by other rules. These "semantic" values are written as `$$` (value of the left-hand side nonterminal) and `$i` (value of the ith right-hand side symbol). They are kept on a special value stack which is maintained automatically by the parser.

Values associated with terminal symbols must be set by the lexical analyzer (more about this in Section *Lexical Analysis*). Actions of the form `$$ := $1` can frequently be omitted, since it is the default action assumed by TP Yacc for any rule that does not have an explicit action.

By default, the semantic value type provided by Yacc is `Integer`. You can also put a declaration like

```
%{
type YYSType = Real;
%}
```

into the definitions section of your Yacc grammar to change the default value type. However, if you have different value types, the preferred method is to use type definitions as discussed in Section *Definitions*. When such type definitions are given, TP Yacc handles all the necessary details of the YYSType definition and also provides a fair amount of type checking which makes it easier to find type errors in the grammar.

For instance, we may declare the symbols NUM and expr in the example above to be of type Real, and then use these values to evaluate an expression as it is parsed.

```
%left '+' '-'
%left '*' '/'
%token <Real> NUM
%type  <Real> expr
%%
expr : expr '+' expr   { $$ := $1+$3; }
     | expr '-' expr   { $$ := $1-$3; }
     | expr '*' expr   { $$ := $1*$3; }
     | expr '/' expr   { $$ := $1/$3; }
     | '(' expr ')'    { $$ := $2;    }
     | NUM
     ;
```

(Note that we omitted the action of the last rule. The "copy action" `$$ := $1` required by this rule is automatically added by TP Yacc.)

Actions may not only appear at the end, but also in the middle of a rule which is useful to perform some processing before a rule is fully parsed. Such actions inside a rule are treated as special nonterminals which are associated with an empty right-hand side. Thus, a rule like

```
x : y { action; } z
```

will be treated as:

```
x : y $act z
$act : { action; }
```

Actions inside a rule may also access values to the left of the action, and may return values by assigning to the `$$` value. The value returned by such an action can then be accessed by other actions using the usual `$i` notation. E.g., we may write:

```
x : y { $$ := 2*$1; } z { $$ := $2+$3; }
```

which has the effect of setting the value of x to

```
2*(the value of y)+(the value of z).
```

Sometimes it is desirable to access values in enclosing rules. This can be done using the notation `$i` with $i \leq 0$. `$0` refers to the first value "to the left" of the current rule, `$-1` to the second, and so on. Note that in this case the referenced value depends on the actual contents of the parse stack, so you have to make sure that the requested values are always where you expect them.

There are some situations in which TP Yacc cannot easily determine the type of values (when a typed parser is used). This is true, in particular, for values in enclosing rules and for the `$$` value in an action inside a rule. In such cases you may use a type cast to explicitly specify the type of a value. The format for such type casts is `$<name>$` (for left-hand side values) and `$<name>i` (for right-hand side values) where `name` is a type identifier (which must occur in a `%token`, precedence or `%type` definition).

3.7 Auxiliary Procedures

The third section of a TP Yacc program is optional. If it is present, it may contain any Turbo Pascal code (such as supporting routines or a main program) which is tacked on to the end of the output file.

3.8 Lexical Analysis

For any TP Yacc-generated parser, the programmer must supply a lexical analyzer routine named `yylex` which performs the lexical analysis for the parser. This routine must be declared as

```
function yylex : Integer;
```

The `yylex` routine may either be prepared by hand, or by using the lexical analyzer generator TP Lex (see Section *TP Lex*).

The lexical analyzer must be included in your main program behind the parser subroutine (the `yyparse` code template includes a forward definition of the `yylex` routine such that the parser can access the lexical analyzer). For instance, you may put the lexical analyzer routine into the auxiliary procedures section of your TP Yacc grammar, either directly, or by using the the Turbo Pascal include directive (`$I`).

The parser repeatedly calls the `yylex` routine to tokenize the input stream and obtain the individual lexical items in the input. For any literal character, the `yylex` routine has to return the corresponding character code. For the other, symbolic, terminals of the input language, the lexical analyzer must return corresponding integer codes. These are assigned automatically by TP Yacc in the order in which token definitions appear in the definitions section of the source grammar. The lexical analyzer can access these values through corresponding integer constants which are declared by TP Yacc in the output file.

For instance, if

```
%token NUM
```

is the first definition in the Yacc grammar, then TP Yacc will create a corresponding constant declaration

```
const NUM = 257;
```

in the output file (TP Yacc automatically assigns symbolic token numbers starting at 257; 1 thru 255 are reserved for character literals, 0 denotes end-of-file, and 256 is reserved for the special error token which will be discussed in Section *Error Handling*). This definition may then be used, e.g., in a corresponding TP Lex program as follows:

```
[0-9]+   return(NUM);
```

You can also explicitly assign token numbers in the grammar. For this purpose, the first occurrence of a token identifier in the definitions section may be followed by an unsigned integer. E.g. you may write:

```
%token NUM 299
```

Besides the return value of `yylex`, the lexical analyzer routine may also return an additional semantic value for the recognized token. This value is assigned to a variable named `yylval` and may then be accessed in actions through the `$i` notation (see above, Section *Grammar Rules and Actions*). The `yylval` variable is of type `YYSType` (the semantic value type, `Integer` by default); its declaration may be found in the `yyparse.cod` file.

For instance, to assign an `Integer` value to a `NUM` token in the above example, we may write:

```
[0-9]+   begin
           val(yytext, yylval, code);
           return(NUM);
         end;
```

This assigns `yylval` the value of the `NUM` token (using the Turbo Pascal standard procedure `val`).

If a parser uses tokens of different types (via a %token <name> definition), then the `yylval` variable will not be of type `Integer`, but instead of a corresponding variant record type which is capable of holding all the different value types declared in the TP Yacc grammar. In this case, the lexical analyzer must assign a semantic value to the corresponding record component which is named `yy`*name* (where *name* stands for the corresponding type identifier).

E.g., if token `NUM` is declared `Real`:

```
%token <Real> NUM
```

then the value for token NUM must be assigned to yylval.yyReal.

3.9 How The Parser Works

TP Yacc uses the LALR(1) technique developed by Donald E. Knuth and F. DeRemer to construct a simple, efficient, non-backtracking bottom-up parser for the source grammar. The LALR parsing technique is described in detail in Aho/Sethi/Ullman (1986). It is quite instructive to take a look at the parser description TP Yacc generates from a small sample grammar, to get an idea of how the LALR parsing algorithm works. We consider the following simplified version of the arithmetic expression grammar:

```
%token NUM
%left '+'
%left '*'
%%
expr : expr '+' expr
     | expr '*' expr
     | '(' expr ')'
     | NUM
     ;
```

When run with the -v option on the above grammar, TP Yacc generates the parser description listed below.

```
state 0:

        $accept : _ expr $end

        '('     shift 2
        NUM     shift 3
        .       error

        expr    goto 1

state 1:

        $accept : expr _ $end
        expr : expr _ '+' expr
```

```
        expr : expr _ '*' expr

        $end    accept
        '*'     shift 4
        '+'     shift 5
        .       error

state 2:

        expr : '(' _ expr ')'

        '('     shift 2
        NUM     shift 3
        .       error

        expr    goto 6

state 3:

        expr : NUM _    (4)

        .       reduce 4

state 4:

        expr : expr '*' _ expr

        '('     shift 2
        NUM     shift 3
        .       error

        expr    goto 7

state 5:

        expr : expr '+' _ expr

        '('     shift 2
        NUM     shift 3
        .       error
```

```
        expr    goto 8

state 6:

        expr : '(' expr _ ')'
        expr : expr _ '+' expr
        expr : expr _ '*' expr

        ')'     shift 9
        '*'     shift 4
        '+'     shift 5
        .       error

state 7:

        expr : expr '*' expr _  (2)
        expr : expr _ '+' expr
        expr : expr _ '*' expr

        .       reduce 2

state 8:

        expr : expr '+' expr _  (1)
        expr : expr _ '+' expr
        expr : expr _ '*' expr

        '*'     shift 4
        $end    reduce 1
        ')'     reduce 1
        '+'     reduce 1
        .       error

state 9:

        expr : '(' expr ')' _   (3)

        .       reduce 3
```

Each state of the parser corresponds to a certain prefix of the input which has already been seen. The parser description lists the grammar rules wich are parsed in each state, and indicates the portion of each rule which has already been parsed by an underscore. In state 0, the start state of the parser, the parsed rule is

```
$accept : expr $end
```

This is not an actual grammar rule, but a starting rule automatically added by TP Yacc. In general, it has the format

```
$accept : X $end
```

where `X` is the start nonterminal of the grammar, and `$end` is a pseudo token denoting end-of-input (the `$end` symbol is used by the parser to determine when it has successfully parsed the input).

The description of the start rule in state 0,

```
$accept : _ expr $end
```

with the underscore positioned before the `expr` symbol, indicates that we are at the beginning of the parse and are ready to parse an expression (nonterminal `expr`).

The parser maintains a stack to keep track of states visited during the parse. There are two basic kinds of actions in each state: *shift*, which reads an input symbol and pushes the corresponding next state on top of the stack, and *reduce* which pops a number of states from the stack (corresponding to the number of right-hand side symbols of the rule used in the reduction) and consults the *goto* entries of the uncovered state to find the transition corresponding to the left-hand side symbol of the reduced rule.

In each step of the parse, the parser is in a given state (the state on top of its stack) and may consult the current *lookahead symbol*, the next symbol in the input, to determine the parse action – shift or reduce – to perform. The parser terminates as soon as it reaches state 1 and reads in the endmarker, indicated by the *accept* action on `$end` in state 1.

Sometimes the parser may also carry out an action without inspecting the current lookahead token. This is the case, e.g., in state 3 where the only action is reduction by rule 4:

```
.        reduce 4
```

The default action in a state can also be *error* indicating that any other input represents a syntax error. (In case of such an error the parser will start syntactic error recovery, as described in Section *Error Handling*.)

Now let us see how the parser responds to a given input. We consider the input string `2+5*3` which is presented to the parser as the token sequence:

State	Stack	Lookahead	Action
0		`NUM`	shift state 3
3	0		reduce rule 4 (pop 1 state, uncovering state 0, then goto state 1 on symbol `expr`)
1	0	`+`	shift state 5
5	1 0	`NUM`	shift state 3
3	5 1 0		reduce rule 4 (pop 1 state, uncovering state 5, then goto state 8 on symbol `expr`)
8	5 1 0	`*`	shift 4
4	8 5 1 0	`NUM`	shift 3
3	4 8 5 1 0		reduce rule 4 (pop 1 state, uncovering state 4, then goto state 7 on symbol `expr`)
7	4 8 5 1 0		reduce rule 2 (pop 3 states, uncovering state 5, then goto state 8 on symbol `expr`)
8	5 1 0	`$end`	reduce rule 1 (pop 3 states, uncovering state 0, then goto state 1 on symbol `expr`)
1	0	`$end`	accept

Table 2: Parse of `NUM + NUM * NUM`.

```
NUM + NUM * NUM
```

Table 2 traces the corresponding actions of the parser. We also show the current state in each move, and the remaining states on the stack.

It is also instructive to see how the parser responds to illegal inputs. E.g., you may try to figure out what the parser does when confronted with:

```
NUM + )
```

or:

```
( NUM * NUM
```

You will find that the parser, sooner or later, will always run into an error action when confronted with errorneous inputs. An LALR parser will never shift an invalid symbol and thus will always find syntax errors as soon as it is possible during a left-to-right scan of the input.

TP Yacc provides a debugging option (`-d`) that may be used to trace the actions performed by the parser. When a grammar is compiled with the `-d` option, the generated parser will print out the actions as it parses its input.

3.10 Ambigious Grammars

There are situations in which TP Yacc will not produce a valid parser for a given input language. LALR(1) parsers are restricted to one-symbol lookahead on which they have to base their parsing decisions. If a grammar is ambigious, or cannot be parsed unambigiously using one-symbol lookahead, TP Yacc will generate parsing conflicts when constructing the parse table. There are two types of such conflicts: *shift/reduce conflicts* (when there is both a shift and a reduce action for a given input symbol in a given state), and *reduce/reduce* conflicts (if there is more than one reduce action for a given input symbol in a given state). Note that there never will be a shift/shift conflict.

When a grammar generates parsing conflicts, TP Yacc prints out the number of shift/reduce and reduce/reduce conflicts it encountered when constructing the parse table. However, TP Yacc will still generate the output code for the parser. To resolve parsing conflicts, TP Yacc uses the following built-in disambiguating rules:

- in a shift/reduce conflict, TP Yacc chooses the shift action.
- in a reduce/reduce conflict, TP Yacc chooses reduction of the first grammar rule.

The shift/reduce disambiguating rule correctly resolves a type of ambiguity known as the "dangling-else ambiguity" which arises in the syntax of conditional statements of many programming languages (as in Pascal):

```
%token IF THEN ELSE
%%
stmt : IF expr THEN stmt
     | IF expr THEN stmt ELSE stmt
     ;
```

This grammar is ambigious, because a nested construct like

```
IF expr-1 THEN IF expr-2 THEN stmt-1
  ELSE stmt-2
```

can be parsed two ways, either as:

```
IF expr-1 THEN ( IF expr-2 THEN stmt-1
  ELSE stmt-2 )
```

or as:

```
IF expr-1 THEN ( IF expr-2 THEN stmt-1 )
  ELSE stmt-2
```

The first interpretation makes an `ELSE` belong to the last unmatched `IF` which also is the interpretation chosen in most programming languages. This is also the way that a TP Yacc-generated parser will parse the construct since the shift/reduce disambiguating rule has the effect of neglecting the reduction of `IF expr-2 THEN stmt-1`; instead, the parser will shift the `ELSE` symbol which eventually leads to the reduction of `IF expr-2 THEN stmt-1 ELSE stmt-2`.

The reduce/reduce disambiguating rule is used to resolve conflicts that arise when there is more than one grammar rule matching a given construct. Such ambiguities are often caused by "special case constructs" which may be given priority by simply listing the more specific rules ahead of the more general ones.

For instance, the following is an excerpt from the grammar describing the input language of the UNIX equation formatter EQN:

```
%right SUB SUP
%%
expr : expr SUB expr SUP expr
     | expr SUB expr
     | expr SUP expr
     ;
```

Here, the `SUB` and `SUP` operator symbols denote sub- and superscript, respectively. The rationale behind this example is that an expression involving both sub- and superscript is often set differently from a superscripted subscripted expression (compare x_i^n to ${x_i}^n$). This special case is therefore caught by the first rule in the above example which causes a reduce/reduce conflict with rule 3 in expressions like `expr-1 SUB expr-2 SUP expr-3`. The conflict is resolved in favour of the first rule.

In both cases discussed above, the ambiguities could also be eliminated by rewriting the grammar accordingly (although this yields more complicated and less readable grammars). This may not always be the case. Often ambiguities are also caused by design errors in the grammar. Hence, if TP Yacc reports any parsing conflicts when constructing the parser, you should use the `-v` option to generate the parser description (`.lst` file) and check whether TP Yacc resolved the conflicts correctly.

There is one type of syntactic constructs for which one often deliberately uses an ambigious grammar as a more concise representation for a language that could also be specified unambigiously: the syntax of expressions. For instance, the following is an unambigious grammar for simple arithmetic expressions:

```
%token NUM
```

```
%%

expr    : term
        | expr '+' term
        ;

term    : factor
        | term '*' factor
        ;

factor  : '(' expr ')'
        | NUM
        ;
```

You may check yourself that this grammar gives `*` a higher precedence than `+` and makes both operators left-associative. The same effect can be achieved with the following ambigious grammar using precedence definitions:

```
%token NUM
%left '+'
%left '*'
%%
expr : expr '+' expr
     | expr '*' expr
     | '(' expr ')'
     | NUM
     ;
```

Without the precedence definitions, this is an ambigious grammar causing a number of shift/reduce conflicts. The precedence definitions are used to correctly resolve these conflicts (conflicts resolved using precedence will not be reported by TP Yacc).

Each precedence definition introduces a new precedence level (lowest precedence first) and specifies whether the corresponding operators should be left-, right- or nonassociative (nonassociative operators cannot be combined at all; example: relational operators in Pascal).

TP Yacc uses precedence information to resolve shift/reduce conflicts as follows. Precedences are associated with each terminal occuring in a precedence definition. Furthermore, each grammar rule is given the precedence of its rightmost terminal (this default choice can be overwritten using a `%prec` tag; see below). To resolve a shift/reduce conflict using precedence, both the symbol and the rule involved must have been assigned precedences. TP Yacc then chooses the parse action as follows:

- If the symbol has higher precedence than the rule: shift.
- If the rule has higher precedence than the symbol: reduce.
- If symbol and rule have the same precedence, the associativity of the symbol determines the parse action: if the symbol is left-associative: reduce; if the symbol is right-associative: shift; if the symbol is non-associative: error.

To give you an idea of how this works, let us consider our ambigious arithmetic expression grammar (without precedences):

```
%token NUM
%%
expr : expr '+' expr
     | expr '*' expr
     | '(' expr ')'
     | NUM
     ;
```

This grammar generates four shift/reduce conflicts. The description of state 8 reads as follows:

```
state 8:

        *** conflicts:

        shift 4, reduce 1 on '*'
        shift 5, reduce 1 on '+'

        expr : expr '+' expr _  (1)
        expr : expr _ '+' expr
        expr : expr _ '*' expr

        '*'     shift 4
        '+'     shift 5
        $end    reduce 1
        ')'     reduce 1
        .       error
```

In this state, we have successfully parsed a + expression (rule 1). When the next symbol is + or *, we have the choice between the reduction and shifting the symbol. Using the default shift/reduce disambiguating rule, TP Yacc has resolved these conflicts in favour of shift.

Now let us assume the above precedence definition:

```
%left '+'
%left '*'
```

which gives * higher precedence than + and makes both operators left-associative. The rightmost terminal in rule 1 is +. Hence, given these precedence definitions, the first conflict will be resolved in favour of shift (* has higher precedence than +), while the second one is resolved in favour of reduce (+ is left-associative).

Similar conflicts arise in state 7:

```
state 7:

        *** conflicts:

        shift 4, reduce 2 on '*'
        shift 5, reduce 2 on '+'

        expr : expr '*' expr _   (2)
        expr : expr _ '+' expr
        expr : expr _ '*' expr

        '*'     shift 4
        '+'     shift 5
        $end    reduce 2
        ')'     reduce 2
        .       error
```

Here, we have successfully parsed a * expression which may be followed by another + or * operator. Since * is left-associative and has higher precedence than +, both conflicts will be resolved in favour of reduce.

Of course, you can also have different operators on the same precedence level. For instance, consider the following extended version of the arithmetic expression grammar:

```
%token NUM
%left '+' '-'
%left '*' '/'
```

```
%%
expr    : expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | '(' expr ')'
        | NUM
        ;
```

This puts all "addition" operators on the first and all "multiplication" operators on the second precedence level. All operators are left-associative; for instance, `5+3-2` will be parsed as `(5+3)-2`.

By default, TP Yacc assigns each rule the precedence of its rightmost terminal. This is a sensible decision in most cases. Occasionally, it may be necessary to overwrite this default choice and explicitly assign a precedence to a rule. This can be done by putting a precedence tag of the form

```
%prec symbol
```

at the end of the corresponding rule which gives the rule the precedence of the specified symbol. For instance, to extend the expression grammar with a unary minus operator, giving it highest precedence, you may write:

```
%token NUM
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
expr    : expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | '-' expr      %prec UMINUS
        | '(' expr ')'
        | NUM
        ;
```

Note the use of the `UMINUS` token which is not an actual input symbol but whose sole purpose it is to give unary minus its proper precedence. If we omitted the precedence tag, both unary and binary minus would have the same precedence because they are represented by the same input symbol.

3.11 Error Handling

Syntactic error handling is a difficult area in the design of user-friendly parsers. Usually, you will not like to have the parser give up upon the first occurrence of an errorneous input symbol. Instead, the parser should recover from a syntax error, that is, it should try to find a place in the input where it can resume the parse.

TP Yacc provides a general mechanism to implement parsers with error recovery. A special predefined `error` token may be used in grammar rules to indicate positions where syntax errors might occur. When the parser runs into an error action (i.e., reads an errorneous input symbol) it prints out an error message and starts error recovery by popping its stack until it uncovers a state in which there is a shift action on the `error` token. If there is no such state, the parser terminates with return value 1, indicating an unrecoverable syntax error. If there is such a state, the parser takes the shift on the `error` token (pretending it has seen an imaginary `error` token in the input), and resumes parsing in a special "error mode."

While in error mode, the parser quietly skips symbols until it can again perform a legal shift action. To prevent a cascade of error messages, the parser returns to its normal mode of operation only after it has seen and shifted three legal input symbols. Any additional error found after the first shifted symbol restarts error recovery, but no error message is printed. The TP Yacc library routine `yyerrok` may be used to reset the parser to its normal mode of operation explicitly.

For a simple example, consider the rule

```
stmt : error ';' { yyerrok; }
```

and assume a syntax error occurs while a statement (nonterminal `stmt`) is parsed. The parser prints an error message, then pops its stack until it can shift the token `error` of the error rule. Proceeding in error mode, it will skip symbols until it finds a semicolon, then reduces by the error rule. The call to `yyerrok` tells the parser that we have recovered from the error and that it should proceed with the normal parse. This kind of "panic mode" error recovery scheme works well when statements are always terminated with a semicolon. The parser simply skips the "bad" statement and then resumes the parse.

Implementing a good error recovery scheme can be a difficult task; see Aho/Sethi/Ullman (1986) for a more comprehensive treatment of this topic. Schreiner and Friedman have developed a systematic technique to implement error recovery with Yacc which I found quite useful (I used it myself to implement error recovery in the TP Yacc parser); see Schreiner/Friedman (1985).

3.12 Yacc Library

The TP Yacc library (`YaccLib`) unit provides some global declarations used by the parser routine `yyparse`, and some variables and utility routines which may be used to control the actions of the parser and to implement error recovery. See the file `yacclib.pas` for a description of these variables and routines.

You can also modify the Yacc library unit (and/or the code template in the `yyparse.cod` file) to customize TP Yacc to your target applications.

3.13 Other Features

TP Yacc supports all additional language elements entitled as "Old Features Supported But not Encouraged" in the UNIX manual, which are provided for backward compatibility with older versions of (UNIX) Yacc:

- literals delimited by double quotes.
- multiple-character literals. Note that these are not treated as character sequences but represent single tokens which are given a symbolic integer code just like any other token identifier. However, they will not be declared in the output file, so you have to make sure yourself that the lexical analyzer returns the correct codes for these symbols. E.g., you might explicitly assign token numbers by using a definition like

  ```
  %token ':=' 257
  ```

  at the beginning of the Yacc grammar.
- `\` may be used instead of `%`, i.e. `\\` means `%%`, `\left` is the same as `%left`, etc.
- other synonyms:
  - `%<` for `%left`
  - `%>` for `%right`
  - `%binary` or `%2` for `%nonassoc`
  - `%term` or `%0` for `%token`
  - `%=` for `%prec`
- actions may also be written as `= { ... }` or `= single-statement;`
- Turbo Pascal declarations (`%{ ... %}`) may be put at the beginning of the rules section. They will be treated as local declarations of the actions routine.

3.14 Implementation Restrictions

As with TP Lex, internal table sizes and the main memory available limit the complexity of source grammars that TP Yacc can handle. However, the maximum table sizes provided by TP Yacc are large enough to handle quite complex grammars (such as the Pascal grammar in the TP Yacc distribution). The actual table sizes are shown in the statistics printed by TP Yacc when a compilation is finished. The given figures are "s" (states), "i" (LR0 kernel items), "t" (shift and goto transitions) and "r" (reductions).

The default stack size of the generated parsers is `yymaxdepth = 1024`, as declared in the TP Yacc library unit. This should be sufficient for any average application, but you can change the stack size by including a corresponding declaration in the definitions part of the Yacc grammar (or change the value in the `YaccLib` unit). Note that right-recursive grammar rules may increase stack space requirements, so it is a good idea to use left-recursive rules wherever possible.

3.15 Differences from UNIX Yacc

Major differences between TP Yacc and UNIX Yacc are listed below.

- TP Yacc produces output code for Turbo Pascal, rather than for C.
- TP Yacc does not support `%union` definitions. Instead, a value type is declared by specifying the type identifier itself as the tag of a `%token` or `%type` definition. TP Yacc will automatically generate an appropriate variant record type (`YYSType`) which is capable of holding values of any of the types used in `%token` and `%type`.

  Type checking is very strict. If you use type definitions, then any symbol referred to in an action must have a type introduced in a type definition. Either the symbol must have been assigned a type in the definitions section, or the `$<type-identifier>` notation must be used. The syntax of the `%type` definition has been changed slightly to allow definitions of the form

  ```
  %type <type-identifier>
  ```

  (omitting the nonterminals) which may be used to declare types which are not assigned to any grammar symbol, but are used with the `$<...>` construct.
- The parse tables constructed by this Yacc version are slightly greater than those constructed by UNIX Yacc, since a reduce action will only be chosen as the default action if it is the only action in the state. In difference, UNIX Yacc chooses a reduce action as the default action whenever it is the only reduce action of the state (even if there are other shift actions).

This solves a bug in UNIX Yacc that makes the generated parser start error recovery too late with certain types of error productions (see also Schreiner/Friedman, *Introduction to compiler construction with UNIX,* 1985). Also, errors will be caught sooner in most cases where UNIX Yacc would carry out an additional (default) reduction before detecting the error.

- Library routines are named differently from the UNIX version (e.g., the `yyerrlab` routine takes the place of the `YYERROR` macro of UNIX Yacc), and, of course, all macros of UNIX Yacc (`YYERROR`, `YYACCEPT`, etc.) had to be implemented as procedures.