Lex and Yacc – Compiler Writer's Tools for Turbo Pascal
Version 2.0

Albert Gräf
FB Mathematik
Johannes Gutenberg-Universität Mainz

November 22, 1989

Abstract

We describe a reimplementation of the compiler writer's tools Lex and Yacc for Borland's Turbo Pascal, running under MS-DOS. These programs are useful tools for the development of compilers, lexical analyzers, and similar applications, and are intended for experienced Turbo Pascal programmers with a background in compiler design, and for courses in compiler construction.

Contents

Introduction

This manual describes two popular compiler writer's tools, Lex and Yacc, which have seen extensive use on the UNIX system, and have been reimplemented by the author for the MS-DOS operating system.

The original (UNIX) versions of these programs are described in [2,3]. Other public domain and commercial remakes of Lex and Yacc are available under MS-DOS, e.g. from DECUS and Mortice Kern Systems. However, in difference to these implementations, the programs described in this manual are for use with Borland's Turbo Pascal, rather than with C. In particular, they support Turbo Pascal as their host and target programming language.

The Turbo Pascal Lex and Yacc versions are an independent development of the author, not containing any fragments from the original sources, or other UNIX stuff (happily, the theory underlying Lex and Yacc has been published, e.g., in [1], und thus is public domain). However, the names Lex and Yacc (which, as far as I know, are not copyrighted) are justified by the fact that the programs described here are intended to be approximately compatible with the original versions.

The intended audience for this manual are experienced (Turbo Pascal) programmers with knowledge of the basics of formal language and compiler theory and students in compiler construction courses, *not* novices or casual programmers. Thus, the manual is particularily concise and compact, while trying to cover all essential aspects of Turbo Pascal Lex and Yacc.

As a supplementary text, we strongly recommend the famous "dragon book" of Aho, Sethi and Ullman [1] which covers all important theoretical and practical aspects of compiler design and implementation.

The manual is organized as follows: Section 1 covers installation requirements and the installation process. Section 2 treats the lexical analyzer generator Lex; subsections are devoted to the format of regular expressions, the implementation of actions, the Lex library unit, character tables, Turbo Pascal tie-ins, implementation restrictions, bugs, and incompatibilities. Section 3 discusses the parser generator Yacc; subsections deal with actions in Yacc grammars, the lexical analyzer routine used by Yacc-generated parsers, the Yacc library unit, ambiguities in Yacc grammars, syntactic error handling, arbitrary value types in actions, the debugging of Yacc-generated parsers, the Yacc language syntax, and, finally, additional features, implementation restrictions and bugs. The appendix contains short descriptions of Lex and Yacc in the style of UNIX manual pages.

Note: Throughout the manual, the terms *Lex* and *Yacc* refer to the

Turbo Pascal versions. The original (UNIX) versions are denoted by the terms *UNIX* or *standard* Lex resp. Yacc.

1 Installation

Installation requirements:

- IBM PC/XT/AT or compatible, 512 KB RAM
- MS-DOS 3.10 or later (may also run under MS-DOS 2.x, but I have not tested it)
- Turbo Pascal 4.0 or later (has been tested with 4.0 and 5.0)

To install Turbo Pascal Lex and Yacc, simply copy the contents of the distribution disk to an appropriate disk and/or directory. You might also wish to put this directory on your DOS `path`. The programs generated with Lex and Yacc will need the LexLib and YaccLib units (`.tpu` files on the distribution disk) when compiled, so you might have to put them anywhere the Turbo Pascal compiler finds them (e.g., in the `turbo.tpl` library).

Here's the contents of the distribution disk:

`lex.exe`	the Lex program
`lexlib.*`	source and `.tpu` file for the `LexLib` unit
`yacc.exe`	the Yacc program
`yacclib.*`	source and `.tpu` file for the `YaccLib` unit
`read.me`	if present, contains addenda to the manual
`makefile`	makefile for the sample programs
`*.l, *.y`	sample Lex and Yacc programs
`man.dvi`	TEX dvi file for the manual

As shipped, the `LexLib` and `YaccLib` units are compiled with Turbo Pascal 4.0. If you're running Turbo Pascal 5.0 or later, you will have to recompile `lexlib.pas` and `yacclib.pas` with your version of the Turbo Pascal compiler.

You can use the `makefile` to compile the sample programs on the distribution disk (see the Turbo Pascal manual for a description of Borland's make, and the `makefile` for a description of its usage and the sample programs).

To run Turbo Pascal Lex and Yacc on your grammar sourcefiles, refer to the manual pages in the appendix for a description of the Lex and Yacc command formats.

2 Lex

Lex is a program to generate lexical analyzers from a given set of input patterns, specified as regular expressions. Table 1 summarizes the regular expressions Lex recognizes. In this table, c stands for any single character, r for a regular expression, and s for a string.

A *Lex program*, or *grammar*, in general, consists of three sections separated with the delimiter `%%`:

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

Both definitions and rules section may be empty, and the auxiliary procedures section may be omitted, together with the second `%%`. Thus, the minimal Lex program is

```
%%
```

(no definitions, no rules).

The *rules* section of a Lex program is a table of regular expressions and corresponding actions, specifying patterns to be matched in the input, and (Turbo Pascal) program statements to be executed when a pattern has been matched:

expression *statement*`;`
⋮

Here, *expression* and *statement* are delimited with whitespace. The statement must be a *single* Turbo Pascal statement (use `begin ... end` for compound statements) terminated with a semicolon; if the statement consists of multiple lines, the continuation lines must be indented with at least one blank or tab character. An action may also be replaced by the symbol `|`, in which case it is assumed that the action for the current rule is the same as that for the next one.

As already indicated, the *auxiliary procedures* section is optional. If it is present, it is assumed to contain valid Turbo Pascal code (such as supplementing routines, or a main program) which is simply tacked on to the end of the output (Turbo Pascal) program Lex produces.

Expression	Matches	Example
c	any non-operator character c	`a`
`\`c	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of line	`^abc`
`$`	end of line	`abc$`
`[`s`]`	any character in s	`[abc]`
`[^`s`]`	any character not in s	`[^abc]`
r`*`	zero or more r's	`a*`
r`+`	one or more r's	`a+`
r`?`	zero or one r	`a?`
r`{`m`,`n`}`	m to n occurrences of r	`a{1,5}`
r_1r_2	r_1 then r_2	`ab`
r_1`|`r_2	r_1 or r_2	`a|b`
`(`r`)`	r	`(a|b)`
r_1`/`r_2	r_1 when followed by r_2	`abc/123`

Table 1: Lex regular expressions (taken from [1, fig. 3.48]).

The *definitions* section of a Lex program may contain *regular definitions* of the form

name *expression*

defining a (regular expression) substitution for an identifier *name* (according to Turbo Pascal syntax). *name* and *expression* must be separated by white-space. Note that in difference to Pascal, upper- and lowercase in identifiers is always distinct.

The value of a regular definition for *name* can be referred to lateron using the notation `{`*name*`}`. Thus, regular definitions provide a sort of "constant declaration" for regular expressions.

From the source grammar, Lex produces an output program, written in Turbo Pascal, that defines a parameterless function

```
function yylex : integer;
```

implementing the lexical analyzer. When called, `yylex` reads an input file (standard input, by default), scanning for the patterns specified in the source grammar, and executing the corresponding actions as patterns are matched.

Normally, `yylex` scans the whole input file and returns with value 0 to the calling program upon encountering end-of-file (actions may also return other values to the calling program, cf. 2.2). Thus, in the normal case, a suitable main program calling `yylex` is something like:

```
begin
  if yylex=0 then { done }
end.
```

Such a main program must be supplied by the programmer, e.g., in the auxiliary procedures section (there is no default main program in the Lex library, as with UNIX Lex).

The lexical analyzer routine `yylex` scans for all patterns simultaneously. If more than one pattern is matched, the longest match is preferred; if there still remains more than one pattern making the longest match, the first such rule in the source grammar is chosen. This makes rules like

```
if                      writeln('keyword if');
[A-Za-z][A-Za-z0-9]*    writeln('identifier ', yytext);
```

work as expected (i.e., input `if` will be matched by the first, `if1` by the second rule).

A Lex program may also be *incomplete* in that it does not specify a pattern for any possible input. In such a case, the lexical analyzer executes a default action on unrecognized parts of the input, which consists of copying the input to an output file (standard output, by default). Thus, the trivial Lex program

```
%%
```

yields a routine that copies the input to the output file unchanged. On the other hand, if the input has to be absorbed completely, the programmer must supply rules that match everything, e.g.:

```
.       |
\n      ;
```

Example: The following Lex program counts words (sequences of non-whitespace characters) in an input file:

```
     uses LexLib;
     var count : integer;
%%
[^ \t\n]+    inc(count);
.            |
\n           ;
%%
begin
  count := 0;
  if yylex=0 then writeln('word count: ', count)
end.
```

A few remarks about the generated lexical analyzer routine are in order. Lex generates table-driven lexical analyzers in DFA technique [1, section 3.7] which usually are both quite compact and fast (though hand-coded lexical analyzers will often be more efficient). In particular, the matching time is proportional to the length of the input, unless ambiguity and lookahead requires a significant amount of rescanning. There are certain pathological regular expressions which cause exponential growth of the DFA table, however, they are rare.

Lex-generated lexical analyzers interface nicely with Yacc, because the `yylex` routine just meets Yacc's requirements of its lexical analyzer routine (actually, it was designed that way). Thus, a `yylex` routine prepared with Lex can be incorporated directly into a Yacc-generated parser.

2.1 Regular Expressions

In the Lex language regular expressions are used to denote string patterns to be matched in the input stream. The basic regular expressions are (cf. table 1):

- single characters: c stands for the literal character c itself, unless it is an operator character (one of `*`, `+`, `?`, etc., discussed below), in which case it must be quoted with the backslash `\`. Non-printable characters are specified using the C-like escapes listed in table 2. Note that `\0` marks end-of-file, and `\n` (newline) stands for end-of-line, i.e. the sequence carriage-return/line-feed in MS-DOS text files.
- strings: "s", where s is any character sequence, stands for the string s. To embed the double quote " in a string, it must be quoted with `\`.

Escape	Denotes
`\b`	backspace
`\t`	tab
`\n`	newline
`\`*nnn*	character no. *nnn* in octal
`\\`	backslash
`\`*c*	character *c* literally

Table 2: Lex character escapes.

- character classes: `[`*s*`]` stands for all characters in *s*, and `[^`*s*`]` denotes the complement of `[`*s*`]`. A `-` sign in a character class denotes ranges, e.g. `[a-z]` is the class of all lowercase letters. The period `.` is an abbreviation for the class of all characters except newline, i.e. `[^\n]`. Note that a character class never contains the end-of-file marker `\0`, unless it is explicitly included.

From these basic elements, larger regular expressions may be formed using the following operators:

- *r*`*`: stands for an arbitrary sequence of *r*'s (0 or more), where *r* is any regular expression.
- *r*`+`: stands for 1 or more *r*'s.
- *r*`?`: stands for 0 or 1 *r*.
- *r*`{`*m*`,`*n*`}`: stands for *m* to *n* *r*'s (where *m* and *n* are nonnegative integers). *r*`{`*m*`}` denotes *excactly* *m* *r*'s.
- r_1r_2: stands for r_1, followed by r_2, where r_1 and r_2 are arbitrary regular expressions.
- r_1`|`r_2: stands for r_1 or r_2.

The operators have been listed in order of decreasing precedence, i.e. `*`, `+`, `?` and `{`*m*`,`*n*`}` bind stronger than r_1r_2 (concatenation), which in turn precedes over `|` (alternation). Parentheses `( ... )` can be used to group regular expressions and override default precedences.

As already mentioned, subexpressions may be abbreviated with names, using regular definitions. If *name* has been defined as an expression *r*,

`{`*name*`}` specifies the regular expression r. Note that (in difference to UNIX Lex) the substituted expression r must always be a complete legal regular expression, which is actually substituted as an expression, not textually. This implies that `{`*name*`}` is treated as a parenthesized expression. Also note, that any references to *name* must follow its definition, i.e. recursive definitions are illegal.

Lex also supplies a number of operators that are used to specify left and right context. The context does not actually belong to the pattern, but determines whether the pattern itself may be matched.

Right context, or *lookahead* is specified using the *lookahead operator* `/`. r_1`/`r_2 stands for r_1, but only if followed by r_2, where r_1 and r_2 are arbitrary regular expressions (which may, however, not contain the lookahead operator themselves). r`$` may be used as an abbreviation for r`/[\0\n]`, i.e. r followed by line end or end-of-file.

The caret `^` stands for the beginning of the line, and thus marks immediate *left context*. More distant left context may be specified using user-defined *start states*. The expression `<`$s_1, \ldots, s_n$`>`r denotes a pattern r that is valid (i.e., may be matched) only if the lexical analyzer is in any of the start states $s_1, \ldots, s_n$. This requires one or more *start state* declarations in the definitions section that list all used start state identifiers (which, like expression names, must be legal Turbo Pascal identifiers).

`%start` s_1 s_2 $\cdots$ s_n

and

`%start` s_1
$\vdots$
`%start` s_n

are examples of valid start state declarations.

By default, the lexical analyzer is in the default start state, which has number 0. Lex also assigns unique numbers to all user-defined start states, and the `begin_` routine (cf. 2.2) can be used in an action or any other routine to put the lexical analyzer in the desired start state.

Start states are useful when certain patterns have to be analyzed differently, depending on some left context (such as a special character at the beginning of the line), or when multiple lexical analyzers are working in concert. Note that a rule without start state prefix is valid in the default start state, as well as in any user-defined start state. This may be used to factor out patterns that are to be matched in either user-defined state.

All the context operators may only appear in rules, not in regular definitions, and they may appear only once. Of course, context operators may be combined, as in `<`s`>^`r_1/r_2 which denotes a pattern r_1 that may only be matched if it occurs at the beginning of a line, is followed by an instance of r_2, and if the lexical analyzer is in user-defined start state s.

2.2 Actions

A rule specifies a regular expression pattern and an action (a Turbo Pascal statement) to be executed when the pattern is matched. Lex supplies a number of variables and routines useful in the programming of actions:

- `yytext`: a string variable, containing the matched text.
- `yyleng`: the length of `yytext`, i.e. `length(yytext)`. Note that the first and last character in `yytext` are `yytext[1]` and `yytext[yyleng]`, respectively.
- `yylineno`: the current line number in the input file; useful, e.g., in giving diagnostics.
- `yymore`: appends the next match to the current one (normally, the next match will overwrite the current one).
- `yyless`: the counterpart of `yymore`; `yyless(`n`)` causes the current match to be restricted to the first n characters, and returns the remaining characters at the end of the match to be reread by the lexical analyzer. This supplies a crude form of "lookahead". Since Lex also supports a more general form of lookahead (cf. 2.1), this routine is largely obsolete.
- `reject`: rejects the current match and executes whatever was "second choice" after the current rule, adjusting the match accordingly. This routine is useful when *all* (possibly overlapping) instances of patterns have to be detected; see the `digram` program on the distribution disk for an example.
- `return`: `return(`n`)`, n an integer value, causes `yylex` to return with the indicated value. This routine is useful when the lexical analyzer is used to partition the input file for a parser; n will then typically denote a token number (cf. 3.2).

- `begin_`: `begin_(`s`)` puts the lexical analyzer in start state s (cf. 2.1), where s is either 0 (default start state) or names a user-defined start state.

These, and other Lex-supplied variables and routines are also discussed in the interface of the `LexLib` unit (file `lexlib.pas` on the distribution disk).

2.3 Lex Library

The `LexLib` (Lex library) unit supplies the input/output routines used by the lexical analyzer. The I/O files are implemented as Pascal `text` files `yyin` and `yyout`. These are – by default – assigned to the redirectable MS-DOS standard input/output devices. However, the user program may also assign them to any suitable files/devices.

`yylex` accesses the input/output files through the following routines:

- `function input : char;`
  returns the next character from the input file.

- `procedure unput(c : char);`
  returns character c to the input buffer to be reread by a subsequent call to input.

- `procedure output(c : char);`
  appends character c to the output file.

All references to the `yyin`/`yyout` files of the `LexLib` unit are made through these three routines. Thus, a user program may well replace them altogether by other routines matching the specifications. This makes it possible for the lexical analyzer to access arbitrary input/output streams, such as special devices or internal memory.

The `LexLib` I/O routines and files may also be accessed directly through actions or the main program. However, care must be heeded under certain circumstances. In particular, direct access to the `yyin` file will bypass the buffering of `unput` characters and thus may sometimes not have the desired results.

The `yywrap` routine is a parameterless boolean function that determines whether the lexical analyzer should perform normal wrapup at end-of-file. If it returns `true`, normal wrapup is performed; if it returns `false`, `yylex` ignores the end-of-file mark and continues lexical analysis. The `LexLib` unit supplies a default version of `yywrap` which always returns `true`. This routine

may be replaced by a customized version that does application dependent processing at end-of-file. In particular, `yywrap` may arrange for more input and return `false` to resume lexical analysis (see the `findproc` program on the distribution disk for an example).

Note that the `LexLib` unit must be loaded with (almost) any Lex program such that the lexical analyzer routine may access I/O and other routines. To achieve this, the line

```
uses LexLib;
```

should be put at the beginning of the Lex program (see section 2.5 on how to incorporate Turbo Pascal source lines into the definitions section of a Lex program).

Refer also to the `LexLib` interface description contained in `lexlib.pas` on the distribution disk for a discussion of the `LexLib` I/O system and other routines.

2.4 Character Tables

The standard character encoding supported both by Lex and the `LexLib` unit is ASCII (to be more precise, IBM's 8-bit extension of ASCII). However the user may supply his own versions of `input`, `unput` and `output` (cf. 2.3), supporting their own character encoding. If such a customized character set is used, Lex must be told about it by means of a *character table* in the definitions section of the Lex program.

This table has the format:

```
%T
charno.    string
⋮
%T
```

Each line of the character table lists a character number (in the target code) and the corresponding ASCII representation(s) of this character. The usual escapes for non-printable characters are recognized (cf. table 2).

Example: To map the lower- and uppercase letters into $1, \ldots, 26$, the digits $0, \ldots, 9$ into codes $27, \ldots, 36$, and newline into 37, the following table may be used:

```
%T
```

```
1               Aa
2               Bb
3               Cc
      ...
26              Zz
27              0
28              1
      ...
36              9
37              \n
%T
```

If a character table is used, all characters (at least those actually used in the Lex grammar) should be in the table; all character numbers must be byte values (0..255), and no character may be mapped into two different codes.

2.5 Turbo Pascal Tie-ins

Frequently, a Lex program will not merely consist of definitions and rules, but also use other routines to be loaded with the lexical analyzer. One example is a main program that calls the `yylex` routine. We have already mentioned, that such a main program, and other supplementary routines, may be placed into the auxiliary procedures section.

For other target language (i.e. Turbo Pascal) tie-ins, the Lex language allows arbitrary code fragments to be included in the definitions and at the beginning of the rules section of the Lex grammar, by the following means:

- Any line in the source grammar starting in column one is assumed to contain Lex code (definitions or rules), which is processed by Lex.
- Any line indented with at least one space or tab character, and any sequence of lines enclosed between `%{` and `%}` is assumed to contain Turbo Pascal code, and is copied to the output file unchanged.

Code in the definitions section is inserted at the beginning of the output program, at global scope, while code at the beginning of the rules section is inserted as local declarations into the action routine that contains the actions of all rules. Thus, an indented line

```
var i : integer;
```

at the beginning of the rules section will declare an integer variable local to the action statements.

As a side-effect, these conventions allow *comments* to be included in a Lex program; comments should then follow host language (i.e. Turbo Pascal) syntax.

Example: A typical setup for a Lex program with necessary declarations, supplementary routines, and main program may look as follows:

```
     uses LexLib;
{ global declarations }
{ Lex definitions }
%%
{ local declarations }
{ Lex rules }
%%
{ supplementary routines }
begin { main program }
  ...
  if yylex=0 then { done };
  ...
end.
```

2.6 Implementation Restrictions, Bugs, and Incompatibilities

Lex poses some restrictions on the sizes of the input grammar and internal tables. Maximum table sizes are printed out by Lex at the end of a translation, along with statistics about actual table space requirements. An error message `table overflow` can also indicate that not enough main memory is available to Lex (possibly because of too many programs loaded into memory).

Since `yytext` is implemented as a Turbo Pascal `string` variable, the maximum size for a matched string is 255.

As implemented, the `reject` routine (cf. 2.2) does not rescan the input, but uses internal state information to determine the next possible match. Thus `reject` combined with modifications of the input stream may yield unexpected results.

There is a subtle (and, as far as I know, undocumented) bug in Lex, concerning certain types of lookahead patterns, that sometimes causes lookahead not to be restored properly. E.g., when the pattern `ab*/b` is matched, the last `b` is never returned to the input, but instead the whole matched sequence will be returned in `yytext`. This actually is a "misfeature", and seems to be comformant with UNIX Lex and even with the method of Aho/Sethi/Ullman for handling the lookahead operator [1, section 3.8], which Lex' treatment of the lookahead operator is based on.

When called, `yylex` partially initalizes itself. This implies that `yymore` and `reject` will not work between different invokations of `yylex`.

As discussed in 2.1, Lex substitutes *expressions*, not text, when a regular definition is referred to with the `{`*name*`}` notation. This is in contrast with the (textual) macro expansion scheme used in UNIX Lex. Although the approach taken in Turbo Pascal Lex is more restrictive, we feel that it is actually an improvement. In particular, regular definitions can be parsed and checked for validity immediately, such that errors in regular definitions can be detected as soon as possible. Also, the meaning of a regular definition is guaranteed to be independent of the context in which it is used.

Another (minor) difference between Turbo Pascal and UNIX Lex in the syntax of regular definitions is that Turbo Pascal Lex requires names for regular expressions to be legal (Turbo Pascal) identifiers, whereas UNIX Lex admits arbitrary character strings as names.

3 Yacc

Yacc ("Yet Another Compiler-Compiler") is a *parser generator*, i.e. a program that translates the specification of an input language by its BNF (Backus Naur Form) grammar into a parser subroutine, written in Turbo Pascal.

Similar to Lex, a Yacc *program* or *grammar* has the form

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

where the first section contains the definitions of the basic input symbols (*terminals*, also termed *tokens*) of the specified language, the second section the *grammar rules* for the *nonterminal* symbols of the language, and the third, and optional, section contains any additional Turbo Pascal code, such as supplementary routines and a main program, which will be tacked on to the end of the Turbo Pascal output program.

The *rules* section of a Yacc program is simply a BNF grammar for the target language, possibly augmented with *actions*, program statements to be executed as certain syntactic constructs are recognized by the parser. By default, the left-hand side of the first rule marks the *start symbol* of the grammar. It is also possible to declare a start symbol explicitly by means of a declaration of the form

`%start` A

in the definitions section, where A is the desired start symbol.

Grammar rules have the general format

A `:` $\beta_1 \cdots \beta_n$ `;`

where A is the left-hand side nonterminal of the rule, and $\beta_1 \cdots \beta_n$ is the (possibly empty) right-hand side sequence of nonterminal and terminal symbols β_i.

The terminating semicolon may be omitted, and several rules A `:` u_i for the same left-hand side nonterminal A may be abbreviated as

Escape	Denotes
'\n'	newline
'\r'	carriage return
'\''	single quote '
'\\'	backslash
'\t'	tab
'\b'	backspace
'\f'	form feed
'*nnn*'	character no. *nnn* in octal

Table 3: Yacc character escapes.

$$\begin{array}{lll} A & \texttt{:} & u_1 \\ & \texttt{|} & u_2 \\ & \vdots & \\ & \texttt{|} & u_n \end{array}$$

Nonterminal symbols are denoted by identifiers (letters, followed by digits and letters, where underscore `_` and period `.` count as letters, and upper- and lowercase are distinct).

Terminal symbols may either be *literals* (single characters enclosed in single quotes) or identifiers that are declared explicitly as terminals by a `%token` definition of the form

`%token` $\alpha_1 \cdots \alpha_n$

By convention, token identifiers are given in uppercase, such that they can be distinguished easily from nonterminal symbols.

In literals, the usual C-like escapes are recognized (cf. table 3).

Grammar rules may be augmented with *actions*, Turbo Pascal statements enclosed between `{ ... }`. Usually, actions appear at the end of rules, indicating the statements to be executed when an instance of a rule has been recognized (cf. 3.1).

The Yacc language is free-format: blanks, tabs, and newlines are ignored, except when they serve as delimiters. Yacc language comments have the format:

`/*` ... *anything except* `*/` ... `*/`

As with Lex, host language (i.e. Turbo Pascal) tie-ins may be specified by enclosing them in `%{ ... %}`. Such code fragments will be copied un-

changed, and inserted into the output file at appropriate places (code in the definitions section at global scope, code at the beginning of the rules section as local declarations of the action routine).

The class of grammars accepted by Yacc is LALR(1) with disambiguating rules (cf. [1, sections 4.7 and 4.8]). Yacc can successfully produce parsers for a large class of formal languages, including most modern programming languages (under UNIX, Yacc has been used to produce parsers for C, Fortran, APL, Pascal, Ratfor, Modula-2, and others).

From the source grammar, Yacc produces an output file containing the parser routine

```
function yyparse : integer;
```

together with any additional Turbo Pascal code supplied by the programmer.

`yyparse` repeatedly calls a lexical analyzer routine `yylex` to obtain tokens from the input file, and parses the input accordingly, executing appropriate actions as instances of grammar rules are recognized. `yyparse` returns with a value of 0 (successful parse terminated at end-of-file) or 1 (fatal error, such as parse stack overflow, or unrecoverable syntax error).

Thus, a main program that calls the parser routine may look as follows:

```
begin
  ...
  if yyparse=0 then { done } else { error };
  ...
end.
```

Main program and lexical analyzer routine must be supplied by the programmer. The `yylex` routine can also be prepared with Lex and then loaded with the parser, cf. 3.2. The main program is usually included at the end of the auxiliary procedures section.

The following is an example of a Yacc grammar for simple arithmetic expressions. Note that the symbol `NUMBER` is declared as a token expected to be returned by the lexical analyzer as a single input symbol.

```
%token NUMBER
%%
expr      : term
          | expr '+' term
```

```
            ;
term        : factor
            | term '*' factor
            ;
factor      : NUMBER
            | '(' expr ')'
            | '-' factor
            ;
```

A Lex program implementing the lexical analyzer for this Yacc program may look as follows:

```
     {$I expr.h} { definition of token numbers produced by
                   Yacc }
%%
[0-9]+          return(NUMBER);
.               |
\n              return(ord(yytext[1]));
                  { other literals returned as their character
                    codes }
```

3.1 Actions

As already indicated, grammar rules may be associated with *actions*, program statements that are executed as rules are recognized during the parse. Among other things, actions may print out results, modify internal data structures such as a symbol table, or construct a parse tree. Actions may also return a value for the left-hand side nonterminal, and process values returned by previous actions for right-hand side symbols of the rule.

For this purpose, Yacc assigns to each symbol of a rule a corresponding value (of type `integer`, by default, but arbitrary value types may be declared, cf. 3.6): `$$` denotes the value of the left-hand side nonterminal, `$`i the value of the i^{th} right-hand side symbol. Values are kept on a stack maintained by the parser as the input is parsed. The lifetime of a value begins when the corresponding syntactic entity (nonterminal or terminal) has been recognized by the parser, and ends when the parser reduces by an enclosing rule, thereby replacing the values associated with the right-hand side symbols by the value of the left-hand side nonterminal.

Nonterminals A obtain their values through assignments of the form `$$ :=` v, where A is the left-hand side of the corresponding rule, and v is some value, usually obtained by a function applied to the right-hand side values `$`i.

Terminals may also have associated values; these are set by the lexical analyzer through an assignment to the variable `yylval` supplied by Yacc (cf. 3.2), as necessary.

As an example, here is an extension of the arithmetic expression grammar, featuring actions that evaluate the input expression and a rule for the new start symbol `line` that is associated with an action that prints out the obtained result.

```
%token NUMBER
%%
line      : expr '\n'         { writeln($1) }
          ;
expr      : term              { $$ := $1 }
          | expr '+' term     { $$ := $1 + $3 }
          ;
term      : factor            { $$ := $1 }
          | term '*' factor   { $$ := $1 * $3 }
          ;
factor    : NUMBER            { $$ := $1 }
          | '(' expr ')'      { $$ := $2 }
          | '-' factor        { $$ := -$2 }
          ;
```

Note that the lexical analyzer must set the values of `NUMBER` tokens, which are referred to by `$1` in the rule `factor : NUMBER`. One can use a Lex rule like

```
     var code : integer;
[0-9]+         begin
                 val(yytext, yylval, code);
                 return(NUMBER)
               end;
```

that applies the Turbo Pascal standard procedure `val` to evaluate a `NUMBER` token.

Actually, we could have omitted the "copy actions" of the form `$$ := $1` in the above grammar, since this is the default action automatically assumed by Yacc for any rule without explicit action.

Yacc also allows actions within rules, i.e. actions that are to be executed before a rule has been fully parsed. A rule like

A `:` β `{ p; }` γ

will be treated as if it was written

A `:` β `$act` γ
`$act : { p; }`

introducing a new nonterminal `$act` matched to an empty right-hand side, and associated with the desired action.

In particular, the action `{ p; }` is treated as if it was a (nonterminal) grammar symbol, and thus can also return a value accessible with the usual `$`i notation. The action itself may also access values of symbols (and other actions) to the left of it. Thus, the rule A `:` β `{ p; }` γ is actually treated as if it consisted of *three* right-hand side symbols; `$1` denotes the value of β, `$2` the value of `{ p; }` (set in `p` by an assignment to `$$`) and `$3` is the value of γ.

Yacc's syntax-directed evaluation scheme makes it particularily easy to implement *synthesized attributes* along the guidelines of [1, section 5.6]. The evaluation of *inherited* attributes can often also be simulated by making use of *marker nonterminals*, see also [1]. To access marker nonterminals outside the scope of the rule to which they belong, Yacc also supports the notation `$`i, where $i \leq 0$, indicating a value to the *left* of the current rule, belonging to an enclosing rule (`$0` denotes the first symbol to the left, `$-1` the second, ...).

Consider

A	`:`	β B
	`|`	$\beta\beta'$ `{ $$ := $1 }` B
B	`:`	γ `{ $$ := f($0) }`

The anonymous marker nonterminal implemented by the action `{ $$ := $1 }` assures that the value of β can always be accessed through `$0` in the third rule. Note that without use of the marker nonterminal, the relative position of β's value on the stack would not be predictable.

Actions within rules, and access to values in enclosing rules, supply flexible means to implement syntax-directed evaluation schemes. However, care

must be heeded that such actions do not give rise to unwanted parsing conflicts caused by ambiguities (cf. 3.4).

3.2 Lexical Analysis

Yacc-generated parsers use a lexical analyzer routine `yylex` to obtain tokens from the input file. This routine must be supplied by the user, and is assumed to return an integer value denoting a basic input symbol. 0 (or negative) denotes end-of-file, and character literals are denoted by their character code. Usually, all other token numbers are assigned by Yacc automatically, in the order in which `%token` definitions appear in the grammar. Token numbers may also be assigned explicitly, by a definition of the form

`%token` α n

where α is the terminal symbol (literal or identifier), and n is the desired token number.

If there is a value associated with an input symbol, `yylex` should assign this value to the variable `yylval` supplied by Yacc. Usually, `yylval` has type `integer`, but this default can be overwritten (cf. 3.6).

Declarations shared by parser and lexical analyzer are put in the *header* (`.h`) file Yacc generates along with the (`.pas`) output file containing the parser routine. The header file declares the `yylval` variable and lists the token numbers; each token identifier is declared as a corresponding `integer` constant.

The header file should thus be included in a context where it is accessible by both the parser and the lexical analyzer. For instance, one may include both header file and lexical analyzer, in that order, in the definitions section of the grammar, by means of the Turbo Pascal include directive (`$I`):

```
%{
{$I header filename }
{$I lexical analyzer }
%}
```

As has already been indicated, the lexical analyzer generator Lex discussed in section 2 of this manual is a useful tool to produce lexical analyzers to be incorporated into Yacc-generated parsers.

3.3 Yacc Library

The Yacc library unit `YaccLib` contains some default declarations used by Yacc-generated parsers. It should therefore be loaded with `yyparse`, which can be achieved by the `uses` clause

```
%{
uses Yacclib;
%}
```

at the beginning of the Yacc program. Note that if the program includes a lexical analyzer prepared with Lex, the `LexLib` unit may also be required:

```
%{
uses Yacclib, LexLib;
%}
```

The routines implemented by the Yacc library are defaults that can be customized for the target application. In particular, these are the `yymsg` message printing routine and the `yydebugmsg` debug message printing routine. The Yacc library also declares defaults for the value type `YYSTYPE` (cf. 3.6) and the size of the parser stack (`yymaxdepth`; see also 3.9).

Refer to the interface description of the `YaccLib` unit for further information. The interface also describes some additional variables and routines which are not actually implemented by the Yacc library, but are contained in the Yacc output file. Some of these will also be mentioned in subsequent sections.

3.4 Ambiguity

If a grammar is non-LALR(1), Yacc will detect *parsing conflicts* when constructing the parse table. Such parsing conflicts are usually caused by inconsistencies and ambiguities in the grammar. Yacc will report the number of detected parsing conflicts, and will try to resolve these conflicts, using the methods outlined in [1, section 4.8]. Thus, Yacc will generate a parser for any grammar, even if it is non-LALR. However, if unexpected parsing conflicts arise, it is wise to consult the parser description (`.lst` file, cf. 3.7) generated by Yacc to determine whether the conflicts were resolved correctly. Otherwise the parser may not behave as expected.

An example of an ambigious grammar is the following, specifying a Pascal-like syntax of `IF-THEN-ELSE` statements:

```
%token IF THEN ELSE
%%
stmt      : IF expr THEN stmt                 /* 1 */
          | IF expr THEN stmt ELSE stmt       /* 2 */
```

The ambiguity in this grammar fragment, often referred to as the *dangling-else-ambiguity*, stems from the fact that it cannot be decided to which `THEN` a nested `ELSE` belongs: is

`IF` e_1 `THEN IF` e_2 `THEN` s_1 `ELSE` s_2

to be interpreted as:

(1) `IF` e_1 `THEN (` `IF` e_2 `THEN` s_1 `ELSE` s_2 `)`

or as:

(2) `IF` e_1 `THEN (` `IF` e_2 `THEN` s_1 `) ELSE` s_2 ?

Let us take a look at how such an ambigious construct would be parsed. When the parser has seen `IF` e_2 `THEN` s_1, it could recognize (reduce by) the first rule, yielding the second interpretation; but it could as well read ahead, shifting the next symbol `ELSE` on top of the parser stack, then parse s_2, and finally reduce by rule 2, which in effect yields the first interpretation. Thus, upon seeing the token `ELSE`, the parser is in a *shift/reduce* conflict: it cannot decide between shift (the token `ELSE`) and reduce (by the first rule).

In the dangling-else example, the grammar – with some effort – can be rewritten to eliminate the ambiguity, see [1, section 4.3]. However, this is not possible in general (there are languages that are intrinsically ambigious). Furthermore, an unambigious grammar may still be non-LALR (see [1, exercise 4.40] for an example). In particular, parsing decisions in an LALR parser are based on one-symbol-lookahead, which limits the class of grammars that may be used to construct a 'pure' LALR parser.

As implemented, Yacc always resolves shift/reduce conflicts in favour of shift. Thus a Yacc generated parser will correctly resolve the dangling-else ambiguity (assuming the common rule, that an `ELSE` belongs to the last unmatched `THEN`).

Another type of ambiguity arises when the parser has to choose between two different rules. Consider the following grammar fragment (for sub- and superscripted expressions in the UNIX equation formatter eqn):

```
%token SUB SUP
%%
expr      : expr SUB expr SUP expr /* 1 */
          | expr SUB expr          /* 2 */
          | expr SUP expr          /* 3 */
```

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 ${x_i}^n$.

The ambiguity arises in an expression of the form e_1 `SUB` e_2 `SUP` e_3. At the end of the expression, the parser can apply both rule 1 (reduce the whole expression) and rule 3 (reduce the subexpression e_2 `SUP` e_3).

This type of conflict is termed *reduce/reduce* conflict. Yacc resolves reduce/reduce conflicts in favour of the rule listed first in the Yacc grammar. Thus, "special case constructs" like the one above may be specified by listing them ahead of the more general rules. In our example, e_1 `SUB` e_2 `SUP` e_3 is always interpreted as an instance of the first rule (which presumably is the intended result).

To summarize, in absence of other strategies (to be discussed below), Yacc applies the following *default disambiguating rules*:

- a shift/reduce conflict is resolved in favour of shift.
- in a reduce/reduce conflict, the first applicable grammar rule is preferred.

In any case, the number of shift/reduce and reduce/reduce conflicts is reported by Yacc, since they could indicate inconsistencies in the grammar. Also, Yacc reports rules that are never reduced (possibly, because they are completely ruled out by disambiguating rules). A more detailed description of the detected conflicts may be found in the parser description (cf. 3.7).

The default disambiguating rules are often inappropriate in cases where an ambigious grammar is deliberately chosen as a more concise representation of an (unambigious) language. Consider the following grammar for arithmetic expressions:

```
%token NUMBER
%%
expr      : expr '+' expr
          | expr '*' expr
          | NUMBER
```

```
          | '(' expr ')'
          ;
```

There are several reasons why such an ambigious grammar might be preferred over the corresponding unambigious grammar:

```
%token NUMBER
%%
expr      : term
          | expr '+' term
          ;
term      : factor
          | term '*' factor
          ;
factor    : NUMBER
          | '(' expr ')'
          ;
```

In particular, the ambigious grammar is more concise and natural, and yields a more efficient parser [1, section 4.8].

The ambiguities in the first grammar may be resolved by specifying *precedences* and *associativity* of the involved operator symbols. This may be done by means of the following *precedence definitions*:

- `%left` *operator symbols*: specifies left-associative operators
- `%right` *operator symbols*: specifies right-associative operators (e.g., exponentiation)
- `%nonassoc` *operator symbols*: specifies non-associative operators (two operators of the same class may not be combined; e.g., relational operators in Pascal)

Each *operator symbol* may be a literal or a token-identifier; token-names appearing in precedence definitions may, but need not be declared with `%token` as well.

Each precedence declaration introduces a new precedence level, lowest precedence first. For the example above, assuming `'+'` and `'*'` to be left-associative, and `'*'` to take precedence over `'+'`, the corresponding Yacc grammar is:

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

This grammar unambigiously specifies how arbitrary expressions are to be parsed; e.g., e_1`+`e_2`+`e_3 will be parsed as `(`e_1`+`e_2`)+`e_3, and e_1`+`e_2`*`e_3 as e_1`+(`e_2`*`e_3`)`.

Yacc resolves shift/reduce conflicts using precedences and associativity in the following manner. With each grammar rule, it associates the precedence of the righmost terminal symbol (this default may be overwritten using a `%prec` tag, see below). Now, when there is a conflict between shift α and reduce r, α a terminal and r a grammar rule, and both α and r have associated precedences $p(\alpha)$ and $p(r)$, respectively, the conflict is resolved as follows:

- if $p(\alpha) > p(r)$, choose 'shift'.
- if $p(\alpha) < p(r)$, choose 'reduce'.
- if $p(\alpha) = p(r)$, the associativity of α determines the resolution:
  - if α is left-associative: 'reduce'.
  - if α is right-associative: 'shift'.
  - if α is non-associative: 'syntax error'.

Shift/reduce conflicts resolved with precedence will, of course, *not* be reported by Yacc.

Occasionally, it may be necessary to explicitly assign a precedence to a rule using a `%prec` tag, because the default choice (precedence of rightmost terminal) is inappropriate. Consider the following example in which `'-'` is used both as binary and unary minus:

```
%token NUMBER
%left '+' '-'
```

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

`UMINUS` is not an actual input symbol, but serves to give unary minus a higher precedence than any other operator. Note that, by default, the last rule would otherwise have the precedence of (binary) `'-'`.

3.5 Error Handling

This section is concerned with the built-in features Yacc provides for syntactic error handling. By default, when a Yacc-generated parser encounters an errorneous input symbol, it will print out the string `syntax error` via the `yymsg` routine and return to the calling program with the value 1. Usually, an application program will have to do better than this, e.g. print appropriate error messages, and resume parsing after syntax errors. For this purpose, Yacc supplies a flexible error recovery scheme based on the notion of "error rules", cf. [1, section 4.8 and 4.9].

The predefined token `error` is reserved for error handling; it is inserted at places were syntax errors are expected, yielding the so-called error rules. A transition on the error token is never taken during a normal parse, but only when a syntax error occurs. In this case the parser pretends it has seen an `error` token immediately before the offending input symbol. Since in general the current state of the parser may not admit a transition on this fictitious `error` symbol, the parser pops its stack until it finds a suitable state, which admits a transition on the `error` token. If no such state exists, the default error handler is used which prints out `syntax error` and terminates the parse. If there is such a state, the parser shifts the error token on top of the stack, and resumes parsing.

To prevent cascades of error messages, the parser then proceeds in a special "error" state in which other errorneous input symbols are quietly

ignored. Normal parse is resumed only after three symbols have been read and accepted by the parser.

As a simple example, consider the rule

```
stmt  : error { action }
```

and assume a syntax error occurs while a statement is parsed. Then the parser will pop its stack, until the token `error`, as an instance of the rule `stmt : error`, can be accepted, shift the `error` token on top of the stack, reduce by the error rule immediately (executing *action*), and resume parsing in error state. The effect is, that the parser assumes that a statement (to which `error` reduces) has been found, and skips symbols until it finds something which can legally follow a statement.

Similarly, the rule

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

will cause the parser to skip input symbols until a semicolon is found, after which the parser reduces by the error rule and executes *action*.

Note that error rules are not restricted to the simple forms indicated above, but may consist of an arbitrary number of terminal and nonterminal symbols.

Occasionally, the three-symbols resynchronization rule is inadequate. Consider, for example, an interactive application with an error rule

```
input     : error '\n'   { write('reenter last line: ') }
            input        { $$ := $4 }
          ;
```

An error in an input line will then cause the parser to skip ahead behind the following line end, emit the message `reenter last line:`, and read another line. However, it will quietly skip invalid symbols on the new line, until three valid symbols have been found. This can be fixed by a call to the routine `yyerrok` which resets the parser to its normal mode of operation:

```
input     : error '\n'   { write('reenter last line: ');
                           yyerrok }
            input        { $$ := $4 }
          ;
```

There are a number of other Yacc-supplied routines which are useful in implementing better error diagnostics and handling:

- `yychar`: an `integer` variable containing the current lookahead token.
- `yyclearin`: deletes the current lookahead token.
- `yynerrs`: current total number of errors reported with `yymsg`.
- `yyerror`: simulates a syntax error; syntactic error recovery is started, as if the next symbol was illegal.
- `yyaccept`: simulates accept action of the parser (`yyparse` returns 0).
- `yyabort`: aborts the parse (`yyparse` returns 1), as if an unrecoverable syntax error occurred.

These, and other routines are also described in the interface of the Yacc library (cf. 3.3).

Syntactic error handling and recovery is a difficult area; see [1] for a more comprehensive treatment of this topic. The reader may also refer to [4] for a more detailed explanation of the Yacc error recovery mechanism and systematic techniques for developing error rules.

3.6 Arbitrary Value Types

As already noted, the default type for the `$$` and `$`i values is `integer` (cf. 3.1). This type can be changed by putting a declaration

```
%{
type YYSTYPE = some_type;
%}
```

into the definitions section of the Yacc program, prior to inclusion of the header and lexical analyzer file.

Yacc also supports explicit declaration of a (record) value type, by means of a definition

```
%union{
name(s) : type;
⋮
}
```

Such a definition will be translated to a corresponding (Turbo Pascal) record declaration which is put into the header file, and determines the type of stacked `$$` values, as well as of the `yylval` variable (cf. 3.2). "Union tags" of the form `<`*name*`>`, where *name* is the name of a component of the record type, can then be assigned to terminal and nonterminal symbols through `%token` and `%type` definitions, respectively.

Consider, for example, the definition:

```
%union{
       integer_val : integer;
       real_val : real;
      }
```

and the grammar rules:

```
%token INT
%%
expr      : expr '+' expr     { $$ := $1 + $3 }
          | expr '*' expr     { $$ := $1 * $3 }
          | INT               { $$ := $1 }
          /* ... */
```

To assign value type `real` to nonterminal `expr`, and type `integer` to the token `INT`, one might say:

```
%token <integer_val> INT
%type <real_val> expr
```

The effect is, that Yacc will automatically replace references to `$$` and `$`i values by the appropriate record tags, i.e. `$$ := $1 + $3` will be treated as `$$.real_val := $1.real_val + $3.real_val`, and the action `$$ := $1` associated with the third rule will be interpreted as `$$.real_val := $1.integer_val`.

Also, when arbitrary value types are used, Yacc checks whether each value referred to by an action has a well-defined type.

Occasionally, there are values whose types cannot be determined by Yacc easily. This is the case for the `$$` value of an action within a rule, as well as for values `$`i, $i \leq 0$ of symbols in enclosing rules. For such values

the notations `$<`*name*`>$` and `$<`*name*`>`*i* must be used, respectively, where `<`*name*`>` denotes the appropriate union tag.

The `expr` grammar on the distribution disk is a 'real-life' example of the use of arbitrary value types.

3.7 Debugging

As experience shows, debugging a parser can be quite tedious. Although, with some experience, writing a grammar is a quite easy and straightforward task, implementing, for instance, a good error recovery scheme may be quite tricky. Yacc supplies two debugging aids that help verify a parser.

First of all, the parser description (`.lst` file) is useful when determining whether Yacc correctly resolved parsing conflicts, and when tracing the actions of a parser. The `.lst` file gives a description of all generated parser states. For each state, the set of *kernel items* and the *parse actions* are given. The kernel items correspond to the grammar rules processed by the parser in a given state; the underscore `_` in an item denotes the prefix of the rule that has already been seen, and the suffix yet to come. The parser actions specify what action the parser takes on a given input symbol. Here, the period `.` denotes the *default action* that is taken on any input symbol not mentioned otherwise. Possible actions are:

- *shift* an input symbol on top of the parser stack, and change the parser state accordingly;
- *goto* a new state upon a certain nonterminal recognized through the previous reduction;
- *reduce* by a certain grammar rule;
- *accept*, i.e. successfully terminate the parse; and
- *error*, start syntax error recovery.

The default action in a state may either be *reduce* or *error*.

The parser description also lists parsing conflicts and rules that are never reduced (cf. 3.4).

Consider the ambigious grammar:

```
%token NUMBER
%%
expr      : expr '+' expr
```

```
          | expr '*' expr
          | NUMBER
          ;
```

This grammar, when fed into Yacc will cause a number of shift/reduce conflicts. For instance, the description of parser state 5 in the `.lst` file will read as follows:

```
state 5:
        shift/reduce conflict (shift 3, reduce 2) on '*'
        shift/reduce conflict (shift 4, reduce 2) on '+'

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

        $end    reduce 2
        '*'     shift 3
        '+'     shift 4
        .       error
```

As is apparent from this description, the conflicts are caused by missing precedences and associativities of `'+'` and `'*'`. Also, it can be seen that for both `'+'` and `'*'` Yacc chose shift, following the default shift/reduce disambiguating rule.

Now let us resolve the conflicts in the grammar by adding appropriate precedence declarations:

```
%token NUMBER
%left '+'
%left '*'
%%
expr      : expr '+' expr
          | expr '*' expr
          | NUMBER
          ;
```

Now, all conflicts are resolved and for the description of state 5 we get:

```
state 5:

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

        .       reduce 2
```

Thus, Yacc correctly resolved the ambiguities by chosing reduction on any input, which corresponds to the higher precedence and left-associativity of `'*'`.

There are situations in which a parser seems to behave "strangely" in spite of an "obviously correct" grammar. If the problem cannot be found by careful analysis of the grammar, it is useful to trace the actions performed by the parser, to get an idea of what goes wrong.

For this purpose a parser may be compiled with defined conditional `yydebug`, e.g.:

```
yacc parser
tpc parser /Dyydebug
```

When run, the parser will print out the actions it performs in each step (together with parser states, numbers of shifted symbols, etc.), which can then be followed on a hardcopy of the parser description.

Debug messages are printed via the `yydebugmsg` routine (cf. 3.3). You may also wish to tailor this routine to your target application, such that it prints more informative messages.

Of course, the discussion above was rather sketchy. A more detailed treatment of these topics, however, would require a presentation of the LALR parsing technique, which is well beyond the scope of this manual. The reader instead is referred to [1, sections 4.7 and 4.8] for more information about the LALR technique. Also, [4] gives a more detailed explanation of (UNIX) Yacc parse tables and debug messages, which can mostly be applied to Turbo Pascal Yacc accordingly.

3.8 Yacc Language Grammar

This section specifies the Yacc language syntax, as a Yacc grammar. Actually, the Yacc language is more naturally expressed by an LR(2) grammar;

the difficulty is to decide on the base of one-symbol lookahead whether an identifier at the end of a rule is followed by a colon, in which case it starts the next rule. Thus, we distinguish the token `C_ID` (an identifier followed by a colon) and "ordinary" identifiers `ID`. It is assumed to be the task of the lexical analysis to determine to which of these two classes an identifier belongs.

The following grammar has been abstracted from the Turbo Pascal Yacc grammar actually used to implement Turbo Pascal Yacc.

```
%token
  ID            /* identifier; also literals enclosed in quotes */
  C_ID          /* identifier followed by a colon */
  NUMBER        /* nonnegative integers */
  TOKEN LEFT RIGHT NONASSOC TYPE START UNION PREC
                /* reserved words: %token, etc. */
  SEP           /* separator %% */
  LCURL RCURL   /* curly braces %{ and %} */
  ',' ':' ';' '|' '{' '}' '<' '>'
                /* single character literals */

%start spec

%%

spec            : defs SEP rules aux_procs
                ;

/* auxiliary procedures section: ************************/

aux_procs       : /* empty: aux_procs is optional */
                | SEP { copy the rest of the file }
                ;

/* definitions section: *********************************/

defs            : /* empty */
                | defs def
                ;
```

```
def             : START ID
                | UNION '{' { copy the union definition } '}'
                | LCURL { copy Turbo Pascal tie-in } RCURL
                | TOKEN tag token_list
                | LEFT tag token_list
                | RIGHT tag token_list
                | NONASSOC tag token_list
                | TYPE tag nonterm_list
                ;

tag             : /* empty: union tag is optional */
                | '<' ID '>'
                ;

token_list      : token_num
                | token_list token_num
                | token_list ',' token_num
                ;

token_num       : ID
                | ID NUMBER
                ;

nonterm_list    : nonterm
                | nonterm_list nonterm
                | nonterm_list ',' nonterm
                ;

nonterm         : ID
                ;

/* rules section: ***************************************/

rules           : rule1
                | LCURL { copy Turbo Pascal tie-in } RCURL rule1
                | rules rule
                ;

rule1           : C_ID ':' body preced
```

```
                ;

rule            : rule1
                | '|' body preced
                ;

body            : /* empty */
                | body ID
                | body action
                ;

action          : '{' { copy action, substitute $$, etc. } '}'
                ;

preced          : /* empty */
                | PREC ID
                | PREC ID action
                | preced ';'
                ;
```

3.9 Additional Features, Implementation Restrictions and Bugs

For backward compatibility, Turbo Pascal Yacc supports all additional language elements entitled as 'Old Features Supported But not Encouraged' in the UNIX manual:

- literals delimited by double quotes and multiple-character literals.
- `\` as a synonym for `%`, i.e. `\\` is `%%`, `\left` is `%left`, etc.
- other synonyms: `%<` = `%left`, `%>` = `%right`, `%binary` = `%2` = `%nonassoc`, `%term` = `%0` = `%token`, `%=` = `%prec`.
- actions of the form `={...}` and `=`*single statement*`;`.
- host language tie-ins (`%{...%}`) at the beginning of the rules section (I think that this last one is really a *must*).

See the UNIX Yacc manual for further information.

As with Lex, Yacc poses some restrictions on internal table sizes for the source grammar and the constructed parser; these are printed out by

Yacc together with statistics about actual table space requirements, after a successful translation of a grammar. Also, make sure that enough main memory is available.

The default size of the parser stack is `yymaxdepth=1024` (cf. 3.3) which should be sufficient for any average application, but may also be enlarged (and shrinked) as needed. Note that right-recursive grammar rules may increase stack space requirements; thus it is a good idea to use left-recursive (and left-associative) rules wherever possible.

Standard (UNIX) Yacc has a bug that causes some (correct) error recovery schemes to hang in an endless loop, see [4]. This bug should be fixed in the Turbo Pascal implementation, at the cost of slightly increased parse table sizes.

Yes, there *is* (at least) one bug in Turbo Pascal Yacc, namely that `%union` definitions (cf. 3.6) are translated to simple Pascal record types. They should be *variant* records instead. This will be fixed in the next release, if there ever is one. Note that this bug does not affect the proper functioning of the parser; it merely increases memory requirements for the parser's value stack. Anyhow, you may work around this by using `%union` definitions of the (Pascal variant record) form

```
%union { case integer of
           1: ( ... ) ;
           2: ( ... ) ;
            ...
       }
```

A final remark about the efficiency of Yacc-generated parsers is in order. The time needed to parse an input of length n is proportional to n. Although this may not convince everyone (Lex makes a similar claim, however most Lex-based analyzers seem to be considerably slower than hand-crafted ones), my experience is that Yacc-generated parsers are in fact *fast*, at least efficient enough for most applications (such as Turbo Pascal Yacc itself). The major bottleneck for compilation speed almost never seems to be the parser, but almost always the lexical analyzer, see [5, section 6.2]. Furthermore, one always has to consider that manual implementation of parsers is usually much more costly, compared to the use of a parser generator.

Personally, I prefer a parser generator, because I'm really a lazy programmer; and if something seems not to be running at optimal speed, so

what? We can always sit back and wait for still more efficient hardware to come (just kidding).

Conclusion

The Turbo Pascal Lex and Yacc versions described in this manual have been designed and tested carefully. I have used them myself to implement, among other applications: a lexical analyzer and parser for Pascal (using a public domain ISO Level 0 grammar, also included in the distribution); Turbo Pascal Yacc itself, using bootstrapping; and a term rewriting system compiler.

Also, quite a lot of smaller text and data processing and conversion routines have been implemented by the author, and others, using these programs.

Personally, I feel that these tools are quite convenient and useful, and can safe a lot of trouble and time in software development, although they surely could still be improved in one direction or the other.

Compiler construction tools are not only useful for the compiler writer, but can also be applied in the development of almost any other software tool that, in some sense, defines an input language. Also, the use of such utilities facilitates rapid prototyping, and enables the programmer to clarify language design issues in early stages of software projects.

Turbo Pascal Lex and Yacc, as a starting point, bring to the Turbo Pascal programmer some of the merits of theoretically founded compiler technology, and thus may facilitate some of his work in trying to produce good, and reliable software.

Author's address: Albert Gräf, FB Mathematik, Johannes Gutenberg-Universität Mainz, 6500 Mainz (FRG). Email: `Graef@DMZRZU71.bitnet`.

References

[1] Aho, Alfred V.; Ravi Sethi; Jeffrey D. Ullman: *Compilers : principles, techniques and tools.* Reading, Mass.: Addison-Wesley, 1986.

[2] Johnson, S.C.: *Yacc – yet another compiler-compiler.* Murray Hill, N.J.: Bell Telephone Laboratories, 1974. (CSTR-32).

[3] Lesk, M.E.: *Lex – a lexical analyser generator.* Murray Hill, N.J.: Bell Telephone Laboratories, 1975. (CSTR-39).

[4] Schreiner, A.T.; H.G. Friedman: *Introduction to compiler construction with UNIX.* Prentice-Hall, 1985.

[5] Waite, William M.; Gerhard Goos: *Compiler construction.* New York: Springer, 1985. (Texts and monographs in computer science).

Appendix: Lex and Yacc Manual Pages

Name Lex – lexical analyzer generator (MS-DOS/Turbo Pascal version)

Synopsis `lex` *lex-file-name*[`.l`] [*output-file-name*[`.pas`]]

Description Lex compiles the regular expression grammar contained in *lex-file-name* (default suffix: `.l`) to the Turbo Pascal representation of a lexical analyzer for the language described by the input grammar, written to *output-file-name* (default suffix: `.pas`; default: *lex-file-name* with new suffix `.pas`).

For each pattern in the input grammar an *action* is given, which is an arbitrary Turbo Pascal statement to execute when the corresponding pattern is matched in the input stream.

The lexical analyzer is implemented as a table-driven deterministic finite automaton (DFA) routine named `yylex`, declared as follows:

```
function yylex : integer;
```

The return value of `yylex` may be 0, denoting end-of-file; all other return values are defined by the programmer and set through appropriate actions.

The `yylex` routine can be compiled with the Turbo Pascal compiler (`tpc` or `turbo`). It is to be called in the context of a Turbo Pascal main program *using* the `LexLib` unit (which can be a Yacc-generated parser or any other program in a separate file, or incorporated into the input specification, and is to be supplied by the programmer).

Example A simple Lex program that counts words in an input file (obtained from standard input) can be implemented as follows:

```
     uses LexLib;
     var count : integer;
%%
[^ \t\n]+    inc(count);
.            |
\n           ;
%%
begin
  count := 0;
```

```
  if yylex=0 then writeln('word count: ', count)
end.
```

To compile and run this program, issue the following commands (assuming the Lex program to be in file `wordcount.l`):

```
lex wordcount
tpc wordcount
wordcount <input-file
```

Diagnostics In case of syntactic or semantic errors in the source file, Lex displays source line numbers and contents, error position and error message; a copy of the error messages is written to the file *lex-file-name* with new suffix `.lst`.

Name Yacc – yet another compiler-compiler (MS-DOS/Turbo Pascal version)

Synopsis `yacc` *yacc-file-name*[`.y`] [*output-file-name*[`.pas`]]

Description Yacc compiles the BNF-like grammar contained in *yacc-file-name* (default suffix: `.y`) to the Turbo Pascal representation of an LALR(1) parser for the specified language, written to *output-file-name* (default suffix: `.pas`; default: *yacc-file-name* with new suffix `.pas`).

Also, it generates a *header file* (*output file name* with new suffix `.h`) containing declarations to be shared between parser and the lexical analyzer routine `yylex` (discussed below), and a *report file* (*yacc-file-name* with new suffix `.lst`) that contains a description of the generated parser.

The grammar rules in the specification can be augmented with *actions*, Turbo Pascal statements to execute when an instance of the corresponding grammar rule has been matched in the input.

The parser is implemented as a table-driven deterministic pushdown-automaton routine `yyparse`, that performs a non-backtracking, bottom-up shift/reduce parse. `yyparse` is declared as follows:

```
function yyparse : integer;
```

The return value of this function is either 0 (normal termination of the parse) or 1 (exception occurred during the parse, e.g. stack overflow, unrecoverable syntax error, or programmer action called `yyabort`).

The `yyparse` routine can be compiled with the Turbo Pascal compiler (`tpc` or `turbo`). It is to be called in the context of a Turbo Pascal main program *using* the `YaccLib` unit and a lexical analyzer routine

```
function yylex : integer;
```

which can also be prepared with Lex; these must be supplied by the programmer. The header file Yacc generates summarizes declarations to be shared between parser and the `yylex` routine.

If the `yyparse` routine is compiled with defined conditional `yydebug`, i.e.

`tpc` *filename* `/Dyydebug`

`yyparse` will trace all parsing actions on standard output.

Example The sample desktop calculator supplied on the distribution disk consists of the main program and input grammar in file `expr.y` and a lexical analyzer in the Lex source file `exprlex.l`. It can be compiled and run by issuing the commands:

```
yacc expr
lex exprlex
tpc expr
expr
```

To trace the steps made by the parser, compile `expr.pas` with

```
tpc expr /Dyydebug
```

Diagnostics When encountering syntactic or semantic errors in an input grammar, Yacc gives diagnostics on standard output and in the report file (*yacc-file-name* with new suffix `.lst`).

Upon successful compilation of the input grammar, Yacc will report the number of shift/reduce and reduce/reduce conflicts encountered when constructing the parser (if there are any); also, Yacc will report the number of grammar rules that are never used in a reduction, and issue warnings when nonterminal grammar symbols do not appear on the left side of at least one grammar rule. Such items, in particular: shift/reduce and reduce/reduce conflicts, are discussed in more detail in the report (`.lst`) file.