Notes about *lex* and *yacc*

Pablo Nogueira Iglesias

December 26, 1999

Contents

1 Format of *lex* and *yacc* input files

Both *lex* and *yacc* input files must have the following structure:

```
definitions
%%
rules
%%
user supplied routines
```

In sections `definitions` and `rules`, lines that start with whitespace are copied verbatim to the generated `lex.yy.c` file—this might change from one particular implementation to another—. Useful for placing comments in both sections.

`%{ %}` are allowed in both `definitions` and `rules`. If placed in `definitions`, *lex* puts the code enclosed between `%{ %}` at the very beginning of the `lex.yy.c` file, before the definition of `yylex()`. Useful to declare prototypes of functions used in the `user routines` section, for header file inclusion, for global variable declarations, etc.

2.1 Definitions

In this section we can give names to *parts* of regular expressions that will be used in the rules section for the purpose of reusability and readability. These definitions are *literally* substituted in the rules section.

2.2 Rules

Lines at the beginning of this section are placed at the beginning of `yylex()`. They must contain local variable declarations or initialisation code—but better use `definitions` for that purpose.

Rules follow the syntax: `regexp whitespace { statements }`. Braces are optional, they are required when the statements take more than one line.

Context: / is the *trailing context*. For example, the rule `abc/de` means "`abc` only when followed by `de`"; the latter is not included as part of `yytext`. It is equivalent to `abcde yyless(3);`. If input is `abdeabcde` at the point in which `abc` is recognised, the next character in the input to be recognised will be `d`. Also, ∧ and `$` are used for context, they are not part of part of `yytext`. `$` is equivalent to `/\n`.

2.3 Operation of `yylex()`

When *lex* compiles the input specification, it generates the C file `lex.yy.c` that contains the routine `int yylex(void)`. This routine reads the input trying to match it with any of the token patterns specified in the rules section. On a match, the associated action is executed. If there is more than one match, the action associated with the pattern that matches more text (included context) is executed. If still there are two or more patterns that match the same amount of text, the action associated with the pattern listed first in the specification file is executed. If no match is found, the default action `ECHO` is executed (see Section 2.4). The input text (lexeme) associated with the recognised token is placed in the global variable `yytext`.

Example: given these rules

```
";"      /* ignore separator */
int      ECHO;
[a-z]+   printf("ID");
```

if the input is "`int;...`" then `int` is echoed, but if the input is "`integer;...`" then `ID` is printed.

When specifying a lexer, take this mechanism into account. Place patterns from the more restrictive to the more general. In this example

```
{ID}     return ID_TOKEN;
"IF"     return IF_KEYWORD;
```

the lexer will always return `ID_TOKEN` for any "IF" string encountered. The rules should be placed form specific to general:

```
"if"      return IF_KEYWORD;
{ID}      return ID_TOKEN;
```

Another issue to have in mind when designing a lexer with *lex* is that *the speed of a lex scanner is independent of the number and complexity of the patterns specified*—see Levine et al. (1995), p. 94. Take for example a case-insensitive programming language with reserved keywords. We may declare a token for each keyword and another for identifiers in the following fashion

```
%%
"IF"   return IF;
"if"   return IF;
"THEN" return THEN;
"then" return THEN;
"ELSE" return ELSE;
"else" return ELSE;
...

{ID}   { yylval.strval = strdup(yytext); return NAME; }
%%
```

This solution will work slightly faster than a solution based on just one pattern for all names and a keyword symbol table lookup to determine the kind of token, due to the extra lookup processing.

2.4 Avoiding the ECHO default action

When `yylex()` cannot find a pattern that matches the input, it just writes `yytext` to `yyout`. Such default action is called `ECHO` and is equivalent to `fprintf(yyout,"%s", yytext);`.

In many cases this can be undesirable, so it might be convenient to write a very last rule to do some action with all the unexpected input. A simple example can be

```
.  return yytext[0];
```

Now the parser program will have to take care of lexical errors concerning unexpected input.

2.5 Inter-operation with *yacc* or otherwise

When the lexer analyses the input, for each recognised token it carries out its associated action. Such action can be any statement, so *lex* can be used for many purposes, not only lexical analysers.

The only way to return from `yylex()` is to place a `return` statement in an action. The value returned *must* be an integer (either a literal or a symbolic constant), otherwise a type error or a type coercion will occur. If no `return` statement is written, execution continues inside `yylex()`, recognising input and carrying out the actions until the end of input is reached (this is useful for example to skip comments, etc.) Every time `yylex()` is called, it picks up where it left off.

The integer returned by `yylex()` must be a token code-number. When interoperating with *yacc*, such codes must be known to both `yylex()` and `yyparse()`. *yacc* can create the `y.tab.h` file (see Section 3.7) containing one `#define` for each symbol specified as `%token` in the *yacc* file. Such symbols cannot be C reserved keywords.

This is a typical (but system-dependent) invocation:

```
$ yacc -d file.y
$ lex file.c
$ gcc -o parser y.tab.c lex.yy.c -ly -ll
```

Note that *yacc* is invoked first.

2.6 Variables and functions used in lex

The routine `yylex()` provides information by way of global variables and the value it returns, but also by way of macros, functions, and options. Figure 1 shows the basic variables and functions.

`yyin`, `yyout`	These are `FILE*`s to the input and output file respectively. Both are accessible and can be assigned to a value other than stdin and stdout (see Section 2.9).
`input()`	Read next character from `yyin`. This is the function invoked by `yylex()` to read the input.
`output()`	Write `yytext` to `yyout`. This is the function invoked by `yylex()` to do an `ECHO`. This function is not supported when generating C++ scanners (see man pages).
`yyless(int n)`	Keeps the *n*th first characters of `yytext` and puts the rest back to `yyin`. For example: `abc/de` is equivalent to `abcde yyless(3);`.
`yymore()`	Keep this token's lexeme in `yytext` when another pattern is matched.
`yywrap()`	Used for multiple input files. Specifies what to do when the `EOF` token is recognised. *flex*'s option `%option noyywrap` instructs the lexer to scan only one file (generates default `yywrap()`), otherwise the funcion must be specified by user. See also `yyrestart()` in documentation.
`yytext`	Array of or pointer to (see man) char where *lex* places the current token's lexeme. The string is automatically null-terminated. It can be modified but not lengthened.
`yyleng`	Integer that holds `strlen(yytext)`.
`yylineno`	Integer that keeps count of lines read. It is maintained automatically by `yylex()` for each `\n` encountered. In *flex*, it is an optional feature that must be specified using `%option yylineno`, for there is a loss of performance.

Figure 1: *lex*'s global variables and functions

2.7 Changing the input buffer size

***lex*:**

```
#undef  YYLMAX
 #define YYLMAX (size)
```

***flex*:**

```
setbuf(int size) {
   yy_current_buffer = yy_create_buffer(yyin,size);
}
```

2.8 Taking input from strings

Normally *lex* reads from `FILE* yyin`, but sometimes we might want to read input from a string or array of char in memory. All versions of *lex* allow this. Here we discuss *flex*'s: redefine the macro `YY_INPUT` used to read blocks of data. Its syntax is

```
YY_INPUT(buffer, result, maxsize)
```

where `buffer` is the character array, `result` is a variable in which to store the number of characters actually read (e.g. `yytext`), and `maxsize` is the size of the buffer. To read from a string, have your version of `YY_INPUT` copy data from your string buffer. Here is an example:

```
%{
#undef YYINPUT
#define YYINPUT (b,r,ms) (r = my_yyinput(b,ms))
%}
...
extern char myinput[];    /* char array buffer */
extern char *myinputptr;  /* current position in myinput */
extern int myinputlim;    /* end of data */

int my_yyintput(char *buf, int maxsize) {
    int n = min(maxsize, myinputlim - myinputptr);

    if (n>0) {
        memcpy(buf, myinputptr, n);
        myinputptr += n;
    }
    return n;
}
```

2.9 Taking input from pipes

Imagine we want a scanner for C source files but we want to make use of the C preprocessor, which can remove comments and perform file inclusion, generating line number stamps of the form `# lineno filename`. To avoid renaming of files by using temporaries, an elegant solution that makes use of pipes is presented in Schreiner et al (1995) and summarised here.

```
#undef CPP
#define CPP "/lib/cpp"

sprintf(buf, "%s %s", CPP, args);
if (yyin = popen(buf, "r"))
  /* Able to open pipe */
else ...
```

Since line numbering is altered by preprocessing, a way to correctly update `yylineno` is to include a pattern in the scanner file that updates `yylineno` according to line stamps, such as

```
^"#"[ \t]*[0-9]+([ \t]+.*)?\n   sscanf(yytext, "# %d", &yylineno);
```

2.10 Error control

In general, error control consists of three related activities: detection, reporting, and recovery—see p. 165-169 of Aho et al. (1990).

1. **Detection.** In *lex*, there is no implicit error detection. The default action is `ECHO`. Error detenction must be done explicitly, writing patterns for illegal input or as mentioned in Section 2.4. The following is an example of accepting illegal input to detect errors:

   ```
   \"[^\"\n]*\"     { yylval.string = strdup(yytext); return DQS; }
   \"[^\"\n]*$      { warning("unterminated string"); }
   ```

2. **Reporting.** The detected errors must be reported in a *precise* and *detailed* fashion. Precise means that the report should include a reference to the exact location where the error occurred. Detailed means that the report message gives enough information about the type of the error.

   In *lex*, reporting can be done in the actions associated to the patterns for illegal input. The following is a code that may be added to any lexer. Its purpose is to report informative error messages (Taken from Levine et al. (1995) p. 246).

```
%{
#include <string.h>
char linebuf[...];   /* Put a size of line */
int yylineno=0, tokenpos;
%}
%%

\n.*   { strcpy(linebuf, yytext-1);  /* Save next line */
         yylineno++;
         tokenpos = 0;
         yyless(1);   /* Flush back all but \n to input */
       }
%%

void yyerror(char *s)
{
   printf("%d: %s:\n%s\n", yylineno, s, linebuf);
   printf("%*s\n", 1+tokenpos, "^");    /* '*' pads w/ whitespace */
}
```

The pattern `\n.*` will take precedence over all others, for it is listed first and it will match the largest amount of input text possible. Every line will be scanned at least twice: one to save the entire line into the variable `linebuf` and again to scan each token in that line.

Variable `tokenpos` is used to keep the position in the line of the mismatched token. Its value must be updated properly and reset to 0 only in the action for `\n.*`. A possible wat to update its value could be to place the statement `tokenpos += yyleng;` as the last statement of each pattern's action.

Here is a simpler definition of `yyerror()` without `tokenpos` is

```
  printf("%d: %s %s in %s\n", yylineno, s, yytext, linebuf);
```

in which the offending token text is printed along with the message and the line number.

3. **Recovery.** It is undesirable to stop processing the input when an error is detected. It should be possible to do as much processing as possible to shorten the edit-compile-test cycle. Error recovery is enhanced with language design (e.g. the use of terminators such as `;` that serve as synchronisation points).

   Since there is no error detection in *lex*, it does not have an error recovery mechanism either. Recovery must be done explicitly in the actions associated to illegal input patterns. A typical recovery mechanism would be to discard `yytext` and to wait for the next matched pattern.

3 YACC

yacc is a parser generator that takes an input file with an attribute-enriched BNF grammar specification, and generates the output C file `y.tab.c` containing the function `int yyparse(void)` that implements its table-driven LALR(1) parser. This function automatically invokes `yylex()` every time it needs a token to continue parsing. Such function can be provided by the user or by a *lex*-generated scanner.

Section `definitions` contains token declarations, the value-stack type, the type declaration of attributes associated to non-terminals and tokens, and the precedences and associativity of tokens.

`%{ %}` are allowed only in `definitions`. Useful to declare prototypes of routines used in the `user routines` section, for header file inclusion, for declaring of global variables, etc.

The `rules` section contains the BNF grammar specification together with the semantic actions for each production rule. It is actually a syntax-directed translation scheme. It is customary to write tokens in uppercase and non-terminals in lower case.

The grammar is automatically checked by *yacc* to assure that:

1. There are rules (productions) for all non-terminals.
2. All non-terminals must be reachable from the axiom.
3. The grammar is not ambiguous
4. The grammar is LALR(1) parsable.

The *yacc* program alerts us if any of these conditions do not hold. The first two are trivial to arrange.

3.1 Grammar ambiguity

If the grammar is ambiguous, it can be disambiguated using disambiguating rules. *yacc* uses two disambiguating rules by default—hence, there are ambiguous grammars for which *yacc* is able to generate a parser—, and also allows the user to disambiguate the grammar by declaring the precedence and associativity of tokens. Production rules take the precedence of the rightmost token on their right hand side. If a production has no tokens then it has no precedence of its own. If the rightmost token does not provide the right precedence, it can be overriden with `%prec T` where `T` is the token whose precedence we want assigned to the rule (Aho et al. (1990) p . 271).

Example: dangling `ELSE`

$$\begin{array}{rcll} stmts & \longrightarrow & stmt & \\ & | & stmts\ stmt & \\ stmt & \longrightarrow & \mathtt{IF}\ expr\ \mathtt{THEN}\ stmts & (1) \\ & | & \mathtt{IF}\ expr\ \mathtt{THEN}\ stmts\ \mathtt{ELSE}\ stmts & (2) \\ & | & other.stmt & \end{array}$$

There is a shift/reduce conflict in the grammar: when the stack contains the parsing state [`IF` *expr* `THEN` *stmts*] and the lookahead token is `ELSE`, *yacc* can either reduce by rule (1) or shift the `ELSE` and attempt to reduce later by rule (2). Notice that `ELSE` $\in$ FOLLOW(*stmts*). If instead than using the default disambiguating rule we want to disambiguate the grammar explicitly, we must declare the following precedences:

```
%nonassoc THEN
%nonassoc ELSE
```

which gives higher precedence to the token `ELSE`. Thus, rule (1) has lower precedence than rule (2). The precedence of the token to shift must be higher than the precedence of the rule to reduce. (See Aho et al. (1990) p. 257-259.)

Precedence and associativity allows the user to keep a readable grammar and indeed helps *yacc* to build fast parsers that perform lesser reductions. See Aho et al. (1990) p. 254-256 for a typical example. On the other hand Levine et al. (1995) recomends the use of precedence and associativity only in expressions. It is usually convenient to rewrite the grammar. If *yacc* cannot understand the grammar, usually neither can people.

By default, a shift/reduce conflict is resolved in favour of shift (try to match as much input as possible), and a reduce/reduce conflict is resolved in favour of the rule listed first in the *yacc* specification. This may cause troubles if the specification text is rearranged and rules are moved!

Aho et al. (1990) p. 270, Johnson (1978) Sections 5 and 6 (especially p. 16), and Levine et al. (1995) Chp. 7, provide a detailed discussion.

3.2 What *yacc* cannot parse

There are unambiguous grammars that *yacc* cannot parse. This is due to the limitations of the LALR(1) technique. For example, the following unambiguous grammar is not LALR(1)-parsable:

$$\begin{array}{lcl} S & \longrightarrow & A\ a\ b \\ & | & B\ a\ c \\ A & \longrightarrow & d \mid e \\ B & \longrightarrow & d \mid f \end{array}$$

A LALR(1) parser fails to parse the sentence dab, for two lookahead symbols are needed to select the right production for reduction. Indeed, after shifting d, the lookahead token is a. The parser cannot tell whether to reduce d by $A \rightarrow d$ or by $B \rightarrow d$ since a belongs to both FOLLOW(A) and FOLLOW(B), which amounts to a reduce/reduce conflict.

3.3 Left recursion

The grammar's productions must be left recursive instead than right recursive. The reasons are explained in Johnson (1978) p. 19. Intuitively, we can see that right-recursive lists make their two productions appear on the same *item*:

$$\begin{array}{llcl} 1: & list & \longrightarrow & \texttt{TOKEN} \\ 2: & & | & \texttt{TOKEN}\ list \end{array}$$

Suppose that $list$ is the axiom; the grammar is enlarged with production 0:

$$\begin{array}{llcl} 0: & list' & \longrightarrow & list \end{array}$$

The initial state is the closure of $\{[0,0]\}$, that is, $\{[0,0][1,0][2,0]\}$. Computing the $list$-successor and the `TOKEN`-successor of this state yields two new states:

$$\begin{array}{lclcl} goto(\ \{[0,0][1,0][2,0]\},\ list) & = & closure(\{[0,1]\}) & = & \{[0,1]\} \\ goto(\ \{[0,0][1,0][2,0]\},\ \texttt{TOKEN}) & = & closure(\{[1,1][2,1]\}) & = & \{[1,1][2,1][1,0][2,0]\} \end{array}$$

When the table is built, $Action[\ \{[1,1][2,1][1,0][2,0]\},\ \texttt{TOKEN}]$ will be a shift to the same state

$$goto(\ \{[1,1][2,1][1,0][2,0]\},\ \texttt{TOKEN}) \quad = \quad closure(\{[1,1][2,1]\}) \quad = \quad \{[1,1][2,1][1,0][2,0]\}$$

pushing `TOKEN` into the stack as well. There exists the danger of the parser's stack being filled with `TOKEN`s if the list is lengthy.

With left-recursive lists, this danger no longer exists, for a reduction is always performed when the first `TOKEN` of the list is encountered:

$$\begin{array}{llcl} 0: & list' & \longrightarrow & list \\ 1: & list & \longrightarrow & \texttt{TOKEN} \\ 2: & & | & list\ \texttt{TOKEN} \end{array}$$

$$\begin{array}{lclcl} goto(\ \{[0,0][1,0][2,0]\},\ list) & = & closure(\{[0,1][2,1]\}) & = & \{[0,1][2,1]\} \\ goto(\ \{[0,0][1,0][2,0]\},\ \texttt{TOKEN}) & = & closure(\{[1,1]\}) & = & \{[1,1]\} \end{array}$$

When the table is built, $Action[\ \{[1,1]\},\ \texttt{TOKEN}]$ will be a reduction—the state contains a completed item—, which will pop the `TOKEN` from the stack and push the *list* non-terminal and the new state.

3.4 Value stack type

Along with the parsing state stack, *yacc* maintains a parallel stack of attribute values associated with each token and non-terminal in the parsing stack. The type of an attribute is defined as `YYSTYPE`, which is `int` by default, but can be redefined within the specification in two ways:

1. Redefining YYSTYPE to a *single type* name as in: `#define YYSTYPE double`. Being a preprocessor definition, for composite type names we must use `typedef`:

   ```
   typedef struct att_type {
      char *sval;
      int   ival;
   };
   #define YYSTYPE att_type
   ```

2. Declaring the value stack type with a `%union` declaration within the *yacc* specification file:

   ```
   %union {
      char *sval;
      int   ival;
   }
   ```

   letting *yacc* create the appropriate `#define`s in the file `y.tab.h` (see Section 3.7). It will also create the global variables `yylval` and `yyval` of type `YYSTYPE` (see Figure 2).

3.5 Computing attributes

In semantic actions, the values of attributes associated to the non-terminal on the left hand side of a production can be computed in terms of the values of attributes associated with the grammatical symbols on the right hand side. The symbol `$$` denotes the former and `$`*i* denotes the attribute value of the *i*th grammatical symbol on the right hand side. The default action is {`$$=$1;`}. Semantic actions are usually placed at the end of a production, for *yacc* generates an ascent or bottom-up parser, and semantic actions should implement a syntax directed definition with *synthesised attributes* only. The actions are thus executed before reduction, and the attribute values of the grammatical symbols on the right hand side will be popped from the value stack and replaced with the attribute value of the left hand side non-terminal.

Nevertheless, evaluation of *inherited attributes* is possible as long as the syntax directed definition is an *l-attributed* definition, by means of accessing atribute values deep in the stack, as described in Aho et al. (1990), Section 5.6 pp. 318–325. The same technique allows the use of interleaved actions in productions, as in $A \to a\ \{\ldots\}\ B\ \{\ldots\}\ C$. In fact, *yacc* creates anonymous markers replacing the former production with

$$\begin{array}{rcl} A & \to & a\ M_1\ B\ M_2\ C \\ M_1 & \to & \lambda\ \{\ldots\} \\ M_2 & \to & \lambda\ \{\ldots\} \end{array}$$

Interleaved actions can have attributes associated to them (associated to the non-terminal marker). Both for these and for "inherited" attributes deep in the stack, *yacc* has no easy way of knowing their type, so it must be specified explicitly. The notation `$<`*type*`>$` explicitly declares the type of the attribute value of the interleaved action,[1] and `$<`*type*`>`*n* declares the type of the attribute value located on $tos - n$ ($n : 0, -1, \ldots$), where *tos* is the top of the value stack. The following example illustrates the use of inherited attributes: C-like declarations have the form

$$\begin{array}{rcl} \mathit{decl} & \to & \mathit{type}\ \mathit{decl\_list}\ \texttt{';'} \\ \mathit{dec\_list} & \to & \texttt{ID} \\ & | & \mathit{decl\_list}\ \texttt{','}\ \texttt{ID} \\ \mathit{type} & \to & \texttt{INT} \mid \texttt{VOID} \mid \ldots \end{array}$$

The type of each identifier on the list is specified first, so *decl_list* should have an associated inherited attribute *t* (for "type"), whose value is taken from the value of *type* and that is available for each identifier

[1]Do not confuse `$<..>$` with `$$`. The type of the left hand side non-terminal attribute is known—it has been declared with a `%type`, so only `$$` stands for that attribute.

on the list when creating its symbol table entry:

```
      decl  →  type
                 {dec_list.t := type.t} /* Inherited attr. */
               decl_list ';'
   decl_list  →  ID
                 { st_insert(ID.lexeme, decl_list.t) }
   decl_list  →  decl_list ',' ID
                 { st_insert(ID.lexeme, decl_list.t) }
```

Function `st_insert(id,t)` creates a symbol table entry for token `ID` with lexeme `id` and type `t`. To implement this l-attributed definition using *yacc* we need only to see that the attribute value of *type* will be on top of the value stack before the first identifier on the list is read, so that after shifting it, the attribute of *type* will be available as an offset from the top of the value stack:

```
%token ID
%token INT
%token VOID
%union {
  enum Types tval;
  char *sval;
}
%type <tval> type decl_list
%type <sval> ID
%%
 type : INT   { $$ = INT_TYPE;}
      | VOID  { $$ = VOID_TYPE;}
      ;
 decl : type decl_list ';'
      ;
 decl_list : ID
               { st_insert($1, $<tval>0); $$ = $0; }
             | decl_list ',' ID
               { st_insert($3, $1); }
```

Note how the type `tval` has been explicitly specified for the inherited attribute in the action.

Implementing inherited attributes this way is more elegant and more readable than using global variables to pass attribute information. Nothing prevents us from having a global variable *t* that holds the type of the current declaration and whose value is set in the actions for *type* and read in the actions for *decl_list*, but appart from being an *ad-hack* solution, parse errors and error recovery can create havoc when discarding input and partially parsed phrases.

Interleaved actions are somewhat similar, they are usually used for housekeeping before parsing certain phrases or for creating new attribute values depending on the partially parsed phrase. For example, logical operators in C must be lazily evaluated (in short-circuit), so that code generation to assembly shoud create appropriate labels and produce branching instructions—the following example assumes a stack discipline of expression evaluation:

```
 %type <tval> expr
 expr : expr AND
         { char *l = new_label();
           emit("pop");
           emit("jump z"); emit(l);
           $<sval>$ = l;
         }
```

```
    {
      emit("pop");
      emit($<sval>3); emit(":");
      emit("push");
      $$ = check_type($1, $4);
    }
```

In the second action, `$<sval>3` refers to the attribute value of the first action (or its marker), which is the 3rd element in the right hand side of the production. Note that its type has been explicitly specified.

Interleaved actions require special care: one should give value to `$$` only in a rightmost action. Recall that `$$` is pushed on the value stack after reduction. If placed in an interleaved action, attribute values of elements to the right of such action will be pushed on top of it, reduction will have trouble when popping from the value stack as many attributes as right hand side elements, and it is undefined what value will be pushed for the non-terminal.

3.6 Variables and functions used in *yacc*

Figure 2 shows the basic functions and variables.

3.7 Invocation

- The `-d` option generates the file `y.tab.h` that contains for each token constant `TOKEN`*i* a line alike the following

  ```
  #define TOKEN1 447
  ```

  This way, the token codes returned by *lex* are those expected by *yacc*. The header file imust be included in the *lex* specification at the `%{%}`s.

- For execution of the parser in tracing mode, `YYDEBUG` must be defined at compile time. A possible solution is

  ```
  gcc -DYYDEBUG y.tab.c lex.yy.c main.c error.c -ll
  ```

  If we have a `.y` file that contains its own `main()`, we can call *yacc* with the `-t` (trace) flag which automatically defines `YYDEBUG` in the output C file. Defining `YYDEBUG` is not enough; for *yacc* to print trace information (states pushed, rules used in reductions,...) variable `yydebug` must be set to a non-zero value before calling `yyparse()`.

3.8 Error control

As mentioned in Section 2.10, error control consists of three related activities:

1. **Detection.** In *yacc*, detection is accomplished when given the current parsing state in top of the stack and the next input token, no shift or reduce action is possible. Empty entries are taken as error entries.

2. **Reporting.** In *yacc*, reporting is done via calls to `yyerror()`. It will print the message "parse error" unless the constant `YYERROR_VERBOSE` is defined in the specification file. In that case it will print the next expected terminal or non-terminal for the current state.

3. **Recovery.** A few error recovery techniques for parsers are discussed in Aho et al. (1990) p. 168-169. Here we mention two that are related to *yacc*'s error recovery mechanism.

yacc* by writing productions for erroneous syntactic constructions. This way detection, reporting and recovery is straightforward. For a given legal production

$$A \quad : \quad \alpha \quad \{\ldots\}$$

we might add more rules that will detect illegal constructions, such as

$$\begin{array}{lll} A & : & \alpha \quad \{\ldots\} \\ & | & \beta \quad \{\texttt{recover(...);}\} \end{array}$$

This technique, however, is not advisable for several reasons: (1) it is nearly impossible to anticipate all incorrect input constructions, (2) it might be difficult to write productions for them in a natural way, and (3) it might impair the readability of the grammar and, worst, it might render the grammar ambiguous with many s/r or r/r conflicts.

(b) **Panic mode synchronisation:** consists of discarding input tokens until the parser finds one or more from which parsing can continue in an error-free state. It is the simplest method but has the disadvantage of generating spurious and cascading errors.[2] The discarded tokens should be as few as possible. To avoid the cascading of errors, a good technique is to keep a count of the number of errors and stop parsing when more than 20, say, have been encountered.

In *yacc*, these two techniques are combined in a new elegant way. Recovery is performed by means of the reserved token `error`, that might be added to any production in the grammar. *yacc* will treat productions with `error` as any other production when generating the parsing table.

> ...[it] is used to find a *synchronisation point* in the grammar from which it is likely that processing can continue. Note that we said *likely*. Sometimes our attempts at recovery will not remove enough of the erroneous state to continue, and the error messages will cascade. Either the parser will reach a point from which processig can continue or the entire parser will abort.
>
> After reporting a [parser error], a yacc parser discards any partially parsed rules [if any] until it finds one in which it can shift an **error** token. It then reads and discards input tokens until it finds one which can follow the **error** token in the grammar... Normally, the parser refrains from generating any more error messages until it has successfully shifted three tokens without an intervening syntax error, at which point decides it has resynchronized and returns to its normal state.[3]

More precisely, given the current state on top of the stack and the next lookahead token, if there is no related shift or reduce action, `yyparse()` invokes `yyerror()` to issue an error message and increments the global variable `yynerrs`. It then *inserts* the `error` token before the one that caused the error as if `error` where the next input token. If the state on top of stack contains a shift action for `error` then the action is carried out, but if the associated action is error, then *yacc* pops its stack looking for a state on top of stack that contains a shift for `error` (note that a state contains a shift action for `error` when it contains the item $A \to \beta$. `error`). If the stack is emptied then `yyparse()` returns with a value of 1.

If `error` is shifted, recovery takes place by way of discarding input. By default, input is discarded until a token is found that may follow `error` in the grammar—a token in FOLLOW(A). Then, the *semantic* action associated to the production is executed and reduction takes place replacing β `error` for A in the stack. To avoid cascading, no more error messages will be issued until *three tokens* are successfully shifted after this reduction. If succesfully recovered, it will continue parsing to the end of input and return with a value of 0. We know that an error occurred because `yynerrs`> 0.

Both discarding and point of recovery can be configured:

[2]For example in C, if an error occurs inside a declaration and input is skipped to the semicolon that ends the declaration, all identifi ers from the point of error will be ignored and an "undeclared id" (semantic) error will be issued for each occurrence in the program.

[3]Levine et al. (1995) p. 248-249.

- **Discarding.** When the item that produced the shift action for `error` is $A \to \beta$. `error` α, there are three possible operations:
    1. If $\alpha = \lambda$, after shifting `error` the semantic action is executed and reduction takes place. Input is discarded until a token in FOLLOW(A) is found. This is the default operation.
    2. If α consists of a sequence of tokens (synchronisation terminals), then input is discarded until a string that matches the sequence is found. Then, the semantic action is executed and β . `error` α is reduced for A.
    3. If α is a sentential form, then `yyparse()` looks for a substring that can be reduced to α. Then, the semantic action is executed and β . `error` α is reduced for A.
- **Error messages.** By default, no more error messages will be issued after the point of error until three tokens are shifted. This operation can be overriden by inserting `yyerrok;` in semantic actions, which instructs `yyparse()` to continue in normal mode as if resynchronised.

3.9 Placement of `error` tokens

Good recommendations for placement of `error` tokens are given in Chapter 4 of Schreiner et al (1995). The following excerpt is taken from Levine et al. (1995) p. 251-252..

> You want to be as sure as possible that resynchronization will succed, so you want error tokens in the highest level rules in the grammar, maybe even the start rule, so there will always be a rule to which the parser can recover. On the other hand, you want to discard as little input as possible before recovering, so you want the error tokens in the lowest level rules to minimize the number of partially matched rules the parser has to discard...

We must be cautious in the use of `error` and `yyerrok;`. There exists, for example, the possibility of infinite loops. A production like

$$A \to \texttt{error}\ \{\ \texttt{... yyerrok; ...}\ \} \qquad (1)$$

will make `yyparse()` fall into an infinite loop. The reason is that on an error, `error` is taken as the next input token, inserting it *before* the lookahead that caused error, which remains as unread input. After error is shifted, `yyerrok;` instructs `yyparse()` to continue as if recovered and then reduction by production (1) takes place. No input has been discarded and the erroneous token is the next lookahead token again!

The macro `yyclearin;` will clear the lookahead buffer which contains the erroneous token. It can be useful when input is discarded in semantic actions instead than leaving it to the automatic operation of `yyparse()`. In the following example, `recover()` calls `yylex()` to skip input and the parser expects tokens in FIRST(β).

$$\begin{array}{lll} A & \longrightarrow & \alpha\ \texttt{error}\ \{\texttt{yyclearin; recover(); yyerrok; }\} \\ & & \beta\ \{\texttt{...}\} \end{array}$$

Semantic actions in error productions usually do some housekeeping like updating symbol table structures accordingly with the discarded input, reclaim syntax tree storage, etc.

4 Writing compilers with *lex* and *yacc*

(1) Write scanner for *lex* and test it. (2) Write parser for *yacc* with just the grammar. Check conflicts and do some trials. (3) Add `error` productions. This may introduce conflics and force some rewriting. (4) Join together scanner and parser and do some trace trials. Cannot declare type of value stack with `%union`, assume default `int` and make scanner return just token codes, otherwise default action `$$=$1` will force type declaration of some non-terminals and type conflicts may arise. (5) Declare attributes and their types. (6) Add semantic actions: it is a *syntax directed translation* where the grammar serves as a control structure

guiding the implementation process. (6.1) Add semantic actions regarding names and constants which are controled by the use of a symbol table. Using the grammar as a guide, check for each ocurrence of a name or a constant, decide what needs to be checked and call the appropriate routines. (6.2) Add semantic actions for type checking declaring type attribute for appropriate nom-terminals. (6.2) Add semantic actions for calculating offsets of variables—memory allocation. (6.3) Add semantic actions for code generation.

References

A. Aho, R. Sethi, and J. Ullman (1990). *Compiladores: Principios, Técnicas y Herramientas*. Addison-Wesley.

Stephen C. Johnson (July 31, 1978). *YACC: Yet Another Compiler-Compiler*. Bell Laboratories. Murray Hill, New Jersey 07974.

J. Levine, T. Mason and D. Brown (1995). *Lex & Yacc*. O'Reilly and Associates.

Axel T. Schreiner and H. George Friedman, Jr (1995). *Introduction to Compiler Construction with UNIX*. Prentice-Hall.

`yyerror()`	This function *must be supplied by the user*. It is called from `yyparse()` on a parse error. Used for error reporting.
`yyparse()`	The parser function. It returns an integer value: zero if there is success or non-zero if unable to parse the token sentence provided by `yylex()`.
`yylval`	Global variable of type `YYSTYPE` that contains the value pushed on a *shift*—i.e., a token attribute value, so it must be set by `yylex()`, usually to a *copy* of `yytext` (`yytext` is modified each time a token pattern matches the input).
`yyval`	Global variable of type `YYSTYPE` that contains the value pushed on a *goto*—i.e., the value `$$`.
`yynerrs`	Global `int` variable that holds the number of errors found. It is automatically incremented by `yyparse()` on each call to `yyerror()`.
`yyerrok;`	It is an action that instructs the parser to continue in normal mode, as if recovered from an error.

`YYRECOVERING()`	If the user calls its own error-reporting routine whose name is other than `yyerror()`, such routine should use the *yacc* macro `YYRECOVERING()` to test if the parser is trying to resynchronise, in which case no error messages should be printed:
	`warning(char *msg)`
	`{`
	`    if (YYRECOVERING()) return;`
	`    ...`
	`    /* print error */`
	`}`

`yydebug`	Global `int` variable that should be set to 1 if `YYDEBUG` has been defined and we want to output trace information (see Section 3.7). It should be declared as extern in `main()` and set to the proper value there.
	`main()`
	`{`
	`    #ifdef YYDEBUG`
	`    extern int yydebug;`
	`    #endif`
	`    ...`
	`    yydebug = 1;`
	`    ...`
	`}`

Figure 2: *yacc*'s variables and functions