5.9. Recovering from Syntactic Errors

Up to this point the parser had only the modest task of determining whether or not an input sequence of symbols belonged to a language. As a side product, the parser also discovered the inherent structure of a sentence. But as soon as an ill-formed construct was encountered, the parser's task was achieved, and the program could as well terminate. For practical compilers, this is of course no tenable proposition. Instead, a compiler must issue an appropriate error diagnostic and be able to continue the parsing process—probably to find further mistakes. A continuation is only possible either by making some likely assumption about the nature of the error and the intention of the author of the ill-formed program or by skipping over some subsequent part of the input sequence, or both. The art of choosing an assumption with a high likelihood of correctness is rather intricate. It has so far eluded any kind of successful formalization because formalizations of syntax and parsing do not take into account the many factors that strongly influence the human mind. For instance, it is a common error to omit inter-punctuation symbols such as the semicolon (not only in programming!), whereas it is highly improbable that one forgets to write a + operator in an arithmetic expression. The semicolon and plus symbol are merely terminal symbols without further distinction for the parser; for the human programmer, the semicolon has hardly a meaning and appears redundant at the end of a line, whereas the significance of an arithmetic operator is obvious beyond doubt. There are many more such considerations that have to go into the design of an adequate recovery system, and they all depend on the individual language and cannot be generalized in the framework of all context-free languages.

Nevertheless, there are some rules and hints that can be postulated and that have validity beyond the scope of a single language such as PL/0. Characteristically, perhaps, they are concerned equally much with the initial conception of a language as with the design of the recovery mechanism of its parser. First of all, it is abundantly clear that sensible recovery is much facilitated, or even made possible, only by a simple language structure. In particular, if upon diagnosing an error some part of the subsequent input is to be skipped (ignored), then it is mandatory that the language contains key words that are highly unlikely to be misused, and that may therefore serve to bring the parser back into step. PL/0 notably follows this rule: every structured statement begins with an unmistakable keyword such as begin, if, while, and the same holds for declarations; they are headed by var, const, or procedure. We shall therefore call this rule the keyword rule.

The second rule concerns the construction of the parser more directly. It is the characteristic of top-down parsing that goals are split up into
subgoals and that parsers call upon other parsers to tackle their subgoals. The second rule specifies that if a parser detects an error, it should not merely refuse to continue and report the happening back to its master parser. Instead, it should itself continue to scan text up to a point where some plausible analysis can be resumed. We shall therefore call this the don’t panic rule. The programmatic consequence of this rule is that there will be no exit from a parser except through its regular termination point.

A possible strict interpretation of the don’t panic rule consists of skipping input text upon detecting an illegal formation up to the next symbol that may correctly follow the currently parsed sentential construct. This implies that every parser know the set of its follow-symbols at the place of its present activation.

In the first refinement (or enrichment) step we shall therefore provide every parsing procedure with an explicit parameter $f_{sys}$ that specifies the possible follow-symbols. At the end of each procedure an explicit test is included to verify that the next symbol of the input text is indeed among those follow-symbols (if this condition is not already asserted by the logic of the program).

It would, however, be very shortsighted of us to skip the input text up to the next occurrence of such a follow-symbol under all circumstances. After all, the programmer may have mistakenly omitted exactly one symbol (say a semicolon); ignoring the entire text up to the next follow-symbol may be disastrous. We therefore augment these sets of symbols that terminate a possible skip by keywords that specifically mark the beginning of a construct not to be overlooked. The symbols passed as parameters to the parsing procedures are therefore stopping symbols rather than follow-symbols only. We may regard the sets of stopping symbols as being initialized by distinct key symbols and being gradually supplemented by legal follow-symbols upon penetration of the hierarchy of parsing subgoals. For flexibility, a general routine called $test$ is introduced to perform the described verification. This procedure (5.17) has three parameters:

1. The set $s_1$ of admissible next symbols; if the current symbol is not among them, an error is at hand.
2. A set $s_2$ of additional stopping symbols whose presence is definitely an error, but which should in no case be ignored and skipped.
3. The number $n$ of the pertinent error diagnostic.

\begin{align*}
\text{procedure test} & \left(s_1, s_2, \text{ symset; } n: \text{ integer}\right); \\
& \text{begin if } \neg(\text{sym in } s_1) \text{ then} \\
& \quad \text{begin error}(n); s_1 := s_1 + s_2; \\
& \quad \quad \text{while } \neg(\text{sym in } s_1) \text{ do \text{getsym}} \\
& \quad \text{end} \\
& \text{end}
\end{align*}

(5.17)
Procedure (5.17) may also be conveniently used at the entrance of parsing procedures to verify whether or not the current symbol is an admissible initial symbol. This is recommended in all cases in which a parsing procedure $X$ is called unconditionally, such as in the statement

$$\text{if } \text{sym} = a_1 \text{ then } S_1 \text{ else}$$

$$\ldots \ldots$$

$$\text{if } \text{sym} = a_n \text{ then } S_n \text{ else } X$$

which is the result of translation of the production

$$A ::= a_1S_1|\ldots|a_nS_n|X$$  \hspace{1cm} (5.18)

In these instances the parameter $s_1$ must be equal to the set of initial symbols of $X$, whereas $s_2$ is chosen as the set of the follow-symbols of $A$ (see Table 5.2). The details of this procedure are given in Program 5.5, which represents the enriched version of Program 5.4. For the reader's convenience, the entire parser is listed again, with the exception of initializations of global variables and of the procedure $\text{getsym}$, all of which remain unchanged.

The scheme presented so far has the property of trying to recover, to fall back into step, by ignoring one or more symbols in the input text. This is an unfortunate strategy in all cases in which an error is caused by omission of a symbol. Experience shows that such errors are virtually restricted to symbols which have merely syntactic functions and do not represent an action. An example is the semicolon in PL/0. The fact that the follow-symbol sets are augmented by certain key words actually causes the parser to stop skipping symbols prematurely, thereby behaving as if a missing symbol had been inserted. This can be seen from the program part that parses compound statements shown in (5.19). It effectively "inserts" missing semicolons in front of key words. The set called $\text{statbegsys}$ is the set of initial symbols of the construct "statement."

$$\text{if } \text{sym} = \text{beginsym then}$$
$$\text{begin getsym; }$$
$$\text{statement}([\text{semicolon, endsym}]+fsys);$$
$$\text{while } \text{sym in } [\text{semicolon}]+\text{statbegsys do }$$
$$\text{begin }$$
$$\text{if } \text{sym} = \text{semicolon then getsym else error; }$$
$$\text{statement}([\text{semicolon, endsym}]+fsys)$$
$$\text{end};$$
$$\text{if } \text{sym} = \text{endsym then getsym else error }$$
$$\text{end }$$

(5.19)

The degree of success with which this program diagnoses syntactic errors and recovers from unusual situations can be estimated by considering the PL/0 program (5.20). The listing represents an output delivered by Program
Fig. 5.6 Modified compound statement syntax.

1. Use = instead of :=.
2. = must be followed by a number.
3. Identifier must be followed by =.
4. const, var, procedure must be followed by an identifier.
5. Semicolon or comma missing.
6. Incorrect symbol after procedure declaration.
7. Statement expected.
8. Incorrect symbol after statement part in block.
10. Semicolon between statements is missing.
11. Undeclared identifier.
12. Assignment to constant or procedure is not allowed.
14. call must be followed by an identifier.
15. Call of a constant or a variable is meaningless.
16. then expected.
17. Semicolon or end expected.
18. do expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression must not contain a procedure identifier.
22. Right parenthesis missing.
23. The preceding factor cannot be followed by this symbol.
24. An expression cannot begin with this symbol.
25. This number is too large.

Table 5.3 Error Messages of PL/0 Compiler.

5.5, and Table 5.3 list a set of possible diagnostic messages corresponding to the error numbers in Program 5.5.

The following program (5.20) was obtained by the introduction of syntactic errors in (5.14) through (5.16).

```
const m = 7, n = 85
var x,y,z,q,r;
   ↑ 5
   ↑ 5
procedure multiply;
   var a,b
```
begin $a := u; \ b := y; \ z := 0$

\[ 5 \]

\[ 10 \]

while $b > 0$ do

\[ 11 \]

begin

\[ 16 \]

if odd $b$ do $z := z + a;$

\[ 19 \]

$a := 2a; \ b := b/2;$

\[ 23 \]

end

end;

procedure divide

\[ 5 \]

var $w$;

\[ 7 \]

const two = 2, three := 3;

\[ 1 \]

begin $r = x; \ q := 0; \ w := y$;

\[ 13 \]

\[ 24 \]

while $w \leq r$ do $w := \text{two} \times w$;

while $w > y$

begin $q := (2 \times q; \ w := w/2);$}

\[ 18 \]

\[ 22 \]

\[ 23 \]

if $w \leq r$ then

begin $r := r - w \ q := q + 1$

\[ 23 \]

end

end;

procedure gcd;

\[ 5 \]

var $f,g$;

begin $f := x; \ g := y$

while $f \neq g$ do

\[ 17 \]

begin if $f < g$ then $g := g - f$

if $g < f$ then $f := f - g$

$z := f$

end;
begin
  \( x := m; y := n; \) \textbf{call} multiply; \\
  x := 25; y := 3; \textbf{call} divide; \\
  x := 84; y := 36; \textbf{call} gcd; \\
  \textbf{call} x; x := \text{gcd}; \text{gcd} = x

\( \uparrow 15 \)

\( \uparrow 21 \)

\( \uparrow 12 \)

\( \uparrow 13 \)

\( \uparrow 24 \)

end . \\
\( \uparrow 17 \)

\( \uparrow 5 \)

\( \uparrow 7 \)

PROGRAM INCOMPLETE

It should be clear that no scheme that reasonably efficiently translates correct sentences will also be able to handle all possible incorrect constructions in a sensible way. And why should it! Every scheme implemented with reasonable effort will fail, that is, will inadequately handle some misconstructions. The important characteristics of a good compiler, however, are that

1. No input sequence will cause the compiler to collapse.  
2. All constructs that are illegal according to the language definition are detected and marked.  
3. Errors that occur reasonably frequently and are true programmer’s mistakes (caused by oversight or misunderstanding) are diagnosed correctly and do not cause any (or many) further stumblings of the compiler (so-called spurious error messages).

The presented scheme performs satisfactorily, although there is always room for improvement. Its merit is that it is built according to a few ground rules in a systematic fashion. The ground rules are merely supplemented by some choices of parameters based on heuristics and experience with actual use of the language.
Program 5.5  PL/0 Parser with Error Recovery.

program PL0 (input, output);
{ PL/0 compiler with syntax error recovery }
labeled 99;

const norw = 11; { no. of reserved words }
    txmax = 100; { length of identifier table }
    nmax = 14;  { max. no. of digits in numbers }
    al = 10;    { length of identifiers }

type symbol =
    (null, ident, number, plus, minus, times, slash, oddsym,
    eql, neg, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
    period, becomes, beginsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, consysym, varsym, procsym);
    alfa = packed array [1 .. al] of char;
    object = (constant, variable, procedure);
    symset = set of symbol;

var ch: char;    { last character read }
    sym: symbol;  { last symbol read }
    id: alfa;     { last identifier read }
    num: integer; { last number read }
    cc: integer;  { character count }
    ll: integer;  { line length }
    kk: integer;
    line: array [1 .. 81] of char;
    a: alfa;
    word: array [1 .. norw] of alfa;
    wsym: array [1 .. norw] of symbol;
    ssym: array [char] of symbol;
    decbegsys, statbegsys, factbegsys: symset;
    table: array [0 .. txmax] of
        record name: alfa;
            kind: object
        end;

procedure error (n: integer);
begin writeln(' line', n: 2);
end { error };

procedure test (s1, s2: symset; n: integer);
begin if ((sym in s1) then
        begin error(n); s1 := s1 + s2;
            while ((sym in s1) do getsym
        end
end { test };

326
procedure block (tx: integer; fsym: symset);
  procedure enter (k: object);
    begin {enter object into table}
      tx := tx + 1;
      with table[tx] do
        begin name := id; kind := k;
        end
    end {enter};
  function position (id: alfα): integer;
    var i: integer;
    begin {find identifier id in table}
      table[0].name := id; i := tx;
      while table[i].name ≠ id do i := i-1;
      position := i
    end {position};
  procedure constdeclaration;
  begin if sym = ident then
    begin getsym;
      if sym in [eq, becomes] then
        begin if sym = becomes then error (1);
          getsym;
          if sym = number then
            begin enter (constant); getsym
            end
        end else error (2)
      end else error (3)
    end {constdeclaration};
  procedure vardeclaration;
  begin if sym = ident then
    begin enter (variable); getsym
    end else error (4)
  end {vardeclaration};
  procedure statement (fsym: symset);
  var i: integer;
  procedure expression (fsym: symset);
  procedure term (fsym: symset);
  procedure factor (fsym: symset);
  var i: integer;

Program 5.5 (Continued)
begin test (fachegsys, fsys, 24);
    while sym in fachegsys do
        begin
            if sym = ident then
                begin i := position (id);
                    if i = 0 then error (11) else
                        if table[i].kind = procedure then error (21);
                end
            end else
            if sym = number then
                begin getsym;
            end else
            if sym = lparen then
                begin getsym; expression ([lparen]+fsys);
                    if sym = rparen then getsym else error (22)
                end;
                test(fsys, [lparen], 23)
        end
    end {factor} ;
    begin {term}.factor (fsys+[times, slash]);
        while sym in [times, slash] do
            begin getsym; factor (fsys+[times, slash])
        end
    end {term} ;
    begin {expression}
        if sym in [plus, minus] then
            begin getsym; term(fsys+[plus, minus])
        end else term(fsys+[plus, minus]);
        while sym in [plus, minus] do
            begin getsym; term(fsys+[plus, minus])
        end
    end {expression} ;
    procedure condition(fsys: symset);
    begin
        if sym = oddsym then
            begin getsym; expression(fsys);
        end else
            begin expression ([eq, neq, lss, gtr, leq, geq]+fsys);
                if ≠(sym in [eq, neq, lss, leq, gtr, geq]) then
                    error (20) else
                        begin getsym; expression (fsys)
                    end
            end
    end {condition} ;

Program 5.5 (Continued)
\begin{verbatim}
begin [statement]
    if sym = ident then
        begin i := position(id);
            if i = 0 then error (11) else
                if table[i].kind \neq variable then error (12);
                getsym; if sym = becomes then getsym else error (13);
                expression(fsys);
        end else
        if sym = callsym then
            begin getsym;
                if sym \neq ident then error (14) else
                    begin i := position(id);
                        if i = 0 then error (11) else
                            if table[i].kind \neq procedure then error (15);
                            getsym
                    end
            end else
            if sym = ifsym then
                begin getsym; condition(then sym, dosym] + fsys);
                    if sym = then sym then getsym else error (16);
                    statement(fsys)
                end else
                if sym = begin sym then
                    begin getsym; statement(semicolon, endsym] + fsys);
                        while sym in [semicolon] + statbegsys do
                            begin
                                if sym = semicolon then getsym else error (10);
                                statement([semicolon, endsym] + fsys)
                            end;
                            if sym = endsym then getsym else error (17)
                    end else
                    if sym = while sym then
                        begin getsym; condition([dosym] + fsys);
                            if sym = dosym then getsym else error (18);
                            statement(fsys);
                        end;
                        test(fsys, [], 19)
        end [statement];
\end{verbatim}
begin \{ block \}
repeat
  if \textit{sym} = \textit{constsymb} then
  begin \textit{getsym};
  repeat \textit{constdeclaration};
    while \textit{sym} = \textit{comma} do
      begin \textit{getsym}; constdeclaration
      end;
    if \textit{sym} = \textit{semicolon} then \textit{getsym} else error (5)
  until \textit{sym} ≠ ident
  end;
  if \textit{sym} = \textit{varsymb} then
  begin \textit{getsym};
  repeat \textit{vardeclaration};
    while \textit{sym} = \textit{comma} do
      begin \textit{getsym}; vardeclaration
      end;
    if \textit{sym} = \textit{semicolon} then \textit{getsym} else error (5)
  until \textit{sym} ≠ ident;
  end;
while \textit{sym} = \textit{procsymb} do
begin \textit{getsym};
  if \textit{sym} = \textit{ident} then
  begin enter (procedure); \textit{getsym}
  end
else error (4);
  if \textit{sym} = \textit{semicolon} then \textit{getsym} else error (5);
block (\textit{tx}, [\textit{semicolon}]+\textit{fys});
  if \textit{sym} = \textit{semicolon} then
  begin \textit{getsym}; test(statbegsys+[ident, procsymb], fsys, 6)
  end
else error (5)
end;
test(statbegsys+[ident], declbegsys, 7)
\textbf{until} \neg(\textit{sym} \textit{in} declbegsys);
\textbf{statement}([\textit{semicolon}, endsym]+\textit{fys});
test(fsys, [ ], 8);
\end{block};
begin \{ main program \}
. . . Initialization (see Program 5.4) . . .
\textit{cc} := 0; \textit{ll} := 0; \textit{ch} := \textit{'}; \textit{kk} := \textit{al}; \textit{getsym};
block (0, [\textit{period}]+decbegsys+statbegsys);
if \textit{sym} ≠ \textit{period} \textbf{then} error (9);
99: writeln