Teaching Syntax in an Introductory Programming Course

Katrin Becker

Department of Computer Science
University of Calgary

All introductory programming students must learn the syntax of the language they are to use. The problems that students have learning syntax are described, and a teaching methodology is suggested. Two types of exercises are explained which will help the students learn syntax, and the reactions of the students who have used them are outlined.

Introduction

Curriculum '78 includes "syntax and semantics of a higher level (problem oriented) language" [AUSI79] in its list of elementary topics and, indeed, most Computer Science programs at post-secondary institutions offer an introductory programming course in the first year. Much has been written [ROSI73, GRIE74] about the need to teach problem-solving skills, and its importance is undeniable, but no matter how a course is organized to facilitate the learning of these skills, the fact remains that the students must also learn the structure of the language they are to use.

"To learn a programming language ... requires both a comprehension of the meaning of the available forms of sentences and a detailed knowledge of the syntactic rules governing the language." [WIRT73] It has been claimed that syntax is the most easily learned part of any language and we shouldn't spend much time on it [KIRK82]. It is true that the syntax is usually the most straight-forward part of a language; however, it is also true that a disproportionately large amount of the available time, both in lectures and on assignments, is spent teaching the structure of the language. This is a result of the fact that the programs must not only be read and understood, but, more importantly, be invented and formulated [WIRT73].

Initially, the goal of the formal study of syntactic analysis was to provide precise definitions of the syntax of a programming language [PRAT75]. It has also provided us with a useful tool for teaching the syntax of a language to students. A survey of several introductory programming texts [BRAI82, CHER80, HUME82, KELL82, MOOR82, TREM80, WELS79] reveals widespread use of syntax diagrams and BNF-like grammars in defining the legal constructs of a language. Conspicuous by its absence however, is any attempt to explain how these formal definitions can actually be used. Often, an attempt is made at defining the term 'syntax', but then the syntax diagrams are simply presented as fact without further explanation. Sometimes the text includes several examples of the construct in question, both correct and incorrect, but still the majority of the time spent on programming assignments by introductory students is used in finding and correcting syntax errors. The 'tools' currently available to the average student are obviously insufficient.

Teaching Syntax

The purpose of this paper is to suggest a method by which students may become familiar with the syntax of a language by actually using the syntax diagrams as tools. Pascal has been chosen as the programming language for several reasons. Perhaps most importantly, it is widely used as an introductory programming language. In addition, Pascal is a language that can be easily defined using syntax diagrams. The materials required for the proposed exercises are readily obtainable (syntax compiler error messages).

The syntax of a language provides a notation for the communication of information between the programmer and the processor [PRAT75]. Clearly, knowing the syntax of a language allows one to write programs that will compile without error (i.e. programs that are free of syntax errors). Learning something of syntax analysis as well as just the rules themselves can be very beneficial to beginning programmers. Students can learn to use syntax diagrams as tools for debugging their own programs. They learn which errors are detectable by the compiler, because they can learn, at least in general terms, how the compiler looks for these errors. Another valuable result is that by doing exercises intended to help them learn the syntax, they will also gain experience in reading programs and following what is actually written rather than what they think is written.

By allowing students to use syntax diagrams as tools, the syntax rules will acquire more meaning than if simply presented as fact along with a few examples. Students can be given exercises that allow them to work with the syntax rules to see how they fit together, and what happens when there is a syntax error. They can actually be shown the difference between syntax and 'logic' errors within programs.

Exercise I

The first set of exercises require only the Pascal syntax diagrams and some sample sections of code. The syntax diagrams used are of a somewhat simplified form (see 'Problems and Comments'). The term 'sections of code' is used rather than programs because, as will be explained later, both partial and complete programs are useful here. The object of the exercise is to reduce the sample code, beginning with simple tokens and ending with the highest level syntactic unit for that particular example. In the case of a finished program, the highest possible syntactic level is <program>. Partial programs should be written such that they reduce to a single syntactic unit (eg. <expression> or <compound statement>). For the purpose of these exercises, the sample code must be syntactically correct to avoid confusion (error detection and handling are covered in the second set of exercises!). The examples used can be varied to illustrate particular classes of constructs. For example, one may wish the students to concentrate their efforts on the different kinds of loops. The examples could then consist of complete loops, even nesting them. It is also possible to isolate procedures, functions, if-then-else statements, declarations, expressions, etc.

The method used to complete the exercise is for the students to make several passes over the sample, rewriting the program each time, but replacing one set of syntactic entities with the next higher set in the process. The first pass would reduce all possible strings to tokens. In order to facilitate the writing, short mnemonics must be chosen for the names of their syntactic entities (a list of these mnemonics can be attached to the syntax diagrams!).

Problems and Comments

After having completed a few examples myself, using just the syntax diagrams as given in the Pascal User Manual and Report [JENS74], it became apparent that

in order to be easily used, they required some modification. Some syntactic entities needed to be broken into several smaller ones (such as <statement> and <block>) while others needed to be simplified for some examples (for instance, the set of diagrams describing literals is too detailed for use in the reduction of complete programs : there is a danger that the students will get caught up in the details and fail to see the larger picture). The modified syntax diagrams appear at the end of this paper.

Although it would be possible to create an on-line system for these exercises in the CAI spirit, there are several reasons for not so doing. In order for these exercises to fulfill their objectives, it must be possible for the students to do many of them. This is accomplished most economically by having the students work on these exercises at home. Enrollment in most introductory programming classes is quite large (especially at universities) and these, together with the other programming classes often tax the available computing facilities to the limit. Adding to this by requiring the students to complete these exercises on-line is likely to reduce the amount of time students can use the computer to write their own programs. One of the values of these exercises is that the students can gain practical experience reading and working with programs without having to be logged on.

Exercise II

The second set of exercises requires a bit more in the way of materials but the exercises can still easily be accomplished using pencil and paper methods. These exercises require the syntax diagrams as previously described, example sections of code (again both complete and partial programs are useful), a list of the error messages a compiler might produce (each numbered), and a set of rules to go with each error message that tells how to recover from the error that caused that message. In essence, the students are asked to 'play compiler' and parse the examples given. Their task is to go through the examples, using the syntax diagrams to guide them and print out the appropriate error message when one is found (one could name the error number and the line number in the example where it was found), follow the recovery rules and continue on until the end of the example is reached. For each example, the syntax diagram used to start with should be named (for complete programs , it would simply be <program>). As these exercises can quickly become quite complicated, it would be helpful for the students to copy out the syntax diagrams (using

mnemonic forms for the syntactic unit names) as they follow them. This will allow them not only to find their place in a particular syntax diagram (it is often necessary to leave one in the middle and return to it at some later point), but it will make backtracking much easier, should it become necessary. It will also leave the student with a 'picture' of the example they just parsed.

This second set of exercises is somewhat more advanced than the first, and therefore may require introduction in steps. A suitable first step is to provide the students with syntactically correct examples and have them simply parse them, writing out the diagrams as they use them. Once this can be done with relative ease it will be possible to begin introducing errors into the examples.

These exercises are intended to give the student some understanding of what the compiler does as well as exposure to a wide range of syntax errors. Some common errors that students often have difficulty with are : too many or not enough 'end' statements, missing closing quotes, missing closing comment delimiters, and misplaced semi-colons. These types of errors often result in a deluge of error messages, most of which convey little or no information about the actual error. Following the exercises will allow the students to see how this happens.

Experience with the Exercises

Early experience with the exercises has been encouraging. To supplement preliminary reactions, both types of exercises were given to a second-year class in January 1983 and the students' reactions have been noted. As the sample programs chosen have not been used in the exercises before, they were given to a second-year class of approximately 80 students instead of the introductory class because they would be better able to cope with possible mistakes (a first year class should be given examples that are known to be managable and not misleading).

The students were given questionnaires and asked to respond to general questions that primarily requested 'yes/no' answers (with room for comments). Half of the students who completed the exercises had had no prior experience with Pascal and the rest had used it in their introductory course but still felt they did not know it well. Most students had seen syntax diagrams before but few had ever used them. Those who were familiar with Pascal used the

exercises as a review and found them quite helpful. The others used the opportunity to familiarize themselves with the Pascal syntax. Almost all felt these exercises would have been helpful in their first course; in particular, they felt the second exercise helped them to find the causes of errors in their own programs.

Several students criticised the syntax diagrams for being incomplete, and in fact the diagrams are currently being refined. With clearer syntax diagrams and carefully chosen sample programs this problem will be corrected. The other main complaint was with the amount of sample code they were given. Many students felt there were too many. This problem can also be corrected by giving small samples throughout the entire course (these students were given the examples all at once!). New samples are currently being devised; the intention being to provide somewhat more formal test results. The exercises will this time be given to a first-year class (in the fall of '83) and they will be tested rather than asked to fill out a questionnaire.

It is clear from the students' comments that these exercises cannot replace lectures, labs, or programming assignments but they do provide a valuable supplement that can be used to some extent, whenever they are required to write a program.

Conclusions

Students in Computer Science, perhaps more than almost any other discipline, must learn how to learn. Over the years it has become apparent that students graduating out of Computer Science programs will have to relearn most of what they know about programming [KIRK82]. Understanding syntax and being able to use syntax diagrams provides these students with a tool that is a step removed from the actual language and therefore can be applied with relative ease when learning other languages.

Aside from its value as a tool to be used later, learning how to use syntax diagrams also provides the student with more immediate gains. The most important of these are : experience in reading programs, learning to recognize common syntax errors, and acquiring a rudimentary understanding of the parsing process.

References

[AUST79] Austig,Richard H. et.al., Ed., "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science - A Report of the ACM Curriculum Committee on Computer Science", CACM Vol.22,No.3, (Mar.'79), pp 147-165

[ROSI73] Rosin,R.F., "Teaching 'About' Programming", CACM Vol.16,No.7, (July '73) pp 435-439

[GRIE74] Gries, D., " What Should We Teach in an Introductory Programming Course?", SIGCSE Bulletin (ACM) 6,1 (Feb.'74) pp 81-89

[WIRT73] Wirth,Niklaus, "Systematic Programming : An Introduction", New Jersey: Prentice-Hall, 1973

[BRAI82] Brainerd,W.S, G.H.Goldberg, and J.L.Gross, "Pascal Programming : A Spiral Approach", San Fransisco: Boyd and Fraser Publishing Company, 1982

[CHER80] Cherry,G.W., "Pascal Programming Structures: An Introduction to Systematic Programming", Reston,Virginia: Reston Publishing Company, 1980

[HUME82] Hume,J.N.P., and R.C.Holt, "UCSD Pascal : A Beginner's Guide to Programming Microcomputers", Reston,Virginia: Reston Publishing Company, 1982

[KELL82] Keller,Arthur, "A First Course in Computer Programming", New York: McGraw-Hill, 1982

[MOOR82] Moore,John B., "Pascal Text and Reference with Waterloo Pascal and Pascal VS", Reston,Virginia: Reston Publishing Co., 1982

[TREM80] Tremblay,Jean Paul, R.B.Bunt, and L.M.Pseth, "Structured Pascal", New York: McGraw-Hill, 1980

[WELS79] Welsh,Jim, and John Elder, "Introduction to Pascal", London: Prentice-Hall International, 1979

[KIRK82] Kirkerud,Bjorn R., "The Teaching of Programming", Teaching Informatics Courses, HLWJackson (Ed.), North-Holland Book Publishing Co., IFIP 1982, pp 25-42

[PRAT75] Pratt, Terrence W., "Programming Language : Design and Implementation", New Jersey : Prentice-Hall, 1975

[JENS74] Jensen,Kathleen, and Niklaus Wirth, "Pascal User Manual and Report", 2nd Ed. New York : Springer-Verlag, 1974

SYNTAX DIAGRAMS

Syntax Diagrams define the structure of a language. They can be used to help you understand how the statements in a language are formed. They can also be used to help you find mistakes in your programs. Each diagram describes what is called a syntactic entity. The boxes denote the components of this entity (some of which need further explanation) and the lines and arrows define the possible combinations.

The exercises you have been given will use the syntax diagrams extensively so that you may become familiar with their utility.

The following explanations supplement and complete the syntax diagrams you have been given.

1. Symbols: Circles denote terminal symbols (i.e. those actually written in Pascal programs.
   : Rounded boxes denote Pascal keywords and are written just as they appear (except 'character', which stands for any single character). The keywords shown do not represent the complete set of Pascal keywords - see your text for a full set..
   : The boxes refer to a syntactic entity, most of which are defined by other syntax diagrams. The following may be considered to be primitive tokens and thus are not defined by syntax diagrams :

   ID : Identifier - any name that is not a keyword

   UI : Unsigned Integer - any positive whole number

   UN : Unsigned Number - any positive number (incl. decimals and scientific notation)

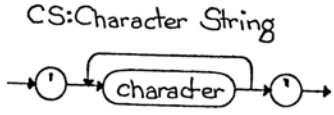
   E : Expression - any epression of the form : unary operator operand operator operand ... or operand operator operand ...

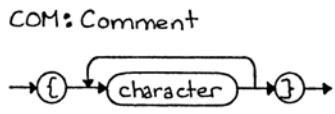
   T : Type - any system or user defined type

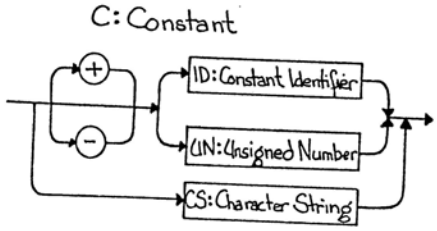
2. Comments may occur anywhere except within : constants, variables, expressions, and parameter lists. A syntax diagram for comments has been provided, but for Exercise I, the comments may be ignored after Pass 1.

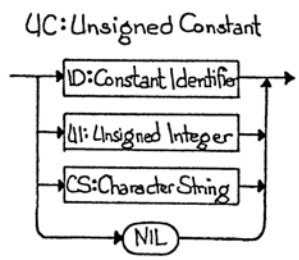
3. For the purposes of these exercises, qualifiers on variables in 'write', 'writeln', 'read', and 'readln' (eg. x:5 or y:5:2) along with the variables themselves may be considered to be expressions.

4. All statements are coded as 'S' to make the reduction somewhat simpler.

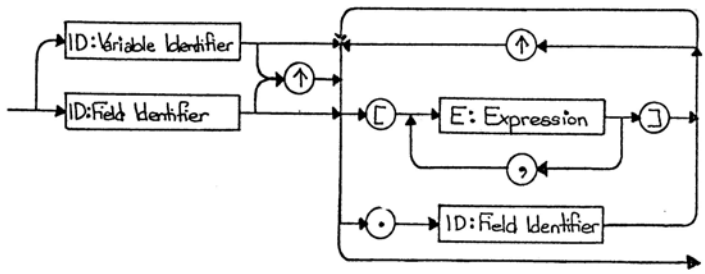




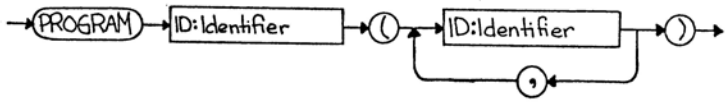




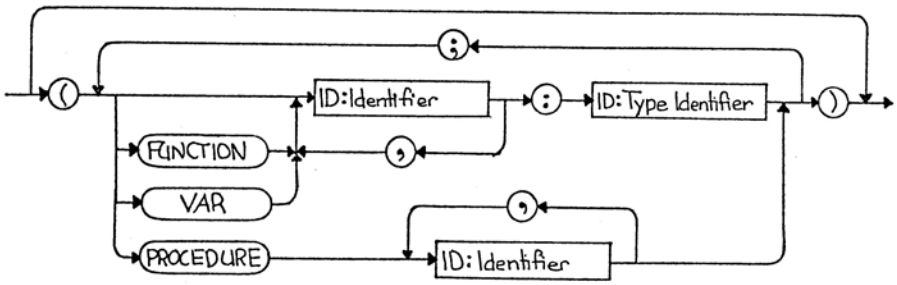
V: Variable



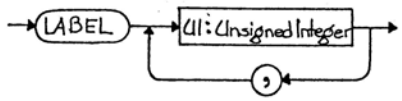
PH: Program Header



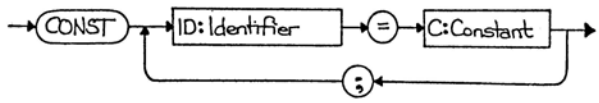
PL: Parameter List



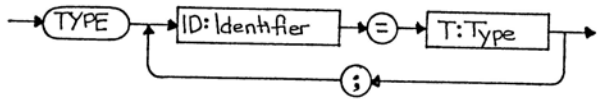
LD: Label Declaration



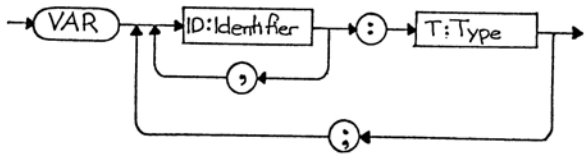
CD: Constant Declaration



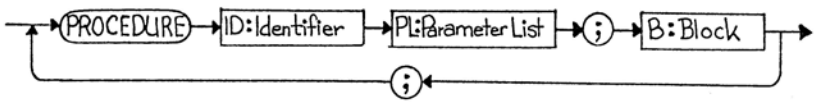
TD: Type Declaration



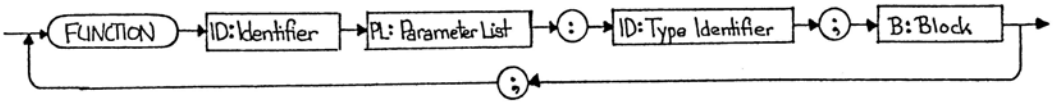
VD: Variable Declaration



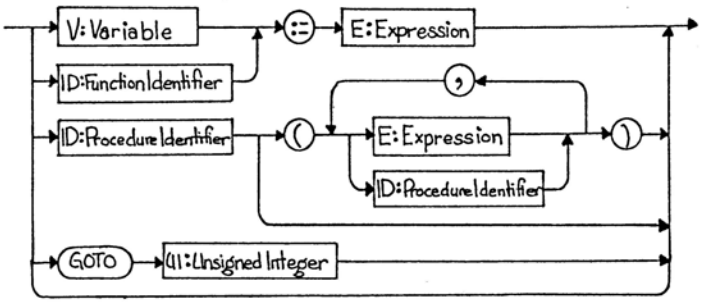
PD: Procedure Declaration



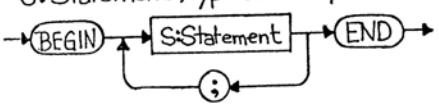
FD: Function Declaration



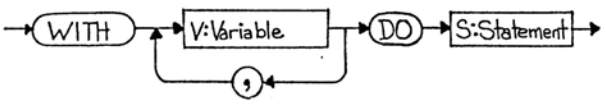
S: Statement, Type 1 : Simple Statement



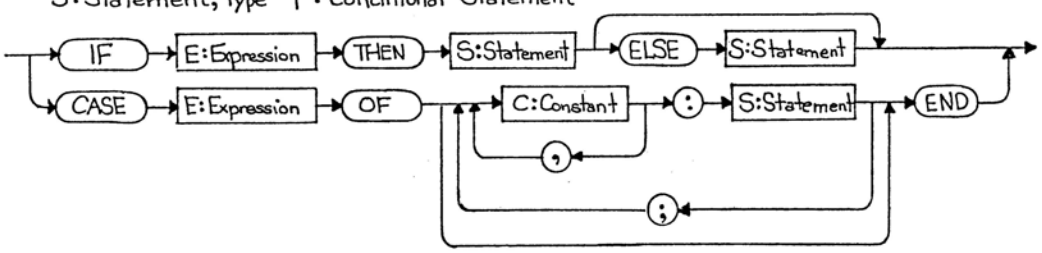
S: Statement, Type 2 : Compound Statement



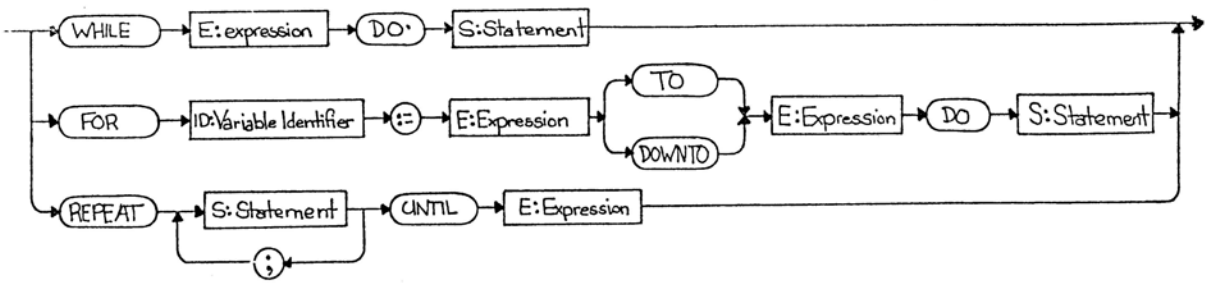
S: Statement, Type 3 : With Statement

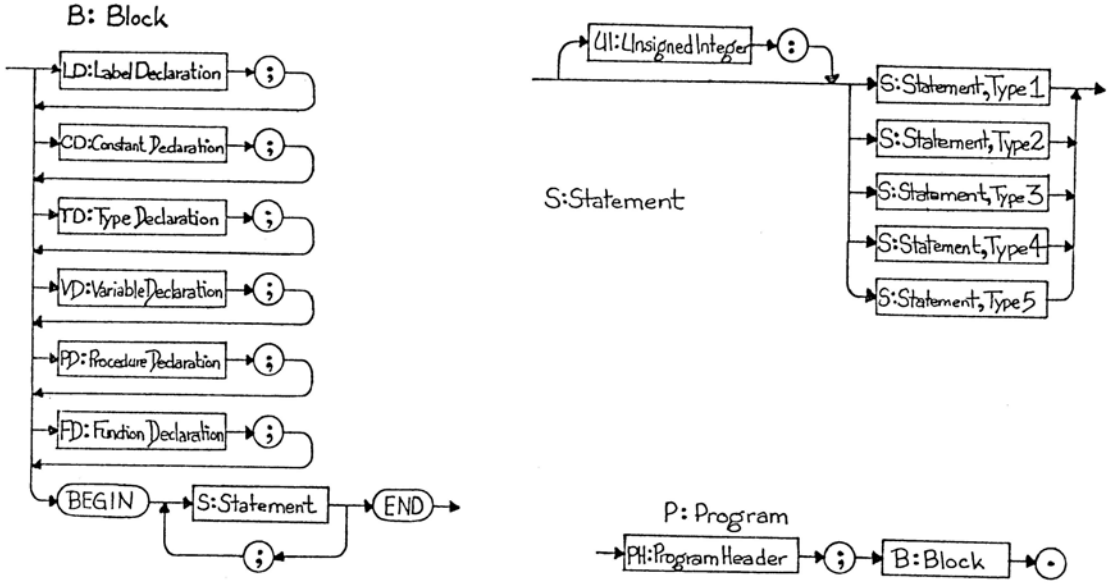


S: Statement, Type 4 : Conditional Statement



S: Statement, Type 5 : Repetative Statement





ERROR MESSAGES

1. Identifier Expected :: assume there was one and continue.

2. 'program' Expected :: insert and continue.

3. ')' Expected :: insert and continue.

4. ':' Expected :: insert and continue.

5. Illegal Symbol :: mark, ignore, and continue.

6. Error in Parameter List :: Search for ')' or end of line, if ')' found first, assume you are at the end of 'PL'; if end of line found first, assume you are at the end of the header for the procedure or function declaration; then continue.

7. 'of' Expected :: insert and continue.

8. '(' Expected :: insert and continue.

9. '[' Expected :: insert and continue.

10. ']' Expected :: insert and continue.

11. 'end' Expected :: insert and continue.

12. ';' Expected :: insert and continue.

13. Integer Expected :: convert number found to integer and continue; or if no number found at all, assume there was one and continue anyways.

14. '=' Expected :: insert and continue.

15. 'begin' Expected :: insert and continue.

16. Error in Declaration Part :: skip to beginning of next declaration or 'begin', whichever comes first.

17. ',' Expected :: insert and continue.

18. ':=' Expected :: insert and continue.

19. 'then' Expected :: insert and continue.

20. 'until' Expected :: insert and continue.

21. 'do' Expected :: insert and coninue.

22. 'to'/'downto' Expected :: insert and continue.

23. 'if' Expected :: insert and continue.

24. Error in Variable :: skip to next delimiter (blank, operator, comma, end of line, '(', ')', etc.) and continue.

25. Sign (+, -) Not Allowed :: ignore and continue.

26. Missing Result Type in Function Declaration :: assume there was one and continue.

27. String Constant Must Not Exceed Source Line :: assume that the end of the string was reached and continue.

28. Too Many Errors on This Source Line :: if there were more than three errors, then ignore the rest of the line and start on the next line 'fresh'.

29. ';' Found Before 'else' :: delete and continue.

30. Unmatched ''' (quote) For String :: go to the end of the line assume that the end of the string has been reached, and continue.

31. '.' Expected :: insert and continue.

EXERCISE I

EXAMPLE:

```
if i > 10
   then begin
          i := 0;
          x := x * x
        end
   else begin
          i := i + 1;
          x := y
        end;
```

PASS 1:

```
if E
   then begin
          ID := E;
          ID := E
        end
   else begin
          ID := E;
          ID := E
        end;
```

PASS 2:

```
if E
   then begin
          S;
          S
        end
   else begin
          S;
          S
        end;
```

PASS 3:

```
if E
   then S
   else S;
```

PASS 4:

```
S;
```

EXERCISE II

EXAMPLE :

```
program one (input, output);

var     i : integer;
begin
        write ('Number please: ');
        while not eof(input) do
        begin
                read (i)
                writeln ('That was a ', i)
                write ('Number please: ')
        end
end.
```

Result after Parsing :

```
P[ PH[ program id ( id, id ) ] ;
   B[ VD [ var id : t ] ;
      begin
        Sl[ id ( CS[ ' characters ' ] ) ] ;
        S5[ while e do
            S2[ begin
                  Sl[ id ( id ) ]
**ERROR           ; Sl[ id ( CS[ ' characters ' ], id ) ]
**ERROR           ; Sl[ id ( CS[ ' characters ' ] ) ]
                  end ]
          ]
      end ]
. ]
```

Error Messages from the previous example :
line 9, error 12
line 10, error 12
Please note that the line numbers refer to the place in the source where the error was detected.