Syntactic diagrams as a tool 
for solving text-processing problems

Carlos Iván Chesñevar
Departamento de Matemática, Universidad Nacional del Sur, 
Alem 1253, (8000) Bahía Blanca, ARGENTINA. 
E-mail: ccchesne@arcriba.edu.ar

1. Introduction

Most introductory undergraduate courses in programming include the resolution of text-processing problems as part of their curricula. These problems are usually to be solved in Pascal (or a Pascal-like language), and demand an intermediate skill of the students for handling sequential-access files using basic io statements, such as read and write. Typical examples of this kind of problems are generating a listing of the words in a text file or evaluating an arithmetic expression read from the keyboard.

We claim that these problems can be grouped into a general class of pattern-matching problems, most of which can be solved using syntactic diagrams. Our approach has some interesting features, such as compelling the student to face text-processing problems using top-down programming as a natural choice, and providing him with a useful tool for formalizing the resolution of text-processing problems.

2. Motivations

Solving text-processing problems as those presented above is not an easy task for those students who deal with programming for the first time. Many well-known textbooks on Pascal programming [2, 3] suggest how to cope with text-processing problems by applying top-down design and stepwise refinement.

Although students tend intuitively to write separate procedures, descomposing the main problem into smaller ones, they get stuck when they try to paste the procedures together and make them work as a whole. The main difficult lies in the fact that the resulting procedures are so tightly interlocked, that any modification or error in one of them affects necessarily the behavior of the others.

As an example, we can consider writing a program to evaluate arithmetic expressions read from the keyboard. 1 After having written separate procedures such as readTerm, read_factor, and read_expression, if it turns out that an extra character has been read somewhere (perhaps because a repeat-until was used instead of a while-do), none of the procedures will work properly, since they will be always processing a character ahead in the expression being read. This shows that even though descomposing the original problem into subproblems may be straightforward in some cases, it is usually not so easy to integrate these subproblems together.

Stepwise refinement and top-down design should not be blamed for causing these difficulties. In fact, the intrinsic complexity of exercises of this kind is at least equivalent to that associated with writing a deterministic finite automaton (DFA) for recognizing a regular expression. We know from experience that intuition alone does not suffice for designing a complex DFA. In this respect, Kleene’s theorem helps us turn this design into a mechanical, top-down procedure. As we will see next, a similar approach can be given for solving text-processing problems using syntactic diagrams.

3. Syntactic diagrams

Syntactic diagrams constitute an alternative notation to BNF for characterizing context-free grammars. Syntactic diagrams were introduced by Conway, and became widely known in the computer science literature since Wirth [4] used them for defining the syntax of PASCAL.

Look-ahead parsing is a technique used in compiler design for implementing top-down parsers. This technique allows us to turn a syntactic diagram representing a particular sequence of symbols into an algorithm for parsing that sequence.

1Keyboard input can be seen as a special kind of text file.
We will use both syntactic diagrams and look-ahead parsing for solving text-processing problems. First, we will characterize the text patterns at issue in terms of syntactic diagrams. Once the corresponding diagrams have been obtained, we will implement parsing procedures associated with these syntactic diagrams using look-ahead parsing. As a result, we will be able to build parsers for the text patterns we are going to work with. Besides, look-ahead parsing will automatically solve the task of determining how the parsing procedures should interact in the main program.

Additional code can be then attached to the resulting procedures, in order to cope with those aspects of the problem not associated with parsing. Descomposing text patterns into syntactic diagrams in a top-down fashion plays an important role, since this will help identify where additional code is needed.

We will now briefly describe how look-ahead parsing works (a complete and more formal description can be found in [1]). Basically, this technique consists in always reading an input symbol ahead in the sequence to be parsed, so that the parser can check if the symbol read was the expected one. There must be no ambiguities, since we are not allowed to backtrack on the sequence parsed so far.

We can outline a strategy for automatically writing parsing Pascal procedures from syntactic diagrams using look-ahead parsing. This strategy can be summarized as follows:

1. Every syntactic diagram for a text pattern $P$ will correspond to a Pascal procedure with the header

   
   \begin{verbatim}
   Procedure P(Var ch:char).
   \end{verbatim}

   This procedure will be the parser associated with the syntactic diagram for $P$.

2. The body of every procedure $P$ can be filled in by turning its associated syntactic diagram into Pascal code. We can distinguish three basic cases in a syntactic diagram: sequencing, decision and iteration. Figure 1 shows how to determine the corresponding pseudo-Pascal code for each of these cases (ovals stand for both terminal and non-terminal symbols).

   Every symbol $S$ appearing in the syntactic diagram for $P$ will be translated into a procedure call associated with the syntactic diagram for $S$.

3. After having completed this translation, every procedure call $P_i(ch)$ that stands for a terminal symbol (character) must be replaced by a read(ch) statement. (The end-of-line character deserves a special treatment, since it must be skipped using readln).

4. Finally, the main program will include “booting” code for reading the first symbol in the text $t$ to be processed, and a calling statement invoking the procedure associated with the syntactic diagram for $t$.

![Syntactic diagrams and its associated pseudo-Pascal codes](image)
Now, we will present two sample programming exercises about text processing, and will see how they can be formalized and solved in terms of syntactic diagrams.

4. Two examples

**Example 1:** Write a program to read a sentence from the keyboard, and count the number of words and letters in the sentence. Words in the sentence are separated by a comma or a semicolon, followed by one or more blanks. The sentence ends with a period.

First, we will determine the corresponding syntactic diagrams (see figure 2). We assume that the sentence has at least one word, and every word consists of one or more letters. Then, for each diagram, we will write its associated Pascal procedure (the `Letter` procedure is not listed since it involves only a single `read` statement). The resulting code is shown in the next page.

```
Procedure Sentence(Var ch:char);  
Begin  
  Word(ch);  
  While (ch = ',' or (ch=';')) do  
    Begin  
      Sep(ch);  
      Word(ch);  
    End;  
  End;  
  read(ch); {skip period}  
End;  

Procedure Word(Var ch:char);  
Begin  
  Letter(ch);  
  While ch in ['A'..'Z','a'..'z'] do  
    do Letter(ch);  
End;  

Procedure Sep(Var ch:char); {..main program..}  
Begin  
  Case ch of  
    ',',': read(ch);  
    ';': read(ch);  
  End;  
  read(ch); {1st.blank}  
  While ch<>'' do  
    read(ch); {skip next blanks}  
End;  
```

Now, we will attach to these procedures additional statements and parameters for counting the number of words and letters in the sentence. The procedures `Sentence` and `Word` happen to be the ones involved in reading words and letters. Thus, some additional code must be attached to them. They could be rewritten as follows:
Our second example is an adapted version of a text-processing problem presented in [2].

**Example 2:** A text file \( f \) contains a list of student names, formatted as shown below. First and last name are separated by one or more blanks. Every text line contains only one student name.

```
John Smith
George Randall
...
```

Write a program to generate a new text file \( g \) from \( f \), with the following format:

```
Smith, J.
Randall, G.
...
```

As in the first case, we will first determine the corresponding syntactic diagrams (see figure 3).

For the sake of simplicity, we assume that an end-of-line character follows immediately the last name in every text line, and that both \( f \) and \( g \) are global variables. The `Letter` procedure stands again for a single `read` statement. The resulting parsing Pascal procedures associated with the diagrams are shown in the next page.
Procedure Student_name(Var ch:char); 
Begin 
  First_name(ch); 
  Blanks(ch); 
  Last_name(ch); 
  Readin(f); {skip EOLN} 
  Read(f,ch); {next char} 
End;

Procedure First_name(Var ch:char); 
Begin 
  Letter(ch); 
  While ch in ['A'..'Z','a'..'z'] 
  do Letter(ch); 
End;

Procedure Last_name(Var ch:char); 
Begin 
  Letter(ch); read(f,ch); 
  While not eoln(f) do While ch=' ' do 
    Letter(ch); read(f,ch); 
End; 

Procedure Blanks(var ch:char); 
Begin 
  Letter(ch); 
  read(f,ch); 
  While not eoln(f) do While ch=' ' do 
    Letter(ch); 
End;

Procedure Student_List(Var ch:char); {..main program...} 
Begin 
  Student_name(ch); 
  While ch in ['A'..'Z','a'..'z'] read(f,ch); 
  do Student_name(ch); 
  Student_List(ch); 
End; 

In order to generate the text file \textit{g}, we will rewrite the procedures \texttt{Student}\_\texttt{name} and \texttt{Last}\_\texttt{name} as follows:

Procedure Student_name(Var ch:char); 
Var Initial:char; 
Begin 
  write(g,ch); {write 1st.letter} 
  Initial:=ch; {store initial} Letter(ch); 
  First_name(ch); {skip 1st.name} 
  Blanks(ch); 
  Last_name(ch); {write last name} 
  Readin(f); {skip EOLN} 
  Read(f,ch); {next char} 
  write(g,Initial); {write initial} 
  write(g,','); write(g,ch); {write last letter} 
  write(g,Initial); write(g,'); 
  writeln(g); {go to next line} 
End;

The main program should also include now a \texttt{rewrite(g)} for initializing the file \textit{g}.

5. Conclusions

The use of syntactic diagrams as a tool for solving text-processing problems was tested during three semesters in introductory programming courses with excellent results.

Students were compelled to formalize the directions of the exercises before solving them. This helped them find out whether some directions were ambiguous, or whether additional information was needed, as they wrote the corresponding syntactic diagrams. Furthermore, designing the correct syntactic diagrams turned out to be the central point in solving most text-processing problems. Students improved also their skills in writing programs to handle sequential files. Since turning syntactic diagrams into algorithms is a rather mechanical task, more complicated exercises—such as writing a parser for a small subset of Pascal—could be introduced.

6. References