SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 13, 163–179 (1983)

A Specification Schema for Indenting Programs

PRABHAKER MATETI*

Department of Computer Science, University of Melbourne, Parkville, Victoria, Australia

SUMMARY

A two level specification of the functional behaviour of a class of indenting programs for Pascal is presented. The transformation that these programs perform on the input text is a composition of splitting input lines, altering the blank space between lexical tokens and computing the margin required in front of each of the split lines. The high level specification is given as a stylized Pascal grammar in Extended BNF. In contrast, the low level specifications, which are operationally closer to a program, and which define how syntactically invalid text is dealt with, require several mathematical functions that capture the essence of these basic transformations. The specifications of an indenting program for Pascal are then obtained as a further elaboration of these functions. Most indentation styles appearing in the literature can be specified with precision using methods developed in this paper. Our experience in this case study indicates that although specifications for real-life programs can be given using simple mathematics, the effort required is still considerable.

KEY WORDS Functional specifications Correctness proofs Pretty-printing Pascal

PREFACE

The present paper is one of a triplet on an indenting program for Pascal. We undertook this exercise with three objectives in mind:

1. The literature sadly lacks real-life programs whose correctness is established by proof rather than by testing. On the other hand, those who have practised proving correctness have been raising the hopes of the readers to such an extent that a single mistake in a published proof gets the widest adverse publicity. We hope that our indenting program and its specifications and proof will serve as examples in this regard.
2. The practising programmer, we find, often uses the lowest level of formalism whereas a student who has just been through correctness methods employs formidable notation and an excess of formalism. The right level for a given program escapes both. It is not easy to say what is a right level. This can only be communicated through examples.
3. There is a myth that giving precise specifications for 'real-life' programs is often not possible. We are quite willing to accept this as a definition of 'real-life' programs but not as a corollary. Another myth is to equate precision with formalism. We hope that these papers will serve as examples where sufficient precision is attained with very little formalism.

Only the reader can tell how far we succeed in fulfilling our objectives.

* Present address: Department of Computer Engineering, Case Western Reserve University, Cleveland, OH 44106, U.S.A.

0038–0644/83/020163–17$01.70 *Received 15 September 1980*

1. INTRODUCTION

That written material expected to be read by humans should be laid out with thought and care is widely appreciated. Yet the layout of many computer programs is poor. To make matters worse, programs written in modern programming languages have many nested levels of control structures and declarations. While compilers for these languages accept 'free-format' input and can distinguish the nesting regardless of how the text input is laid out, most humans are yet to adapt themselves in this fashion.

Laying out the text of a program so that its structure is readily apparent has come to be called 'pretty-printing'. Many sets of rules for pretty-printing exist (e.g. References 1–3). These rules range from such typewriting conventions as always following a comma by a blank and flanking an equality sign by blanks to insisting that reserved words such as **goto** appear only at the beginning of a line and never hidden somewhere in the middle of a line. Much of the work in the layout of a program text is routine once a set of systematic pretty-printing rules is chosen. In fact, several programs that pretty-print the given input exist.

In this paper, we limit ourselves to programs written for Pascal, and use the less pretentious word 'indenting' in preference to 'pretty-printing'. We develop the basic mathematical functions required to specify precisely the input-to-output transformation performed by a class of indenting programs. The companion paper[4] proves the correctness of an indenting program meeting the specifications developed here, and Reference 5 discusses global issues about the program.

The indentation scheme embodied in our specifications below has evolved over a period of years accommodating and adapting the many schemes proposed in the literature. The author finds it satisfactory but is aware of others who do not. The goal of this paper is not to promote this scheme but to show that specification for such programs can be developed with sufficient precision employing simple mathematical notions. Section 2 discusses our expectations of indenting programs. Section 3 establishes notation. Section 4 gives the input-to-output transformation performed by these programs using the syntax definition of Pascal. Section 5 specifies the transformation independently of this syntax definition. Section 6 shows that if the input file contains a legal construct of Pascal then the specifications of Sections 4 and 5 are equivalent.

2. WHAT SHOULD INDENTING PROGRAMS DO?

The specifications of a program are simply our requirements and expectations of it but stated precisely without ambiguity. We ignore certain specifications of a program such as that its length be so much, or that it be written in language X without **goto**s. Instead we will concentrate only on the relationship between the input and output of indenting programs. Such specifications are called functional specifications.[6]

We list some of our expectations of indenting programs below.

1. The most obvious and yet oft-forgotten requirement is that the output of an indenting program should be 'lexically equivalent' to the text input given. Should indenting programs accept only syntactically correct text? No. We believe that indenting programs must accept any text input; if the input happens to be a syntactically correct program, we then expect its output to be properly indented. And if the input is not syntactically correct, the output text should be indented as reasonably as possible. A notion of reasonableness underlies our low

level specifications. The main reasons for insisting that indenting programs also accept 'incorrect' text are:

(i) Syntactic checking unnecessarily overburdens indenting programs.
(ii) Properly indented text helps us quickly identify syntax errors.
(iii) There are many variations of Pascal language in existence.

In fact, only minor modifications should be sufficient to produce an indenting program for other Pascal-like languages.

2. The output from an indenting program should appear 'properly indented'. This notion is made precise in later sections. (The particular indentation scheme that we suggest may not appear 'pretty' to some, but we remind that our interest in the scheme here is only as a concrete running example.) Proper indentation involves essentially three independent activities:
   (i) Each line should be started at the appropriate left margin.
   (ii) Certain constructs of the language should not be hidden in the middle of a line. For instance, it appears important that reserved words such as **while**, **repeat**, **procedure** always appear at the beginning of a line and are never embedded in a line. Reading many Pascal programs convinced us that no line of a listing should contain more than one statement. Multiple assignments on one input line should be split up.
   (iii) Adjusting the inter-word blank spacing so that it is visually appealing. Subjective preferences and special circumstances abound in this matter, and we specify here that this spacing be left unaltered except for splitting.

   We believe that each input line should generate an integral number of output lines. Combining two or more input lines and then splitting them up is often unsatisfactory and leads to complicated 'control language' to specify (in the program being indented) how the lines are to be split.
3. We also believe that indenting programs which produce output always different from their inputs are undesirable. More specifically, if we feed the indented text back to the indenting program as input, the output must be identical to the input. This characteristic of indenting programs is important from a psychological point of view.

It should be borne in mind that no matter how well we specify the input to output transformation whether an alleged indenting program should indeed be called an indenting program, and for what language, has to be judged by subjective considerations. For instance, our definition of lexical analysis is sure to startle some.

Figure 1 gives an example function indented according to our specifications. There are many inputs which produce that indented output. For the sake of concreteness, assume that the input was the text of the Figure shown, but with all leading white space in each line deleted. The reader is encouraged to compute the various functions and predicates that we define below on this input.

3. NOTATION

We denote by $\backslash b$, $\backslash t$, $\backslash n$, $\backslash e$ the characters blank, tab, end-of-line and end-of-file marker, respectively. The first three of these characters are referred to as white characters; we denote by $\%$ any one of these. For simplicity in this paper, we replace each tab by a fixed number, say 8, of blanks and assume from now on that tabs do not occur. By *white space* we mean any (possibly mixed) string of white characters. A *line* is

```
function nexttoken : token;
    var
        i, j, d : cx;
        t       : token;
    procedure dlmtoken;
        begin
        end;
    procedure stdtoken;
        var
            ctemp : packed array [1..tknlMAX] of char;
            k     : cx;
        begin
        end;

    procedure gettoken;
        (* Changes the following global vars :
        .    nextcx     of main prog
        .    tox        of main prog
        .    j, t       of nexttoken
        *)
        var
            i : cx;
            d : (1..2);
        begin
        i := nextcx;
        while  c[i] in WHITECHARS do i := i + 1;
        j := i;
        while  not (c[j] in DELIMITERS)  do
            j := j + 1;
        if  i = j  then begin
            dlmtoken;
            j := j + d
            end
        else
            stdtoken;
        end  (* gettoken *);

    begin
    tox := nextcx - 1;
    while  nextcx > lastcx  do
        begin
        ...
        end;
    ...
    gettoken;
    ...
    nexttoken := t;
    end;
```

Figure 1. An example of an indented function

a string, not containing end-of-lines or end-of-file markers, followed by the end-of-line character. A *file* is a sequence of lines followed by the pseudo-line containing exactly the single character \e.

We deal with several kinds of sequences. We adopt the convention that any single object is also a sequence of length one consisting of that object. The concatenations of strings, segment sequences and token sequences are denoted by |, ! and ∘ respectively. Note that sequences of lines, or of segments are also strings. We use regular expression notation when requiring sequences of a certain pattern. Thus, x ** k stands for the sequence x repeated k times, and x* stands for x ** k, for some $k \geqslant 0$. Unless explicitly stated otherwise, by string we mean a string of characters free of \e. We show strings enclosed in double-quotes. A string x is a prefix of z if $z = x|y$ for some y; x is a suffix of z if $z = y|x$, for some y. The words prefix and suffix have analogous meaning when referring to other kinds of sequences. Empty string, token sequence and segment sequences are denoted respectively by " ", 00 and 0ss.

The specifications require many predicates and mathematical functions. We use names with upper-case letters in them for these. In the definitions read '::=' as 'is defined as'.

4. HIGH LEVEL SPECIFICATIONS

In this section we specify the layout of programs using the Extended BNF grammar[7] of Pascal. As the lexical structure of Pascal is left undefined there we expect the reader to use his own intuitive understanding of how a string is mapped to a token sequence, for the time being. We also ignore, until the next section, the presence of comments, as does the above syntax definition. We also make minor changes to the grammar. For instance, all occurrences of terminal strings are replaced by non-terminals whose names are composed of the letters *nt* followed by the name of the token (in upper case). Thus the nonterminal *nt*REPEAT produces $w|$"*repeat*", where w stands for a (possibly empty) white space

Given a string s with no white space suffix, $s = s_1|s_2|\ldots|s_k$, and the corresponding production rule $n = n_1\, n_2 \ldots n_k$, such that $n \rightarrow^* s$, $n_j \rightarrow^* s_j$, we assume that the s_j do not have white space suffix. However, the s_j may have a prefix white space. This is significant as the white space prefix of each line is, so to speak, all that matters.

We say that a given string s corresponding to a non-terminal n is 'properly laid out' starting at margin m if PLOT(n, s, m) = **true**. The definition of PLOT is given compactly in a syntax-directed way in Figure 2. Each production acts as a template for a conjunction of NEWL and PLOT predicates, which are defined below; substituting actual strings for the non-terminals gives a logical conjunction which can then be evaluated. Each line in the diagram contains one terminal (which we show by the appropriate token) or one non-terminal (and possibly a metabracket) whose indentation from the reference vertical gives the 'ruling margin' increment for it. The presence of a NEWL predicate is indicated by a \n character to the left of the reference vertical.

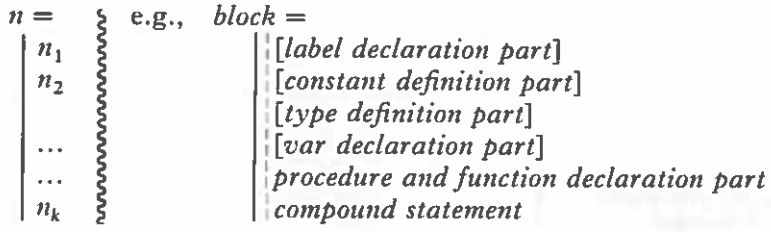
To conserve space, we have omitted from Figure 2 all productions whose specifications are of the form

```
program =
    |program heading
    |    block

program heading =
\n  |ntPROGRAM
    |    identifier
    |    ntLPAREN
    |        ident list
    |    ntRPAREN
    |    ntSEMICOLON

label declaration part =
\n  |ntLABEL
    |    label
    |    {ntCOMMA
    |    label}
    |    ntSEMICOLON

const definition part =
\n  |ntCONST
    |    constant definition
    |    ntSEMICOLON
\n  |    {constant definition
    |    ntSEMICOLON}

type definition part =
\n  |ntTYPE
    |    type definition
    |    ntSEMICOLON
\n  |    {type definition
    |    ntSEMICOLON}

var declaration part =
\n  |ntVAR
    |    var declaration
    |    ntSEMICOLON
\n  |    {var declaration
    |    ntSEMICOLON}

var declaration =
    |identifier
    |{ntCOMMA
    |identifier}
    |ntCOLON
    |    type

procedure declaration =
    |procedure heading
    |    block

procedure heading =
\n  |ntPROCEDURE
    |    identifier
    |    [formal parameter list]
    |    ntSEMICOLON

function declaration =
    |function heading
    |    block

function heading =
\n  |ntFUNCTION
    |    identifier
    |    [formal parameter list]
    |    ntCOLON
    |        type identifier
    |    ntSEMICOLON

formal parameter list =
    |ntLPAREN
    |    formal parameter section
    |    {ntSEMICOLON
\n  |    formal parameter section}
    |ntRPAREN

formal parameter section =
    |[ntVAR      ]
    |ntFUNCTION  ]
    |ident list
    |ntCOLON
    |type identifier ]
    |ntPROCEDURE
    |ident list

actual parameter list =
    |ntLPAREN
    |    expression
    |    {ntCOMMA
    |    expression}
    |ntRPAREN

factor =
    |ntLPAREN
    |    expression
    |ntRPAREN

compound statement =
    |ntBEGIN
    |statement
    |{ntSEMICOLON
\n  |statement}
    |ntEND
```

...contd

```
statement =
    |(label
    |ntCOLON
    |    unlabelled statement    |
    |unlabelled statement        )

if statement =
\n  |ntIF
    |expression
    |ntTHEN
    |    statement
\n  |[ntELSE
    |    statement]

case statement =
\n  |ntCASE
    |expression
    |ntOF
    |    case
    |    {ntSEMICOLON
\n  |    case}
    |ntEND

case =
    |[case label list
    |ntCOLON
    |    statement]

while statement =
\n  |ntWHILE
    |expression
    |ntDO
    |    statement

repeat statement =
\n  |ntREPEAT
    |    statement
    |    {ntSEMICOLON
\n  |    statement }
\n  |ntUNTIL
    |expression

for statement =
\n  |ntFOR
    |identifier
    |ntASSIGN
    |for list
    |ntDO
    |    statement

with statement =
\n  |ntWITH
    |variable
    |{ntCOMMA
    |variable}
    |ntDO
    |    statement

goto statement =
\n  |ntGOTO
    |label


scalar type =
    |ntLPAREN
    |    ident list
    |ntRPAREN

record type =
    |ntRECORD
    |    field list
    |ntEND

record section =
    |[ident list
    |ntCOLON
    |    type]

variant part =
\n  |ntCASE
    |    (identifier
    |    ntCOLON
    |        type identifier
    |        ntOF    |
    |    type identifier
    |    ntOF    )
    |    variant
    |    {ntSEMICOLON
\n  |    variant}


variant =
    |case label list
    |ntCOLON
    |    ntLPAREN
    |        field list
    |    ntRPAREN
```

Figure 2. High level specifications of our indenting scheme



where all the n_j are right next to the references vertical and have no $\backslash n$ character appearing to its left.

4.1. The predicate properly-laid-out

For example, we say that a string named *rptst* produced by the non-terminal repeat statement is properly laid out at m if (1) the reserved word *repeat* is the first word on that line starting at a margin of m, (2) the statements of the loop body obey the rules of Figure 2 recursively, (3) the reserved word *until* is the first word on that line starting at margin m and (4) the expression after *until* obeys the rules recursively. More formally, if the instance *rptst* we are considering had two statements, say *st*1 and *st*2, in its body and *exp* as its expression, and *w*1, *w*2 are white spaces, i.e.

$$rptst = w1 \mid \text{"}repeat\text{"} \mid st1 \mid \text{";"} \mid st2 \mid w2 \mid \text{"}until\text{"} \mid exp$$

then the logical conjunction given by the diagram is:

PLOT(*repeat statement*, *rptst*, *m*) =
PLOT(*nt*REPEAT, *w*1 | "*repeat*", *m*) & NEWL(*w*1 | "*repeat*")
& PLOT(*statement*, *st*1, *m* + UOI)
& PLOT(*ny*SEMICOLON, ";", *m* + UOI)
& PLOT(*statement*, *st*2, *m* + UOI) & NEWL(*st*2)
& PLOT(*nt*UNTIL, *w*2 | "*until*", *m*) & NEWL(*w*2 | "*until*")
& PLOT(*expression*, *exp*, *m* + UOI)

where UOI stands for the unit of indentation. We now define PLOT and NEWL more precisely.

Definition of PLOT

PLOT is a predicate on triplets consisting of a non-terminal, a string and a margin width.

1. PLOT(n, $s \mid c$, m) ::= PLOT(n, s, m), where c is either % or $\backslash e$. Thus we assume below that s has no trailing white space.
2. PLOT(n, s, m) ::= **false**, if n does not produce s. Thus we further assume below that $n \rightarrow^* s$.
3. PLOT(*empty*, " ", m) ::= **true**, for all m.
4. PLOT(t, s, m) ::= ISAT(s, m), where t is a (non-terminal) token.
5. Let $n = n_1 n_2 \ldots n_k$ be a syntax rule of the language. Let $s, s_1, s_2, \ldots, s_k$ be corresponding strings generated from the non-terminals n and the n_i. Then

PLOT(n, s, m) ::=
PLOT(n_1, s_1, m)
& PLOT(n_2, m+RMI(n_1, n_2))
& ...
& PLOT(n_k, s_k, m+RMI(n_1 n_2 ... n_{k-1}, n_k))
& NEWL(s_{i1}) & NEWL(s_{i2}) & ... & NEWL(s_{ip})

where RMI(n_1 ... n_{j-1}, n_j) is the ruling margin increment for the n_j as shown in Figure 2 for that production rule and only the nonterminals n_{i1}, n_{i2} ... n_{ip} has the \n to the left of the reference vertical.

Thus, the above PLOT(*repeat statement*, *rpstst*, m) would be **true** for $m=0$, for example, if *st*1 and *st*2 were empty strings, *w*1 and *w*2 were equal to \n and *exp* did not contain \n.

Definition of ISAT

ISAT(s, m) ::= **true** iff either $s = \%\ *\,|\,\backslash n\,|\,\backslash b\ **\ m\,|\,c\,|\,y$, for some string y and non-white character c, or s does not contain \n.

If s does have a \n, then ISAT(s, m) will be true iff the left-most non-white character of s is exactly m blanks away from the preceding \n.

Definition of NEWL

NEWL(s) ::= **true** iff $s = \%\ *\,|\,\backslash n\,|\,x$, for some string x.

That is, NEWL(s) is true iff s has a \n preceding which there are no non-white characters.

4.2. The indented file

We say that a string s is lexically equivalent to t if both produce the same sequence of tokens. More formally, s and t are lexically equivalent if by replacing the inter-token white space by a single blank, and by deleting any white space prefix/suffix, if any, the resulting strings $s1$ and $t1$ become equal. (See also the next section.)

Given a file FI (the input) an indenting program should produce file FU such that

1. for each i, $1 \leqslant i \leqslant$ number of lines in FI, there exists a u, $1 \leqslant u \leqslant$ number of lines in FU, such that FI[1 .. i] and FU[1 .. u] are lexically equivalent where F[1..n] stands for the first n lines of file F,
2. PLOT(nt, \n|FU, 0) = **true**, and
3. no file with fewer lines than are in FU satisfies the above,

whenever FI is a sentence corresponding to a non-terminal nt of Pascal grammar. This is the specification of indenting programs that appeals to us. Part (3) guarantees that input lines are not split up unnecessarily. In part (2), a \n is prefixed to FU so as to treat the end of line character as a 'new line' character. Without this \n, a NEWL predicate might be false even though the first token of the very first line is at the correct margin. Note that the behaviour of the indenting program is unspecified when FI does not contain a legal construct of Pascal. Note also that part (1) of the specification implies that FU will have at least as many lines as in FI. It also rules out recombination of input lines and then splitting them up into output lines.

5. LOW LEVEL SPECIFICATIONS

We now develop a set of specifications that appear independent of Pascal grammar. Whereas the previous section left undefined the behaviour of indenting programs when invalid constructs of Pascal are given as input, this section specifies what transformation is to be done for an arbitrary input string, and hence an arbitrary sequence of tokens. This latter transformation is designed to coincide with that given above for all syntactically valid constructs of Pascal. An outline of a proof of this fact is given in the next section.

5.1. Lexical analysis

Lexical analysis is a process that breaks up strings into sequences of 'words', more widely known as tokens. We say a character string w is a *word* if TKN(w) is not undefined, where TKN is a partial function that maps character strings to tokens as elaborated below.

Definition of TKN

For a given string w, TKN(w) is defined as t if there is a pair $\langle w, t\rangle$ in one of the following sets; otherwise TKN(w) is undefined.

1. Let w be free of delimiters, namely the following characters: blank, tab, end-of-line, end-of-file, parentheses, braces, semicolon, colon, asterisk, quote and period. (Other conventional delimiters do not concern us.)

$\{$ ⟨"*procedure*" , PROCEDURE⟩,
⟨"*function*" , FUNCTION⟩,
⟨"*program*" , PROGRAM⟩,
⟨"*forward*" , FORWARD⟩
⟨"*repeat*" , REPEAT⟩,
⟨"*record*" , RECORD⟩,
⟨"*extern*" , EXTERN⟩,
⟨"*while*" , WHILE⟩,
⟨"*until*" , UNTIL⟩,
⟨"*label*" , LABEL⟩,
⟨"*const*" , CONST⟩,
⟨"*begin*" , BEGIN⟩,
⟨"*with*' , WITH⟩,
⟨"*type*" , TYPE⟩,
⟨"*then*" , THEN⟩,
⟨"*goto*" , GOTO⟩,
⟨"*else*" , ELSE⟩,
⟨"*case*" , CASE⟩,
⟨"*var*" , VAR⟩,
⟨"*for*" , FOR⟩,
⟨"*end*" , END⟩,
⟨"*of*" , OF⟩,
⟨"*if*" , IF⟩,
⟨"*do*" , DO⟩,
⟨*other w* , ORDINARY⟩ $\}$

2. Let w contain delimiters.

$$\begin{aligned}\{\ &\langle ";" \quad , \text{ SEMICOLON}\rangle,\\ &\langle "'" \quad , \text{ QUOTE}\rangle,\\ &\langle ":" \quad , \text{ COLON}\rangle,\\ &\langle "(" \quad , \text{ LPAREN}\rangle,\\ &\langle ")" \quad , \text{ RPAREN}\rangle,\\ &\langle "\{" \quad , \text{ COMBGN}\rangle,\\ &\langle "\}" \quad , \text{ COMEND}\rangle,\\ &\langle "*" \quad , \text{ ORDINARY}\rangle,\\ &\langle "." \quad , \text{ ORDINARY}\rangle,\\ &\langle \backslash e \quad , \text{ ENDFILE}\rangle,\\ &\langle "(*" \quad , \text{ COMBGN}\rangle,\\ &\langle "*)" \quad , \text{ COMEND}\rangle,\\ &\langle " ::= " \ , \text{ ASSIGN}\rangle\ \}\end{aligned}$$

Note the obvious fact that any string consisting of exactly one non-white character is a token.

The essence of lexical analysis is captured in LEX which produces the token sequence of z in the context of a token sequence T already produced. Recall that 00 denotes the empty token sequence, and $\circ$ denotes concatenation of token sequences.

Definition of LEX

1. LEX(T, " ") ::= 00.
2. LEX(T, "%" | y) ::= LEX(T, y).
3. Let z be free of leading white space. Then LEX(T, z) ::= $t \circ$ LEX(T $\circ\ t, x$), where $z = w \mid x$, and w is the longest prefix of z such that TKN(w) is defined. The token t is TKN(w) unless (i) TKN(w) $\neq$ COMEND and T = S $\circ$ COMBGN $\circ$ ORDINARY*, or (ii) TKN(w) $\neq$ QUOTE and T = S $\circ$ QUOTE $\circ$ ORDINARY*, for some S free of unmatched QUOTES. In the latter two cases, t = ORDINARY.

Definition of TKNSEQ

TKNSEQ(z) ::= LEX(00, z).

Since syntactically correct Pascal programs have one of the delimiters immediately following reserved words, we do not risk non-recognition of such words by ignoring other conventional delimiters (such as operators). Lexical analysis performed by a typical Pascal compiler otherwise matches with LEX except when dealing with comments and strings. In compilers, comments are simply swallowed and the strings are returned as tokens. For our purpose here,, however, the layout of comments is important. It would seem logical then to split a comment into three tokens, namely, COMBGN, the comment contained, followed by COMEND. Since comments can span several lines, this decision would complicate the definitions of functions given in subsequent sections. Thus, we define LEX(T, z) based on the longest prefix of z that is a word, and change the token to ORDINARY if T has an unmatched COMBGN or QUOTE.

For example, the string "(*(*)" is broken into words as "(*" | "(*" | ")" giving the token sequence T1 $\circ$ T3 is a reduced token sequence of T1 $\circ$ T2 T3 if T2 is the token "'*doesn*''*t it*'" is tokenized as "'" | "*doesn*" | "'" | "'" | "*t*" | "*it*" | "'". Note, however, that our line splitting rules do not split a Pascal string that was contained in one source

line. Good style for visual appeal demand that reserved words and comments be flanked by white spaces and we do not see the need to rectify these 'anomalies'.

5.2. Reduced token sequences

We introduce the notion of 'reduced' token sequences which makes it easy to define the functions that give the left margin width of output lines. Intuitively speaking, a token sequence T1 ∘ T3 is a reduced token sequence of T1 ∘ T2 ∘ T3 if T2 is the token sequence of a syntactically 'sensible' Pascal statement. One might insist that T2 correspond to a syntactically correct statement. This, we believe, is overburdening the indenting programs; guaranteeing syntactic correctness is the function of a compiler, not indenting programs. What is syntactically sensible is made clear in the way the mathematical function RED maps a given token sequence to its reduction.

The P, Q, R, S and T below denote token sequences, and *s* and *t* denote single tokens. Expressions of the kind "if T = R ∘ DECL ∘ S for some R and S" are abbreviated as 'if T = R ∘ DECL ∘ S'. The special tokens DECL and PF are devised for the purposes of the RED function below and do not have corresponding words.

Definition of RED (*see notes below*)

1. RED(T ∘ *t*) ::= RED(RED(T) ∘ *t*)
   Thus we assume below that the sequence denoted by T is not reducible any further.
2. Let *t* = PROCEDURE, FUNCTION or PROGRAM. Then RED(T ∘ *t*) ::= R ∘ DECL ∘ PF, where R = S if T = S ∘ DECL, R = T if T does not end with either DECL or LPAREN; ::= T, if T does end with LPAREN.
3. Let *t* = LABEL, CONST, TYPE or VAR. Then RED(T ∘ *t*) ::= R ∘ DECL, where R = S if T = S ∘ DECL, R = T if T does not end with either DECL or LPAREN; ::= T, if T does end with LPAREN.
4. Let *t* = FORWARD, or EXTERN. Then RED(T ∘ *t*) ::= S, if T = S ∘ PF; ::= T, otherwise.
5. RED(T ∘ BEGIN) ::= S ∘ BEGIN, if T = S ∘ PF, or if T = S ∘ PF ∘ DECL; ::= T ∘ BEGIN, otherwise.
6. Let *t* = RECORD, LPAREN, REPEAT, CASE, DO, THEN or COLON. Then RED(T ∘ *t*) ::= T ∘ *t*.
7. RED(T ∘ OF) ::= S ∘ CASE if T = S ∘ CASE ∘ COLON; ::= T, otherwise.
8. Let the pair ⟨*t*, *s*⟩ be one of ⟨RPAREN, LPAREN⟩, ⟨UNTIL, REPEAT⟩. Then RED(T ∘ *t*) ::= R, if T = R ∘ *s* ∘ S where S does not have any tokens *s*; ::= 00, otherwise.
9. RED(T ∘ END) ::= R, if T = R ∘ RECORD ∘ S where S is free of RECORDS; ::= P, if T is free of RECORDS and T = P ∘ *s* ∘ Q where *s* is either a BEGIN, or a CASE and Q does not have any of these tokens; ::= 00, otherwise.
10. RED(T ∘ ELSE) ::= R ∘ ELSE, if T = R ∘ THEN ∘ S where S is free of THENS; ::= ELSE, otherwise.
11. RED(T ∘ SEMICOLON) ::= R ∘ *s*, if T = R ∘ *s* ∘ S where *s* is any token but THEN, ELSE, DO, or COLON and S is a sequence of these tokens only; ::= 00, otherwise.
12. RED(T ∘ *t*) ::= T, for any *t* not covered above.

The many cases in the definition reflect the syntax of the language. It should be clear that many illegal Pascal constructs would result in valid reduced sequences. As

mentioned before, syntax validation is not in the domain of indenting programs we are considering.

Cases 2, 3, 4 and 5 would be simpler if Pascal had a different syntax. The special token DECL indicates that declarations (of labels, constants, types, variables and procedure/functions) are due next. If the last token of T is LPAREN, which can arise in a syntactically correct program only inside the parameter list, the tokens VAR, PROCEDURE and FUNCTION have no effect. The declarations end when a BEGIN is encountered; this is shown in case 5. Case 4 arises because FORWARD and EXTERN are not reserved words. They have the special meaning only when they appear immediately following the procedure headline.

Case 7 arises because of variant records with tag fields. In our specification COLON indents and it is, in this case, terminated by the OF.

5.3. Line splitting

Each split up part of a line is called a *segment*. As we shall see, there is a one-to-one correspondence between input segments and output lines. These two are in fact identical but for the prefix and suffix white spaces.

The function FIRSTSEG maps non-white prefixes of a line to its first segment, using the sets LO, and LC. The function SEGSEQ maps arbitrary strings to segment sequences. The set LO contains all (line opening) tokens whose corresponding words should always appear as the first non-white string in an output line. Similarly, the set LC contains all tokens which always close an output line but allow any immediately following comments. Thus the occurrence of a token from LO in the middle of an input line will split it just to the left of the token. The sets LO, LC are chosen to match the specifications of Figure 2.

LO ::= {PROCEDURE, FUNCTION, PROGRAM, LABEL, CONST, TYPE, VAR,
WHILE, REPEAT, UNTIL, IF, ELSE, CASE, GOTO}
LC ::= {SEMICOLON}

Intuitively, the segmentation of strings as produced by SEGSEQ can be explained as follows. Place imaginary markers as follows: (1) before the very first and after the very last characters of the string, (2) to the immediate right of every $\backslash n$, (3) to the immediate left of a token belonging to LO, and (4) to the immediate right of a token belonging to LC but skipping over comments following it. The strings thus enclosed between pairs of consecutive markers are segments. The functions FIRSTSEG and SEGSEQ imitate this process in a non-operational way.

Recall that we denote by 0ss, the empty sequence of segments, and by ! concatenation of segment sequences.

Definition of FIRSTSEG

Let w be a prefix of a line, and let Q = LEX(T, w).

1. Suppose lc ∘ COMBGN ∘ ORDINARY* is a suffix of T, where lc stands for a token from LC. Then FIRSTSEG(T, w) ::= w, if Q does not contain COMEND; otherwise FIRSTSEG(T, w) ::= x, where x is the longest prefix of w such that LEX(T, x) = ORDINARY* ∘ COMEND ∘ (COMBGN ∘ ORDINARY* ∘ COMEND)*.
2. Suppose lc ∘ COMBGN ∘ ORDINARY* is not a suffix of T. Then let x be the longest prefix of w such that LEX(T, x) does not contain (i) any token from LO except as its first token, or (ii) the subsequence lc ∘ (COMBGN ∘ ORDINARY* ∘ COMEND)* ∘ q, where $q \neq$ COMBGN. Then FIRSTSEG(T, w) = x.

The first segment function FIRSTSEG may be more complex for other indentation schemes. For instance, if we had required that the BEGIN of the code body of a procedure start on a new line, but other BEGINs need not, decomposing an input line into segments can no longer be done on the basis of a set like LO.

Definition of SEGSEQ

Let z be a sequence of lines, and w a prefix of a line.

1. SEGSEQ(" ") ::= 0ss.
2. SEGSEQ($z \mid w$) ::= SEGSEQ(z) ! w, if w is all white;
   SEGSEQ($z \mid w$) ::= SEGSEQ(z) ! DS(z, w), otherwise.
3. DS(z, w) ::= 0ss, if w is all white. Otherwise, DS(z, w) ::= u ! DS($z \mid u$, v) where u = FIRSTSEG(TKNSEQ(z), w), and $w = u \mid v$.

As an example, consider the following string, z, where the end of lines are shown explicitly.

"**if** *b*1 **then** (* *loop* *) **while** *b*2 **do begin**" ¦\n¦
" " ¦\n¦
" $x := f(x)$; (* *c*1 *) {*invariant*} $g(x)$" ¦\n¦
"**end**; (* *of while and if* *)"

The segments of SEGSEQ(z) are given below.

"**if** *b*1 **then** (* *loop* *)" !
"**while** *b*2 **do begin** "¦\n !
" "¦\n !
" $x := f(x)$; (* *c*1 *) {*invariant*}" !
"$g(x)$"¦\n !
"**end**; (* *of while and if* *)"

5.4. Indentation

Most often the indentation (i.e. the width of the left margin) of a given output line depends on the indentation of the previous line and on the reserved words occurring in it. On rarer occasions, the indentation depends also on the reserved words appearing in that line itself. An example of this is the "until". NMG(T) gives the margin the next line should have if the last token of T corresponds to the last word of the current line; CMG(T) gives the margin the current line should have *if* the last token of T corresponds to the first word of the current line. The function MG gives the actual margin of each output line.

Definition of NMG

1. NMG(00) ::= 0.
2. NMG(T) ::= NMG(RED(T)). Thus we assume below that the argument of NMG is reduced.
3. Let t = PF, or BEGIN. Then NMG(T ∘ t) ::= NMG(T).
4. Let t = DECL, RECORD, LPAREN, REPEAT, DO, CASE, THEN, ELSE or COLON. Then NMG(T ∘ t) ::= NMG(T) + UOI.

Definition of CMG

1. CMG(00) ::= 0.
2. Let t = PF, DECL or ELSE. Then CMG(T ∘ t) ::= NMG(T ∘ t) − UOI.
3. Let t = RECORD, LPAREN, REPEAT, DO, CASE, THEN or COLON. Then CMG(T ∘ t) ::= NMG(T).
4. For all t not covered above, CMG(T ∘ t) ::= NMG(T ∘ t).

It follows from the above that CMG(T) = CMG(RED(T)).

Definition of MG

1. MG(0ss) ::= 0.
2. MG(z ! x) ::= CMG(T ∘ t), where t is the first token of LEX(T, x), and T = TKNSEQ(z).

Note that MG maps segment sequences to margins in contrast to NMG and CMG which map token sequences to margins.

5.5. Final specification

Let zi ¦ \e be the input file to indenting program, and let zo be the corresponding output file of an indenting program. Then

$$zo = \text{INDENT}(zi) \text{ ¦ } \backslash e$$

is the relation between them, where INDENT(zi) ::= IND(SEGSEQ(zi)), and IND is given below.

Defintion of IND

IND maps segment sequences to sequences of lines. Let z be a sequence of segments, and x a segment.

1. IND(0ss) ::= " ", the empty string.
2. IND(z ! x) ::= IND(z) ¦ \$b$ ** MG(z ! x) ¦ psTRIM(x) ¦ \$n$, where psTRIM(x) trims x by removing all its prefix and suffix white space.

Note that psTRIM(*i*th *segment of input file*) = psTRIM(*i*th *output line*).

6. THE EQUIVALENCE OF THE TWO SPECIFICATIONS

The low level specifications coincide with the high level specifications in the following sense: Let zi ¦ \$e$ be the text input to an indenting program satisfying our low level specifications. Clearly its output zo = INDENT(zi) ¦ \$e$. Then we say that the two specifications are *coincident* if PLOT(nt, \$n$ ¦ zo, 0) = true, whenever $nt \rightarrow^* zi$. Note that if zi were not a valid construct, PLOT(nt, \$n$ ¦ zo, 0) would be false for all nt. As Figure 2 completely ignores comments, PLOT does not say how comments should be laid out, and we therefore give INDENT complete freedom in this regard.

A proof that the two specifications are coincident proceeds by induction on the syntactic structure of the input zi. As the base step, we show that if zi can be generated by one application of a production rule then zo = INDENT(zi) would satisfy the high level specifications. If, for $1 \leqslant j \leqslant k$, zu_j = INDENT(zi_j), and $N = N_1 N_2 \ldots N_k$ is a production of Pascal grammar such that $N_j \rightarrow^* zi_j$, then the induction hypothesis is that PLOT(N_j, \$n$ ¦ zu_j, 0) = true. We need to show that PLOT(N, \$n$ ¦ zo, 0) = true, where zo = INDENT(zi) and $zi = zi_1 \text{ ¦ } zi_2 \text{ ¦ } \ldots \text{ ¦ } zi_k$.

Recall that we replaced terminal symbols appearing in the right-hand side of productions by their token names and considered the latter as non-terminals. Thus any zi that can be generated in one application can contain only one token and INDENT(zi) contains no blank space in front of this token; thus PLOT(t, $\backslash n$|INDENT(zi), 0) holds for some appropriate token t.

For the inductive step, note that zo = INDENT(zi) can be divided such that $zo = zo_1 | zo_2 | \ldots | zo_k$, no zo_j contains white space suffix, and $N_j \rightarrow^* zo_j$. Further note that the zi_j, zu_j and zo_j are all lexically equivalent. Since zu_j = INDENT(zi_j), zo_j possibly differs from zu_j only in the width of the margin of each line and by containing an extra $\backslash n$ in the prefix of zo. The rest of the proof of this step follows from these observations and is simple but tedious requiring case analyses for each non-terminal N of the grammar. Here we present two such cases—one for the repeat statement and another for the procedure declaration.

Case N = repeat statement

Clearly TKNSEQ(zo_1) = REPEAT and TKNSEQ(zo_{k-1}) = UNTIL. Also, for $1 < j < k-1$, zo_j must equal $c|(zu_j$ with the margin of each line of zu_j increased by UOI blanks). Here the string c is either empty, or is $\backslash n$ depending on the segment sequence SEGSEQ(zi). If a segment boundary fell between zi_{j-1} and zi_j, and if zi_{j-1} did not end with a $\backslash n$, then $c = \backslash n$, else $c =$ "". From the definition of SEGSEQ, it follows that a segment boundary falls between z_{j-1} and zi_j either because zi_{j-1} terminated in a $\backslash n$, or in a token from LC followed by (portions of) comments, or because zi_j begins a token from LO. Since LO and LC were chosen so as to make NEWL predicates true, PLOT(*repeat statement*, $\backslash n|zi$, 0) must be true.

Case N = procedure declaration

We shall make further assumptions below for the sake of simplicity in this illustration. We have that TKNSEQ(zo_1) = PROCEDURE, TKNSEQ(zo_2) = ORDINARY (the corresponding word being the name of the procedure), TKNSEQ(zo_3) = SEMICOLON, assuming that the procedure heading has no parameters, and TKNSEQ(zo_k) = END. Further assuming that the procedure has only variable declaration part, we have TKNSEQ(zo_4) = VAR. Let $zo_5, \ldots, zo_{v-1}$ correspond to this declaration such that TKNSEQ(zo_v) = SEMICOLON, TKNSEQ(zo_{v+1}) = BEGIN. Clearly then, $zo_{v+2} \ldots, zo_{k-1}$ correspond to the code body of the procedure. Note that the value of NMG( ) will be 2*UOI starting from zo_5 until zo_v, both inclusive. After zo_{v+1} it becomes UOI and remains at least UOI until zo_k. As in the previous case, we see that the code body and the variable declaration and hence the procedure declaration thus meet the high level specifications.

7. CONCLUDING REMARKS

This section contains some remarks based on personal experience with this case study in specifying the behaviour of a medium sized program. I wrote the first version of an indenting program in late 1978 mainly as a reaction to the very long, slow and often clumsy indenting programs that were known to me at that time. A year later, I needed a class-room example of a real life program whose specification and proof are given sufficiently rigorously but with as little formalism as possible.

I began writing these specifications believing that it would take no more than 10 hours. I now estimate that about 150 hours were spent, over 9 months, in choosing the style of presentation, discovering the required functions and specifying the behaviour of the program. (The time spent in writing this paper is not included in the estimate.) In contrast, the original program was designed, written and tested in a total of 30 hours. Two revised versions of the program, eliminating many 'minor bugs' in the original, were written during the development of the specifications. A correctness proof of the last version appears in a companion paper.[4] I estimate that the two revisions were done in 20 hours. Thus in my experience, the effort required in specifying a program I thought I understood well was 3 to 4 times more than that required in designing and writing it. I believe that this factor would have been considerably higher if I had less training in this field.

One wonders if the low level specifications could have been written without a certain program in mind, or if they are needed at all. I did have a certain program in mind, and perhaps some of the inelegance is due to this fact. However, writing down these specifications exhibited the subtle errors and inelegant ways of the program that escaped my attention before. In contrast, the writing of high level specifications helped only to explain to others how this program indents. It was important to write the low level specifications because these defined how invalid input would be dealt with. Indeed, half the correctness proof of the indenting program consist of showing that low level specifications coincide with high level specifications.

It is not unfair to say that few practising programmers would be comfortable with the level of formalism used here. While there is certainly scope for improving the notations used in the paper, I believe there will be significant loss of precision with any further decrease in the level of formalism and rigour. The complexity of our specifications, however, truly reflects the complexity of any program meeting them.

This experience has been both delightful and frustrating, at times. I recommend that everyone who writes programs conduct similar experiments as often as possible. As such experimenters are well aware, specifications can and often do contain bugs just as programs do.

REFERENCES

1. P. Grogono, 'On layout, identifers and semicolons in Pascal programs', *SIGPLAN Notices*, **14**(4), 35–40 (1979).
2. H. Ledgard, A. Singer and J. Hueras, 'A basis for executing Pascal programmers', *SIGPLAN Notices*, **12**(7), 101–105 (1977).
3. A. Sale, 'Stylistics in languages with compound statements', *The Australian Computer Journal*, **10**(2), 58–59 (1978).
4. P. Mateti and J. Jaffar, 'A correctness proof of an indenting program' *Software—Practice and Experience*, **13**, (3) (1983) (to be published).
5. P. Mateti, 'Documentation of program *indent*: a model for the complete documentation of computer programs', unpublished class notes, Department of Computer Science, University of Melbourne, Parkville 3052, Australia, 1980.
6. B. H. Liskov and V. Berzins, 'An appraisal of program specifications', in P. Wegner (ed.), *Research Directions in Software Technology*, M.I.T. Press, Massachusetts, pp. 276–301, 1979.
7. N. Wirth, 'Syntax of Pascal in extended BNF', *Pascal News*, **12**, 52–53 (1978).