SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 13, 199–226 (1983)

A Correctness Proof of an Indenting Program

PRABHAKER MATETI* AND JOXAN JAFFAR

Department of Computer Science, University of Melbourne, Parkville, Victoria 3052, Australia

SUMMARY

The correctness of an indenting program for Pascal is proved at an intermediate level of rigour. The specifications of the program are given in the companion paper.[1] The program is approximately 330 lines long and consists of four modules: *io*, *lex*, *stack* and *indent*. We prove first that the individual procedures contained in these modules meet their specifications as given by the entry and exit assertions. A global proof of the main routine then establishes that the interaction between modules is such that the main routine meets the specification of the entire program. We argue that correctness proofs at the level of rigour used here serve very well to transfer one's understanding of a program to others. We believe proofs at this level should become commonplace before more formal proofs can take over to reduce traditional testing to an inconsequential place.

KEY WORDS Correctness proofs Pretty-printing Pascal

'It is one of the chief merits of proofs that they instill a certain scepticism as to the result proved.'

BERTRAND RUSSELL (1903)

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 University, Cleveland OH 44106, U.S.A.

0038–0644/83/030199–28$02.80 *Received 15 September 1980*

1. INTRODUCTION

Current literature in programming methodology urges us to switch to proving our programs correct rather than validating them by thorough testing. Yet the practical world of programming believes this to be simply 'ivory-tower' talk and considers such an attempt uneconomical. Even if we wish to ignore the economic feasibility of proofs, the very formal approach taken in the proof of small programs has made practising programmers wary of it. However the rigour with which a proof may be given can be reduced. There is an intermediate level of rigour which is more convincing than 'hand-waving' and much less formal than, say, first-order logic. Correctness proofs at this level of rigour have long been in use in dealing with combinatorial algorithms. (See, e.g. reference 2.) Most proofs of theorems in college-level mathematics are at this intermediate level. The effort required in following the correctness proof of a program at this level is only marginally greater than that in thoroughly understanding the program. However, designing, structuring and presenting such proofs still requires an effort from most of us (as we found in this case), that is far greater than in the construction of the program itself. We believe that the required effort would decrease as we gain more experience in proving the correctness of large programs.

This paper presents a correctness proof of an indenting program for Pascal at an intermediate level of rigour. The specifications of this program are given in Reference 1. We undertook this task with several objectives in mind, and as a test case for some of our beliefs:

1. The level of understanding and insight gained through correctness proofs is far greater than is possible by any amount of testing. Perhaps far more important is the ease with which such understanding can be passed from the program's author to its other readers through its proof.
2. More and more proofs of reasonably large programs should appear in the published literature in order to win over the practising programmer; economic feasibility can only be attained after they have been won over.
3. Correctness proofs of other programs (be they indenting or not, written in Pascal or not) can be structured on parallel lines to the module structure of the program. If module interfaces are kept to a minimum and if the program is designed with care, correctness proofs follow quite routinely from the program.
4. Several proofs, each at an increasing level of rigour, should be given. Each proof can be regarded as a sketch of the next higher level one, catering to the requirements of all readers, from the devout believer to the very sceptical.
5. To add to the evidence of the claim that large programs can be proved using the same basic techniques employed in proving small programs.

2. PRELIMINARIES

The indenting program we present here is written in a free-style language to emphasize the independence of the proof techniques from the specific programming language used. We ask the reader's indulgence not to get side-tracked by its syntax and control structures. The free-style language offers us notational convenience and displays the modular structure of the program more clearly than is possible, say, in Pascal. The semantics of the language should be self-explanatory in the context of our program. Neither the specification, nor the design of our program is defended here;

References 1 and 3 contains a discussion of these issues. The specifications as given in Reference 1 are prerequisities for this paper.

We present individual procedures and other relevant declarations as we deal with them in the next section. We indicate the type of a variable only when there is a possibility of confusion without it. Within each module the lines are numbered uniquely; the interested reader can put together the complete module by assembling these lines into ascending order.

We have risked using a style of writing in the proof that at times is not smooth reading to spare the reader from the dull and trivial proofs that are readily apparent from the program text and its assertions. The reader should note that it is the intermediate level of rigour we have been alluding to which makes it possible for us to skip such trivial proofs. These can become non-trivial and even interesting at higher levels of rigour.

Both the above measures are adopted primarily to reduce the length of this paper.

2.1. Notations and definitions

In what follows, we need to deal with the 'white' characters blank, tab and end-of-line. These are denoted by $\backslash b$, $\backslash t$ and $\backslash n$, respectively; the symbol % stands, in a comparison or assertion, for any of these three characters. Thus a test such as $c[i] = \%$ stands for $c[i]$ **in** $\{\backslash b, \backslash t, \backslash n\}$ and $c[i] \neq \%$ for its negation. The end-of-file marker is denoted by $\backslash e$. A string all of whose characters are white is said to be *all-white*. Thus, for example, the empty string is all-white.

A character is *ordinary* if it is neither $\backslash n$ nor $\backslash e$. A *line* is a string of ordinary characters, followed by $\backslash n$. An *input file* is a sequence of lines, followed by the 'pseudo-line' consisting of the single character $\backslash e$. Unless stated otherwise all strings and sequences can be empty. String concatenation is indicated by $\mid$. A string x is a prefix of z if $z = x \mid y$ for some y; x is a suffix of z if $z = y \mid x$ for some y. The words prefix and suffix have analogous meanings when referring to other kinds of sequences.

We use regular expression notation when requiring sequences of a certain pattern, e.g. $x ** k$ stands for the sequence x repeated k times, and $x *$ stands for $x ** k$ for some $k \geqslant 0$. Following normal convention, if $k \leqslant 0$, $x ** k$ is the empty sequence.

It will be convenient to think of the sequential input and output files as 'arrays' of lines: $Input[1..i]$ represents the first i lines of input text, and $Output[1..u]$ the first u lines that were output. However, it must be remembered that once, say, $Input[i]$ is read, the next line that can be read is $Input\ [i+1]$ and it is not possible to go back and reread $Input\ [j]$ for any $j \leqslant i$. Similarly once $Output[m]$ is output no $Output[n]$ for $n \leqslant m$ can be altered, and the next line to be output becomes $Output[m+1]$. As usual, $Input[1..i]$ and $Output[1..i]$ are empty when $i<1$.

We use the array c of characters as a line buffer. The string $c[i..j]$ stands for $c[i] \mid c[i+1] \mid \ldots \mid c[j]$; it is empty if $i>j$. The notation $s[1..i].tkn$ stands for $s[1].tkn \circ s[2].tkn \circ \ldots \circ s[i].tkn$, where $\circ$ denotes token sequence concatenation. Again, $s[i..j].tkn$ is the empty sequence 00 if $i>j$.

All variable names are in lower case letters and names of constants and mathematical functions have at least one upper case letter in them.

An *assertion* is a logical expression involving constants and variables of the program. When an assertion is given within the text of a program segment, we enclose it in braces. Ordinary comments are enclosed in (* and *). We denote by $\{A1\}\ P\ \{A2\}$ the statement 'if the assertion $A1$ is true before execution of P, then P terminates and the resulting values will be such that $A2$ is true'. *Note that we include termination, while the above notation is often used to indicate only partial correctness* (i.e. exlcuding termination). For more details about these concepts, see References 4 and 5.

Assertions at program line i are denoted by Ai; at program lines i to j by $Ai..j$. Assertions at the beginning and end of a piece of program are called, respectively, its *entry* and *exit assertions*. An assertion may be *invariant* throughout a program segment P (i.e. it is true before and after the execution of every statement of P). Such assertions are stated only once near the top of P and are emphasized with the marker '†'.

We use the symbol '$==$', as in 'let $P == p$', to denote the value of the program variable p at that point in the program. The scope of P is the procedure/function/program segment in which it appears. Note that P is a constant. In contrast, the phrase 'stands for', as in 'let st stand for the token sequence $s[1..p].tkn$', is an abbreviation, and acts like a macro invocation; in different sentences it may stand for different $s[1..p].tkn$ either because p has changed, or because $s[...]$ has been altered.

In the following, read '$::=$' as 'is defined as.'

1. The length function $\#$ when applied to a token sequence T gives the number of tokens in it. Similarly, if z is a string, $\#z$ gives the number of characters in z. Thus T and z are empty iff $\#T$, $\#z$ are zero.
2. TRIM maps a line (including the pseudo-line) to a line by removing any trailing white space and then appending $\backslash n$. The function psTRIM is like TRIM but it also removes any leading white space.

   TRIM(z) $::= x \mid \backslash n$, where $z = x \mid y$ such that y is the longest white suffix of z.

   psTRIM(z) $::= x \mid \backslash n$, where x is empty if z is all-white; otherwise x is such that $z = w \mid x \mid y$ where w and y are the longest white prefix and suffix of z, respectively.
3. TRUNC also maps lines to lines by truncating, if necessary, to $cxMAX-1$ characters and appending $\backslash n$.

   TRUNC(s) $::= s$, if $\#s \leqslant cxMAX$; $::= x \mid \backslash n$, if $s = x \mid y$ such that $\#x = cxMAX-1$.
4. The function SET gives the set of tokens contained in a token sequence.

   SET(00) $::= \emptyset$, where $\emptyset$ is the empty set;
   SET(T $\circ$ t) $::=$ SET(T) $\cup \{t\}$, where t is a token.
5. UMCOM(T) $::=$ true iff T ends with COMBGN $\circ$ ORDINARY*;
   UMQOT(T) $::=$ true iff T ends with QUOTE $\circ$ ORDINARY*.
6. FIRSTTKN extracts the first token t and its corresponding string w from x in the context of the token sequence T.

   FIRSTTKN(T, x) $::= \langle t, w \rangle$, if x is non-empty and non-white, where LEX(T, x) $= t \circ$ LEX(T $\circ$ t, y), such that $x = w \mid y$. FIRSTTKN is undefined otherwise.

(The function LEX is defined in Reference 1.) All other mathematical functions that we use in assertions are those of Reference 1.

2.2. Global facts and assumptions

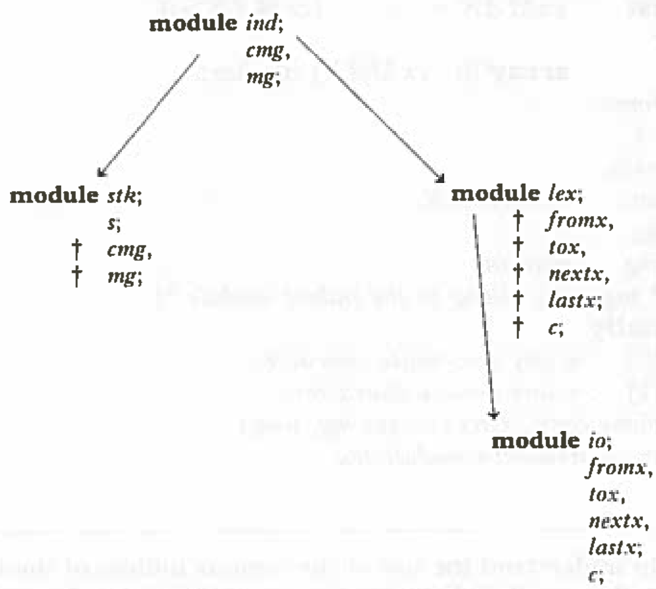
Throughout the proof, the following hold.

1. The program never refers to any undefined variables. Every variable is initialized either at module creation time, or in an assignment in the first reference to it. (In

the not-so-obvious case of the line buffer $c[0..cxMAX]$, every $c[i]$ referenced is such that $1 \leqslant i \leqslant lastx+1$, except in procedure *readline* where i may be zero also.)
2. All procedure calls in the program are such that actual parameters are distinct.
3. We assume that no integer underflows or overflows occur.
4. We also assume that the value of a variable remains unchanged if (i) it does not appear on the left-hand-side of any assignment statement, and (ii) it is not an actual **var** parameter in any procedure call. (Note that this assumption may not hold in some programming languages.) However, when a variable is to remain unchanged but does not satisfy (i) or (ii), then we shall explicitly state and prove this fact.
5. Unless the exit assertion of a procedure or program segment explicitly requires that a variable not locally declared have a certain value, it is implicitly required that all global variables remain unchanged. Without this convention, we would be forced to introduce a number of 'let ...' statements in entry assertions and equality predicates relating these to the global variables in exit assertions.

3. CORRECTNESS PROOF

The program is approximately 330 lines long and consists of four modules: *io*, *lex*, *stack* and *indent*. Figure 1 shows the interrelationships among these modules. An arrow from module A to module B indicates that A calls procedures of B. Also indicated



† denotes that this data structure belongs to another module.

Figure 1. Module interrelationships

are the data structures shared among modules. We prove first that the individual procedures contained in these modules meet their specifications as given by the entry and exit assertions. The correctness proof of the main procedure of *indent* acts as a global proof and establishes that the interfacing between modules is correct and that the specifications of the entire program are met.

As the reader will soon realize, our assertions are of crucial importance but their proofs are often routine and trivial. In fact, any of our procedures or program segments may be replaced with another (and yet the entire program meets the global specifications) so long as the new procedure or segment meets its specification as given in entry and exit assertions. For example, a naïve algorithm appears here as procedure *stdtoken* in module *lex* whereas the 'production' version of our program running under Unix replaces it by a much faster algorithm whose correctness can be proven separately. Our omission of straightforward proofs is further justified by this interchangability of procedures.

3.1. io

All input is done by the procedure *readline* and all output by procedure *printline* of this module. *readline* inputs the next line from the input file into the line buffer array c and trims the suffix white space if present. *printline* removes the prefix white space from the string $c[fromx\,..\,tox]$ and prints the remaining characters on one line with a left-margin of some number of blanks. $c[0]$ is initialized to any non-white character so as to act as a sentinel in leftward scanning (line 33) done in *readline* to trim off the suffix white space. $c[1]$ is initialized so that it is not undefined when the very first call to *readline* is made.

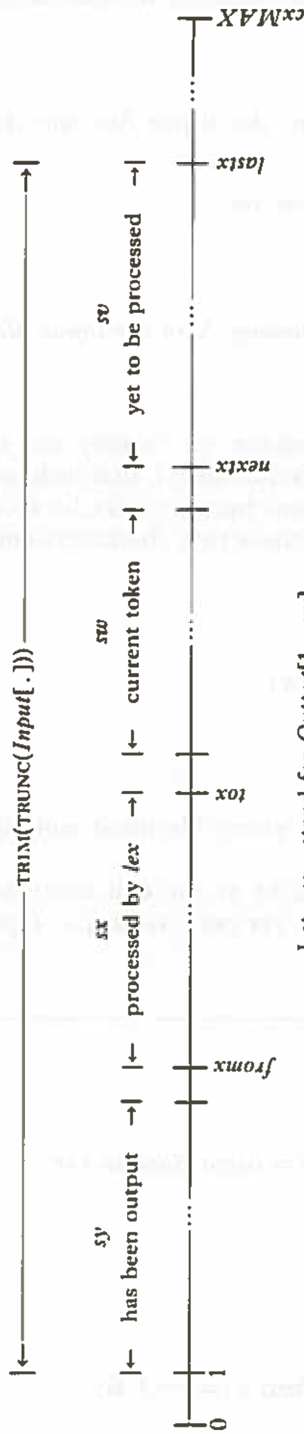
```
0   module io;
1   const    cxMAX = ...;     {cxMAX>0}
2   var
3     c      : array [0..cxMAX] of char;
4     fromx,
5     tox,
6     nextx,
7     lastx  : 0..cxMAX;
8     mg,
9     nmg    : margin;
10    (* mg, nmg belong to the indent module *)
11  initially
12    c[0]   := any non-white character;
13    c[1]   := any non-\e character;
14  (* above const, vars (except mg, nmg)
15    are shared with module lex
16  *)
```

Figure 2 will help understand the use of the various indices of the line buffer c. The names $sv, \ldots, sz$ will be used globally in the rest of the paper. Note that some strings, e.g. sx standing for $c[fromx\,..\,tox]$, can be empty (i.e. $fromx > tox$).

The following invariant $cbfINV$ holds throughout the program after the very first call to *readline* has been executed:

$cbfINV(kk)$ stands for

$$1 \leqslant fromx \leqslant tox + 1 \leqslant nextx \leqslant lastx + 1 \leqslant cxMAX$$
$$\&\quad c[1\,..\,lastx + 1] = \text{TRIM}(\text{TRUNC}(Input[kk])).$$



Let sz stand for $Output[1..nu]$,
sy stand for $c[1..fromx-1]$,
sx stand for $c[fromx..tox]$,
sw stand for $c[tox+1..nextx-1]$, and
sv stand for $c[nextx..lastx]$

Figure 2. The line buffer and its indices

Note that $c[lastx+1] = \backslash n$ always, and *cbfINV* holds throughout the indenting program for an appropriate *kk*, except in the body of readline and before the very first input line has been read.

3.1.1. *readline*

The procedure *inputchar* reads from the input file one character at a time. We assume that it satisfies

```
{   \e of input file has not been read yet
&   X is the input done so far
}
inputchar(k);
{   k = character immediately following X in the input file
&   X|k is the input done so far
}.
```

(In some computer systems, the character $\backslash n$, $\backslash e$ may not exist, but instead have standard functions *eoln* and *eof* (or the equivalent), that indicate if the end of a line or file has been reached. In such cases these characters can be accommodated in a pair of variables, one to indicate if it is one of these two characters and the other the value of the character. We then require

```
{   eoln                 {   eof
}                        }
inputchar(k)     and     inputchar(k)
{   k = \n               {   k = \e
}                        }
```

in addition to the above.)

According to our definition of files, every file must contain at least one line (the pseudo-line containing $\backslash e$).

Because *cbfINV*(*kk*) holds for some *kk* at the exit assertion of readline, $c[1] \neq \backslash e$ implies that the last input line has not yet been read; i.e. *Input*[*II*] does exist, if the input done so far is *Input*[1 .. *II*−1].

```
procedure readline;
  var i;
  {   c[0] ≠ %
  &   let Input[1 .. II−1] == input done so far
  &   c[1] ≠ \e
  }
  i := 1;
  inputchar(c[i]);
  if c[i] ≠ \e then
      while c[i] ≠ \n do
          if i < cxMAX then i := i+1 fi;
          inputchar(c[i])
      od;
      {   1 ≤ i ≤ cxMAX
```

```
31          &   c[1 .. i] = TRUNC(Input[II])
32          }
33          while c[i] = % do i := i−1 od;
34          (* must terminate since c[0] ≠ %*)
35          fi;
36      c[i+1]   := \n;
37      lastx    := i;
38      fromx    := nextx := 1;
39      tox   := 0;
40      {    1 = fromx = tox+1 = nextx ≤ lastx+1 ≤ cxMAX
41      &   c[1 .. lastx+1] = TRIM(TRUNC(Input[II]))
42      &   input done so far = Input[1 .. II]
43      }
44      end proc;
```

Note that *lastx* can be 0 at exit from this procedure. This occurs iff the line read was all-white. Reading the *e* can occur only at line 24 because the last line of a file is the pseudo-line containing exactly the one character *e*. If the operating system environment is such that 'text files' often do not satisfy our definition of a file, it is necessary to include a check for end-of-file in the **while** loop at line 26.

3.1.2. *printline*

The procedure *outputchar* appends one character at a time to the output file. We assume that it satisfies

```
{ let X == output done so far & K == k
}
outputchar(k);
{ X¦k = output done so far & k = K
}.
```

The predicate $k = K$ of the above exit assertion essentially states that k is unchanged by *outputchar*. If we drop this from our specification of *outputchar*, we cannot guarantee that the contents of the buffer c are still TRIM(TRUNC(*Input*[*II*])). Observe that even if *fromx* > *tox* or if *c*[*fromx* .. *tox*] is white space only, printline prints m blanks and a *n*. This might appear extravagant and instead one might think of outputting only a *n*; but this would make the specification of output as a function of input considerably more complicated.

```
45  procedure printline;
46      const  OUTLL  = ...;  (* maximum output line length, OUTLL > 0*)
47      var i, j, m;
48      {    let Output[1 .. VV−1] == output done so far
49      &   let M == mg, N == nmg
50†     &   1 ≤ fromx ≤ tox+1 ≤ lastx+1
51      }
52      i := fromx;
```

```
53    while (c[i] = % & i≤tox) do
54          i := i+1
55          od;
56    {     c[i .. tox] = psTRIM(c[fromx .. tox])
57    }
58    m    := 0;
59    if (0<mg+tox−i<OUTLL) then m := mg fi;
60    for j := 1 to m do outputchar(\b)          od;
61    for j := i to tox do outputchar(c[j])      od;
62    outputchar(\n);
63    fromx := tox+1;
64    mg := nmg;
65    {     fromx = tox+1 & mg = N
66    &     m = Margin(M, c[fromx .. tox])
67    &     Output[VV] = \b ** m ¦ psTRIM(c[fromx .. tox]) ¦ \n
68    &     output done so far = Output[1 .. VV]
69    }
70    end proc;
```

The function *Margin* (line 66) maps an integer and string pair to a non-negative integer. $Margin(k, s) ::= k$, if $0 \leqslant k + \#\text{psTRIM}(s) < OUTLL$, and $::= 0$ otherwise.

3.2. lex

Procedures contained in this module are *nexttoken*, *newline*, *firsttokeninline* and *initlex*, of which *nexttoken* maps character strings into token sequences.

```
module lex;
 const cMAX = ...;
       DELIMITERS = {\b, \t, \n, \e, ";", "{", "}", "'",
                     "=", ":", "(", ")", "*"};
 var
       c           : array [0 .. cxMAX] of char;
       fromx,
       tox,
       nextx,
       lastx       : 0 .. cxMAX;   (* these vars belong to module io *)
       incomment,
       instring    : boolean;
       tokenno     : integer;
 initially
       incomment := instring := false;
       tokenno := 0;
```

The invariant *lexINV(uu)* holds before and after every call to the routines of this module for an appropriate *uu* (cf. Figure 2):

lexINV(uu) stands for

$$\begin{aligned}
&\mathrm{TKNSEQ}(sx \mid sw \mid \backslash b \mid sv) = \mathrm{TKNSEQ}(sx \mid sw \mid sv)\\
\&\quad & tokenno = \#\mathrm{TKNSEQ}(sx \mid sw)\\
\&\quad & (tokenno > 0 \rightarrow \#\mathrm{TKNSEQ}(sw) = 1)\\
\&\quad & incomment = \mathrm{UMCOM}(\mathrm{TKNSEQ}(Output[\,..\,uu] \mid sx \mid sw))\\
\&\quad & instring = \mathrm{UMQOT}(\mathrm{TKNSEQ}(Output[1\,..\,uu] \mid sx \mid sw))
\end{aligned}$$

(The functions TKN, TKNSEQ and LEX are defined in Reference 1. Note the insertion of a blank into the argument of TKNSEQ in the first predicate of *lexINV*. Without this subtle device, it would be more complex to describe the properties of *nextx*. If $nextx > fromx$, there are two possibilities: (i) the character $c[nextx]$ is a delimiter and therefore the previous token ended at $nextx-1$; (ii) the character $c[nextx]$ is not a delimiter and therefore the previous token is either a single or double character token starting with a non-white delimiter. On the other hand, *nextx* will equal *fromx* when a line has just been read and the first token from it is yet to be extracted.

Note that the value of *fromx* and *lastx* change only indirectly via calls to procedures of module *io*.

3.2.1. *nexttoken*

Let us first consider three principal segments—*gettoken, dlmtoken* and *stdtoken*— of procedure *nexttoken*. The program segments *dlmtoken* and *stdtoken* are given only for the sake of completeness. Far more efficient algorithms can be constructed for these; however, their correctness can be established separately from the entire program. The naïve algorithms implement the definition of TKN (see Reference 1) straightforwardly and we omit their proofs.

gettoken. This program segment obtains the longest prefix $c[nextx\,..\,j-1]$ of $c[nextx\,..\,lastz]$ such that $c[nextx\,..\,i-1]$ is all-white and $\mathrm{TKN}(c[i\,..\,j-1])$ is defined as *t*.

```
16  program segment  gettoken
17    of nexttoken;
18    {  c[lastx] ≠ % & c[lastx+1] = %
19    &  1 ≤ nextx ≤ lastx
20    }
21    i := nextx
22    while c[i] = % do i := i+1 od;
23    {  nextx ≤ i ≤ lastx
24    &  c[nextx..i−1] = % ** (i−nextx)
25    &  c[i] ≠ %
26    }
27    j := i
28    while c[j] not in DELIMITERS do
29      j := j+1
30    od;
31    {  A23..26
32    &  i ≤ j ≤ lastx+1
```

```
33     &  c[i..j−1] is DELIMITERS-free
34     &  c[j] in DELIMITERS
35     }
36     if  i = j then
37         dlmtoken;
38         j := j + d
38     else
40         stdtoken;
41         fi;
42     {  nextx < j ≤ lastx + 1
43     &  t = TKNSEQ(c[nextx..j−1])
44     &  TKNSEQ(c[nextx..j−1] ¦ \b ¦ c[j..lastx])
45     =  TKNSEQ(c[nextx..lastx])
46     }
47     end program segment;
```

The proof of this program segment readily follows once we see that assertions $A23..26$ and $A31..35$ hold at the stated points. That $i \leqslant lastx$ in $A23$ follows from $c[lastx] \neq \%$ of entry assertion. Because $c[last0] \neq \%$, and $c[lastx+1] = \%$ j may indeed equal $lastx+1$ after the **while**-loop at line 28 terminates, as implied by $A32$.

If $i = j$, then $c[i]$ is a delimiter, and *dlmtoken* (see line 37) would return with $t = \text{TKN}(c[i..i+d-1])$ and $d = 1$ or 2. After line 38, we get $t = \text{TKN}(c[i..j-1])$. If $i < j$, then $nextx \leqslant i < j$ and *stdtoken* (see line 40) would return with $t = \text{TKN}(c[i..j-1])$. Since $c[nextx..i-1]$ is all-white (from $A24$), $\text{TKNSEQ}(c[nextx..j-1]) = \text{TKN}(c[i..j-1]) = t$. Thus $A42..43$ hold.

Since $c[nextx..i-1]$ is all-white, in order to establish $A44..45$, we need only show that $\text{TKNSEQ}(c[i..j-1] \mid \backslash b \mid c[j..lastx]) = \text{TKNSEQ}(c[i..lastx])$, i.e. that insertion of a blank between $c[j-1]$ and $c[j]$ into the buffer will not alter the token sequences produced. If $c[i..j-1]$ were delimiter-free, $c[j]$ must be a delimiter and inserting a blank just before it does not matter. On the other hand, if $c[i]$ were a delimiter, then *dlmtoken* has considered the largest possible token starting with $c[i]$ and hence once again blank insertion to the left of $c[j]$ does not alter the token sequence produced. Thus $A42..46$ hold.

```
48  program segment dlmtoken
49     of gettoken
50     {  0 ≤ j ≤ lastx
51     &  c[j] in DELIMITERS − {\b, \t, \e}
52     }
53     ⟨t, d⟩ := (
54       cases
55         c[j] = ";"                 : ⟨SEMICOLON , 1⟩;
56         c[j] = "{"                 : ⟨COMBGN    , 1⟩;
57         c[j] = "}"                 : ⟨COMEND    , 1⟩;
58         c[j] = "'"                 : ⟨QUOTE     , 1⟩;
59         c[j] = ")"                 : ⟨RPAREN    , 1⟩;
```

```
          c[j] = "="                        : 〈ORDINARY  , 1〉;
          c[j] = \e                         : 〈ENDFILE   , 1〉;
          c[j] = ":" & c[j+1] = "="         : 〈ORDINARY  , 2〉;
          c[j] = ":" & c[j+1] ≠ "="         : 〈COLON     , 1〉;
          c[j] = "(" & c[j+1] = "*"         : 〈COMBGN    , 2〉;
          c[j] = "(" & c[j+1] ≠ "*"         : 〈LPAREN    , 1〉;
          c[j] = "*" & c[j+1] = ")"         : 〈COMEND    , 2〉;
          c[j] = "*" & c[j+1] ≠ ")"         : 〈ORDINARY  , 1〉;
          end cases                 );
    {(TKN(c[j..j+d]) is undefined
    &  (d = 1 or d = 2)
    &  t = TKN(c[j..j+d-1])
    }
    end program segment;
```

```
program segment stdtoken
    of gettoken;
    {   i<j
    &   c[i..j-1] is DELIMITERS-free
    }
    t :=      (
      case c[i..j-1] of
          "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;
```

```
105         other        :   ORDINARY;
106         end case
107         )
108     {   t = TKN(c[i..j-1]
109     }
110     end program segment;
```

Nexttoken. *Nexttoken* first checks to see if all characters of $c[1..lastx]$ have been processed. If so more input is read until a non-white line is obtained. If the first character of c is $\backslash e$ this indicates an end of file condition.

Nexttoken then obtains the longest prefix $c[nextx..j-1]$ of $c[nextx..lastx]$ such that $c[nextx..i-1]$ is all-white, and TKN$(c[i..j-1])$ is defined. It then updates *nextx* to j. The returned token t equals TKN$(c[i..j-1])$ if it is not within a comment or a string; otherwise t will be ORDINARY unless TKN$(c[i..j-1])$ = ENDFILE. In the assertions below *ioDONE*(*ii*, *uu*) stands for

```
    input done so far = Input[1..II+ii]
  & Input[II+1..II+ii-1] are all white
  & Input[II+ii] is not all white
  & Output done so far = Output[1..UU+uu]
  & (uu = 0
    or  uu>0 & mg = nmg
    &   Output[UU+1] = \b**M¦C[F..L]
    &   Output[UU+2..UU+uu] = (\b**mg)**(uu-1)
    ).
```

```
111 function nexttoken returns t;
112     var t, i, j, d;
113
114     {   let F == fromx, T == tox, N == nextx, L == lastx, M == mg
115     &   let C[0..L+1] == c[0..L+1]
116     &   let Input[1..II] == input done so far
117     &   let Output[1..UU] == output done so far
118     &   II>1 → c[1]≠\e
119     &   cbfINV(II) & lexINV(UU)
120     }
121     tox := nextx-1;
122     while nextx>lastx do
123        printline;
124        readline;
125        tokenno := 0;
126        od;
127     {   let NI == the number of times lines 123..125 are executed
128     &   (NI = 0 & F = fromx ≤ tox+1 = N = nextx < L+1 = lastx+1
129     or  NI>0 & 1 = fromx = tox+1 = nextx < lastx+1
130     )&  ioDONE(NI, NI) & cbfINV(II+NI) & lexINV(UU+NI)
131     }
```

```
132  gettoken;
133  nextx := j;
134  tokenno := tokenno+1;
135  if t≠ENDFILE then
136     cases
137          incomment:
138               if t = COMEND then incomment := false
139               else t := ORDINARY fi;
140          instring:
141               if t = QUOTE then instring := false
142               else t := ORDINARY fi;
143          not (instring or incomment):
144               cases
145                    t = COMBGN : incomment := true;
146                    t = QUOTE : instring := true;
147                    other      : (* do nothing *);
148                    end cases
149          end cases;
150     fi;
151  {    let NI == number of times lines 123..125 are executed
152  &    (NI = 0 & F = fromx ≤ tox+1 = N < nextx ≤ L+1 = lastx+1
153  or   NI>0 & 1 = fromx = tox+1 < nextx ≤ lastx+1
154  )&   ioDONE(NI, NI) & cbfINV(II+NI) & lexINV(UU+NI)
155  &    ⟨t, sw⟩ =
156       FIRST(TKNSEQ(sz|sx), sw|sv)
157  }
158  end proc;
```

Consider $A127..131$. We have from $A119$ that $cbfINV(II)$ holds. Thus if $NI=0$ (i.e. lines 123..125 were not executed at all), $Input[II]$ cannot be all-white. Because if it is, $N = nextx > L = lastx$ and NI must be greater than zero. Thus $ioDONE(0, 0)$ and hence $A127..131$ hold trivially. On the other hand, if $NI>0$, $A129$ must hold for all the NI lines thus read. The first line output by this loop will be $C[F..L]$ and the subsequent $NI-1$ lines must be all-white. The loop must terminate because the last line of every file is the pseudo-line which is not all-white and readline will let $nextx \leqslant lastx$, and $Input[II+NI]$ must not be all-white. Thus $ioDONE(NI, NI)$ holds. That $cbfINV(II+NI)$ holds is guaranteed by the exit assertion of readline, and that $lexINV(UU+NI)$ is true follows readily because we have $lexINV(UU)$ at $A119$, $Output[UU+1..UU+NI]$ is all-white and $1 = fromx = tox+1 = nextx < lastx+1$. Hence $A127..131$ hold.

We now show that $\{A127..131\}$ lines 132..150 $\{A151..157\}$. In this part of the proof, whether $NI>0$ or not does not matter. While $A128..129$ imply that $tox+1 = nextx$, $A152..153$ imply that now $tox+1 < nextx$. This essentially guarantees that 'progress' will be made in every invocation of *nexttoken*. Without this, *nexttoken* can trivially satisfy its exit assertion by doing nothing and returning the previous token. That $tox+1 < nextx$ after line 133 follows from the exit assertion ($A42$) of *gettoken*.

That $lexINV(UU+NI)$ holds is immediate from $A44..45$ and lines 134..150, and $ioDONE(NI, NI)$ continues to hold as our files are sequential. Since lines 134..150 contain no calls to readline, $cbfINV(II+NI)$ still holds. From $lexINV(UU+NI)$ and noting that (i) just before execution of line 133 we have $A43$, (ii) $nextx = tox+1$(from$A128..129$) and (iii) just after execution of line 133 $nextx = j$, we get $A155..157$. This completes the proof of *nexttoken*.

3.2.2. *newline, firsttokeninline and initlex*

The following three procedures are self-explanatory.

```
159  procedure newline;
160    {   A114..120, entry assertion of nexttoken
161    &   let TN == tokenno
162    }
163    if  tokenno > 1 then
164        printline;
165        tokenno := 1
166        fi;
167    {   (TN > 1 & NU = 1 or TN ≤ 1 & NU = 0)
168    &   ioDONE(0, NU) & cbfINV(II) & lexINV(UU + NU)
169    }
170    end proc;
171  function firsttokeninline returns b;
172    var b;
173    {  true
174    }
175    b := (tokenno = 1);
176    {  b ↔ tokenno = 1
177    }
178    end proc;
179  procedure initlex;
180    {   c[0] ≠ % & c[1] ≠ \e
181    &   no input has been done so far
182    }
183    readline;
184    {  exit assertion of readline & cbfINV(1)
185    }
186    end proc;
```

3.3. stk

This module implements a stack which is used by the main module. Note that the ORDINARY token is never stacked, and *stk* safely uses ⟨ORDINARY, 0⟩ as a sentinel at the bottom of the stack. We thus have the following property holding before and after every call to procedure of this module:

$$s[0] = \langle \text{ORDINARY}, 0\rangle$$
$$\& \quad 0 \leqslant p \leqslant pMAX$$
$$\& \quad (\text{SET}(s[1\,..\,p].tkn) \cap \{\text{ORDINARY}\}) = \varnothing.$$

The main program uses the stack in such a way that a certain invariant *stkINV* to be given later is a loop invariant of main.

We believe the proofs of the five procedures below are straightforward and hence omit them.

```
module stk;                        (*implements a stack *)
  const pMAX = ...                 {pMAX>0}
  var
        p :- 1..pMAX+1;
        s: array[0..pMAX] of ⟨tkn: token, mgn: margin⟩;
  initially
        s[0] := ⟨ORDINARY, 0⟩;
        p := 0;
procedure   stack(t: token, m: margin);
  {  let P == p, S[0..P] == s[0..P]
  }
  if (t≠ORDINARY & p<pMAX) then
        p := p+1;
        s[p] := ⟨t, m⟩;
        fi
  {    s[0..P] = S[0..P]
  &    (p = P+1 & s[p] = ⟨t, m⟩
  or p = P & (P = pMAX or t = ORDINARY)
  )}
  end proc;
procedure   unstack;
  {  let P == p, S[0..P] == s[0..P]
  }
  if p>0 then p := p-1 fi
  {    P = 0 → s[0..p] = S[0..P]
  &    P > 0 → s[0..p] = S[0..P-1]
  }
  end proc;
procedure stktop (var t: token, var m: margin);
  {  let P == p, S[0..P] == s[0..P]
  }
  ⟨t, m⟩ := s[p];
  {  ⟨t, m⟩ = S[P]
  }
  end proc;
function stackhas(sot : set of token) returns b;
  var q, b;
```

```
    {  let P == p, S[0..P] == s[0..p]
    }
    q := p;
    while s[q].tkn not in (sot ∪ {ORDINARY}) do
        q := q−1
        od;
    b := (q>0)
    {  b ↔ (sot ∩ SET(s[1..p].tkn) ≠ Ø)
    }
    end proc;
procedure unstackuntil (
            sot : set of token,
            var m : margin );
    var t;
    {  let P == p, S[0..P] == s[0..P]
    }
    repeat
        ⟨t, m⟩ := s[p];
        p := p−1
    until (t in sot ∪ {ORDINARY});
    if p<0 then p := 0 fi;
    {  SET(S[p+2..P].tkn) ∩ sot = Ø
    &  (p≥0 & S[p+1].tkn in sot & m = S[p+1].mgn
    or p = 0  &  m = 0  &  (P = 0  or  S[1] not in sot)
    )}
    end proc;
procedure unstackwhile(
            sot: set of token;
            var m: margin );
    var t;
    {  let P == p, S[0..P] == s[0..P], M == m
    &  ORDINARY not in sot
    }
    while s[p].tkn in sot do
       m := s[p].mgn;
       p := p−1
    od
    {  p = P → m = M
    &  p<P →
    (  SET(S[p+1..P].tkn) ⊆ sot
    &  S[p].tkn not in sot
    &  m = S[p+1].mgn
    )}
    end proc;
```

Note that *unstackuntil* unstacks at least one item whereas *unstackwhile* unstacks as long as the top item is in the given set.

3.4. Program indent

This module contains the so-called 'main' program *indent* which controls all other procedures either directly or indirectly. We first consider the following important segment of the program.

3.4.1. *calcredcnmg*

Program segment *calcredcnmg* computes the indentations resulting from the current token *t* and updates the variables *cmg* and *nmg*. These two variables respectively take the CMG and NMG values of the token sequence of the input file so far seen. It also maintains on the stack the reduced token sequence. (See Reference 1 for the definitions of NMG, CMG and REDuced token sequences.) In the assertions

$stkINV(T)$ stands for

$$s[1\,..\,p].tkn = \text{RED}(T)$$
$$\&\quad s[i].mgn = \text{NMG}(s[1\,..\,i-1].tkn)\ \textit{for all}\ i,\ 1 \leqslant i \leqslant p.$$

where T is a token sequence.

```
program segment calcredcnmg
    of main;
    var t0, t1, m0, m1, n, sot;
    {    stkINV(T)
    &    cmg = nmg = NMG(T)
    }
    case t of
        PROCEDURE,
        FUNCTION,
        PROGRAM,
        LABEL,
        CONST,
        TYPE,
        VAR:
            stktop(t0, m0);
            if t0 ≠ LPAREN then
                if t0 ≠ DECL then stack(DECL, nmg)
                else cmg := nmg := m0 fi;
                nmg := nmg + UOI;
                if t in {PROCEDURE, FUNCTION, PROGRAM} then
                    stack(PF, nmg)
                    fi;
                fi;
        OF :
            stktop(t0, m0);
            unstack;
            stktop(t1, m1);
            if (t0 = COLON  &  t1 = CASE) then cmg:= nmg := m1 + UOI
            else stack(t0, m0) fi;
        BEGIN   :
```

```
                stktop(t0, m0);
                if t0 = DECL then
                        unstack;
                        cmg := nmg := m0
                        fi;
                stktop(t0, m0);
                if t0 = PF then
                        unstack;
                        cmg := nmg := m0
                        fi;
                stack(t, nmg);
        END:
                if stackhas({RECORD}) then
                        sot := {RECORD}
                else sot := {BEGIN, CASE} fi;
                unstackuntil(sot, nmg);
                cmg := nmg;
        RPAREN   :
                unstackuntil({LPAREN}, nmg);
                cmg := nmg;
        LPAREN,
        REPEAT,
        CASE,
        DO,
        THEN,
        RECORD,
        COLON    :
                stack(t, nmg);
                nmg := nmg + UOI;
        UNTIL    :
                unstackuntil({REPEAT}, nmg);
                cmg := nmg;
        ELSE     :
                unstackuntil({THEN}, cmg);
                nmg := cmg + UOI;
                stack(t, cmg);
        SEMICOLON   :
                unstackwhile({THEN, ELSE, DO, COLON}, nmg);
                cmg := nmg;
        other    :
                (* do nothing *);
        end case;
  {     stkINV(T ∘ t)
  &     nmg = NMG(T ∘ t)
  &     cmg = CMG(T ∘ t)
  }
  end program segment;
```

The proof here is mechanical since it 'executes' the definitions of RED, NMG, and CMG literally. Two sample proofs are given below; others are similar.

Case $t =$ PROCEDURE. The *stkINV* in the entry assertion implies that $t0$, after execution of line 14, is the last token of RED(T).

Suppose now that $t0 =$ LPAREN. Then RED($T \circ$ PROCEDURE) = RED(T) by definition, and the *stack*, *cmg* and *nmg* remain unchanged, thus establishing the exit assertion.

Suppose $t0 \neq$ LPAREN. Suppose further that $t0 =$ DECL (and hence t is a token from a nested procedure); i.e. $s[1..p].tkn =$ RED(T) $= R \circ$ DECL for some R. Then by the assignment $cmg := nmg := m0$ and $stkINV(T)$ in the entry assertion, we have that

$$cmg = nmg = m0 = \text{NMG}(s[1..p-1].tkn) = \text{NMG}(R).$$

Execution of line 18 then sets $nmg = \text{NMG}(R) + UOI$ and after line 21, the stack contains $R \circ \text{DECL} \circ \text{PF} = \text{RED}(T \circ \text{PROCEDURE})$ as defined thus establishing $stkINV(T \circ t)$. Also *cmg* now equals

$$nmg - UOI = \text{NMG}(T \circ t) - UOI = \text{CMG}(T \circ t),$$

as defined. Thus $A72..75$ hold.

A similar proof is given if $t0 \neq$ DECL.

Case $t =$ UNTIL. The $stkINV(T)$ of the entry assertion implies that after execution of *unstackuntil* (line 60) $s[1..p].tkn$ (call this Q) will either be 00, if RED(T) did not contain any REPEAT tokens, or Q will be such that RED(T) $= Q \circ$ REPEAT $\circ R$ and R contains no REPEAT tokens. Clearly $stkINV(T \circ t)$ is established. Since $nmg = \text{NMG}(Q)$ by *unstackuntil* and line 61 sets $cmg = nmg = \text{NMG}(Q) = \text{NMG}(\text{RED}(T \circ t)) = \text{CMG}(T \circ t)$ as required by the definition of CMG, we have $A72..75$.

3.4.2. *main*

Control is passed to *main* after the initializations in the modules are performed. The main program employs *lex* to give it the token sequence corresponding to the input text. Observe that in the body of the repeat-loop there are no calls to the module *io*. All input/output of text is caused indirectly by calls to the procedures of module *lex*.

To understand the assertion $indINV(ni, nu)$ below more readily see Figure 2.

The assertion $indINV(ni, nu)$ of the program below stands for:

$$\begin{aligned} & sz = \text{INDENT}(Input[1..ni-1] \mid sy) \\ \& \quad & \langle t, sw \rangle = \text{FIRSTTKN}(\text{TKNSEQ}(sz \mid sx), sw \mid sv) \\ \& \quad & cbfINV(ni) \\ \& \quad & lexINV(nu). \end{aligned}$$

Also let $segINV(st)$ stand for

$$st = \text{FIRSTSEG}(\text{TKNSEQ}(sz), st),$$

and let $mgnINV(su)$ stand for

$$nmg = cmg = \text{NMG}(\text{TKNSEQ}(su)) \quad \& \quad mg = \text{MG}(\text{SEGSEQ}(su)).$$

Intuitively, the first predicate of *indINV* asserts that the output so far is the indented version of the input done so far, *segINV* asserts that *st* does not contain more than one segment and *mgnINV* asserts that the margin variables are correct. Note that in the following, the value of *cmg* is important only when the previous token starts on a new line when *mg* takes *cmg*'s value (line 114) as required by the definition of MG.

```
77  program indent;
78      const
79          UOI = ...;          (* unit of indentation *)
80          LO  = {...};        {ENDFILE not in LO}
81          LC  = {...};        {ENDFILE not in LC}
82      var
83          m0,
84          mg,
85          cmg,
86          nmg      : margin;
87          t, t0    : token;
88          carry    : boolean;
89      initially
90          mg := cmg := nmg := 0;
91          carry := false;
92      {   next line to be read is Input[1]
93      &   next line to be output becomes Output[1]
94      &   indINV(0, 0)  &  segINV(sx|sw)  &  mgnINV(sz|sx|sw)
95      }
96      initlex;
97      repeat
98          {   let Input[1 .. II] = = input done so far
99          &   let Output[1 .. UU] = = output done so far
100         &   let SZ = = sz, SX = = sx, SW = = sw
101         &   indINV(II, UU)  &  segINV(sx|sw)  &  mgnINV(sz|sx|sw)
102         }
103         t := nexttoken;
104         compute-red-cnmg:
105         {   indINV(II+NI, UU+NI) & segINV(sx) & mgnINV(sz|sx)
106         }
107         calcredcnmg;
108         {   indINV(II+NI, UU+NI)  &  segINV(sx)
109         &   nmg = NMG(TKNSEQ(sz|sx|sw))
110         &   cmg = CMG(TKNSEQ(sz|sx|sw))
111         &   mg  = MG(SS(sz|sx))
112         }
113         if t in LO then newline fi;
114         if firsttokeninline then mg := cmg fi;
115         cmg := nmg;
116         { indINV(II+NI, UU+nu)  &  segINV(sx|sw)
                & mgnINV (sz|sx|sw)
117         &   nu = NI+ord(t in LO)
118         }
```

```
119          if t in LC then
120              t := nexttoken;
121              while t = COMBGN do
122                  repeat
123                      t := nexttoken
124                  until t in {COMEND, ENDFILE};
125                  if t≠ENDFILE then t := nexttoken fi;
126                  od;
127              newline;
128              goto compute-red-cnmg;
129              fi;
130      until t = ENDFILE;
131      outputchar(\e);
132      {   last line is read
133      &   let Input[1..JJ] == input done so far
134      &   let Output[1..KK] == output done so far
135      &   indINV(JJ,KK)
136      }
137      end program;
```

The proof below depends on the function SEGSEQ that produces segment sequences form input lines and on the function MG that determines the margin of output lines corresponding to these segments. We shall make use of the fact

$\{ SZ = \text{INDENT}(Input[1\,..\,ni-1] \mid SY)$
$\&\ mg = \text{MG}(\text{SEGSEQ}(SZ \mid SX))\ \&\ segINV(SX)$
$\}$
printline
$\{\ \ sz = output\ done\ so\ far = \text{INDENT}(Input[1\,..\,ni-1] \mid SY \mid SX)\}.$

Further note that in the following, $NI \geqslant 0$, $NU \geqslant 0$ and $MI \geqslant 0$.

We perform the proof of $\{A92\,..\,95\}$ lines 96 .. 131 $\{A132\,..\,136\}$ in five parts, the last part being a termination proof.

(1) The first part is $\{A98\,..\,102\}$ line 103 $\{A105\,..\,106\}$. Firstly note that $A98\,..\,102$ implies the entry assertion of *nexttoken*.

Case $NI = 0$. No output is done and so $sz = SZ$ and $sz = \text{INDENT}(Input[1\,..\,II] \mid sy)$ continues to hold from $A101$. Now $sx = SX \mid SW$ by the exit assertion of *nexttoken*, and $SX \mid SW = \text{FS}(\text{TKNSEQ}(sz),\ SW \mid SW)$ from *segINV* in $A101$. Thus $segINV(sx)$ holds. Again since $sz \mid sx = SZ \mid SX \mid SW$, $mgnINV(sz \mid sx)$ holds from *mgnINV* in $A101$ since *nmg*, *cmg*, *mg* and the stack are unchanged.

Case $NI > 0$. The predicate *ioDONE* in the exit assertion of *nexttoken* and $mgnINV(sz \mid sx \mid sw)$ in $A101$ imply that $sz = \text{INDENT}(Input[1\,..\,II+NI] \mid sy)$ holds. Now both *sy* and *sx* are actually empty as implied by $A153$ of *nexttoken*, and so $segINV(sx)$ holds trivially. Since $\text{TKNSEQ}(sz \mid sx) = \text{TKNSEQ}(SZ \mid SX \mid SW)$ again by *ioDONE* in the exit assertion of *nexttoken* and because the stack is unchanged, $mgnINV(sz \mid sx)$ holds.

In both cases, $cbfINV(II+NI)$ & $lexINV(UU+NI)$ & $\langle t, sw \rangle =$ FIRSTTKN (TKNSEQ($sz \mid sx$), $sw \mid sv$) follow from the exit assertion of *nexttoken*. Thus $indINV(II+NI, UU+NI)$ holds, and hence again $A105..106$ holds.

(2) By letting TKNSEQ($sz \mid sx$) be the T in the entry assertion of *calcredcnmg*, we have the second part $\{A105..106\}$ *calcredcnmg* $\{A108..112\}$.

(3) The third part is $\{A108..112\}$ lines 113 .. 115 $\{A116..118\}$. Note that at $A108$, sx is either empty or it is $SX \mid SW$.

Case t in LO. Suppose $sx = SX \mid SW$. By the definition of SEGSEQ, $segINV(sx)$ holds whereas $segINV(sx \mid sw)$ does not. From $mg =$ MG(SEGSEQ($sz \mid sx$)) of $A111$ and $ioDONE$ in the exit assertion of *newline*, we get $sz =$ INDENT($Input[1..II+NI] \mid sy$). Again by this exit assertion, $cbfINV(II+NI)$ and $lexINV(UU+NI+1)$ holds. Since $\langle t, sw \rangle$ is unchanged, $segINV(sx \mid sw)$ holds, again because sx is empty. At this stage

$$nmg = \text{NMG}(\text{TKNSEQ}(sz \mid sx \mid sw)),\ cmg = \text{CMG}(\text{TKNSEQ}(sz \mid sx \mid sw))$$

and $stkINV$(TKNSEQ($sz \mid sx \mid sw$)) continue to hold since nmg, cmg and the stack are unchanged; however, mg is yet to be set correctly.

The **then** body of line 114 must be executed because as a result of *newline*, *firsttokenline* must return true. It is here that we now get $mg = cmg =$ CMG(TKNSEQ($sz \mid sx \mid sw$)) and this is MG(SEGSEQ($sz \mid sx \mid sw$)) by definition. We thus get $A116..117$ after execution of line 115.

A similar argument suffices if we had assumed above that sx were empty and not $SX \mid SW$. Note that nu in $A116$ is $NI+1$ in the former case and NI in this one. Thus $nu = NI + \mathbf{ord}(t\ \mathbf{in}\ LO)$ using the ordinal function of Pascal.

Case t not in LO. Line 110 has no effect and since $indINV$ continues to hold, mg is $cmg =$ CMG(TKNSEQ($sz \mid sx \mid sw$)) from the **then** body of line 114, which is executed if and only if sx were empty; on the other hand, if *firsttokeninline* were false, then $segINV(sx \mid sw)$ and $\langle t, sw \rangle =$ FIRSTTKN(TKNSEQ($sz \mid sx$), $sw \mid sv$) ensures that mg remains at MG(SEGSEQ($sz \mid sx$)) which by definition is MG(SEGSEQ($sz \mid sx \mid sw$)). Once again $A116..A117$ holds after execution of line 115.

(4) The proof of the fourth part $\{A116..117\}$ lines 119 .. 128 $\{A108..112\}$ proceeds along similar lines as that of line 113.

Suppose t is not in LC at $A116..117$. Then either t is ENDFILE, in which case the exit assertion $A132..136$ immediately follows from $A116$, or we loop, in which case $A98..102$ follow from $A116$ again (but with appropriate new values for II, UU, SZ, SX and SW).

Suppose t is in LC at $A116..117$. Then clearly lines 120 .. 126 call *nexttoken* repeatedly so as to get the token immediately following t of $A116$ that is not within a comment, or until *nexttoken* returns an abrupt occurrence of ENDFILE. Since the exit assertion of any previous call implies the entry assertion of the next call, and since $lexINV$ holds at $A116$ and throughout this repeated calling of *nexttoken*, all the tokens thus returned belong are in {COMBGN, COMEND, ORDINARY} except the very last one returned.

It is easy to see that $indINV(II+NI+MI, UU+nu+MI)$ holds after execution of line 126 because if $MI>0$, then it must be due to the cumulative effects of the NI in $ioDONE$ in the exit assertion of *nexttoken* which at some times must have been nonzero. Let $SX0$ and $SW0$ be *sx* and *sw* respectively at $A116$ so that $segINV(SX0 \mid SW0)$ holds there. Now any string *st* that is a (string of) comments is such that SEGSEQ(sx0 | sw0 | *st*) is one segment *unless st* contains $\backslash n$. In the latter case, let $st = su1 \mid \backslash n \mid su2$ where $su2$ does not contain any $\backslash n$. It then follows from *indINV* that $sx = su2$. Since $su2$ contains only tokens as mentioned above, $segINV(sx)$ holds. That $mgnINV(sz \mid sx)$ holds is straightforward since the tokens COMBGN, COMEND and ORDINARY have no effect on *nmg*, *cmg*, *mg* and the stack.

It now remains to show that $A105 \mathinner{..} 106$ holds after a call to *newline* in line 127. The proof here is almost identical to that in line 113 where SEGSEQ(*sx*) holds but SEGSEQ(*sx* | *sw*) does not.

After execution of lines 127 .. 128, $A105 \mathinner{..} 106$ hold with the values of $II+NI$ and $UU+NI$, respectively, replaced by the new values $II+NI+MI$ and $UU+nu+MI$. This concludes the proof of the fourth part.

Initially $A98 \mathinner{..} 102$ hold, i.e. at the first time execution enters the **repeat** loop, since II is 1, UU is 0 and *sz*, *sx*, and *sw* are empty. This concludes the proof that $\{A92 \mathinner{..} A95\}$ lines 96 .. 131 $\{A132 \mathinner{..} 136\}$, with the proviso that the program terminates.

(5) It now remains for us to prove that the program terminates. We do this by showing that $sz \mid sw$ increases after every execution of *nexttoken*. Hence by the finiteness of the input file, we are done.

We see that either $A152$ or $A153$ of *nexttoken* must hold. If $A153$ holds, then *sz* is increased and *sx* is empty. If $A152$ holds, since $nextx > N$ (where N is the value that *nextx* had at the entry assertion of *nexttoken*) and $tox+1 = N$, only *sx* has increased. We show now that at least one call to *nexttoken* will be made between any two executions of line 115: after execution of line 115, if t is in LC, line 120 implies this; otherwise, the **goto** at line 128 is not executed and so line 103 ensures this.

Clearly the loops in lines 121 .. 126 always terminate implying that only a finite time will be spent after execution of line 115 before execution reaches it again, or reaches line 131. This completes the termination proof.

4. DISCUSSION

Although it was more than a decade ago that foundations of correctness proofs were laid by Floyd, Naur and Hoare (see e.g. Reference 5), one cannot say with conviction that a correctness proof technology has now emerged. The ratio of programmers who practice giving correctness proofs to those who do not is negligibly small. The reasons for this phenomenon will be long debated. We have the suspicion that one cause for this has been the high level of rigour and formalism in the example proofs of pioneers like Dijkstra, Hoare and others (see e.g. References 6 and 7), and a shortage of examples of proofs at the intermediate levels of rigour.

It is widely recognized that competent programmers adopt certain paradigms familiar to them when designing programs. They are forever searching for newer or different paradigms to add to their collection. Such practices should be encouraged (see e.g. Reference 8) as principles of systematic design. Although we can say that these do exist—however few they may be—in the context of designing programs, paradigms and styles for assertions and proofs of classes of small programs are yet to emerge. What little exists is buried deep beneath heavy notation and formalism or rigour. And the correctness of correctness proofs has become exceedingly important.

Also, the main goal in the published correctness proofs has generally been to establish the correctness of the program being considered rather than establishing the essence of the proof, exploring what level of rigour is appropriate and the selection of the best way to structure and present a proof.

We are aware of many conscientious programmers who do use reasoning, in addition to testing, to convince themselves and other sympathetic people that their programs work. These programmers and the published literature shied away from documenting such efforts extensively mainly for two reasons: (1) their informal notation and arguments cannot be taken as proofs 'beyond all doubt' that the program in question meets its specifications, and (2) their methods have nothing original—they travel the road paved by Floyd and Dijkstra. In spite of these reasons, we believe that the programming community will benefit if such efforts are documented widely. Such efforst will (1) demonstrate to a wide audience the usefulness of 'reasoning' as against testing, and (2) reduce the effort required to produce these correctness arguments as a result of the experience gained both by the authors and readers.

The present paper is intended to be one such effort, and we urge the reader to lower his expectations of the possible benefits from proofs (of the kind advocated here) to a modest and realistic level. We should not expect proofs of this kind to establish 'beyond all doubt' that the program meets its specification. We should be content if all such a proof does is to raise the confidence level with which we say that it is plausible that the program is correct.

We do not claim that we have been entirely successful in achieving all our objectives. What is most disconcerting is that an estimated total of 250 man-hours were spent in discovering the assertions, choosing the right notation and the style of presentation. In contrast, we estimate that a total of only 60 hours were spent in the design, implementation and testing of all three versions of the program developed during the proof process. We believe that this figure would have been considerably lower if we had other example proofs (at this level of rigour) of medium-sized programs to emulate.

Below we discuss some of the issues that must be understood before assessing the approach taken in the proof of indent.

4.1. Pitfalls

As Gerhart and Yellowitz[9] point out, modern methodologies are not infallible. When the level of rigour is decreased, this danger further increases.

Hidden assumptions

The most serious of all dangers in informal and less rigorous proofs is that incorrect programs may be 'proved' correct as a result of hidden assumptions in the minds of both the author and the reader of such proofs. (For a related discussion see Reference 10.) Neither may be aware of such assumptions and hence neither foresees the possibility that an occasional hidden assumption may indeed be invalid. Hidden assumptions can go unnoticed for a long time. Only the diligent reader can tell us if we are guilty of hidden assumptions in the proof above.

Ambiguity and imprecision

Appropriately chosen high-level notations can be very helpful by supporting our intuitive understanding of a sentence. A notation such as $c[i\,..j] = \%\ {**}\ m$ is no less precise nor is it less unambiguous than the first-order formula

$$
\begin{array}{l}
(m<0 \;\&\; j<i) \\
\text{or} \\
(m=j-i+1 \\
\& \\
\forall k(i\leqslant k\leqslant j \rightarrow (c[k] = \backslash b \text{ or } c[k] = \backslash t \text{ or } c[k] = \backslash n)) \\
).
\end{array}
$$

But since the notation is in an informal and incompletely specified language both imprecision and ambiguity can result (for instance in the interpretation of various operators, and their precedence). Although we do not claim that our notations of programming language and of the language of assertions cannot be improved further, we do claim that there is no loss of precision or of unambiguity.

Wrong inferences

The possibility of incorrectly inferring from known facts exists in all proofs be they of programs or of mathematical theorems. Increasing the formalism and decreasing the 'quanta' of inference in each individual step makes it possible to check them mechnically. This is extremely tedious for humans, and not yet practical for computers. We regard wrong inferences as being less serious than hidden assumptions as one's colleagues are more likely to bump into the latter.

4.2. Some technical issues

Our informal way of proving raises some technical issues among which we briefly mention two:

Forward substitutions

In our proofs of $\{P\}$ lines $i\,..\,j$ $\{Q\}$, our arguments were of the form 'assume P is true and consider lines $i\,..\,j$ whose execution results in such and such changes finally resulting in Q being true'. This technique, known as symbolic execution, is a variation of forward substitution.[11] Forward substitutions performed formally are of the form

$$
\begin{array}{l}
\{P(x)\} \\
x := exp(x) \\
\{\exists X(P(X) \quad \& \quad x = exp(X))\}
\end{array}
$$

where P is any property and *exp* any expression involving the variable x. Intuitively, X is the value of x just before the execution of the assignment. Clearly by continuing this process for a large program such as ours, an uncomfortably large number of existential quantifiers will be produced.

We have avoided this problem by saving the old value of (in this case) x by our binding mechanism '*let* $X == x$'. Thus no existential quantifiers are required because we can now write

$$
\begin{array}{l}
\{let\ X == x \quad \& \quad P(x)\} \\
x := exp(x) \\
\{P(X) \quad \& \quad x = exp(X)\}.
\end{array}
$$

While backward substitutions are more common in formal proofs, we have chosen forward substitutions which are more intuitive being close to (symbolic) execution.

Procedure calls

The formal rules available in current literature for handling procedure calls are weak. In our proofs, we have regarded most of them as simply macro-calls. This is quite reasonable since (i) all actual parameters are distinct, (ii) all parameter variables are local to the procedure and (iii) all updating of global variables in a procedure is explicitly asserted.

5. CONCLUSION

The rigour with which a proof may be given varies, and the conjured up expectations differ markedly. We have given here a proof at an intermediate level of rigour of an indenting program for Pascal. It is more convincing than hand-waving and much less formal than, say, first-order logic-like proofs. We do not claim that our proof establishes beyond doubt the correctness of the program. Our objectives would have been served if the reader's confidence in the program matches that which he may have had after considerable testing of the program. Speaking from personal experience, we can say that our own understanding of the program increased markedly and we have a better insight of the problem and the lapses of lexical structure of Pascal. We sincerely doubt if this level of understanding and insight would have been possible by elaborate testing.

REFERENCES

1. P. Mateti, 'A specification schema for indenting programs', *Software—Practice and Experience*, **13**, 163–179 (1983).
2. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Pitman and Computer Science Press, 1978.
3. P. Mateti, 'Documentation of a program *indent*: a model for the complete documentation of computer programs', *Technical Report* (forthcoming), Department of Computer Science, University of Melbourne, Australia, 1980.
4. R. C. Linger, H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Massachusetts, 1979.
5. Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.
6. E. W. Dijkstra, 'Formal techniques and sizable programs', in K. Samelson (ed.), *E.C.I. Conference 1976*, Lecture Notes in Computer Science #44, Springer-Verlag, 1976.
7. E. W. Dijkstra, 'A more formal treatment of a less simple example', in F. L. Bauer and M. Broy (eds.), *Program Construction*, Lecture Notes in Computer Sciences #69, Springer-Verlag, 1976.
8. R. W. Floyd, 'The paradigms of programming', *CACM*, **22** (8), 455–460 (1971).
9. S. Gerhart and L. Yelowitz, 'Observations of fallibility in applications of modern programming methodologies', *IEEE Transactions on Software Engineering*, **SE-2** (3) 195–207 (1976).
10. I. Lakotas, *Proofs and Refutations: The Logic of Mathematical Discovery*, Cambridge University Press, 1976.
11. J. A. Darringer and J. C. King, 'Applications of symbolic execution to program testing', *Computer*, **11** (4), 51–59 (1978).