AN ABSTRACT OF THE THESIS OF

Kirt Alan Winter for the degree of Master of Science in Computer Science presented on March 18, 1988.

Title: A Prototype Intelligent Prettyprinter

Redacted for privacy

Abstract approved: _____
Curtis R. Cook

Prettyprinters are software tools that format program source code so that it conforms to certain standards of consistency and hence improves readability. Traditionally, these standards were fixed for a particular prettyprinter as indicated by a literature survey, with very little or no supporting evidence that the formatting style improves readability. Given these uncertanties, IPP (Intelligent PrettyPrinter), a prettyprinter described in this thesis, was designed in an effort to introduce flexibility to prettyprinters.

This thesis describes a flexible prototype prettyprinter that learns the user's preferred format style from a sample program supplied by the user. Although IPP was designed for Pascal, the concepts and techniques appear extensible to other programming languages.

A Prototype Intelligent Prettyprinter

by

Kirt Alan Winter

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed March 18, 1988

Commencement June 1988

APPROVED:

Redacted for privacy

Professor of Computer Science in charge of major

Redacted for privacy

Head of Department of Computer Science

Redacted for privacy

Dean of Graduate School

Date thesis is presented March 18, 1988

TABLE OF CONTENTS

LIST OF FIGURES

A PROTOTYPE INTELLIGENT PRETTYPRINTER

1. INTRODUCTION

The term "prettyprinting" was defined by Ledgard as: "...the spacing of a program to illuminate its logical structure."[LEDG75] Thus, a prettyprinter is a program that formats a source code listing to illuminate its structure. A prettyprinter allows a programmer to concentrate on the functionality of the code and not how it is formatted.

There is no commonly accepted program source code format or layout for most if not all programming languages. Therefore, individual preferences have led to a wide variety of formatting styles. Prettyprinters have to this point tended to reflect the formatting preferences of the prettyprinter's designer.

This thesis will describe some of the issues involved in prettyprinting, as well as the implementation of a prototype intelligent prettyprinter (IPP) for Pascal source code. IPP "learns" the style preferred by the user by examining a piece of sample source code formatted in the preferred style.

The second chapter describes the issues behind prettyprinting and formatting style, and motivates the development of a flexible prettyprinter. The third chapter examines the user interface issues that underlie a flexible prettyprinter implementation. Chapter four

takes a look at the IPP's capabilities and design. In chapter five, the problems and limitations that exist within IPP (and perhaps flexible prettyprinters in general) are discussed. It also discusses the extendibility of IPP to other programming languages. Finally, chapter six reviews IPP's contributions to prettyprinting, and discusses possible future directions for research in programming style and prettyprinting.

2. PRETTYPRINTERS AND FORMATTING STYLE

As mentioned in the introduction, prettyprinting is an effort to illuminate the logical structure of a program, and thereby improve its readability.[LEDG75] There have been many articles which describe various prettyprinters and/or formatting styles for the Pascal programming language ([BATE81], [CRID78], [GROG79], [MARC81], [PETE77]). In the following discussion, we will also concentrate on prettyprinting issues as they relate to Pascal.

This chapter describes the functions of a prettyprinter, source code formatting style guidelines, and the inflexibility of current prettyprinters.

2.1. PRETTYPRINTER FUNCTIONALITY

In changing the spacing of a program, a prettyprinter needs much of the functionality of a language compiler. The prettyprinter must recognize keywords, language constructions, and the white spaces that appear between them. The prettyprinter replaces the white spaces (carriage returns, tabs and spaces) in the program using formatting instructions that specify what layout the source code listing should have. In this manner, the format of the program is transformed into the style known by the prettyprinter. There are numerous prettyprinters available for most languages, each having different formatting conventions which the implementer believes best displays the logical structure of the

program. A basic question is how does one go about choosing a particular set of formatting conventions or guidelines for formatting?

2.2. CHOOSING A FORMATTING STYLE

In Software Maintenance: The Problem and Its Solution [MART83], the authors devote an entire chapter to programming style issues, stating (among other things) the following:

> "A good style is simple, consistent, and complies with standard conventions. ... Above all, it clarifies, not obscures what the author is communicating."[MART83]

This implies that a style should in some manner improve the readability of the source code. Martin and McClure also emphasize that consistency is a major goal.

> "A basic principle of structured programming is that standardization of the program will improve its readability." [MART83]

For the most part, language textbooks suggest very little when it comes to formatting style, except for consistency.

> When you write programs, try to approximate the format of programs shown in this book. Above all, be consistent. [COOP85]

However, consistency is difficult to define. Is a programmer who indents single statements four spaces in an IF construction but five spaces in a WHILE construction practicing consistent style? What about a programmer who indents one space extra for each level of indentation (four for the first control structure, five

for the second, etc.)? What if the programmer places the BEGIN of a BEGIN-END block on the same line as the WHILE keyword, but on the next line after an IF keyword?

Hence, there are two guidelines to follow in the design of any prettyprinter. First, according to Ledgard's original goal [LEDG75], it should improve the readability of the program. Second, it should enforce some form of consistent style.

2.3. CURRENT PRETTYPRINTERS

Enforcement of formatting consistency seems to be the major function of most prettyprinters. The "consistent" style rules are generally ones that are chosen by the author of the prettyprinter. The choice of formatting style (and the resulting layout of the source code) range from the expected to the slightly bizzare as shown by the IF statement examples in Figure 2-1.

For the most part, the originator of the style offers little support that his/her style rules improve readability. No experimental or empirical data supporting the choice of a particular set of style rules is mentioned.

```
A)  [BATE81]
     if condition
     then begin
       statement
     ; statement
     ; statement
     end {if}

B)  [GROG79]
     if condition
        then begin
                  statement;
                  statement;
                  statement
             end

C)  [PETE77]
     if condition then
        begin
           statement;
           statement;
           statement
        end
```

Figure 2-1 Examples of IF Formatting Styles

2.4. STYLE AND COMPREHENSION

If we assume that there is no difference in the effort required to produce any particular formatting style (as suggested by using a prettyprinter), then one could argue that comprehension should be the deciding factor in choosing a programming style.

However, at this point in time, the contribution of formatting style to the comprehension of source code is only partially understood. There is some evidence that layout or typographic style affects readability ([OMAN87], [HANS85]), but the magnitude of its impact is not known. Certainly no one has found an ideal style.

In one experiment [OMAN87], students were given a comprehension test involving functionally identical pieces of Pascal source code differing only in formatting style. The subjects were also asked to give their opinion of the style that the sample was written in. One of the three format styles led to better understanding (more correct answers in less time) of the function of the source code, but interestingly, there was no significant difference in the students' opinions of the three styles.

It would seem highly unlikely that there is a "best" format for comprehension, in the sense that a majority will have superior comprehension for that style compared to all others. It would seem more plausible that no one style will be best for any sizable group and that a majority will have different preferences. Hence, format style seems highly individualistic.

Given these observations, we now turn to the main limitation of most current prettyprinters.

2.5. FLEXIBILITY NEEDED

It has been this author's experience that when presented with the question: "Why don't you use the brand X prettyprinter on your code?" the usual response is in the form: "Because it doesn't put the darn <token1> in the right spot after the <token2>."

Even if a prettyprinter's designer believes he/she has developed THE correct way to format a program, a user

may prefer a different format. And since we as yet do not know which style is "best," we have no way of knowing whether the programmer's choice of spots for <token2> is better or worse than anyone else's.

What appears to be needed is a prettyprinter with the flexibility to allow a user to specify the formatting style that he/she prefers. It would provide all the consistency advantages of traditional prettyprinters plus it would allow users to specify the format style they prefer.

One notable exception to the inflexible prettyprinters produced up to this point is a LISP prettyprinter that allows the user to control the appearance of the output [WATE83]. This prettyprinter (written in LISP) allows users to specify their own "deformat" functions that provide a template for various types of control structures and functions. However, writing a "deformat" function could hardly be described as a trivial task, at least for a new user.

IPP (Intelligent Prettyprinter), the prototype prettyprinter described in this thesis, is a flexible prettyprinter for Pascal source code. IPP can produce a wide range of formatting styles because it learns the user's preferred style conventions from a sample of code supplied by the user.

3. SPECIFYING A FORMATTING STYLE

Formatting styles are composed of numerous rules. What is the best way to actually go about specifying these rules to a prettyprinter? It would seem to depend heavily on the formatting method used by the prettyprinter. In IPP, for example, it was determined early that the "whitespace token" method would be used to capture the formatting style. A whitespace token is defined as the collection of newlines, tabs, and spaces that lie between keywords, symbols, or language constructions in a language source program. By capturing the rules for these whitespace tokens, a prettyprinter can duplicate a particular formatting style. A quick count showed that at least 40 of these tokens would be required to capture just the control structures of the Pascal language.

The problem addressed in this chapter is how the user can conveniently specify these tokens to the prettyprinter.

3.1. A MENU SYSTEM

One can envision a menu system that would allow a user to set up an formatting style by multiple menu selections, each followed by some type of "dialog" box which would allow the specification of one or more whitespace tokens. Since each whitespace token consists of two values (more on this later), simple arithmetic shows that at least 80 values must be entered (or checked

if some default method is used) to specify one formatting style.

While one could probably imagine a friendlier way (as opposed to typing integers) to select the tokens, there is still a considerable amount of information that has to be entered into the formatter. Menus certainly are part of the state of the art in user interfaces (Macintosh OS, Microsoft Windows, IBM Presentation Manager), but it would be tedious and burdensome for the user to have to specify whitespace token rules in this manner.

To insure the widest possible coverage of language users, the final target systems of a prettyprinter should include the command line environments as well (UNIX, VMS, MS-DOS). This is another argument against embedding whitespace token specification in layers of menus and dialogs that can be quite cumbersome in the more traditional environment where menus often require extensive keyboard interaction for a user to select options. While user-interface designs do (as they probably should) change depending upon the target system, it does seem that depending on too many features of one type of interface would limit the portability of a prettyprinter.

3.2. AN ASCII "STYLE SPECIFICATION" FILE

Another specification form that was considered was a simple ASCII file that defined the required whitespace

token rules. This would allow a user much the same utility of a menu system that used defaults (a user only has to change parts of the file). This kind of interface is adaptable to both command-line and window/menu systems. However, users would still have to examine each of the possible settings to determine if they would need to be modified to specify a new style rule.

3.3. LEARNING FROM A CODE SAMPLE

From a programmer's standpoint one can best describe one's source code format style by showing another person what it looks like. The question arises: Why couldn't a similar method be used to specify formatting style to a pretty-printer?

The answer (within some limits) is that it can. In parsing a source program for output, a prettyprinter recognizes the spaces between keywords in control structures and replaces them. By extending the prettyprinter so that it not only recognizes, but analyzes and saves whitespace tokens in another file, one can create a prettyprinter that "learns" a style from a code sample provided to it. This is the method that is used to specify a particular formatting style to IPP.

The code sample can be any program with the desired formatting style. It may be a program written by the user or another programmer. Or it may be the special program supplied with IPP that contains instances of all whitespace token rules that IPP recognizes. All the user

has to do is to use an editor to format this program in the style that he/she prefers.

```
program learn_if_elses;

begin
        if (color = yellow) then
                paint_it
        else
                dont_paint_it;
        if not running then
        begin
                fix_it;
                check_it
        end
        else if not(running_well) then
                tune_it_up
        else
        begin
                brag_to_friends;
                take_care_of_it
        end
end.
```

Figure 3-1 A Style For IPP to Learn

```
program
test_if_elses;

begin
        if crying then begin
                feed_baby;
                if baby_not_full then feed_it_more
                else put_bottle_away;
                burp_baby
                end
        else if wet then change_baby
        else begin
                play_with_baby;
                talk_to_baby
                end
end.
```

Figure 3-2 A Piece of Code For IPP to Format

```
program test_if_elses;

begin
    if crying then
    begin
        feed_baby;
        if baby_not_full then
            feed_it_more
        else
            put_bottle_away;
        burp_baby
    end
    else if wet then
        change_baby
    else
    begin
        play_with_baby;
        talk_to_baby
    end
end.
```

Figure 3-3 Applying a Formatting Style

If we tell IPP to learn the style from the small program in Fig. 3-1 using the default settings for tab expansion (four spaces) and have it format the program in Fig. 3-2, then the resulting output is shown in Fig. 3-3. Note that many source code editors allow a tab character to be defined in terms of spaces. To allow for this kind of editor and to make IPP's job easier (only spaces and newlines have to be considered), the tab expansion size is left as a command line option, with a default of four spaces.

So, what did IPP learn from the code in Fig. 3-1? First, IPP learned that the name of the program should follow the keyword PROGRAM with one space between them. Other things that IPP learned were where a begin should appear in IF and ELSE constructions, that the components of a compound statement should line up one tab stop deeper than their controlling BEGIN, as well as where to

put THEN keywords, and how to handle ELSE-IF constructions.

A current limitation of this "show me" type of learning is the absolute (as opposed to relative) alignment of source code. Examples of this type of formatting are often found in comment blocks and case statements. This will be more fully discussed in the "Issues and Limitations" chapter.

4. DESIGN AND IMPLEMENTATION OF IPP

IPP was designed and implemented with Marca's four rules of formatter design in mind [MARC81]. They are:

1. Be consistent in generating output
2. Allow easy specification of input
3. Never lose text because it cannot properly format
4. Work at a reasonable speed

4.1. IPP CAPABILITIES

IPP is targeted for the Turbo Pascal dialect of the Pascal language and thus supports standard Pascal. It is intended to show the feasibility of developing a truly flexible and modifiable prettyprinter. By alternatively analyzing and replacing "whitespace tokens" in source files, IPP can emulate a wide variety of different formatting styles. While IPP is certainly an interesting exercise in prettyprinter development, it is not intended as a finished product but as a tool to facilitate further study of the issues it raises.

Currently, IPP recognizes and uses 48 whitespace tokens (each composed of two integers) to capture formatting style. (These whitespace tokens are listed in Appendix A.) IPP does not analyze data declarations, nor does it analyze anything below the statement level. However, it should become clear that the concepts used to capture other control structures' layouts could easily be extended to these two cases as well.

IPP has been developed as a command-line executed program which uses redirection as its source code

input/output form. Options from the command line control the tab to space conversion, and mode (learn or format). For example, if we wanted IPP to learn the formatting style in the file "style1.pas," then the command line would be (the "L" meaning "learn," and the "-3" meaning that tab stops occur every 3 spaces):

ipp L <style1.pas -3

After executing this command IPP is now ready to apply the style that it learned from "style1.pas." If we wish IPP to format the file "messypro.pas" and create a formatted file called "neatpro.pas," then the following command line would be used:

ipp <messypro.pas >neatpro.pas

In the learn mode, IPP is given a source file to analyze in order to imitate its format. IPP identifies and analyzes whitespace tokens, storing them in a style sheet that is later written to disk.

In the format mode, IPP is given a source file to format. IPP identifies whitespace tokens and replaces them in the output stream with those previously stored in its "style sheet" file. This changes the source file's original format to the formatting style previously learned by IPP.

4.2. WHITESPACE TOKENS

The whitespace in a source program file is (in freely formatable languages) completely ignored by the compiler. Thus, any number of "whitespace" characters

can be used between keywords and symbols. Whitespace only serves as a delimiter between the tokens of the language. However, the way in which one uses whitespace can have a great effect on the visual layout of the source code.

For IPP, a whitespace token consists of two integer values. One is for the number of newline characters, and the other for the change of indentation from the previous line. For example, consider the Pascal FOR statement (labeled with "(a)") in Fig. 4.1.

```
(a) FOR id := value TO value DO
(b) FOR <FOR1> id := value <FOR2> TO <FOR3> value <FOR4> DO <FOR5 or FOR6>
```

Figure 4-1 A FOR Statement and IPP's Whitespace Extensions

There are several places (represented by spaces in Fig. 4-1(a)) that we could choose to examine for whitespace tokens. In IPP, this Pascal construction is represented in Fig. 4-1(b). The areas enclosed <FOR1>, <FOR2>, ..., <FOR6> signify where IPP looks for whitespace. Fig. 4-2 a formatted FOR statement and its whitespace token values.

```
FOR X := 1 to 5
 DO
   BEGIN

Whitespace Token          Change        Newlines
      <FOR1>                +1             0
      <FOR2>                +1             0
      <FOR3>                +1             0
      <FOR4>                +1             1
      <FOR5>                +2             1
```

Figure 4-2 What IPP learns From an Example FOR Statement

In the example, the <FOR5> and <FOR6> tokens are used depending upon whether the FOR's code is a single statement or a BEGIN-END block. If IPP is in the output mode, then it simply finds and replaces whitespace tokens with ones previously learned. For other areas of whitespace, the whitespace is either placed directly in the output, or modified to line up with the beginning of the previous line, depending on the situation.

It is certainly possible to look for more instances of whitespace than IPP does. In fact, for a truly usable prettyprinter one would probably consider adding many more. For example, IPP does not look below the statement level to format. This means that arithmetic expressions, function or procedure calls, boolean expressions, and several other features that could be formatted are not.

4.3. IPP SOURCE CODE MODULE FUNCTIONS

IPP consists of six separately compilable C modules and was developed on an MS-DOS system using Microsoft C 4.0. The total size of the source code is approximately 45,000 bytes. The complete IPP source code is included in Appendix B.

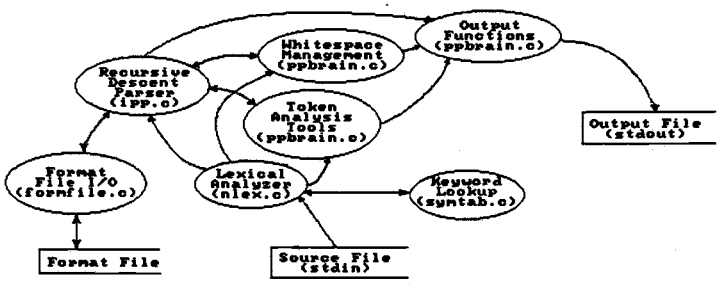

Figure 4-3 Data Flow in IPP

4.3.1. NLEX.C

The NLEX.C module is responsible for the basic lexical analysis of the source code. In addition to the operations performed by most lexical analyzers, NLEX returns whitespace tokens that are used and/or replaced by the PPBRAIN.C module. It is also necessary to look ahead up to two tokens in the input stream in some cases, and NLEX contains the code necessary to perform this operation.

4.3.2. SYMTAB.C

SYMTAB.C is a simple module that simply serves to identify keywords in identifiers returned by NLEX. Since

the number of keywords that need to be identified in Pascal is small, a simple binary search is used to determine if a given identifier is in fact a keyword.

4.3.3. FORMFILE.C

FORMFILE.C performs all input and output associated with style files. A style file is simply a collection of approximately 100 integers which describe the formatting characteristics that are to be used by the prettyprinter.

4.3.4. QUEUE.C

QUEUE.C is a reusable module developed to implement the queue abstract data type. It is used to hold tokens that were retrieved from the lexer in look-ahead operations. It uses LISP-like pointer structures to implement a very general queue.

4.3.5. PPBRAIN.C

PPBRAIN.C contains the functions which perform the low-level formatting transformations. Functions are provided to allow easy look-ahead for particular tokens, as well as for input and output of Pascal source code. Most of the functionality required by the recursive decent parser for learn and output is defined here. In fact, this module is the only one which interprets whether the prettyprinter is in learn or output mode.

4.3.6. IPP.C

IPP.C consists of the main program and the recursive decent parser for Pascal. It currently implements all Pascal control structures, but does not parse below the

statement level. It frequently calls a look-ahead function to insure that the appropriate places for whitespace tokens are recognized.

Various other files are used to declare structures and enumerated types that are used by the other modules.

5. ISSUES AND LIMITATIONS

IPP was passed on to several "beta testers" to collect their comments. In general, they were quite pleased with the results. One tester used it to format a program written by a programmer who was just learning Pascal (having been a FORTRAN programmer for some years) and lined everything up at the left hand margin. There was clearly some excitement in his eyes as the previously unreadable code was reformatted to reflect his personal style, as he was in charge of maintaining that program.

In this chapter, some of the issues raised by IPP's use are discussed.

5.1. WHY PASCAL?

Early in the development of IPP, it was determined that one language should be targeted for ease of development and experimentation. Pascal was chosen for two major reasons. First, Pascal has an LL(1) grammar, making it easy to construct a recursive decent parser that recognizes it. Second, since Pascal is similar to other programming languages with roots in ALGOL, observations that apply to Pascal should be easily extendible to similar programming languages, if not to languages in general.

5.2. CONSISTENCY IN IPP

While a consistent formatting style is widely accepted as important for program comprehension, we have seen that there is some difficulty in deciding where to

draw the line for consistency when it comes to a specific language. IPP does draw a few of these. For example, IPP expects the statements enclosed by BEGIN-END pairs to all have the same indentation from the BEGIN keyword, and expects all BEGIN-END pairs to be done in the same way. The method in general does not place any restrictions on this (and even IPP could be easily modified to allow BEGIN-END pairs controlled by an IF to be indented differently than those controlled by a WHILE, for example), but designing a prettyprinter with the ultimate in flexibility may require larger sample files than anyone is prepared to provide. It may even become prohibitive to store all of the whitespace tokens.

IPP also ignores any physical line length. Unless the source file breaks up a long statement into two or more lines, IPP will not. Extending IPP to handle these conditions was not within the scope of the prototype, but could probably be added by including two cases of whitespace within statements, or in some manner designating which tokens or constructions are fair game to split on.

Another limitation of IPP is the fact that all whitespace tokens are stored in relative terms as the change in indentation from the previous line. Certain formatting features could be characterized as more "absolute" than "relative"; a common example is the

alignment of comments in some programmers' code. This is discussed in detail in section 5.6.

5.3. INCLUSION OF ERROR MESSAGES

IPP, being a prototype, is a simplification in many respects. Error messages are not provided, and errors may cause unpredictable results, depending on the type.

5.4. SYNTAX ERROR CHECKING

IPP assumes a syntactically correct source program. In many cases, IPP will "hang" if a syntactically incorrect program is given to it in either phase. For a production model it would be advisable to detect errors and at least terminate gracefully.

5.5. INCONSISTENT STYLE SAMPLES

As discussed in an earlier section, inconsistency of style is a difficult problem. A prettyprinter like IPP will inevitably be faced with a programmer who uses two different formatting styles at some point. To IPP, this means replacing a previously defined whitespace token with another. This is easy enough to detect, and an appropriate error message could be provided, but IPP simply replaces the previous value of the token without a message. The current advantage of this is that while comments may cause problems in learning, IPP only remembers the last method used; the programmer's intent will be preserved as long as the last example used is understandable.

Going one step further, one might choose to allow more flexibility than is usually considered consistent. For example, one might wish to allow a user to indent BEGIN-END pairs of IF statements differently than those of FOR statements. In this case, a check of several related whitespace tokens could be used to warn of a potentially inconsistent style. IPP could choose to warn users who place the BEGIN on the same line as the IF, but on the line after a WHILE, that their style was inconsistent. Options could be included on the command line to control how "picky" the prettyprinter will be. This would create a prettyprinter of real value in detecting the presence of inconsistent formatting practices.

5.6. ABSOLUTE ALIGNMENT TECHNIQUES

It appears that the most serious impediment to the method used by IPP to store whitespace tokens is the comment block. Consider the code example in Fig. 5-1.

```
for x := 1 to 10 do                          {for all items}
begin                                        {do these steps}
    {do check} check(j[x]);                  {examines limits}
    compute_upon(j[x])                       {add to totals}
end;
```

Figure 5-1 Absolutely Aligned Comment Blocks

First of all, we have a series of comments that are lined up at an absolute point (which doesn't depend on the indentation of the code). Second, comments occur in

the areas previously assumed to only contain whitespace. This brings up a sticky problem. IPP's lexer returns the comment block within the whitespace token, but which area of whitespace (the one before the comment, the one after after the comment, or perhaps some combination of the two) is the whitespace token that the programmer intended?

Looking between the DO and BEGIN keywords of Fig. 5-1 above illustrates the problem that occurs if we learn the first case (the one before the comment). The first whitespace token is simply a series of spaces. We can tell that the programmer probably intends it to be specifically for the alignment of the comment, but how would a prettyprinter determine that? If we instead choose to learn the second whitespace (the one after the comment), that causes problems with a comment that occurs on the same line as, but before the statement that should be aligned (see Fig. 5-2). What about the case when we have "WS comment WS comment WS" (as in Fig. 5-1 between the BEGIN and the first statement in the block, where the programmer would probably have the prettyprinter learn the second whitespace token)?

```
{Learn Sample}
for x := 1 to 10 do
begin
 {do check} check(j[x]);
            compute_upon(j[x])
end;
```

```
{Possible Misinterpretation}
for x := 1 to 10 do
begin {do check} check(j[x]);
compute_upon(j[x])
end;
```

Figure 5-2 Problems With Preceding Comments

Of the possible methods of handling these situations, IPP chooses to ignore all whitespace and comments until the last whitespace before the token. Fortunately, the absolutely aligned comment that precedes a statement seems to occur much less frequently than the one absolutely aligned at the end, so IPP usually does a reasonable job of formatting these cases.

The aforementioned area of difficulty is less prevelant in the output mode, where (in IPP) the last whitespace is the one that is replaced. If IPP learned the correct method originally from the example in Fig. 5-1 (the second whitespace being chosen), then the output of this particular statement would be...

```
for x := 1 to 10 do                                  {for all items}
begin                                                {do these steps}
    {do check}
    check(j[x]);              {examines limits}
    compute_upon(j[x])                               {add to totals}
end;
```

Figure 5-3 Manifestation of Comment Problems

While it does not look exactly as we would like, it at least has all statements aligned between the BEGIN and END.

This is the essence of the problem. To correctly handle cases like this, one needs a more global view of the source code in order to make intelligent decisions about where to put comments. This problem is even more

formidable in the learn mode, where it is difficult to condense the whitespace and comment series into the indentation truly desired by the programmer. It is this sort of difficulty that suggests that the file that is fed to IPP-like prettyprinters in the learn mode should be devoid of these comments until a better way of handling them is developed. This might even require a two-pass prettyprinter.

This problem of absolutely aligned comment blocks and other "absolute" features of formatting style certainly deserve further work. In particular, correct interpretation of them during the learn phase requires special consideration.

5.7. EXTENDIBILITY TO OTHER LANGUAGES

To prettyprint another language in an IPP-like manner, one must simply modify the grammar's language to include whitespace tokens in the appropriate place. The advantage of Pascal is that it is an LL(1) grammar (LL(2) with the addition of whitespace tokens) and a recursive descent parser can easily be made for it. It is somewhat more difficult for languages that do not fall into this category (C for example).

The ultimate solution is probably some sort of toolset that can be used with a YACC-like program. The problem with YACC is that unlike the recursive descent parser which is top down, YACC is bottom up. This means that one would most likely have to construct some sort of

parse tree and then print when the appropriate level is reached. For a series of nested statements, this point is hard to determine, and may require the construction of very large parse trees. The essence of the problem is: IPP-like prettyprinters need to pass information down while YACC-like parser generators are good at passing information up.

6. CONCLUSIONS

6.1 SUMMARY OF CONTRIBUTIONS

This thesis has described the need for, and the development of a flexible prettyprinter for Pascal source code. To summarize, IPP represents two advances in the evolution of prettyprinter thought.

6.1.1. FLEXIBILITY

IPP is flexible, it can learn a wide variety of formatting styles. While there has been at least one other flexible prettyprinter developed for LISP [WATE83], IPP is apparently the first flexible Pascal pretty-printer.

6.1.2. USER INTERFACE

As was discussed earlier in this thesis, the specification of a formatting style to a prettyprinter raised some interesting user interface issues. IPP solves the problem of the volume of information that has to be given to the prettyprinter by learning a formatting style from a sample piece of code. This is much more convenient for the user than the LISP example [WATE83] discussed earlier.

6.2. APPLICATIONS OF INTELLIGENT PRETTYPRINTERS

Intelligent prettyprinters like IPP have many applications, just as traditional prettyprinters do. However, the flexibility of a prettyprinter can do more than encourage its use. The following sections describe briefly some possible uses of IPP-like prettyprinters.

6.2.1. ADHERENCE TO STYLE STANDARDS

In a company that enforces certain formatting rules to insure that all source code is consistent and readable could simply feed a sample file to an IPP-like pretty-printer and then require that all source code be by the prettyprinter before being accepted.

IPP makes it possible for members of development teams who find that the company's style causes them difficulty in understanding or maintaining the code to choose to keep their own style file available and work with source code in a style they prefer. While it is still unclear as to what styles lead to better comprehension (and what productivity factors lie therein), there are clearly differences. Hence IPP accommodates both a company standard and individual preferences.

6.2.2. STYLE VS. COMPREHENSION RESEARCH TOOL

IPP could be used to easily develop experimental material for studies of the effects of formatting style on comprehension. There are several interesting possibilities in this area.

One might choose to examine novice programmers in an attempt to learn which textbooks in Pascal use formats that lead to better comprehension. Are these conclusions applicable to experts, and if not, what styles are better for experts? If more than one good style exists, are

there some common formatting features that can be identified?

Another study could look at a person's "personal" style and whether it is a good style for his/her comprehension. If so, to what degree can the style vary before a comprehension difference is noted, and is this a sharp line, or a gradual drop off?

6.2.3. IDENTIFICATION OF AUTHORSHIP

If IPP-like prettyprinters can be extended to the point that comments can be handled gracefully in the learn phase, can they be used to identify a particular author of a program? In this case the style files generated by the prettyprinter could be compared to determine how close one style is to another.

According to [OMAN87A], formatting style factors (among them formatting style) are in fact a good identifying tool to use in determining the authors of programs. He also points out that using traditional software complexity metrics for identifying authors is unreliable when seeking to identify possible instances of student plagiarism on programming assignments.

Are programmers' styles different enough to use their IPP style sheets as fingerprints? In the IPP example, there exist N^{48} different style files, with N itself theoretically being infinite, but practically much smaller. It seems like style files would be a good identifier, but it might depend on how much a programmer

"develops" their style, and how much they imitate another one.

Another interesting exercise would be to examine students beginning with their first exposure to a language, and following them through several years of experience. How dynamic is one's style? At what point (if any) does it start to become more static? At what point could a prettyprinter be used to detect plagiarism in student programming assignments?

6.3. SUGGESTIONS FOR FURTHER PRETTYPRINTER RESEARCH

IPP is certainly not the last word in prettyprinter development. At best, IPP can serve to focus attention on the issues behind formatting style, as opposed to just arbitrarily advocating one style over another.

As mentioned earlier, IPP has some trouble with comment blocks and other absolutely aligned style features. These problems, and especially how to properly interpret them in the learn mode deserves some study.

It would be an extremely informative exercise to implement an IPP-like prettyprinter for a language other than Pascal. While it appears that the techniques learned in developing IPP for formatting Pascal are directly applicable to other languages, there will no doubt be some sticky implementation details to work out. These could lead to a refinement of the method itself.

In particular it would be interesting to attempt to apply IPP's ideas to a non ALGOL-like language or non-procedural languages.

BIBLIOGRAPHY

[BATE81] Bates, R. M. (1981) A Pascal Prettyprinter With a Different Purpose. Association for Computing Machinery SIGPLAN Notices, Vol. 16, No. 3.

[COOP85] Cooper, D. and Clancy, M. (1985) "Oh! Pascal!" W. W. Norton & Company, New York.

[CRID78] Crider, J. E. (1978) Structured Formatting of Pascal Programs. Association for Computing Machinery SIGPLAN Notices, Vol. 13, No. 11.

[GROG79] Grogono, P. (1979) On Layout, Identifiers and Semicolons in Pascal Programs. Association for Computing Machinery SIGPLAN Notices, Vol. 14, No. 4.

[HANS85] Hansen, J. and Sands, B. (1985) Some Design Considerations for a "C" Source Code Pretty Printer. Association for Computing Machinery SIGSMALL/PC Notes, Vol. 11, No. 2.

[KERN78] Kernighan, B. W. and Plauger, P. J. (1978) "The Elements of Programming Style" McGraw-Hill, New York.

[LEDG75] Ledgard, H. F. (1975) "Programming Proverbs." Hayden, Rochelle Park, New Jersey.

[MARC81] Marca, D. (1981) Some Pascal Style Guidelines. Association for Computing Machinery SIGPLAN Notices, Vol. 16, No. 4.

[MART83] Martin, J. and McClure, C. (1983) "Software Maintenance: The Problem and Its Solutions" Prentice-Hall, New Jersey.

[OMAN87] Oman, P. and Cook, C. (1987) A Paradigm for Programming Style Research. Oregon State University Computer Science Department Technical Report. 87-60-7.

[OMAN87A] Oman, P. and Cook, C. (1987) Programming Style Authorship Analysis. Oregon State University Computer Science Department Technical Report. 87-60-11.

[PETE77] Peterson, J. L. (1977) On the Formatting of Pascal Programs. Association for Computing Machinery SIGPLAN Notices, Vol. 12, No. 12.

[WATE83] Waters, R. C. (1983) User Format Control in a LISP Prettyprinter. Association for Computing Machinery Transactions on Programming Languages and Systems, Vol. 5, No. 4.

Appendices

APPENDIX A: MAP OF WHITESPACE TOKENS

The locations of IPP's 48 whitespace tokens are indicated in BNF formalism by the following constructions. Note that this is not a complete grammar of the language that IPP recognizes, but rather a guide to show the locations that IPP learns whitespace.

```
compound-statement ::= BEGIN <BEGIN1> statement { ; statement } <BEGIN2> END

program ::= PROGRAM <PROG1> id <PROG2> ( parameters ) <PROG3> ; <PROG4>
          | PROGRAM <PROG1> id <PROG2> ; <PROG4>

procedure ::= PROCEDURE <PROC1> id <PROC2> ( parameters ) <PROC3> ; <PROC4>
            | PROCEDURE <PROC1> id <PROC2> ; <PROC4>

function ::= FUNCTION <FUNC1> id <FUNC2> ( parameters ) <FUNC3> : <FUNC4>
             id <FUNC5> ; <FUNC6>
           | FUNCTION <FUNC1> id <FUNC2> : <FUNC4> id <FUNC5> ; <FUNC6>

while ::= WHILE <WHILE1> condition <WHILE2> DO <WHILE3> compound-statement
        | WHILE <WHILE1> condition <WHILE2> DO <WHILE4> single-statement

repeat ::= REPEAT <REPEAT1> statement { ; statement } { ; } <REPEAT2> UNTIL
           <REPEAT3> condition

if ::= IF <IF1> condition <IF2> THEN <IF3> compound-statement { else }
     | IF <IF1> condition <IF2> THEN <IF4> single-statement { else }

else ::= ELSE <ELSE1> { if }
       | ELSE <ELSE2> compound-statement
       | ELSE <ELSE3> single-statement

with ::= WITH <WITH1> record-variable-list <WITH2> DO <WITH3>
         compound-statement
       | WITH <WITH1> record-variable-list <WITH2> DO <WITH4>
         single-statement

for ::= FOR <FOR1> assignment <FOR2> TO <FOR3> id <FOR4> DO <FOR5>
        compound-statement
      | FOR <FOR1> assignment <FOR2> TO <FOR3> id <FOR4> DO <FOR6>
        single-statement

case ::= CASE <CASE1> id <CASE2> OF <CASE3> case-list { ; case-list }
         { else } <CASE6> END

case-list ::= : <CASE4> compound-statement
            | : <CASE5> single-statement
```

APPENDIX B: A SAMPLE SESSION

In this section a sample of IPP reformatting a Pascal source code file is presented.

We will assume that all files are located in the same directory. First, we will have IPP learn the style reflected in the program file "style.pas." Note that this program does not make semantic sense, but since IPP is only concerned with it being syntactically correct, it doesn't matter. Also, the command line option -n (in this case -4) tells IPP how far apart tab stops should be, as IPP expands tabs to their equivalent amount of spaces.

Next, we will give IPP a program to format. The files ("example.in," "example.out") are listed in their entirety in this appendix, while the file "style.pas" is listed in Appendix C.

SAMPLE SESSION

```
C> ipp L <style.pas -4
program learn_everything (arglist);

procedure learn_case (arglist);

begin
    case choice of
        'D', 'd' :
            statement;
        'M', 'm' :
            begin
                statement1;
                statement2
            end;
        else
            writeln
    end
end;

function learn_if_elses (arglist) : return_type;

begin
    if condition then
        statement
    else
        statement;
    if condition then
    begin
        statement;
        statement
    end
    else if condition then
        statement
    else
    begin
        statement;
        statement
    end
end;

procedure learn_whiles_n_repeat_untils;

begin
    while condition do
        statement;
    while condition do
    begin
        repeat
            statement;
            statement
        until condition;
        statement
    end
end;
```

```
function learn_fors_n_withs : return_type;

begin
    statement;
    for variable := initial to limit do
        statement;
    with variable_name do
    begin
        statement;
        for variable := initial to limit do
        begin
            statement;
            statement
        end;
        statement;
        statement
    end;
    with variable_name do
        statement;
    statement
end;

begin
    statement
end.

C> ipp <example.in >example.out

C> _
```

EXAMPLE.IN

```
{$U+}
PROGRAM sssianal (input, output);

CONST
   Num_Operators = 163;
   Plus_Sign     = '+';
   Star          = '*';
   Dollar_Sign   = '$';
   Space         = ' ';
   Colon         = ':';
   Dot           = '.';
   Comma         = ',';
   Tab           = #9;
   LineFeed      = #10;
   cha           = '!';
   chz           = '}';
TYPE
  Operator_Rec = RECORD
                   Name : STRING[8];
                   Freq : INTEGER;
                 END;
   Nametype    = STRING[8];

VAR
  Char_Values   : ARRAY[cha..chz,0..4] OF INTEGER;
  Offset        : ARRAY[0..4] OF INTEGER;
  Command :  ARRAY [1..Num_Operators] OF Operator_Rec;
  Comment_count,
  Misc_count,
  Blank_count,
  Varlist_count,
  Line_count :  INTEGER;
  i,j,k,p  :  INTEGER;
  In_File,
  Out_File : TEXT;
  In_FileName,
  Out_FileName :  STRING[14];
  Line,
  InputLine : STRING [126];
  FirstCh : CHAR;
  Name    : Nametype;
  ch      : CHAR;
  No_command,
  flag,
  MoreInput,
  More_of_Command : BOOLEAN;


FUNCTION segment(word: Nametype): INTEGER;
  VAR
    i, j, k, prime, value : INTEGER;
  BEGIN
    prime := 1009;
    value := 0;
    j := length(word);
    FOR i := 1 to j DO
      BEGIN
        k := ord(word[i]) - ord('A') + 1;
        IF (k < 0) OR (k > 26)
          THEN
            k := 27;
        value := ((value * 27 ) MOD prime) + k;
      END;
    value := (value MOD prime) MOD 191;
    value := (((value * 59) MOD 191) + 131) MOD 191;
    segment := value MOD 5;
  END;   {  segment  }
```

```
BEGIN

  { Housekeeping  }

  Comment_count := 0;
  Misc_count := 0;
  Blank_count := 0;
  Line_count := 0;
  Varlist_count := 0;

  { Input character value assignments, offsets, and hash table word  }

  assign(In_File,'C:sssihash.wds');
  reset(In_File);
  FOR ch := cha TO chz DO
    BEGIN
      FOR i := 0 TO 4 DO
        BEGIN
          read(In_File,Char_Values[ch,i]);
        END;
      readln(In_File);
    END;
  FOR i := 0 TO 4 DO
    BEGIN
      read(In_File,Offset[i]);
    END;

  { Input Command Names and set frequencies to 0  }

  readln(In_File);
  FOR i := 1 TO Num_Operators DO
    BEGIN
      readln(In_File,Command[i].name);
      Command[i].freq := 0;
    END;

  { Start to process DB/C program file }

  write('Please enter input file name ');
  readln(In_FileName);
  writeln(lst);
  writeln(lst,'Name of input file is ',In_FileName);
  assign(In_File, In_FileName);
  reset(In_File);
  WHILE NOT EOF(In_File) DO
    BEGIN
      readln(In_File,Line);
      Line[Length(Line)] :=  Space;
      Line_count := Line_count + 1;
      FirstCh := Line[1];
      More_of_Command := TRUE;
      i := 1;
      IF (FirstCh = Dot) OR (FirstCh = Star)
        THEN { Increment comment line count  }
           Comment_count := Comment_count + 1
        ELSE  { Process Command line }
          BEGIN
              IF FirstCh <> Space
                THEN
                  BEGIN { Read past label   }
                    i := pos(Space,Line);
                    Delete(Line,1,i-1);
                    i := 1
                  END;

          { See if rest of line blank  }
```

```
No_command := TRUE;
FOR j := 1 TO Length(Line) DO
  IF Line[j] <> Space
    THEN
      No_command := FALSE;
IF No_command
  THEN
    Blank_count := Blank_count + 1
  ELSE
    BEGIN

      { Find Command name }

      REPEAT
        i := i + 1;
      UNTIL Line[i] <> Space;
      Delete(Line,1,i-1); {Delete spaces before command name }
      i := 1;
      j := pos(Space,Line);

      { Assign Name the command name }

      IF j = 0
        THEN
          writeln(lst,'Error in command length name line = ',Line)
        ELSE
          BEGIN
            Name := copy(Line,1,j-1);
            Delete(Line,1,j-1)
          END;

      { Increment frequency of operation Name   }

      j := segment(Name);
      k := Char_Values[Name[1],j] + Char_Values[Name[length(Name)],j]
           + Length(Name) + Offset[j];
      IF Command[k].Name = Name
        THEN
          Command[k].Freq := Command[k].Freq + 1
        ELSE
          BEGIN
            Misc_count := Misc_count + 1;
            writeln(lst,Line);
          END;

      {See if command continues to next line}

      i := Length(Line);
      WHILE Line[i] = Space DO
        i := i - 1;
      IF Line[i] <> Colon
        THEN
          More_of_Command := FALSE;

   { Count operands in LOAD, STORE, and CLEAR Operators  }

      IF (Name = 'LOAD') OR (Name = 'STORE') OR (Name = 'CLEAR')
        THEN
          BEGIN
            REPEAT
              j := pos(Comma,Line);
              IF j > 0
                THEN
                  BEGIN
                    Varlist_count := Varlist_count + 1;
                    Delete(Line,1,j)
                  END
           UNTIL j = 0;
```

```
                        WHILE More_of_Command DO
                          BEGIN
                            Readln(In_File,Line);
                            Line_count := Line_count + 1;
                            Line := Line + Space;

                           {See if command continues to next line}

                            i := Length(Line);
                            WHILE Line[i] = Space DO
                              i := i - 1;
                            IF Line[i] <> Colon
                              THEN
                                More_of_Command := FALSE;

                           { Count operands in line  }

                            REPEAT
                              j := pos(Comma,Line);
                              IF j > 0
                                THEN
                                  BEGIN
                                    Varlist_count := Varlist_count + 1;
                                    Delete(Line,1,j)
                                  END
                            UNTIL j = 0;
                          END;
                      END;
                    WHILE More_of_Command DO
                        BEGIN
                          readln(In_File,Line);
                          Line_count := Line_count + 1;
                          Line[Length(Line)] :=  Space;

                      {See if command continues to next line}

                          i := Length(Line);
                          WHILE Line[i] = Space DO
                            i := i - 1;
                          IF Line[i] <> Colon
                            THEN
                              More_of_Command := FALSE;
                        END;
                  END;
        END;
    END;
  close(In_File);
  { Output report of counts  }
  FOR i := 1 TO Num_Operators DO
    IF Command[i].Freq > 0
      THEN
        writeln(lst,Command[i].name:10,Command[i].freq:4);
        writeln(lst,'LC = ',Line_count:5,' CC = ',Comment_count:5,' BC = ',
          Blank_count:5,' MC = ',misc_count:5,' Varlist = ',Varlist_count:6);
END.
```

EXAMPLE.OUT

```
{$U+}
PROGRAM sssianal (input, output);

CONST
   Num_Operators = 163;
   Plus_Sign     = '+';
   Star          = '*';
   Dollar_Sign   = '$';
   Space         = ' ';
   Colon         = ':';
   Dot           = '.';
   Comma         = ',';
   Tab           = #9;
   LineFeed      = #10;
   cha           = '!';
   chz           = '}';
TYPE
  Operator_Rec = RECORD
                  Name : STRING[8];
                  Freq : INTEGER;
                END;
   Nametype    = STRING[8];

VAR
  Char_Values   : ARRAY[cha..chz,0..4] OF INTEGER;
  Offset        : ARRAY[0..4] OF INTEGER;
  Command :  ARRAY [1..Num_Operators] OF Operator_Rec;
  Comment_count,
  Misc_count,
  Blank_count,
  Varlist_count,
  Line_count :  INTEGER;
  i,j,k,p  :  INTEGER;
  In_File,
  Out_File : TEXT;
  In_FileName,
  Out_FileName :  STRING[14];
  Line,
  InputLine : STRING [126];
  FirstCh : CHAR;
  Name    : Nametype;
  ch      : CHAR;
  No_command,
  flag,
  MoreInput,
  More_of_Command : BOOLEAN;


FUNCTION segment (word: Nametype) : INTEGER;

VAR
    i, j, k, prime, value : INTEGER;
  BEGIN
    prime := 1009;
    value := 0;
    j := length(word);
    FOR i := 1 to j DO
    BEGIN
        k := ord(word[i]) - ord('A') + 1;
        IF (k < 0) OR (k > 26) THEN
            k := 27;
        value := ((value * 27 ) MOD prime) + k;
    END;
    value := (value MOD prime) MOD 191;
    value := (((value * 59) MOD 191) + 131) MOD 191;
    segment := value MOD 5;
END;   {  segment  }
```

```
BEGIN

  { Housekeeping  }
    Comment_count := 0;
    Misc_count := 0;
    Blank_count := 0;
    Line_count := 0;
    Varlist_count := 0;

  { Input character value assignments, offsets, and hash table word  }

    assign(In_File,'C:sssihash.wds');
    reset(In_File);
    FOR ch := cha TO chz DO
    BEGIN
        FOR i := 0 TO 4 DO
        BEGIN
            read(In_File,Char_Values[ch,i]);
        END;
        readln(In_File);
    END;
    FOR i := 0 TO 4 DO
    BEGIN
        read(In_File,Offset[i]);
    END;

  { Input Command Names and set frequencies to 0  }

    readln(In_File);
    FOR i := 1 TO Num_Operators DO
    BEGIN
        readln(In_File,Command[i].name);
        Command[i].freq := 0;
    END;

  { Start to process DB/C program file }

    write('Please enter input file name ');
    readln(In_FileName);
    writeln(lst);
    writeln(lst,'Name of input file is ',In_FileName);
    assign(In_File, In_FileName);
    reset(In_File);
    WHILE NOT EOF(In_File) DO
    BEGIN
        readln(In_File,Line);
        Line[Length(Line)] :=  Space;
        Line_count := Line_count + 1;
        FirstCh := Line[1];
        More_of_Command := TRUE;
        i := 1;
        IF (FirstCh = Dot) OR (FirstCh = Star) THEN { Increment comment line count  }
            Comment_count := Comment_count + 1
        ELSE  { Process Command line }
        BEGIN
            IF FirstCh <> Space THEN
            BEGIN { Read past label   }
                i := pos(Space,Line);
                Delete(Line,1,i-1);
                i := 1
            END;

          { See if rest of line blank  }
```

```
No_command := TRUE;
FOR j := 1 TO Length(Line) DO
    IF Line[j] <> Space THEN
        No_command := FALSE;
IF No_command THEN
    Blank_count := Blank_count + 1
ELSE
BEGIN

        { Find Command name }
    REPEAT
         i := i + 1;
    UNTIL Line[i] <> Space;
    Delete(Line,1,i-1); {Delete spaces before command name }
    i := 1;
    j := pos(Space,Line);

        { Assign Name the command name }

    IF j = 0 THEN
        writeln(lst,'Error in command length name line = ',Line)
    ELSE
    BEGIN
        Name := copy(Line,1,j-1);
        Delete(Line,1,j-1)
    END;

        { Increment frequency of operation Name   }

    j := segment(Name);
    k := Char_Values[Name[1],j] + Char_Values[Name[length(Name)],j]
    + Length(Name) + Offset[j];
    IF Command[k].Name = Name THEN
        Command[k].Freq := Command[k].Freq + 1
    ELSE
    BEGIN
        Misc_count := Misc_count + 1;
        writeln(lst,Line);
    END;

        {See if command continues to next line}

    i := Length(Line);
    WHILE Line[i] = Space DO
        i := i - 1;
    IF Line[i] <> Colon THEN
        More_of_Command := FALSE;

     { Count operands in LOAD, STORE, and CLEAR Operators  }

    IF (Name = 'LOAD') OR (Name = 'STORE') OR (Name = 'CLEAR') THEN
    BEGIN
        REPEAT
             j := pos(Comma,Line);
             IF j > 0 THEN
             BEGIN
                 Varlist_count := Varlist_count + 1;
                 Delete(Line,1,j)
             END
        UNTIL j = 0;
```

```
                    WHILE More_of_Command DO
                    BEGIN
                        Readln(In_File,Line);
                        Line_count := Line_count + 1;
                        Line := Line + Space;

                              {See if command continues to next line}

                        i := Length(Line);
                        WHILE Line[i] = Space DO
                            i := i - 1;
                        IF Line[i] <> Colon THEN
                            More_of_Command := FALSE;

                              { Count operands in line  }

                        REPEAT
                             j := pos(Comma,Line);
                             IF j > 0 THEN
                             BEGIN
                                 Varlist_count := Varlist_count + 1;
                                 Delete(Line,1,j)
                             END
                        UNTIL j = 0;
                    END;
                END;
                WHILE More_of_Command DO
                BEGIN
                    readln(In_File,Line);
                    Line_count := Line_count + 1;
                    Line[Length(Line)] :=  Space;

                        {See if command continues to next line}

                    i := Length(Line);
                    WHILE Line[i] = Space DO
                        i := i - 1;
                    IF Line[i] <> Colon THEN
                        More_of_Command := FALSE;
                END;
            END;
        END;
    END;
    close(In_File);
  { Output report of counts  }
    FOR i := 1 TO Num_Operators DO
        IF Command[i].Freq > 0 THEN
            writeln(lst,Command[i].name:10,Command[i].freq:4);
    writeln(lst,'LC = ',Line_count:5,' CC = ',Comment_count:5,' BC = ',
    Blank_count:5,' MC = ',misc_count:5,' Varlist = ',Varlist_count:6);
END.
```

APPENDIX C: A COMPLETE "LEARN" FILE

The following piece of code contains all instances of whitespace that IPP learns. Thus, a simple way to teach IPP a formatting style is to format this file in the desired form, and pass it to IPP in the "learn" mode. Other samples of code may be used, but they may leave some whitespace tokens undefined.

```
program learn_everything (arglist);

procedure learn_case (arglist);

begin
    case choice of
        'D', 'd' :
            statement;
        'M', 'm' :
            begin
                statement1;
                statement2
            end;
        else
            writeln
    end
end;

function learn_if_elses (arglist) : return_type;

begin
    if condition then
        statement
    else
        statement;
    if condition then
    begin
        statement;
        statement
    end
    else if condition then
        statement
    else
    begin
        statement;
        statement
    end
end;
```

```
procedure learn_whiles_n_repeat_untils;

begin
    while condition do
        statement;
    while condition do
    begin
        repeat
             statement;
             statement
        until condition;
        statement
    end
end;

function learn_fors_n_withs : return_type;

begin
    statement;
    for variable := initial to limit do
        statement;
    with variable_name do
    begin
        statement;
        for variable := initial to limit do
        begin
            statement;
            statement
        end;
        statement;
        statement
    end;
    with variable_name do
        statement;
    statement
end;

begin
    statement
end.
```

APPENDIX D: IPP SOURCE CODE LISTING

IPP.C

```
#include <stdio.h>
#include "nlex.h"
#include "whitedef.h"
#include "ipp.def"

extern void store_array(struct spc *);
extern void init_lexer(void);
extern struct lex_struct *lex(int ,int );
extern struct lex_struct *look_ahead(int ,int ,int );
extern struct lex_struct *new_lex_ptr(void);
extern struct lex_struct *dispose_lex_ptr(struct lex_struct *);
extern void output_white(struct lex_struct *);
extern void output(struct lex_struct *,int ,int ,int );
extern void setup_array(struct spc *);
extern void white_manage(int ,int ,struct lex_struct *);
extern void rw_until(int ,int ,int ,int ,int );
extern void rw_until_look(int ,int ,int ,int ,int );
extern void rw_until_look_1(int ,int ,int ,int ,int );
extern int do_white_space(int ,int ,int ,int );

void do_for();
void do_while();
void do_if();
void do_repeat();
void do_with();
void do_case();
void do_function();
void do_procedure();

SPACE white_stor[FEND];

void do_arg_list(expand, indent, input)
int expand, indent, input;
{
    rw_until(RPAREN, expand, indent, FALSE, input);
}
```

```
void do_statement(expand, indent, input)
int expand, indent, input;
{
    LEX_PTR lex_ret;
    int done = FALSE;

    lex_ret = look_ahead(1, expand, indent);
    if(lex_ret->token == WHITE_SPACE)
        lex_ret = look_ahead(2, expand, indent);
    while(!done)
    {
        switch(lex_ret->token)
        {
            case CASEkw:    rw_until(CASEkw, expand, indent, TRUE, input);
                            do_case(expand, indent, input);
                            lex_ret = look_ahead(2, expand, indent);
                            done = TRUE;
                            break;

            case IFkw:      rw_until(IFkw, expand, indent, TRUE, input);
                            do_if(expand, indent, input);
                            lex_ret = look_ahead(2, expand, indent);
                            done = TRUE;
                            break;

            case WHILEkw:   rw_until(WHILEkw, expand, indent, TRUE, input);
                            do_while(expand, indent, input);
                            lex_ret = look_ahead(2, expand, indent);
                            done = TRUE;
                            break;

            case FORkw:     rw_until(FORkw, expand, indent, TRUE, input);
                            do_for(expand, indent, input);
                            lex_ret = look_ahead(2, expand, indent);
                            done = TRUE;
                            break;

            case WITHkw:    rw_until(WITHkw, expand, indent, TRUE, input);
                            do_with(expand, indent, input);
                            lex_ret = look_ahead(2, expand, indent);
                            done = TRUE;
                            break;

            case REPEATkw:  rw_until(REPEATkw, expand, indent, TRUE, input);
                            do_repeat(expand, indent, input);
                            lex_ret = look_ahead(2, expand, indent);
                            done = TRUE;
                            break;

            case SEMI:      rw_until_look_1(SEMI, expand, indent, TRUE, input);
                            done = TRUE;
                            break;

            case ELSEkw:
            case UNTILkw:
            case ENDkw:     done = TRUE;
                            break;

            default:        lex_ret = lex(expand, indent);
                            output(lex_ret, indent, TRUE, input);
                            lex_ret = look_ahead(1, expand, indent);
                            if((lex_ret->token == SEMI) || (lex_ret->token == ENDkw))
                                done = TRUE;
                            lex_ret = look_ahead(2, expand, indent);
                            break;
        }
    }
}
```

```
void do_statement_series(expand, indent, input)
int expand, indent, input;
{
    LEX_PTR lex_ret;
    int done = FALSE;

    while(!done)
    {
        do_statement(expand, indent, input);
        lex_ret = look_ahead(1, expand, indent);
        if(lex_ret->token != SEMI)
            done = TRUE;
        else
            rw_until(SEMI, expand, indent, FALSE, input);
    }
}

void do_begin_end(expand, indent, input)
int expand, indent, input;
{
    indent = do_white_space(BEGIN1, expand, indent, input);
    do_statement_series(expand, indent, input);
    indent = do_white_space(BEGIN2, expand, indent, input);
    rw_until(ENDkw, expand, indent, TRUE, input);
}

void do_sub_clause(tok1, tok2, expand, indent, input)
int tok1, tok2, expand, indent, input;
{
    LEX_PTR lex_ret;

    lex_ret = look_ahead(2, expand, input);
    if(lex_ret->token == BEGINkw)
    {
        indent = do_white_space(tok1, expand, indent, input);
        rw_until(BEGINkw, expand, indent, FALSE, input);
        do_begin_end(expand, indent, input);
    }
    else
    {
        indent = do_white_space(tok2, expand, indent, input);
        do_statement(expand, indent, input);
    }
}
```

```
void do_internals(expand, indent, input)
int expand, indent, input;
{
    LEX_PTR lex_ret;
    int done = FALSE;

    lex_ret = look_ahead(1, expand, indent);
    while(!done)
    {
        switch(lex_ret->token)
        {
            case PROCEDUREkw:   lex_ret = lex(expand, indent);
                                output(lex_ret, indent, TRUE, input);
                                do_procedure(expand, indent, input);
                                lex_ret = look_ahead(1, expand, indent);
                                break;

            case BEGINkw:       lex_ret = lex(expand, indent);
                                output(lex_ret, indent, TRUE, input);
                                do_begin_end(expand, indent, input);
                                done = TRUE;
                                break;

            case FUNCTIONkw:    lex_ret = lex(expand, indent);
                                output(lex_ret, indent, TRUE, input);
                                do_function(expand, indent, input);
                                lex_ret = look_ahead(1, expand, indent);
                                break;

            default:            lex_ret = lex(expand, indent);
                                output(lex_ret, indent, FALSE, input);
                                lex_ret = look_ahead(1, expand, indent);
                                break;
        }
    }
}

void do_program(expand, indent, input)
int expand, indent, input;
{
    LEX_PTR lex_ret;

    rw_until(PROGRAMkw, expand, indent, FALSE, input);
    indent = do_white_space(PROG1, expand, indent, input);
    rw_until(ID, expand, indent, FALSE, input);
    indent = do_white_space(PROG2, expand, indent, input);
    lex_ret = look_ahead(1, expand, indent);
    if(lex_ret->token != SEMI)
    {
        rw_until(LPAREN, expand, indent, FALSE, input);
        do_arg_list(expand, indent, input);
        indent = do_white_space(PROG3, expand, indent, input);
        rw_until(SEMI, expand, indent, FALSE, input);
    }
    else
        rw_until(SEMI, expand, indent, FALSE, input);
    do_white_space(PROG4, expand, indent, input);
    do_internals(expand, indent, input);
    rw_until(PERIOD, expand, indent, FALSE, input);
}
```

```
void do_case_body(expand, indent, input)
int expand, indent, input;
{
    LEX_PTR lex_ret;

    do
    {
        rw_until(COLON, expand, indent, TRUE, input);
        do_sub_clause(CASE4, CASE5, expand, indent, input);
        lex_ret = look_ahead(3, expand, indent);
        if(lex_ret->token == ELSEkw)
            rw_until_look(ELSEkw, expand, indent, TRUE, input);
        lex_ret = look_ahead(2, expand, indent);
    }
    while((lex_ret->token != ENDkw) && (lex_ret->token != ELSEkw));
}

void do_case(expand, indent, input)
int expand, indent, input;
{
    LEX_PTR lex_ret;

    indent = do_white_space(CASE1, expand, indent, input);
    rw_until_look(OFkw, expand, indent, FALSE, input);
    indent = do_white_space(CASE2, expand, indent, input);
    rw_until(OFkw, expand, indent, FALSE, input);
    indent = do_white_space(CASE3, expand, indent, input);
    do_case_body(expand, indent, input);
    lex_ret = look_ahead(2, expand, indent);
    if(lex_ret->token == ELSEkw)
        do_else(expand, indent, input);
    rw_until_look(ENDkw, expand, indent, FALSE, input);
    do_white_space(CASE6, expand, indent, input);
    rw_until(ENDkw, expand, indent, FALSE, input);
}

void do_for(expand, indent, input)
int expand, indent, input;
{
    indent = do_white_space(FOR1, expand, indent, input);
    rw_until_look(TOkw, expand, indent, FALSE, input);
    indent = do_white_space(FOR2, expand, indent, input);
    rw_until(TOkw, expand, indent, FALSE, input);
    indent = do_white_space(FOR3, expand, indent, input);
    rw_until_look(DOkw, expand, indent, FALSE, input);
    indent = do_white_space(FOR4, expand, indent, input);
    rw_until(DOkw, expand, indent, FALSE, input);
    do_sub_clause(FOR5, FOR6, expand, indent, input);
}

void do_while(expand, indent, input)
int indent, expand, input;
{
    LEX_PTR lex_ret;

    indent = do_white_space(WHILE1, expand, indent, input);
    rw_until_look(DOkw, expand, indent, FALSE, input);
    indent = do_white_space(WHILE2, expand, indent, input);
    rw_until(DOkw, expand, indent, FALSE, input);
    do_sub_clause(WHILE3, WHILE4, expand, indent, input);
}
```

```
void do_if(expand, indent, input)
int indent, expand, input;
{
    LEX_PTR lex_ret, lex_ret1;

    indent = do_white_space(IF1, expand, indent, input);
    rw_until_look(THENkw, expand, indent, FALSE, input);
    indent = do_white_space(IF2, expand, indent, input);
    rw_until(THENkw, expand, indent, FALSE, input);
    do_sub_clause(IF3, IF4, expand, indent, input);
    lex_ret1 = look_ahead(1, expand, indent);
    lex_ret = look_ahead(2, expand, indent);
    if((lex_ret->token == ELSEkw) || (lex_ret1->token == ELSEkw))
        do_else(expand, indent, input);
}

void do_else(expand, indent, input)
int indent, expand, input;
{
    LEX_PTR lex_ret;

    do_white_space(IF6, expand, indent, input);
    rw_until(ELSEkw, expand, indent, FALSE, input);
    lex_ret = look_ahead(2, expand, indent);
    if(lex_ret->token == IFkw)
    {
        indent = do_white_space(ELSE1, expand, indent, input);
        rw_until(IFkw, expand, indent, FALSE, input);
        do_if(expand, indent, input);
    }
    else
        do_sub_clause(ELSE2, ELSE3, expand, indent, input);
}

void do_repeat(expand, indent, input)
int indent, expand, input;
{
    indent = do_white_space(REPEAT1, expand, indent, input);
    do_statement_series(expand, indent, input);
    indent = do_white_space(REPEAT2, expand, indent, input);
    rw_until(UNTILkw, expand, indent, FALSE, input);
    indent = do_white_space(REPEAT3, expand, indent, input);
    do
    {
        lex_ret = look_ahead(2, expand, indent);
        output(lex(expand, indent), indent, TRUE, input);
    }
    while((lex_ret->token != ENDkw) && (lex_ret->token != SEMI));
    if(lex_ret->token == SEMI)
        rw_until_look_1(SEMI, expand, indent, FALSE, input);
}

void do_with(expand, indent, input)
int indent, expand, input;
{
    LEX_PTR lex_ret;

    indent = do_white_space(WITH1, expand, indent, input);
    rw_until_look(DOkw, expand, indent, FALSE, input);
    indent = do_white_space(WITH2, expand, indent, input);
    rw_until(DOkw, expand, indent, FALSE, input);
    do_sub_clause(WITH3, WITH4, expand, indent, input);
}
```

```
void do_procedure(expand, indent, input)
int indent, expand, input;
{
    LEX_PTR lex_ret;

    indent = do_white_space(PROC1, expand, indent, input);
    rw_until(ID, expand, indent, FALSE, input);
    indent = do_white_space(PROC2, expand, indent, input);
    lex_ret = look_ahead(1, expand, indent);
    if(lex_ret->token != SEMI)
    {
        rw_until(LPAREN, expand, indent, FALSE, input);
        do_arg_list(expand, indent, input);
        indent = do_white_space(PROC3, expand, indent, input);
        rw_until(SEMI, expand, indent, FALSE, input);
    }
    else
        rw_until(SEMI, expand, indent, FALSE, input);
    do_white_space(PROC4, expand, indent, input);
    do_internals(expand, indent, input);
    rw_until(SEMI, expand, indent, FALSE, input);
}

void do_function(expand, indent, input)
int indent, expand, input;
{
    LEX_PTR lex_ret;

    indent = do_white_space(FUNC1, expand, indent, input);
    rw_until(ID, expand, indent, FALSE, input);
    indent = do_white_space(FUNC2, expand, indent, input);
    lex_ret = look_ahead(1, expand, indent);
    if(lex_ret->token != COLON)
    {
        rw_until(LPAREN, expand, indent, FALSE, input);
        do_arg_list(expand, indent, input);
        indent = do_white_space(FUNC3, expand, indent, input);
        rw_until(COLON, expand, indent, FALSE, input);
    }
    else
        rw_until(COLON, expand, indent, FALSE, input);
    do_white_space(FUNC4, expand, indent, input);
    rw_until(ID, expand, indent, FALSE, input);
    do_white_space(FUNC5, expand, indent, input);
    rw_until(SEMI, expand, indent, FALSE, input);
    do_white_space(FUNC6, expand, indent, input);
    do_internals(expand, indent, input);
    rw_until(SEMI, expand, indent, FALSE, input);
}
```

```
main(argc, argv)
int argc;
char *argv[];
{
    int expand = 4, indent = 0, learn = FALSE, count;

    init_lexer();
    for(count = 1; count <= argc; count++)
    {
        if(argv[count][0] == 'L')
            learn = TRUE;
        if(argv[count][0] == '-')
            expand = argv[count][1] - '0';
    }
    if (!learn)
    {
        clean_array(white_stor, FEND);
        read_array(white_stor);
        do_program(expand, indent, learn);
    }
    else
    {
        clean_array(white_stor, FEND);
        do_program(expand, indent, learn);
        store_array(white_stor);
    }
    close_lexer();
    return(0);
}
```

PPBRAIN.C

```
#include <stdio.h>
#include "nlex.h"
#include "whitedef.h"

#define DEBUG2 1

extern void store_array(struct spc *);
extern void init_lexer(void);
extern struct lex_struct *lex(int ,int );
extern struct lex_struct *look_ahead(int ,int ,int );
extern struct lex_struct *new_lex_ptr(void);
extern struct lex_struct *dispose_lex_ptr(struct lex_struct *);
extern char *new_copy(char *);

extern SPACE white_stor[FEND];

/*****************************************************************
 * Given a LEX_PTR that contains a WHITE_SPACE token, output_    *
 * white simply sends the space to the stdout.                   *
 *****************************************************************/
void output_white(white)
LEX_PTR white;
{
    int count;

    for(count = 0; count < white->rets; count++)
        printf("\n");
    for(count = 0; count < white->change; count++)
        printf(" ");
}

/*****************************************************************
 * Takes the token given and outputs it to the screen if the     *
 * input argument is FALSE (== 0).  If it is a whitespace and    *
 * the replace argument is TRUE, and it has one or more returns *
 * associated with it, then the change field of the token is     *
 * replaced with the number passed in indent.                    *
 *****************************************************************/
void output(tokn, indent, replace, input)
LEX_PTR tokn;
int indent, replace, input;
{
    if(!input || DEBUG2)                        /* DEBUG2 TRUE MEANS */
    {                                           /* TO OUTPUT ON LEARN */
        printf("%s", tokn->buffer);
        if(tokn->token == WHITE_SPACE)
        {
            if(replace && tokn->rets)
                tokn->change = indent;
            output_white(tokn);
        }
    }
    tokn = dispose_lex_ptr(tokn);
}

/*******************************************************
 * Clears the array that stores the white space tokens. *
 *******************************************************/
void setup_array(white_stor)
SPACE white_stor[];
{
    int count;

    for(count = 0; count < FEND; count++)
        white_stor[count].change = white_stor[count].rets = 0;
}
```

```
/*****************************************************************
 * Manages the array that stores the white space tokens.  This *
 * allows a change in the data structure that the tokens are   *
 * stored in without affecting any other parts of the program. *
 *****************************************************************/
void white_manage(input, white_loc, value)
int input, white_loc;
LEX_PTR value;
{
    if(input)
    {
        white_stor[white_loc].rets = value->rets;
        white_stor[white_loc].change = value->change;
    }
    else
    {
        value->rets = white_stor[white_loc].rets;
        value->change = white_stor[white_loc].change;
    }
}

/*****************************************************************
 * Reads input tokens directly (and outputs them) until it has  *
 * read and output the token specified in the argument "token". *
 *****************************************************************/
void rw_until(token, expand, indent, replace, input)
int token, expand, indent, replace, input;
{
    LEX_PTR lex_ret;

    lex_ret = lex(expand, indent);
    while(lex_ret->token != token)
    {
        output(lex_ret, indent, replace, input);
        lex_ret = lex(expand, indent);
    }
    output(lex_ret, indent, replace, input);
}

/*****************************************************************
 * Reads tokens until the second token in the queue matches the *
 * token argument passed to it.  Outputs those that fall off    *
 * the front of the queue.                                      *
 *****************************************************************/
void rw_until_look(token, expand, indent, replace, input)
int token, expand, indent, replace, input;
{
    LEX_PTR lex_ret;

    lex_ret = look_ahead(2, expand, indent);
    while(lex_ret->token != token)
    {
        lex_ret = lex(expand, indent);
        output(lex_ret, indent, replace, input);
        lex_ret = look_ahead(2, expand, indent);
    }
}
```

```
/*****************************************************************
 * Reads tokens until the next token in the queue matches the    *
 * token argument passed to it.  Outputs those that fall off     *
 * the front of the queue.                                       *
 *****************************************************************/
void rw_until_look_1(token, expand, indent, replace, input)
int token, expand, indent, replace, input;
{
    LEX_PTR lex_ret;

    lex_ret = look_ahead(1, expand, indent);
    while(lex_ret->token != token)
    {
        lex_ret = lex(expand, indent);
        output(lex_ret, indent, replace, input);
        lex_ret = look_ahead(1, expand, indent);
    }
}

/*****************************************************************
 * If the next token in the input is a WHITE_SPACE it is pulled  *
 * from the input.  Otherwise, a WHITE_SPACE token is created    *
 * and set to 0 rets, 0 change.  The white_manage function is    *
 * then called to make changes to either the token or the        *
 * stored style file, depending on whether we are inputing or    *
 * outputting.  If the indentation changes as a result of reading *
 * or retrieving WHITE_SPACE tokens, then the new indentation is *
 * passed back by the function (and even if there isn't).        *
 *****************************************************************/
int do_white_space(white_loc, expand, indent, input)
int white_loc, expand, indent, input;
{
    LEX_PTR lex_ret;

    lex_ret = look_ahead(1, expand, indent);
    if(lex_ret->token == WHITE_SPACE)
        lex_ret = lex(expand, indent);
    else
    {
        lex_ret = new_lex_ptr();
        lex_ret->token = WHITE_SPACE;
        lex_ret->buffer = new_copy("\0");
        lex_ret->rets = lex_ret->change = 0;
    }
    white_manage(input, white_loc, lex_ret);
    if(lex_ret->rets)
        indent += lex_ret->change;
    output(lex_ret, indent, TRUE, input);
    return(indent);
}
```

FORMFILE.C

```
#include <stdio.h>
#include "whitedef.h"
#include "formfile.def"

extern SPACE white_stor[FEND];

#define FORM_FILE "current.frm"
#define TEST_FILE "test.frm"

/*****************************************************************
 * Given a filename write_stor writes the style sheet in stor to *
 * that file.                                                    *
 *****************************************************************/
int write_stor(filename, stor, size)
char *filename;
SPACE stor[];
int size;
{
    FILE *stream;
    int count;

    stream = fopen(filename, "w");
    for(count = 0; count < size; count++)
    {
        putw(stor[count].rets, stream);
        putw(stor[count].change, stream);
    }
    fclose(stream);
}


/*****************************************************************
 * Given a filename read_stor reads the style sheet in that file *
 * into the array passed as stor.                                *
 *****************************************************************/
int read_stor(filename, stor, size)
char *filename;
SPACE stor[];
int size;
{
    FILE *stream;
    int count;

    stream = fopen(filename, "r");
    for(count = 0; count < size; count++)
    {
        stor[count].rets = getw(stream);
        stor[count].change = getw(stream);
    }
    fclose(stream);
}

void store_array(out)
SPACE out[];
{
    write_stor(FORM_FILE, out, FEND);
}

void read_array(in)
SPACE in[];
{
    read_stor(FORM_FILE, in, FEND);
}
```

```
void clean_array(stor, size)
SPACE stor[];
int size;
{
    int count;

    for(count = 0; count < size; count++)
    {
        stor[count].rets = 0;
        stor[count].change = 0;
    }
}
```

SYMTAB.C

```
#include "nlex.h"
#include "symtab.def"

#define TOKEN_COUNT 23

extern char lexbuf[IDSIZE];
extern int lineno;
extern int line_offs;
extern int token_nm;

static char *keywords[TOKEN_COUNT] =
{
    "ARRAY",
    "BEGIN",
    "CASE",
    "CONST",
    "DO",
    "ELSE",
    "END",
    "FOR",
    "FUNCTION",
    "IF",
    "NOT",
    "OF",
    "PROCEDURE",
    "PROGRAM",
    "RECORD",
    "REPEAT",
    "THEN",
    "TO",
    "TYPE",
    "UNTIL",
    "VAR",
    "WHILE",
    "WITH"
};

enum tokens tokenval[TOKEN_COUNT] =
{
    ARRAYkw,
    BEGINkw,
    CASEkw,
    CONSTkw,
    DOkw,
    ELSEkw,
    ENDkw,
    FORkw,
    FUNCTIONkw,
    IFkw,
    NOTkw,
    OFkw,
    PROCEDUREkw,
    PROGRAMkw,
    RECORDkw,
    REPEATkw,
    THENkw,
    TOkw,
    TYPEkw,
    UNTILkw,
    VARkw,
    WHILEkw,
    WITHkw
};
```

```
/*********************************************************************
 * Uses a simple binary search to determine if an identifier is a  *
 * keyword or not.  Based on the search function described in K&R. *
 *********************************************************************/
int symtab_lookup(identifier)
char *identifier;
{
    int half, low = 0, high = TOKEN_COUNT, compare, count, length;
    char copy[IDSIZE];

    length = strlen(identifier);
    strcpy(copy, identifier);
    for(count = 0; count < length; count++)
        copy[count] = toupper(copy[count]);
    high--;
    while(low <= high)
    {
        half = (low + high) / 2;
        compare = strcmp(copy, keywords[half]);
        if (compare < 0)
            high = half - 1;
        else if (compare > 0)
            low = half + 1;
        else
            return((int) tokenval[half]);
    }
    return(ID);
}
```

NLEX.C

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <malloc.h>
#include "nlex.h"
#include "queue.h"
#include "nlex.def"

#define CMT_BEGIN '{'
#define CMT_END '}'
#define DEC_POINT '.'

#define FALSE 0
#define TRUE 1

#define SINGLE_QUOTE '\047'
#define MAX_LOOK 10

int lineno = 1, token_nm = 0, line_offs = 1;
char lexbuf[IDSIZE];
QUEUE token_q;

extern int symtab_lookup(char *);

int allocated_strings = 0, allocated_LEX_PTRs = 0;
int freed_strings = 0, freed_LEX_PTRs = 0;
LEX_PTR lex_array[MAX_LOOK];
int lex_alloc[MAX_LOOK];

void malloc_stats()
{
    fprintf(stderr, "\nLEX_PTRs:  %d\t allocated   %d\t freed", allocated_LEX_PTRs,
                    freed_LEX_PTRs);
    fprintf(stderr, "\nstrings :  %d\t allocated   %d\t freed", allocated_strings,
                    freed_strings);
    fprintf(stderr, "\nQ Items :  %d\t items in the queue", queue_size(token_q));
}

/*****************************************************************
 * When given a string, new_copy allocates memory and creates a *
 * copy of the string.                                           *
 *****************************************************************/
char *new_copy(str)
char *str;
{
    char *temp;

    temp = (char *) malloc((strlen(lexbuf) + 1) * sizeof(char));
    if(temp == NULL)
    {
        fprintf(stderr, "\n***->Memory Allocation Error in new_copy()");
        malloc_stats();
        exit(1);
    }
    else
    {
        strcpy(temp, str);
        allocated_strings++;
        return(temp);
    }
}
```

```
/****************************************
 * Allocates the storage for one token. *
 ****************************************/
LEX_PTR new_lex_ptr()
{
    LEX_PTR temp;

    temp = (LEX_PTR) malloc(sizeof(LEX_STRUCT));
    if(temp == NULL)
    {
        fprintf(stderr, "\n*** Memory Allocation Error in new_lex_ptr()");
        malloc_stats();
        exit(1);
    }
    else
    {
        temp->buffer = NULL;
        allocated_LEX_PTRs++;
        return(temp);
    }
}

/*****************************************************************
 * Returns the storage associated with one token to the system *
 *****************************************************************/
LEX_PTR dispose_lex_ptr(trash_ptr)
LEX_PTR trash_ptr;
{
    if(trash_ptr->buffer != NULL)
    {
        free(trash_ptr->buffer);
        freed_strings++;
    }
    free(trash_ptr);
    freed_LEX_PTRs++;
    return(NULL);
}

/*****************************************************************
 * Initalizes the lexical analyzer.  Creates a queue that is used *
 * for looking ahead in the input.                                *
 *****************************************************************/
void init_lexer()
{
    token_q = new_queue();
}

/*********************************************************
 * Empties the lexer's queue before a normal termination. *
 *********************************************************/
void close_lexer()
{
    int count;
    LEX_PTR temp;

    count = queue_size(token_q);
    while(count-- > 0)
    {
        temp = dequeue(token_q);
        dispose_lex_ptr(temp);
    }
    token_q = del_queue(token_q);
}
```

```
/*************************************************************
 * Expands tab characters to a series of spaces, given the   *
 * distance between tab stops through the tab_size argument. *
 *************************************************************/
int tab_expand(line_offs, tab_size)
int line_offs, tab_size;
{
    return(((line_offs / tab_size + 1) * tab_size + 1) - line_offs);
}

/*************************************************************
 * The lexical analyzer.  Updates global variables to reflect *
 * the current token in the input queue.                      *
 *************************************************************/
int lexan(tab_size)
int tab_size;
{
    int t, b;

    while(TRUE)
    {
        t = getchar();
        switch (t)
        {
            case '~':
            case EOF:
                return(END_INPUT);
            case ',':
                strcpy(lexbuf, ",");
                return(COMMA);
            case '(':
                strcpy(lexbuf, "(");
                return(LPAREN);
            case ')':
                strcpy(lexbuf, ")");
                return(RPAREN);
            case '[':
                strcpy(lexbuf, "[");
                return(LBRACK);
            case ']':
                strcpy(lexbuf, "]");
                return(RBRACK);
            case ';':
                strcpy(lexbuf, ";");
                return(SEMI);
            case ':':
                lexbuf[0] = t;
                t = getchar();
                if (t == '=')
                {
                    lexbuf[1] = t;
                    lexbuf[2] = '\0';
                    return(ASSIGNOP);
                }
                else
                {
                    ungetc(t, stdin);
                    lexbuf[1] = '\0';
                    return(COLON);
                }
            case '=':
                strcpy(lexbuf, "=");
                return(EQ);
            case '$':
                strcpy(lexbuf, "$");
                return(EQ);                     /* A kludge to get $ in */
            case '#':
                strcpy(lexbuf, "#");
                return(EQ);                     /* Likewise for # */
```

```
case '^':
    strcpy(lexbuf, "^");
    return(EQ);                     /* Likewise for ^ */
case '<':
    lexbuf[0] = t;
    t = getchar();
    if (t == '=')
    {
        strcpy(lexbuf, "<=");
        return(LE);
    }
    else if (t == '>')
    {
        strcpy(lexbuf, "<>");
        return(NE);
    }
    else
    {
        ungetc(t, stdin);
        strcpy(lexbuf, "<");
        return(LT);
    }
case '>':
    lexbuf[0] = t;
    t = getchar();
    if (t == '=')
    {
        strcpy(lexbuf, ">=");
        return(GE);
    }
    else if (t == '<')
    {
        strcpy(lexbuf, "<>");
        return(NE);
    }
    else
    {
        ungetc(t, stdin);
        strcpy(lexbuf, ">");
        return(GT);
    }
case '.':
    strcpy(lexbuf, ".");
    return(PERIOD);
case '*':
case '/':
    lexbuf[0] = t;
    lexbuf[1] = '\0';
    return(MULOP);
case '+':
case '-':
    lexbuf[0] = t;
    lexbuf[1] = '\0';
    return(ADDOP);
case CMT_BEGIN:
case ' ':
case '\t':
case '\n': return(get_white_space(t, tab_size));
           break;
case SINGLE_QUOTE:                  /* Single Quote */
    {
        ungetc(t, stdin);
        return(get_string());
    }
default:
    if(isdigit(t))
    {
        ungetc(t, stdin);
        return(get_number());
    }
```

```
                else if(isalpha(t) || (t == '_'))
                {
                    ungetc(t, stdin);
                    return(get_id());
                }
            }
        }
    }

/*********************************************************************
 * Returns the change in indentation and number of carriage returns  *
 * in a WHITE_SPACE token when given the white space string.  Updates *
 * the current indentation level.  All information returned in the    *
 * "result" argument.                                                 *
 *********************************************************************/
void calc_space(result, indent)
LEX_PTR result;
int indent;
{
    int ctr, length, count = 0, crs = 0, spot = 0;

    length = strlen(result->buffer);        /* Determine string length */
    if(length > 0)
    {
        ctr = length - 1;
        while((result->buffer[ctr] != CMT_END) && (ctr >= 0))
            ctr--;                          /* Find beginning or CMT_END */
        spot = ctr + 1;                     /* But we go one too far back */
        for(ctr = spot; ctr < length; ctr++)    /* Loop to count CRs and SPC */
        {
            if(result->buffer[ctr] != '\n')     /* Add to space count? */
                count++;
            else                            /* No, add to CR count. */
            {
                count = 0;
                crs++;
            }
        }
    }
    result->buffer[spot] = '\0';            /* Remove tokenized whitespace */
    result->rets = crs;                     /* Return # of CRs and */
    if(crs > 0)
        result->change = count - indent;    /* change in indentation */
    else
        result->change = count;             /* just spaces here */
}

/*********************************************************************
 * If previous look_aheads have filled the queue, then the first item *
 * in the queue is returned.  Otherwise it calls lexan to get the     *
 * next token.                                                        *
 *********************************************************************/
LEX_PTR lex(expand, indent)
int expand, indent;
{
    LEX_PTR ret_val;

    if(queue_size(token_q) > 0)
        ret_val = dequeue(token_q);
    else
    {
        ret_val = new_lex_ptr();
        ret_val->token = lexan(expand);
        ret_val->buffer = new_copy(lexbuf);
    }
    if(ret_val->token == WHITE_SPACE)
        calc_space(ret_val, indent);
    return(ret_val);
}
```

```
/*************************************************************************
 * Looks ahead in the input n tokens.  If there are already n tokens in *
 * the queue, then the nth one is returned.  Otherwise, lex is called   *
 * to get the appropriate number of tokens from the input stream.       *
 *************************************************************************/
LEX_PTR look_ahead(n, expand, indent)
int n, expand, indent;
{
    int count, look_for, new_token;
    LEX_PTR temp;

    if(n > queue_size(token_q))
    {
        look_for = n - queue_size(token_q);
        while((look_for-- > 0) && ((new_token = lexan(expand)) != END_INPUT))
        {
            temp = new_lex_ptr();
            temp->token = new_token;
            temp->buffer= new_copy(lexbuf);
            enqueue(temp, token_q);
        }
    }
    return(queue_item(n-1, token_q));
}

/*********************************************************
 * Collects whitespace and comments into a single token. *
 *********************************************************/
int get_white_space(t, tab_size)
int t, tab_size;
{
    int b, temp, count;

    b = 0;
    do
    {
        lexbuf[b++] = t;
        if(t == '\t')
        {
            temp = tab_expand(line_offs, tab_size);
            b--;
            for(count = 0; count < temp; count++)
                lexbuf[b++] = ' ';
            line_offs += temp;
        }
        if(t == '\n')
        {
            lineno++;
            line_offs = 1;
        }
        if(t == CMT_BEGIN)
            b = do_comment(t, b);
    }
    while(isspace(t = getchar()) || (t == CMT_BEGIN));
    ungetc(t, stdin);
    lexbuf[b] = '\0';
    return(WHITE_SPACE);
}
```

```
/************************************************
 * Inserts a comment into the current whitespace. *
 ************************************************/
int do_comment(t, count)
int t, count;
{
    while((t = getchar()) != CMT_END)
    {
        lexbuf[count++] = t;
        line_offs++;
        if (t == '\n')
        {
            lineno++;
            line_offs = 1;
        }
    }
    lexbuf[count++] = t;
    return(count);
}

/***************************************
 * Used by lexan to get a Pascal string. *
 ***************************************/
int get_string()
{
    int t, b = 0, done = 0;

    lexbuf[b++] = getchar();
    while (!done)
    {
        t = getchar();
        if (t == SINGLE_QUOTE)
        {
            t = getchar();
            if (t == SINGLE_QUOTE)
                lexbuf[b++] = t;
            else
            {
                 ungetc(t, stdin);
                 done = 1;
            }
        }
        else
            lexbuf[b++] = t;
    }
    lexbuf[b++] = SINGLE_QUOTE;
    lexbuf[b] = '\0';
    return(STRING);
}
```

```
/****************************************
 * Used by lexan to get numeric tokens. *
 ****************************************/
int get_number()
{
    int t, b = 0;

    t = getchar();
    lexbuf[b++] = t;
    while (isdigit(t = getchar()))
        lexbuf[b++] = t;
    if(t == DEC_POINT)
    {
        lexbuf[b++] = t;
        while (isdigit(t = getchar()))
            lexbuf[b++] = t;
        if('E' == toupper(t))
        {
            lexbuf[b++] = t;
            t = getchar();
            if ((t == '+') || (t == '-'))
                lexbuf[b++] = t;
            else
                ungetc(t, stdin);
            while(isdigit(t = getchar()))
                lexbuf[b++] = t;
            ungetc(t, stdin);
        }
        else
            ungetc(t, stdin);
        lexbuf[b] = '\0';
        return(REAL);
    }
    else
        ungetc(t, stdin);
    lexbuf[b] = '\0';
    return(INTEGER);
}

/*****************************************
 * Used by lexan to get identifier tokens. *
 *****************************************/
int get_id()
{
    int t, b = 0;

    t = getchar();
    while (isalnum(t) || (t == '_'))
    {
        lexbuf[b++] = t;
        t = getchar();
    }
    ungetc(t, stdin);
    lexbuf[b] = '\0';
    return(symtab_lookup(lexbuf));
}
```

QUEUE.C

```
#include <stdio.h>
#include <malloc.h>
#include "queue.def"

struct lnode {
    char *data;         /* Points to item in the list */
    struct lnode *next; /* Points to the next item */
};

typedef struct q_def {
    struct lnode *rem;      /* Points to the removal end of the queue */
    struct lnode *add;      /* Points to the addition end of the queue */
    int count;              /* Total # of items now in the array */
    struct lnode *last_ptr; /* Points to the last item accessed w/o removing */
    int last_count;         /* Rank (starts at 0) of last item accessed */
} QUEUE, *QUEUE_PTR;

struct lnode *l_free(item)
struct lnode *item;
{
    free(item);
    return(NULL);
}

struct lnode *l_alloc(item)
char *item;
{
    struct lnode *new_l;

    new_l = (struct lnode *) malloc(sizeof(struct lnode));
    if (new_l != NULL)
    {
        new_l->data = item;
        new_l->next = NULL;
    }
    return(new_l);
}

/*********************************************************************/
/* Returns a pointer to an initialized new queue.  The queue is      */
/* initially empty, since both the head and tail pointers are        */
/* set to NULL, and the count is set to 0.  The interesting thing    */
/* about designing the queue in this manner, is that you have one    */
/* set of functions which can operate on multiple queues.  Also,     */
/* by using casts in the proper places, these functions will work    */
/* on any data type!                                                 */
/*********************************************************************/
QUEUE_PTR new_queue()
{
    QUEUE_PTR new_q;

    new_q = (QUEUE_PTR) malloc(sizeof(QUEUE));
    if (new_q != NULL)
    {
        new_q->add = new_q->rem = new_q->last_ptr = NULL;
        new_q->count = new_q->last_count = 0;
    }
    return(new_q);
}
```

```
/*********************************************************/
/* Returns the memory taken up by a queue to the system. */
/* Note that it even works correctly if the queue isn't  */
/* yet empty by moving down the item list and freeing it.*/
/*********************************************************/
QUEUE_PTR del_queue(que)
QUEUE_PTR que;
{
    struct lnode *next, *crnt;
    int count;

    for (count = 1, crnt = que->rem; count <= que->count; count++)
    {
        next = crnt->next;
        free(crnt->data);
        crnt = l_free(crnt);
        crnt = next;
    }
    free(que);
    return(NULL);
}

/*********************************************************************/
/* Adds the val pointer to the queue.  If the count == 0, we know    */
/* that we must write over the add pointer, and set the rem pointer  */
/* to also point to the item just added.  In this way, we don't      */
/* have to compare raw pointer values to determine when we reach     */
/* end of the data stored in the queue.                              */
/*********************************************************************/
void enqueue(val, que)
char *val;
QUEUE_PTR que;
{
    struct lnode *new;

    new = l_alloc(val);
    if (new != NULL)
    {
        if (que->count > 0)
            que->add->next = new;
        else
            que->rem = new;
        que->add = new;
        que->count++;
    }
}

/*********************************************************/
/* Returns a pointer to a character, the first one to have */
/* entered the queue after first checking count to make    */
/* certain that there are actually values in the queue.    */
/*********************************************************/
char *dequeue(que)
QUEUE_PTR que;
{
    struct lnode *junk;
    char *item;

    if (que->count > 0)
    {
        junk = que->rem;
        item = junk->data;
        que->rem = que->rem->next; /* if that was last, then it is */
        que->count--;              /* set to NULL automatically */
        junk = l_free(junk);
        return(item);
    }
    else
        return(NULL);
}
```

```
/*********************************************************/
/* Returns a pointer to the nth member of the queue list.  */
/*  NOTE: Numbering of the items begins at 0.              */
/*********************************************************/
struct lnode *queue_member(n, que)
int n;
QUEUE_PTR que;
{
    struct lnode *crnt;
    int count;

    crnt = que->rem;
    if(n == 0)
        return(crnt);
    else
    {
        for(count = 1; count <= n; count++)
        {
            crnt = crnt->next;
            if(crnt == NULL)
                return(NULL);
        }
        return(crnt);
    }
}


/*************************************************/
/* Returns a pointer to the nth item in the queue. */
/*  NOTE: Numbering of the items begins at 0.      */
/*************************************************/
char *queue_item(n, que)
int n;
QUEUE_PTR que;
{
    struct lnode *current;

    current = queue_member(n, que);
    return(current->data);
}
```

```
/*******************************************************/
/* Deletes the nth item in the queue, and returns its */
/* associated storage blocks to the system.           */
/*******************************************************/
void del_item(n, que)
QUEUE_PTR que;
int n;
{
    struct lnode *target, *pred;

    if(n < que->count)
    {
        if(n > 0)
        {
            pred = queue_member(n-1, que);
            target = pred->next;
            pred->next = target->next;
            que->count--;
            free(target->data);
            target = l_free(target);
        }
        else if(n = 0)
        {
            target = que->rem;
            que->rem = target->next;
            que->count--;
            free(target->data);
            target = l_free(target);
        }
    }
}

/*******************************************************/
/* Returns the current size of the queue passed to it. */
/*******************************************************/
int queue_size(que)
QUEUE_PTR que;
{
    return(que->count);
}
```

NLEX.H

```
#define IDSIZE 5000
#define TRUE 1
#define FALSE 0
#define STOR 1
#define RETR 0


enum tokens
{
    MULOP,
    RELOP,
    ADDOP,
    ASSIGNOP,
    ARRAYkw,
    BEGINkw,
    BOOLEANkw,
    CASEkw,
    CONSTkw,
    DOkw,
    ELSEkw,
    ENDkw,
    FORkw,
    FUNCTIONkw,
    IFkw,
    INTEGERkw,
    NOTkw,
    OFkw,
    PROCEDUREkw,
    PROGRAMkw,
    REALkw,
    RECORDkw,
    REPEATkw,
    THENkw,
    TOkw,
    TYPEkw,
    UNTILkw,
    VARkw,
    WHILEkw,
    WITHkw,
    PERIOD,
    COMMA,
    LPAREN,
    RPAREN,
    COLON,
    SEMI,
    LBRACK,
    RBRACK,
    ID,
    INTEGER,
    REAL,
    PLUS,
    MINUS,
    OR,
    UNKNOWN,
    END_INPUT,
    EQ,
    NE,
    LT,
    LE,
    GT,
    GE,
    STRING,
    WHITE_SPACE
};
```

```
typedef struct lex_struct
{
    int token;              /* Holds the token's ID number */
    char *buffer;           /* Holds the ASCII rep. of the token */
    int rets;               /* If WHITE_SPACE, it holds the # of <cr>s */
    int change;             /* Holds the change in indentation (or spaces) */
} LEX_STRUCT, *LEX_PTR;
```

WHITEDEF.H

```
typedef struct spc
{
    int rets;       /* # of <CR>s in the white space string */
    int change;     /* Total # of spaces to add (+) or subtract (-) */
} SPACE, *SPACE_PTR;

enum space_names
{
    PROG1,
    PROG2,
    PROG3,
    PROG4,
    BEGIN1,
    BEGIN2,
    PROC1,
    PROC2,
    PROC3,
    PROC4,
    FUNC1,
    FUNC2,
    FUNC3,
    FUNC4,
    FUNC5,
    FUNC6,
    WHILE1,
    WHILE2,
    WHILE3,
    WHILE4,
    REPEAT1,
    REPEAT2,
    REPEAT3,
    IF1,
    IF2,
    IF3,
    IF4,
    IF5,
    IF6,
    ELSE1,
    ELSE2,
    ELSE3,
    WITH1,
    WITH2,
    WITH3,
    WITH4,
    FOR1,
    FOR2,
    FOR3,
    FOR4,
    FOR5,
    FOR6,
    CASE1,
    CASE2,
    CASE3,
    CASE4,
    CASE5,
    CASE6,
    FEND
} dummy;
```

QUEUE.H

```
typedef char *QUEUE;

extern QUEUE new_queue();
extern QUEUE del_queue(char *);
extern void enqueue(struct lex_struct *, char *);
extern LEX_PTR dequeue(char *);
extern LEX_PTR queue_item(int, char *);
extern int queue_size(char *);
```

IPP.DEF

```
/*global*/  void do_arg_list(int ,int ,int );
/*global*/  void do_statement(int ,int ,int );
/*global*/  void do_statement_series(int ,int ,int );
/*global*/  void do_begin_end(int ,int ,int );
/*global*/  void do_sub_clause(int ,int ,int ,int ,int );
/*global*/  void do_internals(int ,int ,int );
/*global*/  void do_program(int ,int ,int );
/*global*/  void do_for(int ,int ,int );
/*global*/  void do_while(int ,int ,int );
/*global*/  void do_if(int ,int ,int );
/*global*/  void do_else(int ,int ,int );
/*global*/  void do_repeat(int ,int ,int );
/*global*/  void do_with(int ,int ,int );
/*global*/  void do_procedure(int ,int ,int );
/*global*/  void do_function(int ,int ,int );
/*global*/  int main(int ,char * *);
```

PPBRAIN.DEF

```
/*global*/  void output_white(struct lex_struct *);
/*global*/  void output(struct lex_struct *,int ,int ,int );
/*global*/  void setup_array(struct spc *);
/*global*/  void white_manage(int ,int ,struct lex_struct *);
/*global*/  void rw_until(int ,int ,int ,int ,int );
/*global*/  void rw_until_look(int ,int ,int ,int ,int );
/*global*/  void rw_until_look_1(int ,int ,int ,int ,int );
/*global*/  int do_white_space(int ,int ,int ,int );
```

FORMFILE.DEF

```
/*global*/  int write_stor(char *,struct spc *,int );
/*global*/  int read_stor(char *,struct spc *,int );
/*global*/  void store_array(struct spc *);
/*global*/  void read_array(struct spc *);
/*global*/  void fill_array(struct spc *,int );
/*global*/  void clean_array(struct spc *,int );
/*global*/  void test_array(struct spc *);
```

SYMTAB.DEF

```
/*global*/  int symtab_lookup(char *);
```

NLEX.DEF

```
/*global*/  void malloc_stats(void);
/*global*/  char *new_copy(char *);
/*global*/  struct lex_struct *new_lex_ptr(void);
/*global*/  struct lex_struct *dispose_lex_ptr(struct lex_struct *);
/*global*/  void init_lexer(void);
/*global*/  void close_lexer(void);
/*global*/  int tab_expand(int ,int );
/*global*/  int lexan(int );
/*global*/  void calc_space(struct lex_struct *,int );
/*global*/  struct lex_struct *lex(int ,int );
/*global*/  struct lex_struct *look_ahead(int ,int ,int );
/*global*/  int get_string(void);
/*global*/  int get_number(void);
/*global*/  int get_id(void);
```