A Prototype Intelligent Prettyprinter for Pascal

Kirt A. Winter
Hewlett Packard
Electronic Design Division
P.O. Box 617
MS 1U05L
Colorado Springs, CO 80901-0617
(winter@hpldola.hp.com)

Curtis R. Cook
Computer Science Department
Oregon State University
Corvallis, OR 97331-390
(cook@mist.cs.orst.edu)

Introduction

A prettyprinter is a program that formats a source code listing to illuminate its structure and thereby improve its readability. A prettyprinter replaces white spaces (new lines, returns, tabs, spaces) in the program using formatting instructions that specify what layout the source code should have. Prettyprinters are a common software tool and often there are several prettyprinters available for most widely used programming languages. For most, if not all languages, there is no commonly accepted program source code format or layout. The little programming style research on the effect of formatting factors (primarily indentation) in comprehension has been inconclusive, but seems to suggest that there is no best source program layout style. In addition most programmers have developed (or learned) a formatting style that they prefer. Hence there are a wide variety of formatting styles, with little evidence to suggest that a "best" style exists. Despite this, prettyprinters up to this point have tended to reflect the formatting preferences of the prettyprinter's designer.

In this article we describe IPP, a prototype flexible, intelligent prettyprinter for Pascal that "learns" the formatting style preferred user by examining a sample of source code formatted in the preferred style. We will describe the motivation, design, implementation, capabilities and limitations of IPP. Even though IPP was designed for Pascal, its principles and concepts apply to flexible prettyprinters for other programming languages (though not necessarily all programming languages).

Prettyprinters and Formatting Style

As mentioned in the Introduction, prettyprinting is illuminating the logical structure of the source to improve its readability [LEDG75]. There have been many articles which describe various prettyprinters and/or formatting styles for Pascal { [BATE81], [CRID78], [GROG79], [MARC81], [PETE77]}. In this paper we will concentrate on prettyprinting issues as they relate to Pascal, although most of these issues apply to other programming languages as well.

In changing the spacing of a program, a prettyprinter needs much of the functionality of a language compiler. The prettyprinter must recognize keywords, language constructs, and white spaces (new lines, returns, tabs, and spaces) that appear between them. It replaces the white spaces in the program using the formatting instructions for that construct and hence transforms the format of the source code into the style known by the prettyprinter. There are numerous prettyprinters available for most programming languages, each with a different formatting convention that the implementor believes best displays the logical structure of the program.

The goal of formatting style is to improve the readability of the source code. Formatting style rules for Pascal (and most programming languages) vary greatly in textbooks and articles. And while textbooks and articles agree that a consistent formatting style is important, they do not carefully define what they mean by consistency. For instance, does consistency exclude using an indentation pattern for the compound statement in the WHILE statement that is slightly different from the one used for the THEN and ELSE parts of a conditional statement?

Enforcement of formatting consistency seems to be the major function of most prettyprinters. The "consistent" set of style rules are generally the ones selected by the author of the prettyprinter. As might be expected the formatting styles range from the expected to the slightly bizarre. No experimental or empirical data supporting the choice of a particular set of formatting style rules is mentioned.

If we assume that there is no difference in the effort required to produce any particular formatting style (by using a prettyprinter), then one could argue that comprehension should be the deciding factor in choosing a formatting style. However, at this point in time, the contribution of formatting style to source code comprehension is only partially understood. There is evidence that source layout has a larger than expected impact on program comprehension and the ease of performing maintenance tasks [OMAN89]. Oman and Cook [OMAN89] present principles for program layout and commenting and an implementation of the principles that significantly improves program comprehension.

It would seem highly unlikely that there is a "best" formatting style for comprehension in the sense that a majority programmers 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 of programmers will have different preferences. Hence format style seems highly individualistic.

All of this suggests the need for a prettyprinter with the flexibility to allow the users to specify the formatting style they prefer. It would provide all the consistency advantages of traditional prettyprinters while allowing users to specify their own formatting style. Current prettyprinters are inflexible or have very limited flexibility in that they allow the user to choose a few options such the number of spaces on indentation. There are two notable exceptions. One is a LISP prettyprinter that allows the user to control the appearance of the output by user specified "deformat" functions that provide templates for various types of control structures and functions [WATE83]. But writing "deformat" functions could hardly be described as a trivial task, at least for a new user. The second is a formatter for Logitech's Modula-2 that learns a format style from a template file [LOGI87]. The user edits the syntactical constructs in the template file to reflect his or her style preferences. The prettyprinter extracts the formatting data from the template file.

IPP (Intelligent PrettyPrinter)

IPP achieves both consistency and flexibility. Early in the design of IPP it was determined that the white space tokens that lie between keywords, symbols, or language constructions would be used to capture the formatting style. By capturing the rules for these white space tokens, a prettyprinter can duplicate a wide variety of formatting styles. A quick count showed that at least 40 of these tokens would be required to capture just the control structures of Pascal.

We considered using menu systems or a simple ASCII file for the user to specify these rules. Both of these appeared overly tedious and burdensome for the user. It seemed, from a programmer's standpoint, that the best way to describe a preferred source code format style was to show an example of it to another person. Thus it was decided to have IPP learn the user's preferred style conventions from a sample of code provided by the user. We made this decision and completed IPP before we discovered a similar scheme was used in Logitech's Modula-2 prettyprinter [LOGI87]. However, the Logitech prettyprinter requires the use of a special template file and not any sample of code.

In parsing a source program for output, a prettyprinter recognizes the spaces between keywords and control structures (white space tokens) and replaces them using formatting instructions. By extending the prettyprinter so that it not only recognizes, but analyzes and saves white space tokens in another file, one can create a prettyprinter that "learns" a style from a code sample provided to it. This is the method used to convey a particular formatting style to IPP. The code sample can be any program with the desired formatting style, either provided by the user in the form of code that was written by the user or another programmer or by modifying the special program supplied with IPP that contains instances of all Pascal white space token rules recognized by IPP.

Figures 1-3 illustrate how IPP works. IPP learns the style from the small program in Figure 1 using the default settings for tab expansion (four spaces) and applies what it learned on the program in Figure 2 to produce the reformatted version in Figure 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 new lines have to be considered), the tab expansion size is left as a command line option, with a default of four spaces.

What did IPP learn from the code in Figure 1? First IPP learned that the name of the program should follow the keyword PROGRAM with one space between them. Other things 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 discussed later in the section on Limitations.

Design and Implementation of IPP

IPP is targeted for the Turbo Pascal dialect of the Pascal language and thus supports standard Pascal. Currently it recognizes 48 white space tokens (each composed of two integers) to capture formatting style. IPP does not analyze data declarations, nor does it analyze anything below the statement level.

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, the command line

```
ipp L <style1.pas -3
```

causes IPP to learn ("L") the formatting style in the file "style1.pas" with tab stops set at every 3 spaces. After executing this command IPP is now ready to apply the style it learned from "style1.pas". Then the command to format the file "messy.pas" according to this style and create a formatted file called "neat.pas" is

```
ipp <messy.pas >neat.pas
```

In the learn mode, IPP is given a source file to analyze in order to imitate its format. IPP identifies and analyzes white space 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 white space 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.

The white space in a source program file is (in freely formatable languages) completely ignored by the compiler. Any number of white space characters can be used between keywords and symbols as the white space only serves as a delimiter between tokens of the language. However, the way in which one uses white spaces can have a great effect on the visual layout of the source code. For instance indentation is used to indicate nested structures and levels of nesting.

For IPP, a white space token consists of two integer values. One is for the number of new line characters, and the other for the change of indentation from the previous line. In learn mode IPP analyzes the white spaces in the construct and sets the appropriate integer values in the style sheet. In format mode IPP simply finds and replaces white space tokens in a construct with one previously learned for that construct. For other instances of white space, the white space is either placed directly in the output, or modified to line up with the beginning of the previous line, depending on the situation.

IPP consists of six separately compilable C modules (totaling approximately 45K bytes) and was developed on an MS-DOS system using Microsoft C 4.0.

1. NLEX.C: Performs basic lexical analysis of the source code. In addition to the operations performed by most lexical analyzers, NLEX returns white space tokens that are used and/or replaced by the PBRAIN.C module.

2. SYMTAB.C: Uses a binary search to identify keywords in identifiers returned by NLEX.

3. 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. QUEUE.C: A reusable module developed to implement the queue which is used to hold tokens that were retrieved from the NLEX in look-ahead operations.

5. PBRAIN.C: Contains the function 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 descent parser for learn and output is defined here.

6. IPP.C: Contains the main program and the recursive descent 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 white space tokens are recognized.

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

Limitations

Since IPP is a prototype designed primarily to test the feasibility of a prettyprinter learning formatting style, it has several limitations.

1. It is certainly possible to look for more instances of white space than IPP does. In fact, for a truly usable prettyprinter, one would probably consider adding many more. For example, IPP does not analyze data declaration or analyze anything below the statement level. This means that arithmetic expressions, function or procedure calls, Boolean expressions, and several other features that could be formatted are not. However, it should be clear that the concepts used to capture other control structures' layouts could easily be extended to these cases as well.

2. IPP assumes statements enclosed by BEGIN-END pairs all have the same indentation.

3. IPP ignores any physical line length.

4. Since all white space tokens are stored in relative terms as the change in indentation from the previous line, IPP does not preserve absolutely aligned constructs such as in-line comments. In-line comments may be aligned at the beginning of a line or block of compound statements or at the end of a line. IPP must decide whether to use the white space before or after the comment block, or perhaps some combination of the two. In addition for lines containing a statement with a comment aligned at the end of the line, the number of spaces between the end of the statement and the comment may vary. IPP uses the white space after the comment. However, it should be noted that dealing with comments is one of the hardest problems in writing a prettyprinter [RUBI83].

5. Error messages are not provided, and errors may cause unpredictable results. IPP does not check the code sample it learns from for consistency. Also IPP assumes a syntactically correct source program.

Conclusions

IPP is a flexible prettyprinter that can learn a wide variety of formatting styles. It solves the problem of the volume of information to be given to specify a formatting style by learning the preferred formatting style from a sample piece of code.

Because of its flexibility IPP accommodates both a company formatting standard and individual preferences. Thus it possible for members of a development team, who find that the company's style standard causes them difficulty in understanding or maintaining code, to format source code they work with into a style they prefer and then convert it back to the company standard when they are finished. IPP could also be used to easily develop experimental materials for studies of the effects of formatting style on comprehension.

For those interested, IPP is available in MS-DOS executable form for a $15 handling fee. It will be provided on a 5.25" media formatted to 360K by MS-DOS 3.0. Included will be the executable version of the program, a brief "user's manual" file, and the previous mentioned program which contains all white space tokens that IPP recognizes, in several "stylish" forms. The version of IPP supplied was created to handle the Turbo Pascal 3.0 language (no separately compiled UNITS). Requests should be mailed to Kirt Winter, 2340 Rossmere, Colorado Springs, CO 80919.

References

[BATE81] Bates, R.M., A Pascal Prettyprinter With a Different Purpose. ACM SIGPLAN Notices, Vol. 16, No. 3, Mar. 1981, pp. 10- 17.

[CRID78] Crider, J.E., Structured Formatting of Pascal Program. ACM SIGPLAN Notices, Vol. 13 No. 11, Nov. 1978, pp. 15-22.

[GROG79] Grogono, P., On Layout, Identifiers and Semicolons in Pascal Programs. ACM SIGPLAN Notices Vol. 14 No. 4, Apr. 1979, pp 35-40.

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

[LOGI87] Logitech Modula-2 manual, Logitech Inc., Fremont, California, 1987, pp. 17-19.

[MARC81] Marca, D. Some Pascal Style Guidelines. ACM SIGPLAN Notices Vol. 16, No. 4, Apr. 1981, pp. 70-80.

[OMAN89] Oman, P. and Cook, C.R., Typographic Style is More than Cosmetic, Oregon State University Computer Science Technical Report 89-60-5 (Submitted for publication).

[PETE77] Peterson, J.L., On the Formatting of Pascal Programs. ACM SIGPLAN Notices Vol. 12, No. 12, pp. 83-86.

[RUBI83] Rubin, L., Syntax-directed Pretty Printing - A First Step Towards a Syntax-Directed Editor, IEEE Transactions on Software Engineering, Vol. SE-9, No. 2, Mar. 1983, pp 119-127.

[WATE83] Waters, R.C., User Format Control in a LISP Prettyprinter. ACM Transactions on Programming Languages and Systems, Vol 5, No. 4.

```
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 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_with_baby
                               end
end.
```

Figure 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_with_baby
    end
end.
```

Figure 3: Result of applying IPP to Figure 2.