The PretzelBook

second edition

Felix Gärtner

June 11, 1998

Contents

1 Introduction **5**
- 1.1 Do Prettyprinting the Pretzel Way . . . 5
- 1.2 History . . . 6
- 1.3 Acknowledgements . . . 6
- 1.4 Changes to second Edition . . . 6

2 Using Pretzel **7**
- 2.1 Getting Started . . . 7
  - 2.1.1 A first Example . . . 7
  - 2.1.2 Running Pretzel . . . 9
  - 2.1.3 Using Pretzel Output . . . 9
- 2.2 Carrying On . . . 10
  - 2.2.1 The Two Input Files . . . 10
  - 2.2.2 Formatted Tokens . . . 10
  - 2.2.3 Regular Expressions . . . 10
  - 2.2.4 Formatted Grammar . . . 11
  - 2.2.5 Prettyprinting with Format Instructions . . . 12
  - 2.2.6 Formatting Instructions . . . 13
- 2.3 Writing Prettyprinting Grammars . . . 16
  - 2.3.1 Modifying an existing grammar . . . 17
  - 2.3.2 Writing a new Grammar from Scratch . . . 17
  - 2.3.3 Context Free versus Context Sensitive . . . 18
  - 2.3.4 Available Grammars . . . 19
  - 2.3.5 Debugging Prettyprinting Grammars . . . 19
  - 2.3.6 Experiences . . . 21

3 Pretzel Hacking **23**
- 3.1 Adding C Code to the Rules . . . 23
  - 3.1.1 Example for Tokens . . . 23
  - 3.1.2 Example for Grammars . . . 24
  - 3.1.3 Summary . . . 25
  - 3.1.4 Tips and Tricks . . . 26
- 3.2 The Pretzel Interface . . . 26
  - 3.2.1 The Prettyprinting Scanner . . . 27
  - 3.2.2 The Prettyprinting Parser . . . 28
  - 3.2.3 Example . . . 29
- 3.3 Building a Pretzel prettyprinter by Hand . . . 30
- 3.4 Obtaining a Pretzel Prettyprinting Module . . . 30
  - 3.4.1 The Prettyprinting Scanner . . . 30
  - 3.4.2 The Prettyprinting Parser . . . 31
- 3.5 Multiple Pretzel Modules in the same Program . . . 31
- 3.6 Prettyprinting for non-LaTeXians . . . 32

3.6.1 Other Markup Formatters 32
3.6.2 Going for HTML 33

4 Pretzel meets `noweb` 35
4.1 Prettyprinting in `noweb` – How it works 35
4.1.1 A `noweb` Prettyprinter for C 36
4.1.2 A `noweb` Prettyprinter for Java 36
4.1.3 Writing Prettyprinting Grammars for `noweb` 37
4.1.4 Debugging 38
4.1.5 Making the Best Use of It 38
4.1.6 Some Naming Conventions 38
4.2 Problems 39

5 On Prettyprinting 41
5.1 Prettyprinting with Format Commands 41
5.2 A Short History of Prettyprinting 42
5.2.1 Historical Notes 42
5.2.2 The Language Dependent Front End 44
5.2.3 The Language Independent Back End 45
5.2.4 The Set of Format Commands 45
5.2.5 Open Prettyprinting Problems 47

6 Future Work 49

7 Reference 51
7.1 The Concept of Pretzel 51
7.1.1 The Input Files 52
7.2 The Format of the Input Files 52
7.2.1 The Formatted Token File 53
7.2.2 The Formatted Grammar File 54
7.2.3 Comments and Code 56
7.3 Synopsis of `pretzel` and `pretzel-it` 56
7.3.1 `pretzel-it` 56
7.3.2 `pretzel` 56

Bibliography 57

Index 62

Chapter 1

Introduction

> "I do think however that it is important that you are given some control over the basic prettyprinting style, as it can get very frustrating to see your code systematically reformatted in a style that you dislike."
>
> Marc van Leeuwen [63]

1.1 Do Prettyprinting the Pretzel Way

What is Pretzel?

Welcome to Pretzel. Pretzel is a prettyprinter generator, i.e. a system hat builds prettyprinters. The good thing about Pretzel is that it builds prettyprinters with full user control. The new thing about Pretzel is that it doesn't build standalone prettyprinting programs, but rather builds modules that can be turned into standalone programs easiliy, but can also be incorporated into your own software systems.

What is this book about?

This book is the ultimate source of information about Pretzel. It explains nearly all (known and unknown) facets of the Pretzel system and should be usefull to anybody using the system.

How should I read this book?

First of all, I expect that you are aquainted with the issue of prettyprinting. Why would you use a system like Pretzel if you weren't?! (However, if you want to freshen up your understanding of the subject matter, you should have a glance at chapter 5 at some pleasing time, which gives an overview over the area of prettyprinting and the history of the concepts behind it.)

Essentially, you should start with the first chapter, that is the tutorial of Pretzel. This might be the only part of this document that most people will read. Later chapters deal with the more intricate parts of using Pretzel (like chapter 3, "Pretzel hacking") or with combining Pretzel with the popular programming tool `noweb` (chapter 4, "Pretzel meets `noweb`").

The final chapter is the reference manual of Pretzel. Consult this section only in emergencies.

Where can I get Pretzel?

You can probably get Pretzel through the site or person through which you got this document. Pretzel is free software, i.e. you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation (either version 2 of the License, or (at your option) any later version).

You can get the latest version of the distribution by looking at the following WWW address, which points to the Pretzel homepage:

`http://www.iti.informatik.th-darmstadt.de/~gaertner/pretzel/`

The latest distribution resides at:

`http://www.iti.informatik.th-darmstadt.de/~gaertner/pretzel/code/`

The current release number of Pretzel is 2.0. Look out for the latest information on the Pretzel homepage, if you want to be up to date.

How do I install Pretzel?

The distribution contains a file called `README` that contains further instructions how to install Pretzel. To date, Pretzel runs on UNIXisch like machines and has been tested under HP-UX, AIX RS6000 platforms and under Linux. It uses the GNU `g++` compiler and the common UNIX tools `flex` and `Bison`.

1.2 History

The Pretzel system started off as a minor project at the Institut für Theoretische Informatik at the Technical University of Darmstadt, Germany early 1993. It was revised during a sabbatical at Trinity College, Dublin in late 1993 and version 1.1 was finished back in Darmstadt in the summer of 1994. In early 1996 encouragement from the net led me to take up the project again and prepare release 2.0, which main feature was the ability to interface with `noweb`. This release was out by christmas 1996. Work continues mostly on building new prettyprinting grammars for several languages.

1.3 Acknowledgements

Pretzel is by far no finished system and a lot still needs to be done. However I have to thank firstly Joachim Schrod for initiating this project and his encouragement to hang on to it for so long. I also have to thank Lee Wittenberg and Norman Ramsey for their help on release 2.0 and to Holger Uhr for pointing me to a bug. Thanks also to Roger Kehr and all those other students whom I have discussed this topic with (whether they wanted or not). I am very grateful to the Institut für Theoretische Informatik here in Darmstadt and to Professor Waldschmidt for providing the computing resources for this project. Release 2.0 comes as a christmas present for me and the people on the USENET `comp.programming.literate` newsgroup, who — with their discussions — have helped to improve my understanding of this subject and — with their citations — also have helped to impove this book by far.

1.4 Changes to second Edition

The second edition of the PretzelBook features a dramatically enhanced chapter on interfacing with `noweb` and minor changes to the chapter on Pretzel hacking.

Chapter 2

Using Pretzel

> "I'm using CWEB to write chapters for books, and really appreciate the pretty-printing and automatic indexing and cross-referencing."
>
> Tim Kientzle [26]

This chapter explains how to use Pretzel in an everyday setting. It covers the things you need to know if you want to build a Pretzel prettyprinter from an existing set of Pretzel input files or if you want to modify an existing input or write a totally new one from scratch. It assumes that the Pretzel system has already been installed on your system so that you can invoke the `pretzel-it` command from your command prompt. You don't need any prior experience with Pretzel or LaTeX to start, maybe just a slight idea of what kind of things regular expressions and context free grammars are, what prettyprinting is about and how your source code looks best in a prettyprinted way.

2.1 Getting Started

This section will give a first introduction to using Pretzel. It will show how to build a prettyprinter for a simple programming language, which will be a subset of standard Pascal. After reading this section and doing the excercises, you will be able to build arbitrary prettyprinters from the ready-to-go input files included in the Pretzel distibution.

2.1.1 A first Example

The input to Pretzel will always be *two* files. Let's start with these files first and look at Pretzel in action. So you might startup your computer, invoke your favourite text editor and create two files called `simpas.ft` and `simpas.fg`. The suffixes `.ft` and `.fg` are the extensions that Pretzel expects for its input files. But before we learn about what they stand for, fill in the files with the following information:[1]

```
/* simpas.ft -- a simple pretzel example */

%%

";"             SEMI
"="             |
```

[1] You can spare yourself from typing the data if you look into the subdirectory `languages/examples` of the Pretzel distribution.

```
"<"             |
">"             |
"<="            |
">="            |
":="            BINOP
"if"            IF
"then"          THEN
"else"          ELSE
"begin"         BEG
"end"           END
[0-9]+          NUM

[a-zA-Z][a-zA-Z0-9]*    ID

[\t\ \n]        // eat up whitespace
```

Here's the second file, which should be called `simpas.fg`:

```
/* simpas.fg -- a simple pretzel example */

%token SEMI BINOP IF THEN ELSE BEG END NUM ID

%%

stmt_list : stmt
| stmt_list SEMI stmt       { $1 $2 force $3 }
;
stmt    : IF exp THEN stmt          { $1 " $" $2 "$ " $3 " " indent $4
                                      outdent }
        | IF exp THEN stmt ELSE stmt
                                    { $1 " $" $2 "$ " $3 indent force
                                      $4 outdent force
                                      $5 indent force $6 outdent force }

        | BEG stmt_list END         { $1 " " $2 " " $3 }
        | exp     { "$" $1 "$" }
;
exp     : ID
        | NUM
        | exp BINOP exp
;
```

The language that is covered by these first Pretzel specifications is PASCAL with simple expressions of the form $a \leq b$ or $a = b$ (you can see which operators are allowed from the `pascal.ft` file). Also, the only statements allowed are assignments, the **if-then**-construct and the compound statement (i.e. statements enclosed within **begin** and **end**). So the following (not prettyprinted) code would be valid in these terms:[2]

```
if a=1 then b:=1;
if a<=1 then begin b:=2 end else b:=a
```

[2] Of course you can't do much with this "language", but remember that it's only an introductory example. Once you've understood the basics of this section it'll be easy to extend the specifications to cover all of PASCAL.

2.1.2 Running Pretzel

Now we are ready to start: Having these files in one of your directories, simply type:

```
pretzel-it simpas simpaspp
```

This will cause your machine to rattle and hum and if all goes well, it will return saying "done." and leave an executable called `simpaspp`. This file is the prettyprinter for the specified language!

By default, the prettyprinter reads text from the standard input and writes formatted code to the standard output. So try to feed the small example to the program and look what it returns:

```
simpaspp <small-example.simpas
```

This will result in a few lines that will have the name "pretzel" written everywhere and look something like this:

```
if $a=1$ then \pretzelindent{}$b:=1$\pretzeloutdent{};
\pretzelforce{}if $a<=1$ then\pretzelindent{}
\pretzelforce{}begin $b:=2$ end\pretzeloutdent{}
\pretzelforce{}else\pretzelindent{}\pretzelforce{}$b:=a$\pretzeloutdent{}
\pretzelforce{}
```

In effect, this is LaTeX code that you can use within your own documents.

2.1.3 Using Pretzel Output

There's a small document prepared to demonstrate the use of Pretzel output in your own documents. It's the file `example-frame.tex`.

```
\documentclass{article}
\usepackage{pretzel-latex}
\begin{document}

This is a small example produced by Pretzel and formatted with \LaTeX{}.

\begin{ppcode}
  \input{small-example.tex}
\end{ppcode}

\end{document}
```

If you save the output of the prettyprinter `simpaspp` in a file by redirecting the standard output (let's call this file `small-example.tex`) and run LaTeX on the frame, the resulting document will look something like this:

if $a = 1$ then $b := 1$;
if $a <= 1$ then
  begin $b := 2$ end
else
  $b := a$

Now what do you think of this? Is it what you expected? It is obvious that this is the code that has been input to the prettyprinter. But it is not obvious, how the output corresponds to the information written down in the two input files to Pretzel. This will be the subject of the next subsection. But, alas, you have worked your way through this first section and know now, how to use Pretzel in it's simplest way.

2.2 Carrying On

This section will show you how the basic prettyprinting method works that Pretzel uses. After reading this section you will be able to understand roughly what the contents of Pretzel input files mean and how to modify existing input files to go with your wishes.

2.2.1 The Two Input Files

The prettyprinting method used by a Pretzel prettyprinter is quite simple. It reads the input, which usually is the plain source code, cuts this text into small pieces called *tokens*, and tries to rearrange them in such a fashion that the user will call æsthetic. Of course, the relative order of the input tokens will (usually) not be changed, but the amount of whitespace inbetween might be enlarged or reduced, line breaks may be inserted at special places etc. All this is specified by rules that the user writes. These rules describe, how tokens are to be put together and how they in turn should be formatted.

The two input files of Pretzel reflect this separation of concerns. One file tells the prettyprinter which small tokens to expect and the second file states, how they should be put together again and how this should look. The first file is called the *formatted token file*, because it contains token descriptions. The second file is called the *formatted grammar file*, because that's what you call a set of such rules: a grammar.

2.2.2 Formatted Tokens

In the simple example from above the first file (`simpas.ft`) was the formatted token file. Looking at it you'll see that all the reserved words from our PASCAL subset appeared there together with all the bits that make up expressions (like the binary operators $<$, $\leq$, and so on). Accompanying these strings were special words in uppercase (like `IF`, `ELSE`) which are the symbolic names of these tokens. A vertical line in place of such a name means, that the name of this token should be the same as the one immediately following the current one. From now on, these tokens are referenced only by their symbolic name.

Most of these token definitions are quite simple, except those from the last three lines. If you already know, how a token looks like in the source code, you can just write it down and assign a symbolic name to it. However, if you don't know exactly what it looks like (which is the case with normal identifiers or strings for example), you'll have to specify an input pattern that the prettyprinter must match in order to find and name the token. This pattern is specified by using *regular expressions*.

2.2.3 Regular Expressions

A case in which you'll need this feature of Pretzel crops up with any language using identifiers for example. In PASCAL, identifiers begin with a letter between 'a' and 'z' (in upper- or lowercase) followed by zero, one or more letters or digits. This is expressed by the regular expression

```
[a-zA-Z][a-zA-Z0-9]*
```

Regular expressions are read from left to right. The squared brackets mean: "a character from this set or range". A star attached to such a range means: "zero, one or more of these characters." The regular expression for integer numbers looks like this:

```
[0-9]+
```

The plus sign means "one or more of these characters", which means one or more consecutive digits in this context.

This might be enough for a first introduction into regular expressions. See the manual pages `flex(1)` or `flexdoc(1)` for a complete reference on how to write regular expressions that match more complex patterns. However, it might be important to note here that the order of the regular expressions in the input file is important. When the prettyprinter reads its input it tries to match the regular expressions starting from the top. If it finds one or more patterns that match the input file, it will take the longest match (counting the characters matched). If there are several matches with equal length it will take the one that comes first in the file. Most notably, the pattern for identifiers is found at the very end of the file. If it had stood right at the beginning, all reserved words like `if` and `else` would habe been matched as `ID` tokens and not as `IF` or `ELSE`.

2.2.4 Formatted Grammar

The above explanations should be enough about the formatted token file. But formatted tokens are only half the input to Pretzel. The other half is the formatted grammar file. It specifies the rules by which these tokens are put together again.

The rules are specified in form of a *context free grammar*. A rule specifies what tokens to look for, to put together and what the resulting 'thing' is. Actually, a grammar specifies tokens too, only in a more complex way: it tells how complex tokens are put together by adding zero, one or more simpler tokens. These simple tokens may be the ones from the formatted token file, but also new ones specified in the grammar.[3] When such a rule is actually executed, we say that the simpler tokens are *reduced* to a (more) complex token.

A rule contains a 'resulting' token, a colon followed by a series of tokens from which the one preceding the colon is made. A rule is terminated by a semicolon. Several rules reducing to the same token can be appended by using a vertical line. The example above showed that an expression (token name is `exp`) can be built from the simple tokens `ID` (an identifier) or a `NUM` token (a number), or in turn can be made from two expressions that have a `BINOP` token inbetween. This is really recursive! Similar things count for statements (token name `stmt`). There is no restriction on the order of the rules as they appear in the formatted grammar file, except that the prettyprinter will always want to end up with the token that is mentioned first in the file. In our example, the prettyprinter will always end up with a `stmt_list` token. If this isn't possible, it will be unhappy and fail.

There are only two more things to say about formatted grammars: firstly there are these strange looking constructs amidst curling braces behind some of the rules, and secondly, there are these funny `%token` things at the top.

The `%token` things on the top of the formatted grammar file are the *token declarations*. Every simple token that is named within the formatted token file has to be declared here. This is just a way of telling the prettyprinter which tokens actually come directly from the input. They have to be there in order for the prettyprinter to work.

The other things inside the curling brackets are a little more interesting: they define how the major aspects of the output will look like. Major aspects are indentation and line breaks. This gets more to the heart of the matter of prettyprinting and we'll see that the stuff inside the brackets is as important as the rules outside. But this is so important that it deserves a subsection of its own.

[3] Token names can be chosen as you like. By convention, use lowercase for normal tokens and all uppercase for tokens that appear in the formatted token file. Do not try to call a token `error`. See section 2.3.5 to see why.

2.2.5 Prettyprinting with Format Instructions

This subsection introduces the way in which the user controls prettyprinter actions.

A lot of early prettyprinters were very simple: they read the programming language source code and scanned it for special words (like **begin**). If they found one of these patterns, they executed a special action. As most of the common output devices at that time were dot matrix line printers or dumb ASCII terminals the actions performed when a reserved word like **begin** were encountered were simple too. If for example the writer of the prettyprinter wanted a line break before the **begin** and a little indentation too, he would code this directly into his program. He would keep a local count of the amount of indentation, increment and decrement it when appropriate and indent to this amount when breaking lines.

Pretzel works in a similar but slightly more flexible way. We have seen that the formatted token file contains patterns that the prettyprinter tries to match to the input. This is exactly what the simple prettyprinters did, but nothing more. To implement a simple prettyprinter, we would only need to specify the patterns for the tokens and add some information, what the output device should do if this pattern can be matched. An example will show, that this is not enough, if you strive for full user control over prettyprinting.

For example look at the following PASCAL code fragment.

if $i > 1$ **then** $i := i - 1$;
if $b \leq i$ **then**
  $b := b + 1$
else $b := b - 1$;

Say the user wants the first statement (an **else**-less **if**) to be prettyprinted without a forced line break behind the **then** keyword and the second statement (a simple **if-then-else**) to have a line break behind the keywords and additional indentation to the statements to enhance readability. This is hardly expressible in terms of "formatted tokens", because it would be necessary to go back in the output and insert a line break within material that might have already been output to the screen or to paper.

To specify this distinction you have to tell the prettyprinter something about what **if-then**s are in difference to **if-then-else**s. That's where the formatted grammar comes into action.

If you look at the `simpas.fg` example above, you'll notice two rules that handle these two different cases. Here are both rules again to look at:

```
stmt    : IF exp THEN stmt          { $1 " $" $2 "$ " $3 " " indent $4
                                      outdent }
        | IF exp THEN stmt ELSE stmt
                                    { $1 " $" $2 "$ " $3 indent force
                                      $4 outdent force
                                      $5 indent force $6 outdent force }
;
```

The second line is more interesting at the moment: it means that any `IF` token followed by `exp`, `THEN`, `stmt`, `ELSE` and `stmt` tokens will be reduced again to a `stmt` token. The attached description in curled braces tells the prettyprinter to insert a forced line break (*force*) behind the `THEN` token if this rules fires. Additionally, the statement following the `THEN` should have a little more indentation before it (*indent*) which should be taken back afterwards (*outdent*). The same is specified for the second statement following the `ELSE` token. What is done here is to specify

something like an attachment (or special attribute) of the token which this rule reduces to (i.e. the `stmt` token on the left hand side). This attachment can use the attachments of the tokens on the right hand side by referring to the sequence number in which they appear there. This means that `$1` references the attachment of the `IF` token, `$2` the attachment of the `exp` token and so on. All the attachments between the curled braces are put together and form the attachment of the newly formed token (the `stmt` token). Apart from referring to the attributes of existing tokens, you can additionally insert special formatting instructions (like *indent*, *outdent* or *force*) and strings (like the `" $"`) into the sequence as "extra" attachments. All these attachments together form the attribute of the new `stmt` token.

The special commands that have appeared above, e.g. the *indent* or *force* commands, are called *format commands* or *formatting instructions*. Subsequently the prettyprinting method used by Pretzel prettyprinters is called *prettyprinting with format commands*. The main assumption is that we have a text formatter at hand that uses *control sequences* (or *tags*) inside the actual text body for specifying all different ways of formatting (such as fonts, indentation, spacing, alignment). Another term for this way of formatting is *in-text procedural markup* and systems using this technique are commonly called *document compilers*[4]. The idea behind the prettyprinting method presented here is to systematically insert special formatting instructions into the source code which are left to a typesetter to interpret. The history behind this method and the details of the algorithm are explained in the special chapter "On Prettyprinting" (chapter 5). From now on, simply think of formatting instructions as actions that will be performed on the typeset output before it appears on the screen or on paper.

2.2.6 Formatting Instructions

Now, we'll have a look at the formatting instructions that Pretzel offers.

Controlling Indentation

We have met three formatting instructions already: *indent*, *outdent* and *force*. They in effect control the amount of indentation. The *indent* primitive increases the level of indentation by one unit, *outdent* does the same in a decreasing fashion. However, the new level of indentation only becomes visible after a forced line break, which corresponds to the *force* command.

Let's look at an example. Here we have a simple **while** loop in PASCAL. If the input sequence is

```
while (i<size) do indent begin i:=i+1; end; outdent
```

the output will simply look like this:

while $(i < size)$ **do begin** $i := i + 1$; **end**;

However, if a **force** command is inserted behind the *indent* command, we'll get the expected output:

while $(i < size)$ **do**
  begin $i := i + 1$; **end**;

So the *indent* and *outdent* primitives cause only future lines to be in- or "out"dented and have no effect on the present line that is just processed. The indentation shows only if there is a forced line break after the amount of indentation has been changed.

[4]Common examples of such typesetting systems are TEX and the UNIX tools Nroff/Troff.

Why do *indent* and *outdent* to behave in this way? To understand this, we should consider the above example without the keywords **begin** and **end** surrounding the assignment. The result

while $(i < size)$ **do**
  $i := i + 1;$

really does look a bit stupid. In cases like this we would like the controlled statement to stand on the same line as the header because the program structure doesn't need to be visualized in this simple case by indentation. The only thing one has to to here is to remove the *force* command from the input sequence and we will get:

while $(i < size)$ **do** $i := i + 1;$

The interesting fact is that this doesn't change our global rule that the body of a **while**-statement should be *indent*ed from it's keyword. So the *indent* and *outdent* commands can always be inserted before and after the controlled statement. The only thing to worry about is whether or not to insert a *force* (which is always recommended before a **begin**, since this nearly always starts a sequence of two or more statements).

Controlling Line-breaks

Let's see, how you can control line breaking. Consider for example a long block of statements in PASCAL that contain extremely long names for identifiers as in the following example:

```
begin string_buffer[string_buffer_pointer]:=c;
string_buffer_pointer:=string_buffer_pointer + 1; end;
```

We don't want line breaks to appear within identifiers and we don't want things like this to crop up when prettyprinting:

begin *string_buffer*[*string_buffer_pointer*] := *c*; *string_buffer_pointer*
:= *string_buffer_pointer* + 1; **end**;

Here, a statement is broken across two lines, but we would prefer line breaks to occur only between two adjacent statements (and usually also before the **end**) like in:

begin *string_buffer*[*string_buffer_pointer*] := *c*;
string_buffer_pointer := *string_buffer_pointer* + 1;
end;

So the conclusion is, that we have to tell the formatter, where it is possible to break a line, if the current block of text should exceed the right margin. This is done with the *break_space* primitive. By inserting a *break_space* amidst two statements (and even after the **begin** and before the **end**) the above problem cannot happen any more. A *break_space* tells the typesetter, to break a line at this point, if he is looking for a place to break somewhere near it.

But even now, a single statement might be too long to fit onto one line. Consider for example a **while**-loop or an **if**-statement with a very long expression that governs it, as in:

```
while (str_buf[str_buf_ptr]<>c) and (str_buf_ptr<=buf_size) do
```

Inserting *break_space*s before and after the "`and`" would seem to be a good solution to this line-breaking problem. But breaking the line after the "`and`", say, would result in

while (*str_buf*[*str_buf_ptr*] <> *c*) **and**
(*str_buf_ptr* ≤ *buf_size*) **do**

what of course is intolerable. We want a long continuous line to be indented a little if it is broken, but the amount of indentation for the continuing line must be more as the "normal" indentation of controlled statements inf conditionals for example. This distinction visually separates adjacent (and indented) statements from a single statement that fills two lines. To this effect, there exists a new format command, namely the *optional line break* command *opt*. There are ten different *opt*s that specify the badness of a line break at the point of it's occurence. They are called *opt0*, *opt1*,..., *opt9*. A smaller number encourages line breaking at this point more than a higher number. So *opt0* will nearly always lead to a line break whereas *opt9* leads to a line break only if there is no *opt0*,..., *opt8* primitive somewhere near and the line *has* to be broken around this point.[5] So the *opt* command denotes an optional line break with the continuation line indented a little more with respect to the normal indentation.

So in the example we therefore say:

```
while (str_buf[str_buf_ptr]<>c) and opt4 (str_buf_ptr<=buf_size) do
indent force begin ...force end
```

depending on whether we rather liked a line break before or after the "`and`". This would then yield the correct way of formatting, as in:

while (*str_buf*[*str_buf_ptr*] <> *c*) **and**
(*str_buf_ptr* ≤ *buf_size*) **do**
begin...
end

Note however that a *break_space* is always a better place to break a line than an optional line break denoted by *opt*.

Additional Commands

The list of format commands is not yet complete. There are still a few more primitives to learn, but they are not as important as the other commands that were presented up to this point. The following primitives were introduced to achieve better formatting results in special situations.

The first one is called *backup* and denotes a little negative horizontal space. This command must be preceded by a *force*, because it doesn't make sense to backup within a line if it's beginning has already been output. So the two commands "*force*, *backup*" appearing in a row can be seen as a short form for the sequence "*outdent*, *force*, *indent*". The *backup* command is used in cases where statements are preceded by labels or simply where a line should stick out to the left a bit in order to emphasize it.

The next additional command is called *big_force*. As the name implies this primitive acts like a *force*, but on a bigger scale. *big_force* not only forces a line break; it also inserts a little extra vertical space after the break. You can use the *big_force* primitive to separate two adjacent lines that semantically do not belong together, like two function declarations.

If you are trying to typeset C or C++ programs, you will need a feature to typeset so called preprocessor directives. These are commands beginning with a # that can influence the source code before compilation. Normally these lines appear beginning at the left margin so we want a command that can be used to typeset

[5] In TEX terms the digit n in the name of the *opt* command specifies the penalty associated with this break point. The penalty in the default interpretation of *opt* is about $10 \cdot n$.

lines like this. This command is called *no_indent*. It causes the **current** line to be printed flushleft. So if you want to prettyprint a C or C++ preprocessor line, you should say

force no_indent #`PREPROCESSOR_DIRECTIVE` *force*

and the line following this directive will be indented as much as the line before the directive.

Finally the last command of the present command set is called *cancel*. To understand this command I have to mention a tricky detail of the presented algorithm that I have omitted in the intoduction of this section: There I said that format commands are simply output by the prettyprinter one by one just like strings, only that they are "marked" as typesetting commands. Actually the prettyprinter also does a little bit of interpretation in advance. But these exceptions from the rule can be caught in two phrases. The first one is of global importance and the second one is related to the *cancel* command:

1. A sequence of consecutive *break_space*, *force*, and/or *big_force* commands is replaced by a single command (the maximum of the given ones).

2. The *cancel* command cancels any *break_space*, *opt*, *force* or *big_force* command that immediatly precede or follow it and also cancels any *backup* command that follows it.

So it seems that the *cancel* command is a command that mixes everything up again after we have put everything into a neat and tidy form! But this only seems so. The interpretation done by the prettyprinter according to the two rules stated above, was introduced to be more generous and more flexible with the placement of format primitives into the source code. Because, after all, the format commands will not be inserted by hand (as the reader might think after viewing the examples given above), but they will be inserted automatically by the prettyprinter according to the prettyprinting grammar that we have been introduced to above.

Summary of Format Commands

Table 2.1 shows a complete list of all the format command primitives introduced in this part of the book. The only command that wasn't mentioned up to now is the *null* primitive that serves as a neutral (or empty) command. Now, with the knowledge of all the intrinsic and extrinsiv features of the prettyprinting method of Pretzel prettyprinters, you are ready to read, write or modify prettyprinting grammars at your will. Now, you have total control over the prettyprinting style; fully user controlled prettyprinting is in your grasp.

2.3 Writing Prettyprinting Grammars

> "Knowing `CWEB` and its sources, it is clear to me that doing a good job (or even a fairly good job) about prettyprinting is not a simple task."
>
> Marc van Leeuwen [64]

Yes, this is true. In terms of Pretzel, "doing a good job about prettyprinting" is writing a good prettyprinting grammar. For simple languages you might spend a day or two preparing a good grammar, but for more complex languages (and more complex wishes to the look of the output) you might spend weeks and still not be happy.

null	empty command
indent	indents the next line a little more
outdent	takes back the last indentation (de-indent)
force	forces a line break
break_space	denotes a possible space for a line break
opt n, $n \in \{0, \ldots, 9\}$	denotes an optional line break with the continuation line indented a litte with respect to the normal starting position. This line break will score a penalty of $10n$.
backup	denotes a small backspace
big_force	forces a line break and inserts a little extra space
no_indent	causes the current line to be output flushleft
cancel	obliterates any *break_space*, *opt*, *force* or *big_force* command that immediatly precedes or follows it and also cancels any *backup* command that follows it.

Figure 2.1: List of format command primitives

2.3.1 Modifying an existing grammar

If you're lucky, somebody else had done the work already for you. In such cases, simply grab a copy of the two Pretzel input files, run `pretzel-it` and off you go. The Pretzel homepage [14] should be a good starting point for the quest for a prettyprinting grammar for your language.

But what if this grammar doesn't suit your preferences? What if it prettyprints ugly code? Well, then you have to plunge into the files and modify them. Making small changes that will work is quite easy. For example, if you only dislike the way certain symbols come out of the prettyprinter, look in the formatted token file and try to figure out, where you can change it. Or if you dislike forced line breaks at certain places, just go into the grammar and remove the *force* from the attribute definition.

If your wishes cover more major aspects of the layout, then you should make a copy of the file and rename it to something that reflects the change. Also, you might put a new comment at the beginning of the file that describes who you are and what this new grammar does differently than the grammar you started from. Now you can start to experiment. The good thing about this is that you can always fall back to the original copy and start from there again. If you get your new grammar working, contact the Pretzel home page [14] so it can be added to the list of files for certain languages. Other people can then use your grammar and don't have to go through the same work again.

For similar languages (like Java and C) you could try to take a grammar for the first one and try to modify it that it fits the second one. In any case, studying existing grammars is always very helpful if you don't have much experience.

2.3.2 Writing a new Grammar from Scratch

If you're unlucky, and you can't find a suitable grammar then you will have to start from scratch. If you're in this situation, there are two possible ways of continuing:

1. Most programming languages are context free and so there should be a context free grammar for your language available either in the reference works (i.e. the books) that "define" the language, or somewhere in the internet,

where there could be compilers for your language available in the source, and if so, a context free grammar for the parser of that compiler is almost always included there. You could start with such a grammar and simply add formatting instructions to the rules as you wish.

2. You could start entirely from scratch, look at the language and start writing a prettyprinting grammar that works with the code you are working on. This might seem more work than the first alternative, but sometimes is still feasible.

To go with the first alternative, you will have to have a full blown language grammar available. But these grammars (as for example with C or C++) tend to be rather large and ugly, resulting in prettyprinting grammars, that are difficult to understand and difficult to modify. However, a prettyprinting grammar doesn't have to be a full blown language grammar after all. The fact that such a grammar will solemnly be used to create a "prettyprinted" translation of a bit of source code it doesn't necesserily have to have any resemblence to a language grammar. In PASCAL for instance, **for**-, **while**- and **else**less **if**-statements are normally formatted in a very similar way; so the grammar mustn't distinguish between the three constructs. This results in prettyprinting grammars that can be quite different from the normal grammars of the programming languages. In most cases they are much smaller.

However, if you start from scratch and don't want to use an existing reference grammar, you could still consult it once and again if you are stuck with your own one. Like the language grammar, the prettyprinting grammar must be able to parse *every* construct that will appear in the input. But it's no problem if the prettyprinting grammar might swallow code that isn't syntactically correct. We're talking Prettyprinters here and not Compilers. Sections 2.3.3 and 2.3.5 will elaborate more on the issue of "grammar correctness." In any case, having a look at available prettyprinting grammars in advance helps a lot.

2.3.3 Context Free versus Context Sensitive

Looking at the history of the prettyprinting method that Pretzel employs (see chapter 5), we can see that the grammars that the first systems started with were context sensitive. This means that there actually are ready to use context sensitive prettyprinting grammars around for many languages such as PASCAL [29] and C/C++ [33]. There is even a system similar to Pretzel but with a more complete approach called SPIDER [49] that amoung other things created a prettyprinter from a context sensitive prettyprinting grammar.

So why does Pretzel use context free grammars anyway? The answer to this problem lies in the tools that Pretzel relies on. The prettyprinters that Pretzel generates are actually parsers, and instead of writing a whole new parser generator, Pretzel relies on the well known UNIX tool Bison, that does just the work we need. But, in effect, Bison input is context free! Hmmm.

Now context free grammars are a little more restrictive than context free ones, and so it is not possible to transform any context sensitive grammar of your choice into a context free one. This is unfortunate, but this is a fact we have to live with. My knowledge of formal language theory is not very deep, but I suspect that parsing with context sensitive grammars is not trivial and that this is the reason why no generators like Bison exist in a context sensitive flavour.

So context free is what we have to live with. From studying existing prettyprinting systems (again see chapter 5 for details) it seems that context sensitive is the way to go in prettyprinting, and this seems bad for Pretzel.[6] But Pretzel is a pro-

[6] I actually *is* a quite severe restriction and will probably be the reason why Pretzel will not be widely used in practice. The extension of being able to add C code to the attribute definition only partially eases the situation (see section 3.1 for details).

totype and (at the moment) the best you can get as a prettyprinter generator and a context sensitive implementation of Pretzel is left for a following student generation to pursue. The good thing about context free grammars however is that every context free grammar is also a context sensitive grammar, and so — if a context sensitive version of Pretzel is ever built — every existing and working prettyprinting grammar can be used with the new system without a single change. At least this is good news. For more information on this topic, also see chapter 6 on "Future Work."

2.3.4 Available Grammars

All those prettyprinting grammars that are available in Pretzel format are contained in the latest distributions of Pretzel (at least all that I know of). If you have a good grammar for your favourite programming language at hand, let me know and I'll include it in the distribution too.

All these input files are available individually from the Pretzel homepage [14]. So if your Pretzel installation is rather old, then surf to that URL and look if there's something new on the page.

To date (July 9, 1997) I am aware of:

- C (by Felix Gärtner)
  Directory: `languages/cee`
  Files: `cee.ft` and `cee.fg`

- Pascal (by Felix Gärtner)
  Directory: `languages/pascal`
  Files: `pascal.ft` and `pascal.fg`

- Java (for `noweb` by Lee Wittenberg)
  Directory: `contrib/leew`
  Files: `java.ft` and `java.fg`

- Java (for `noweb` by Felix Gärtner)
  Directory: `contrib/noweb/java.latex`
  Files: `javafx.ft` and `javafx.fg`
  Comments: Based on the grammar by Lee Wittenberg with slight changes, but enhanced to do indexing.

- Java (for `noweb` by Holger Uhr)
  Directory: `contrib/huhr`
  File: `javahu.fg`
  Comments: Replaces `java.fg` from Lee Wittenbergs version in `contrib/leew`.

Always see the `README` files in the directories for details.

2.3.5 Debugging Prettyprinting Grammars

> "Rather, it [the need to type '`@;`' after each refinement] is caused by an error made in converting `WEB` to `CWEB` that was never corrected, namely a failure to recognise the fundamentally different roles of semicolons in Pascal and C. In Pascal semicolons separate statements, whether simple (e.g., assignments) or compound, so if any such statement is followed sequentially by another one, a semicolon is required after the first; the requirement still holds if the first statement is a refinement. In other words, in Pascal one has 'refinement → statement' always, without having to consider `LINE_END`, and without having to use '`@;`'

> (either a real semicolon is required after it, or nothing is, e.g., when `ELSE` follows). In C however, no symbol separates successive statements, but things like assignments are expressions rather that statements, and they only become statements by incorporating a following semicolon. Since refinements almost always stand for (compound) statements, they should be given that category; however in (Levy/Knuth) `CWEB` they are given the same category as assignments, which is 'expression'. This forces one to write a semicolon after each refinement, even though that really becomes a spurious empty statement; if such an empty statement would mess up the real syntax (the compiler would fail) then one has to write '`@;`' instead."
>
> Marc van Leeuwen [62]

Almost certainly, the first experiences with your own (or other's) prettyprinting grammars will lead to situations where source code is not prettyprinted the way you wanted and you don't know why. The insight that follows from this situation is very frustrating: nothing comes for free! Searching for bugs in prettyprinting grammars is as necessary as in normal programs.

The `error` Token

If a Pretzel prettyprinter can't handle a sequence of input code, it will output *nothing*, not even an error message. But there is still a way to notice this fact. There is a special `error` token that may be used in such circumstances. You can place it into your grammar at any place, and it will match a sequence of tokens not covered by the other rules in question.

A handy place to put an `error` token is right at the beginning of the grammar. Say, a `final` token is the token that everything reduces to in your grammar. To notice a parse error, you can for example write

```
final   : exp
        | stmt
  // check for parse errors:
        | error  { "syntax error" }
;
```

and so the text "syntax error" appears in the output everywhere Pretzel wasn't able to parse the input.

You may place the `error` token in any place you like, as long as it is on the right hand side of the colon. Using the `error` token in other places than at the end, you may produce grammars that are more robust and can handle unforseen constructs more flexibly.

Watching the Parse

The `pretzel-it` program offers a `-d` option that produces a prettyprinter which is in "debugging mode" by default. This means that during prettyprinting, the program will output a lot of debugging information to the standard error stream (usually the screen). This information displays what kind of token has been read from the input (and how it looks) and what rule of the prettyprinting grammar has been chosen. Additionally, the `-d` option produces a file that contains an analysis of your prettyprinting grammar. If your input name is, say, `foo`, then this file will be called `foo.output`.

Here's an example of the debugging output (taken from a run on the `small-example.simpas` example):

```
Starting parse
Entering state 0
Reading a token: Next token is 260 (IFif)
Shifting token 260 (IF), Entering state 1
Reading a token: Next token is 266 (IDa)
Shifting token 266 (ID), Entering state 4
Reducing via rule 7 (line 195), ID  -> exp
state stack now 0 1
Entering state 8
Reading a token: Next token is 259 (BINOP=)
Shifting token 259 (BINOP), Entering state 11
Reading a token: Next token is 265 (NUM1)
Shifting token 265 (NUM), Entering state 3
Reducing via rule 8 (line 196), NUM  -> exp
state stack now 0 1 8 11
Entering state 15
Reading a token: Next token is 261 (THENthen)
Reducing via rule 9 (line 197), exp BINOP exp  -> exp
[...]
Entering state 14
Reducing via rule 2 (line 102), stmt_list SEMI stmt  -> stmt_list
state stack now 0
Entering state 5
Now at end of input.
Shifting token 0 ($), Entering state 19
Now at end of input.
```

Every time, the scanner reads a token, the token code (like `IF` or `ID`) is printed which is followed by the actual contents of the token (like "`a`" in case of the identifier). Every time the parser reduces a token, it prints the rule that it uses. The number of the rules and also the state information will become clear if you look at the `simpas.output` file and learn about the internals of the Pretzel parsing mechanism (by looking at the manual page `bison(1)`).

Context Free again

As described above, context free grammars are pretty restrictive, sometimes even more restrictive than we like. Note that rules like

```
decl_head id :   int_like id
;
```

to turn an `int_like` token into the head of a declaration if it is followed by an identifier are context senstive and therefore forbidden.

See also the next chapter, especially section 3.1.4 for a few more tips and tricks for writing and debugging prettyprinting grammars.

2.3.6 Experiences

> "With `CWEB`, there'll always be something someone wants and can't have, because it tries to do too much in its kernel. Witness the recent complaint in this newsgroup from someone trying to put makefiles in the `CWEB` sources. `CWEAVE` tries to be the perfect C programming tool right out of the box, and so comes up completely unable to handle makefiles."
>
> Barry Schwartz [58]

Remember, writing good prettyprinting grammars is difficult. So don't overdo it. It's pretty difficult (if not impossible) to come out with a "complete" grammar that will handle all cases brilliantly. So if you're happy with the way your code looks, that's fine. If something turns out wrongly, ask yourself, what actually *is* wrong here?

Chapter 3

Pretzel Hacking

In this chapter we'll have a look into the guts of Pretzel. We'll see how the internals of `pretzel-it` work and what other nice things you can do with the system. This chapter is aimed at programmers who want to produce simple, efficient and elegant prettyprinters for their language. However, I recommend reading at least the first subsection about adding C code to the rules, even when you dislike hacking.

3.1 Adding C Code to the Rules

Yes, you cann add your own C code to the attribute definition parts of the formatted grammar and formatted token files. A typical application of this feature is to dynamically typeset constructed type names (as **typedef**s in C or **class**es in Java) correctly.

The problem with Pretzel is, that we would like to perform different actions depending on the context of the token that the prettyprinter finds. For example, in C normal identifiers should be typeset in italics, but identifiers that are the names of **typedef**s should be typeset in bold. Pretzel can't handle this situation without tricky manipulations of the attribute definitions, because the type of grammar that Pretzel uses is essentially context free (see section 2.3.3 for a discussion of related problems).

3.1.1 Example for Tokens

How handle this situation? Pretzel allows you to add C code to your attribute definitions. Here is a small example, how this could be used in formatted token files:

```
[a-zA-Z][a-zA-Z0-9_]*    ID  { "{\\it "  [escaped_underlines(yytext)] "}" }
```

The purpose of this line is to scan tokens that are identifier names and return "`ID`" type tokens to the prettyprinting parser, but also to modify the appearance of the matched text before returning. In this example there is C code inside the attribute definition. This code is encircled in angled brackets "`[`" and "`]`" and can be any sequence of C statements with one central property: The code part must reduce to the **Attribute**$*$ class, i.e. the return type must me a pointer to an **Attribute** object. The **Attribute** class is a special C++ class that belongs to the Pretzel system and is contained in the runtime library.[1]

To this effect, the *escaped_underlines* function called above must have the syonpsis:

[1] A description of how to use objects of this class and related functions can be found in the `noweb` file `attr.nw` in the directory `attr` of the Pretzel distribution.

Attribute∗ *escaped_underlines*(**char**∗ *s*);

The matched text is accessible via a string (i.e. a **char**∗ variable) called *yytext*. The delimiters of code are angled brackets as explained above. If angled brackets appear in the C code itself, any closing bracket must be escaped by a backslash in order for Pretzel to recognize it correctly.

To help simplify the synopsis of the *escaped_underlines* function, we may use special functions in the runtime library that belong to the **Attribute** class. These are the two functions *create* and *join*. The *create* function takes a string (i.e a **char**∗) and turns it into an attribute. So, if you have written a function, say *esc_ul*, that turns a string into the same string with escaped underlines, then an implementation of the function above could look like this:

```
Attribute* escaped_underlines(char* s)
{
  return create(esc_ul(s));
}
```

The *join* function simply joins two or more (up to ten) attributes together into an new one.

3.1.2 Example for Grammars

But you can do more with this feature. Here's an example code fragment that handles the "correct" formatting of **typedef**s in C. In C you can construct your own shorthands for types by using the **typedef** construct, thus making normal identifiers like *Length* to type names. Here's an example, if you're not aquainted with C:

```
typedef int Length;
Length l;
```

The identifier *Length* has the status of a type name and this should be reflected in the typesetting style. So the way we want it would be like this:

```
typedef int Length;
Length l;
```

The idea behind a simple solution to this problem is to use a simple lookup table to keep track of all identifiers that have been **typedef**ed. Then, if we have to prettyprint an identifier, we simply look into the table and from the result we know whether we can use boldface or italics.

A prettyprinter for C[2] might have rules like these to handle **typedef** statements:

```
typedef : TYPEDEF_LIKE INT_LIKE ID      [ install($3); ]
                                        { $1 "\\ " $2 "\\ {\\bf " $3 "}" }
;
```

(We assume that *install* is a function that installs an identifier in the lookup table and that *lookup* is a function that tells us if an identifier is in the table.)

What we see here is called *starting code*. Starting code is code in angled brackets that is placed before the attribute definition. It can contain any sequence of C code

[2]Like the one in the `languages/cee` subdirectory of the Pretzel distribution.

that you like and will be executed before the attribute of the reduced token is even touched. You can use it to place declarations of variables or any other actions that should take place before the attribute is pieced together.

So here, in this rule, we install an identifier that is **typedef**ed into the table. The identifier is typeset in bold already (but you could put it into any font you like). Now, every time we handle an identifier within declarations or expressions, we lookup it's name in the table. A rule like this does the job:

```
id      : ID            { [lookup($1) ? create("{\\bf ") :
                              create("{\\it ")] $1 "}" }
;
```

Note that we are making use of the conditional expression of C. Remember that the code within the attribute definition has to turn out as type **Attribute**$*$ and so we use the *create* function again to turn a string into an attribute. So here again we use code *within* attribute definitions and all in all, we can use identifiers as `id` tokens in the grammar and know that they are typeset correctly.

Apart from code before the attribute definition and code within attribute definitions, there can also be code after the attribute definition which is called *ending code*. Ending code can be any kind of sequence of C statements. They will be executed *after* the attribute definition has been put together and *before* the final token identifier id returned. So in effect you yourself may play God and return a token identifier yourself. The following extract from a formatted token file is an example:

```
[a-zA-Z][a-zA-Z0-9]*      ID    { ** }    [ return(ID); ]
```

Here, the **return** statement is executed just before the prettyprinting scanner itself returns the `ID` token, so here you could leave the ending code away and nothing would be changed. But of course, you could put an **if** statement there and return something else as you wish. The symbolic names of all the tokens may be accessed with the code parts simply by their names you gave them.

Now you might wonder, what the sequence

```
[a-zA-Z][a-zA-Z0-9]*      ID      [ return(ID); ]
```

would mean. Tricky you! Is this starting or ending code? Well, Pretzel doesn't want to care about these kind of questions, and so it simply declares them illegal. If you want to use either starting code, ending code or both, you *need* an attribute definition.

3.1.3 Summary

This is how Pretzel expects you to write code inside your Pretzel input files:

- Code fragments are bracketed within angled brackets. Any angled brackets that appear within the C code must be escaped with a backslash.
- Starting code and ending code are written outside of the attribute definition. Both are totally optional, but if you want to specify either or, you need an attribute definition.
- Code parts within attribute definitions must return a pointer to an **Attribute** class object.
- Within the formatted token file, the matched text is visible to you in form of a **char**$*$ variable called *yytext*. The symbolic names of the tokens are available by the same name that Pretzel gives them.

- Starting code, code within attribute definitions and ending code is totally optional. But at any place where they are allowed, only one bracketed code bit may be placed.

Common routines to escape identifiers, to build and manage lookup tables, to convert to and from **Attribute**∗ or to output debug information can be found in the files belonging to the C prettyprinter in the directory `languages/cee` of the Pretzel distribution.

3.1.4 Tips and Tricks

One thing I have found usefull is to slip debug output into the Pretzel input files via C code added to the attribute definitions. For example, I have written a function called *debug_print* that takes a string and an **Attribute**∗ and outputs the contents of the attribute to *stderr*. Here is what it could look like:

```
static void debug_print(char* this_is, Attribute* stuff)
{
  cerr << "***␣" << this_is << ":" << endl;
  Latex_cweb_output os(cerr);
  stuff→print(os);
}
```

This will use a **Latex_cweb_output** object (see the file `output.nw` in the `output` directory of the Pretzel distribution or section 3.6 for details) to output the contents of an attribute to the standard error stream. The idea now is to insert a code fragment that outputs the contents of an attribute at special places in the grammar, like for example before the prettyprinter returns:

```
final : exp          { "$" $1 "$" }  [ debug_print("Final exp", $1); ]
```

This will output the contents of the expression `exp` to the standard error stream before reducing it to a `final` token and exiting.

Also, Pretzel prettyprinters do not insert newlines into the output if they are not told to. So, if no special care is taken, most prettyprinters will output one single long line of prettyprinted code. This might upset some typesetting systems that have restricted input buffers and are used to handle one line at a time. This problem may be avoided by manually inserting newline characters from time to time into the output, as for example after every statement:

```
stmt : exp semi      { $1 $2 force "\n" }
```

This will lead to better readable input and will make a lot of formatting programs happier.

There are a couple of handy functions available that do stuff like escaping underlines and turning an attribute into a string. Look at the files `javafx.ft` and `javafx.fg` in the `contrib/noweb/java.latex` directory of the Pretzel distribution for details. They are self-contained Pretzel input files that have a hash table built into them and do all converting by themselves.

3.2 The Pretzel Interface

The actual Pretzel program (called `pretzel`) does only half the job of `pretzel-it`. It simply builds ready to use source code for the specified prettyprinter in terms of input files to the two standard UNIX tools flex and Bison. Using these tools

will result in compilable C++ source code that contains the actual prettyprinting module.

Pretzel prettyprinters consist of two parts:

1. a *prettyprinting scanner* that cuts the input into small pieces (called *tokens*), and
2. a *prettyprinting parser* that joins the tokens returned by the scanner together and does the actual prettyprinting.

Both, the scanner and the parser, are seperate modules with a well defined interface. In this section I will present this interface.

3.2.1 The Prettyprinting Scanner

The prettyprinting scanner is a C++ prettyprinting scanner class. Every prettyprinting scanner produced by Pretzel is a subclass of an abstract scanner class that looks like this:

```
#include<iostream.h>
#include"attr.h"

class Pscan {
public:
  Pscan(istream*) {};
  ~Pscan() {};
  virtual int scan(Attribute**) = 0;
};
```

A Pretzel scanner must be associated with an C++ input stream object that is passed to the constructor of the class. The scanner will read characters from this input stream when scanning. The actual scanning if performed inside the *scan* member function. It's return codes are integers and identify the token that it has just scanned. If the input is empty, it returns 0.

The subclass that Pretzel produces will always be a subclass of the above class. It's interface is similar:

```
class PSCAN_NAME : public Pscan {
public:
  PSCAN_NAME(istream*);
  ~PSCAN_NAME();
  int scan(Attribute**);
};
```

The name of the generated class is `PSCAN_NAME` by default and has to be redefined using preprocessor macros, such as for example

```
#define PSCAN_NAME Pscan_for_pascal
```

By default the name of the derived subclass will be `Ppscan`. A header file containing the derived scanner class could look like this (by default the header file is called `Ppscan.h`). Note that there is no precaution against double inclusion.

```
/* header file for a prettyprinting scanner */

/* NB: This file is NOT protected against double inclusion! */

#include "Pscan.h" // include abstract base class
```

```
#ifndef PSCAN_NAME
#define PSCAN_NAME Ppscan
#endif

class PSCAN_NAME : public Pscan {
public:
  PSCAN_NAME(istream*);
  ~PSCAN_NAME();
  int scan(Attribute**);
};
```

This is exactly the default header file that you can find in the `include` directory of the Pretzel distribution. If Pretzel is installed on your system, you should find this file (and the header file for the abstract base class) in the global Pretzel include directory too.[3]

3.2.2 The Prettyprinting Parser

The interface to the prettyprinting parser is similar in structure to the prettyprinting scanner. We have an abstract base class called `Pparse` that has this interface:

```
#include<iostream.h>
#include"attr.h"
#include"output.h"

class Pparse {
public:
  Pparse() {};
  ~Pparse() {};
  virtual int prettyprint(istream*, ostream*) = 0;
  virtual int prettyprint(istream*, Output*) = 0;
};
```

There are the usual functions to create and destroy a prettyprinting parser plus an overloaded virtual member function *prettyprint*. The normal use of *prettyprint* uses C++ input and output streams. It reads from an input streams and writes prettyprinted code to the output stream.

The second version of this member has an *Output* class pointer as second argument. This is a version for experts. See section 3.6 for an example how to utilize this.

The prettyprinting parser class which is actually generated is again a subclass of this abstract base class. It's name may be redefined using C preprocessor macros.

```
class PPARSE_NAME : public Pparse {
public:
  PPARSE_NAME();
  ~PPARSE_NAME();
  int prettyprint(istream*, ostream*);
  int prettyprint(istream*, Output*);

  void debug_on();
  void debug_off();
};
```

[3]This directory will usually be called `/usr/local/lib/pretzel/include` or something alike.

The two extra functions *debug_on* and *debug_off* enable and disable debugging output. Debugging output gives an indication what the scanner returns and what kind of rules the parser is reducing. It is really a lot of information.[4]

As you can see, the name of the generated class is `PPARSE_NAME` by default and has to be redefined using preprocessor macros, such as for example

```
#define PPARSE_NAME Prettyprinter_for_pascal
```

By default the name of the derived subclass will be defined as `Ppparse` (note the three 'p's). This is the default header file for the generated parser:

```
/* header file for a prettyprinting parser */

/* NB: This file is NOT protected against double inclusion! */

#include "Pparse.h" // include abstract base class

#ifndef PPARSE_NAME
#define PPARSE_NAME Ppparse
#endif

class PPARSE_NAME : public Pparse {
public:
  PPARSE_NAME();
  ~PPARSE_NAME();
  int prettyprint(istream*, ostream*);
  int prettyprint(istream*, Output*);

  void debug_on();
  void debug_off();
};
```

This file resides in the `include` directory of the Pretzel distribution and together with the header of the abstract base class is also is found in the Pretzel `include` directory of the system (if Pretzel is installed).

3.2.3 Example

A minimal file that utilizes a Pretzel prettyprinter could look like this:

```
#include <iostream.h>
#include "Ppparse.h"

int main () {
    Ppparse prettyprinter;
    prettyprinter.prettyprint(&cin,&cout);
    return(0);
}
```

Firstly, the actual prettyprinter class is included. Then we declare a simple prettyprinter and simply call *prettyprint*. Here, the input and output streams are connected to the standard input and the standard output. If this *main* routine is compiled and linked with the objects produced from the output of Pretzel, the prettyprinter will work as specified in the two input files. The prettyprinting parser internally calls the prettyprinting scanner by its default interface.

[4]The information provided using the `-d` option of `pretzel-it` program comes from using these functions (see section 2.3.5).

3.3 Building a Pretzel prettyprinter by Hand

If you need to produce a prettyprinting module by hand you'll have to directly invoke Pretzel, flex, Bison and your C++ compiler in the right order. To ease this, you can use the `Makefile` in the PASCAL subdirectory of the distribution (`languages/pascal`).

To produce (only) the prettyprinting parser from the file `pascal.fg` for example, you could type the following:

```
 $ pretzel -g pascal
This is pretzel, version 2.0.
Processing the formatted grammar file (pascal.fg -> pascal.y).
No errors found.
 $ bison -d pascal.y
pascal.y contains 21 shift/reduce conflicts.
 $ mv pascal.tab.h ptokdefs.h
 $ g++ -c -I$PRETZEL_INCLUDE -g pascal.lex.c
```

We assume that the `PRETZEL_INCLUDE` environment variable points to the Pretzel include directory. The `-d` option of Bison will produce a token header file that the scanner expects (the scanner must know, which tokens the parser awaits and which token codes they have). The scanner expects these definitions under the name `ptokdefs.h`.

The above sequence of commands should leave a `pascal.tab.o` file in your directory. This is the object code of the prettyprinting parser with the default interface (i.e. the generated class' name is `Ppparse` and is declared in the include file `Ppparse.h`). You can now use this prettyprinter in your own programs: by including the header file the prettyprinter class will become visible; by linking the object with your own program you'll get your executable. (Note that you'll need to provide a prettyprinting scanner youself in this case or simply produce a suitable one in a similar way using Pretzel.)

3.4 Obtaining a Pretzel Prettyprinting Module

In the preceding section I have explained how to produce a Pretzel prettyprinter by hand. Now we'll see how you can change the default interface of the produced prettyprinting scanner and parser.

3.4.1 The Prettyprinting Scanner

Say, you need a prettyprinting scanner class called *Fooscan* in a header file called `Fooscan.h`. All you have to do is to make a copy of the default header file `Ppscan.h` (which you can find in the Pretzel include directory), change its name to `Fooscan.h`. Then you have to change the one single **#define** statement therin, i.e. instead of

```
#ifndef PSCAN_NAME
#define PSCAN_NAME Ppscan
#endif
```

you'll have to write

```
#define PSCAN_NAME Fooscan
```

Within your formatted token file, you'll have to tie in the same definition like this:

```
%{
#define PSCAN_NAME Fooscan
%}
```

It is important that this text appears in the declarations section of the formatted scanner file, i.e. before the first line that contains the double percent sign '`%%`'.

Compiling and linking the source produced by Pretzel and flex you'll get the object code that contains the desired prettyprinting scanner. The scanner expects to find a token header file under the default name `ptokdefs.h` with definitions of the symbolic names and token codes of the tokens it should be able to scan. Either you produce this file by hand or you let Bison do the job for you (using the `-d` option if you have a prettyprinting grammar). Look into a standard `ptokdefs.h` file to see what it should contain. The name of the token header file `ptokdefs.h` may also be changed. You have to define the `PTOKDEFS_NAME` macro within your formatted token file like this:

```
%{
#define PTOKDEFS_NAME "fooscantoks.h"
%}
```

Now the scanner will include the file `fooscantoks.h` and look for token code definitions in there. (Again, this redefinition must be in the declarations section of the formatted token file.)

3.4.2 The Prettyprinting Parser

Changing the interface of the prettyprinting parser works in a similar way as just explained for the prettyprinting scanner. The differences lie in the macro names that have to be redefined.

The name of the derived prettyprinting parser class is `PPARSE_NAME`. If you say in the definitions section of the formatted grammar file for example

```
%{
#define PPARSE_NAME Foopp
%}
```

then your prettyprinter class will be called `Foopp`. Internally it expects the prettyprinting scanner with it's default interface (i.e. under the names `Ppscan` for the class and `ptokdefs.h` for the token header file). If you redefine the same macros as in the formatted token file, then you redefine the interface under which the prettyprinter expects the scanner. So lines like these in the formatted grammar would urge the prettyprinter to use a `Fooscan` class object for scanning:

```
%{
#define PPARSE_NAME Foopp
#define PSCAN_NAME Fooscan
%}
```

3.5 Multiple Pretzel Modules in the same Program

Using multiple Pretzel modules within a single program has been greatly simplified by introducing the object oriented interface that has been explained in section 3.2. The main thing is to change the names of the two scanners/parsers produced by

Pretzel to something distinct and then to compile and link then together. That's the theory.

But there's a problem here: the scanners and parsers produced by flex and Bison contain a couple of global variables whose names will clash when you try to link them together. This may be circumvented by using special options of flex and Bison to change the prefix of these globals when a scanner/parser is built. As I haven't needed this feature in the past (but only tested it), I haven't prepared an additional option to `pretzel-it` to do this in a nice and easy way. At the moment you'll still have to consult the manual pages `flex(1)`, `flexdoc(1)` and `bison(1)` for details. If this is a feature you need, tell me and I'll add it to `pretzel-it` (see also chapter 6, "Future Work").

Another thing that sounds interesting, but that I have not tested yet, is to use several instances of a single prettyprinter class produced by Pretzel in the same program. If you have ever tried it, I'm very interested in your results (see chapter 6, "Future Work" again).

3.6 Prettyprinting for non-LaTeXians

> "Of course `CWEAVE` does visual formatting, at least for the part that cannot be left to be visually formatted by TeX: it issues forced line breaks, explicit changes of indentation level etc. The logical markup is really contained in the source document, and not (fully) in the intermediate file produced by `CWEAVE`. However `CWEAVE` does not do complete visual formatting: for instance many symbols like operators are produced as macros, so that you can easily change their appearance by overriding the default definitions, and breaking of long lines is also left to TeX's line breaking algorithm."
>
> Marc van Leeuwen [60]

Pretzel was build with LaTeX in mind, but this doesn't necessarily mean that you are restricted to using this as back end of your prettyprinter. Through the way that Pretzel was designed, it is quite easy to extend it to work with any other markup formatting system. However, if you are opting for immediate screen output or things like HTML, you're not lost either. First, we'll see how to handle the first case, and then let's go and look at things like HTML.

3.6.1 Other Markup Formatters

Internally, all pieces of code that handle the direct output of characters or format commands to the output stream are encapsulated in a C++ class called **Output**. The interface and implementation of this class are described in the `noweb` source file `output.nw` in the `output` subdirectory of the Pretzel distribution. From looking at this file, you'll see that there is an abstract base class **Output** and that the standard class, which Pretzel prettyprinters use, is the **Latex_cweb_output** class[5], a subclass of **Output**. It is within this subclass that the actual translation of the format command *force* into the string "`\pretzelforce{}`" takes place for example.

If you are going for something like Troff, you can similarly derive a new subclass of **Output** (for example by copying the original class) and simply replace the text bits like "`\pretzelindent`" with the strings appropriate for Troff. Now, you have to instruct the generated prettyprinter not to use the default (**Latex_cweb-**) output, but to use your new one. For this sake, the second version of the *prettyprint* member function of the generated prettyprinter may be used (see section 3.2, "The Pretzel

[5] The name of this class results from the goal, to give a LaTeX `CWEB` touch to the output [57].

Interface"). Simply pass an object of your new output class to the function and the prettyprinter will use it instead. By deriving a new subclass, Troff and LaTeX prettyprinters may coexist peacefully.

3.6.2 Going for HTML

What if you don't have a real markup formatting system at hand and want to go for something in the range of a vt100 terminal, line printer or HTML page?

Bad news is that you'll have to do without the full blown format command set. Proper interpretation of the line breaking commands *break_space* and *opt*, the commands *backup* and *no_indent* cannot be guaranteed in these cases. However, good news is that the Pretzel distribution contains a derived **Output** class called **Ascii_output** that may be used to output prettyprinted code with this restricted command set to the screen (or to any device suitable for ASCII characters). The format commands that cannot be executed properly are still valid, but result in no actions when outputting code. See the file `asciioutput.nw` in the `output` subdirectory of the Pretzel distribution for details.[6]

[6] An example `noweb` prettyprinter that emits HTML code can be found in the `contrib/noweb/cee.html` directory of the distribution. See also chapter 4, "Pretzel meets `noweb`".

Chapter 4

Pretzel meets noweb

> "noweb is designed to meet the needs of literate programmers while remaining as simple as possible. [...] The primary sacrifice relative to WEB is that code is not prettyprinted."
>
> Norman Ramsey [48]

The noweb system [50] by Norman Ramsey is a simple tool for literate programming, a programming style first introduced and named by Donald Knuth in 1984 [31]. The main goal of this new style is to enhance the readability and understandability of a program's source code to an extent that makes it enjoyable to read. This section will explain, how the Pretzel system relates to this new "paradigm" [9] and how it can be especially usefull in conjunction with noweb.

This chapter explains how you can use Pretzel prettyprinters within noweb. It expects that you know what literate programming is about and that you have had a go at using the noweb system. The articles by Knuth [31], Bentley [4] and Denning [11] provide good and readable introductions to the field, while the paper by Cordes and Brown [9] discusses the paradigm and the book by Knuth [32] provides a collection of related articles. Concerning noweb, there is a lot of good introductory material available online (see for example the noweb home page [48]). As you will need noweb anyway to install and run Pretzel, it is a good idea to try a small example now if you are unaquainted with it. Late sections will need detailed knowledge about the concepts of noweb as explained in the original paper by Ramsey [50] or the ultimate reference to noweb's interior by the same author [47]. The noweb extensions of Pretzel are still very much experimantal and far from making noweb as powerfull as language dependent literate programming tools.

4.1 Prettyprinting in noweb – How it works

> "They [the prettyprinting filters] can perform either or both of two tasks:
>
> - choose fonts and glyphs to represent each source token
> - choose indentation and line breaks of your code
>
> I find these features of more cost than benefit (except possibly when preparing for book publication), but lots of people like them.
>
> Noweb takes the reasonable position that the programmer is the best judge of where to put the line breaks and how much to indent code. In some languages (Miranda, Haskell, awk, Icon), line breaks and/or indentation carry meaning, and to change them would be to change the meaning of the user's program."

Norman Ramsey [52]

A primary design goal in developing noweb was to have a literate programming tool that was simple enough to learn easily, and also suitable for easy modification and extensions. This second goal was achieved by it's "pipelined architechture" [47]. Internally noweb converts the input files into a stream of items which — like in a pipeline — are squeezed through special programs called *filters* that perform simple transformations on these data items. The format of the data items in the pipeline is called the "noweb pipeline representation" which interested readers can study in detail elsewhere [47]. Devising a prettyprinter for noweb means building a prettyprining filter for it. Such a filter can then be inserted into the pipeline.

4.1.1 A noweb Prettyprinter for C

In the contrib/noweb/cee.latex you can find the definitions of a simple prettyprinter for C. Change into this directory and look at the files cee.ft and cee.fg. They are normal Pretzel input files that have been modified slightly to work with noweb. Before we look at these modifications, let's try to build a noweb prettyprinting filter first. To do this, you can use the -n option of pretzel-it in the following way:

```
pretzel-it -n cee prettycee
```

This will produce a noweb prettyprinter called prettycee in the current directory.[1] This filter can then be sent into action using the -filter option of noweave. For an example, type:

```
noweave -delay -filter prettycee ceetest.nw > ceetest.tex
```

This will mangle the noweb input file ceetest.nw and output LaTeX source to ceetest.tex. Now run it through LaTeX and have a look at the result.[2] What do you think?

4.1.2 A noweb Prettyprinter for Java

A pretty good Pretzel prettyprinter that works with Java and noweb can be found in the contrib/noweb/java.latex/ directory of the Pretzel distribution. It features a fine tuned prettyprinting grammar that is quite robust (contributed in core by Lee Wittenberg), correct typesetting of class names (in bold, as god ment them to be) and rudimentary automatic indexing facilities.

After installing Pretzel and making the file prettyjava (by simply typing 'make'), try the filter on Java files of your own. The command to use then is:

```
noweave -delay -filter prettyjava -index file.nw > file.tex
```

Note that the -filter switch comes before -index. To get this stuff typeset correctly, your file needs to access the pretzel-noweb.sty LaTeX style file (which gets installed when Pretzel gets installed). For this reason, you need to input it withing your document. The frames of my noweb files always look like this:

[1] The noweb extensions of Pretzel must have been installed on your system in order for this to work. See the top README file of the Pretzel distribution for details.

[2] People who are interested in the internals of this operation should consult the file pretty.nw which contains the noweb prettyprinter API and the file nowebpretzelpp.nw which contains a description of the interface between Pretzel and noweb. Both can be found in the directory contrib/noweb/general of the Pretzel distribution.

```
\documentclass{article}
\usepackage{noweb}
\usepackage{pretzel-noweb}
%
\begin{document}

... % rest of the noweb file

\end{document}
```

Note that you still have to include the `noweb.sty` package before accessing `pretzel-noweb.sty`.

This prettyprinter shows, how it can be made possible to include explicit format commands within the code a la `CWEB`. For example you can say

```
a:=1; //@cancel
```

and the *force*d line break that follows every statement will be canceled at this place. The only other format command that is possible here is *force*, which can be coded as `//@/` (a little like `CWEB`) or `//@force`, but in practice all other format commands can easily be added. Note that the format commands are comments in Java so they don't bother the tangling process.

4.1.3 Writing Prettyprinting Grammars for `noweb`

> "The most significant downsinde of not using `CWEB` is lack of code prettyprinting; however, after prettyprinting code for quite some time I got tired of it and I am now a nuweb minimalist."
>
> Przemek Klosowski [27]

Now, how do the `cee.ft` and `cee.fg` files differ from their non-`noweb` counterparts? From your point of view, consider every code chunk as a seperate input to the prettyprinter you write. So the only things to note when building a prettyprinter for `noweb` is that you may have arbitrary code fragments instead of full blown programs or functions, and that chunk uses may appear anywhere within the input code.

Chunk uses come as individual lines into the prettyprinter. These lines are in raw pipeline representation, i.e. look something like this:

```
@use Foo Bar
```

The scanner must detect these lines and basicly must wrap them up into a token that doesn't get changed. Thus, the formatted token file contains a line like this:

```
^"@use\ ".*          CHUNK  { "\n" ** "\n" }
```

The '`^`' ensures that the input matches from the beginning of the line. The two newlines in the attribute definition will result in the line being passed out of the prettyprinter unchanged. (The prettyprinter internally accumulates the code lines of an entire code chunk and then prettyprints it into a string. Then he cuts the prettyprinted code into lines before inserting it into the pipeline again. Lines that don't start with an '`@`' are prefixed with "`@text␣`".) This chunk token now can be used within the formatted grammar.

Because such a chunk token may appear everywhere in the text, it would be nice to have some rule that will stick a chunk token to any token that precedes it. As we have no possibility of using wildcards in token names, we use a simple trick: at the end of the grammar there is a rule like this for every possible token name:

```
^"@use\ ".*     CHUNK   { "\n" ** "\n" }
^"@".*          IGNORE  { "\n" ** "\n" }
```

The special `IGNORE` tokens result from another line in the formatted token file. This line reads as follows and is placed behind the line that matches chunk uses:

```
^"@".*          IGNORE  { "\n" ** "\n" }
```

As the prettyprinter input may contain other lines of internal `noweb` representation code (they all start with '`@`'), they need to be passed through the prettyprinter untouched too. This is what these two rules are good for.

Note that the `ceetest.nw` file includes the `noweb.sty` LaTeX style file as well as the file `pretzel-noweb.sty`. It is important that the `noweb` style comes before the `pretzel-noweb` style, because the latter redefines a macro from the prior (see also section 4.2).

4.1.4 Debugging

You may debug the prettyprinting filter by setting an environment variable called `PRETZEL_NOWEB_DEBUG` to the value "`on`". Debug information will be turned off again when you `unset` the variable. This is a simple, but easy to use method of controlling debug output. The output is much like the one explained in section 2.3.5 on page 19, and the tips and tricks from section 3.1.4 come in handy too. Suggestions for improvements are welcome.

I regularly run the parsing within Emacs (using the shell command possibility) ans then can easily search the debug output and follow how the tokens are put together.

4.1.5 Making the Best Use of It

> "The one area where language dependent LP tools are ahead is identifier indexing, but I suspect that for C++, this is an almost impossible task anyway."
>
> Matthias Neeracher [42]

Having understood how `noweb` prettyprinters differ from normal Pretzel prettyprinters, it is easy to convert files in either direction. But now, we also get language dependent information within the prettyprinter nearly for free and we can for example start to build an index of identifier definitions and uses.

For example, in the formatted token file, you could write:

```
[a-zA-Z][a-zA-Z0-9_]*    ID
                         {"%\n@index use " + ** + "\n"
                          "{\\it "  [escaped_underlines(yytext)] "}" }
```

Here, every identifier will finally be preceded by a line that will tell the indexer of `noweb` about the appearance of the identifier in this chunk. See the "`noweb` Hacker's Guide" [47] for details on the keywords that are allowed. But note, that if you're doing it this way the `-index` option of `noweave` has to appear *after* the `-filter` option on the command line.

4.1.6 Some Naming Conventions

If you are looking at example files of `noweb` prettyprinting filters, it's quite handy to notice some conventions that I have followed. The formatted grammar and the formatted token file should carry a name as a clear indication what programming

language they contain (e.g. `cee.ft` or `java.fg`). The prettyprinting filter for `noweb` that is produced should be called `pretty...` with the dots replaced by the programming language name (e.g. `prettycee` or `prettyjava`).

In the development directories you'll find special `noweb` files meant to test the prettyprinters, not to be of any programming use. They include most of the syntax constructs of the programming language in question. These files are very instructive when building a new prettyprinter for noweb or testing one that you have changed.

4.2 Problems

> "I think the problem here is not so much that prettyprinting is inherently bad as that many languages don't benefit from it so much. ALGOL and PASCAL, I believe, benefit quite a bit from appropriate prettyprinting. These are languages where you write out a lot of whole words: 'begin', 'end', 'do', 'then' ... You get the idea. Languages like C and Icon depend a lot on symbolic notation. For the large part, all `CWEB` does is replace one set of symbols with another that some people happen to prefer. But, still, `CWEB` puts reserved words and defining words in boldface and identifiers in italics; what of that? In PASCAL, doing that helps mark out the form of a construct. In C, on the other hand, the constructs are delimited by things like parentheses and braces, and so putting things in different typefaces doesn't help that much. OK, `CWEB` indentifies, but you don't need weave to do that; Emacs can do that for you, or you can do it yourself, and the web source will benefit from it. The main thing that I might want from `noweb` but don't get is typesetting of comments, and I can live without that because I don't need a great many comments."
>
> Barry Schwartz [58]

Handling of comments in prettyprinting remains an unsolved problem to date (see section 5.2.5). This is partially due to the fact that that comments can crop up anywhere in the code and that they themself can again contain quoted code. With `noweb` this problem isn't so severe, because chunks mostly do not contain a lot of comments as they are moved to the documentation parts of the literate program. But still a nice way of handling comments needs to be found.

Also, quoted code in documentation chunks isn't prettyprinted. This is partially a deficit of the prettyprinter API and partially a structural problem. One would need two distinct prettyprinters to handle normal chunk code and quoted code seperately because you don't want to have forced line breaks within documentation.

Another problem arises from the `noweb.sty` LaTeX style. The environment for setting code naturally is rather restrictive and a macro needs to be changed in order to allow the Pretzel macros to work correctly.[3] All this signals, that Pretzel an `noweb` have not become real friends yet; they still have to work on their relationship.

[3] I'm not a TeX freak after all. Thanks to Lee Wittenberg for his help on this.

Chapter 5

On Prettyprinting

> "`noweb` does automatic indexing and cross-referencing for some languages, including C, Icon, TEX, and Standard ML. Some people have made it prettyprint other language, like Icon and Object-Oriented Turing. I haven't been overwhelmed by the results, but then I'm notoriously difficult to convince of the value of prettyprinting."
>
> Norman Ramsey [51]

The term *prettyprinting* means the rearrangement of a program's source code to illuminate it's logical structure and thus enhance readability. The term goes back to Henry Ledgard and his "Programming Proverbs" [35]. Prettyprinting has become a field of interest because many of todays popular programming languages are so called "free-format" languages, where there are basically no column-position or line-boundary restrictions on statements, declarations, or comments.

As a matter of fact, formatting and all other aspects of readability depend heavily on personal taste and skill. This has lead to a wide range of suggestions and standards concerning the formatting of programming languages, especially for the PASCAL [23] language [1, 2, 3, 6, 10, 16, 18, 20, 34, 37, 45, 40]. The consistency of these proposals has also enabled construction of automatic formatting algorithms and tools that are usually called *prettyprinters* (or *indenting programs*). In the literature general examples of prettyprinters have been presented among others for the languages ALGOL [39, 59], PL/I [8], LISP [15, 19, 65, 66], Ada [43], and of course for PASCAL [21, 22, 3, 67, 1].

However, technological advance in the area of automatic typesetting has led to a set of powerful formatting tools called *typesetters*, or *typesetting systems*, of which TEX [30] is maybe the most apparent to computer scientists. These tools allow the preparation of documents in high quality suited for publication. Among the features of these systems are automatic line- and page-breaking, powerful macro-processing facilities and convenient devices for typesetting mathematical formulas. Seen in this light, automatic program formatting is only a simple instance of automatic typesetting [55, p. 652]. So prettyprinting algorithms that rely on professional typesetters can be simpler and produce better results at the same time, because a lot of the formatting problems (e.g. alignment, line-breaking, typesetting mathematical formulas) can be taken on by the typesetter.

In this chapter deals with the general issue of prettyprinting. It places a wider view on the field and tries to locate the Pretzel system within the latest research.

5.1 Prettyprinting with Format Commands

> "Well, I must confess, I was very close to throwing the whole `CWEB` away

> and switching to something without any prettyprinting, like `nuweb` or `noweb`. I installed `nuweb`, rewrote the `Matrix2D` example into it, ran it through and – returned back to `CWEB`. I probably got spoiled by the nice-looking output it produces – almost always."
>
> Jan Dvorak [12]

The prettyprinting method used by Pretzel can be called *prettyprinting with format command primitives* and goes back to a similar method used by Knuth in his original `WEAVE` system for PASCAL (see section 5.2 for details). Like most other prettyprinting algorithms this method functions similar to a *compiler*, i.e. an input file is processed and translated into an output file according to special rules. Here, the input file is the patch of source code that shall be prettyprinted and the output file is the text suited for the typesetter. The translation rules represent the special way in which source code should be formatted.

The main assumption is that we have a text formatter that uses *control sequences* (or *tags*) inside the actual text body for specifying all the different ways of formatting (such as fonts, indentation, spacing, etc.). Another term for this way of formatting control is *in-text procedural markup* and systems using this technique are commonly called *document compilers* (though this might only refer to the way they work, not to viewing text processing as programming). Common examples of such typesetting systems are TEX and the UNIX tools Nroff/Troff. During the processing of the input the only thing it does is to enrich the stream of incoming information with special control sequences that I call *format commands* (or *format command primitives*). These commands are tags that are left for the typesetter to interpret. The set of format commands used by Pretzel is basically the initial set used by Knuth. To understand this procedure, see chapter 2. The original documents by Knuth [29] and Knuth and Levy [33] present the algorithm in (in)formal detail.

5.2 A Short History of Prettyprinting

> "Nevertheless I strongly disagree with people saying that pretty-printing is not worth the effort for anyone in general; this depends very much on ones particular situation and purpose. In some cases pretty-printing may even be the main reason to opt for literate programming."
>
> Marc van Leeuwen [61]

In this section I will give a brief chronological overview over the area of prettyprinting, try to structure the field, and will show, how the present system fits into it.

5.2.1 Historical Notes

The prettyprinting of code has a long tradition that not only originates from the compulsion to typeset programs for publication. The readability or even the sheer beauty of a prettyprinted algorithm have also been a key motivation to this part of computer science.

The first people to call special attention to formatting issues were probably Peter Naur, Myrtle Kellington and William McKeeman. While Myrtle Kellington, as executive editor of ACM publications, helped to develop high quality programming-language typography standards, Naur was the first to include such formatting standards into his report on the ALGOL 60 language [41] and McKeeman was the first to present a prettyprinting algorithm for it [39]. This reaches back to the early 60s. In fact, McKeeman's algorithm is the first actual prettyprinter to be found in the literature. Its purpose was

> "[...] to edit ALGOL 60 text that is difficult to read because, for example, the ALGOL has been transcribed from printed documents, or written by inexperienced programmers [...]"

Later, other prettyprinters were presented for PL/I [8], LISP [15], and finally PASCAL [21]. These systems simply looked at the input text searching for special keywords. When encountering a keyword (which could be a **begin** or an opening brace, for example) a certain action would be triggered which would result in a line break, change of indentation, etc.

In these first approaches to the issue of prettyprinting the basic actions that prettyprinting includes were already visible. These concern (in order of importance):[1]

- indentation and folding (i.e. determining line breaks) of source code
- the additional spacing of syntactic constructs (like expressions)
- using additional typographic means (e.g. different fonts)

The main disadvantage of the early systems was their language dependence. The style of prettyprinting was hard-wired into the system, which was basicly a big "case" switch over all known constructs that were to be treated specially. The main advantage of them, however, was their simplicity and their error handling capabilities (which is also the reason why such and similar tools are still widely used today). But it was soon clear that this approach wasn't adequate and that structural changes had to be made to the concept.

In their paper titled "A One-pass Prettyprinter" [19] Hearn and Norman introduce a fundamentally new idea into this area. They simplify the whole concept of prettyprinting by introducing structure.

> "The new method that we propose here will be described in terms of a pair of coroutines. One of these will be responsible for producing a stream of characters that represent the program being printed, the other makes decisions about how these characters should be displayed." [19, p. 52]

This is a fundamental distinction, namely that between *formatting policy* and *formatting algorithm* [2] and is a vital step on the road to language independence.[2]

A problem that arises from this separation is: how do the two coroutines communicate? Hearn and Norman use a simple FIFO buffer in which the first coroutine inserts text and "special markers" that indicate its decisions on the policy. These markers were mainly special blanks that indicate possible line breaks.

> "The markers will contain enough information for the formatting process to discover what level of indentation would be appropriate to use were a line break to be inserted at that point."

This resembles a special kind of 'communication protocol' and is the first hint to a thing that I have called 'format command primitive' throughout this chapter. Other authors [22, 44, 65, 5, 56, 55, 24] have taken up this idea and have introduced different sets of format commands that change and increase the communication facilities between these two processes. The two most elaborate to date are those of Rubin [56] and that of Knuth and Levy [33], that is used here.

[1] In the late 70s there was one additional point on this list, namely the introduction of 'connector lines' into the prettyprinted output [7, 46]. But this method hasn't found too much support since.

[2] This idea however has also introduced a slight blur in the terminology, as some authors refer to prettyprinting as consisting only of the actual "printing" on paper (i.e. the formatting algorithm) [44, 24] and others still mean the entire process.

The separation introduced by Hearn and Norman is an example of 'separation of concerns': the language dependent parts of prettyprinting are separated from the actual typesetting issues that arise, when putting text to paper. Blaschek and Sametinger [5] call this a distinction between "language dependent front end and language independent back end".

5.2.2 The Language Dependent Front End

The earliest suggestions how to build the language dependent front end go back to Oppens fundamental paper on prettyprinting [44]. Though his main concern is the language independent back end, he makes suggestions about the "preprocessor" that should drive it. The main idea is to make a full syntactic parse of the prettyprinted code and to use the resulting parse tree to drive the prettyprinter. The format commands are either assumed to be implicit in the tree (i.e. every single branch is treated as a structural block) or they are explicitly inserted into the tree during the parse. Oppen writes:

> "First, notice that the information needed by the prettyprinter can often conveniently be represented directly in the grammar [...]. We modify the grammar of the language to contain prettyprinting information as above, where [...] [the formatting commands] are nonterminals mapping only the empty string." [44, p. 475]

This aspect has at last enabled people to use formal methods to describe layout rules. Mateti's paper [38] is another approach in the same direction and is the first to adapt this method to PASCAL.[3] The fact that rules for layout and grammar are often closely related [36] also is a strong indication that this method is in fact adequate. The trend towards formal specification in this area has indeed been "very fortunate" [68] and has also proved to be useful in other areas, such as syntax-directed editors [56].

To this end Rose and Welsh [55] have advocated the integration of format rules within the language syntax on the language design level. They state that "program format decisions [should be put] in the domain of the language's designer, rather than its several implementators or numerous users, which implies uniformly formatted programs of improved readability and therefore usability." [55, p. 651] Their idea does *not* try to impose rigid formatting rules on users of existing or future languages. Instead they propose a metasyntax and a set of guidelines that constrain and direct the language designer in the placing of format commands. These format commands include commands for indentation, as well as for line breaks and optional line breaks (so called "fold options"). The metasyntax and the guidelines only effect rather global issues of formatting but ensure consistency across language borders. They also present a folding algorithm that implements the format commands from their list.

This last approach by Rose and Welsh is currently the 'state of the art' in program layout issues. Woodman has published a thorough discussion of this scheme and has proposed refinements [68] that however do not alter the main points. It is an approach that in some way surpasses the area of prettyprinting in that it forces the formatting grammar to be equal to the language reference grammar. However, Knuth shows that a prettprinting grammar can be much simpler than a full-size language grammar, because it is able to treat semantically different contructs equally as they are formatted in the same way (e.g. **if**, **for** and **while** statements).

[3] It is interesting that Mateti's solution was the result of formalization (due to verification needs) and that Oppen's proposals emerged from the language independent nature of his approach.

5.2.3 The Language Independent Back End

The language independent back end implements the formatting algorithm. It actually makes the formatting decisions that the formatting policy has outlined and communicated to it via the format commands. The main concern of most algorithms is line breaking (or *folding*). The proposals by Rose and Welsh [55] and Oppen [44] have been the most influential ones on other systems [65, 24, 5] in the literature. However, in both articles the authors refer to the future of their proposals for the formatting algorithm:

> "The changes advocated must be seen in the light of current technological advances. Automatic program formatting is a very simple instance of automatic typesetting, as that provided by Knuth's TEX system for technical or mathematical text." [55, p. 652]

> "It [the algorithm] is not, however, as sophisticated as it might be, and certainly cannot compete with typesetting systems (such as TEX) for preparing text for publication." [44, p. 466]

As the `WEB` system shows, TEX has been used as a back end for a prettyprinter, but it has remained the only one that I am aware of. This leads to the question, why so little prettyprinters faciliate typesetters as back ends for their output? This question is very striking, as typesetters offer powerful capabilities, and for example, TEX's line breaking algorithm has been advocated to be very suitable for exactly this task [28].

Oppen gives an answer to this in his paper:

> "However, it [the algorithm] seems to strike a reasonable balance between sophistication and simplicity, and to be appropriate as a subcomponent of editors and the like." [44, p. 466]

It is surely the question whether you are prettyprinting on screen or preparing a publication. But the other part of the answer is maybe not so obvious and has to do with the traditional fears in the programming community.

A lot of prettyprinters have been built as parts of larger programming environments and were constituent parts of this larger system. A lack of modularity in the design (as for example the fundamental distinction presented in these sections and visialized in figure 5.1) often makes the resuing of subcomponents impossible. The prettyprinters hide somewhere in the larger system and are difficult to extract.

Another point is that an old habit of programmers is their strive for independence. They do not want their systems to rely on other systems. But this habit is hopefully getting less frequent today.

5.2.4 The Set of Format Commands

The format commands passed between the front end and the back end of the prettyprinter are the only means to convey the formatting policy to the actual formatter (see figure 5.1 that visualizes the modularity of a prettyprinter). This means that every typesetting feature must be expressible in them. In conjunction with this work it is important to ask, if the command set provided in the current Pretzel system is sufficient to format every desired feature that we can possibly think of. The answer is clearly 'No!', as there still are open prettyprinting problems. We will have a look at them later. But just how sufficient is the command set?

First we'll look at whether it allows full control over line breaks and indentation. A detailed comparison shows that the command set used by Rose and Welsh is a

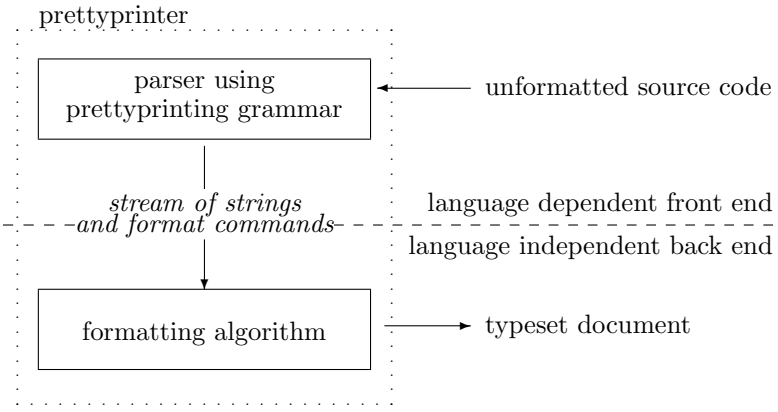

Figure 5.1: The module structure of modern prettyprinters

subset of Knuth's command set (*indent*, *outdent*, *force*, *breakspace*, *opt*).[4] Rose and Welsh in fact show by example, that any language syntax can be transformed into a formatted syntax that in itself is capable of specifying all desirable folds and changes of indentation. The main precondition for this transformation is, that the starting grammar is context free. None of the five basic commands is superfluous (as we have seen in the first part of this chapter) so we can call these basic commands the *necessary command set*.

The necessary command set enables the first point of the basic actions of a prettyprinter mentioned on page 43. The other two points (additional local spacing of syntactic constructs; using different fonts) are things that depend on the capabilities of the typesetter and are not so much a question of the format command set. Thanks to procedural markup, as long as the prettyprinter allows us to insert strings between any two tokens at lexical level we are able to handle these two points too. However, in Rubin's rather assembler like command set [56, p. 122] we see that the necessary command set still lacks full control over vertical and horizontal spacing. One can argue over what parts of this issue belong into the domain of the typesetter and what should be left to the user's control, but there surely is an obvious need to express additional vertical separation in terms of a simple format command. In our case this is the *big_force* command.

The other command that gives better control over horizontal space is *backup*. Both *backup* and *big_force* can be simulated by sequences of commands from the necessary command set.[5] So they are not necessary but convenient short-hands.

[4]The commands used by Rose and Welsh are denoted by the special symbols *m, o, r, i* and *s*. A stack is used to keep track of the indentation of the preceding lines. The symbol *m* pushes the current horizontal print position onto the stack and the symbol *o* pops the stack again. The metasyntax insists that every string produced from a non-terminal is bracketed by an *m*...*o* pair. As no *m* and *o* are allowed elsewhere in a rule body, the amount of indentation is always restored to the prevoius value during formatting.

The symbol *r* denotes a line feed and a carriage return to the margin that is stored in top of the stack, whereas *i* increments the top-of-stack margin value and then performes an *r*. The *s* symbol is used to emphasize strings produced from a rule of the grammar. It is only allowed at the beginning of a rule body and directs the folding algorithm to force line breaks before and after the string produced from this rule. Optional line breaks (so called *folding options*, denoted '[*i*]' or '[*r*]') can be inserted between the non-terminals in the rule body.

The most obvious analogy is the *i* symbol. It corresponds to the sequence *indent*, *force* and since every indent is followed by a *force* or made superfluous by a matching *outdent* you may transform every occurence of *indent* into an optional *i*. The *r* symbol, performed inside the *i*, denotes a *force* command. By popping the margin stack, the *o* symbol plays the role of the *outdent* command. The two fold options '[*r*]' and '[*i*]' correspond to *break_space* and *opt*.

[5]The sequence '*force*, *backup*' is equal to '*outdent*, *force*, *indent*' and '*big_force*' is something

necessary command set N	=	{ *indent*, *outdent*, *force*, *break_space*, *opt* }
convenient command set C	=	$N \cup$ { *backup*, *big_force* }
sufficient command set S	=	$C \cup$ { *cancel*, *no_indent* }

Figure 5.2: Naming of subsets of the command set.

We will call them, together with the basic commands the *convenient command set*.

Typesetting lines of code flushleft is not a feature commonly needed in modern structured programming languages, except in C/C++ when using preprocessor directives (and still they do not belong to the actual language itself). The *no_indent* command caters for them but it is hardly necessary in other languages, where alone the idea of excluding single lines from the overall indentation frame violates all rules of good program layout.

Finally the *cancel* command does not add any striking new features to the command set. It just allows more flexibility in the placement of format commands in the output. A well placed *cancel* command can obliterate a dozen rules of your prettyprinting grammar and thus can deflate prettyprinting grammars very much. So if you define sufficiency as incorporating convenience and flexibility we could call this whole set of presented commands a *sufficient command set*. This, however, naturally excludes the open problems mentioned in the next subsection. The two subsets of this sufficient command set are summarized in table 5.2.

5.2.5 Open Prettyprinting Problems

> "No there isn't, and there won't be [a method of vertically aligning statements in `CWEB`]. It is true that`CWEB` forgets your alignment, but you should realise that it also forgets *any* layout feature present in your source file (e.g., line breaks). Apart from the fact that it would be very hard to integrate any alignment processing into `CWEAVE`'s ordinary parsing operations, it is an impossible task for `CWEAVE`, since it knows nothing about the actual width of identifiers and other items. So the only possibility would be to transmit alignment instructions to TEX, but TEX typesets code in paragraph mode (breaking lines to the width of the page if necessary), and there is no way to mix that with an alignment."
>
> Marc van Leeuwen [63]

If you have ever tried to use a prettyprinter for your own code you will surely have come across problems concerning user control. Yehudai states, that

> "[...] it is not clear that any automated indentation scheme will be adequate, as I may choose different ways to lay out the same construct in different parts of my program." [69, p. 85]

Woodman gives an example for this point that he calls "adaptive combs" [68, p. 616], where a long **if/elsif** statement in Modula-2 is cited as:

if *longexpression1* **then**
  longstatement1
elsif *expr2* **then** *stmt2*
elsif *longexpression3* **then**
  longstatement3
else
  longstatement4

like '*force*, *null*, *force*', depending on the exact interpretation of the typesetter.

```
  end
```

Here the statement *stmt2* appears on the "tooth" of the comb, whereas the context rather suggests that it should appear between the teeth like all other statements. This is a question of *context sensitive formatting* and it is an open question, how to deal with this problem in terms of format commands.[6]

Another open problem falls into the same domain, but deals with horizontal spacing instead of line breaking. Blaschek [5, p. 701] points out that his system is not able to align constructs horizontally, as for example in:

```
with stmt^ do
    position    := pos
    decLabel    := label
    next        := nil
  end
```

This is a problem that no prettyprinter to date has mastered in an automatic fashion.

It is an open question whether these problems could be solved using intelligent algorithms that learn a layout style by example, like for instance the 'intelligent' prettyprinter by Winter and Cook [67]. But maybe the worst (and oldest) prettyprinting problem is how to typeset comments! Papers as early as those of Mohilner [40] and Jackel [22] address this topic and the core of the problem surely lies in the nature of comments. The paper "Programming languages should NOT have comment statements" by Kaelbling [25] and that by Grogono [17] strongly argue that at last comments should be treated equally as parts of the language, not as an add-on that is forgotten at first point during compilation. Grogono brings this to the point by stating that "Comments are second class citizens." [17, p. 80] They are mostly allowed anywhere in the source code and prettyprinters who want to preserve them in the output have severe problems doing that without destroying the program's format, because they have to be incorporated into the prettyprinting grammar.

There have been attempts to do this in a structured way, i.e. to divide comments into classes and to treat each class differently [53, 56, 13, 54]. Rose and Welsh distinguish between pre- and postcomments and state that comments must move with the syntactic elements that they refer to [55, p. 660]. Kaelbling even goes as far as to demand "scoped comments", i.e. comments that explicitly show to which part of the program they belong.

All these attempts have the same goal, but the problem of prettyprinting comments will only be solved, if they are treated as an equal part of a programming language and thus are included in the reference grammar.

[6] Woodman proposes the idea of "linked folds", i.e. a special set of format commands that force a number of consecutive folds to be treated equally, but doesn't elaborate on it.

Chapter 6

Future Work

> "In automatic formatting one should avoid interpreting layout features of the source text unless they are so special that the user can always avoid supplying such features inadvertently (e.g., one could imagine blank lines in the program code being automatically converted to '`@#`', but even there some programmers may feel that this cramps their input style). [...] it would be a good thing however if any popular layout style could be selected as an option to the prettyprinter (just like LaTeX allows selection of a document style independently from its contents) [...]"
>
> Marc van Leeuwen [62]

After having finally implemented and tested the Pretzel program I have noticed, that the program doesn't do very much at all, although it suits the specifications that belonged to this project. To enjoy beautifully formatted source code you still have to construct your prettyprinting grammar. As is seems, the work of fine tuning such a grammar that the prettyprinter is able to handle the last formatting detail you desire is quite tedious and time-consuming, especially if you start to build your grammar from scratch. But after constructing such a grammar for Pascal and applying it to a few everyday-example Pascal source codes, I was astonished how easy it was to change the looks of the prettyprinted text. I think that starting with a formal grammar of your favourite language and trying to enhance it and transform it into a prettyprinting grammar is surely a better way to reach your goal. I suppose that the Pretzel program could be the right tool to help you with this task.

However, since only few people have used Pretzel until today there will surely be a lot of things people miss when using the program. Here are a few things I have thought of already:

Allow `/*...*/` comments. In places where the user has an empty production in the formatted grammar file or an empty token definition in the formatted token file file, it would be nice to have a C like commenting feature with opening and closing delimiters. The '`//`' comment delimiter isn't nice if you want to add an attribute definition to an empty production.

Copy comments. It surely would make the generated flex and Bison files more readable, if Pretzel would copy the comments from the formatted token and the formatted grammar files into them.

Generate `%token` definitions. If one would impose the restriction to use only uppercase identifiers as terminal token names Pretzel could automalically generate the list of `%token` definitions needed to identify terminals in the formatted grammar file.

Change prettyprinting grammar format. It would be nice to be able to insert format commands directly into the grammar rules, like for example in:

`WHILE expr DO` *indent* `stmt` *outdent force* $\longrightarrow$ `stmt`

This is a point that effects fundamentals of the *build_pparse* function. Other parts of Pretzel are not involved. This is a change that would be good for new users, since this way of specifying the grammar is much more intuitive.

Include files for grammars. As parts of prettyprinting grammars occur frequently in many different programming languages (such as the formatting of expressions) it would be nice to be able to include files that contain these definitions with a simple command in the formatted token and formatted grammar files. On a more abstract level one could think of a notion of modules of grammars, i.e. parts of a grammar that can be called with arguments to suit local demands ("grammar templates"?). But it is still an unanswered question whether this is practical, because of the lack of method to specify the interface of such a module.

Enhance `noweb` support. This includes managing multiple prettyprinting filters and automatically selecting a filter for the right language (if you mix languages in `noweb` files). The `noweb` support seems the place where the most work can be done to get a practical system.

Suggestions are always welcome.

Chapter 7

Reference

> "I used to think that prettyprinting was the cats meow, but after becoming a bit accustomed to `noweb` I find that a `CWEB` printout looks somewhat like gibberish."
>
> Barry Schwartz [58]

This chapter contains a complete and detailed reference of the Pretzel system. This is information that mostly is also contained in the manual pages `pretzel(1)` and `pretzel-it(1)`.

7.1 The Concept of Pretzel

The concept of Pretzel is visualized in figure 7.1. From a formal formatting description of a programming language P, Pretzel generates a prettyprinting function that can be used to prettyprint source code in P. To get the actual prettyprinting program, you only have to supply a main program (or use the one that comes with Pretzel). If the way of prettyprinting needs to be changed, the user only needs to change formal parts in the input and use Pretzel again to get an enhanced prettyprinter.

Figure 7.2 on page 53 shows the generating process in slightly more detail. The figure shows that the program Pretzel doesn't generate a prettyprinter immediately; what it does is to produce code that can be turned into a C++ prettyprinting class with the help of two tools:

flex A POSIX.2 compliant lexical analyser generator.

Bison A POSIX.2 compliant parser generator. The version used with Pretzel has to produce C++ compliant source.

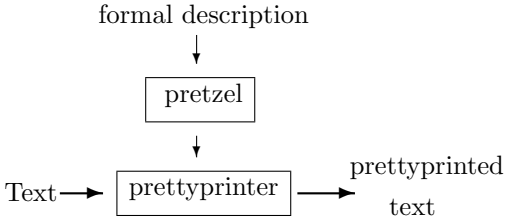

Figure 7.1: The concept of Pretzel.

Users that are already familiar with the basics of both tools will find it easier to read the following text because the way of specifying input to Pretzel is quite close to the ways used by them. However, this text aims at people who don't have such knowledge and will explain everything that is necessary.

Also visible from this figure is that Pretzel creates two things:

- A prettyprinting scanner class and
- a prettyprinting parser class (containing the actual prettyprinter).

The prettyprinting parser is the actual prettyprinter function that you normally want to get when using Pretzel. The prettyprinting scanner is used by the parser and takes care of the token formatting. Both parts are separate modules with a well defined interface and can be used individually. Hence, you can write your own scanner or parser if you really need features that the ones produced by Pretzel don't have.

7.1.1 The Input Files

The word "pretty" is very subjective and so programming language source code can be printed in a "pretty" kind of fashion in almost infinitely many ways, according to the taste and preferences of the user. If the task of prettyprinting is handed over to the computer, the user must specify in detail all of his wishes towards the appearance of the output of the prettyprinter.

To generate a prettyprinter, Pretzel needs two descriptions that it expects to find in two different files:

A formatted token file. This file should contain all information concerning the individual formatting of tokens (i.e. identifiers, reserved words, etc.).

A formatted grammar file. This file should contain a prettyprinting grammar, i.e. a context free grammar enhanced with formatting commands. This information is necessary to handle the more global aspects of prettyprinting that need information about the language context (i.e. indentation, line breaks, etc.).

The name of the first file might be a little misleading, since it doesn't contain formatted tokens. Instead it tells you what you will get from the definitions therein. A name like "token formatting file" might have been a little better, but the similarity in names between the two input files sounded good and seemed to outplay this small inconsistency.

The special formats of these two files are described later in section 7.2. The interface to the generated Pretzel classes is described in detail in section 3.2.

7.2 The Format of the Input Files

Now we turn to a vital thing for the user of Pretzel: The format of the Pretzel input files. Here the user has to put a formal description of the underlying programming language and (with the help of format commands) state how its text should be formatted.

We have seen that Pretzel takes two input files: The formatted token file and the formatted grammar file. We suggest to use the suffix '`.ft`' for a formatted token file and '`.fg`' for a formatted grammar file.

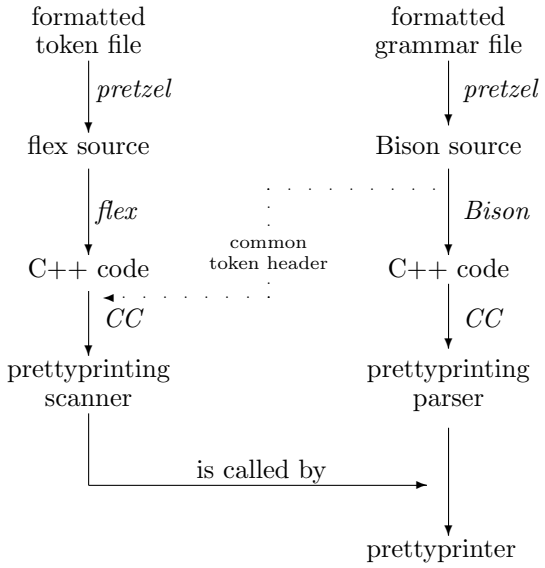

Figure 7.2: The current process of generating a prettyprinter with Pretzel.

7.2.1 The Formatted Token File

The formatted token file contains a list of token definitions with their corresponding "prettyprinted" form. The prettyprinted form of a token will be called an *attribute* or a *translation*.

The general outline of the formatted token file is

declarations
%%
token definitions

Normally, the *declarations* part is empty. You can put a general description of the file here (as a C comment) and redefinitions of the default interface go here as well (see section 3.2 for more).

The *token definitions* section of the formatted token file contains a series of token definitions of the form:

pattern *token* {*attribute*}

The *pattern* must be a valid regular expression (in terms of `flex`) and must be unindented. The *token* specifies the symbolic name of the token for the pattern and begins at the first non-whitespace character after the pattern. The token name must be a legal name for an identifier in PASCAL notation and *must be all in upper case*. (Underlines are allowed but not at the beginning of a word.)

The *attribute* for this token, that is it's prettyprinted form, consists of all text between the two curling brackets '{' and '}'. Attributes can be either simple strings (surrounded by double quotes) or format commands (like *force*, *indent*) or a combination of both joined together by an optional '`+`' sign. Attribute definitions can cover several lines and the starting '{' needn't stand on the same line as the token definition; however subsequent lines must be indented with at least one blank or one tab. Attributes can also contain C code. See section 3.1 or the manual page `pretzel(1)` for details.

If you define strings as part of an attribute definition, you have to specify them in a C kind of fashion, i.e. you can insert newlines and tabs with '`\n`' and '`\t`'.

But if you want to insert a backslash into a string, you mustn't forget to put two backslashes ('\\') into the input file. This is especially noteworthy if you are using TEX as typesetter, because TEX uses a backslash as a prefix for typesetting commands.

If the definition of the attribute is omitted Pretzel creates an attribute for this pattern by default. The default attribute consists of the string containing the text matched by the corresponding pattern.

The user himself may also refer to the matched text by using the sequence '**'. Thus

```
"foo"       BAR
"foo"       BAR     { ** }
"foo"       BAR     { "foo" }
```

all have the same meaning.

You can use a '|' sign as a token name; this signals that the current regular expression has the same token name (and also the same attribute) as the token specified in the following line (empty lines are ignored). An attribute definition behind a '|' is illegal. However you may specify regular expressions with neither a token name nor an attribute to give a default rule or to eat up whitespace.

The following examples are all legal token definitions (and please note the dot in the very last line):

```
[0-9]           DIGIT

"{"             OPEN            { "\\{" indent force }

[a-z][a-z0-9]*  ID              { "{\\it " + ** + "}" }

"function"      |
"procedure"     PROC_INTRO      { big_force ** }

[\t\ \n]        |
.
```

The declarations and the token definitions must be separated by a line containing only the two characters %%. So the shortest possible formatted token file is

```
%%
```

but this doesn't seem of any use, does it?

7.2.2 The Formatted Grammar File

In the formatted grammar file the user encodes the general prettyprinting grammar for the programming language. This is done by specifying a context free grammar of the language and by adding information about the creation of new attributes in every rule.

The formatted grammar file is the second and last input to the Pretzel program. Its general outline looks like this:

token declarations
%%
grammar rules

The *token declarations* section may be empty and the separator between the two parts of the file (`%%`) must appear unindented on a single line by itself. Before we look at these declarations, let's have a look at the grammar rules.

The *grammar rules* section contains the collection of rules of the context free grammar that can be accompanied by an attribute definition. A rule is specified by stating the resulting token, a colon and then the series of tokens which will be reduced by this rule. The rule is ended by a semicolon. A block definition in PASCAL for example might look like this:

```
block : BEGIN stmt_list END
;
```

Following the token list on the right side of the colon can be an attribute definition; this definition states, how the translation of the produced symbol is obtained from the tokens on the right side of the rule.

An attribute definition is bracketed amidst curling brackets '{' and '}' and can again consist of strings (in double quotes) and format commands or both joined together with '`+`'. But here you can also refer to the attributes of the tokens on the right side of the rule. This is done in a slightly awkward notation with a number that is preceded with a '`$`' dollar sign. The numbers refer to the order of appearance of the symbols on the right side of the rule. So '`$1`' refers to the first token of the rule, '`$2`' to the second, ...

Again attribute definitions are allowed to span several lines and strings must be specified in C manner. They can also contain C code as described in section 3.1.

For example, here again is the possible definition of a block in PASCAL, now with an example attribute definition:

```
block : BEGIN stmt_list END   { $1 + $2 + force + $3 }
;
```

The attribute of a `block` will therefore consist of the attributes of the `BEGIN` and `stmt_list` tokens, joined together with a *force* command and the translation of the `END` token.

The attribute definition may be omitted. If this is so, Pretzelwill by default form the attribute of the produced symbol from the simple concatenation of the attributes on the right side of the rule. For instance

```
stmt : block SEMI
;
```

means the same as:

```
stmt : block SEMI       { $1 + $2 }
;
```

Of course you may also have empty right sides of a rule (to produce things out of nothing) or simply concatenate two or more rules resulting in the same symbol with a '`|`'. So the following are legal rules:

```
stmt_list   :                       { force }
            | stmt_list stmt SEMI   { $1 $2 $3 force }
;
```

To end this subsection, we have to return to the *token declarations* section of the formatted grammar file. Here we have to insert a special line for every terminal token that appears in the grammar rules. These definitions are of the form '`%token` *tokenname*'. This part of the formatted grammar file is owed to Bison and should be removed in subsequent versions.

7.2.3 Comments and Code

There is a very simple way of putting comments into the formatted token and formatted grammar files. This is done in a C++ kind of manner by preceding the comment with a double slash ('`//`'). All characters between this sign and the end of the line are ignored by pretzel.

In both files you can put additional C++ code before and after the definitions/grammar sections. If you want to insert code at the end of your file, you have to put a second '`%%`' on a line by itself and put the code behind it. C/C++ Code before the definitions/rules section has to be tied in with a '%{', '%}' pair. Inserting extra code is interesting for people who want to call this code from within the attribute definitions. See section 3.1 for details.

7.3 Synopsis of `pretzel` and `pretzel-it`

7.3.1 `pretzel-it`

The shell script `pretzel-it` uses Pretzel to build a simple prettyprinter executable. It minimizes building a Pretzel prettyprinter to just one shell command.

You have to provide the same two input files to `pretzel-it` as to Pretzel. These two files are called the formatted token file (suffix `.ft`) and the formatted grammar file (suffix `.fg`). Both files need to have the same prefix. From this input, `pretzel-it` generates an executable prettyprinter.

To get to know the options, type

```
pretzel-it -h
```

at the command line. The full usage is:

```
pretzel-it [-iqvdnh] language ppname
```

Here's an explanation of the options:

-i Don't remove intermediate products of pretzeling.

-q Run quietly.

-v Verbose mode, print shell commands before invoking (for debugging).

-d Turn prettyprinter debugging features on by default (for debugging the prettyprinting grammar).

-h Print full usage message.

-n Noweb mode, will produce a prettyprinting filter `ppname` compatible to Norman Ramsey's `noweb` literate programming system. The filter can be inserted into the `noweb` pipeline using `noweave`'s `-filter` option.

See also the manpage `pretzel-it` and chapters 2 and 4.

7.3.2 `pretzel`

Pretzel is invoked by typing

```
pretzel
```

at the command line. The full usage of Pretzel can be obtained using the `-h` option. It is:

```
pretzel [-qtgdh] [-o outfile] (prefix | file1 file2)
```

Here's an explanation of the options:

-q Run quietly (no screen output).

-t Process formatted token file only.

-g Process formatted grammar file only.

-d Run in debug mode (i.e. print out debugging information on the screen while running).

-h Show full usage.

-o outfile Names of the produced output files begin with "outfile".

The options `-t` and `-g` are mutually exclusive, i.e. you can't choose both at the same time.

The command line parameters have different meanings depending on whether one ore two names are given. If there is only one parameter, it specifies the prefix of both formatted token and formatted grammar files. The suffixes '`.ft`' and '`.fg`' are assumed. But if there are two parameters at the command line, Pretzel will take the first as full name of the formatted token file and the second as full name of the formatted grammar file. In this case, the output files will get a default name. The output files will have endings '`.l`' (token file) and '`.y`' (grammar file).

Bibliography

[1] M. Arab. Enhancing program comprehension: formatting and documenting. *ACM SIGPLAN Notices*, 27(2):37–46, February 1992.

[2] P. A. Bailes and A. Salvadori. A semantically-based formatting discipline for Pascal. *Software — Practice & Experience*, 14(3):235–251, March 1984.

[3] R. M. Bates. A Pascal prettyprinter with a different purpose. *ACM SIGPLAN Notices*, 16(3):10–17, March 1981.

[4] Jon Bentley. Programming pearls—literate programming. *Communications of the Association for Computing Machinery*, 29(5):364–369, May 1986.

[5] G. Blaschek and J. Sametinger. User-adaptable prettyprinting. *Software — Practice & Experience*, 19(7):687–702, July 1989.

[6] R. Bond. Another note on Pascal indentation. *ACM SIGPLAN Notices*, 14(12):47–49, December 1979.

[7] H. M. Clifton. A technique for making structured programs more readable. *ACM SIGPLAN Notices*, 13(4):58–63, April 1978.

[8] K. Conrow and R. G. Smith. NEATER2: A PL/I source program reformatter. *Communications of the ACM*, 13(11):669–675, November 1970.

[9] David Cordes and Marcus Brown. The literate-programming paradigm. *Computer*, 24(6):52–61, June 1991.

[10] J. Crider. Structured formatting of Pascal programs. *ACM SIGPLAN Notices*, 13(11):15–22, November 1978.

[11] Peter J. Denning. Announcing literate programming. *Communications of the Association for Computing Machinery*, 30(7):593, July 1987.

[12] Jan Dvorak. Re: `CWEB` & C++ trouble with 'const'. Posting in `comp.programming.literate`, November 1995.

[13] P. Fritzson. Adaptive prettyprinting of abstract syntax applied to Ada and Pascal. Research report, University of Linköping, Sweden, 1983.

[14] Felix Gärtner. Pretzel home page. WWW URL: `http://www.iti.informatik.th-darmstadt.de/~gaertner/pretzel`.

[15] I. Goldstein. Prettyprinting, converting list to linear structure. Technical Report 279, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., 1973.

[16] P. Grogono. On layout, identifiers and semicolons in Pascal programs. *ACM SIGPLAN Notices*, 14(4):35–40, April 1979.

[17] P. Grogono. Comments, assertions, and pragmas. *ACM SIGPLAN Notices*, 24(3):79–84, March 1989.

[18] G. G. Gustafson. Some practical experiences formatting PASCAL programs. *ACM SIGPLAN Notices*, 14(9):42–49, September 1979.

[19] A. C. Hearn and A. C. Norman. A one-pass prettyprinter. *ACM SIGPLAN Notices*, 14(12):50–58, December 1979.

[20] R. Heckert. A Pascal indentation philosophy. *Computer Language*, pages 37–39, September 1985.

[21] Jon Hueras and Henry Ledgard. An automatic formatting program for PASCAL. *ACM SIGPLAN Notices*, 12(7):82–84, July 1977.

[22] M. Jackel. A formatting parser for Pascal programs. *ACM SIGPLAN Notices*, 15(7–8):58–63, July–August 1980.

[23] K. Jensen and N. Wirth. *PASCAL — User Manual and Report*. Springer, third edition, 1985.

[24] M. O. Jokinen. A language-independent pretty printer. *Software — Practice & Experience*, 19(9):839–856, September 1989.

[25] M. J. Kaelbling. Programming languages should NOT have comment statements. *ACM SIGPLAN Notices*, 23(10):59–60, October 1988.

[26] Tim Kientzle. When to use prettyprinting. Posting in `comp.programming.literate`, October 1994. Correct subject header might be different. Date here: 6 Oct 1994.

[27] Przemek Klosowski. Re: I want to produce postscript output from cweb. Posting in `comp.programming.literate`, November 1996.

[28] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software — Practice & Experience*, 11:1119–1184, 1981.

[29] Donald E. Knuth. The `WEB` system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.

[30] Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, Massachusetts, 1983.

[31] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[32] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[33] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, Reading, MA, USA, 1993.

[34] H. Ledgard, A. Singer, and J. Hueras. A basis for executing PASCAL programmers. *ACM SIGPLAN Notices*, 12(7):101–105, July 1977.

[35] Henry F. Ledgard. *Programming Proverbs*. Hayden, Rochelle Park, New Jersey, 1975.

[36] D. W. Leinbaught. Indenting for the compiler. *ACM SIGPLAN Notices*, 15(5):41–48, May 1980.

[37] D. Marca. Some Pascal style guidelines. *ACM SIGPLAN Notices*, 16(4):70–80, April 1981.

[38] P. Mateti. A specification scheme for indenting programs. *Software — Practice & Experience*, 13:163–179, 1983.

[39] William M. McKeeman. Algorithm 268. *Communications of the ACM*, 8:667–668, 1965.

[40] Patricia R Mohilner. Prettyprinting PASCAL programs. *ACM SIGPLAN Notices*, 13(7):34–40, July 1978.

[41] Peter Naur et al. Report on the algorithmic language Algol 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[42] Matthias Neeracher. Re: Simple lp experiment. Posting in `comp.programming.literate`, April 1996.

[43] D. Norris. An Ada prettyprinter. *Journal of Pascal and Ada*, 3(4):29–48, April 1984.

[44] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and System*, 2(4):465–483, October 1980.

[45] James L. Peterson. On the formatting of Pascal programs. *ACM SIGPLAN Notices*, 12(12):83–86, December 1977.

[46] J. Ramsdell. Prettyprinting structured programs with connector lines. *ACM SIGPLAN Notices*, 14(9):74–75, September 1979.

[47] Norman Ramsey. The noweb hacker's guide. included in the noweb distribution, also available via the Noweb home page [48].

[48] Norman Ramsey. Noweb home page. WWW URL: http://www.cs.virginia.edu/ nr/noweb/intro.html.

[49] Norman Ramsey. Weaving a language-independent `WEB`. *Communications of the Association for Computing Machinery*, 32(9):1051–1055, September 1989.

[50] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[51] Norman Ramsey. When to use prettyprinting. Posting in `comp.programming.literate`, October 1994. Correct subject header might be different.

[52] Norman Ramsey. Re: Prettyprinting in noweb (was: the underscore dilema). Posting in `comp.programming.literate`, April 1996.

[53] Frederic Richard and Henry F. Ledgard. A reminder for language designers. *ACM SIGPLAN Notices*, 12(12):73–83, December 1977.

[54] P. N. Rorbillard. Automating comments. *ACM SIGPLAN Notices*, 24(5):66–70, April 1989.

[55] G. A. Rose and J Welsh. Formatted programming languages. *Software — Practice & Experience*, 11:651–669, 1981.

[56] Lisa F. Rubin. Syntax-directed pretty printing — a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, SE-9(2):119–127, March 1983.

[57] Joachim Schrod. Latex cweb — a bundle that allows you to use latex as the documentation markup of your cweb program. WWW URL: `ftp://ftp.th-darmstadt.de/ pub/programming/literate-programming/ c.c++/cweb-sty-1.1.1.tar.gz`.

[58] Barry Schwartz. When to use prettyprinting. Posting in `comp.programming.literate`, October 1994. Correct subject header might be different. Date here: 5 Oct 1994.

[59] R. Scowen, D. Allin, A. L. Hillman, and M. Shimell. SOAP – A program which documents and edits Algol60 programs. *The Computer Journal*, 14(2):133–135, 1971.

[60] Marc van Leeuwen. When to use prettyprinting. Posting in `comp.programming.literate`, October 1994. Correct subject header might be different. Date here: 7 Oct 1994.

[61] Marc van Leeuwen. Differences between CWEB and WEB prettyprinting grammars. Posting in `comp.programming.literate`, February 1995. Correct subject header might be different.

[62] Marc van Leeuwen. On interpretation of layout features. Posting in `comp.programming.literate`, February 1995. Correct subject was different.

[63] Marc van Leeuwen. Alignment of assignments in `CWEAVE`. Posting in `comp.programming.literate`, March 1996. Correct subject header might be different (date and number here: 2986, 19 Mar 1996.

[64] Marc van Leeuwen. Prettyprinting in noweb (was: the underscore dilema). Posting in `comp.programming.literate`, April 1996.

[65] R. Waters. User format control in a LISP prettyprinter. *ACM Transactions on Programming Languages and Systems*, 5(4):513–531, October 1983.

[66] R.C. Waters. Using the new common LISP pretty printer. *Lisp and Symbolic Computation*, V(2):27–34, April–June 1992.

[67] K. Winter and C. Cook. A prototype intelligent prettyprinter for Pascal. *ACM SIGPLAN Notices*, 24(9):116–125, September 1989.

[68] M. Woodman. Formatted syntaxes and Modula-2. *Software — Practice & Experience*, 16(7):605–626, July 1986.

[69] A. Yehudai. Automatic indentation versus program formatting. *ACM SIGPLAN Notices*, 15(10):85–87, 1980.

Index

`/*...*/` comments, 49
`//` comments, 49
[, code delimiters, 23
2.0, 6

ACM, 42
Ada, 41
"adaptive combs", 47
adding code, 23
Additional spacing, 43
AIX, 6
ALGOL, 41, 42
Algorithm-policy distinction, 43
ASCII, 33
Assembler like command set, 46
attachments, 13
`attr.nw`, 23
Attribute class, 23
attribute definitions, 11
attributes, 13
Automatic typesetting, 41
available grammars, 19

Back end, 44
backup, 15
backup, 15, 47
Basic actions of prettyprinters, 43
Beauty, 42
big_force, 15, 46
Bison, 6, 11, 18, 32, 52
Blaschek, G., 44, 48
books, 18
breakspace, 14
build_pparse, 50

C, 17, 19
C, 16, 47, 55
C code in rules, 23
C preprocessor, 16
C++, 16, 47, 51, 56
cancel, 16, 37, 47
"case" construct, 43
christmas 1996, 6
Classes of comments, 48
code delimiters, 23
code in attributes
    summary, 25
colon, 11
combining rules, 11
Command set
    assembler like, 46
    convenient, 47
    sufficiency, 45
    sufficient, 47
Comments, 48
    in Pretzel input files, 56
Communication protocol, 43
`comp.programming.literate`, 6
Compilers
    and prettyprinters, 42
    for documents, 42
compilers, 18
complex languages, 16
complex tokens, 11
Connector lines, 43
Context free grammar, 46, 54
context free grammar, 11
context free grammars, 18
Context sensitive formatting, 48
context sensitive grammars, 18, 21
Control sequences, 42
control sequences, 13
controlling indentation, 13
controlling line breaks, 14
Convenient command set, 47
Conventions, 52
Cook, C., 48
Copy comments, 49
Coroutines, 43
create, 24
CWEB, 7
`CWEB`, 37

`-d` option of Bison, 30
`-d` option of `pretzel-it`, 20, 29
Darmstadt, 6
date
    up to, 6
debug off function, 29
debug on function, 29

debug print, 26
debugging grammars, 19
debugging mode, 20
Deflating prettyprinting grammars, 47
`-delay` switch, 36
Document compilers, 42
documents, own, 9
Dublin, 6
Dvorak, Jan, 42

Early prettyprinters, 43
Editors
    syntax directed, 44
Empty tokens, 49
ending code, 25
Equality, 48
error token, 20
escaped underlines, 24
everyday setting, 7
`example-frame.tex`, 9
Exceptions, 16
executable prettyprinter, 9
expectations, 9
Explicit format commands, 44

`.fg` suffix, 52, 57
FIFO buffer, 43
File formats, 52
Filename conventions, 52
`-filter` switch, 36
flex, 6, 11, 32, 52
flexdoc, 11, 32
Flexibility, 16
Folding, 43, 45
Folding algorithm (Rose and Welsh), 44
force, 13
force, 37
formal language theory, 18
Formal methods, 44
Format commands, 43
    backup, 15, 47
    big_force, 15, 46
    cancel, 16, 47
    explained, 16
    explicit, 44
    implicit, 44
    no_indent, 16, 47
    null, 16
    summary, 16
format commands
    additional, 15
    backup, 15
    breakspace, 14
    extra, 37
    opt, 15
Formatted grammar file, 52, 54
formatted grammar file, 10, 11
Formatted token file, 52, 53
formatted token file, 10
    placement of regular expressions, 11
formatting, 13
Formatting algorithm, 43
Formatting algorithms, 41
formatting instructions, 12, 13
Formatting policy, 43, 45
Formatting standards, 41
"free format" languages, 41
Free Software Foundation, 6
Front end, 44
`.ft` suffix, 52, 57

Gärtner, Felix, 19
Gärtner, Felix, 19
GNU `g++` Compiler, 6
GNU General Public License, 6
good news, 19
grammar rules, 13
grammar, context free, 11
grammars
    prettyprinting, 16
Grogono, P., 48
Guidelines (Rose and Welsh), 44

handling identifiers, 10
Hearn, A. C., 43, 44
History of prettyprinting, 42
Hmmm, 18
homepage
    Pretzel, 6, 17
Horizontal spacing, 48
HP-UX, 6
HTML, 32
hum, rattle and, 9

Implicit format commands, 44
In-text procedural markup, 42
Include files for grammars, 50
indent, 13
Indentation, 43
indentation, 13
Indenting programs, *see* Prettyprinters
`-index` switch, 36
Indexing with `noweb`, 36
indexing, automatic, 7
Inituitive grammar format, 50

install, 24
installing Pretzel, 6
Intelligent prettyprinter, 48
ITI, 6

Jackel, M., 48
Java, 17, 19, 36
join, 24

Kaelbling, M. J., 48
Kehr, Roger, 6
Kellington, Myrtle, 42
Kientzle, Tim, 7
Klosowski, Przemek, 37
Knuth, Donald E., 35, 42–44, 46
    command set by, 42

`.l` suffix, 57
Language (in)dependence, 44
language definition grammars, 18
Language dependent front end, 44
Language independent back end, 44, 45
`languages/examples`, 7
LaTeX, 9, 32
Latex cweb output, 26
Ledgard, Henry F., 41
van Leeuwen, Marc, 5, 16, 20, 32, 42, 47, 49
Levy, Silvio, 43
Lexical level, 46
line breaks, 14
Linked folds, 48
Lisp, 41, 43
literate programming, 35
lookup table, 24

Markup, 42, 46
markup, 13
match iput patterns, 10
Mateti, P., 44
McKeeman, William, 42
Metasyntax (Rose and Welsh), 44
Modula-2, 47
Modularity, 45
modules, 5
Mohilner, Patricia R., 48
muliple modules, 31

`-n` option of `pretzel-it`, 36
naming conventions, 38
Naur, Peter, 42
Necessary command set, 46
Neeracher, Matthias, 38
newlines, 26
newsgroup, 6
"No", 45
no_indent, 16, 47
Norman, A. C., 43, 44
noweb, 6, 35, 36
    prettyprinter API, 36
    problems, 39
noweb.sty, 39
`noweb.sty`, 37
Nroff, 13, 42
null, 16

Omissions, 49
Oppen, Derek C., 44
opt, 15
outdent, 13
own documents, 9

P, 51
Parse tree, 44
parser generator, 18
parsing, 18
Pascal, 19
Pascal, 18, 41–44, 49, 53, 55
Personal taste, 41
PL/I, 41, 43
placing regular expressions, 11
Policy-algorithm distinction, 43
POSIX, 52
"postcomments" (Rose and Welsh), 48
`Pparse` class, 28
`PPARSE_NAME` macro, 28, 31
"precomments" (Rose and Welsh), 48
Preprocessing of format commands, 16
Preprocessor, 16
"preprocessor" for prettyprinting, 44
Prettprinting scanner, 52
prettyprint function, 28
Prettyprinter
    first, 42
    intelligent, 48
prettyprinter
    executable, 9
    generator, 5
Prettyprinters
    basic actions, 43
    early systems, 43
    for other languages, 41
    running on keywords, 43
Prettyprinting, 41
    history, 42
    language (in)dependence, 44
    problems, 47
prettyprinting

grammars, 16
history, 12
idea, 13
modules, 5
with format commands, 13
Prettyprinting grammar
include files, 50
Prettyprinting grammar, 16, 49, 50
and comments, 48
deflating, 47
prettyprinting grammar, 11
watching parse, 20
prettyprinting grammars, 18
available, 19
prettyprinting module, 30
Prettyprinting parser, 52
prettyprinting parser, 27
prettyprinting parser interface, 28
Prettyprinting problems
worst and oldest, 48
prettyprinting scanner, 27
prettyprinting scanner class, 27
Pretzel
currect release, 6
example output, 9
file extensions, 7
history, 6
homepage, 6, 17
input files, 7, 10
installing, 6
interface, 26
output, 9
prettyprinting method, 10
Pretzel, 49
concept, 51
options, 57
Pretzel
obtaining, 6
ultimate source, 5
`PRETZEL_INCLUDE` environment variable, 30
`pretzel-it`, 9
option `-d`, 20
`pretzel-noweb.sty`, 37
Procedural markup, 42, 46
Program formatting, 41
Pscan class, 27
`Pscan.h` header file, 27
`PSCAN_NAME` macro, 27, 30
`ptokdefs.h` header file, 30

Ramsey, Norman, 6, 35, 36, 41
rattle and hum, 9
`README`, 6
recursion, 11
reducing tokens, 11
reference grammars, 18
regular expressions, 10
placement in formatted token file, 11
restrictions, 21
restrictions of Pretzel, 19
Reuseability, 45
Rigid formatting rules, 44
robust grammars, 20
Rose, G. A., 44–46, 48
RS6000, 6
Rubin, Lisa F., 43, 46

Sametinger, J., 44
Schrod, Joachim, 6
Schwartz, Barry, 22, 39, 51
"scoped comments" (Kaelbling), 48
scratch, 17
Second class citizens, 48
semicolon, 11
Separation of concerns, 44
setting, everyday, 7
`simpas.ft`, 7
`simpas.fg`, 7
`simpaspp`, 9
`small-example.tex`, 9
Spacing
additional, 43
horizontal, 48
horzontal, 46
vertical, 46
"special markers", 43
SPIDER, 18
standard input, 9
standard output, 9
starting code, 25
State of the art, 44
students, 6
Sufficiency of command set, 45
Sufficient command set, 47
Summary of format commands, 16
symbolic names, 10
syntax error, 20
Syntax-directed editors, 44

Tags, 42
tags, 13
Taste
personal, 41
TEX, 13, 41, 42, 45
text formatter, 13
tips and tricks, 26

`%token` declarations, 11
`%token` definitions, 49
token header file, 30
tokens, 10, 11
    symbolic names, 10
Tooth, 48
Tradition, 42
Tricky details, 16
Trinity College, Dublin, 6
Troff, 13, 42
Typesetters, 41
Typesetting systems, 41
Typesetting comments, 48

Uhr, Holger, 6, 19
ultimate source, 5
UNIX, 6, 13, 18
UNIX, 42
USENET, 6
User control, 47
user control
    full, 5
using Pretzel output, 9

vertical line, 11

Waldschmidt, Helmut, 6
watching the parse, 20
`WEAVE`, 42
WEB, 18
`WEB`, 45
Welsh, J., 44–46, 48
Winter, K., 48
Wittenberg, Lee, 6, 19, 36
Woodman, M., 44, 47
writing grammars, 17

`.y` suffix, 57
Yehudai, A., 47
yytext, 24