Department of Computer Science

Fachhochschule Dortmund

University of Applied Sciences and Arts

Project Thesis

Parse-free code formatting

Tim Seckinger

2018-04-11

Contents

Abstract 1

1 Introduction 2

2 An ordinary formatting procedure 4

2.1 Lexing 4
2.2 Parsing 5
2.3 Printing 7
2.3.1 Spacing 7
2.3.2 Indentation 7
2.3.3 Hard line breaks 8
2.3.4 Soft line breaks 8
2.4 Summary 9

3 Eliminating the parsing step 10

3.1 Potential benefits 11
3.1.1 Efficiency 11
3.1.2 Streaming 11
3.1.3 Adaptability 11
3.2 Drawbacks 12
3.2.1 Consistency 12
3.2.2 Power 12
3.2.3 Flexibility 12
3.3 Uses 13
3.4 Summary 13

4 Lisp-like syntax 14

4.1 Grammar 14
4.2 Desired code format 15
4.3 An algorithm for the printing concerns 15
4.3.1 Spacing 15
4.3.2 Indentation 16
4.3.3 Hard line breaks 18
4.3.4 Soft line breaks 18

4.4 Extensions 24
4.4.1 Aligning indentation style 24
4.4.2 Comments 26
4.5 Summary 28

5 Implementing a formatter 29
5.1 Lexing 29
5.1.1 Examples 30
5.1.2 Structure 32
5.1.3 Whitespace 32
5.1.4 Parentheses 33
5.1.5 Operators & prefixes 33
5.1.6 Number literals 33
5.1.7 Identifiers, keywords & boolean literals 35
5.1.8 Line comments 35
5.1.9 Block comments 36
5.1.10 Empty lines 36
5.2 Printing 37
5.2.1 Spacing 38
5.2.2 Hard line breaks 39
5.2.3 Soft line breaks 41
5.2.4 Indentation 44
5.3 Formatting example 46

6 Conclusion 48

Bibliography 50

Appendix 53

Abstract

We propose an approach to code formatting based entirely on lexical analysis of the input code, disposing the need for expensive syntactic analysis. This approach has the potential to improve the resource efficiency of a formatter and make it easily adaptable to different languages, while sacrificing consistency and beauty of the output format to some degree. We develop a parse-free algorithm that handles all essential aspects of code formatting for a simple Lisp-like language. We implement a lexer and a printer adhering to this algorithm.

German Abstract

Es wird ein Ansatz zur Codeformatierung vorgestellt, der vollständig auf lexikalischer Analyse des Inputcodes basiert und ohne aufwändige syntaktische Analyse auskommt. Dieser Ansatz kann potenziell die Ressourceneffizienz eines Formatters verbessern und ihn leicht an andere Sprachen anpassbar machen, tauscht dafür jedoch im Gegenzug ein gewisses Maß an Konsistenz und Ansehnlichkeit des Ausgabeformats ein. Es wird ein Algorithmus für eine einfache Lisp-ähnliche Sprache entwickelt, der ohne Parsing alle essentiellen Aspekte der Codeformatierung umsetzt. Dieser Algorithmus wird in einem Lexer und einem Printer implementiert.

1 Introduction

Code formatting, also known as *code beautification* or *prettyprinting*, has long been a staple in software development tooling. Researchers have noticed its necessity and started working on solutions decades ago, [Rub83] and the widespread availability of such coding assistance has increased ever since.

Some languages — such as Java — are commonly written in an integrated development environment (IDE). The emergence of those full-fledged tooling packages has helped render formatting tools ubiquitous. Other languages — such as JavaScript — are more commonly written using a rather lightweight editor, often because the language is less verbose or simply for historical reasons. But even some of those editors — such as the web development-focused Microsoft Visual Studio Code — now include formatting tools in their core distributions together with Git version control integration, syntax highlighting and other tooling that is deemed essential, disposing the need to install an additional plugin that provides code formatting support. [18d]

These formatters usually work by lexing and parsing the input code in a process similar to the one a compiler for the same language would utilize, but then instead of proceeding to transform or semantically analyze the generated syntax tree, they use the syntactic information obtained by parsing to print out the token stream obtained by lexing in a defined format that humans consider readable and comprehensible, optionally preserving the shape of the original code to some degree. [BH13]

While many of the algorithms used in common code formatters operate on plain streams of tokens, [Opp80] they also require contextual information to recognize the meaning of tokens and groups of tokens and determine appropriate positions for the insertion of spaces or line breaks. Parsing provides that information, which we can build smart printing routines on, but it is also an expensive operation on top of the aforegoing lexing and forces us to couple our formatter implementation more tightly to the syntactical grammar and thus the concrete target language.

We introduce a procedure that skips the parsing step and build a formatter that can move on from the lexing step straight to printing. This procedure will generate less information about the code we are operating on and thus necessarily limit our capability to generate code formatted in a consistent and customizable output format.

At the same time, our formatter will become quite easy to adapt to vastly different languages, because we only need to know its token types instead of the entire syntactic grammar, and we can potentially achieve higher formatting performance and start emitting output earlier, because the necessity for an initial parsing step no longer applies.

Structure In Chapter 2, we will roughly outline how code formatters typically work. Once we have established fundamental comprehension of their technical background, we can then move on to propose an alternative method that relinquishes parsing. We address potential benefits and uses, but also drawbacks of this method in Chapter 3.

Based on a simple Lisp-like target language, we then design a parse-free printing algorithm that deals with all the major aspects of code formatting and gracefully handles most edge cases in Chapter 4. The very same algorithm then serves as the theoretical foundation that we build a complete implementation of a formatter for the language upon and test it with example code input in Chapter 5.

The goal is for this implementation to tokenize code in the given Lisp-like language and print it back out with certain guarantees on spacing based on surrounding token types, indentation based on expression nesting and line breaks based on expression nesting, line lengths and the original input code shape.

2 An ordinary formatting procedure

In this chapter, we examine the sequence of steps that most formatting pipelines use to process the source code. We will get a grasp on the purpose of the lexing and parsing steps, how they compare to the corresponding routines in a compiler and how the syntax tree format can affect the formatting result. We deal with the printing process more in-depth and break it up into four key aspects. We briefly address some specifics of languages and formatters that a concrete formatter may or may not have to tackle, such as 'virtual tokens' and original code shape retention.

A code formatter that takes an unformatted source code file as its input and produces a formatted source code file as its output typically operates in three main steps: [ZB14][Zhu+16]

1. *lexing* (also known as *tokenizing* or *lexical analysis*)
2. *parsing* (also known as *syntactic analysis*)
3. *printing* (also known as *unparsing*[Opp80])

2.1 Lexing

Lexing transforms a source code string to an array of tokens, such as keywords, identifiers, literals and operators. For the input string

```
if(a==b){c=a+2*b;}
```

in a C-like language, it might return the tokens:

```
if, (, a, ==, b, ), {, c, =, a, +, 2, *, b, ;, }
```

This step is not exclusive to formatters; lexing is a common operation that is also performed by compilers and other static analysis tools that operate on input source code. However, lexers often differ slightly depending on the purpose of a tool. For example, a compiler might completely ignore comments in the source code and not create any tokens from them, because the final machine code instructions it emits would not contain comments anyway.

Some lexers also generate virtual tokens that are not directly present as characters in the source code. The Python language reference describes a lexing procedure that generates `INDENT` and `DEDENT` tokens in order to preserve information about line indentation. [17d, Chapter: 2.1.8. (Lexical Analysis — Line Structure — Indentation)] The Go programming language specification describes a lexing procedure that generates semicolon tokens automatically according to a few simple rules, [17c, Chapter: Lexical Elements — Semicolons] while the ECMAScript language specification describes a more complicated automatic semicolon insertion mechanism for the parsing step. [17a, Chapter: 11.9 (Lexical Grammar — Automatic Semicolon Insertion)]

2.2 Parsing

Parsing transforms an array of tokens to a *parse tree* (also known as *concrete syntax tree*) or, more commonly, to an *abstract syntax tree* (*AST*). The former still contains all of the information about the tokens in the original source code, while the latter is on a higher level and only describes the structure of the program represented by the source code.

This difference can also have an impact on formatting, depending on the individual tree format. A formatter that uses a parse tree might preserve the brackets in the expression `1 + (2 * 3)`, while a formatter that uses an abstract syntax tree might not be able to do so, because all it sees is a multiplication expression inside of an addition expression, not whether the original source was `1 + 2 * 3` or `1 + (2 * 3)` or `((1) + ((2) * ((3))))`. Therefore, the formatter would always output the first variation, regardless of any parentheses occurring in the input source code; or it would always output the second variation in case it is configured to generate code that displays the precedence of multiplication over addition more clearly.

For the input tokens

```
if, (, a, ==, b, ), {, c, =, a, +, 2, *, b, ;, }
```

in a C-like language, a parser might return the abstract syntax tree shown in Figure 1 on a separate page.

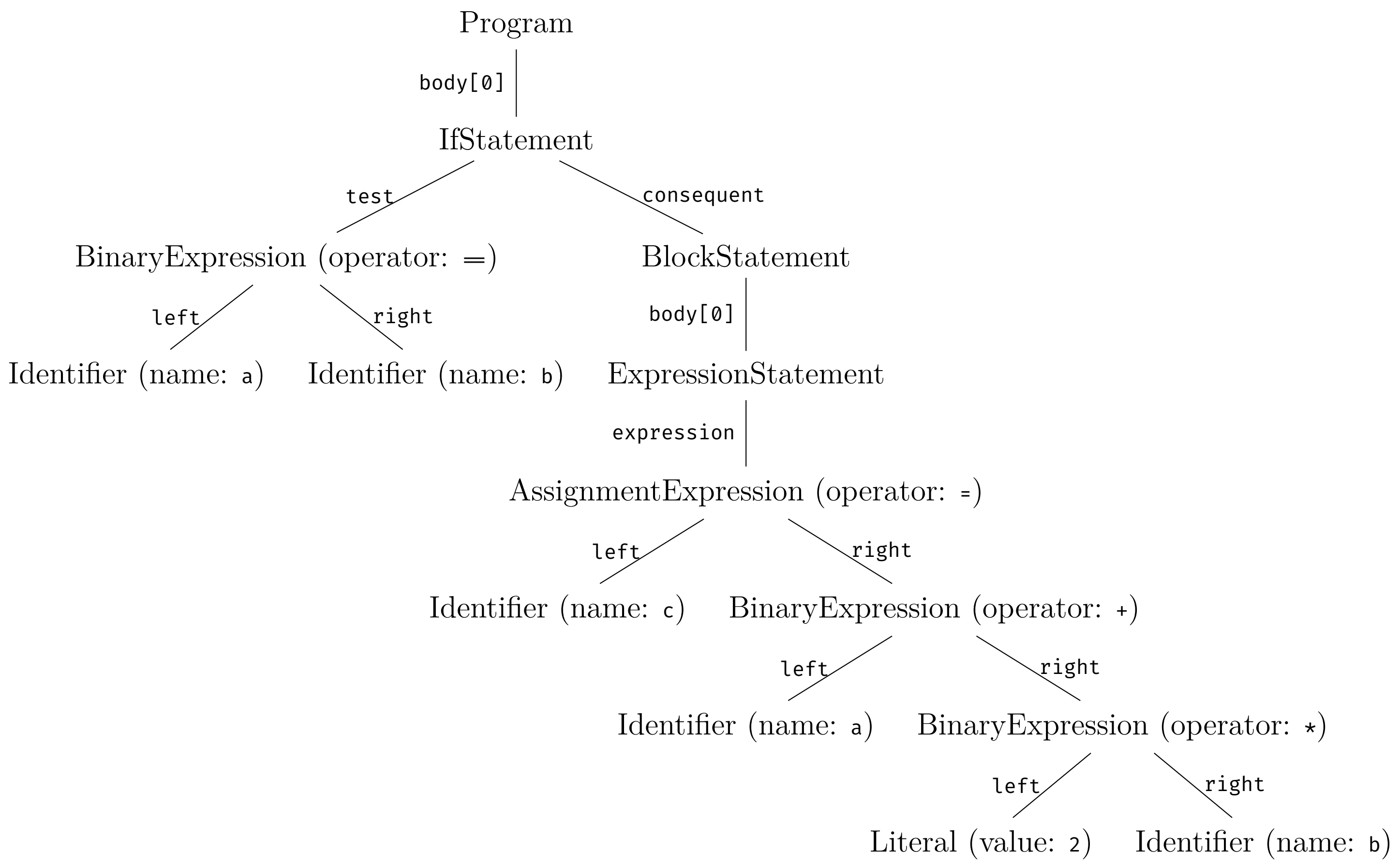

Figure 1: An *abstract syntax tree* for a short if statement, based on the ESTree Spec [Her+17] for ECMAScript ASTs.

2.3 Printing

Printing transforms an abstract syntax tree (or parse tree) to a string of source code that matches the desired format.

For the input AST shown in Figure 1 in a C-like language, a printer might return the code string:

```
if (a == b) {
  c = a + 2 * b;
}
```

Above example shows the first three of the four main concerns of the printing step in a code formatter, which we will all describe in the following.

2.3.1 Spacing

Spacing is necessary because we want to see gaps for instance between the `a` and `==` tokens as well as after the `)` token. We do, however, not want to see gaps for instance between the `(` and `a` tokens as well as between the `b` and `;` tokens. Adding spaces where none existed in the original source code or removing spaces where they existed in the original source code might also change the semantics or even validity of the code. For example, removing a space between a keyword and an identifier would combine those tokens into a single, longer identifier. For a less obvious example that shows how dangerous this can be, consider the following: In JavaScript,

```
console.log(1 .toString())
```

is a valid program and prints the result of calling the method `toString` on the number `1`, whereas

```
console.log(1.toString())
```

is invalid, because `1` and `.` are no longer separate tokens; instead, they now form a single number literal token (`1.`) that is followed immediately by an identifier (`toString`).

2.3.2 Indentation

Indentation helps us understand the control flow, among other nestable structures, in the source code more easily. We want to increase the indentation level at the start of a block statement and decrease it at the end. In the aforementioned example, the assignment inside of the consequent block of the if statement has its indentation level increased by one, relative to the lines immediately above and below that are not entirely contained inside the block statement.

2.3.3 Hard line breaks

Hard line breaks force the insertion of a line break between two tokens. We want this to happen in the obvious positions after an opening brace, closing brace or semicolon as well as before a closing brace. The if statement example code given earlier has been populated with hard line breaks after the opening brace, before the closing brace and furthermore after the closing brace at the end of the example code.

Depending on our desired format we also want to use hard line breaks in some less obvious situtations. For example, it might be useful to apply them before switch cases in order to reformat

```
switch(a) {
  case 1: case 2:
    a = 3;
}
```

to

```
switch(a) {
  case 1:
  case 2:
    a = 3;
}
```

2.3.4 Soft line breaks

The fourth aspect that was missing in the given example is necessary because our output devices can only display characters in a row up to their finite width and because we can only comfortably read lines up to a finite length.

Soft line breaks are used in places where the formatter is allowed to break if it helps avoid exceeding the given maximum allowed line length. Most lines of code will contain more than one soft line break and usually more soft line breaks than we actually need to bring the line length down below the limit. Therefore, an algorithm is required to select those soft line breaks that split up the line optimally by our standards. Given the input

```
someVar = someFunc(someParam, someOtherParam);
```

and line size 50, do we want to print

```
someVar =
  someFunc(someParam, someOtherParam);
```

, breaking at the highest possible node in the AST, or

```
someVar = someFunc(
  someParam, someOtherParam);
```

, distributing the line lengths as evenly as possible, or

```
someVar = someFunc(someParam,
  someOtherParam);
```

, breaking as late as possible, or perhaps

```
someVar = someFunc(
  someParam,
  someOtherParam,
);
```

, grouping the parameter list like we would group statements in a block, which can be especially handy if the language allows us to place the trailing comma in the function call? This concern is arguably the most complex aspect and has been subject to a substantial amount of research in the past. [Hug95][Wad98]

Side note: shape retention With regard to all four concerns, different formatters may choose to preserve the original shape of the source code to a varying degree. For example, it can be helpful to print empty lines exactly in those places where the original source code contained them, [Che+17, Section: Empty lines] because these empty lines usually function as separators between logically coherent sections of the code, which the printer cannot possibly guess from the tree structure.

2.4 Summary

We now know what tokens are, what kind of information the different types of syntax trees contain, and which formatting steps can be used to translate between these artifats and from or to source code. In the next chapter, we propose a formatting procedure that operates on a strict subset of these steps and artifacts and outline its distinct benefits and drawbacks.

We have also unraveled four mostly separate aspects of printing formatted source code. Breaking up the complex printing task into these aspects enables us to take a structured approach to printing later in the course of this paper, where we will conceive and implement an algorithm by building it up one aspect at a time.

3 Eliminating the parsing step

Many of the works on prettyprinting that have been published — such as the well-known one by Derek C. Oppen [Opp80] — propose algorithms that place line breaks into a stream of tokens. Such a stream could be generated by a lexer; a parser would only be required to obtain a tree structure according to the syntactic grammar of the language. But for any conventional formatting algorithm, by the time it comes to actually applying the algorithm to a specific language, parsing the tokens becomes necessary. This is because it is sometimes hard and often impossible to recognize coherent blocks in the token stream, deduct the meaning of each token without knowledge about its greater surroundings or retrieve any other information that could be used for determining suitable spacing or line break positions when going left to right over code written in any practical language.

An example of this problem arises with the ambigious meaning of the `-` (minus) symbol [BH13, Chapter: Introduction], which many languages use both as a unary operator that computes the additive inverse of its operand and as a binary operator that subtracts its right-hand operand from its left-hand operand. Consider the following use of the 'minus' symbol with its immediate neighbor tokens:

```
), -, x
```

One might intuitively proclaim that this minus symbol has to be a binary operator, subtracting the value stored as identifier `x` from whatever subexpression the parenthesis on the left closes. But the syntactic grammars of most languages are not that simple; the whole statement could be

```
if (a == b) -x;
```

, or the language might allow parenthesized type casts and the statement looks like this:

```
int y = (int) -x;
```

There are good reasons to parse the source code in order to get better insight into its structure before attempting to prettyprint it, but what if we still try to create a formatter that actually omits that parsing step, printing out nicely formatted source code just based on the tokens recognized by the lexer?

3.1 Potential benefits

We outline some advantages that such a formatter would have in the following sections.

3.1.1 Efficiency

Parser implementations commonly have quadratic or worse polynomial time complexity in theory, but achieve near-linear time complexity for the language grammars used in practice. [PF11] [PHF14] But even with time complexity similar to lexing, parsing still adds another step on top that is even more expensive than lexing itself.

In a quick test run performed by the author, the popular JavaScript parser 'acorn' [18a] took around three to five times as long to parse a large bundle of JavaScript code than to just tokenize the same code. Parsing can be a major bottleneck in the formatting pipeline, so its elimination, especially when combined with a printing algorithm focused on efficiency, could speed up the whole process of formatting significantly.

We can also expect the memory usage footprint of the formatter to be lower in comparison to an ordinary formatter, because we do not need to hold a large tree structure in the memory.

3.1.2 Streaming

A parse-free formatter could potentially be capable of writing the first line of output right after reading the first few tokens of input and continuing to write quite consistently as it progresses through the input code, because conceivable implementations would never need to look at more than a section between two hard line breaking tokens at once. Implementing such a capability would be way more trivial than it is for formatters that parse the code first.

3.1.3 Adaptability

A formatter that only has to understand the lexical grammar of a language, but not its syntactic grammar, can be more easily adapted to format other languages. For instance, different C-like languages can be formatted without implementing formatting of block statements that are surrounded by curly braces more than once.

3.2 Drawbacks

These benefits, however, come at a cost for our formatting style, which we address in the following sections.

3.2.1 Consistency

It is clear that we will not be able to achieve the same degree of consistency for some of the harder aspects of formatting when the abstract syntax of expressions is the same, but the concrete syntax differs. For instance, a parenthesized expression may be split with line breaks in a different way than a semantically equivalent expression without parentheses.

3.2.2 Power

We also cannot remove unnecessary parentheses like some formatters that print code from an AST do [17b] or trivially make other significant changes to the code, because the lack of syntactic information renders us unable to easily judge whether

- such a modification would alter the semantics of the code and
- such a modification would actually increase the readability of the code.

3.2.3 Flexibility

Information about constructs in the code and about the greater picture is often unavailable and we have to deal with tight limitations on what we can safely change in the code without altering semantics. This means that we cannot always just pick a preferred formatting style that we want to produce, but instead have to accept compromises on the output style and focus on what we *can* produce efficiently.

Allowing for further user configuration options of the desired output format would be borderline inconceivable at this point. However, a lack of configuration options is not necessarily a bad thing as varying formatting style can fragment a language ecosystem and lead to unnecessary controversy. For this reason, both formatters included in the core toolkit of their language [13] and third-party formatters [Che+18] have recently opted to value convention over configuration and provide no or only the most indispensable configuration options.

3.3 Uses

This kind of formatter would not be suitable for some of the common use cases, including both formatters integrated into an IDE or editor that run automatically or on key press and formatters that provide a command-line interface (CLI) for use by developers. Those will usually be applied to small or medium-sized source code files, so the increased performance when compared to traditional formatters does not provide a sufficiently large benefit to justify a trade-off in quality of the code style.

There are, however, some cases where the efficiency and potential streaming capability could turn out to be useful for a more 'on-the-fly' formatting. Browser developer tools usually include a formatter for JavaScript files, because those files are typically served as minified code that has no spacing, no line breaks and consequentially no indentation at all. Besides minification, the JavaScript preprocessing pipeline often also includes a bundling step that merges all script and module files from the original source code into a single or very few files in order to eliminate the need to serve each source file in a separate HTTP request. Because of this common practice, the bundled files delivered to the browser can be quite large — sometimes multiple megabytes in size — and the formatting process can take multiple seconds to complete. Decreasing the total formatting time and the delay until the first lines are printed could improve the developer experience in this use case, while the 'less pretty' formatting style would be tolerable, since the code produced is usually looked at rather briefly and rarely modified.

We could also provide a CLI for the formatter and apply it to code files viewed in a terminal, where a large delay before displaying the content would be unacceptable. In particular, the formatter could be invoked on external code, perhaps downloaded or decompiled code. The `less` pager utility supports file preprocessing by an arbitrary program through the `LESSOPEN` environment variable; [Nud] this could be used for a formatter just as it is sometimes used for syntax highlighting and it would greatly benefit from streaming-enabled formatting. Some IDEs and editors can invoke an external program to process a file on keypress as well.

3.4 Summary

By using a formatting pipeline that consists of just the lexing and printing steps, we can hope to achieve higher performance and efficiency while still keeping the formatter easily adaptable to different languages. Increased source code throughput can be helpful for formatting large minified bundles of code, as they are often shipped to web browsers nowadays. At the same time, we have to condone that the quality and flexibility of our output format will inevitably deteriorate with a parse-free approach.

4 Lisp-like syntax

To gain insight on how a parse-free formatter could be built, let us take a look at a small language with Lisp-like syntax. This is a good starting point because Lisp dialects tend to have among the simplest lexical and syntactic grammars. The language does not need to be complete enough for practical usage, but it should reflect the core syntactic concepts that Lisps share.

4.1 Grammar

We will not define the complete and formal lexical or syntactic grammar of the language, because most of that information would only be valuable for a parser, or for a lexer with the goal of producing tokens that can be easily parsed. Instead, we will define the types of tokens we have to deal with and only build up an intuitive understanding of how the language works based on the Lisp-like languages already out there, because intuition is also what we need decide how a 'good' formatting style looks.

The following ABNF specification outlines the primitives that may occur in the source code. As stated earlier, this is not a full lexical grammar — there is no lexing goal nonterminal and whitespace is not accounted for.

```
identifier  = ALPHA *(ALPHA / DIGIT)
keyword     = "quote" / "lambda" / "defun" / "let"
keyword     =/ "if" / "and" / "or"
prefix      = "'" / "&"
operator    = "=" / "+" / "-" / "*" / "/"
boolLiteral = "true" / "false"
numLiteral  = 1*DIGIT
leftPar     = "("
rightPar    = ")"
```

We could, of course, conceive lots of additional keywords, operators, types of literals, prefixes and other tokens, but not many of them would be interesting with regard to code formatting, because they would be treated like some other token that we already have in our list.

4.2 Desired code format

These tokens are for example sufficient to build the factorial algorithm from the Lisp programming language Wikipedia page: [18c]

```
(defun factorial (n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

We would consider this code example well-formatted — the spacing is consistent, nested expressions are indented and line breaks are used to split up the top-level expression that would otherwise be too long to comprehend it quickly.

One might also notice that there are no inherent differences in formatting between identifiers, keywords, operators and literals. This is somewhat consistent with their treatment in code, where they are all conceptually just elements of a list that is denoted by an S-expression and they can often be used interchangably. This realization of course further simplifies the formatting of an already simple language.

4.3 An algorithm for the printing concerns

Now we can come up with an algorithm that operates on a stream of tokens mostly by just going through them left to right, does some processing that takes into account the four primary formatting concerns and prints out the code in a format similar to that of the code example shown above.

4.3.1 Spacing

For our Lisp-like syntax, we want to separate most tokens with a space, with the following exceptions:

- No space after a `leftPar`,
- no space before a `rightPar` and
- no space after a `prefix`, which is not visible in the example.

To achieve this, all we need to do is define for each type of token whether it allows a leading space and whether it allows a trailing space. Then, we can look up between every two tokens whether the token on the left allows a trailing space and whether the token on the right allows a leading space and insert a space only if both do. Applying this procedure on the tokens

```
(, a, (, ', b, c, ), )
```

results in the code output:

```
(a ('b c))
```

It also works for a prefix preceding a list, printing the tokens

```
(, a, ', (, b, c, ), )
```

as

```
(a '(b c))
```

4.3.2 Indentation

To achieve an indentation similar to that of given example, we can keep track of the level of nested parentheses surrounding us using a counter. That counter is initialized with 0, incremented by 1 after we pass a `leftPar` and decremented by 1 before we pass a `rightPar`. After a line break is placed, we can determine the size of the whitespace we need to print by looking at the current state of the counter n. We print n tabs if our type of indentation is tabs, or $n * m$ spaces if our type of indentation is spaces, m being a positive integer denoting how large our indentation steps should be. In the code examples shown here, we indent with two spaces each.

We have not yet introduced a mechanism for inserting line breaks between tokens so everything would be printed in a single line and there is nothing to indent yet, but for visualization purposes we will assume that line breaks will be placed some particular spots and represent them as special hard break tokens.

If we use the counter method to generate indentation for the tokens

```
(, avg, num1, <HARDBREAK>,
(, add, num2, <HARDBREAK>,
num3, ), <HARDBREAK>,
num4, )
```

, we get the printed result:

```
(avg num1
  (add num2
    num3)
  num4)
```

This format is well-readable in that the indentation and dedentation make it quite easy to see which expression each identifier belongs to. It also saves a lot of space by never indenting more than required by the level of expression nesting, so we will not have to split up lines too often when dealing with the soft line breaking aspect.

It is, however, not exactly the same formatting style that we see in the 'factorial' example. Using our method, we would get the following code for the factorial function, again assuming that the line breaks remain the same:

```
(defun factorial (n)
  (if (= n 0) 1
    (* n (factorial (- n 1)))))
```

Comparing this to the original factorial example code

```
(defun factorial (n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

, we notice that while the first two lines are indented in the same way that we managed to reproduce, but the last line is indented by six spaces instead of four. Here, another indentation method is applied, one that aligns the start of the second and all following elements of a list by setting the indentation of the third and all following elements to the column where the second element was printed. This results in deeper indenting that takes up more space, but might be considered more readable by some. Unlike the example code, we will have to choose one of the two styles of indenting, but we can alternatively produce the latter 'aligning' style as well and will do so later in the chapter.

Finally, let us briefly clarify why we need to decrement the counter *before* a `rightPar`, although we increment it *after* a `leftPar`. This is only relevant when there is a line break right before a `rightPar`. Consider the following code:

```
(add someLargeIdentifier someVeryLargeIdentifier)
```

If our line length is limited to 25 characters, the only way to print the code without violating the limit is

```
; line size 25
(add someLargeIdentifier
  someVeryLargeIdentifier
)
```

, putting the `rightPar` on a new line because it would be one character too much to appear immediately after `someVeryLargeIdentifier`. If this happens, we want the `rightPar` to be aligned with the corresponding `leftPar`. Not doing this would look particularly weird if there are deeper nested expressions, since the outer `rightPar` would appear as if it belonged to the inner `leftPar`:

```
; line size 30
(add someLargeIdentifier
  (sub someNestedIdentifier
    someOtherNestedIdentifier)
  )
```

4.3.3 Hard line breaks

The only source code position where we always want to break is after the completion of a top-level expression. We can use the same counter we introduced for indentation for this purpose as well by inserting a hard line break after a `rightPar` token if and only if the counter has been decremented to 0. This way, the tokens

```
(, define, x, (, +, 1, 2, ), ), (, display, x, )
```

will be printed as

```
(define x (+ 1 2))
(display x)
```

, with hard breaks inserted after the second and third `rightPar`, but not the first.

We can also use hard line breaks to preserve empty lines that may have been present in the original source code. This feature has already been mentioned in the introduction and is usually indispensable for any practical formatter, because developers will need those empty lines in their code to structure it clearly. However, this does require changes to the lexer, so it can already insert hard line break tokens into the token stream whereever it finds a line that does not contain any token. Our counter-based hard break insertion will also need to take into account that it might find hard breaks already present in the token stream and must not insert another one if the `rightPar` is already followed by one, otherwise we might end up printing more than one consecutive empty line.

4.3.4 Soft line breaks

As mentioned before, the placement of soft line breaks is the most complicated aspect, but also a well-explored one, with many good algorithms already developed and in use. Most of those are also quite efficient as far as time and space complexity are concerned, usually requiring asymptotically linear resources as the number of tokens in a line grows.

Established algorithms We could, for example, use the algorithm proposed by Bagge & Hasu. [BH13, Chapter: Line Breaking] For our Lisp-like language, we would be able to provide the necessary control tokens 'nesting start' and 'nesting end' without any parsing, simply by looking at the parentheses — other languages might for example have 'overloaded' tokens that complicate the matter and may require parsing to generate useful control tokens.

It is important to keep in mind soft lines breaks will usually be the computationally most expensive aspect and are thus likely to be a critical path that affects performance the most. The other concerns usually do not require us to look at more than two tokens at once or go to a previous token in the stream at any time. There is not necessarily a hard border between contextual token processing and parsing; if our processing becomes increasingly complex because the language we want to format has a complex grammar or our line breaking algorithm requires extensive meta information, we might already be *parsing* a simplified version of the syntactical grammar.

For all of the strategies to achieve desirable soft line breaking behavior that were given in the introductory paragraphs on soft line breaks, we can come up with efficient approximations or exact algorithms. An algorithm like the one Bagge & Hasu have developed can be used for breaking at positions that would be high in a syntax tree constructed by parsing the tokens. An algorithm like Oppen's [Opp80] and its derivates can be used for placing breaks so that parameter lists and other tokens that belong together form groups — given that the language is simple enough so we can infer the necessary information without parsing.

Greedy line-filling Another one of the strategies, breaking as late as possible, is trivial to achieve and requires no more than one pass through the tokens in the line. In general, however, it does not print very well-readable code unless we make some modifications beyond its basic form.

The indentation rules we have already established will take care of inserting the correct number of spaces or tabs whenever we insert a break. We keep track of the already printed line length by initializing it with the number of columns already filled out by the indentation and adding the number of characters in any token we print to it. We only print a token if it would not increase our column counter beyond the maximum allowed line length. Otherwise, we insert a line break and reset the counter, indenting as usual. Then we proceed with the remaining tokens until the next hard break.

We also need to carefully consider the interaction with the other printing concerns. If we break between two tokens that were separated by a space, the line break must *replace* that space. We can neither print the space before the line break, producing a line with a trailing space, nor can we print it after the line break, increasing the indentation of the new line beyond the indentation generated from our nesting counter. Concerning indentation, we must additionally take into account that `rightPar`s may be printed as the first token after a soft line break, and that we defined those to decrease the indentation level *before* being printed. This means we must process not only the *after*-effect on indentation of the previous token, but also the *before*-effect of the following token *before* printing the indentation whitespace that a new line after a soft break requires.

This approach will ensure that too many tokens between two hard breaks will be spread across multiple lines, and it may do so in a reasonable way, for example when printing the following tokens with line size 25:

```
(, let, something, ), (, identifier, 012345,
43210, (, someIdentifier, 0123456, 789, ), )
```

The resulting code with our standard indentation, spacing and hard breaking method and this greedy line-filling algorithm for soft breaks is:

```
; line size 25
(let something)
(identifier 012345 43210
  (someIdentifier 0123456
    789))
```

Avoiding undesired separation While this may seem like reasonable formatting style, it is really just a lucky token arrangement and line length. If we increase the line length by 1, the code we produce looks weird:

```
; line size 26
(let something)
(identifier 012345 43210 (
    someIdentifier 0123456
    789))
```

If we keep the line length but replace `identifier` with `something`, something similar happens:

```
; line size 25
(let something)
(something 012345 43210 (
    someIdentifier
    0123456 789))
```

The most annoying issue about both of the 'weird' code formats we produced is that `someIdentifier` is separated from its preceding `leftPar` by a line break, especially because the same thing is not applied to the code snippets `(identifier` or `(something` that occur earlier and have the same structure. We want to avoid the separation of token sequences like `(x` and its counterpart `y)`, including the case where multiple subsequent parentheses occur: `((x` or `y))`. We also want to avoid a similar separation issue if a break is inserted immediately after a prefix token, so constructs like `&x` always remain together. The latter would not only look terrible, it may also be forbidden by some languages with special treatment of prefixes.

We notice that for the language we are dealing with, the token pairs that must not be separated by line breaks are exactly those token pairs that do not allow the insertion of a space between them. Because of this, we can base a system of line break allowance on our established spacing rules, effectively treating tokens that are printed with no space inbetween them as single tokens.

This makes the two problematic examples more bearable and in this case even uniform among one another as well as consistent with the unproblematic example:

```
; line size 26
(let something)
(identifier 012345 43210
  (someIdentifier 0123456
    789))

; line size 25
(let something)
(something 012345 43210
  (someIdentifier 0123456
    789))
```

Inevitable length limit violation Excessive expression nesting leading to deep indentation that consumes most of the available columns or simply the occurence of overly large tokens can force us to violate the restrictions on maximum line length. Even if we ignore the separation avoidance mechanism we just introduced, there is no way, including breaking at every single possible position, to place soft line breaks so the following code does not exceed the line length limit:

```
; limit ⊣
(
  abc
  (
    def
    (
      ghi
      (
        jkl
      )
    )
  )
)
```

We could limit the width of indentation that we print to make this problem less likely to occur. But we cannot entirely eliminate the problem this way — if we have to print a large token, the line length may still be exceeded, even without extraordinarily deep indentation:

```
; limit 25 -------------|
(
  abc
  (
    someVeryLargeIdentifier
  )
)
```

To mitigate the effect of such a violation and still print the remainder of the tokens in a reasonable way, we bump the maximum line length up to match the overhang we had to use for the offending token until the next hard line break. The following example is once again printed with our regular algorithm, including separation avoidance.

```
; limit 15 ---|
(abc 12345
  12345
  largeIdentifier
  0123456 0123456
  0123456
  0123456)
```

The second `12345` does not fit into the same line anymore and is placed on a new line instead. `largeIdentifier` does not fit into that line, so it is placed on a new line as well. However, even on that new line, the token takes up too much horizontal space, overhanging by 2 characters. We have no choice but to print it like that and to temporarily bump our line length to 17. Because of the increased line length, the following line can hold two `0123456` tokens, which would otherwise have been printed on separate lines. The final two `0123456`s cannot be printed on one line, because we cannot separate the latter one from its closing `rightPar`, which in turn cannot be printed on the same line because it would occupy column 18, exceeding even our increased line length.

Retrospectively, we could have also printed both `12345`s on the same line, because our line length for this part of the code ended up being 17 instead of 15, but extending the algorithm to do this would require giving up the linear processing order and going back to a token we have already printed.

Uniform length distribution There is still one strategy for placing soft line breaks that we disregarded so far: Distributing the tokens so each partial line is about the same in size. There are multiple ways to implement this strategy. Unfortunately, none of them is computationally quite as trivial as the greedy line-filling algorithm we discussed before — going through the tokens left to right once will not suffice here. For a solution that is still quite fast and simple, we can split a line that is too long between its two most central tokens. For example, the 21-characters long (after spacing) line

```
(a bc def ghij klmno)
```

would be split into

```
(a bc def
ghij klmno)
```

, with the partial lines being 9 and 11 characters long (before indentation). We then need to recalculate the indentation for the part after the split. Taking the indentation into account, we determine for both parts whether they still exceed the length limit, and for each one that does, repeat the split recursively.

Just like with greedy line-filling, we can treat tokens without a space between them as one token to avoid undesired separation of them. Inevitable length limit violations are not as easy to deal with, because by the time we detect such an offense, we are at a leaf of the recursion tree. We could extend the algorithm to give feedback on how many columns of horizontal space a partial line ended up using, so further invocations know that they can use that extra space.

Note that this algorithm is not optimal in distributing the length uniformly. Consider the tokens:

```
(, aaa, b, c, dd, )
```

Given line size 6, our algorithm would insert a soft line break between `b` and `c`, and then between `c` and `dd`, producing the code

```
(aaa b
  c
  dd)
```

as opposed to the optimal length distribution:

```
(aaa
  b c
  dd)
```

It may prove useful to pick a slightly higher ratio than 50% to split at. This is because practical lisp code tends to have higher levels of nesting towards the end of an expression, so we can expect the code after the split to have more indentation whitespace that occupies some extra columns.

4.4 Extensions

Having established a basic formatting procedure, we can address some extensions to the algorithm that may be necessary or nice to have.

4.4.1 Aligning indentation style

When discussing our method for indenting code in this Lisp-like language, we noticed that the example code for an implementation of the factorial function used two different styles of indentation.

```
(defun factorial (n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

The middle line is indented according to the depth of expression nesting and matches the indentation that our algorithm would have assigned to the line. The last line, however, is indented with six spaces instead of the four we would expect if the same indentation style had been used.

This alternative indentation style uses the column where the second element of a list started as a baseline for any further elements that are part of the same list and need to be assigned an indentation level. The following example uses this style consistently, albeit containing strangely placed line breaks that cause the weird appearance:

```
(a b c
   d (e f
        g) (h i
              j (k l
                   m)
              n)
   o)
```

Using spaces for indentation is mandatory with this style, since tabs cannot provide sufficiently granular control to achieve exact alignment.

To implement this method of indentation, we can use a stack (*LIFO*) data structure containing numeric values that describe the indentation level per nesting level.

- When printing a `leftPar`, we push a 0 onto the stack.
- When printing a token after a space and the stack holds a 0, we replace the 0 with the column where the token starts.
- When printing a `rightPar`, we pop a number off the stack.

Whenever we need to look up the indentation level required for the current position, we can just look it up on top of the stack.

This is the example from above, annotated at each line break with the current stack values as zero-based column indices:

```
;()
(a b c
;(3)
   d (e f
;(3, 8)
        g) (h i
;(3, 14)
              j (k l
;(3, 14, 19)
                   m)
;(3, 14)
              n)
;(3)
   o)
;()
```

Early line breaks It is possible that we need to provide an indentation level before encountering the second element of the current list, when the stack still contains a zero at the top. This could happen if there is a line break immediately after the `leftPar` and the first identifier:

```
(aaa
b c) ; how to indent this line?
```

It is even more likely to occur if the first expression is more complex than a single identifier:

```
((a1 a2 a3
     a4 a5)
b c) ; how to indent this line?
```

There are multiple reasonable options to handle this case, including

- using the indentation level from the previous nesting level, plus an offset, and
- aligning with the *first* element of the list instead.

Escalating indentation It might also be a good idea to use this indentation style only if the indentation level does not jump up more than a certain threshold, and fall back to the nesting-based style otherwise. The pure form of this indentation method can provoke extremely deep indentation very quickly:

```
(firstElementIsVeryLong second
                        ((first element is complex) second
                                                    third))
```

It is also dangerous when paired with certain soft line breaking algorithms, such as the greedy line-filling method we defined:

```
; line size 20
(aaaa (bbbb cccc
            dddd))
```

4.4.2 Comments

The treatment of comments varies significantly for different lexers. Lexers used in compilers often discard them right away, but for the purpose of code formatting, they need to be preserved and must thus appear in the token stream like any other token. As such, we can assign formatting rules to them like we did for identifiers, prefixes, `leftPar`s and other token types.

For line comments (`;`), we must place a hard line break behind them, so we do not risk reformatting

```
(a ; comment
b)
```

to

```
(a ; comment b)
```

, modifying the program syntax and in this case making it invalid. Block comments (`#| |#`) do not carry this risk. Spaces around comments, as well as potential soft breaks, are desired.

Using these rules, we can output comments in a variety of ways depending on the context they appear in, for both line comments:

```
(a b) ; at the end of a line

(a b cdefghijklmnopqrstuvwxyz)
; on their own after a used soft line break
```

, and block comments:

```
(a b c
#| at the beginning of a line |# d e)

(a b c #| or right in the middle |# d e)

(a b c
#| spanning
multiple
lines |#
d e)
```

We could go further and convert line breaks inside block comments to hard line breaks and split the block comment into multiple tokens of type block comment, while only the first keeps the starting comment delimiter `#|`, and only the last keeps the terminating comment delimiter `|#`. This would have the effect that our printer understands that the comment is not entirely on one line, and it might in the multi-line example above print

```
(a b c #| spanning
multiple
lines |# d e)
```

instead, using some more of the available space on the first and last lines of the block comment for the remaining tokens.

It would also cause the printer to apply indentation inside the block comment. Previously, indentation was primarily relevant after soft line breaks, because our hard line breaks occurred in locations where the nesting and indentation were zero. With hard line breaks from block comments appearing in the middle of expressions, we could profit from the existing indentation algorithm to align comments appropriately:

```
(a (b (c (d (e (f (g
            h #| multi
            line
            comment |# )))))))
```

looks a lot better than:

```
(a (b (c (d (e (f (g
            h #| multi
line
comment |# )))))))
```

It is possible to perform some more complex formatting on comments, such as indenting text inside of block comments and inserting soft line breaks inside of block comments and even converting line comments to block comments to avoid violating the line length limit, but we will not deal with these advanced operations in the scope of this paper.

4.5 Summary

In this chapter, we used a simple Lisp-like language to explore how we can design a formatting algorithm based entirely on lexical information. We found simple rules for spacing and hard line breaks and defined methods to generate output with two distinct indentation styles. We outlined multiple different ways to insert soft line breaks into overly long lines of code and found ways to handle edge cases such as the inevitable violation of line length limits.

In the next chapter, we will pick a concrete formatting style and implement a formatter based on the theoretical foundation layed out here.

5 Implementing a formatter

In this chapter, we will use the insight into formatting rules for our Lisp-like syntax that we gained and the concrete algorithms we developed to implement a complete formatter that can transform input code adhering to the syntax, regardless of its formatting, to equivalent output code that fulfills our code formatting requirements. We will write the formatter in JavaScript, compliant with ECMAScript 2018, as two strictly separated functions that realize the lexing and printing steps. The full source code of the finished lexer and printer functions can be found in the appendix at the end of this paper.

5.1 Lexing

Our formatter is built to do without parsing any code, but we still need to split the input code into individual tokens. Therefore, we will start by creating a lexer that can recognize all the token types we defined for our lisp-like language, including line and block comments from the comments extension. We could use a parser generator such as ANTLR to generate a lexer only, but due to the simple nature of our language, a lexer will be sufficiently trivial to implement ourselves.

To implement the lexer, we will first define its signature and give example input and output that describes its desired behavior and can be used as a basic test suite for the lexer function. We construct a code skeleton for the basic structure of the lexer function and then successively add lexing rules for each type of token, including comments and empty lines.

The lexer function has the signature

```
(code: string) ⇒ {
  type: 'leftPar' | 'rightPar' | 'operator' | 'prefix' |
    'numLiteral' | 'boolLiteral' | 'keyword' | 'identifier' |
    'lineComment' | 'blockComment' | 'emptyLine',
  value: string,
}[]
```

, taking the input code as a string and returning an array of token objects with a property `type` from a fixed pool of strings that contains a type identifier for each possible token that can occur in the code, and a a property `value` that contains the snippet from the input code that represents the corresponding token. The token types are exactly those we defined in the language grammar, plus `'lineComment'` and `'blockComment'` for representing the comments we defined as an extension to the language, which have no effect on the code semantics, but are treated like any other token for the purpose of formatting, and `'emptyLine'` for preserving empty lines in the code, which we deemed essential to the hard breaking algorithm of a formatter.

5.1.1 Examples

The following example input code strings and corresponding expected output token arrays demonstrate the intended functionality that the lexer should provide and can be used as test cases to verify that a lexer implementation works roughly as expected. These examples are not a complete test suite that covers all cases.

Regular token types The lexer recognizes the regular token types as defined by the language grammar:

```
(lambda id1 &id2
  (= true 123))
```

is tokenized to:

```
[
  { type: 'leftPar', value: '(' },
  { type: 'keyword', value: 'lambda' },
  { type: 'identifier', value: 'id1' },
  { type: 'prefix', value: '&' },
  { type: 'identifier', value: 'id2' },
  { type: 'leftPar', value: '(' },
  { type: 'operator', value: '=' },
  { type: 'boolLiteral', value: 'true' },
  { type: 'numLiteral', value: '123' },
  { type: 'rightPar', value: ')' },
  { type: 'rightPar', value: ')' },
]
```

Whitespace inbetween the tokens is ignored, unless it is picked up by the *empty line* rule.

Line comments The lexer recognizes line comments as used in the comments extension:

```
(a ;comment
b)
```

is tokenized to:

```
[
  { type: 'leftPar', value: '(' },
  { type: 'identifier', value: 'a' },
  { type: 'lineComment', value: ';comment' },
  { type: 'identifier', value: 'b' },
  { type: 'rightPar', value: ')' },
]
```

Block comments The lexer recognizes block comments as used in the comments extension:

```
(a #|com
ment|# b)
```

is tokenized to:

```
[
  { type: 'leftPar', value: '(' },
  { type: 'identifier', value: 'a' },
  { type: 'blockComment', value: '#|com\nment|#' },
  { type: 'identifier', value: 'b' },
  { type: 'rightPar', value: ')' },
]
```

Empty lines The lexer inserts a token where the code contains an empty line:

```
(a

b)
```

is tokenized to:

```
[
  { type: 'leftPar', value: '(' },
  { type: 'identifier', value: 'a' },
  { type: 'emptyLine', value: '' },
  { type: 'identifier', value: 'b' },
  { type: 'rightPar', value: ')' },
]
```

Illegal characters The lexer throws an error if the code includes an illegal character:

```
(a ~ b)
```

results in a lexer error.

5.1.2 Structure

The lexer function holds two essential values:

- the current position in the source code: `let position = 0;`
- a mutable array of the tokens recognized so far: `const tokens = [];`

The return value in the success case is the `tokens` array.

The backbone of the lexing process is formed by the loop:

```
while (position < code.length) {
  let char = code[position];

  // ... token detection here ...

  throw new Error(
    `unexpected character ${char} at position ${position}`);
}
```

It runs until the position cursor reaches the end of the input code and reads the current character into the `char` variable in every iteration. We can then insert arbitrary token detection code that pushes new tokens to the `tokens` array into the loop where the placeholder comment is positioned. This code is also responsible for advancing the `position` cursor and triggering the next iteration using `continue;` after doing its work. If the token detection code did not successfully match a token, we have found an illegal character in the source code and throw an error.

5.1.3 Whitespace

Whitespace primarily includes spaces, tabs, line feeds and carriage returns, all of which are characters which will not be included in any token but may serve as separators between tokens, for example two identifiers that would be treated as one if there were no whitespace between them.

It can be matched with the regular expression:

```
const whitespaceRegex = /\s/;
```

We use that regular expression to skip all of it:

```
if (whitespaceRegex.test(char)) {
  position++;
  continue;
}
```

5.1.4 Parentheses

Parentheses are single-character tokens that can have exactly one value — `(` for `leftPar`s and `)` for `rightPar`s.

We push a new token whenever we detect one of them with

```
if (char === '(') {
  tokens.push({ type: 'leftPar', value: char });
  position++;
  continue;
}
```

and the equivalent for `rightPar`s afterwards.

5.1.5 Operators & prefixes

Operators are single-character tokens consisting of one of the characters:

```
const operators = ['=', '+', '-', '*', '/'];
```

If the current character is one of those in the `operators` array, we push a new token of type `operator`:

```
if (operators.includes(char)) {
  tokens.push({ type: 'operator', value: char });
  position++;
  continue;
}
```

For prefixes, we do the same based on the characters:

```
const prefixes = ["'", '&'];
```

5.1.6 Number literals

Number literals are a little more complex to recognize, because they are the first token so far to have variable length.

Initial detection is simple using the a regex that matches digits:

```
const numRegex = /[0-9]/;

if (numRegex.test(char)) {
  // ...
}
```

But after the initial digit, the number literal may contain further digits. We use a loop to collect them in a string, running until we find a character that is not a digit:

```
let numberLiteral = '';

do {
  numberLiteral += char;
  char = code[++position];
} while (numRegex.test(char));
```

Afterwards, we can push the token as usual, with the value set to the string of accumulated digits:

```
tokens.push({ type: 'numLiteral', value: numberLiteral });
continue;
```

The `position` cursor has already been advanced beyond the number literal in the last iteration of the loop, so we do not need to increment it like we did in the other blocks.

This also implies that the lexer will accept any token immediately after the number literal — the code

```
(123abc)
```

would be tokenized into the number literal `123` and, by a rule we have yet to implement, the identifier `abc`. The language might not allow this pattern to occur in the code without any separating whitespace, so we may accept code that is not valid — and in this concrete case later reformat it to valid code that conforms to what was likely the intended code in the first place:

```
(123 abc)
```

In any case, accepting a superset of code that is actually valid by the rules of the language is not something we can avoid, because the lack of parsing means that we can only check the validity of the input according to the lexical grammar, but not according to the syntactic grammar. We can tolerate these instances of reformatting code that is invalid, because our printing algorithm will never introduce significant changes that destroy the code further and it will recover a well-formatted state after the user fixes the code error and reformats once again.

5.1.7 Identifiers, keywords & boolean literals

Identifiers, keywords and boolean literals require the same loop structure as number literals. Instead of testing with the `numRegex`, we test with

```
const alphaRegex = /[a-zA-Z]/;
```

for the first character in the if statement, and

```
const alphaNumRegex = /[a-zA-Z0-9]/;
```

for the remaining characters in the do-while statement and store them all in

```
let name = '';
```

Only once the loop has completed, we can use the knowledge about all keywords and boolean literals that exist

```
const booleans = ['true', 'false'];
const keywords = ['quote', 'lambda', 'defun',
                  'let', 'if', 'and', 'or'];
```

to determine whether the token we found belongs to one of those categories or whether it is an identifier:

```
let type;
if (booleans.includes(name)) type = 'boolLiteral';
else if (keywords.includes(name)) type = 'keyword';
else type = 'identifier';
```

Finally, we can push a token with the type we determined for the token:

```
tokens.push({ type, value: name });
continue;
```

5.1.8 Line comments

For detecting line comments, we use the regular loop structure. The initial detection checks for the semicolon character:

```
if (char === ';') {
  // ...
}
```

The loop runs until the end of the line by checking for line breaks:

```
do {
  // ...
} while (char !== '\n');
```

Line break characters may vary depending by operating system, but we will assume `\n` for the sake of simplicity and stay focused on the actual formatting algorithms.

After the loop, we push a token as usual and continue with the main loop.

5.1.9 Block comments

The final block of code for lexing detects block comments. Those are delimited by two characters (`#| ... |#`), so we perform two checks before entering the loop:

```
if (char === '#') {
  let commentChar = code[++position];
  if (commentChar === '|') {
    // ...
  }
}
```

If the inner check fails, we will fall through to the throw statement at the bottom of the lexer main loop and report that the character after the `#` is invalid.

If both conditions are truthy, we have detected a block comment. We initialize the variable to store its text with the character `#` instead of an empty string, because we have already advanced the cursor beyond its occurrence in the code, so the loop will start at the pipe character (`|`):

```
let text = '#';
```

The exit condition for the loop is the detection of the block comment terminator (`|#`), which will be the last bit of code included in the text of the token:

```
do {
  text += commentChar;
  commentChar = code[++position];
} while (!text.endsWith('|#'));
```

The block ends by pushing a new token and continuing the main loop:

```
tokens.push({ type: 'blockComment', value: text });
continue;
```

5.1.10 Empty lines

To detect empty lines in the input code, we need to store an additional state variable across iterations of the loop, tracking the number of line breaks we have seen without a token in between:

```
let consecutiveBreaks = 0;
```

We could alternatively initialize this variable to 1 if we wanted to also respect an empty line right at the beginning of the code.

We need to reset the variable to 0 whenever we find a character that is not whitespace and will thus either generate a token or cause a lexer error. We accomplish this by inserting the statement

```
consecutiveBreaks = 0;
```

below the if statement that checks for whitespace.

Finally, we increment the `consecutiveBreaks` counter and push an `emptyLine` token if it has reached 2 by inserting the following code *above* the if statement that checks for whitespace:

```
if (char === '\n')
  if (++consecutiveBreaks === 2)
    tokens.push({ type: 'emptyLine', value: '' });
```

5.2 Printing

Analogous to the lexer, we implement the printer by defining its signature and basic structure and then successively enriching the code with logic that for each aspect of printing that we need to tackle. When the final bit of printing logic is implemented, we can look at some examples of code that is passed through the finished lexer and printer functions for reformatting.

Once our lexer has tokenized the input code, the array of tokens it generated is passed into the printer. The printer will then generate well-formatted output code from those tokens. Its signature is the inverse of the lexer:

```
(tokens: {
  type: 'leftPar' | 'rightPar' | 'operator' | 'prefix' |
    'numLiteral' | 'boolLiteral' | 'keyword' | 'identifier' |
    'lineComment' | 'blockComment' | 'emptyLine',
  value: string,
}[]) => string
```

The high-level structure of the printer function is as follows:

```
const print = tokens ⇒ {
  let code = '';
  // GLOBAL_STATE ...

  for (const { type, value } of tokens) {
    // PRE_PRINT ...

    // print token
    code += value;

    // POST_PRINT ...
  }

  return code;
};
```

The printer iterates over all input tokens and prints them to the output code. Outside of that loop, it holds some global state, most notably the output code generated so far. As we gradually implement more functionality for the aspects of printing over the course of this section, we will add more state variables as well as computations and further printing before and after a token is printed in the loop.

5.2.1 Spacing

We have established the convention that tokens should always be separated by a space (or line break) *unless*

- the former of them is of type `leftPar`,
- the former of them is of type `prefix` or
- the latter of them is of type `rightPar`.

To implement space separation according to these rules in our parser, we can choose to either

- print a space before the current token if it is not a `rightPar` and we have memorized the previous token not to be a `leftPar` or `prefix`, or to
- print a space after the current token if it is not a `leftPar` or `prefix` and a lookahead reveals the next token not to be a `rightPar`.

We choose the first option, as a lookahead would be less intuitive to implement given our printer structure.

Current token constraint In the `PRE_PRINT` position, we conditionally print a space, depending on whether the current token is a `rightPar`:

```
if (allowsSpaceBefore(type)) code += ' ';
```

with `allowsSpaceBefore` defined as:

```
const allowsSpaceBefore = type => type !== 'rightPar';
```

This ensures that a we do not print a space before a closing parenthesis.

Previous token constraint To prevent printing a space after an opening parenthesis or a prefix, we have to hold state that tells us whether the previous token allows a space behind it across iterations of the token loop:

```
let prevAllowsSpace = false;
```

We initialize this variable to `false` because we do not want a space right at the beginning of the source code.

At `POST_PRINT`, we update the variable based on the current token, which will be the previous token in the next iteration:

```
prevAllowsSpace = allowsSpaceAfter(type);
```

with `allowsSpaceAfter` defined as:

```
const allowsSpaceAfter = type => !['leftPar', 'prefix'].includes(type);
```

We amend the conditional space printing statement to also respect this constraint:

```
if (prevAllowsSpace && allowsSpaceBefore(type)) code += ' ';
```

5.2.2 Hard line breaks

We need to insert a hard line break whenever a `rightPar` decreases the current level of nesting to zero. We track the level of nesting with another state variable declared outside of the printer loop:

```
let nestingLevel = 0;
```

In the `POST_PRINT` position, after the assignment of `prevAllowsSpace`, we track changes to the nesting level and increment it after a `leftPar`:

```
if (type === 'leftPar') nestingLevel++;
```

For `rightPar`s, we need to not only decrement the nesting level, but also print a line break if it has reached zero because of the decrement. This line break must also effect our spacing, making sure that no space is printed in addition to the line break in the next loop iteration. We achieve this with the following code:

```
if (type === 'rightPar') {
  if (--nestingLevel === 0) {
    code += '\n';
    prevAllowsSpace = false; // line break replaces space
  }
}
```

Additionally, we need to insert a hard line break after a line comment to ensure that the following tokens are not interpreted as part of the comment. So we add another check for `lineComments` that inserts a hard line break in the same way, disregarding the current `nestingLevel`.

Preservation of empty lines When discussing the hard line breaks for our Lisp-like language, we also noticed that it is important to preserve empty lines that were present in the input code, because the author of the code will have inserted those to aid reader comprehension of the code structure. We have built functionality in our lexer specifically to insert tokens where empty lines occur; now we need to adapt the printer to use those tokens in order to generate empty lines again.

An empty line can be inserted by printing two consecutive line break characters. To keep track of how many of those we have printed we introduce another state variable:

```
let consecutiveBreaks = 2;
```

We will use this variable to ensure that we never print more than two consecutive line break characters, because we do not want more than one consecutive empty line in our output code. We initialize it with 2, pretending that we have already printed an empty line at the beginning of the code, because we do not want to print an actual empty line before a meaningful character at the start.

When we encounter an `emptyLine` token in the printer loop, between the assignment of `prevAllowsSpace` and the parentheses handling, we count the variable `consecutiveBreaks` up to 2 and print a line break for each increment:

```
if (type === 'emptyLine') {
  while (consecutiveBreaks < 2) {
    code += '\n';
    consecutiveBreaks++;
    prevAllowsSpace = false; // line break replaces space
  }
}
```

For any other token, we reset `consecutiveBreaks` to 0, because we know that we have stopped the series of line breaks by printing the value of the current token in this iteration:

```
if (type === 'emptyLine') {
  // ...
} else {
  consecutiveBreaks = 0;
}
```

To avoid printing two empty lines if an `emptyLine` token occurs after a `rightPar` token that causes a regular hard break insertion, we also need to add the statement `consecutiveBreaks++;` in the code block where we print that kind of line break.

5.2.3 Soft line breaks

Next, we add soft line breaking capabilities to our formatter. We use the greedy line-filling algorithm, as it operates in a simple, linear way that can be implemented in our formatter without adding significantly to its code complexity. We will also employ our strategy for gracefully handling inevitable length limit violations, but will omit the separation avoidance, which would require more fundamental changes to the printer implementation.

Let us take a look at what our printer currently looks like. With a few simplifications applied, notably the extraction of the hard line break insertion logic that we implemented into its own function, `breakLine()`, the code we have at this point is shown in Figure 2 on a separate page.

We declare a variable to track the amount of characters we printed since the last line break outside of the loop:

```
let lineLength = 0;
```

In `breakLine`, we reassign it to 0 so we can start over when we enter a new line. We also declare a length limit that we will try to never exceed. This limit could be set by configuration options; we will assume a length limit of 42 for our purposes:

```
const MAX_LINE_LENGTH = 42;
```

Another change that we need to make to the current code concerns the spacing information in the first statement of the loop. We need to know whether a space will be printed before the token, because whitespace also counts towards the line length. We extract the condition into a variable, and leave some space before the space printing to insert our soft line breaking logic into:

```
let spaceBefore = prevAllowsSpace && allowsSpaceBefore(type);

// soft line breaks

if(spaceBefore)
  code += value;
```

```
const print = tokens => {
  let code = '';
  let nestingLevel = 0;
  let prevAllowsSpace = false;
  let consecutiveBreaks = 2;

  // helpers
  const breakLine = () => {
    code += '\n';
    consecutiveBreaks++;
    prevAllowsSpace = false; // line break replaces space
  };

  // main loop
  for (const { type, value } of tokens) {
    if (prevAllowsSpace && allowsSpaceBefore(type))
      code += ' ';

    // print token
    code += value;

    // set previous information for next iteration
    prevAllowsSpace = allowsSpaceAfter(type);

    // hard line break
    if (type === 'emptyLine')
      while (consecutiveBreaks < 2)
        breakLine();
    else consecutiveBreaks = 0;

    if (type === 'leftPar')
      nestingLevel++;
    if (type === 'rightPar' && --nestingLevel === 0)
      breakLine();
    if (type === 'lineComment')
      breakLine();
  }

  return code;
};
```

Figure 2: The printer code before implementation of soft line breaking.

In the most common and simple case, we can just go ahead and print our token on the current line, perhaps with a preceding space. Then we will simply add the length of the token value to the current `lineLength`, possibly adding another 1 on top if the token will be printed with a space in front of it.

```
let printLength = value.length + spaceBefore;

// what if the token does not fit?

lineLength += printLength;
```

On a side note, adding the boolean `spaceBefore` in this code snippet is merely a shorter way of writing `+ (spaceBefore ? 1 : 0)`.

But if the token does not fit on the same line anymore without exceeding the length limit, we will need to print a line break first:

```
if (lineLength + printLength > MAX_LINE_LENGTH) {
  breakLine();

  // No space after all, just the break
  if (spaceBefore) printLength--;
  spaceBefore = false;
}
```

In that case, we also prohibit printing a space before the current, because the line break takes its place instead. If necessary, the space is also subtracted back from the length that we determined for the current token, so we do not start the new line with an off-by-one `lineLength` value.

Handling length limit violation If a very long token or the indentation that we will introduce in our printer next force us to print beyond the line length limit, we decided that we want to temporarily increase the limit to the length of the violating line until the next hard line break. This helps mitigate the effect of the violation, especially if it was caused by indentation, which is likely to remain deep for further lines we need to print.

To implement this mechanism, we have to start distinguishing between hard and soft line breaks in the printer. We create two functions `softBreak` and `hardBreak` that both delegate to `breakLine`:

```
const softBreak = () => {
  breakLine();
};
const hardBreak = () => {
  breakLine();
};
```

The `breakLine` call we just added becomes `softBreak`, while the calls in the conditional statements that handle `emptyLine`s, `rightPar`s and `lineComment`s become `hardBreak`s.

To track the current, possibly increased, line length limit until the next hard break, we use a variable initialized with

```
let maxLineLength = MAX_LINE_LENGTH;
```

and reset to the same constant in the `hardBreak` function. We replace the usage of the constant in the condition that we introduced for deciding whether to insert a soft line break by a usage of this variable.

To make sure that the variable is increased after we printed a token beyond the limit, we reassign it to the current line length or itself, whichever is higher, after the soft line breaking code:

```
maxLineLength = Math.max(lineLength, maxLineLength);
```

5.2.4 Indentation

For the last aspect to implement, indentation, we will use the regular nesting-based method, not the possible aligning variant we identified as an extension.

The first small change we need to make stems from the realization that decrementing the nesting level because of a `rightPar` needs to happen *before* printing the token and potentially inserting a soft line break in front of it. So we move the decrement up to the beginning of the loop body and then later on we merely read and compare the nesting level:

```
// nesting decrement needs to happen before print
if (type === 'rightPar') nestingLevel--;

// most of the other code is here

if (type === 'rightPar' && nestingLevel === 0) hardBreak();
```

Now we can add the actual code for printing indentation spaces. After calling `breakLine()` from `softBreak()`, we generate an indentation string with a total of `nestingLevel * INDENT_SIZE` spaces. Just like we defined `MAX_LINE_LENGTH` as 42, we define `INDENT_SIZE` as 2 for further examples. We append the indentation string to the code and update the current `lineLength` to match:

```
const softBreak = () => {
  breakLine();

  const indentSize = nestingLevel * INDENT_SIZE;
  const indentation = Array(indentSize)
    .fill(' ')
    .join('');

  code += indentation;
  lineLength = indentSize;
};
```

We can now also move the assignment of 0 to `lineLength` from `breakLine()` into `hardBreak()` — it would only get overwritten anyway.

Interaction with empty lines This indentation code works, but has an awkward interaction with empty lines. It is possible to generate code like this:

```
; limit ⊣
(xxxxx (
    xxxxx

xxxxx
    xxx))
```

An empty line only generates hard breaks, so the third identifier in this example is not indented at all.

We can avoid this by changing the empty line insertion code from

```
while (consecutiveBreaks < 2) hardBreak();
```

to

```
if (consecutiveBreaks === 0) hardBreak();
  softBreak();
```

, so only the first break (if present at all) is a hard line break, and the second break is a soft line break that indents the next line correctly.

This indentation could be considered a violation of the rule that the nesting level decreases *before* a `rightPar`, potentially printing something like

```
; limit ⊣
(xxxxxxxxx
  )
```

because we do not actually indent while printing the `rightPar`, but while printing the `emptyLine`, however this is a rare edge case that is probably not worth introducing an extra check for.

5.3 Formatting example

If we run some Lisp-like code based on arbitrary Lisp examples [18c] [14] [18b] — including the factorial example we used earlier — through the lexer and printer, we get the results shown in Figure 3 on a separate page.

```
; Input:
; factorial
(defun factorial(n)(if(=n 0)1(*n(factorial(-n 1)))))

; fibonacci
(defun fib(n)(if(lte n 2)1(+(fib(-n 1))(fib(-n 2)))))

; birthday paradox, already well-formatted
(defconstant yearsize 365)
(defun birthdayparadox (probability numberofpeople)
  (let ((newprobability (* (/
      (- yearsize numberofpeople) yearsize) probability)))
    (if (lt newprobability (/ 1 2))
        (1+ numberofpeople)
        (birthdayparadox newprobability
                         (1+ numberofpeople)))))

; Output:
; factorial
(defun factorial (n) (if (= n 0) 1 (* n (
        factorial (- n 1)))))

; fibonacci
(defun fib (n) (if (lte n 2) 1 (+ (fib (-
          n 1)) (fib (- n 2)))))

; birthday paradox, already well-formatted
(defconstant yearsize 365)
(defun birthdayparadox (probability
    numberofpeople) (let ((newprobability
        (* (/ (- yearsize numberofpeople)
            yearsize) probability))) (if (
        lt newprobability (/ 1 2)) (1 +
        numberofpeople) (birthdayparadox
        newprobability (1 + numberofpeople
        )))))
```

Figure 3: Exemplary formatting input and output code

6 Conclusion

We have learned about general-purpose and formatter-specific lexing, parsing and printing as operations that convert between source code, tokens and syntax trees in Chapter 2. For the essential printing operation that distinguishes formatting algorithms, we extracted four key aspects:

1. *spacing*
2. *indentation*
3. *hard line breaks* and
4. *soft line breaks*.

The separation of these concerns proved to be a very useful tool in handling the complexity of printing code and allowing for an incremental approach to the conception and implementation of an algorithm.

In Chapter 3, we uncovered the reasons why pretty formatting usually requires parsing and omitting this very step implies incurring severe drawbacks. Weighing those drawbacks up against the benefits, we determined a few select use cases where parse-free formatting may actually be a good idea.

We developed an algorithm to print the tokens of a Lisp-like language in Chapter 4, taking the four aspects of printing into account. Spacing rules were easily dealt with, although this may be a property of Lisp-like languages that lack advanced concepts such as unary versus binary operations. Hard line breaks were not difficult to handle either, but that too may change with concepts such as block statements and control flow statements. Indentation left some room for interpretation with no apparent right way of doing it. Unsurprisingly, soft line breaks turned out to be the hardest aspect, both in the variety of possible approaches and the quality level achievable with limited syntactic information.

We took our algorithmic groundwork and made a formatter implementation out of it in Chapter 5. We managed to realize the complete process of formatting with moderately complex and lengthy code, finally enabling us to probe parse-free formatting in practice. The given example shows that previously unformatted code is printed out in a significantly more readable, albeit not perfect, way. The spacing and indentation is appropriate, but soft line breaks could be placed much better; they often cut right through expressions with our implementation.

Code that was already formatted by a human, however, becomes significantly less readable because the soft line breaks are placed without awareness of the structure of the expressions formed by the surrounding tokens. This is not to blame on omitting the separation avoidance algorithm — in the example code, the relevant tokens were rarely separated anyway. Trying to break at a high point in the syntax tree like many established algorithms do may have helped with this issue.

We have managed to implement the basics of formatting without a parser. The example code had its empty lines preserved; comments and hard breaks are in the right places. The indentation works fine, but it is ruined by the soft line breaks that we did not manage to place sensibly for our Lisp-like language. If we want do better in this aspect, we will probably need to look at the expression tree structure in a way that essentially equates to parsing or at least give up on the strict approach of never looking at more than two consecutive tokens and never backtracking in the token stream.

Outlook It would be interesting to implement a less minimalistic parse-free formatter for a real, widely used programming language. Such a formatter could be compared with existing formatters for the same language to evaluate how significant the performance benefit is.

Some languages could perhaps be better formatted than a Lisp dialect with this method, because they have many small statements instead of the few large expressions from Lisp, reducing the impact of a suboptimal soft line breaking algorithm. It might also make sense to implement a formatter that specifically targets the use cases we expect a parse-free routine to be most suitable for — large batches of minified code.

Bibliography

[13] *go fmt your code*. Jan. 2013. URL: https://web.archive.org/web/20180411132030/https://blog.golang.org/go-fmt-your-code (visited on 04/11/2018).

[14] *Generating Fibonacci series in Lisp using recursion?* Apr. 2014. URL: https://web.archive.org/web/20180310151412/https://stackoverflow.com/questions/23065846/generating-fibonacci-series-in-lisp-using-recursion (visited on 03/10/2018).

[17a] *ECMAScript 2017 Language Specification*. 7th edition of ECMA-262. Ecma International. June 2017. URL: http://web.archive.org/web/20171208103931/https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (visited on 12/08/2017).

[17b] *Releasing Prettier 1.0*. Apr. 2017. URL: https://web.archive.org/web/20180119121016/https://jlongster.com/prettier-1.0 (visited on 04/11/2018).

[17c] *The Go Programming Language Specification*. Google LLC and Go contributors. June 2017. URL: http://web.archive.org/web/20171207185023/https://golang.org/ref/spec (visited on 12/07/2017).

[17d] *The Python Language Reference*. 3.6.4. Python Software Foundation. Dec. 2017. URL: http://web.archive.org/web/20171228181357/https://docs.python.org/3/reference/index.html (visited on 12/28/2017).

[18a] *acornjs/acorn: A small, fast, JavaScript-based JavaScript parser*. 2018. URL: https://github.com/acornjs/acorn/tree/5da6f356e93e30892c0fbe4d0c024897d9071e89 (visited on 03/10/2018).

[18b] *Birthday paradox - Code examples - Common Lisp - Wikipedia.* 2018. URL: `https://en.wikipedia.org/w/index.php?title=Common_Lisp&oldid=828130382#Birthday_paradox` (visited on 03/10/2018).

[18c] *Examples - Lisp (programming language) - Wikipedia.* 2018. URL: `https://en.wikipedia.org/w/index.php?title=Lisp_(programming_language)&oldid=816850347#Examples` (visited on 03/10/2018).

[18d] *JavaScript in VS Code.* Microsoft corporation and Visual Studio Code contributors. Mar. 2018. URL: `https://github.com/Microsoft/vscode-docs/blob/9af6ca98389dd2bca7f1da1c22ef1a667a2697de/docs/languages/javascript.md` (visited on 03/07/2018).

[BH13] Anya Helene Bagge and Tero Hasu. "A Pretty Good Formatting Pipeline". In: *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings.* Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Cham: Springer International Publishing, 2013, pp. 177–196. ISBN: 978-3-319-02654-1. DOI: `10.1007/978-3-319-02654-1_10`.

[Che+17] Christopher Chedeau et al. *Prettier Formatter Documentation — Rationale.* Nov. 2017. URL: `http://web.archive.org/web/20171230163559/https://prettier.io/docs/en/rationale.html` (visited on 12/30/2017).

[Che+18] Christopher Chedeau et al. *Prettier Formatter Documentation — Option Philosophy.* Feb. 2018. URL: `https://web.archive.org/web/20180411131501/https://prettier.io/docs/en/option-philosophy.html` (visited on 04/11/2018).

[Her+17] Dave Herman et al. *The ESTree Spec.* Oct. 2017. URL: `https://github.com/estree/estree/blob/1da8e603237144f44710360f8feb7a9977e905e0/es5.md` (visited on 10/05/2017).

[Hug95] John Hughes. “The design of a pretty-printing library”. In: *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text.* Ed. by Johan Jeuring and Erik Meijer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 53–96. ISBN: 978-3-540-49270-2. DOI: 10.1007/3-540-59451-5_3.

[Nud] Mark Nudelman. *less(1): opposite of more - Linux man page.* Free Software Foundation. URL: http://web.archive.org/web/20170911212415/https://linux.die.net/man/1/less (visited on 09/11/2017).

[Opp80] Derek C. Oppen. “Prettyprinting”. In: *ACM Trans. Program. Lang. Syst.* 2.4 (Oct. 1980), pp. 465–483. ISSN: 0164-0925. DOI: 10.1145/357114.357115.

[PF11] Terence Parr and Kathleen Fisher. “LL(*): The Foundation of the ANTLR Parser Generator”. In: *SIGPLAN Not.* 46.6 (June 2011), pp. 425–436. ISSN: 0362-1340. DOI: 10.1145/1993316.1993548.

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. *Adaptive LL(*) Parsing: The Power of Dynamic Analysis.* Mar. 2014. URL: https://web.archive.org/web/20180130161857/http://www.antlr.org/papers/allstar-techreport.pdf (visited on 01/30/2018).

[Rub83] L. F. Rubin. “Syntax-Directed Pretty Printing — A First Step Towards a Syntax-Directed Editor”. In: *IEEE Transactions on Software Engineering* SE-9.2 (Mar. 1983), pp. 119–127. ISSN: 0098-5589. DOI: 10.1109/TSE.1983.236456.

[Wad98] Philip Wadler. “A Prettier Printer”. In: *Journal of Functional Programming.* Palgrave Macmillan, 1998, pp. 223–244.

[ZB14] Vadim Zaytsev and Anya Helene Bagge. “Parsing in a Broad Sense”. In: *Model-Driven Engineering Languages and Systems.* Ed. by Juergen Dingel et al. Cham: Springer International Publishing, 2014, pp. 50–67. ISBN: 978-3-319-11653-2.

[Zhu+16] Zirun Zhu et al. “Parsing and Reflective Printing, Bidirectionally”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering.* SLE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 2–14. ISBN: 978-1-4503-4447-0. DOI: 10.1145/2997364.2997369.

Appendix

The lexer

```
const operators = ['=', '+', '-', '*', '/'];
const prefixes = ["'", '&'];
const booleans = ['true', 'false'];
const keywords = [
  'quote',
  'lambda',
  'defun',
  'let',
  'if',
  'and',
  'or',
];

const whitespaceRegex = /\s/;
const alphaRegex = /[a-zA-Z]/;
const numRegex = /[0-9]/;
const alphaNumRegex = /[a-zA-Z0-9]/;

const tokenize = code => {
  let position = 0;
  const tokens = [];

  let consecutiveBreaks = 0;

  while (position < code.length) {
    let char = code[position];

    // empty lines
    if (char === '\n')
      if (++consecutiveBreaks === 2)
        tokens.push({ type: 'emptyLine', value: '' });

    // whitespace
    if (whitespaceRegex.test(char)) {
      position++;
      continue;
    }
```

```
consecutiveBreaks = 0;

// parentheses
if (char === '(') {
  tokens.push({ type: 'leftPar', value: char });
  position++;
  continue;
}
if (char === ')') {
  tokens.push({ type: 'rightPar', value: char });
  position++;
  continue;
}

// operator
if (operators.includes(char)) {
  tokens.push({ type: 'operator', value: char });
  position++;
  continue;
}

// prefix
if (prefixes.includes(char)) {
  tokens.push({ type: 'prefix', value: char });
  position++;
  continue;
}

// number literal
if (numRegex.test(char)) {
  let numberLiteral = '';

  do {
    numberLiteral += char;
    char = code[++position];
  } while (numRegex.test(char));

  tokens.push({ type: 'numLiteral', value: numberLiteral });
  continue;
}

// identifier, keyword, boolean literal
if (alphaRegex.test(char)) {
  let name = '';

  do {
    name += char;
    char = code[++position];
  } while (alphaNumRegex.test(char));

  let type;
```

```
      if (booleans.includes(name)) type = 'boolLiteral';
      else if (keywords.includes(name)) type = 'keyword';
      else type = 'identifier';

      tokens.push({ type, value: name });
      continue;
    }

    // line comment
    if (char === ';') {
      let text = '';

      do {
        text += char;
        char = code[++position];
      } while (char !== '\n');

      tokens.push({ type: 'lineComment', value: text });
      continue;
    }

    // block comment
    if (char === '#') {
      let commentChar = code[++position];
      if (commentChar === '|') {
        let text = '#';

        do {
          text += commentChar;
          commentChar = code[++position];
        } while (!text.endsWith('|#'));

        tokens.push({ type: 'blockComment', value: text });
        continue;
      }
    }

    throw new Error(
      `unexpected character ${char} at position ${position}`,
    );
  }

  return tokens;
};
```

The printer

```
const MAX_LINE_LENGTH = 42;
const INDENT_SIZE = 2;

const allowsSpaceBefore = type => type !== 'rightPar';
const allowsSpaceAfter = type =>
  !['leftPar', 'prefix'].includes(type);

const print = tokens => {
  let code = '';

  let nestingLevel = 0;
  let lineLength = 0;
  let maxLineLength = MAX_LINE_LENGTH;

  let consecutiveBreaks = 2;
  let prevAllowsSpace = false;

  // helpers
  const breakLine = () => {
    code += '\n';
    consecutiveBreaks++;
    prevAllowsSpace = false; // line break replaces space
  };

  const softBreak = () => {
    breakLine();

    const indentSize = nestingLevel * INDENT_SIZE;
    const indentation = Array(indentSize)
      .fill(' ')
      .join('');

    code += indentation;
    lineLength = indentSize;
  };
  const hardBreak = () => {
    breakLine();

    lineLength = 0;
    maxLineLength = MAX_LINE_LENGTH;
  };

  // main loop
  for (const { type, value } of tokens) {
    let spaceBefore =
      prevAllowsSpace && allowsSpaceBefore(type);

    // nesting decrement needs to happen before print
    if (type === 'rightPar') nestingLevel--;

    // soft line breaks
    let printLength =
      value.length + spaceBefore; // (spaceBefore ? 1 : 0)

    if (lineLength + printLength > maxLineLength) {
      softBreak();

      // No space after all, just the break
      if (spaceBefore) printLength--;
      spaceBefore = false;
    }
    lineLength += printLength;
    maxLineLength = Math.max(lineLength, maxLineLength);

    // spacing
    if (spaceBefore) {
      code += ' ';
    }

    // print token
    code += value;

    // set previous information for next iteration
    prevAllowsSpace = allowsSpaceAfter(type);

    // hard line break
    if (type === 'emptyLine') {
      if (consecutiveBreaks === 0) hardBreak();
      softBreak();
    } else {
      consecutiveBreaks = 0;
    }

    if (type === 'leftPar') nestingLevel++;
    if (type === 'rightPar' && nestingLevel === 0) hardBreak();
    if (type === 'lineComment') hardBreak();
  }

  return code;
};
```