1 Abstract

Pretty-printers have been studied for over 50 years, and many efficient algorithms have been proposed and implemented. Algebras have been developed to describe pretty-printers formally, and pretty-printer libraries have been developed from those algebras. However, far less work was done to improve the capabilities of pretty-printers.

We propose that considering pretty-printing as a form of compilation simplifies the work required to construct a pretty-printer and allows programmers to define formatting specifications not possible previously. Pretty-printing code is equivalent to compiling text in some formal language onto another representation of the same text in the same language such that the output conforms to a set of predefined layout specifications. We will show that using an intermediate representation made up of two-dimensional *boxes* is sufficient and appropriate for describing the spacial layout of the formatted source code.

We propose a DSL with which users may define format specifications. We claim that, when used along with the grammar of the formatted language, this DSL is sufficient to generate pretty-printers that formats code by compiling it. To demonstrate these claims, we developed a system for generating such pretty-printers, and with it, we create pretty-printers for various languages and specifications.

We do not study the complexity of pretty-printers since we reduce the computational complexity of pretty-printing to that of compiling. Instead, we concentrate on generalizing the capabilities of pretty-printers and reducing the work required when creating a pretty-printer for a new language or a new formatting specification for and existing language.

Contents

List of Figures

3 Introduction

Pretty-printers are tools used to reformat source code. They are one of the tools that support programming and programming language, including compilers, interpreters, code browsers, etc. Pretty-printing is currently understood as a problem on its own, in a language-agnostic way, and some language-agnostic pretty-printers such as `astyle` (`http://astyle.sourceforge.net`) and `uncrustify` (`http://uncrustify.sourceforge.net`) are provided as standalone software.

Pretty-printers have been used in LISP as early as the 1960's [1] [2] [3], and have been studied in a language-agnostic setting as early as the 1980's [4][5] with many implementations presented since [6][7]. Pretty-printing has also been discussed as a special case of parsing in cases were the parser semantics are invertible [8]. However, these works have considered pretty-printing only as a tool for laying out textual documents under spatial constraints.

We propose that considering pretty-printing is a form of compilation, allows us to generalize pretty-printing to more than just a way to conform to spatial constraints. This generalized pretty-printing allows us to pretty-print source code by specifying not just layout constraints but also coding styles or naming conventions, for instance. Furthermore, one could pretty-print source code into another context such as LaTeX or even pretty-print code onto non-textual formats such as graphs and images. We will explore this idea by creating a generalized pretty-printer generator that lets the user define the transformation of the formatted source code using a *Domain Specific Language* (DSL).

1. Firstly we define the notion of generating parsers and generic *Abstract*

Syntax Trees (AST), which allow our system to be language-agnostic, so the user needs only provide the formal syntax of language using *Backus-Naur Form* (BNF) syntax.

2. Next, we define a *Box* language that we will use as the *Intermediate Representation* (IR) of our pretty-printer. We transform the input source code by manipulating this IR in a two-dimensional plane. We claim that this IR is sufficient to represent to describe the spatial and visual properties of the formatted source code.

3. To create our pretty-printer generator, we define a DSL called UGLY, which defines a set of operations that can be applied to input code with respect to the language structure defined in the formal syntax of the language. UGLY lets the user define a set of rules that controls how the pretty-printer will format the input code.

4. We use our generator to create and demonstrate pretty-printers for a few languages and coding styles.

5. Lastly, we compare the features and capabilities of our generated pretty-printers with those created by existing methods.

4 Background

4.1 Writing Parsers

Before parser-generators were introduced in the 1960s, implementing a programming language required constructing a parser for that language manually from the ground up. In addition to implementing the AST and semantics of the parser, programmers had to implement the parsing algorithm itself. The same is true for pretty-printing: Pretty-printers had to be written from scratch. However, in the case of parsing, this is no longer the case: Programmers have available optimized parser-generators as well as parser-combinators libraries. Parser-generators have existed since the late 1970's [9] and have been developed for most modern programming language [10][11][12].

In order to define the semantics of a parser created using a parser-generator, a specialized language is commonly used to instruct the parser generator on

how to create the parser semantics of the generated parser. This language is the DSL of the parser generator. To create a parser with a parser generator, the programmer needs to formulate the language specifications and the parser semantics in the DSL of the generator. The parser-generator will transform these specifications into an executable parser component and construct from them a parser by adding the mechanisms for reading input, writing the output and transferring control between the various parser components according to the provided specification. However, due to limitations in parser-generators, in situations where the syntax of the language is context-sensitive or when the programmer wishes change the way control is transferred between parser components or the way output is written, the programmer still needs to write small parts of the parser manually.

4.1.1 Parser-Combinators

Another method for creating parsers that do not require the programmer to use a DSL nor to implement the mechanics of parsing is *Parser-Combinators* [13]. This method allows the writer to compose complex parsers from simpler ones, saving the user from having to code by hand the specifics of how the component parsers work together. Parser-combinators embed a grammar as a first-class object in a programming language, which makes it possible to compose, modify and decorate parsers using tools defined in the programming language, making it easier to generate parsers by composition. The parser semantics, however, can never be derived from the grammar automatically, and so is either implemented manually, or a generic implementation is generated. This generic implementation of parser semantics will at most be able to record the structure of the syntax that was parsed.

Due to how parser-combinators embed the syntax into the programming language, they produce *Top-Down Parsers*. This kind of parsers derive the AST from the input text by starting with the root AST node and then deriving its children nodes using the production-rules in the grammar and the input string. To implement a parser this way, the programmer assigns a recursive

procedure for each production-rule in the grammar. The set of all of these procedures makes up the parser. Top-down parsers that break the grammar of the language into mutually recursive functions are called *Recursive Descent Parsers*.

4.2 Pretty-Printers

Pretty-printers have been studied and discussed extensively since they were introduced as integral parts of LISP systems in the 1960s. In this section, we discuss the leading works in the area which focus on improving the complexity and simplifying the implementation of pretty-printers.

4.2.1 Oppen's Algorithm

Oppen proposed a general algorithm for pretty-printing [4] that is intended to be a component in a text editor. The Algorithm first defines the input code as a stream of *Strings* and *Delimiters*. A string is a sequence of characters with no delimiters in it and delimiters are either blank characters (newline, space, tab, etc.) or special tokens, a left double bracket ($\boxed{[\![}$) and a right double bracket ($\boxed{]\!]}$), used to denote a logically contiguous block. These special delimiter tokens are used to keep the algorithm language agnostic. The algorithm does not describe a front-end which reads the input code and turns it into a stream of strings and delimiter. Instead, a programmer who implements the algorithm is expected to provide the front-end that adds the special delimiters to the stream based on the specific language semantics.

The algorithm is split into two processes that work in tandem, a *Scanner* and a *Printer*. The printer takes a stream and its length and prints it without exceeding the predetermined line width. It uses a stack to keep track of the current indentation width, called the *Indentation Stack*. Whenever it reads a token, t, it chooses how to act like so:

1. If t is a string, it prints t and keeps track of the remaining line space.
2. If t is a left double bracket token, it pushes another indentation on the

indentation stack and continues to the next token

3. If t is a right double bracket token, it pops the indentation stack and continues and continues to the next token.

4. If t is a blank, it checks if the length of the following block can fit in the remaining line space. If so, it outputs the blank and continues. Otherwise, it starts a new line indenting it according to the indentation stack and continues to the next token.

The scanner reads the input stream, consuming tokens and collecting them into a queue along with their lengths. The length 0 is associated with the token $]\!]$, the sum of lengths in a block is associated with the token $[\![$ and the length associated with blanks is the length of the block. When the scanner reads a $]\!]$ token, it sends the tokens in the queue to the printer process. However, if the scanner only invokes the printer at the end of a logical block, because the entire document can be a single logical block, its space-complexity would be linear in the size of the document, and data might only be printed once the entire document was scanned. Using the insight that the printer never writes more than the remaining space on the line at that point, the scanner can invoke the printer whenever it reads a length larger than the remaining space on the current line. This is done by replacing the queue used by the scanner with a data structure that can be read from both ends.

This algorithm requires time $O(n)$ and space $O(m)$ where n is the length of the input and m is the width of a line. It also outputs pretty-printed text as soon as a line been scanned. Many implementations have been written for this algorithm [6][14][15] and it is the basis for the Ocaml pretty-printer [16].

However, this algorithm serves only as the back-end of a pretty-printer and requires a dedicated scanner for every language. This scanner is used to tokenize the input code as well as to express logical blocks of tokens that should not, if possible, be broken into separate lines. It does this by inserting into the stream of strings and delimiters the double bracket tokens to enclose the blocks. This makes it harder to use an implementation of Oppen's algorithm with a hand-crafted front-end. Therefore, it is more likely that a complete

pretty-printer would instead be written from scratch.

Oppen's algorithm, as most pretty-printers, only deals with the question of where and how to partition code into lines. When formatting code, we might want to have a higher degree of control over the delimiters; some control over the strings themselves; or some non-uniform formatting of the input based on some factors other than line width.

4.2.2 Unifying Parsing and Pretty-Printing

The notion of generating pretty-printers was studied in the past. In 2011 Rendel and Ostermann explored the idea of uniting parsing and pretty-printing using invertible parsers [8].

Rendel and Ostermann defined a language of *Syntax Descriptors*. This language can be used to define a two-way relationship between abstract and concrete syntax. Using this method, creating a parser also defines a pretty-printer to unparse any ASTs created by that parser. To define their language, they use concepts derived from parser-combinators, defining operators that can be used to combine parsers into larger parsers. They define a set of combinators with which their invertible parsers are built:

1. $\diamond\!\!*\!\!\diamond$ - The *Catenation* combinator. The parser $(p_1 \diamond\!\!*\!\!\diamond p_2)$ is defined by parsing the first portion of the input using p_1 and parsing the second portion of the input using p_2 and returns the result as a pair. The pretty-printer $(p_1 \diamond\!\!*\!\!\diamond p_2)$ is defined to print the first member of a tuple using the printer p_1 and then print the second tuple member using p_2.

2. $\diamond\!\!|\!\!\diamond$ - The *Disjunction* (or *Choice*) combinator. The parser $p_1 \diamond\!\!|\!\!\diamond p_2$ is defined by applying both p_1 and p_2 to the input and storing both results. The pretty-printer for this combinator is defined by first trying to apply p_1 and if that fails to apply p_2.

3. $\diamond\!\!\$\!\!\diamond$ - The *Semantic* (or *Pack*) combinator. The parser $f \diamond\!\!\$\!\!\diamond p$ is defined by applying f to the parsed result returned by p. The pretty-printer for this combinator is defined to apply f^{-1} to the AST node

and then unparse the result. The definition for the pretty-printer of this combinator requires that f be an *Invertible Function* (also called an *Isomorphism*).

The requirement for using only invertible functions for parser semantics is relaxed a bit, requiring functions to only being *Partially Invertible*. A function f is partially invertible if the function f^{-1} can only be applied to parts of the range of f where it is invertible. This relaxed requirement is possible because we know that f^{-1} will only be applied to valid AST nodes, and so we do not mind if f cannot be inverted for some parts of the domain which are not valid AST nodes.

Using this syntax-descriptors language, they proceed to demonstrate how to construct simple parser and pretty-printer pairs for a trivial language to show that they can indeed produce a parser and pretty-printer pair. Next, they set out to show that limiting the semantics of the parsers to just partially invertible functions is practical. To do this, they define a set of combinators over partially invertible functions with which we can construct a complete algebra: identity, composition, product, subset and fold-left combinators were defined. Using these combinators, they constructed the syntax for a small arithmetic language that required more complex parser semantics than their original example language. The language parser must be constructed manually from the above combinators, but this way a constructs the pretty-printer as a part of the parser.

The drawback of this method is that constructing parsers using only partially invertible functions, while sufficient for most cases, can be difficult. This method only lends itself to constructing parsers using parser combinator techniques, so existing parsers for languages will have to be rewritten in order to use this technique. It is more likely for language developers to hand-craft a pretty-printer for their parsers, rather than rewrite those existing parsers.

Also, the nature of invertible functions tightly couples the parser and the pretty-printer, forcing us to choose a single format for the pretty-printer at

the time of constructing the parser. This tight coupling means a user of the constructed parser has very little control over the output of the pretty-printer.

4.2.3 Universal Pretty-Printers

Creating a pretty-printer is a difficult task, often requiring a language expert to write parts (or all) of the pretty-printer and define the desired output format. Commonly, these format specifications require hundreds of lines of code, and as languages grow, these specifications grow more complex. Also, since programmers create new languages relatively often, and because there is currently no way to generate a complete pretty-printer automatically, each new language requires a dedicated pretty-printer to apply those formatting specifications. Parr and Vinju set out to solve this issue by leveraging AI to create a universal pretty-printer, *CodeBuff* [17]

To pretty-print a new language, `CodeBuff` requires a corpus of code written in that language from which it learns the correct formatting for that language. When the user provides this corpus to `CodeBuff`, it tries to learn when programmers choose to add a blank or a line-break between any two symbols, and also how far to indent a new line. Learning which delimiter to insert between tokens from this corpus is called the *Training Phase* of `CodeBuff`.

The training phase for language D requires the user to provide a lexer and a parser for D and a corpus of code written in D. Using those, `CodeBuff` creates a statistical model that represents the coding style of the author of the corpus. This model consists of whitespaces and their types (space, tab, line-break, etc.), the token following each whitespace, and the features of that token. Such a set is called an *Exemplar* for the training phase. After collecting all exemplars, they are sent to the machine learning algorithm, which constructs from them a classifier that maps features of a token onto a likelihood that a specific type of whitespace precedes it.

`CodeBuff` uses *k-Nearest Neighbor* (kNN) machine learning model as a classifier. This kind of classifier compares an input features vector, X, to all the

features vectors in its training set, finding the k nearest vectors to X, X_k, and from X_k it chooses the vector which is most common in the corpus. This chosen vector describes an action for `CodeBuff` to perform on the token associated with the feature vector X. The possible actions are:

1. `nl` - Inject a newline
2. `p` - Inject a space
3. `(align, t)` - Left-align the token with `t`
4. `(indent, t)` - Indent the token from `t`
5. `nothing` - Do nothing with the token

With the classifier ready, `CodeBuff` formats code written in D by applying the action returned from the classifier for each token in the input.

Parr and Vinju show results of using their pretty-printer for common languages such as Java and SQL. They show that using machine learning to learn the formatting style is a viable way to create a universal pretty-printer. However, the effectiveness of the pretty-printer depends significantly on the quality and size of the training corpus, and constructing a good feature-set is non-trivial and critical to the success of the pretty-printer.

5 Pretty-Printing as compiling

We propose a new method for constructing pretty-printers. We consider a pretty-printer as a special kind of compiler that uses a parser that creates a language-agnostic AST as a front-end and a two-dimensional layout language as a back-end.

We use a simple parser that we designed only to create an AST describing the structure of the syntax of the language. This simple parser does not store any language-specific information like types for literal data or differentiating between different special-forms in the language. Using this kind of parser

allows us to generate the front-end with no knowledge of the semantics of the language that we are pretty-printing, and so our pretty-printer is nearly language agnostic. The only language-specific part the user needs to provide is the formal syntax of the language.

We use a two-dimensional visual language made up of *Box* objects as our IR. This visual language corresponds roughly to similar ideas found in Donald Knuth's *TexBook* [18]. Using this language, we create *boxes* from the terminals in the BNF of the language, and then compose the *boxes* in a two-dimensional plane to layout the code. We layout the *boxes* in a structure defined by the formatting the user chooses.

5.1 Goals

Our goal in creating this system is to show that using compiler techniques we can create pretty-printers with capabilities that match or exceed those found in existing methods, as well as making the task of creating pretty-printers more straightforward by removing any requirements from the user that do not relate directly to formatting the target language. We aim to show that we can achieve these improvements without introducing new limitations on the pretty-printer, the formatting specification or the target language.

While there are new pretty-printing methods that help create pretty-printers easily by embedding them in the parser of the language or learning the desired format from examples, there are still no pretty-printers that can enforce some common coding styles without requiring the implementation or the user to provide the pretty-printer with the semantics of the syntax of the language. For instance, aligning a list of assignment statements around the assignment sign in Pascal, like so:

```
program example;
var x, variable, newline: integer;
begin
```

```
  x        := 1;
  variable := 2;
  newline  := 10
end.
```

Defining this formatting style is impossible with existing methods (with the possible exception of `CodeBuff`). We will demonstrate how a user of our system can apply such formatting with ease.

When the syntax of a language has multiple concrete forms represented by a single AST node, the only method which is not language-specific that allows the user some control over which concrete syntax will represent the AST during pretty-printing is *Invertible Parsers*. No language-agnostic pretty-printer today can control this in a context-aware way. For instance, in Pascal, we can require that only multi-line comments be used outside the scope of a function, so the following code:

```
// Returns the square of <num>
function sqr(num: integer): integer;
begin
    // Set the return value
    sqr := num * num
end
```

Will be pretty-printed as:

```
{ Returns the square of <num> }
function sqr(num: integer): integer;
begin
    // Set the re-tun value
    sqr := num * num
end
```

The reason most pretty-printers do not support such formattings is that it requires adding a token to the code to terminate a multi-line comment or replace the token for starting a single-line comment with a token that starts a multi-line comment. Currently, there are no methods that can contextually apply such transformations. We will show how to apply such transformations using our system with just a few lines of code.

6 Grammar LaYout language - UGLY

We designed a DSL to allow the user to define the desired formatting of the input text, the *Universal Grammar LaYout* (UGLY) language. Using this DSL, the user can define the formatting of each syntactic element of the source code. This formatting is defined using operators. The pretty-printer applies these operators to the concrete syntax during runtime. Conditionals were added to the language to give users control over each operator is applied to a syntactic element.

UGLY code is compiled into a set of transformation functions that take a generic AST, and produce *boxes* form the AST nodes. Then, UGLY composes These *boxes* into a single *box* according to the requirements defined in the UGLY code. The resulting *box* contains the entire input text and is structured according to the user's desired format.

UGLY is a declarative language. Its code is made up of *Operators*, *Conditions* and *Settings*. During pretty-printing, UGLY transforms the text by applying operators to the input code. The values in UGLY are *literals*, *non-terminals* and *lists*. Conditions in UGLY control whether an operator is applied or not based on the hierarchy of the tree nodes. Operators define how UGLY will create *boxes* as well as how it will combine *boxes* together into a *box* AST. This *box* AST is the output of our UGLY pretty-printer

7 Compilers Pipelines

This system contains three compiler pipelines. The first is the pretty-printer, which pretty-prints code, and the other two are the BNF and UGLY compilers, which construct the front-end and the back-end of the pretty-printer, respectively.

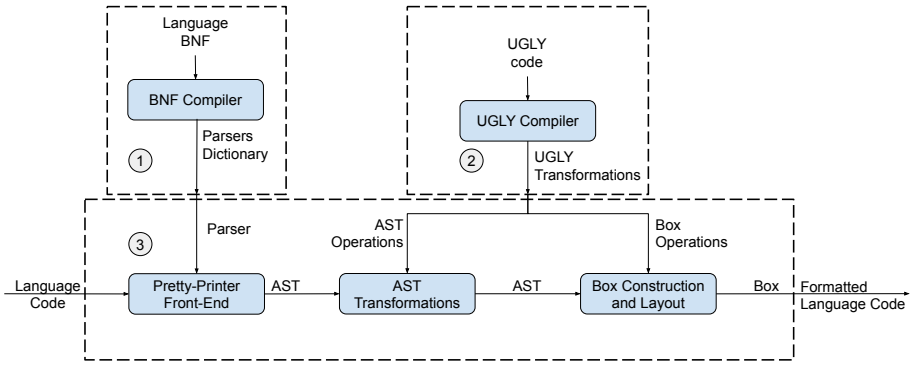


Figure 1: The three pipelines. (1) BNF compiler, (2) UGLY compiler and (3) Pretty-printer

7.1 BNF Compiler

The front-end of our pretty-printer generator is a compiler from formal grammars to language parsers.

A common language used to define the syntax of a formal language is *Backus-Naur Form* (BNF) [19]. We extended the BNF syntax to give users more control over how the parsers generated from BNF behave. These extensions help users define the syntax of a language with respect to how it will later be pretty-printed. We use this extended BNF syntax to generate parsers for formal languages. This extended BNF is a formal language for describing formal languages, *Meta-Language*. We then create a threaded compiler for this

meta-language. Our compiler takes as input an extended BNF and produces a set of parsers for the production-rules defined in the BNF.

We construct our generic parsers using parser-combinators since they allow us to use elements of the grammar of the language as first-class objects. Using the grammar as a first-class object, lets us combine parts of the grammar into a complete parser without having to represent the parser manually using the native data types of the programming language. We first construct parsers for the leaves of the syntax: literals and non-terminals, then we construct parsers defined with compound expressions: concatenations, disjunctions, differences, and modifiers by combining parsers for literals and non-terminals. We combine expressions with non-terminals into productions producing a map from non-terminals to their parsers. The user can either define which non-terminal is the root of the AST later or define a non-terminal named `S` which is the default root.

7.1.1 Generating Parsers

Our pretty-printer generator takes a BNF to generate the parser for our pretty-printer. This parser parses code in the language described by the BNF into generic ASTs. This component is our BNF compiler.

Because grammars may (and often do) contain recursive productions, it is impossible to generate a parser for each production-rule individually. We first need to set up a namespace containing all parsers in the BNF, and only then, using that namespace to resolve non-terminals, can we generate working parsers. To do this, we first generate parser constructors for every production-rule in the grammar. These parser constructor objects contain all the behavior of their resulting parsers, except resolving non-terminals on the *Right-Hand Side* (RHS) of the production they define.

Our namespace is a dictionary that maps non-terminals onto the parser constructor objects for their production-rule. The parser constructors take this dictionary and resolve all the non-terminals in the production-rule, generating

a parser.

One limitation of our parser generation method is that the simple implementation of parser-combinators creates top-down parsers that can only parse grammars with no left-recursion and no ambiguities. We could overcome these limitations with more advance parser-combinator methods [20], but this would increase the complexity of our system while not contributing to demonstrating its capabilities.

7.1.2 Generic Abstract Syntax

Our BNF compiler produces language parsers that, in turn, produce *Generic ASTs*. The language parsers produce these generic ASTs by outputting an abstract syntax describing only the relationship between syntactic elements. We designed this abstract syntax to allow the pretty-printer maximum flexibility in transforming the input. The abstract syntax only describes the hierarchy between syntactic elements, the production-rules used to parse these elements and the concrete syntax for the element.

Since we do not use this generic abstract syntax to define any language semantics, we do not need to assign any language-specific parser semantics. Because we do not have to define any parser semantics, we can automatically create parsers from the BNF, describing the syntax of the language using parser-combinators.

7.2 UGLY Compiler

Our system uses an UGLY compiler to take UGLY code and generate the code-generator for the pretty-printer. This code generator construct *boxes* from generic ASTS. Those *boxes*, in turn, can be turned into documents.

The Ugly compiler uses an UGLY specific abstract syntax. The root of the AST holds a list of `statement` nodes that hold the various statements in the source. The UGLY compiler converts the AST into a set of operations and conditions.

The UGLY runtime transforms conditions into predicates, which control the flow of the runtime, and operations are transformed into functions for UGLY to apply during runtime. This part of the system is where we leverage the generic ASTs created by the generic parsers. Because we use the same abstract syntax for any input language, we can design the UGLY runtime without knowing anything about the code that UGLY will be transforming. The only requirement is for the user to use the runtime correctly (i.e., Conform to the operators' signature).

7.3 Pretty-Printer

We construct the front-end of the pretty-printer using the BNF compiler. The generated parser parses the concrete syntax into a generic AST. Using a BNF compiler to generate parsers is how we decouple UGLY operations from any specific language.

The back-end takes a generic AST and traverses it, constructing a *box* from each AST node according to the defined operations and conditionals encoded into the back-end by the UGLY compiler. The *boxes* are composed together into a single *box* which describes the pretty-printed code.

7.4 Boxes

The pretty-printers we generate with our system use *boxes* as the IR on which they perform their operation. The *box* objects that comprise this IR are a two-dimensional visual object implementing a tree structure. Our implementation is based on the *box* languages presented in the TexBook [18] which was also the basis for the Caml and Ocaml languages' formatting libraries [16].

A *box* is a two-dimensional object with a width, height, structure, a decorator and data. A *box* can either be a *simple box* containing just text or a composition of nested, smaller, *boxes*. Our pretty-printer formats the generic AST constructing *boxes* from the AST and manipulating these *boxes*. We Manipulate *boxes* using the following operations:

1. *Constructing boxes* from string
2. *Combining boxes* to create larger *boxes*
3. *printing boxes* to produce a text document
4. *Decorating* them to alter the way they are printed

Constructing a *box* from a string of length `n` creates a *box* with height and width of 1 and `n` respectively[1]. The *box* created is a `simple` *box*.

To compose *boxes*, the user specifies the orientation[2] in which *boxes* are combined. There are seven types of structures for composite *boxes*:

1. *Horizontally Top-Aligned boxes* are constructed side-by-side, aligning their top edges
2. *Horizontally Bottom-Aligned boxes* are constructed side-by-side, aligning their bottom edges
3. *Horizontally Middle-Aligned boxes* are constructed side-by-side, aligning them along their horizontal center lines
4. *Vertically Left-Aligned boxes* are constructed one atop the other, aligning their left edge
5. *Vertically Right-Aligned boxes* are constructed one atop the other, aligning their right edge
6. *Vertically Center-Aligned boxes* are constructed one atop the other, aligning their vertical center lines
7. Or *Inserted boxes* are inserted into the structure of another *box* in the right-most bottom row

Composing *boxes* creates a new *box* containing a list of sub-boxes oriented as specified.

[1] A constructor that takes multi-line strings was also be created, but proved unnecessary for our uses.

[2] We did not find a coding style that required horizontally middle and bottom aligned *box* compositions, but we included it in our *box* language for completeness.

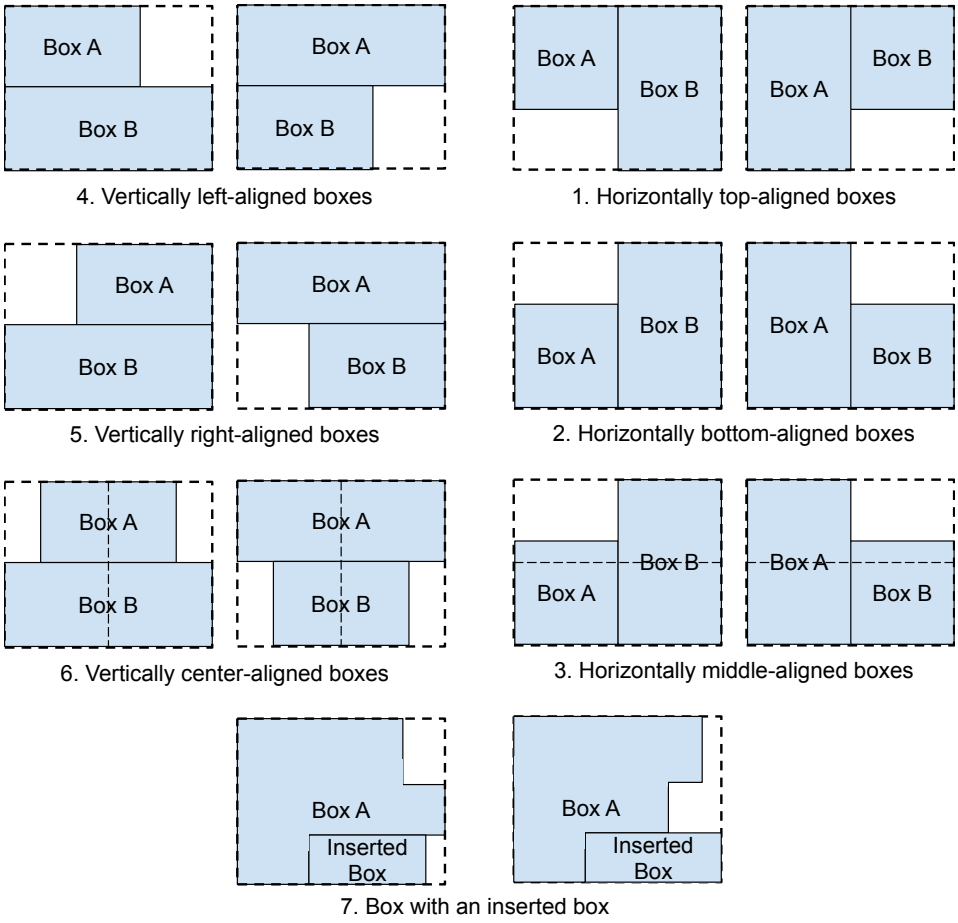


Figure 2: Types of *box* structures

The first six operations are for appending *boxes* to one another on their boundaries. We use the last operation, `insert`, to modify the *box* to include within its structure a new *box*, specifically in the bottom-right corner of the *box*. This operation is required to implement some common styles in programming languages, as seen in section 8.

To decorate a *box* we need to provide a string transformation function with which we create the decorated *box*. This decorated *box* applies the given transformation to the elements of the *box* when they are accessed.

Finally, we print a *box* by recursively printing out the *boxes* from which it

was built and appending the resulting text according to the *box* structure. A simple *box* is printed by outputting the string it contains.

8 Implementation

8.1 Extending BNF

Our generic parser generator takes as input a formal syntax in BNF. This formal syntax for our BNF can be found in appendix A.

We use our extended BNF to generate parsers that, in turn, generate generic ASTs. The pretty-printer manipulates these ASTs during formatting. We added syntactic forms to the BNF syntax to help the user control the formatting, as described in section 7.1.1.

We added three syntactic forms to the standard BNF syntax:

1. Case-insensitive literals, enclosed in quasi-quotes (`)
2. Ranges, denoted by a dash (-) between two single-character literals
3. a difference operator, denoted with a slash (\) sign.

We added *Case-Insensitive Literals* to allow the user to reduce the number of non-terminals used to allow case-insensitive syntactic elements. We added *Ranges* to remove the need for large disjunctions that represent ranges (e.g., $'a' -' z'$ is much easier to read and write compared to $'a'|'b'|...|'z'$). We added the *Difference* operator to allow using wild-card non-terminals (i.e., `<any-char>`) for comments and strings. Without the difference operator, these syntactic elements would have required much larger production-rules.

8.2 Generating Parsers

We use our BNF syntax to generate parsers. Because of this, we added features to the BNF syntax to support the generated parser. We added features for *Skipping*, *Trace Debugging* and *Callbacks*. The skip feature is an expression

modifier (similar to Kleene-Star), and we denote it with a hash ($_{\square}$) token. When a user applies this operator to an expression, the parser will omit any concrete syntax derived by that expression from the AST. This operator is useful for reducing the size of the AST to simplify it when UGLY programmers debug their UGLY code.

We added the *Trace* modifier to allow trace-debugging the generated parser. The extended BNF syntax allows the user to either trace a production-rule by modifying the LHS of the rule. The syntax for trace debugging is a dollar ($\boxed{\$}$) character. The trace modifier will cause the parser to display which tokens it consumed during parsing. Users should use this feature to locate bugs in their BNF input.

We added *Callbacks* to the syntax as an expression modifier denoted with an identifier (the name of the callback) enclosed in braces. We use callbacks to assign operations to syntactic elements in the grammar without using UGLY code. Defining callbacks for syntactic elements allows the user to run the generated parser for a specific non-terminal and apply the pretty-printer to the AST fragment created by that non-terminal. Callbacks are useful when the user wants to test the effect of operators without having to write a syntactically complete input code. Callbacks are an application of the transformations of the pretty-printer. These are the callbacks the user can apply:

1. `align-to-second` - When the user applies this callback to a non-terminal that produces a list, the list elements are vertically-aligned and padded such that they are aligned vertically around the second syntactic elements in the list elements.
2. `center-off-second` - Similar to `align-to-second` except that this operator adds the padding before the second syntactic element, so all the vertically aligned list elements are also aligned along their left edge.
3. `downcase` - Converts all the alphabetic characters derived by the expression to lower case.
4. `indent` - Inserts a tab before the text derived by the expression.

5. `line-wrap` - Applies line wrapping on long lines derived by the expression.

6. `no-wrap` - Disables line wrapping for a sub-tree derived by the expression. Users can use `no-wrap` to control `line_wrap`.

7. `noop` - The identity transformation.

8. `require-space` - When a user applies this callback to an expression, whitespaces are appended and prepended to the text derived by that expression. The operator adds spaces as necessary such that text starts and ends with at least one whitespace.

9. `single-line` - Replaces newlines with spaces in the derived text.

10. `skip` - A callback implementation of the *skip* construct. This operator removes the AST node to which it is applied.

11. `spaced` - Appends and prepends whitespaces to the text derived by the expression.

12. `squeeze` - When applied to an expression with the Kleene-Star or Kleene-Plus modifier, this callback will replace the list of derived elements with the first element of the list.

13. `upcase` - Converts all the alphabetic characters derived by the expression to upper case.

14. `vertical` - When the user applies this callback to a nonterminal that derives a list, this callback causes every element in the list to be printed on a separate line.

15. `vert-insert` - Same as `vertical`, but the last element in the list is appended to the end of the line of the second-to-last element.

16. `vert-list` - Same as `vertical`, but can be applied to a parent node of the list. The callback will traverse the tree to find the first nested list and apply `vertical` to that list.

We constructed a parser for this extended BNF syntax manually using parser-combinators, an we use it as the front-end for our extended BNF compiler. The BNF compiler generates a parser constructor for each of the production-rules in the grammar. It then returns a mapping from the non-terminals to

these parser constructors. These parser constructors take the mapping from non-terminals to parser constructors, and with that, they construct a generic parser.

The front-end of our pretty-printer uses the following abstract syntax when parsing input:

```
type ast =
  | Pack of string * bnf_ast
  | Nt of string * bnf_ast
  | List of bnf_ast list
  | Leaf of string
```

Generic abstract syntax

We use the three nodes: `Nt`, `List` (for catenation, Kleene-star and Kleene-plus) and `Leaf` (literals) to describe how the grammar breaks the concrete syntax into syntactic elements. We use the `Pack` node is to support the callbacks extension.

Disjunctions in BNF only describe choices. A disjunction on the RHS of a production-rule means that a non-terminal may be derived in more than one way. Our pretty-printer does not use any information about how input is parsed, so we do not need to describe disjunctions. Instead, our pretty-printer only needs to know the relationship between the syntactic elements in the input. Because our pretty-printer does not use the input described by disjunctions, we do not include disjunctions in the abstract syntax.

8.3 Boxes

To define our *box* AST as our IR, we needed a recursive structure that can hold data as well as decorate that data as defined in some of our transformations. Such a structure is best defined in our implementation language, `Ocaml`, using the following `box` class:

```
class box :
    (* rows -> cols -> structure -> getter -> box *)
    int -> int -> box structure -> (int -> int -> char) ->
    object ('a)
      val box_struct : box structure
      val cols : int
      val rows : int
      val getter : int -> int -> char
      val decorator : char -> char
      val tags : string list
      method box_struct : box structure
      method decorate : (char -> char) -> 'a
      method get : int -> int -> char
      method tag : string -> 'a
      method to_string : string
    end
```

Box class

The behavior of a *box* is almost entirely defined by the `getter` member, and most operations on *boxes* are manipulations of this object. We construct a simple *box* from a string by creating a getter for that input string:

```
let string_to_box s =
  (* getter wraps String.get with some dimension checks *)
  let getter s r c =
    if (r != 0) then ' '
    else try String.get s c
         with e -> ' ' in
    new box 1 (String.length s) Simple (getter s)
```

Box construction

To compose *boxes* together we need to create a new getter for the resulting composite *box*. We impalement this composite *box* getter by creating a selector function that chooses which *box* in the composite contains the requested row and column. This is an example of such selector method for the case of horizontal composite *boxes*:

```
let rec horz_select_box boxes col =
  match boxes with
  | [] -> (col, empty_box)
  | b::[] -> (col, b)
  | b::boxes -> if col < (b#width) then (col,b)

let horz_compose_top boxes structure =
    let r = List.fold_left (fun r b -> max r (b#height)) 0 boxes in
    let c = List.fold_left (fun c b -> c+(b#width)) 0 boxes in
    let s = HorzTop boxes in
    (* g dispatches requests to encapsulated getters *)
    let selector = fun row col ->
      let col,b = (horz_select_box boxes s) in
      (b#get row col) in
    new box r c s selector;;
```

Box composition

We added the ability to *decorate* a *box* to support coding styles that require changing input code, such as changing text to upper case. The *box* object will apply the function stored in the `decorator` member to the result of the `getter` when the data in the *box* is accessed.

To write out the *box* back into a string, the user uses the `to_string` method which iterates all the points in the *box* based on the `rows` and `cols` fields, and invokes the `getter`

A *box* can have `tags`. These are strings used to identify *boxes* and are useful for debugging. Tags do not change the behavior of a *box*.

8.3.1 Inserting *boxes*

To support some coding styles we implemented a special operation that can be used to combine *boxes*, *Insert*. This operation inserts a *box* to the bottom right empty position in the *box*. To demonstrate this, we will pretty-print an *S-expressions* (sexprs) list defined with the production-rule:

```
<list> ::= '(' <s-expression>* ')'
```

And we will pretty-print the following sexprs

```
(define x 1)
```

The user might require the pretty-printer to print this list vertically, having each list element in a different line like so:

```
(define
  x
  1)
```

Naturally, the *boxes* will be constructed for each syntactic elements like so:

1. *box* for `(`
2. *box* for `define`
3. *box* for `x`
4. *box* for `1`
5. *box* for `)`

Due to the way we define `<list>`, the sexprs `define`, `x` and `1` are combined into a single *box*, representing the production of `<s-expression>*`. The result will be a single *box* for the left-parenthesis, another *box* for the list of sexprs and a *box* for the right-parenthesis, like so:

1. *box* for `(`
2. A vertical composite *box* for `define`, `x` and `1`
3. *box* for `)`

These three *boxes* will then be combined horizontally to produce[3]:

```
######VL1#
#(define #
# x      #
# 1     )#
######3,8#
```

We determine the width of the nested *box* by the width of its longest sub-box (i.e., the width `define` symbol), and when we append the *box* containing the closing parenthesis token, we append it to the boundary of the composite *box*. This is how we get those whitespaces between the `1` sexpr and the `right parenthesis`.

To combine the three *boxes* into our desired *box* structure, we use the `insert` operation. Instead of appending the *box* holding the `right parenthesis`, it will recreate the composite *box* by inserting the `right parenthesis` into it at the bottom right position, producing two *boxes* to be combined:

1. A *box* for `(`
2. A vertical composite *box* for `define`, `x`, `1` into which the *box* for `)` is inserted to the right of the *box* containing `1`.

These *boxes* are combined horizontally to produce:

```
#####VL1#
#(define#
# x     #
# 1)    #
#####3,7#
```

[3]We use pounds (#) frame to display the bounds of the *box* clearly. The numbers and letters on the top of the frame denote the structure and number of *boxes* in the *box* (non-recursive). The numbers on the bottom denote the width and height of the *box*.

We implemented this `insert` operation by traversing the *box* AST, keeping track of the orientations in the path. Once we reach the leaf, we add to it the inserted *box* and backtrack reconstructing the composite *boxes* with the same orientation. Iterating the *box* structure is done like so:

```
let boxes = match b#box_struct with
| Simple ->  [b;insert]
| (VertLeft l
  | VertRight l
  | VertCenter l
  | HorzBot l) -> insert_last l
| (HorzTop l
  | HorzMid l) ->
   let anchor_height, anchor_idx = find_insert_anchor l in
   if (anchor_idx = (List.length l) - 1)
   then insert_last l
   else List.mapi (fun i b -> if (i = anchor_idx)
                              then (add_insert b anchor_height)
                              else b)
                  l
```

This method collects all *boxes* along a path in the AST of *boxes*. The traversal ends when it reaches a simple *box*, called the *Anchor*, to which we add the insert.

The anchor *box* is the absolute bottom-most *box* in the structure, and in that *box* we recursively find the bottom-right edge. When we reach a simple *box*, we append the inserted *box* to its right. To find the path to this *box* structure we always choose the bottom most *box* in a vertically oriented *box* or the leftmost in bottom-horizontally oriented *boxes*, and we use `find_insert_anchor` on horizontally oriented *boxes* to find the highest *box*:

```
let find_insert_anchor boxes =
  List.fold_lefti (fun (m, mi) i b -> if (b#height >= m)
                                      then (b#height, i)
```

```
                                    else (m, mi))
                (-1,-1)
                boxes
```

We search the highest *box* because the bottom-most *box* in the structure must be nested inside the highest *box* in the structure.

8.4 UGLY

We wrote the grammar of our DSL in our extended BNF. UGLY is a declarative language made up of *Statements*. A statement can be either an *Operation*, a *Setting* or an *Under* block.

UGLY code is a set of instructions defining how to transform code based on the grammar of the language in which the code is written. UGLY will not transform a non-terminal in the grammar wit no operations associated with it in the UGLY code.

8.4.1 UGLY Parser

The first stage of our UGLY compiler is a hand-written parser we use to parse UGLY code into an UGLY-specific AST. The syntax for UGLY can be found in appendix B.

The abstract syntax for UGLY is:

```
type statement =
    | Setting of string * expr
    | Op of string * expr list
    | Under of bool * expr * statement list

  type expr =
    | Num of int
    | List of expr list
    | Lit of string
```

```
    | LitCI of string
    | Nt of string
```

UGLY abstract syntax

We describe the global settings with a `statement` of type `Setting`. Global settings control line width, tab width, the newline character and the space character. We describe the applying operators to syntactic elements with a `statement` of type `Op`. Applying operators is how the user tells UGLY to transform a syntactic element during pretty-printing. We describe conditioning the application of operators with a `statement` of type `Under`. Using `Under` statements, the user conditions the application of operators to a specific sub-trees in the AST. Using this construct, the user can define contextual formatting of code.

Operations and `under` conditions in UGLY support non-terminals, literals and lists of non-terminals and literals. In the abstract syntax, we use the type `expr.Nt` for non-terminals, `expr.Lit` and `expr.LitCI` for literals and `List` for lists. The remaining expression type, `expr.Num`, denotes numbers which may be found under `Setting` nodes to define line width and tab width. We do not use `expr.Num` in operators because none of the implemented operators required any numeric parameters. If an operator is added that uses numeric parameters, no changes need to be made to the abstract syntax or the parser.

8.4.2 Settings

In UGLY, `settings` are global definitions and affect to all operations. We use `settings` to define maximal line width for wrapping, tab width for indentations, newline characters for wrapping and appending/replacing text and whitespaces for indentation and appending/replacing text. The syntax for settings is a leading hash (`#`) character followed by the name and value of the setting. For instance:

```
#line_limit=80
#tab_width=2
```

The above UGLY statements will create a pretty-printer that limits the line width of the formatted text to 80 characters and indent lines with two spaces.

In UGLY, settings have a lexical scope that extends over to the end of the code or to the next setting statement that shadows an earlier setting value. This shadowing allows the user to define different settings for groups of ugly statements (e.g., Changing tab-width from 2 to 4 when indenting certain syntactic elements).

8.4.3 Under-Statements

An `Under` expression is transformed by the UGLY compiler to predicates that test the hierarchy between nodes in the generic AST. Using these predicates, the user can restrict the execution of operators in the block nested in the `Under` AST node. We transform an `Under` expression to predicates by constructing a function that tests whether the AST node it receives has correct name or content:

```
let rec expr_to_pred neg e =
        match e with
        | UGLY.Nt s -> (fun e ->
          match e with
          | AST.Nt (str,_) ->
             (s = str) == neg
          | _ ->
             false)
        | (UGLY.Lit s | UGLY.LitCI s) -> (fun e ->
          match e with
          | AST.Leaf str -> (s = str) == neg
          | _ -> false)
```

8.4.4 Transforming Input Code

Our pretty-printer only transforms the input code as requested by the UGLY programmer. To add a transformation to the code, the programmer needs to apply an operation to a syntactic element. We implemented most operations in UGLY by adding the `Pack` node in the correct location in the AST, calling a callback to be applied on the AST when we construct a *box* from it. For instance, indenting a syntactic element in UGLY with the `indent` operator is implemented by:

```
let op_indent pack_params name preds params layout =
  let apply_to = get_nt (List.nth params 0) in
  let indent_width = NumParam (layout.tab_width) in
  let rec op preds ast =
    let preds = test_preds preds ast in
    let apply_op = preds == [] in
    match ast with
    | Leaf _ -> ast
    | List asts -> List (List.map (fun ast -> op preds ast) asts)
    | Nt (nt, ast) when (nt = apply_to) && apply_op ->
       Pack("indent", [indent_width], Nt(nt, op preds ast))
    | Nt (nt, ast) ->
       Nt(nt, op preds ast)
    | Pack(n, p, ast) -> Pack(n, p, op preds ast) in
  op preds
```

The `params` argument holds the parameter of the UGLY operation. In this case, which non-terminal should be indented. The `layout` argument holds the definitions of how our document should be laid out (line width, tab width, newline character and whitespace character). The `preds` list holds the conditionals defined by any enclosing `under` statements.

We traverse the AST, testing the predicates. We remove any satisfied predicate from the list of predicates when moving to the children nodes in the AST. When the list `preds` is empty, if we find the target non-terminal, we pack that

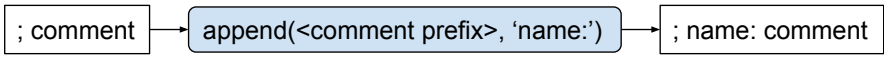
AST node with an `indent` callback.

To apply the `indent` callback to a *box* we append an appropriately wide blank *box* before it[4]:

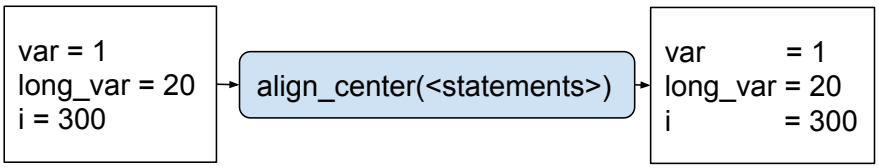
```
let indent params ast name =
  let NumParam(width) = List.nth params 0 in
  let b = ast_to_box ast in
  let indent = string_to_box (String.make width ' ') in
  horz_top [indent;b]
```

8.4.5 Operations in UGLY

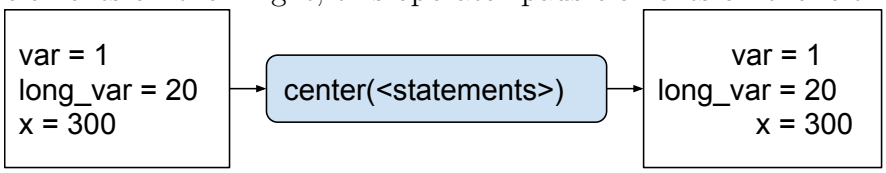
1. `append(nt, lit)` - Appends `lit` to the string derived by the non-terminal `nt`.



2. `align_center(nt)` - Takes the list of syntactic elements derived by `nt` and structures their *boxes* vertically, aligning their centers to the same column. The leftmost elements of the list are padded on their right end to facilitate the centering.

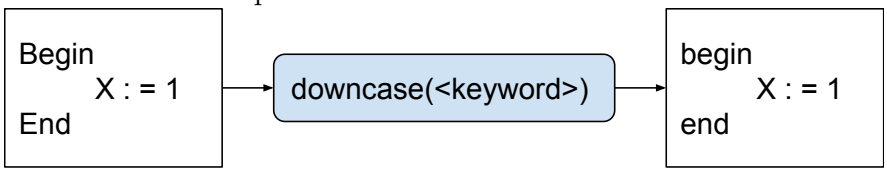


3. `center(nt)` - Same as `center` except that instead of padding the first elements on their right, this operator pads elements on the left.

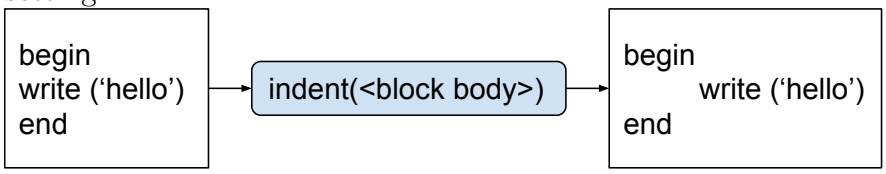


[4]The function `ast_to_box` is the main transformation function that takes generic ASTs and transforms them into *boxes* according to the callbacks in the AST

4. `downcase(nt, lit)` - Turns all alpha-characters under `nt` into their lower-case counterparts.

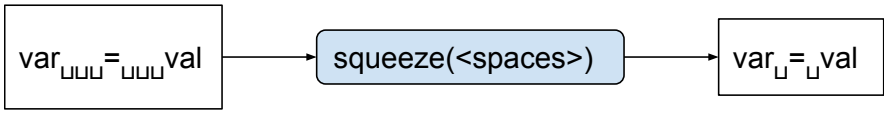


5. `indent(nt)` - Indents the sub-tree rooted in `nt` according to the `tab_width` setting.

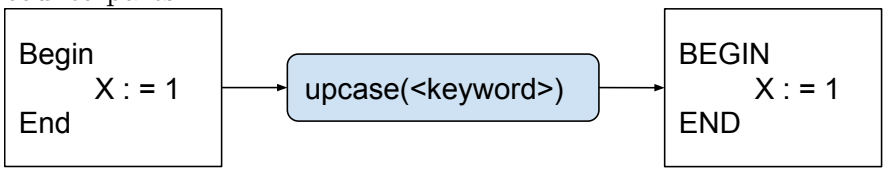


6. `skip(nt)` - Omits anything derived by `nt` from the output.

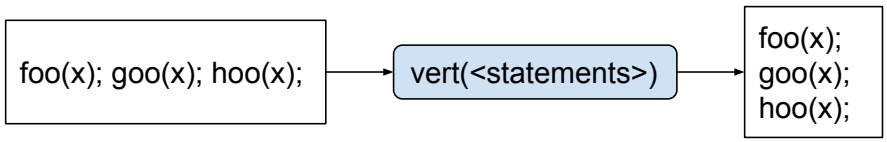
7. `squeeze(nt)` - Replaces a multiple consecutive identical syntactic elements derived by `nt` with a single instance of the syntactic element.



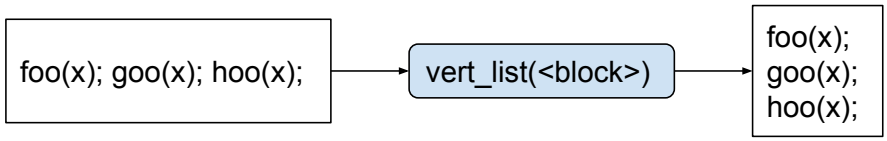
8. `upcase(nt)` - Turns all alpha-characters under `nt` into their upper-case counterparts.



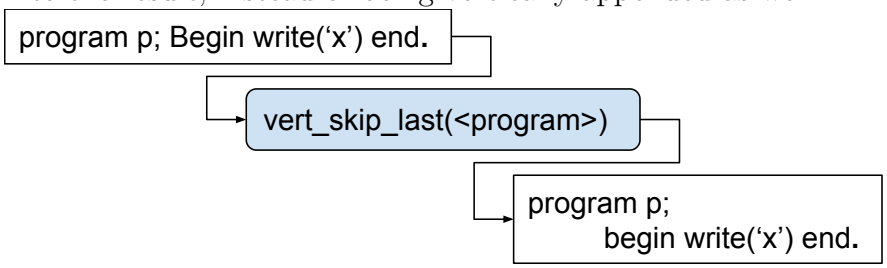
9. `vert(nt)` - The syntactic elements derived directly by `nt` are constructed horizontally. There is a version of this operator, called `verts` which takes a list of non-terminals instead of just one. This version of the operator is useful since we usually apply `vert` to many syntactic parts of a language.



10. `vert_list(nt)` - Same as `vert`, but the operation is applied to the first nested list under `nt`.



11. `vert_skip_last(nt)` - Same as `vert`, but the last element is ~insert~ed into the result, instead of being vertically appended as well.



To support line-wrapping, we added a pair of special operators to UGLY:

1. `line_wrap(nt)` - `line_wrap` enforces the maximal `line_limit` setting. We apply this operator automatically to the *box* created after applying all previous operators. It is not applied by the user.

2. `no_wrap(nt)` - Disables line wrapping for elements under `nt`. Useful for languages that support line wrapping, when the user wants to exclude some language constructs from line limitation (e.g., Comments or strings).

We exclude `line_wrap` and its associated operator `no_wrap`, from the list of operators the user can apply because line wrapping might modify the structure of the *box* AST, and such modifications would break any relationship between the *box* structure and the syntax of the input language making it very difficult to predict the effect of operators.

8.5 Worst-Case Complexity

8.5.1 Front-end

We generate our front-end parsers using an implementation of parser-combinators which produces top-down parsers. Because these parsers construct the AST by deriving the root of the tree down to the leaves, they need to choose which production-rules to apply whenever a non-terminal which has more than one production-rule (i.e., a disjunction) is derived. When deriving such a non-terminal, the parser may choose a production-rule which rejects the input token stream. At this point, the parser will need to *Backtrack* by discarding the sub-tree rooted in *nt* and try another production-rule.

Backtracking parsers require, in the worst case, time exponential in the length of the input [20], therefore our generated parsers will also require time exponential in the length of the input. We chose to use parser combinators for our front-end despite this drawback. Parser combinators integrate easily into our system, and their time complexity did not interfere with demonstrating the capabilities of our system. A more efficient implementation of this front-end can be created using more efficient implementations of parser combinators or entirely different parser generating methods such as generating DSL code for an efficient parser-generator (such as *lex* & *yacc*) and then using that parser-generator behind the scenes to produce the front-end of our pretty-printer.

8.5.2 Back-end

The back-end of our pretty-printer is made up of a set of transformations that we apply to the generic ASTs or *boxes* AST. All our transformations, at most, construct a constant multiple of *boxes* from any single AST node. Specifically, when aligning a `List` AST node to a central element with the `center` (and similarly `align_center`) operation, each AST node in the list is described with 4 *boxes*:

1. Padding
2. Left-hand-side elements

3. Center element

4. Right-hand-side elements

Finally, the `List` node itself is described with 3 *boxes*, each describing a column of *boxes*. The left-hand-side *boxes*, center *boxes* and right-hand-side *boxes*.

After all of the operators are applied, we apply the `line_wrap` operator to the resulting *box* also using time linear in the number of *boxes* in the structure. The `line_wrap` operator only traverses the tree once. At each *box*, it measures the lengths of its sub-boxes and, if a line break is needed, breaks the *box* into three *boxes*:

1. Top row

2. Indentation

3. Bottom row

Since operations at most traverse the *box* structure a constant number of times while constructing, combining and decorating *boxes*, they will require at most $O(m)$ time to transform m *boxes*, and because there is a constant number of *boxes* constructed for each AST nodes, $m = O(n)$, so the operations will run in $O(n)$ time for an AST of size n.

Because we load the entire AST into memory in order to transform it and we never keep multiple copies of the same AST node, the space-complexity of our back-end is linear in the size of the AST.

9 Demonstration

To run our pretty-printer, the user needs to provide three inputs:

1. The specifications in UGLY

2. The source language BNF

3. And the code in the source language.

9.1 Pretty-Printing JSON

A common use for pretty-printers is to pretty-print data, rather than code. A simple example would be JSON. The JSON grammar we used can be found in appendix C. We will take the following JSON string:

```
{"Name": "Code Generators", "Uses": "Generating code from DSL or
 ↪  data",
 "Types": [{"Family": "Parser-Generators",
     "Introduced": "1960s",
     "Common Use": "Generating a compiler's parsers front-end",
     "Members": [{"Name": "YACC","Language": "C","Published":
      ↪  1975},
         {"Name" : "BISON","Language": "C","Published": 1985},
        {"Name" : "ANTLR","Language": "Java","Published": 1992},
         {"Name" : "JavaCC","Language": "Java","Published":
          ↪  1996}]},
    {"Family": "Lexer Generators",
     "Introduced": "1970s",
     "Common Use": "Generating lexers/scanners",
     "Members": [{"Name": "flex", "Language": "C", "Published":
      ↪  1987},
         {"Name": "re2c", "Language": "C/C++", "Published":
          ↪  1994}]},
    {"Family": "Pretty-Printer Generators",
     "Introduced" : "2010s","Common Use" : "Generating
      ↪  beautifiers",
     "Members": [{"Name": "UGLY", "Language": "UGLY DSL",
      ↪  "Published": 2019}]}]}
```

JSON string

This JSON string was written manually on an editor that does not support any formatting of JSON. It is tough to read because the indentation, line breaks and other whitespaces are non-uniform.

A simple formatting rule-set would be for objects and arrays to be ordered vertically and for there to be exactly one space after a colon and no spaces before it. The UGLY code for this is:

```
verts([<object members>, <members>, <array elements>
       <elements>])
replace(<SPACES>,'')
append(<MEMBER_COLON>, ' ')
```

UGLY for JSON strings

We apply `verts` to lists of syntactic elements. This operation instructs the system to construct these lists using vertical compositions of `boxes` aligning their left edges.

`replace` takes a non-terminal or a literal and replaces it with a new literal (in this case with the empty string). The `append` command is similar, except that it appends a literal to the non-terminal (using the `insert` operation) instead of replacing it.

To generate a JSON pretty-printer, the above UGLY code is used along with the JSON syntax (see appendix C). That pretty-printer can then be used on the above JSON string to produce:

```
{"Name": "Code Generators",
 "Uses": "Generating code from DSL or data",
 "Types": [{"Family": "Parser-Generators",
            "Introduced": "1960s",
            "Common Use": "Generating a compiler's parsers
             ↪  front-end",
            "Members": [{"Name": "YACC",
                         "Language": "C",
                         "Published": 1975},
                        {"Name": "BISON",
                         "Language": "C",
                         "Published": 1985},
```

```
                {"Name": "ANTLR",
                 "Language": "Java",
                 "Published": 1992},
                {"Name": "JavaCC",
                 "Language": "Java",
                 "Published": 1996}]},
 {"Family": "Lexer Generators",
  "Introduced": "1970s",
  "Common Use": "Generating lexers/scanners",
  "Members": [{"Name": "flex",
                 "Language": "C",
                 "Published": 1987},
                {"Name": "re2c",
                 "Language": "C/C++",
                 "Published": 1994}]},
 {"Family": "Pretty-Printer Generators",
  "Introduced": "2010s",
  "Common Use": "Generating beautifiers",
  "Members": [{"Name": "UGLY",
                 "Language": "UGLY DSL",
                 "Published": 2019}]}]}
```

Pretty-printed JSON

9.2 Pretty-Printing Pascal

We will use a subset of the Pascal language to illustrate using the system. The BNF for this subset of the language can be found in appendix D.

We will demonstrate the pretty-printer on the following code:

```
Program p; Var y:integer; x:string Begin y:=1; write('some long string', x) End.
```

First we will only run the built-in line wrapper in the pretty-printer, so our UGLY code will be just one line specifying a limited on linewidth:

```
#line_limit=50 ; Limit lines to 50 columns
```

Executing the pretty-printer we get:

```
Program p;
```

```
Var y:integer; x:string
    Begin y:=1; write('some long string', x) End.
```

The pretty-printer measures the syntactic elements in the AST to find the element which overflows the line. It then rearranges the *boxes* to create a structure that does not overflow.

Just breaking overflowing lines is likely not all of the formattings we want to apply. To demonstrate how to apply greater control to the format, we will use a slightly more complex source code input. Below is a program that calculates permutations. It is written without any formatting except for line wrapping, so that it would fit in a page of this document (60 columns).

```
program permutations; var p: array[1 .. 10] of integer;
is_last: boolean; n: integer; demo: string; long_variable:
integer procedure next; var i, j, k, t: integer begin
is_last := true; i := n - 1; while i > 0 do begin if p[i] <
 p[i + 1] then begin is_last := false; break end; i := i - 1
 end; if is_last = false then begin j := i + 1; k := n;
while j < k do begin t := p[j]; p[j] := p[k]; p[k] := t; j
:= j + 1; k := k - 1 end; j := n; while p[j] > p[i] do  j :=
j - 1; j := j + 1; t := p[i]; p[i] := p[j]; p[j] := t end
end begin while (n < 10) do begin p[n] := n; n := n + 1 end;
write('Calculating all permutations for n = ', n, stdout);
while is_last = false do next end.
```

We will want to print every statement, block, procedure and variable vertically. The body of any code block should be indented with respect to the block in which it is nested. We also want the `begin` and `end` keywords to be printed in upper case so they will be easier to see in the text. Finally, we want to remove spaces where they do not contribute, specifically inside indexes and array declarations. To do this, we will use the following UGLY code:

```
#line_limit=40
#tab_width=2
```

```
; Layout program vertically
verts([<block>, <declarations>, <variables declarations>,
       <variable declaration part>, <procedure declaration>,
       <procedure declaration part>, <compound statement>,
       <block statements>, <statements>, <while statement>,
       <if statement>])

; The trailing period at the end of a program should not
;  be placed on a new line
vert_skip_last(<program>)

; Block indentations
indent(<variables declarations>)
indent(<block statements>)

; Remove spaces in expressions used as array indices since
;  they don't improve readability
under <indexed variable>{
      under <expression> {
            skip(<SPACES>)
      }
}

; Turn keywords to upper case for better
;  visibility
upcase(<COMPOUND_STATEMENTS_PREFIX>)
upcase(<COMPOUND_STATEMENTS_POSTFIX>)
upcase(<PROGRAM_PREFIX>)
upcase(<IF>)
upcase(<THEN>)
upcase(<ELSE>)
upcase(<WHILE>)
upcase(<DO>)
upcase(<VAR>)


; Prevent line_wrap from braking up arguments for write
;  statements (including their separating comma)
no_wrap(<write arg>)
```

`vert_skip_last` works just like `verts`, except that we append the last syntactic element in the list to the rest of the elements. We use `vert_skip_last` to place the trailing period on the same line as the `END` token. The operator `indent` first creates a blank *box* of width `tab_width` (2 in our case) and horizontally composes it to indent a syntactic element.

`upcase` belongs to a different class of operations. It does not affect the way *boxes* are composed, but instead, it directly modifies the way the pretty-printer prints the *box*. In this case, the operator applies a function to turn any printed character into its upper-case equivalent.

We nest an `under` expression inside another `under` expression to say that the nested condition (`under <expression>`) should only be tested when the first condition (`under <indexed variable>`) has already been met. Inside the inner `under` expression, we use the `skip` operation to remove a syntactic element from the output.

Finally, we apply `no_wrap` to arguments of `write` statements. In our example, we use this so that we do not break the long string that is sent to `write` into separate lines, even though it exceeds the line width defined with `line_limit`. Using `no_warp` is meant to preserve the semantics of the input code, as allowing a string to be line-wrapped would not be semantically equivalent to the original code. We did not apply `no_wrap` to the `string constant` non-terminal because we wanted to include the comma that follows the string as well, as most coding styles keep such delimiters on the same line as their preceding token.

Running the pretty-printer produced by this UGLY code on our example code produces:

```
PROGRAM permutations;
VAR
  p: array[1 .. 10] of integer;
  is_last: boolean;
  n: integer;
```

```
  demo: string;
  long_variable: integer
procedure next;
VAR
  i, j, k, t: integer
BEGIN
  is_last := true;
  i := n - 1;
  WHILE i > 0 DO
  BEGIN
    IF p[i] <  p[i+1] THEN
    BEGIN
      is_last := false;
      break
    END;
    i := i - 1
  END;
  IF is_last = false THEN
  BEGIN
    j := i + 1;
    k := n;
    WHILE j < k DO
    BEGIN
      t := p[j];
      p[j] := p[k];
      p[k] := t;
      j := j + 1;
      k := k - 1
    END;
    j := n;
    WHILE p[j] > p[i] DO
    j := j - 1;
    j := j + 1;
    t := p[i];
    p[i] := p[j];
    p[j] := t
  END
END
BEGIN
```

```
  WHILE (n < 10) DO
  BEGIN
    p[n] := n;
    n := n + 1
  END;
  write(
    'Calculating all permutations for n = ',
    n, stdout);
  WHILE is_last = false DO
  next
END.
```

9.3 Pretty-Printing BNF

To show a couple of other styling features, we will use our extended BNF language as an example. The BNF for our extended BNF language can be found in appendix A. We will use the following BNF for this example:

```
<digit> = '0'-'9'
<num> := ' '* <digit>+ ' '*
<CI word> -> `a`-`z`*
<word> ::= 'a'-'z'*
<S> ::= <num> | <word> | <CI word>
```

Since we use line breaks as the delimiter for productions in our BNF, we have no use for the line wrapper, so we will not be setting any line limits in this example. However, we need to define some vertical display for the code. We want to display the code using the common style of aligning all the ”expand-to“ symbols on a single column, and centering the lines around them.

Also, many symbols denote `expands-to` in various BNF syntaxes. Our BNF supports all common forms. However, we might wish to print our BNF with a single ”expands-to“ symbol used. For this we can use the following code:

```
replace(<expand-to>, '::=')    ; Use " ::= " for "expand-to"
align_center(<grammar>)        ; Align productions to center
```

`replace` replaces the *box* created by `<expand-to>` with a *box* containing whatever litter the user provided in the second argument.

`align_center` is similar to `vert` in that it operates on lists of syntactic elements, but unlike `vert`, it scans the entire list and for each element, it produces three new elements. These are: the left-hand side, the center and the right-hand side. Each element is padded and aligned as needed to produce a list of *boxes* centered around a single element.

Formatting the grammar with this UGLY code will produce:

```
  <digit> ::= '0'-'9'
    <num> ::= ' '* <digit>+ ' '*
<CI word> ::= `a`-`z`*
   <word> ::= 'a'-'z'*
      <S> ::= <num> | <word> | <CI word>
```

Lastly, we might want to create a conditional formatting of code. For instance, we might want to define rules that enforce any case-insensitive literals to be printed in upper case. We can create this formatting rule with the `upcase` operator, but since both the `<lit>` non-terminal and the `<lit-ci>` non-terminal in our extended BNF grammar use the same underline non-terminal, `<lit-char>`, to derive letters, we need to limit the effect of `upcase` to just characters derived by `<lit-ci>`. We will use the `under` UGLY directive for this like so:

```
replace(<expand-to>, '::=')
align_center(<grammar>)
under <range-ci> {
      upcase(<lit-char>)
}
```

The way `under` works is by attaching predicates to operators. While traversing the AST, UGLY will test these predicates. Only if the predicates have been satisfied UGLY will apply the `upcase` operator.

Running this pretty-printer on the above BNF produces:

```
  <digit> ::= '0'-'9'
```

```
    <num> ::= ' '* <digit>+ ' '*
<CI word> ::= `A`-`Z`*
   <word> ::= 'a'-'z'*
      <S> ::= <num> | <word> | <CI word>
```

10 Conclusion

Our goal was to show that using a compiler as a pretty-printer would allow us to create system that can generate pretty-printers with minimal code written by the user, and for the resulting pretty-printers to have greater capabilities than those found in traditional pretty-printers. We compare the usability of our pretty-printing system with a few existing methods for creating pretty-printers in the following table:

Name	Formatting specification language	User Input	Language Family	Space Complexity	Time Complexity
UGLY	UGLY	Language syntax	LL(K)[5]	Linear	Exponential[5]
Invertible Language Descriptors	Invertible function combinators	Language parser	LL(K)	Linear	Exponential
Oppen's Pretty-Printer	N/A[6]	Language scanner	CFG	Linear in the maximal line width	Linear
CodeBuff	N/A	Language scanner, parser and training corpus	CFG	$O(N)$ for a corpus of size N	Training: $O(NlogN)$ for a corpus of size N Formatting: $O(N^2logN)$ for a document of size N

As shown in the table, our pretty-printers system requires the user to provide only the desired formatting and the syntax of the language, while other techniques require significantly more code to produce a pretty-printer. In contrast, our system requires UGLY code to define the desired formatting in our system, while in Oppen's pretty-printer and `CodeBuff` no formatting specification is required.

The complexity of our generated pretty-printer is not optimal, and our supported family of languages is not as large as that offered by Oppen's pretty-printer. These limitations come from our choice of parser-combinators, which

may backtrack, as our parser construction method. We decided to use this method since we did not want to focus on the complexity of pretty-printing, which has a lower bound similar to parsing. To improve the efficiency of the system, a different front-end may be incorporated at the cost of ease of implementation.

We compare the capabilities of different pretty-printing methods in the following table:

Name	width Control line	User definable format	String transformations	Sound output
UGLY	Possible	Yes	Yes	No
Invertible Language Descriptors	Possible	No	partial	Yes
Oppen's Pretty-Printer	Yes	No	No	Yes
CodeBuff	No	Partially[6]	No	Yes

The table above shows how a pretty-printer created using UGLY carries more capabilities than most common pretty-printers. The user has complete control over the formatting of the source code, and with the correct UGLY code, the user can control the line width as well as with any other pretty-printer. UGLY is also the only system that offers complete control over the strings of the pretty-printed code, allowing changing case, enforcing syntax choices (e.g., the type of parentheses to use in an *S-expression* or the type of string quotes to using in Python) and even adding text.

The drawback of this level of control in UGLY is that the output of the pretty-printer is not guaranteed to be syntactically correct. Since the user can change any token or even character in the input stream, incorrect UGLY code will

likely format code into a syntactically incorrect output. This issue is unavoidable, and so no effort was made to try and resolve it. To mitigate any errors being introduced by the pretty-printer, it is possible to run the pretty-printed output back through the parser of the pretty-printer, and this will verify that the pretty-printer introduced no syntax errors during formatting.

We say Invertible Language Descriptors can produce pretty-printers that can partially manipulate strings because when writing the parsers, we have complete control over their inverse functions. This does not allow for user-defined string manipulations, but it does allow, for instance, embedding the equivalent of the `upcase` or `downcase` operators in UGLY to language keywords while implementing the parsers for those keywords.

`CodeBuff`, being a learning pretty-printer, allows minimal user control, which might make it seem a poor choice. However, this is because the corpus defines the format, and so the user still has a pretty-printer that enforces a predetermined format.

11 Future work

11.1 Improving Generic ASTs

We designed our generic ASTs to hold as much information as possible while being language agnostic. Using a further extended BNF, we could improve on this to allow smarter formatting. If we allowed the user to group productions on the RHS of a production-rule, we could more easily and robustly define certain UGLY operations (e.g., `center`).

Also, for many BNFs and inputs that we tested, many tree nodes were redundant (e.g., Empty nodes in a list), and some nodes are needlessly broken into multiple nodes. This happens because of how grammars are usually written in BNF to allow for better readability. A post-process to analyze the tree and remove redundant nodes would significantly reduce the size of the AST and thus improve the runtime and space requirements of the system. When

pretty-printing large files (>1KB), these inefficiencies become significant.

11.2 Extending the *box* language

The language we used to construct and compose *boxes* is powerful enough to construct most coding styles. However, some coding styles require more control over composition. For instance, reducing the indentation level of a line of code below the indentation of its containing *box* (i.e., code block) is required to produce styles such as:

```
(define function_name (lambda (x)
            (+ x 1)))
```

In the above code, the lambda's body is indented relative to the defined symbol, and not the lambda expression. While uncommon, this coding style is used sometimes and would be hard to define using our current *box* language.

To enable such coding styles, *box* composition should be defined using orientation (i.e., horizontal or vertical) as is done currently and an offset instead of relative positions (i.e., left, right or center and top, bottom or middle). The offset would allow a continuous line on which we can position *boxes* relative to one another. Implementing negative indentation would be done by composing the *box* holding the nested code horizontally with the *box* containing the nesting code. Negative indentation will be defined by settings the offset of the composition to negative `tab_width`.

11.3 Selectors

Currently, the user may only define a single formatting for a language, and our system will apply it to any input. However, there is usually no single format specification which is optimal for any input. To allow more than one formatting to be defined, the user should be allowed to define sets of formatting specifications with UGLY. The system should assign a utility value for each possible formatting of the input and choose the nicest one.

Since the notion of 'nicest' formatting is obviously subjective, a learning component can also be added to the system which will assign these values. Some work in this field has already been done by Terence Parr and Jurgen Vinju [17].

11.4 UGLY Variables

Currently, UGLY has the computational power of a stack-automata. To improve this, and make it equivalent to a stack-automata, we will need to add variables to the specification. Variables that hold concrete syntax will allow the system and the user to parameterize and condition operations based on the inputted code. A *Derived-From* construct can be added to the language that allows the user to define a variable that holds the text derived by a production-rule. The user can use variables as conditions for executing UGLY code or as a parameter for operators. Using this type of variables will make it possible to write UGLY specification to transform data from one representation to another (e.g., JSON to XML), generating HTML pages from parts of the code or embedding the input code in LaTeX documents.

A second benefit of introducing variables to UGLY is related to ease of use. When a user of UGLY wants to apply an operation on multiple production-rules in the BNF, they have to write UGLY code applying their transformation to each of these rules. Many times, these production-rules will all be the RHS of a single production-rule (e.g., applying an operation to all the elements in a catenation). To support this, a *Right-Hand-Of* constructor could be introduced that will define a variable that holds a list of all the elements of the right hand of a production-rule. We can use these variables in operators that will apply the operator to all elements in the list. It should be considered whether adding disjunctions to the generic ASTs would be beneficial in this case.

12 Summary

We believe that considering pretty-printing as a form of compilation is a step towards improving the way people write and read code. Using using compilers as pretty-printers allows users more complete control over the representation of their code. A code maintainer can define and enforce syntactic conventions on the code base allowing many people to collaborate while keeping the code format consistent. Using UGLY, the process of defining the coding conventions can be done incrementally and without having to delve into the workings of the pretty-printer which formats it.

Using the intermediate representation of the *box* language we have also been able to apply complex typesetting (e.g., `center`) to input code with minimal effort, while using existing methods this would require a pretty-printer to have a great deal of knowledge about the syntax of the target language. Using our system on several languages and inputs shows that it usually takes less than 15 UGLY statements to produce a pretty-printer that formats a code into a commonly used coding style without any prior knowledge about the programming language.

Our original goal was to show that treating pretty-printing as compilation opens up better ways to use pretty-printers as tools for processing text written in a formal language. Using our UGLY system, we have shown that with this approach, we can create a pretty-printer generator relatively easily that produces pretty-printers with more capabilities than any existing pretty-printer today. There are still several notions to explore with this idea that seems promising we look forward to seeing how they will improve pretty-printing.

13 References

References

[1] Daniel G Bobrow, D. Lucille Darley, Daniel L. Murphy, Cynthia Solomon, and Warren Teitelman. The bbn-lisp system. Technical Report AFCRL-66-180, Air Force Cambridge Research Laboratories, 1966.

[2] The stanford ai lisp 1.6 system manual. Technical Report AFCRL-66-180, Stanford AI Laboratoriess, 1969.

[3] Ira Goldstein. Pretty-printing, converting list to linear structure. Technical Report Lab memo No. 279, MIT A. I. Lab, 1973.

[4] Dereck C. Oppen. Prettyprinting, October 1980.

[5] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, pages 53–96. Springer Verlag, 1995.

[6] S. Doaitse Swierstra. Linear, online, functional pretty printing, 2004.

[7] Philip Wadler. A prettier printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.

[8] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing, 2011.

[9] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[10] Stephen C. Johnson and Ravi Sethi. Unix vol. ii, 1990.

[11] Alon Lavie and Masaru Tomita. Glr* – an efficient noise-skipping parsing algorithm for context free grammars, 1993.

[12] Terence J Parr, T. J. Parr, and R W Quong. Antlr: A predicated-ll(k) parser generator, 1995.

[13] W.H. Burge. *Recursive programming techniques*. Addison-Wesley Series in Electrical and Computer Engineering. Addison-Wesley Longman, Incorporated, 1975.

[14] Olaf Chitil. Pretty printing with lazy dequeues. *ACM Trans. Program. Lang. Syst.*, 27:163–184, 2005.

[15] S. doaitse Swierstra and Olaf Chitil. Linear, bounded, functional pretty-printing. *J. Funct. Program.*, 19(1):1–16, January 2009.

[16] Richard Bonichon and Pierre Weis. Format Unraveled. In *28ièmes Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

[17] Terence Parr and Jurgen Vinju. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 137–151, New York, NY, USA, 2016. ACM.

[18] Donald E. Knuth. *The TeXbook*. Addison-Wesley Professional, 1986.

[19] Daniel D. McCracken and Edwin D. Reilly. Backus-naur form (bnf), 2003.

[20] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages*, PADL'08, pages 167–181, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] ECMA-404. The json data interchange syntax, December 2017.

[22] Jim Welsh and R. M. McKeag. *Structured System Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1980.

14 Appendix A - BNF for BNF Syntax

The syntax for BNF described below is an extension on the commonly used BNF syntax with references from The Language of Languages by Matthew Might

---

```
;; Token production-rules
<comment> ::= (';' (<any-char> \ <newline>)* <newline>)
<spaces> ::= (<whitespace>*)
<alpha> ::= 'a'-'z' | 'A'-'Z'
<alphanum> ::= <alpha> | '0'-'9'
<expand-to> ::= <spaces> ['::=' | ':=' | '=' | '->'] <spaces>
<disj-op> ::= <spaces> '|' <spaces>
<caten-op> ::= ' '*
<sub-op> ::= <spaces> '\\' <spaces>
<kstar> ::= '*' <spaces>
<kplus> ::= '+' <spaces>
<maybe> ::= '?' <spaces>
<skip> ::= '#' <spaces>
<trace> ::= '$' <spaces>
<cb-lparen> ::= <spaces> '{' <spaces>
<cb-rparen> ::= <spaces> '}' <spaces>
<nest-lbracket> ::= <spaces> '[' <spaces>
<nest-rbracket> ::= <spaces> ']' <spaces>
<nest-lparen> ::= <spaces> '(' <spaces>
<nest-rparen> ::= <spaces> ')' <spaces>
<nt-langle> ::= <spaces> '<'
<nt-rangle> ::= '>' <spaces>
<lit-lquote> ::= <spaces> '\''
<lit-rquote> ::= '\'' <spaces>
<lit-ci-lquote> ::= <spaces> '\`'
<lit-ci-rquote> ::= '\`' <spaces>
<escaped> ::= '\`' | '\'' | '\\'
<range-dash> ::= '-'
<production-term> ::= <newline>

;; Expression elements
<ident> ::= <alpha> [<alphanum> | '_' | '-' | ' ']*
```

```
<escape> ::= '\\' <escaped>
<lit-char> = [<any-char> \ <escaped> | <escape>]
<range-char> ::= <lit-lquote> <lit-char> <lit-rquote>
<range> ::= <range-char> <range-dash> <range-char>
<epsilon> ::= <nt-langle> 'epsilon' <nt-rangle>
<lit-ci> ::= <lit-ci-lquote> <lit-char>+ <lit-ci-rquote>
<lit> ::= <lit-lquote> <lit-char>+ <lit-rquote>
<literal> ::= <lit> | <lit-ci>
<term> ::= <range> | <literal>
<non-term> ::= <nt-langle> <ident> <nt-rangle>
<nested-expr> ::= <nest-lparen> <expr> <nest-rparen> |
                  <nest-lbracket> <expr> <nest-rbracket>
<operand> ::= [<non-term> | <term> | <nested-expr>] <mod>?

;; Expression operations
<mod> ::= <kstar> | <kplus> | <maybe> | <skip> | <trace>
<caten> ::= <operand> (<caten-op> <operand>)*
<disj> ::= <caten> (<disj-op> <caten>)*
<diff> ::= <disj> (<sub-op> <disj>)?
<callback> ::= <cb-lparen> <ident> <cb-rparen>
<expr> ::=  <diff> (<callback>?)

;; Productions
<expr> ::=  <diff>
<lhs> ::= <non-term> <trace>?
<rhs> ::= <expr>
<production> ::= <lhs> <expand-to> <rhs> (<production-term>*)

;; Initial productions ("S")
<grammar> ::= (<comment> | <production>)+
```

Extended BNF syntax

15 Appendix B - BNF for UGLY Syntax

```
<COMMENT> := ';' <any>* <newline>
<SPACE> := ' ' | <tab> | <newline> | <COMMENT>
```

```
<UNDER> := <SPACE>* `under` <SPACE>*
<OP_L_BRACKET> := <SPACE>* '(' <SPACE>*
<OP_R_BRACKET> := <SPACE>* ')' <SPACE>*
<NT_L_BRACKET> := <SPACE>* '<' <SPACE>*
<NT_R_BRACKET> := <SPACE>* '>' <SPACE>*
<LIST_L_BRACKET> := <SPACE>* '[' <SPACE>*
<LIST_R_BRACKET> := <SPACE>* ']' <SPACE>*
<BLOCK_L_BRACKET> ::= <SPACE>* '}' <SPACE>*
<BLOCK_R_BRACKET> ::= <SPACE>* '{' <SPACE>*
<LIT_L_QUOTE> := <SPACE>* '\''
<LIT_R_QUOTE> := '\'' <SPACE>*
<LIT_CI_L_QUOTE> := <SPACE>* '\`'
<LIT_CI_R_QUOTE> := '\`' <SPACE>*
<DIGIT> := ('0'-'9')
<ALPHA> := (`a`-`z`)
<SEP> := <SPACE>* ',' <SPACE>*
<ARG_SEP> := <SPACE>* ',' <SPACE>*
<ALPHANUM> ::= (<ALPHA> | <DIGIT>)

<id> ::= <SPACE>* (<ALPHANUM> | '_')+ <SPACE>*
<num> ::= <SPACE>* <DIGIT>+ <SPACE>*
<nt name> ::= (<ALPHANUM> | ' ' | '_' | '-')+
<nt> ::= <NT_L_BRACKET> <nt name> <NT_R_BRACKET>
<lit char> ::=  <any> \ <LIT_R_QUOTE>
<lit> ::=  <LITCI_L_QUOTE> <lit char>* <LIT_R_QUOTE>
<litci char> ::= <any> \ <LIT_CI_R_QUOTE>
<lit ci> ::= <LIT_CI_L_QUOTE> <litci char>* <LIT_CI_R_QUOTE>
<literal> ::= <lit> | <lit ci>
<simple value> ::= <nt> | <literal>
<list elements> := <element> (<SEP> <simple value>)+ | <element>
<list> ::= <LIST_L_BRACKET> <list elements> <LIST_R_BRACKET>
<value> ::= <list> | <simple value>
<arg> ::= <value>
<args> ::= <arg> (<ARG_SEP> <arg>)* | <epsilon>
<operation> ::= <id> <OP_L_BRACKET> <args> <OP_R_BRACKET>
<under> := <NOT>? <UNDER> <value> <block>
<block> := <BLOCK_L_BRACKET> <statement>* <BLOCK_R_BRACKET>
<statement> := <operation> | <let> | <under> | <if>
<ugly> :=   (<statement>)+
```

BNF grammar

16 Appendix C - BNF for JSON Syntax

The syntax described below is based on the ECMA standard for JSON [21]

```
<S> := <json>
<json> := <value>

<value> := <object> | <array> | <string> | <number> | <bool> |
 ↪  <NULL>

<bool> := <TRUE> | <FALSE>

<object> := <OBJ_OPEN> <object members> <OBJ_CLOSE>
<object members> := <members> <member>
<members> := (<member> <MEMBER_SEP>)*
<member> := <member name> <MEMBER_COLON> <value>
<member name> := <string>

<array> := <ARRAY_OPEN> <array elements> <ARRAY_CLOSE>
<array elements> := <elements> <element>
<element> := <value>
<elements> := (<value> <ARRAY_SEP>)*

<string> := <STRING_OPEN> <string chars> <STRING_CLOSE>
<string chars> := <string char>*
<string char> := <any-char> \ (<ESCAPE> | <STRING_CLOSE>) |
 ↪  <escaped>
<escaped> := <ESCAPE> (<ESCAPED_CHARS> | <hex>)
<hex> := <HEX_PREFIX> <HEX_DIGIT> <HEX_DIGIT> <HEX_DIGIT>

<number> := <integer> <fraction>? <exponent>?
<integer> := <NEG>? <DIGIT>+
<fraction> := <FRACT_PREFIX> <DIGIT>+
<exponent> := <EXP_PREFIX> <SIGN> <DIGIT>+

; Note that the original JSON syntax does not support comments
<comment> := '#' (<any-char> \ <newline>)* <newline>
<SPACES> := (' ' | <tab> | <newline> | <comment>)*
```

```
<OBJ_OPEN> := <SPACES> '{' <SPACES>
<MEMBER_SEP> := <SPACES> ',' <SPACES>
<MEMBER_COLON> := <SPACES> ':' <SPACES>
<OBJ_CLOSE> := <SPACES> '}' <SPACES>

<ARRAY_OPEN> := <SPACES> '[' <SPACES>
<ARRAY_SEP> := <SPACES> ',' <SPACES>
<ARRAY_CLOSE> := <SPACES> ']' <SPACES>

<STRING_OPEN> := <SPACES> '"'
<ESCAPE> := '\\'
<ESCAPED_CHARS> := '"' | '\\' | 'b' | 'f' | 'n' | 'r' | 't'
<HEX_DIGIT> := '0'-'9' | 'a'-'f' | 'A'-'F'
<STRING_CLOSE> := '"' <SPACES>

<FRACT_PREFIX> := '.'
<EXP_PREFIX> := 'e'

<NEG> := '-'
<DIGIT> := '0'-'9'
<SIGN> := '+' | '-'

<TRUE> := <SPACES> 'true' <SPACES>
<FALSE> := <SPACES> 'false' <SPACES>
<NULL> := <SPACES> 'null' <SPACES>
```

JSON grammar

17 Appendix D - BNF for Pascal Syntax

The syntax described below is based on the mini-pascal syntax described in Structured System Programming [22]

```
<S> ::= <program>
<program> ::= <program header> <block> <program footer>
<program header> ::= <PROGRAM_PREFIX> <identifier> <SEMICOLON>
<program footer> ::= <PERIOD>
```

```
<block> ::= <variable declaration part> <procedure declaration part> <statement part>

<declarations> =  (<variable declaration> <SEMICOLON>)*
<variables declarations> =  <declarations> <variable declaration>
<variable declaration part> ::= <VAR> <variables declarations>| <empty>

<variable declaration> ::= <identifier> ((<COMMA> <identifier>)*) <COLON> <type>

<type> ::= <array type> | <simple type>

<array type> ::= <ARRAY_PREFIX> <L_BRACKET> <index range> <R_BRACKET> <ARRAY_TYPE_PREFIX> <simple type>

<index range> ::= <integer constant> <RANGE_ELLIPSIS> <integer constant>

<simple type> ::= <identifier>

<procedures declarations> ::= (<procedure declaration> <SEMICOLON>)*
<procedure declaration part> ::= <procedures declarations> <procedure declaration> | <epsilon>

<procedure header> ::= <PROCEDURE_PREFIX> <identifier> <SEMICOLON>

<procedure declaration> ::= <procedure header> <block>

<statement part> ::= <compound statement>

<statements> ::= (<statement> <SEMICOLON>)*
<block statements> ::=  <statements> <statement>

<compound statement> ::= ((<COMPOUND_STATEMENTS_PREFIX>) <block statements>  <COMPOUND_STATEMENTS_POSTFIX>)

<statement> ::= <structured statement> | <simple statement>
```

```
36
37 <simple statement> ::= <assignment statement> | <read statement>
    ↪  | <write statement> | <procedure statement>
38
39 <assignment statement> ::= <variable> <ASSIGN_OP> <expression>
40
41 <procedure statement> ::= <identifier>
42
43 <read statement> ::= <READ_PREFIX> <L_PAREN> <input variable>
    ↪  (<COMMA> <input variable>)* <R_PAREN>
44
45 <input variable> ::= <variable>
46 <write arg> ::= <output value> <COMMA>
47 <write statement> ::= <WRITE_PREFIX> <L_PAREN> <write arg>*
    ↪  <output value> <R_PAREN>
48
49 <output value> ::= <expression>
50
51 <structured statement> ::= <compound statement> | <if statement>
    ↪  | <while statement>
52
53 <if header> ::=  <IF> <expression> <THEN>
54 <if statement> ::= <if header> <statement> <else> | <if header>
    ↪  <statement>
55 <else> ::= <ELSE> <statement>
56
57 <while header> ::= <WHILE> <expression> <DO>
58 <while statement> ::= <while header> <statement>
59
60 <expression> ::= <simple expression> <relational operator>
    ↪  <simple expression> |  <simple expression>
61
62 <simple expression> ::= <sign>? <term> (<adding operator>
    ↪  <term>)*
63
64 <term> ::= <factor> (<multiplying operator> <factor>)*
65
66 <factor> ::= <variable> | <constant> | <L_PAREN> <expression>
    ↪  <R_PAREN> | <NOT> <factor>
67
```

```
<relational operator> ::= <EQ_OP> | <NEQ_OP> | <LT_OP> |
    ↪ <LTE_OP> | <GTE_OP> | <GT_OP>

<sign> ::= <POS_SIGN> | <NEG_SIGN>

<adding operator> ::= <ADD_OP> | <SUB_OP> | <OR_OP>

<multiplying operator> ::= <MUL_OP> | <DIV_OP> | <AND_OP>

<variable> ::=  <indexed variable> | <entire variable>

<indexed variable> ::= <array variable> <L_BRACKET> <expression>
    ↪ <R_BRACKET>

<array variable> ::= <entire variable>

<entire variable> ::= <identifier>

<empty> = <epsilon>

<constant> ::= <integer constant> | <string constant> |
    ↪ <identifier>

<integer constant> ::= <SPACES> <DIGIT>+ <SPACES>

<identifier> ::= <SPACES> <ident first char> <ident char>*
    ↪ <SPACES>
<ident first char> ::= <ALPHA> | '_'
<ident char> ::= <alphanumeric> | '_'

<alphanumeric> ::= <ALPHA> | <DIGIT>

<string constant> ::= '\'' (<any-char> \ '\'')* '\''

<line comment> ::= <COMMENT_PREFIX> (<any-char> \ <newline>)*
    ↪ <newline>
<multiline comment> ::= <MULTILINE_COMMENT_PREFIX> (<any-char> \
    ↪ <MULTILINE_COMMENT_POSTFIX>)* <MULTILINE_COMMENT_POSTFIX>

<comment> ::= <line comment> | <multiline comment>
```

```
102
103 <SPACE> ::= (' ' | <tab> | <newline> | <comment>)
104 <SPACES> ::= <SPACE>*
105 <PROGRAM_PREFIX> ::= <SPACES> `program` <SPACES>
106 <SEMICOLON> ::= <SPACES> ';' <SPACES>
107 <PERIOD> ::= <SPACES> '.' <SPACES>
108 <VAR> ::= <SPACES> `var` <SPACES>
109 <COMMA> ::= <SPACES> ',' <SPACES>
110 <COLON> ::= <SPACES> ':' <SPACES>
111 <L_BRACKET> ::= <SPACES> '[' <SPACES>
112 <R_BRACKET> ::= <SPACES> ']' <SPACES>
113 <L_PAREN> ::= <SPACES> '(' <SPACES>
114 <R_PAREN> ::= <SPACES> ')' <SPACES>
115 <ARRAY_PREFIX> ::= <SPACES> `array` <SPACES>
116 <ARRAY_TYPE_PREFIX> ::= <SPACES> `of` <SPACES>
117 <RANGE_ELLIPSIS> ::= <SPACES> '..' <SPACES>
118 <PROCEDURE_PREFIX> ::= <SPACES> `procedure` <SPACES>
119 <COMPOUND_STATEMENTS_PREFIX> ::= <SPACES> `begin` <SPACES>
120 <COMPOUND_STATEMENTS_POSTFIX> ::= <SPACES> `end` <SPACES>
121 <READ_PREFIX> ::= <SPACES> `read` <SPACES>
122 <WRITE_PREFIX> ::= <SPACES> `write` <SPACES>
123 <ASSIGN_OP> ::= <SPACES> ':=' <SPACES>
124 <DIGIT> ::= '0'-'9'
125 <ALPHA> ::= 'a'-'z' | 'A'-'Z'
126 <IF> ::= <SPACES> `if` <SPACES>
127 <THEN> ::= <SPACES> `then` <SPACES>
128 <ELSE> ::= <SPACES> `else` <SPACES>
129 <WHILE> ::= <SPACES> `while` <SPACES>
130 <DO> ::= <SPACES> `do` <SPACES>
131 <NOT> ::= <SPACES> `not` <SPACES>
132 <EQ_OP> ::= <SPACES> '=' <SPACES>
133 <LT_OP> ::= <SPACES> '<' <SPACES>
134 <GT_OP> ::= <SPACES> '>' <SPACES>
135 <LTE_OP> ::= <SPACES> '<=' <SPACES>
136 <GTE_OP> ::= <SPACES> '>=' <SPACES>
137 <NEQ_OP> ::= <SPACES> '<>' <SPACES>
138 <POS_SIGN> ::= <SPACES> '+'
139 <NEG_SIGN> ::= <SPACES> '-'
140 <ADD_OP> ::= <SPACES> '+' <SPACES>
141 <SUB_OP> ::= <SPACES> '-' <SPACES>
```

```
142 <OR_OP> ::= <SPACES> `or` <SPACES>
143 <MUL_OP> ::= <SPACES> '*' <SPACES>
144 <DIV_OP> ::= <SPACES> '/' <SPACES>
145 <AND_OP> ::= <SPACES> `and` <SPACES>
146 <COMMENT_PREFIX> ::= '//'
147 <MULTILINE_COMMENT_PREFIX> ::= '{'
148 <MULTILINE_COMMENT_POSTFIX> ::= '}'
```

Pascal grammar