On the relationship between parsing and pretty-printing

Clovis Eberhart

August 27, 2012

Contents

Introduction

Parsing and pretty-printing are both very important in computer science. Parsing maps strings to structured internal pieces of data that corresponds to them, while pretty-printing maps these structured internal pieces of data to strings that correspond to them.

Because communication in a computer can only be achieved by use of (bit)string, parsing and printing are ubiquitous in computer science. Every time some data is printed onto a screen for someone to read, it has to be printed in a human-readable way. Every time a human inserts data into a computer, this human-readable data is parsed into easily usable data. Every time some data is transferred through any protocol, this data must first be printed by the sender into a (bit)string and sent to the receiver, who has to parse this string to recover the data.

Moreover, parsers and pretty-printers are also important in other fields that use them, for example in linguistics to find the structure of and produce natural language sentences or in biology to analyse DNA sequences.

On the one hand, it is quite obvious that parsers and pretty-printers are somehow related to each other. On the other hand, they are typically implemented separately, which leads to redundancy, which in turn may lead to inconsistency in protocols that use those parsers and pretty-printers, and even security holes in these protocols.

Recently, some frameworks have been suggested to unify parsing and pretty-printing, for example [9]. They aim at implementing parsing and pretty-printing together to reduce the workload and avoid inconsistencies between parsers and pretty-printers. However, as far as we know, no formal definition of what consistency between a parser and a pretty-printer is has ever been given. Therefore, if implementing parsing and pretty-printing together does reduce the programmer's workload, it does not ensure that the parser and pretty-printer will be consistent.

The problem I tried to solve during my internship was to find a definition of what consistency between a parser and a pretty-printer is. Even though this problem may sound easy at first, the fact that there is no answer to it yet even though parsers and pretty-printers are crucial in computer science hints that this problem is actually quite complicated.

We will first see a brief state of art of parsing and pretty-printing. Then, I will show how I found a definition for consistency between parsing and pretty-printing, give this definition, some properties that can be ensured for consistent parsers and pretty-printers, and work on an example. I'll finish by showing the limits of this definition and studying the case of functional parsing and pretty-printing, which could prove to be very interesting as it could lead to a more practical definition than the general one.

I will warn readers beforehand that, since what I did was search a definition, there is no definite answer here, as there is no way to prove that my definition is the 'right' one. All I can do is give some ideas of why I think it is a good definition, give properties that result from this definition and show that it works on some non-trivial examples.

Chapter 1

State of the Art

1.1 Parsing

The notion of parsing is linked to that of grammar in general, and most of the time to that of context-free grammar (when I will write grammar from now on, it will mean context-free grammar unless specified otherwise). We suppose that the reader is familiar with them, as well as the notions of production of a grammar and derivation tree associated to a production.

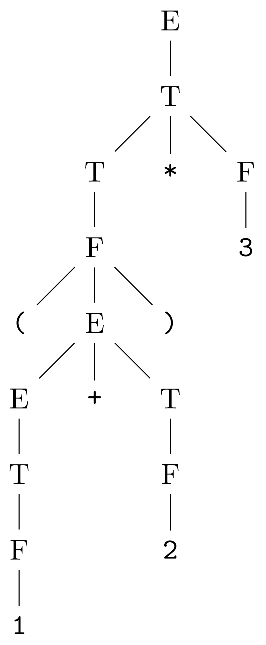
What we call parsing here is the action of mapping a string to a structured internal piece of data. However, most of the time, when someone refers to parsing in computer science, they are thinking of an algorithm that takes a string as its input and returns a (or all the) derivation tree(s) that produce this string according to a given grammar. I will call this *parsing according to a grammar*.

This has been thoroughly studied from an algorithmic point of view, and there are a lot of algorithms to parse according to a grammar. One of the, if not the, most well-known of these algorithms is the Earley algorithm [3]. Dick Grune and Ceriel J.H. Jacobs have published a book about parsing in which many algorithms and techniques to parse according to a grammar can be found [4]. There are some very advanced features in these parsers to improve their complexity and parse according to larger classes of context-free grammars, but, in our case, we're not interested in which particular algorithm to use as long as it returns all the derivation trees associated to the string, so we won't consider a particular algorithm and just treat parsing according to a grammar as a function that returns all the derivation trees associated to an input string.

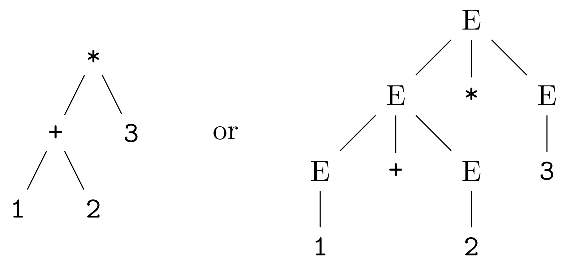
Let us introduce a 'toy' grammar for the sake of example. This grammar is theoretically used to parse arithmetic expressions:

$\mathcal{G}_{toy} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, E)$ where the set of non-terminals is $\mathcal{N} = \{E, T, F\}$, the set of terminals is $\mathcal{T} = \mathbb{N} \cup \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{(}, \texttt{)}\}$ (even if this not authorized in the definition of a grammar because T is infinite, this could be simulated, but would just make the example much more complicated without adding anything interesting), the set of derivation rules is $\mathcal{R} = \{E \rightarrow E\texttt{+}T \mid E\texttt{-}T \mid T,\ T \rightarrow T\texttt{*}F \mid T\texttt{/}F \mid F,\ F \rightarrow \texttt{(}E\texttt{)} \mid \texttt{i} \text{ for } i \in \mathbb{N}\}$ and the start symbol is E.

Setting aside the particular algorithm used, parsing `"(1+2)*3"` according to $\mathcal{G}_{toy}$ gives (the derivation tree has been rotated for space efficiency matters):



However, most of the time, parsing according to a grammar is not what *real* parsers do. When a string is parsed, the result is not the derivation tree in a given grammar. For example, in the example above, the parentheses hold no information, as the expression is already organized as a tree. Moreover, the imbrication of Es, Ts and Fs does not give any information either. The result that one could expect from a parser parsing arithmetic expressions would be something along the lines of:



It is crucial to note that these trees, if they are seen as derivation trees, are derivations of `"123"` and `"1+2*3"` respectively, and not `"(1+2)*3"`. Therefore, parsing cannot be defined as a function that maps strings to their derivation trees in a certain grammar, but has to be something more general.

A functional way of seeing a parser is to say that parsers map strings to some internal data type α. But, to use the full power of functional programming, an interesting approach is to combine simple parsers to create more complex ones by using parser combinators [6], which are higher-order functions that accept parsers as their input and produce a new parser. One of these combinators is the sequential combinator `<*>`. When it is applied to two parsers p_1 and p_2, the result p_1`<*>`p_2 uses p_1 to parse the start of the string, and p_2 to parse the rest of the string (the result being the result of p_1 (that has to be a function) applied to the result of p_2). This indicates that a natural way to describe a functional parser is to say that not only it returns an element with type α, but it also returns a string that is 'the part of the string that hasn't been parsed'. Moreover, such a parser is typically non-deterministic, so a parser has type $P(\alpha) = \alpha \to List(\alpha \times String)$.

For example, if p is a parser that parses decimal integers and maps them to integers, it would have type $P(int) = \alpha \to List(int \times String)$ and the result of the parser on `"0123"` would be:

$$p(\texttt{"0123"}) = [(0, \texttt{"123"}); (1, \texttt{"23"}); (12, \texttt{"3"}); (123, \varepsilon)]$$

Wadler showed very early that parsers form a monad [11], a structure stemming from category theory that was proved useful in handling a number of computational problems (e.g. adding side effects to purely functional programming languages [7]).

Definition (monad)**.** *A monad $(T, return, bind)$ is a triple made of a type constructor T that maps every type α to $T\alpha$ and two functions: return with type $\alpha \to T\alpha$ and bind with type $T\alpha \to (\alpha \to T\beta) \to T\beta$. They must fulfill the following properties:*

- $\forall x.\ bind\ x\ return = x$
- $\forall x, f.\ bind\ (return\ x)\ f = f\ x$
- $\forall x, f, g.\ bind\ (bind\ x\ f)\ h = bind\ x\ (\lambda x. bind\ (f\ x)\ g)$

For parsers, the type constructor is $P(\alpha) = \alpha \to List(\alpha \times String)$, $return\ x$ is the parser that doesn't consume any input and returns x. $bind\ p\ f$ is a little bit more complex: the parser $bind\ p\ f$ first applies p which returns a list of (value, string) pairs, applies f to each value, resulting in a parser, which in turn is used to parse the corresponding strings, then, all the resulting lists are flattened into a single list (even though this may sound twisted, this is only a way to represent sequential composition).

In the end, even though monads are a framework that is adapted to the study of parsers, I still haven't given the definition of parsing. That is because I found no formal definition of parsing other than that of parsing according to a grammar. Even in the case of parsing according to a grammar, when one tries to define parsing for more classes of grammars than context-free grammars, the definition is not formal any longer. The best definition I found for parsing according to a grammar is the following [4]:

Parsing is the process of structuring a linear representation in accordance with a given grammar.

1.2 Pretty-Printing

Pretty-printing is the action of associating to some internal structured data a 'pretty' string that represents it. Compared to the literature and research on parsing, the literature and research on pretty-printing is slim. There is no solid theory behind pretty-printing. Contrary to parsing, for which parsing according to a grammar is non-trivial, (pretty-)printing a given tree where the terminal symbols are exactly the leaves of the tree is very easy. Therefore, there is no such thing as *pretty-printing according to a grammar.*

Even though there are a libraries in a lot of languages to pretty-print (*Haskell*, *OCaml*, *Coq*...), the techniques used differ in all those libraries and they're all ad hoc (for example, the *OCaml* library uses what they call *boxes*, while Hughes's library (in *Haskell* [5]) uses horizontal and vertical composition of documents).

Pretty-printing can also be seen from a functional point of view (for example, it is seen from a functional point of view in Hughes's library). In that case, since a pretty-printer maps internal structured data to a ‘pretty’ string, it can be seen as a function with type $P(\alpha) = \alpha \to String$. However, as far as I know, there is no counterpart of the classic parser-combinator framework for pretty-printers.

However, the question of knowing what pretty-printing is is pretty tricky, since the notion of beauty is subjective. For some people, writing a moderately complex function on a single line is what they prefer, while others like skipping lines and putting lots of indentations. This suggests that there is something that looks somewhat arbitrary in pretty-printers.

1.3 Unification of Parsing and Pretty-Printing

Some frameworks have been suggested to unify parsing and pretty-printing. One of them, called *invertible syntax descriptions* has been suggested by Rendel and Ostermann [9]. It aims at producing parsers and pretty-printers using two instances of the same description. This helps reduce the programmer's workload when he has to program a parser and a pretty-printer, and it also aims at avoiding inconsistency between the parser and the pretty-printer (however, since there is no definition of what inconsistency is in their paper, they do not prove anything). They recursively build parsers and pretty-printers using some combinators inspired by the classic parser combinators.

The combinator `<$>` has type $(\alpha \to \beta) \to P(\alpha) \to P(\beta)$ (where P can be instantiated as a parser or a pretty-printer). It is inspired from a classic parser combinator, f`<$>`p parses the string using p and then applies f to the result. However, for pretty-printers, f`<$>`p may print according to p, but there is nothing to apply f to. What we actually want to do is something that would be more or less the inverse of parsing, that is use f^{-1} on the input, then print using p. This is why we need some sort of inverse to this function.

Let us note $Just(A)$ the set $A \cup \{*\}$ where $*$ is not in A.

Definition (partial isomorphism)**.** A partial isomorphism *between* A *and* B *a function* $f : A \to Just(B)$ *such that there exists a function noted* $f^{-1} : B \to Just(A)$ *that satisfies the property:*
$\forall a \in A, \forall b \in B.\ f(a) = b \Leftrightarrow a = f^{-1}(b)$.

The isomorphisms are partial to allow parsers and pretty-printers to fail.

Now, if we allow `<$>` to be used only with partial isomorphisms, we can instantiate f`<$>`p for both parsers and pretty-printers. Therefore, it is necessary to change the type of `<$>` to $Iso(\alpha, \beta) \to P(\alpha) \to P(\beta)$, where $Iso(\alpha, \beta)$ is the type of partial isomorphisms between α and β. The instantiation is the same as above for parsers, and for pretty-printers, we use f^{-1} on the input and then print with p.

The combinator `<*>` has type $P(\alpha) \to P(\beta) \to P(\alpha \times \beta)$. It is inspired from the classic sequential parser combinator, that has type $P(\alpha \to \beta) \to P(\alpha) \to P(\beta)$ (but it is more general, because the classic combinator cannot be defined for pretty-printers). For parsers, p_1`<*>`p_2 uses p_1 to parse the beginning of the string, then p_2 to parse the rest of the string, and returns the pair of their results. For pretty-printers, p_1`<*>`p_2 uses p_1 to print the first element of the pair, p_2 to print the second and concatenates the results.

The combinator `<|>` has type $P(\alpha) \to P(\alpha) \to P(\alpha)$. It expresses choice between two parsers/pretty-printers. The instantiation is easy this time: p_1`<|>`p_2 tries both parsers/pretty-printers, uses p_1 if it returns something and uses p_2 otherwise.

They also define basic parsers and pretty-printers. *pure* has type $\alpha \to P(\alpha)$. *pure* x can be instantiated as a parser that associates x to the empty string, and as a pretty-printer that prints the empty string when it has input x and fails otherwise. *token* has type $P(Char)$, it can be instantiated as a parser that associates to each character itself and fails on the empty string, and as a pretty-printer that prints the character it receives as its input.

They then show how to use the basic parsers/pretty-printers and the combinators to describe more complex parsers and pretty-printers, giving the example of a parser/pretty-printer for lists.

More recently, Affeldt, Nowak and Oiwa have used invertible syntax descriptions to prove that a parser and a pretty-printer for a network protocol (a fragment of TLS) are consistent [1]. In this specific case, since the parser and pretty-printer are inverse to each other, it is obvious that they are consistent (the hard part was to prove that they are inverse to each other). The idea of studying the relation between parsing and pretty-printing stems from this work.

Chapter 2

Global Parsing and Pretty-Printing

2.1 Attempt at Defining Parsing and Pretty-Printing

After having read a few articles about parsing and pretty-printing, it came to me that there was no formal definition of parsing and pretty-printing. The obvious thing to do to study the relation between parsing and pretty-printing was first to define parsing and pretty-printing.

Seeing how there is a convenient framework for parsing (monads), I wondered whether the same kind of framework could be found for pretty-printing. It is well-known to people who have studied parsing and pretty-printing that pretty-printing cannot be expressed as a monad. Indeed, as I wrote above, a pretty-printer can be seen as a function that has type $\alpha \rightarrow String$, since it prints internal data. If pretty-printing could be expressed as a monad, one should be able to write a *bind* function with type $(\alpha \rightarrow String) \rightarrow (\alpha \rightarrow \beta \rightarrow String) \rightarrow String \rightarrow \beta$, which is impossible because we need to produce an element with type β but no function can produce it. The only solution left is that *bind p f* must always fail, but then *result* and *bind* cannot verify the laws of the monad. Parametrized monads, which can express more than monads (for example, it can express memory usage with changing types while monads can only express memory usage without changing types) aren't more useful than monads here as the very same argument shows that they can't express pretty-printing.

Then, I wondered if it is possible to express pretty-printing using comonads. A comonad has a type constructor W and two functions *extract* with type $W\alpha \rightarrow \alpha$ and *cobind* with type $W\alpha \rightarrow (W\alpha \rightarrow \beta) \rightarrow W\beta$, which must also satisfy some properties. However, they cannot express pretty-printing as we cannot write the *extract* function for pretty-printing. Indeed, this function would have type $(\alpha \rightarrow String) \rightarrow \alpha$, so we need to produce an alpha no function can produce. The only choice left is that the pretty-printer must always fail, but as above, this contradicts the comonad laws.

These attempts at defining parsing and pretty-printing made me realise that maybe there is no way to define parsing and pretty-printing separately. The problem that arises when I tried to define them separately was that there is always something *arbitrary* in their definition. When a parser parses a string, the result depends entirely on what we want to do with it. In real parsers, for example parsers produced using *yacc*, parsers parse the string according to a given grammar, but they then transform the derivation tree they obtain into internal data. How they transform this derivation tree into internal data is arbitrary. Like in the example in the beginning, I gave two possible trees that a parser that parses `"(1+2)*3"` can return, but it could also return 9, *arithmetic expression*, *non-empty string* or a variety of other results...

What a string is parsed into makes sense only because there is a pretty-printer to print the data that was parsed. Since both the data and the strings don't have other meanings than the ones we give them, the data is not related to the strings a priori. That is why I say that when a parser parses a string into some data or when a pretty-printer prints some data into a string, what string is linked to what internal element is arbitrary as long as parsers and pretty-printers are considered on their own.

2.2 Definition of a Parser/Pretty-Printer Pair

Before I began my internship, my supervisor, David Nowak, had studied parsing and pretty-printing and had tried to find a definition of the consistency between parsing and pretty-printing. He found some relations that he thought parsing and pretty-printing should verify. The most important are, if we note f a parser and g the corresponding pretty-printer, $f \circ g \circ f = f$ and $g = g \circ f \circ g$ (with the formalism he used (functional parsing), the relation $f \circ g = id$ didn't hold, while it holds with the formalism I use). These relations and the fact that parsing and pretty-printing are nearly inverse to each other made him think that maybe parsing and pretty-printing form a Galois connection. Therefore, my approach was to see if the notion of Galois connection could be used to define parsing and pretty-printing.

There are two points we saw only retrospectively and that support the idea that the notion of Galois connection is appropriate to define parsing and pretty-printing. A Galois connection is made of two functions. The previous attempts I made at defining parsing and pretty-printing failed inter alia because parsing and pretty-printing were considered on their own, which is not the case with Galois connections. The other point is that there is a notion of 'beauty' necessary to define pretty-printers, so we need to be able to say if a string is prettier than another one. This means that there is an order (actually a pre-order) on strings induced by the pretty-printer. Since Galois connections are defined between pre-ordered sets, the fact that there is a natural pre-order on strings supports the idea of using Galois connections.

We remind the reader that a pre-order is a reflexive and transitive binary relation and a (partial) order is a reflexive, transitive and antisymmetric relation.

We will note $\mathcal{S}$ the set of strings, $\mathcal{T}_\mathcal{G}$ the set of derivation trees induced by $\mathcal{G}$, and $\mathcal{A}$ will denote a set of semantics. We will note A, A_1, A_2... subsets of $\mathcal{A}$ and S, S_1, S_2... subsets of $\mathcal{S}$.

Let us recall what a Galois connection is:

Definition (Galois connection)**.** *A* Galois connection *between two pre-ordered sets* $(A, \leqslant)$ *and* $(B, \preceq)$ *is a pair of functions* (f, g)*, with* $f : A \to B$ *and* $g : B \to A$*, that verifies:*
f and g are monotone
$\forall a \in A, \forall b \in B.\ a \leqslant g(f(a))$ *and* $f(g(b)) \preceq b$

I roughly describe here my vision of how parsers and pretty-printers operate. A parser parses a string according to a given grammar and then associates a semantics in $\mathcal{A}$ to the resulting derivation tree. A pretty-printer chooses the 'prettiest' derivation tree corresponding to a semantics and then prints the corresponding string.

Therefore, there are three levels involved in parsing and pretty-printing: strings, derivation trees and semantics. The arbitrary part is the link between the derivation trees and the semantics. Only when parsing and pretty-printing are considered together does the link between strings and semantics make sense, as they intuitively have to verify some properties. In my opinion, that is the heart of parsing and pretty-printing.

Using what I said before, we can see a parser as a function $f : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, since there may be several derivation trees corresponding to a single string and these trees may be associated to different semantics. Therefore, parsers can be non-deterministic. However, a pretty-printer is a function $g : \mathcal{A} \to \mathcal{S}$, since there is a single 'prettiest' derivation tree corresponding to a semantics, and therefore a single 'prettiest' string associated to a semantics.

Since there is an asymmetry between the domains and codomains of f and g, to define a Galois connection, we need to lift the functions so that the range of f is the codomain of g and vice versa.

We will define two kinds of *lifted functions.* If $f : A \to B$, we note $f^{\to} : \mathcal{P}(A) \to \mathcal{P}(B)$ the function defined by $f^{\to}(\{a_i \,|\, i \in I\}) = \{f(a_i) \,|\, i \in I\}$. If $f : A \to \mathcal{P}(B)$, we note $f^* : \mathcal{P}(A) \to \mathcal{P}(B)$ the function defined by $f^*(\{a_i \,|\, i \in I\}) = \bigcup_{i \in I} f(a_i)$.

To pretty-print a semantics, a pretty-printer chooses the 'prettiest' derivation tree associated to that semantics and prints the string that corresponds to that derivation tree. Therefore, we need a formal way to define the 'prettiest' derivation tree. The way I formalized it was to use a partial order.

Definition (beauty order)**.** *A partial order $\sqsubseteq$ on a set $\mathcal{T}$ of derivation trees is called a* beauty order *corresponding to a set of semantics $\mathcal{A}$ if it verifies the following properties:*

$\mathcal{T}$ is partitioned in $Card(\mathcal{A})$ classes $\{C_i \,|\, i \in I\}$ such that $\forall t, t' \in \mathcal{T}.\ t \sqsubseteq t'$ implies that t and t' are in the same class ;

for each class C_i there exists a strictly greatest element for $\sqsubseteq$ restricted to C_i, formally: $\exists t \in C_i. \forall t' \in C_i \backslash \{t\}.\ t' \sqsubseteq t \wedge t \not\sqsubseteq t'$.

The idea behind the definition of a beauty order is that $t_1 \sqsubseteq t_2$ if and only if t_2 is prettier than t_1. The properties are meant to ensure that the derivation trees that can be compared are only the ones that corresponds to the same semantics (each class is seen as the set of trees that are associated to a given semantics, so we need as many classes as semantics) and that, for each semantics, there is exactly one prettiest derivation tree (the greatest element of the class associated to that semantics).

However, this is not enough yet. We would like to derive a pre-order on $\mathcal{S}$ using the beauty order on $\mathcal{T}_{\mathcal{G}}$ by doing $S_1 \preceq S_2$ if and only if $T_1 \sqsubseteq^{\to} T_2$ where T_i is the set of derivation trees corresponding to a string in S_i. However, this requires that $\sqsubseteq$ be lifted to $\mathcal{P}(\mathcal{T}_{\mathcal{G}})$.

After having searched for a canonical way to lift an order from any set A to $\mathcal{P}(A)$ without finding any, I decided to lift the order on $\mathcal{T}_{\mathcal{G}}$ using the properties of the beauty order.

Definition (lifted order)**.** *If $\leqslant$ is an order on any set A, we call* lifted order *and note $\leqslant^{\to}$ the relation on $\mathcal{P}(A)$ defined by: $A_1 \leqslant^{\to} A_2$ if and only if $\forall a_1 \in A_1. \exists a_2 \in A_2.\ a_1 \leqslant a_2$ and $\forall a_2 \in A_2. \exists a_1 \in A_1. \exists a^+ \in A.\ a_1 \leqslant a^+ \wedge a_2 \leqslant a^+$.*

In general, $\leqslant^{\to}$ is not even a pre-order.

Properties. *Here are two properties on lifted orders.*

If $\leqslant$ is the equality on $\mathcal{A}$, then $\leqslant^{\to}$ is the equality on $\mathcal{P}(\mathcal{A})$.

If $\sqsubseteq$ is a beauty order on $\mathcal{T}_{\mathcal{G}}$, then $\sqsubseteq^{\to}$ is a pre-order on $\mathcal{P}(\mathcal{T}_{\mathcal{G}})$.

Proof. *Let us prove these two properties.*

Let us suppose that $\leqslant$ is the equality on $\mathcal{A}$. If $A_1 = A_2$, then $A_1 \leqslant^{\to} A_2$ (very easy, take $a_1 = a_2 = a^+$ in the definition of lifted order). If $A_1 \leqslant^{\to} A_2$, then $A_1 \subseteq A_2$ (first part of the definition of lifted order) and $A_2 \subseteq A_1$ (second part of the definition), so $A_1 = A_2$. Therefore $A_1 \leqslant^{\to} A_2$ if and only if $A_1 = A_2$.

We need to prove that $\sqsubseteq^{\to}$ is reflexive and transitive. Reflection was done above. Let us suppose $T_1 \sqsubseteq^{\to} T_2$ and $T_2 \sqsubseteq^{\to} T_3$. Be $t_1 \in T_1$, according to the first part of the definition of *lifted order, there is $t_2 \in T_2$ such that $t_1 \sqsubseteq t_2$, and using the definition a second time, there is $t_3 \in T_3$ such that $t_2 \sqsubseteq t_3$, so $t_1 \sqsubseteq t_3$. Be $t_3 \in T_3$, using the second part of the definition, there exists t_2 and t^+ such that $t_2 \sqsubseteq t^+$ and $t_3 \sqsubseteq t^+$. Therefore, t_2 and t_3 are in the same class. Using the definition one more time, we have there exists $t_1 \in T_1$ that is in the same class. Since we know there is a greatest element in that class (let us note it $\tilde{t}$), we have $t_1 \sqsubseteq \tilde{t}$ and $t_3 \sqsubseteq \tilde{t}$. Therefore, $\sqsubseteq^{\rightarrow}$ is transitive.*

We can now define the pre-order $\preceq$ on $\mathcal{S}$ derived from the pre-order $\sqsubseteq^{\rightarrow}$ on $\mathcal{P}(\mathcal{T_G})$ as we wanted to: $S_1 \preceq S_2$ if and only if $T_1 \sqsubseteq T_2$ where T_i is the set of derivation trees associated to a string in S_i ($\preceq$ is obviously a pre-order).

Since there is no natural way to compare the semantics (especially if we see that order as a way to compare beauty), we will define the order on $\mathcal{A}$ as the equality. When we lift it to $\mathcal{P}(\mathcal{A})$, it is still the equality. With this, can finally give the definition of a parser/pretty-printer pair.

Definition (parser/pretty-printer pair)**.** *Let $\mathcal{A}$ be a set of semantics and $\mathcal{G}$ a context-free grammar. A* parser/pretty-printer pair *is a pair a functions (f, g) with $f : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$ and $g : \mathcal{A} \rightarrow \mathcal{S}$ such that $(f^*, g^{\rightarrow})$ is a Galois connection between the pre-ordered sets $(\mathcal{P}(\mathcal{S}), \preceq)$ and $(\mathcal{P}(\mathcal{A}), =)$ where $\preceq$ is derived from a lifted beauty order on $\mathcal{T_G}$ corresponding to $\mathcal{A}$.*

2.3 Properties

Let us look in greater detail at the definition of a parser/pretty-printer pair. Let us translate the definition of a Galois connection in the particular case of parsing and pretty-printing.

The parser f^* must be monotone, which means that if $S_1 \preceq S_2$ then $f^*(S_1) = f^*(S_2)$. This means that two sets of strings are comparable only if they have the same semantics, which is pretty natural, since that pre-order is supposed to measure the beauty of a string, two strings that don't have the same semantics cannot be compared.

The pretty-printer $g^{\rightarrow}$ must be monotone, which means that if $A_1 = A_2$ then $g^{\rightarrow}(A_1) \preceq g^{\rightarrow}(A_2)$, which is obviously always true.

$\forall S \subseteq \mathcal{S}.\ S \preceq g^{\rightarrow}(f^*(S))$, which means that every set of strings is smaller, i.e. less beautiful, than when it is parsed and pretty-printed (so we really are *pretty*-printing).

$\forall A \subseteq \mathcal{A}.\ f^*(g^{\rightarrow}(A)) = A$, which means that semantics are unchanged through pretty-printing and parsing.

The properties $f \circ g \circ f = f$ and $g \circ f \circ g = g$ we wanted to verify cannot be verified because the range and codomains don't match, but the same properties are verified for the lifted functions $f^* \circ g^{\rightarrow} \circ f^* = f^*$ and $g^{\rightarrow} \circ f^* \circ g^{\rightarrow} = g^{\rightarrow}$ (since $f^* \circ g^{\rightarrow} = id$).

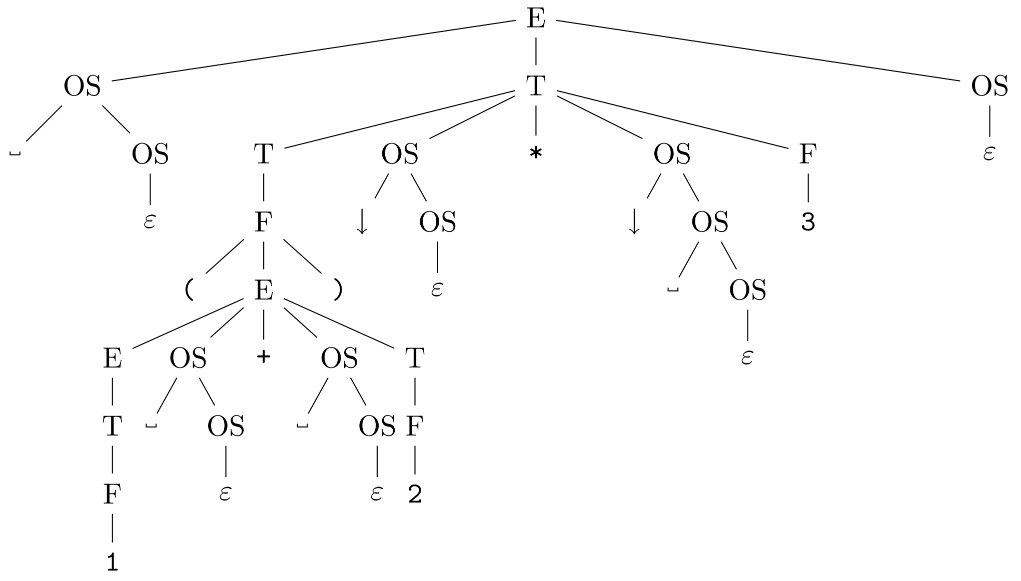
Another property we have is that when the prettiest string associated to a semantics is parsed, the result is exactly that semantics. Indeed, if $g(a) = s$ and $f(s) = A$ where $A \neq \{a\}$, this contradicts the fact that $f^*(g^{\rightarrow}(\{a\})) = \{a\}$. It seems reasonable to ask for the prettiest string associated to a semantics to be associated to this semantics only. A counter-example one may think of could be the programming language C, where the prettiest form of the integer 1 could be `"1"`, and `"1"` would have two semantics: 1 as an integer and 1 as a real number. However, this is not really a counter-example, as `"1"` *does not* have two semantics. What is important to notice here is that the definition given is for *global* parsing and pretty-printing, which means that `"1"` is not seen as a substring of another expression to be parsed (for example `"1 + 0.2"`), in which `"1"` has a real number semantics, but is seen as a string on its own. If `"1"` is typed on its own in C, its semantics is only that of the integer 1.

2.4 Example

This time, let us give another, more flexible, grammar for arithmetic expressions.

$\mathcal{G}_{ex} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, E)$ where the set of non-terminals is $\mathcal{N} = \{E, T, F, OS\}$, the set of terminals is $\mathcal{T} = \mathbb{N} \cup \{$`+`, `-`, `*`, `/`, `(`, `)`, $\downarrow$, ␣$\}$ (as before, this is not authorized, but could be simulated), the set of derivation rules is $\mathcal{R} = \{E \to OS\ E\ OS$`+`$OS\ T\ OS \mid OS\ E\ OS$`-`$OS\ T\ OS \mid OS\ T\ OS, T \to T\ OS$`*`$OS\ F \mid T\ OS$`/`$OS\ F \mid F,\ F \to$ `(`E`)` $\mid$ `i` for $i \in \mathbb{N}, OS \to \downarrow OS \mid$ ␣ $OS \mid \varepsilon\}$ and the start symbol is E. ␣ represents a space and $\downarrow$ a line break. OS thus indicates optional spacing. This grammar is more flexible than $\mathcal{G}_{toy}$ and can typically be used to parse arithmetic expressions written by humans. It can parse strings such as `"(1+2)*3"`, but also `"(1␣+␣2)␣*␣3"` and `"␣(1␣+␣2)↓ *↓ ␣3"`. I apologize for the notations which aren't exactly easy to read.

For example, the derivation tree of `"␣(1␣+␣2)↓ *↓ ␣3"` is:



Let us define a set of semantics $\mathcal{A}$ as the set of binary trees with nodes labelled with $+$, $-$, $*$ or $/$ and leaves are labelled with integers.

Let us define the following parser: $f : \mathcal{S} \to \mathcal{P}(\mathcal{A})$ that parses the string according to the grammar $\mathcal{G}$ and then translates the result into the 'expected' semantics in $\mathcal{A}$ (the link between trees in $\mathcal{T}_{\mathcal{G}}$ and $\mathcal{A}$ is very simple).

There are pretty much no constraints on pretty-printers. Since they're just functions, we can choose 'manually' a derivation tree associated to a semantics (the only thing it must verify is that it *is* associated to this semantics) and claim that it is the prettiest one. However, it may be useful to have a pretty-printer that is easily computable.

We will define different pretty-printers for these semantics using a slightly simplified version of the formalism of *boxes* used in the pretty-printing library in *OCaml* (this method is ad hoc, but this is only meant to be an example). The idea behind boxes is that when a string is inside a box, if there is a line break in this string, instead of going to the start of the next line, there is a tabulation that is aligned with the start of the box.

Here is an example: if we write `[]` to denote a box, then the string `"([foo↓bar])"` is `"(foo↓␣bar)"` and the string `"1␣+␣[foo↓bar]"` is `"1␣+␣foo↓␣␣␣␣bar"`.

The way we will use boxes is very simple: we will define the prettiest form recursively on the semantics tree, and the prettiest forms of the subtrees will be put inside boxes. In the next paragraphs, I will often write 'pretty-print' for 'parse then pretty-print' for a lighter style.

2.4.1 Optional Spacing

The first and most simple pretty-printer we can define is the following: write everything on the same line, with parentheses around every sub-expression and a single space around each operator. This would pretty-print "␣(1␣+␣2)↓ *↓ ␣3" to "((1)␣+␣(2))␣*␣(3)". Another very simple pretty-printer that can be defined is the following: write the tree $op(t_1, t_2)$ as "␣(s_1)↓o↓␣(s_2)" where s_i is the prettiest form of t_i and o corresponds to the operator. This would pretty-print "␣(1␣+␣2)↓ *↓ ␣3" to "␣(␣(1)↓␣␣+↓␣␣␣(2))↓*↓␣(3)".

Let us explain why they are parser/pretty-printer pairs according to the definition we gave. First, we need to define the beauty order. The classes are $\mathcal{T}_{\mathcal{G}_{ex}} = \bigsqcup_{a\in\mathcal{A}} C_a$ where C_a contains all the trees associated to the semantics a (we will always define the classes this way). The beauty order $\sqsubseteq$ on C_a is defined by $\forall t, t' \in C_a.\ t \sqsubseteq t'$ if and only if $t = t' \vee t' = \tilde{t}$ where $\tilde{t}$ is the tree we defined recursively as the prettiest one. Then, the properties that define a Galois connection follow easily:

f^* is monotone: if $S_1 \preceq S_2$, that means that $T_1 \sqsubseteq T_2$ (where T_i is the set of derivation trees associated to S_i), which implies that T_1 and T_2 have trees in exactly the same classes C_{a_i}, so are associated to exactly the same semantics, so $f^*(S_1) = f^*(S_2)$.

$\forall S \subseteq \mathcal{S}.\ S \preceq g^{\rightarrow}(f^*(S))$: be $S \subseteq \mathcal{S}$, then $g^{\rightarrow}(f^*(S))$ is a set the set of prettiest strings corresponding to the semantics $f^*(S)$ (by definition of f, g and $\sqsubseteq$). Since the prettiest strings correspond to a single semantics, we have that the set of derivation trees associated to $g^{\rightarrow}(f^*(S))$ has trees in exactly the same classes as the set of derivation trees associated to S. Moreover, the former possesses all the greatest elements of these classes, so $S \preceq g^{\rightarrow}(f^*(S))$.

$\forall A \subseteq \mathcal{A}.\ f^*(g^{\rightarrow}(A)) = A$: be $A \subseteq \mathcal{A}$, $(g^{\rightarrow}(A))$ is the set of prettiest strings associated to these semantics. Since each prettiest string is associated to a single semantics, $f^*(g^{\rightarrow}(A)) = A$.

While these two parser/pretty-printers pairs handle optional spacing, they do not handle operator priority nor associativity, since the pretty-printers put parentheses everywhere.

2.4.2 Priority and Associativity

What we can do is get rid of the useless parentheses (here, it is done for the first pretty-printer, but it could be done for the second one as well). To do that, one possibility is to try to pretty-print $op(t_1, t_2)$ as the following in order and reject if the semantics is wrong (we just need to parse the resulting string to know that): "s_1␣o␣s_2", "(s_1)␣o␣s_2", "s_1␣o␣(s_2)" and "(s_1)␣o␣(s_2)" (we know that the last one will work anyway). This would pretty-print "␣(1␣+␣2)↓ *↓ ␣3" to "(1␣+␣2)␣*␣3". It would also pretty-print "(1␣*␣2)␣+␣3" to "1␣*␣2␣+␣3". This handles operator priority. It also handles left associativity: "(1␣+␣2)␣+␣3" would be pretty-printed to "1␣+␣2␣+␣3" (it would be the same for other operators). However, it doesn't handle right associativity: "1␣+␣(2␣+␣3)" will be pretty-printed to "1␣+␣(2␣+␣3)". That is because "1␣+␣(2␣+␣3)" will be parsed to $+(1, +(2, 3))$, which is different from $+(+(1, 2), 3)$, which is the semantics associated to "1␣+␣2␣+␣3".

Like in the previous part, we could show that they define parser/pretty-printer pairs (the beauty order is defined more or less the same way).

Therefore, we want to review our semantics and how we parse. When we see a tree with a subtree $+(t_1, +(t_2, t_3))$ or $*(t_1, *(t_2, t_3))$ after parsing, we change the subtree to $+(+(t_1, t_2), t_3)$ or $*(*(t_1, t_2), t_3)$ respectively. We have created a parser for a new set of semantics (more complicated than the previous one, but that will handle right associativity as well). If we do the same as above, we handle optional spacing, operator priority and associativity, three important features of parsing and pretty-printing.

2.4.3 A More Advanced Pretty-Printer

The pretty-printers we have defined until now are still really simple. Even though we don't want to do something too complex, a few other features would be nice. The feature I propose here is one that could in my opinion be useful. There are a lot of pretty-printers that are meant to print human-readable data. However, all the examples I know except one [8] disregard a very important point: when data is printed, it is printed onto a screen (most of the time), and that screen's width is bounded. The pretty-printer I want to write will write everything on a single line until it grows too long for the line, at which point it will decompose the arithmetic expression according to its structure.

The algorithm proceeds as follows: to pretty-print $op(t_1, t_2)$, we suppose we have a list of strings associated to t_1 and t_2 ordered by decreasing beauty order (let us call them $l_i = [s_{i,1}, ..., s_{i,n_i}]$) and we want to create the list of strings associated to $op(t_1, t_2)$ ordered by decreasing beauty order (the prettiest string associated with $op(t_1, t_2)$ will be the head of the list).

If $s_{1,1}$ and $s_{2,1}$ are shorter than the line, consider the string "$s_{1,1}$␣o␣$s_{2,1}$". There are three possibilities: if the semantics is wrong then discard the string, if it is longer than the line then put it at the end of the list, if it is smaller than the line then put it at the head of the list. Do the same for the strings "($s_{1,1}$)␣o␣$s_{2,1}$", "$s_{1,1}$␣o␣($s_{2,1}$)" and "($s_{1,1}$)␣o␣($s_{2,1}$)" in that order. If a string has already been put at the head or the end of the list, the new string is put 'inside' the list (once a string is put somewhere in the list, it 'does not move'). Repeat the same with s_{1,j_1} and s_{2,j_2} as long as these strings are smaller than the line (the exact order in which the s_{1,j_1} and s_{2,j_2} are considered is not really important, $s_{i,j}$ must only be considered before $s_{i,j+1}$).

Only for $s_{i,j}$ smaller than the line (with the same constraint on the order as above), consider the following strings in order "␣s_{1,j_1} ↓o↓␣s_{2,j_2}", "␣s_{1,j_1} ↓o↓␣(s_{2,j_2})", "␣(s_{1,j_1})↓o↓␣s_{2,j_2}" and "␣(s_{1,j_1})↓o↓␣(s_{2,j_2})" and do the same as above.

Now, for every s_{1,j_1} and s_{2,j_2} (the order in which they are considered must still follow the same rule as above), consider the strings "␣s_{1,j_1} ↓o↓␣s_{2,j_2}", "␣s_{1,j_1} ↓o↓␣(↓␣␣s_{2,j_2} ↓␣)", "␣(↓␣␣s_{1,j_1} ↓␣)↓o↓␣s_{2,j_2}" and "␣(↓␣␣s_{1,j_1} ↓␣)↓o↓␣(↓␣␣s_{2,j_2} ↓␣)" in that order and do the same as above.

The resulting list is associated to $op(t_1, t_2)$.

This time, the beauty order is a little more complex. Even though the classes are defined the same way, we now have a list $[s_1, ..., s_n]$ of strings ordered by decreasing beauty order. We can then define $\sqsubseteq$ on C_a as $s \sqsubseteq s_n \sqsubseteq s_{n-1} \sqsubseteq \ldots \sqsubseteq s_2 \sqsubseteq s_1$ for $s \notin \{s_1, ..., s_n\}$. This time again, the properties that define a parser/pretty-printer pair follow (the reasoning is the same as before).

Please note that this algorithm (and the previous ones as well) do not aim at being efficient, they're only meant to be proofs of concept. Also, while this does seem to work on examples with low tree depth, I don't know how this handles examples with higher tree depth, since the order on the strings associated to a semantics become complex.

Chapter 3

Functional Parsing and Pretty-Printing

3.1 Limits of Global Parsing and Pretty-Printing

Even though the definition of global parsing and pretty-printing sounds very powerful there are some limits to it. Actually, it is also *because* it is very powerful that there are some limits to it. Let us illustrate this point with a simple example. If we have a parser/pretty-printer pair that parses integers that can be written in two different ways (let us say i can be written `"i"` in decimal and `"0xi"` in hexadecimal). And let us code Turing machines into integers. Now, the parser is very simple: it just maps the decimal and hexadecimal representations of an integer to that integer. The pretty-printer is more complex: it maps i to its decimal representation if the Turing machine that corresponds to i stops and to its hexadecimal representation if it doesn't stop. Theoretically, those two functions verify the definition of a parser/pretty-printer pair (and there is no reason to think they're not a parser and a pretty-printer), but they cannot be effectively computed. This shows that this definition is very general, but, because of this generality, it cannot always be verified that two functions form a parser/pretty-printer pair.

Another practical limitation of global parsing and pretty-printing is that, to use this definition, we need a grammar to parse the expressions and we need to find a way to specify how to associate derivation trees to semantics. This may become a very difficult task when the expressions studied become complex (for example, if we want to write a parser/pretty-printer pair for a programming language like *C* or *OCaml*). Therefore, it seems reasonable to study the case of functional parsing and pretty-printing, as there would be no need to explicitly define the whole grammar and how the derivation trees of this grammar are associated to semantics: we would only need to do that for basic parsers and pretty-printers, and show how semantics are transformed when parsers and pretty-printers are combined.

3.2 Attempts at Defining a Functional Parser/Pretty-Printer Pair

The main difference between a global parser/pretty-printer pair and a functional one is that, while a global parser parses the whole string, a functional parser can parse the beginning of the string and leave the rest for another functional parser to parse. Of course, when a functional parser is considered as a global parser, the only relevant results are those that were found while parsing the whole string. For example, a functional parser for arithmetic expressions that parse the string `"1*(2+3)"` will return a list with the semantics associated to `"1"` and the remaining string `"*(2+3)"` and the semantics associated to `"1*(2+3)"` and the remaining string ε. When it is considered as a global parser, we're only interested in the semantics associated to `"1*(2+3)"`, but when it is seen as a functional parser, both semantics are interesting because we don't know how much of the string that parser had to parse (because there are other parsers to parse the rest of the string).

This difference in behavior induces a difference in the nature of the objects we manipulate. While global parsers only had to associate semantics to strings, and therefore were functions $f : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, they now have to return strings with each semantics, and therefore are functions $f : \mathcal{S} \to \mathcal{P}(\mathcal{A} \times \mathcal{S})$. Moreover, for pretty-printers to have a symmetrical behavior (because we hope that the definition of functional parsing and pretty-printing will still be based on Galois connections), they will have to include the fact that semantics are paired with strings, so they have type $g : \mathcal{A} \times \mathcal{S} \to \mathcal{S}$, the idea being that g maps (a, s) to the prettiest string associated to a concatenated with s.

Another very important difference between global and functional parsing/pretty-printing is that, in functional parsing/pretty-printing, pretty-printers can fail (this wasn't the case with global pretty-printing).

Therefore, the sets we need to consider are a little different: parsers are still functions $f : \mathcal{S} \to \mathcal{P}(\mathcal{A} \times \mathcal{S})$, but pretty-printers are now functions $g : \mathcal{A} \times \mathcal{S} \to \mathit{Just}(\mathcal{S})$, where $*$ denotes failure. We can lift functions to $f^* : \mathcal{P}(\mathit{Just}(\mathcal{S})) \to \mathcal{P}(\mathit{Just}(\mathcal{A} \times \mathcal{S}))$ and $g^{\to} : \mathcal{P}(\mathit{Just}(\mathcal{A} \times \mathcal{S})) \to \mathcal{P}(\mathit{Just}(\mathcal{S}))$.

The idea I followed was to stick to the framework of *invertible syntax descriptions*: find a definition that would be verified for basic parsers and that would be preserved when combining parsers. Of course, this definition should also fall back on that of a global parser/pretty-printer pair when the strings are ignored. What we hope for is that by relaxing the definition of a Galois connection, we will be able to define functional parsing and pretty-printing.

We tried to relax it this way: f^* and $g^{\to}$ must still be monotone, $\forall S \subseteq \mathit{Just}(\mathcal{S})$. $S \preceq g^{\to}(f^*(S))$ must still hold, but the last condition becomes:

$\forall A \subseteq \mathit{Just}(\mathcal{A})$. $\pi(f^*(g^{\to}((\mathit{map}\ \sigma)(A)))) = A$ where $\sigma : a \mapsto (a, \varepsilon)$ and π returns the list of as such that (a, ε) is in its input list.

What we're trying to do here is considering f^* and $g^{\to}$ as global: it doesn't change anything from the string's point of view, but from the semantics point of view, we have to get rid of the places where strings appear.

However, this doesn't work, as it is not preserved by sequential combination of parsers. Here is a counter-example. Let f be the parser that parses a string to ϵ if it is empty and $\bar{\epsilon}$ otherwise. Let g be a pretty-printer that corresponds to f: it maps ϵ to ε and $\bar{\epsilon}$ to `"a"`. Now, let us consider the parser $f' = f$`<*>`f, the corresponding pretty-printer $g' = g$`<*>`g and the semantics $(\epsilon, \bar{\epsilon})$. We have $g'^{\to}(\{((\epsilon, \bar{\epsilon}), \varepsilon)\}) = \{(\varepsilon, \texttt{"a"}), \varepsilon\} = \{\texttt{"a"}\}$ and $f'^*(\{\texttt{"a"}\}) = \{((\epsilon, \epsilon), \texttt{"a"}); ((\epsilon, \bar{\epsilon}), \varepsilon); ((\bar{\epsilon}, \epsilon), \varepsilon)\}$, so $\pi(f^*(g^{\to}((\mathit{map}\ \sigma)(\{(\epsilon, \bar{\epsilon})\})))) = \{(\epsilon, \bar{\epsilon}); (\bar{\epsilon}, \epsilon)\} \neq \{(\epsilon, \bar{\epsilon})\}$.

We haven't found a definition for functional parsing/pretty-printing yet. It could prove to be a very interesting direction for future research, especially if someone wants to use this definition to formally prove real parser/pretty-printer pairs.

Conclusion

In my opinion, the subject I studied during my internship was very important, as the definition I found of a parsing/pretty-printing pair, if it is accepted by the researchers of the community, may help formally verify parsers and pretty-printers and perhaps lay the foundations of a formal theory of parsing and pretty-printing. If it is not accepted, I hope that my work will at least reawaken the interest of researchers for the question of knowing what it means for a parser and a pretty-printer to be consistent.

Though, in retrospect, the result I found doesn't look so complicated, I think I could only find it because I had a different way to look at the problem of parsing and pretty-printing, more specifically that I couldn't imagine them separately.

There are two main research directions that emerge now. The first one is to find a way to make the definition I found practical. One way that looks very interesting is to find a definition of functional parsing and pretty-printing, as it can be a very important definition because it will probably be more practical. Another way of making the definition practical is the implementation of a tool like *yacc*. *yacc* produces a parser based on a given grammar. It could be interesting to develop a tool that produces both a parser and a pretty-printer corresponding to it at the same time.

Another possible research direction would be to work on the orders used in the definition. It could prove interesting to see if the definition can be extended to other orders on semantics than equality to model some functions for whose nature of parser/pretty-printer is disputable. It would also be interesting to characterize the possible orders on strings without having to rely on the derivation trees as it could lead to a more simple definition.

Finally, I would like to thank everyone who welcomed me and helped me during my stay in Japan: David Nowak for supervising my internship, Philippe Codognet for doing a lot of the paperwork for my admission at the University of Tokyo, all the members of the Japanese-French Laboratory for Informatics for their warm welcome, Masami Hagiya, the Dean of the Graduate School of Information Science and Technology at the University of Tokyo, for officially being my supervisor and accepting my application to the Graduate School, Kaori Sato from the Office of International Relations of the Graduate School for her help with the administration at the University of Tokyo, and anyone else that I may have forgotten.

Bibliography

[1] Reynald Affeldt, David Nowak, and Yutaka Oiwa. Formal network packet processing with minimal fuss: invertible syntax descriptions at work. In *Proceedings of the sixth workshop on Programming languages meets program verification*, PLPV '12, pages 27–36, New York, NY, USA, 2012. ACM.

[2] Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 86–97, New York, NY, USA, 2005. ACM.

[3] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 26(1):57–61, January 1983.

[4] Dick Grune. *Parsing Techniques: A Practical Guide.* Springer Publishing Company, Incorporated, 2nd edition, 2010.

[5] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 53–96, London, UK, UK, 1995. Springer-Verlag.

[6] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[7] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[8] Dereck C. Oppen. Pretty-printing. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, October 1980.

[9] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. *SIGPLAN Not.*, 45(11):1–12, September 2010.

[10] Peter Smith. The galois connection between syntax and semantics.

[11] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.