Correct, Fast LR(1) Unparsing

François Pottier

Inria Paris

We describe an extension of the LR(1) parser generator Menhir with new features that aim to facilitate *unparsing*, that is, transforming abstract syntax trees back into text. Our method supports non-LR(1) grammars decorated with precedence declarations and guarantees correct unparsing, by which we mean that parentheses or other disambiguation symbols are inserted where necessary. Furthermore, it allows the user to control other aspects of the unparsing process, such as layout. Our contributions include a novel view of unparsing as a composition of several successive transformations; the novel concept of *disjunctive concrete syntax trees* (DCSTs); a fast algorithm that converts DCSTs to ordinary concrete syntax trees (CSTs), thereby deciding where disambiguation symbols must be inserted; and the automated generation of safe APIs for the construction of DCSTs and deconstruction of CSTs.

> In the beginning were the words, and the words were trees, and the trees were words.
>
> Kats, Visser and Wachsmuth [KVW10]

Broadly speaking, *parsing* transforms text into a tree-structured representation; *unparsing* is the opposite transformation. Whereas parsing is a challenging problem, which has been heavily studied in the last sixty years [GJ08], unparsing has received little attention, if any at all. At first, this may seem natural. Whereas parsing aims to *discover* tree structure, unparsing *destroys* this structure—what could possibly be easier?

One must recall, however, that parsing usually includes a transformation of parse trees into abstract syntax trees (ASTs). This transformation typically *destroys* information, such as the placement of parentheses or other syntactic delimiters. Unparsing must *reconstruct* this information. Thus, unparsing is not so easy after all, and *it is possible to get it wrong*. An incorrect unparser might insert too few parentheses and emit a sequence of tokens that does not have the intended meaning.

Parsing and unparsing in multiple steps Parsing is traditionally decomposed into multiple steps, recalled in in Figure 1:

1. A stream of characters is *lexed*, i.e., turned into a stream of tokens.
2. The stream of tokens is *grown*, i.e., turned into a concrete syntax tree (CST).
3. The concrete syntax tree is *interpreted*, i.e., turned into an abstract syntax tree (AST).

Step 2 alone, or the combination of steps 2 and 3, are often referred to as "parsing". We reserve "parsing" to refer to the composition of steps 1, 2, and 3. Concrete syntax trees are also known as "parse trees".

In the OCaml world, step 1 is usually performed with the help of a lexer generator, such as `ocamllex`. Steps 2 and 3 are performed with the help of a parser generator, such as

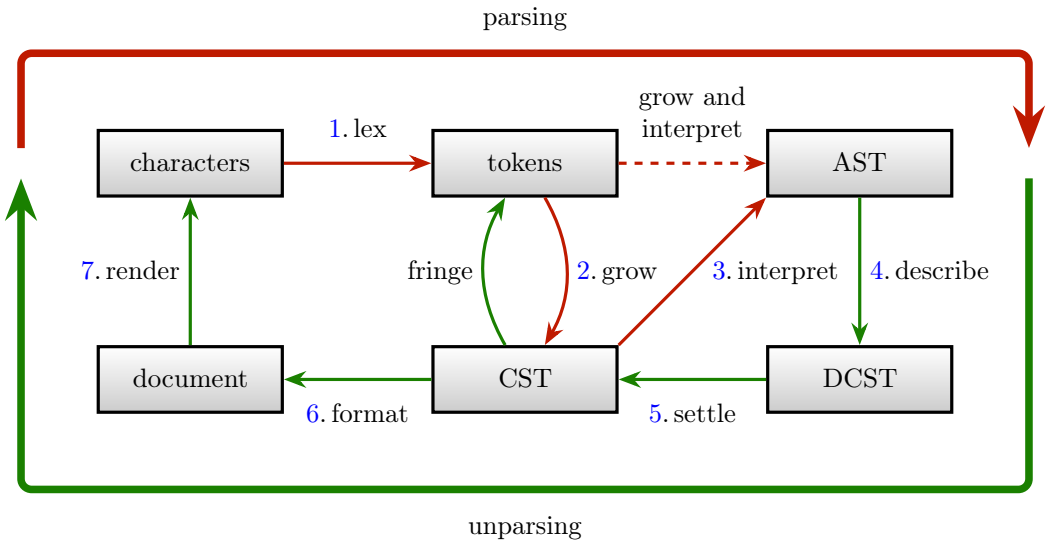


Figure 1. From text to abstract syntax trees and back

Menhir [PRG23]. The latter two steps are usually combined: the grammar is decorated with *semantic actions* that indicate how to transform CSTs into ASTs, so the generated parser transforms a stream of tokens directly into an AST, without building a CST.

We propose to decompose unparsing into several steps, also shown in Figure 1:

4. An AST is first *described*, i.e., turned into a *disjunctive concrete syntax tree* (DCST). This step must be programmed by the user (§1, §2). A DCST may contain *disjunction* nodes. They allow the user to indicate that a subtree admits multiple descriptions, e.g., without or with surrounding parentheses. In the next step, a choice between these descriptions must be made.
5. This DCST is *settled*, i.e., turned into a CST c. This step is performed by an algorithm that we provide (§3). This algorithm settles every choice: at every disjunction node, it decides which subtree must be used. This algorithm provides a correctness guarantee: it chooses a CST c such that the round-trip property $\mathit{grow}(\mathit{fringe}(c)) = c$ holds. It fails if it is unable to satisfy this property.
6. The CST is *formatted*, i.e., turned into a data structure that may include layout information. The *documents* offered by the `PPrint` library [Pot23] are an example of such a data structure. We provide a default implementation of this step in the form of a visitor class for CSTs (§4). By overriding selected methods, the user can modify the default behavior and express a custom layout policy.
7. The document is *rendered*, i.e., turned into a string. This step is performed by the `PPrint` library or by whichever layout library the user chooses to rely upon.

It is up to the user to ensure the correctness of the steps that she implements. The correctness of *describe* is expressed by the equation $\mathit{interpret} \circ \mathit{settle} \circ \mathit{describe}(t) = t$, where t is an AST. The correctness of *format* and *render* is expressed by $\mathit{grow} \circ \mathit{lex} \circ \mathit{render} \circ \mathit{format}(c) = c$, where c is a CST. Ensuring that this property holds requires some care; for instance, it may require sometimes inserting whitespace between two consecutive tokens.[5]

Contribution We describe an extension of Menhir with new facilities for unparsing:

- a DCST construction API (a set of constructor functions), generated, for use in step 4;
- the settlement algorithm, written once and for all, which performs step 5;
- a CST deconstruction API (a visitor class), generated, for use in step 6.

By decomposing unparsing into several steps, we give the user the freedom to control or customize certain steps. The tasks that we leave to the user include programming step 4 and customizing step 6 insofar as desired. We do not attempt to automate these tasks. Automating step 4 would amount to automatically reversing the semantic actions. Removing the need or the ability to customize step 6 would require giving the user some other way of expressing a layout policy. We have not explored these avenues, but others have (§5).

A correctness guarantee Because we let the user implement or customize steps 4 and 6, we cannot guarantee that these steps are correct. Thus, we cannot guarantee an end-to-end round-trip property of the form $\mathit{parse}(\mathit{unparse}(t)) = t$, where t is an AST. Still, we *can* and do guarantee a more restricted property, namely the correctness of step 5, which converts a DCST to a CST. We guarantee that this step always produces a viable CST, where a CST c is *viable* if the round-trip property $\mathit{grow}(\mathit{fringe}(c)) = c$ holds.

The function *fringe* maps a CST to its fringe, a sequence of tokens. The partial function *grow*, which models a deterministic parser, transforms a sequence of tokens into a CST. Thus, the equation $\mathit{grow}(\mathit{fringe}(c)) = c$ means that it is safe to display the CST c in the form of the sequence of tokens $\mathit{fringe}(c)$: there is no danger that this sequence might be rejected or misread as a different CST.

The reverse round-trip property $\mathit{fringe}(\mathit{grow}(s)) = s$ states that if the parser accepts the sentence s and constructs a CST c then the fringe of this CST is s. This means that *grow* is a correct parser. We expect this property to hold, but it is not the focus of this paper.

The reader may wonder: are there non-viable CSTs? and could one fuse steps 4 and 5 and let the user implement a direct transformation of ASTs to CSTs? Our answer is two-fold:

- If the grammar is in the class LR(1) [Knu65, GJ08] then every concrete syntax tree is viable. In this special case, we *could* trust the user to implement a transformation of ASTs into CSTs, as there is no danger that the user might build a non-viable CST.
- If the grammar is not in this class then the deterministic parser that is constructed by Menhir, after resolving LR(1) conflicts, is incomplete: there exist well-formed CSTs that the parser will never produce. In other words, the function *grow* is not surjective. As a result, there exist non-viable CSTs: if a CST c lies outside the range of *grow*, then $\mathit{grow}(\mathit{fringe}(c)) = c$ cannot hold. Furthermore, it is difficult to tell which CSTs are viable and which are not viable: this requires detailed knowledge of the LR(1) automaton and of the manner in which conflicts have been resolved, perhaps under the influence of user-provided *precedence declarations*. Therefore, we *do not* trust the user to build CSTs: she might, by mistake, construct non-viable CSTs.

Our approach, then, is to ask the user to build DCSTs. Thanks to disjunction nodes, a single DCST represents a family of many possible CSTs. Each member of this family may or may not be viable. It is then up to our settlement algorithm to choose a viable CST, if there exists one, among this family. The algorithm has access to a description of the LR(1) automaton and can therefore determine which CSTs are viable.

In other words, one could say that we let the user indicate where parentheses *may* be inserted, and we decide where to *actually* insert parentheses so as to ensure viability.

We do not guarantee optimality, that is, minimal use of parentheses. Instead, we offer a biased disjunction: our settlement algorithm favors the left branch. In most real-world scenarios, this lets the user achieve optimality simply by building disjunction nodes whose meaning is: "first, try displaying the subtree d without parentheses; if this is not permitted, display d surrounded with parentheses". In such a disjunction node, the subtree d is *shared* between the two disjuncts. Thus, although "T" stands for "tree", DCSTs are often DAGs.

If the grammar is in the class LR(1), then the user typically constructs DCSTs that do not contain any disjunction nodes (§1). These DCSTs are also CSTs, and, because in this case every CST is viable, they are viable CSTs. Then, the settlement algorithm behaves as the identity function.

Settlement algorithms We discuss two settlement algorithms, both of which we have implemented. One algorithm is complete: if the DCST d, viewed as a family of CSTs, contains at least one viable member, then $settle(d)$ must succeed. This algorithm, which exploits memoization, has linear time and space complexity in the size of the DAG d, where the constant factor depends on the sizes of the alphabet and automaton. Unfortunately, in practice, it is slow. The other algorithm is simpler, much faster in practice, and has linear time complexity in the tree size of d, which we define below. However, it is incomplete: under certain conditions, it can fail even though the family d contains a viable CST.

Definitions We fix a context-free grammar [GJ08]. We write a, b for terminal symbols and A for a nonterminal symbol. We write s for a sentence (a string of tokens) and α for a sentential form (a string of terminal or nonterminal symbols). We write κ for a *token*. We assume that there is a mapping *sym* of tokens to terminal symbols.

A *concrete syntax tree* (CST) is a tree whose nodes are *terminal nodes*, labeled with a token κ, or *nonterminal nodes*, labeled with a production $A \to \alpha$. A terminal node has no children. A nonterminal node has children whose number and head symbols must match α. Thus, $c ::= T\ \kappa \mid N\ (A \to \alpha, \vec{c})$. The *head symbol* of a CST c, written $head(c)$, is defined by the equations $head(T\ \kappa) = sym(\kappa)$ and $head(N\ (A \to \alpha, \vec{c})) = A$.

A *disjunctive concrete syntax tree* (DCST) may have binary disjunction nodes: thus, $d ::= T\ \kappa \mid N\ (A \to \alpha, \vec{d}) \mid d \vee d$. The two children of a disjunction node $d_1 \vee d_2$ must have the same head symbol, which is considered the head symbol of the disjunction node: $head(d_1 \vee d_2) = head(d_1) = head(d_2)$. A DCST d represents a family of CSTs; we write $c \in d$ when the CST c is a member of this family.

A DCST can be represented as a DAG in memory. We write $\#d$ for the *DAG size* of d, that is, for the number of vertices involved in its representation as a DAG. We write $|d|$ for the *tree size* of d. This function is defined by $|T\ \kappa| = 1$, $|N\ (A \to \alpha, \vec{d})| = 1 + \Sigma|\vec{d}|$, and $|d_1 \vee d_2| = \max(|d_1|, |d_2|)$. Thus, $|d|$ is the maximum size of a CST c such that $c \in d$.

1 Working with a Conflict-Free Grammar

A simple grammar, expressed in Menhir's syntax, appears in Figure 3. This grammar describes arithmetic expressions that include integer literals, additions, multiplications, and parenthesized subexpressions. This grammar is stratified: three syntactic categories, also known as nonterminal symbols, are distinguished. A *factor* is either an integer constant or a parenthesized expression; a *term* is either a factor or the multiplication of a term and a factor; an *expression* is either a term or the addition of an expression and a term. Menhir accepts this grammar without reporting any conflict: this guarantees that this grammar is in the class LR(1), therefore is unambiguous.

To see intuitively why this grammar is unambiguous, we suggest this exercise: construct a CST whose fringe forms the token sequence `INT`(1); `ADD`; `INT`(2); `MUL`; `INT`(3), which corresponds to the characters `1+2*3`. The reader will find that there exists only one such tree, namely the one shown in Figure 4a.

In the grammar of Figure 3, the productions carry semantic actions that construct ASTs. ASTs form an algebraic data type whose definition appears in Figure 2. ASTs do not keep track of parentheses: the semantic action for the production `factor: LPAR; expr; RPAR` (line 12) does not construct an AST node.

An unusual feature of the grammar in Figure 3 is that each production is assigned a unique name via a `@name` attribute. These names are used in the construction of data constructor names and method names in the generated APIs for DCSTs and CSTs. The reader may wish to peek ahead at Figures 5 and 13.

The code in Figures 2 and 3 is written by the programmer. Throughout the paper, hand-written code is shown on a `green background`, whereas code generated by Menhir appears on a `yellow background`.

```
type binop = BAdd | BMul          (* Binary operators *)
type expr =                       (* Expressions *)
  | EConst of int
  | EBinOp of expr * binop * expr
type main = expr
```

Figure 2. Arithmetic expressions: the module `AST`

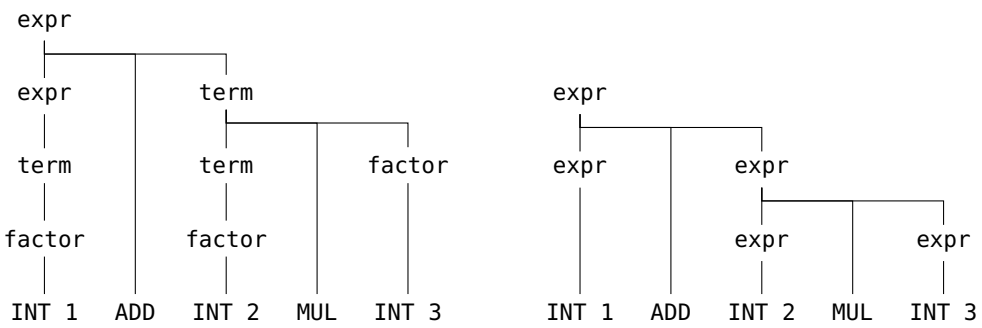
```
%token<int> INT                 (* Tokens *)
%token      ADD  "+"
%token      MUL  "*"
%token      LPAR "("
%token      RPAR ")"
%token      EOL
%start<AST.main> main           (* Start symbol *)
%{ open AST %}
%%                              (* Productions and semantic actions *)
factor:
| i = INT                       { EConst i }            [@name const]
| LPAR; e = expr; RPAR          { e }                   [@name paren]

term:
| f = factor                    { f }                   [@name factor]
| t = term; MUL; f = factor     { EBinOp (t, BMul, f) } [@name mul]

expr:
| t = term                      { t }                   [@name term]
| e = expr; ADD; t = term       { EBinOp (e, BAdd, t) } [@name add]

main:
| e = expr; EOL                 { e }                   [@name eol]
```

Figure 3. Stratified grammar: the parser



(a) Stratified grammar (Figure 3) **(b)** Non-stratified grammar (Figure 7)

Figure 4. Concrete syntax trees for `1+2*3`

```
module DCST : sig
  type factor
  type term
  type expr
  type main
  val factor_choice: factor -> factor -> factor (* Constructors for [factor] *)
  val const: int -> factor
  val paren: expr -> factor
  val term_choice: term -> term -> term         (* Constructors for [term] *)
  val factor: factor -> term
  val mul: term -> factor -> term
  val expr_choice: expr -> expr -> expr         (* Constructors for [expr] *)
  val term: term -> expr
  val add: expr -> term -> expr
  val main_choice: main -> main -> main         (* Constructors for [main] *)
  val eol: expr -> main
end
```

Figure 5. Stratified grammar: signature of the module `DCST`

```
let rec factor : AST.expr -> DCST.factor = function
  | EConst i              -> DCST.const i
  | e                     -> DCST.paren (expr e)

and term       : AST.expr -> DCST.term = function
  | EBinOp (e1, BMul, e2) -> DCST.mul (term e1) (factor e2)
  | e                     -> DCST.factor (factor e)

and expr       : AST.expr -> DCST.expr = function
  | EBinOp (e1, BAdd, e2) -> DCST.add (expr e1) (term e2)
  | e                     -> DCST.term (term e)

and main       : AST.main -> DCST.main = function
  | e                     -> DCST.eol (expr e)
```

Figure 6. Stratified grammar: translating ASTs to DCSTs

Transforming ASTs to CSTs As explained earlier, we propose to break down "unparsing" into several steps. The first two steps, namely steps 4 and 5, aim to transform an AST t into a CST c. Given an AST t, the problem is to construct a CST c that corresponds to t. Writing $interpret_A$ for the transformation of CSTs to semantic values that is defined by the semantic actions, the problem can be stated as follows:

> Given a semantic value t for a nonterminal symbol A,
> produce a CST c whose head symbol is A, such that $interpret_A(c) = t$.

When the grammar is in the class LR(1), there is no danger of picking a non-viable CST c, that is, a CST whose fringe is rejected by the parser or understood by the parser as a different CST c'.[1]

[1] Suppose the grammar is in the class LR(1). Then, there exists a correct and complete parser, which we model as a partial function *grow* of sequences of tokens to CSTs. Let us prove that every CST c satisfies $grow(fringe(c)) = c$. Let c be a CST, and let s stand for its fringe. Then, s must be a grammatically valid sentence. Because the parser is complete, it must accept this sentence: $grow(s)$ must be defined. Let us refer to this CST as c'. Because the parser is correct, we must have $fringe(c') = s$, that is, $fringe(c') = fringe(c)$. Because the grammar is unambiguous, two distinct CSTs cannot have the same fringe: so, $c' = c$ must hold. In other words, $grow(fringe(c)) = c$ holds.

A simple and safe way of implementing the transformation of ASTs to CSTs is as a family of mutually recursive functions whose structure mirrors the structure of the grammar. For each nonterminal symbol A, one writes a function, also named A, whose specification is the one given above: this function transforms an AST t into a CST c whose head symbol is A while respecting the constraint $\mathit{interpret}_A(c) = t$.

By following this pattern, for the stratified grammar of Figure 3, it is straightforward to implement a correct and optimal transformation of ASTs to CSTs. The code is shown in Figure 6. Technically, this code transforms ASTs into DCSTs that do not contain any disjunction nodes. Thus, it implements step 4 of Figure 1. Step 5 in this case is trivial—it is essentially the identity function—and is performed by Menhir's settlement algorithm (§3).

A DCST Construction API The code in Figure 6 relies on a strongly-typed API for constructing DCSTs, offered by the module `DCST`, whose signature is shown in Figure 5. This module is generated by Menhir based on the grammar in Figure 3.

The module `DCST` offers a family of abstract types of DCSTs. For each nonterminal symbol A, there is an abstract type, also named A, of DCSTs whose head symbol is A. Furthermore, for these abstract types, the module `DCST` offers a family of constructor functions. For each production, there is a constructor function, whose name is determined by the production's `@name` attribute. This constructor function takes parameters whose number and types match the right-hand side of the production. This explains why the constructor `const`, which corresponds to the production `factor: INT`, takes one argument of type `int`. If a parameter corresponds to a terminal symbol that does not carry a semantic value, such as `LPAR` or `RPAR`, then this parameter is omitted. This explains why the constructor `paren`, which corresponds to the production `factor: LPAR; expr; RPAR`, takes just one argument of type `expr`. Furthermore and finally, for each nonterminal symbol A, there is a constructor function that constructs a disjunction node. These functions are named `factor_choice`, `term_choice`, `expr_choice`, etc. The code in Figure 6 does not use these functions; their use is illustrated in the next section (§2).

The module `DCST` allows constructing DCSTs but offers no way of deconstructing or inspecting a DCST. The only way of doing anything useful with a DCST is to convert it to a CST. This is done by the settlement algorithm (§3).

Transforming ASTs to CSTs—More Comments Let us momentarily come back to the transformation in Figure 6. The code is straightforward. At each level A of the grammar, if the AST node at hand can be directly represented by a DCST constructor for the nonterminal symbol A, then this constructor is used; otherwise a recursive call to the next lower level is performed. A recursive call from the lowest level (`factor`) back to the highest level (`expr`) is made possible by the constructor `DCST.paren` (line 3), which corresponds to the production `factor: LPAR; expr; RPAR`.

The functions in Figure 6 always construct well-formed DCSTs. This property is enforced by the strongly-typed API of the module `DCST`. If the programmer mistakenly attempted to construct an ill-formed tree, this would be detected by the OCaml type-checker. For instance, attempting to call the function `expr` or `factor` in a position where one must call `term` would give rise to a static type error. This is a pleasant and valuable guarantee.

2 Working with Conflicts and Precedence Declarations

In practice, people commonly work with grammars that lie *outside* the class LR(1). Indeed, manually refactoring a grammar so as to fit in the class LR(1) is usually unpleasant, sometimes difficult, and can lead to a blow-up in the size of the grammar. Instead, following an approach first suggested by Aho, Johnson and Ullman [AJU75] and first implemented in `yacc` [Joh75], people often prefer to keep a non-LR(1) grammar and annotate it with

precedence declarations. When building an LR(1) automaton for such a grammar, the parser generator encounters *conflicts*, that is, situations where several actions are possible, such as "shift", "reduce production p_1", and "reduce production p_2". The parser generator exploits precedence declarations to statically *resolve conflicts*: at parser construction time, in each situation where several actions are possible, at most one action is chosen. Thus, a deterministic LR(1) automaton is obtained.[2]

This automaton is a correct parser, but, because some legitimate actions have been removed, it is *incomplete*: there exist CSTs that it cannot construct. In other words, there exist non-viable CSTs. The trees shown in Figure 9 are examples: the parser will never construct them. As explained earlier, we do not trust the user to write a correct transformation of ASTs into *viable* CSTs. Indeed, because the user usually does not have good knowledge and understanding of the LR(1) automaton, it seems extremely difficult for her to ascertain that only viable CSTs are ever constructed. Instead, our approach is to ask the user to implement a transformation of ASTs into DCSTs and to provide an algorithm that transforms DCSTs into viable CSTs.

An Ambiguous Grammar and its Automaton This approach is illustrated by the grammar in Figure 7. This grammar does not distinguish several subcategories of expressions, such as `factor`, `term`, and `expr`: there is a single nonterminal symbol `expr`. This grammar is ambiguous. The precedence declarations on lines 7 and 8 indicate how to resolve the conflicts that appear when an LR(1) automaton for this grammar is constructed.

The LR(1) automaton that corresponds to this decorated grammar is shown in Figure 8. The vertices of this graph are the states of the automaton. Each state is identified by a number (in blue). Each state (except state 0, which is the initial state) also carries an *incoming symbol* (in red). This terminal or nonterminal symbol is in fact the label of every edge that enters this state. Edges labeled with a terminal symbol are shown as plain edges; edges labeled with a nonterminal symbol are dashed. Thus, starting in state 0, following the path labeled with `expr; ADD; expr` leads to state 8. State 8 is the only state where the automaton's behavior depends on the lookahead symbol, that is, on the first unconsumed token of the input stream. If this symbol is `MUL` then the automaton consumes this symbol and follows the transition from state 8 to state 5. This is known as "shifting". If this symbol is `ADD`, `RPAR`, or `EOL`, then the automaton does not consume this symbol, and reduces the production `expr: expr; ADD; expr`.[3] This means that the automaton moves back by three steps, thereby going back to state 0 or to state 1, depending on its history, which is stored in its stack. There, it follows a transition labeled with the left-hand side of this production, namely `expr`, thereby reaching state 10 or state 3. This is known as taking a "goto" transition.

For the purposes of this paper, it does not matter how precedence declarations are used to resolve conflicts, because our settlement algorithm (§3) relies on a description of the deterministic LR(1) automaton and does not care how this automaton is constructed. Nevertheless, let us briefly explain the effect of the precedence declarations in Figure 7.

The declaration `%left ADD` suggests that "addition is left-associative". Technically, this implies that in state 8, when the lookahead symbol is `ADD`, reduction must be preferred over shifting. Thus, because of the precedence declaration, a transition labeled `ADD` out of state 8, which could have existed, is suppressed. As a result, after reading `1 + 2`, if the next input symbols are `+ 3`, then the parser constructs an AST for the subexpression `1 + 2` and considers this subexpression as the first operand of the addition `_ + 3`.

Similarly, the declaration `%left MUL` implies that in state 6, when the lookahead symbol is `MUL`, reduction must be preferred over shifting. This causes the removal of a transition

[2] For example, at the time of writing, the precedence declarations found in the OCaml compiler's parser involve 36 distinct levels and resolve shift/reduce conflicts in 142 states (out of 2068 states).

[3] For readability, in Figure 8, we write "reduce" but do not indicate which production must be reduced. This automaton is so simple that, in each state, at most one production can be reduced. We let the reader reconstruct which one.

```
%token<int> INT                 (* Tokens *)
%token      ADD  "+"
%token      MUL  "*"
%token      LPAR "("
%token      RPAR ")"
%token      EOL
%left       ADD                 (* Priority levels: weakest to strongest *)
%left       MUL
%start<AST.main> main           (* Start symbol *)
%{ open AST %}
%%                              (* Productions and semantic actions *)
%inline op:
| ADD                           { BAdd }                [@name add]
| MUL                           { BMul }                [@name mul]

expr:
| LPAR; e = expr; RPAR          { e }                   [@name paren]
| i = INT                       { EConst i }            [@name const]
| e1 = expr; op = op; e2 = expr { EBinOp (e1, op, e2) } 

main:
| e = expr; EOL                 { e }                   [@name eol]
```

Figure 7. Non-stratified grammar: the grammar

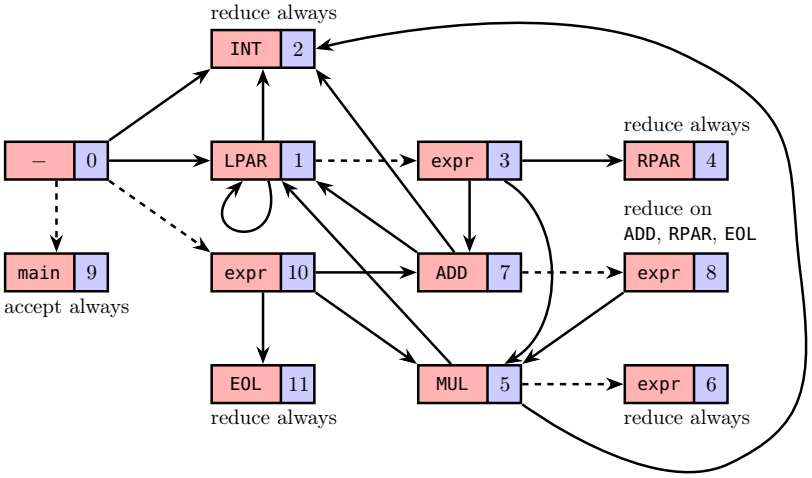
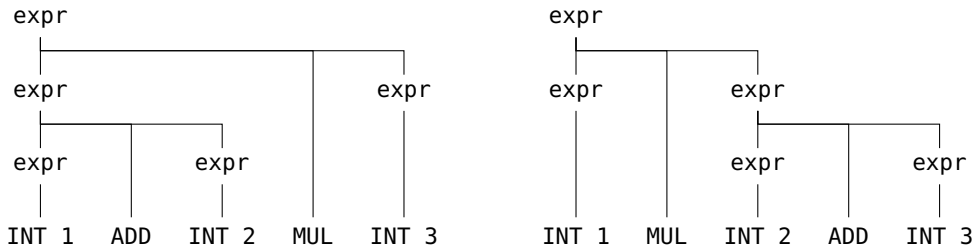


Figure 8. Non-stratified grammar: the LR(1) automaton



(a) A non-viable CST for `1+2*3` **(b)** A non-viable CST for `1*2+3`

Figure 9. Non-stratified grammar: two non-viable concrete syntax trees

```
0  module DCST : sig
1    type expr
2    type main
3    val expr_choice: expr -> expr -> expr         (* Constructors for [expr] *)
4    val paren: expr -> expr
5    val const: (int) -> expr
6    val add: expr -> expr -> expr
7    val mul: expr -> expr -> expr
8    val main_choice: main -> main -> main         (* Constructors for [main] *)
9    val eol: expr -> main
10 end
```

Figure 10. Non-stratified grammar: signature of the module `DCST`

```
1  let possibly_paren (e : DCST.expr) : DCST.expr =
2    DCST.expr_choice e (DCST.paren e)
3
4  let rec expr (e : AST.expr) : DCST.expr =
5    possibly_paren @@
6    match e with
7    | EConst i              -> DCST.const i
8    | EBinOp (e1, BAdd, e2) -> DCST.add (expr e1) (expr e2)
9    | EBinOp (e1, BMul, e2) -> DCST.mul (expr e1) (expr e2)
10
11 and main       : AST.main -> DCST.main = function
12   | e                     -> DCST.eol (expr e)
```

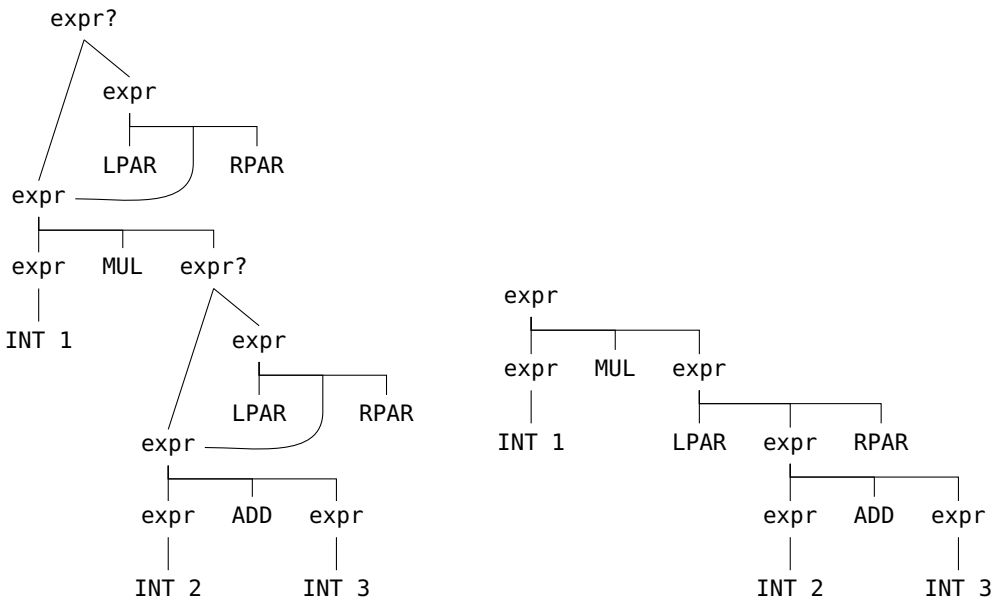
Figure 11. Non-stratified grammar: translating ASTs to DCSTs

labeled `MUL` out of state 6.

Finally, the order in which the two `%left` declarations appear suggests that "`MUL` takes precedence over `ADD`". Technically, this implies that in state 8, when the lookahead symbol is `MUL`, shifting must be preferred over reduction, and that in state 6, when the lookahead symbol is `ADD`, reduction must be preferred over shifting. This causes the removal of a reduction action in state 8 and the removal of a transition labeled `ADD` out of state 6. Thus, for example, the text `1+2*3` gives rise to the concrete syntax tree depicted in Figure 4b.

Translating ASTs to DCSTs As in the previous section (§1), for the grammar of Figure 7, Menhir generates a module `DCST`, whose interface is shown in Figure 10. By relying on this API, the user is expected to implement a transformation of ASTs to DCSTs. Such a transformation appears in Figure 11. The main point of interest resides in the use of `possibly_paren` at line 5. Using the constructor function `DCST.expr_choice`, this auxiliary function constructs a disjunction node whose children are `e` and `DCST.paren(e)`. Thus, it expresses the fact that there is a choice between displaying the subtree `e` without parentheses and displaying it surrounded with parentheses.

This code constructs DCSTs that are analogous to the one shown in Figure 12a. There, a binary disjunction node is identified by the label `expr?`. This DCST is a DAG: some internal nodes are reachable via multiple paths. Because this DCST involves two disjunction nodes, it can be viewed as a compact description of a family of four different CSTs. Some members of this family are viable CSTs; some are not. For instance, the concrete syntax tree shown in Figure 12b is a viable member of this family: it is obtained by choosing the left child of the outermost disjunction node and the right child of the innermost disjunction node. On the other hand, the concrete syntax tree in Figure 9b, which is obtained by choosing the



(a) A disjunctive concrete syntax tree **(b)** A viable concrete syntax tree

Figure 12. Non-stratified grammar: a DCST and one of the CSTs that it describes

left branch of both disjunction nodes, is a non-viable member of this family.

Let us say that a DCST is viable if at least one of the CSTs that it describes is viable. In our approach, it is up to the user to construct a viable DCST. The code in Figure 11 does this. Indeed, by using `possibly_paren` at every AST node, this code allows parentheses to be inserted at every node, and it is intuitively clear that if parentheses are inserted everywhere then a viable CST is obtained. If for some reason the user constructs a DCST that is not viable (for example, by forgetting to use `possibly_paren` at line 5) then the settlement algorithm (§3) fails. So, such a mistake is detected; however, it is detected only at runtime.

3 Settling Choices

The distinction between DCSTs and CSTs, and the settlement algorithm, which offers a bridge between these concepts, are the key technical contributions of this paper.

API The API of the settlement algorithm appears in Figure 14. The type of the function `Settle.main` reflects the purpose of this algorithm: it transforms a DCST whose head symbol is the start symbol `main` into some viable CST whose head symbol is also `main`. If it cannot do so, it reports a failure by returning `None`.

The API of the module `CST` is shown and discussed in the next section (§4). Let us point out that the type `CST.main` is abstract, so the user has no way of constructing CSTs besides invoking `Settle.main`. As a result, every value of type `CST.main` must be a *viable* CST.

Viability Relative to a Context Our discussion so far has relied on a simple abstract model of parsing, and our definition of viability has been based on this model. We have modeled a deterministic parser as a partial function *grow*: if s is a sequence of tokens then $grow(s)$ is either undefined or a CST. Then, we have written that a CST c is viable if and only if the round-trip property $grow(fringe(c)) = c$ holds. This models how a parser behaves

```
module CST : sig
  type expr
  type main
  class virtual ['r] reduce : object     (* This class is used while formatting (§4) *)
    method virtual zero : 'r             (* Document construction methods *)
    method virtual cat : 'r -> 'r -> 'r
    method virtual text : string -> 'r
    method virtual visit_INT : int -> 'r (* Visitor methods for terminal symbols *)
    method visit_ADD : 'r
    method visit_MUL : 'r
    method visit_LPAR : 'r
    method visit_RPAR : 'r
    method virtual visit_EOL : 'r
    method visit_expr : expr -> 'r       (* Visitor methods for nonterminal symbols *)
    method case_paren : expr -> 'r
    method case_const : int -> 'r
    method case_add : expr -> expr -> 'r
    method case_mul : expr -> expr -> 'r
    method visit_main : main -> 'r
    method case_eol : expr -> 'r
  end
end
```

Figure 13. Non-stratified grammar: signature of the module `CST`

```
module Settle : sig
  val main: DCST.main -> CST.main option
end
```

Figure 14. Non-stratified grammar: signature of the module `Settle`

when it is applied to a complete input. However, a less abstract and more compositional view is now needed: we must discuss how an LR(1) parser behaves when it is applied to an input fragment.

An LR(1) automaton maintains a *stack*, which records what input fragment has been consumed and reflects how this input fragment has been interpreted. A stack is a sequence of *cells*, where a cell is a pair of a state and a CST: $\sigma ::= \epsilon \mid \sigma \cdot (q, c)$. In every cell (q, c), the head symbol of the CST c is also the incoming symbol (§2) of the state q. We refer to it as *the cell symbol* of this cell. At every time, the stack σ forms a path in the automaton's state diagram. The stack σ determines the *current state* $cur(\sigma)$ of the automaton: it is the state found in the top stack cell. If the stack is empty, it is the initial state.

An LR(1) automaton also maintains a *remaining input*, a nonempty sequence of tokens. A pseudo-token **#** is used as a terminating sentinel. If κ is the first element of the remaining input then $sym(\kappa)$ is known as the *lookahead symbol*.

An LR(1) automaton makes progress via two kinds of actions. A *shift* action consumes a token κ and pushes the cell (q', c) onto the stack, where q' is the target of a transition labeled $sym(\kappa)$ out of the current state, and where c is the terminal node $T\ \kappa$. A *reduce* action for a production $A \to \alpha$ first pops several cells off the stack, whose sequence of cell symbols is α, and whose sequence of CSTs is $\vec{c}$. It then pushes the cell (q', c) onto the stack, where q' is the target of a transition labeled A out of the current state, and where c is the nonterminal node $N\ (A \to \alpha, \vec{c})$. At every step, the choice of the next action is determined solely by the current state and by the lookahead symbol.

The behavior of an LR(1) automaton, when faced with a certain input *segment*, depends

both on the input that the parser has read before reaching this segment, which is summarized by the automaton's current state, and on the lookahead symbol, that is, on the first input symbol that lies beyond this segment. Therefore, when looking at a CST c that is intended to correspond to a segment of the input, it does not make sense to ask whether this CST is "viable" without any qualification. Instead, one should ask whether c is viable *with respect to a certain state q and a certain lookahead symbol b*. This notion can be defined as follows:

Definition 1 (Relative Viability). A CST c is *viable* with respect to (q, b) iff for every stack σ such that $cur(\sigma) = q$ and for every nonempty sentence $\kappa \cdot s$ such that $sym(\kappa) = b$, the automaton, beginning with the stack σ and the remaining input $fringe(c) \cdot \kappa \cdot s$, reaches (in zero or more steps) the stack $\sigma \cdot (q', c)$ and the remaining input $\kappa \cdot s$.

In this definition, because the stack $\sigma \cdot (q', c)$ must correspond to a path in the automaton's state diagram, the state q' must be the target of a transition labeled $head(c)$ out of q. Therefore, for c to be viable with respect to (q, b), it is necessary that there exist a transition labeled $head(c)$ out of q.

In short, c is viable with respect to (q, b) if and only if the automaton, beginning in state q and faced with the fringe of c followed with the lookahead symbol b, consumes precisely this fringe and constructs precisely the tree c, which it pushes onto its stack.

Our earlier notion of "viability" without qualification corresponds to viability with respect to the automaton's initial state q_0 and sentinel symbol **#**.

Suppose c is a nonterminal node $N\ (A \to \alpha,\ \vec{c})$. If c is viable with respect to (q, b) then the following two facts must be true: (1) out of the state q, there exists a path, whose edge labels form the string α, to some state q'; (2) in state q', with lookahead symbol b, the automaton is willing to reduce the production $A \to \alpha$. When both conditions are satisfied, we say that $A \to \alpha$ is *apparently viable* with respect to (q, b). This is a necessary condition for c to be viable with respect to (q, b).

Compositional Specification of a Settlement Algorithm A compositional settlement algorithm solves *problems* of the form (d, q, b) where d is a DCST, q is a state, and b is a terminal symbol. A *solution* to such a problem is a CST c such that $c \in d$ and c is viable with respect to (q, b). A settlement algorithm can also return $\bot$, which represents a failure. Because a solution can exist only if the state q has an outgoing transition labeled $head(d)$, we restrict our attention to problems that satisfy this condition. A *complete* algorithm fails only when no solution exists; an *incomplete* algorithm can fail even if a solution exists.

An Eager Settlement Algorithm A linear-time, incomplete settlement algorithm *settle* can be described as follows. Assume the problem is (d, q, b), where q has an outgoing transition labeled $head(d)$. Then, compute $settle(d, q, b)$ as follows:

1. If d is a terminal node $T\ \kappa$, return this terminal node.
2. If d is a nonterminal node $N\ (A \to \alpha,\ \vec{d'}\,)$,
   a) Verify that $A \to \alpha$ is apparently viable with respect to (q, b). If not, fail.
   b) Settle each of the subtrees $\vec{d'}$, *from right to left*, via recursive calls to *settle*, yielding a sequence of CSTs $\vec{c}'$. Then, return the nonterminal node $N\ (A \to \alpha,\ \vec{c}'\,)$.
3. If d is a disjunction node $d_1 \vee d_2$, where d_1 is a nonterminal node $N\ (A \to \alpha,\ \vec{d'}\,)$,
   a) If $A \to \alpha$ is apparently viable with respect to (q, b) then return $settle(d_1, q, b)$;
   b) otherwise return $settle(d_2, q, b)$.

More explanations about step 2b are needed. The check in step 2a guarantees that there exists a path out of the state q whose edge labels form the string α. This is also the string of the head symbols of the subtrees $\vec{d'}$. The source states of the edges along this path are the states that must be used in the recursive calls to *settle*. A key remark is that these states are known before any of the recursive calls takes place. So, there is no requirement that the recursive calls be performed from left to right. We perform them from

right to left, which is crucial, because the result of settling a subtree is needed to compute the lookahead symbol that must be used while settling the next subtree towards the left. Formally, if the sequence $\vec{d'}$ is decomposed as $\vec{d'_1} \cdot d' \cdot \vec{d'_2}$, and if settling $\vec{d'_2}$ has produced $\vec{c}'_2$, then the lookahead symbol that must be passed as an argument to the next recursive call, $\mathit{settle}(d', -, -)$, is the first element of the nonempty sequence $\mathit{sym}(\mathit{fringe}(\vec{c}'_2)) \cdot b$.

In step 3, we assume that the left disjunct d_1 is a nonterminal node. This causes no loss of generality. Indeed, a disjunction node whose disjuncts are terminal nodes is pointless; and a disjunction node whose left-hand child is itself a disjunction can be re-associated.

This algorithm is correct: assuming that q has an outgoing transition labeled $\mathit{head}(d)$, if $\mathit{settle}(d, q, b)$ returns c then $c \in d$ holds and c is viable with respect to (q, b).

This algorithm runs in time $O(|d|)$, that is, in linear time in the tree size of the DCST d. In steps 3a and 3b, it is crucial that *settle* is recursively applied to d_1 or d_2, but not both. Thus, even if d_1 and d_2 share a subtree, this subtree is examined only once by the algorithm. The tests in steps 2a and 3a can be carried out in constant time, provided certain tables are precomputed. In step 2b, in order to efficiently compute the lookahead symbol that must be passed to each recursive call, it is convenient to adopt the convention that $\mathit{settle}(d, q, b)$ returns not only a CST c, but also the first symbol of the sequence $\mathit{sym}(\mathit{fringe}(c)) \cdot b$.

This algorithm is unfortunately incomplete. The source of incompleteness lies in step 3a. The apparent viability condition in step 3a is necessary for the recursive call $\mathit{settle}(d_1, q, b)$ to succeed, but does not guarantee that this call will succeed. If this call fails, then $\mathit{settle}(d_1 \vee \vec{d_2}, q, b)$ fails as well, even though the subproblem (d_2, q, b) may have a solution. In other words, the algorithm is incomplete because it commits to the left-hand disjunct as soon as the apparent viability test succeeds, even though apparent viability does not imply viability. Completeness can be obtained by adding a backtracking mechanism: in step 3a, if $\mathit{settle}(d_1, q, b)$ fails, then return $\mathit{settle}(d_2, q, b)$. With this modification, the algorithm is complete, but has exponential time complexity, because d_1 and d_2 can (and usually do) share subtrees (recall Figure 12a).

A Memoizing Settlement Algorithm A complete settlement algorithm whose worst-case time complexity is polynomial can be given. This algorithm decomposes the original problem into subproblems of the form (d, q, a, b). A solution of such a subproblem is a CST c such that $c \in d$ and c is viable with respect to (q, b) and the first symbol of $\mathit{fringe}(c) \cdot b$ is a. Memoization ensures that each subproblem is considered at most once. Compared with the eager algorithm, this algorithm requires a more complex internal representation of DCSTs: every DCST node must carry a unique identifier, and its "nullable" flag and "first" set must be precomputed. An apparent viability test is not needed, but can be used to speed up this algorithm.

Because this algorithm spends constant time and constant space on each subproblem, its time and space complexity is the number of subproblems that can ever be considered. Thus, in the worst case, it is $O(\#d.|Q|.|\Sigma|^2)$, where $\#d$ is the DAG size of the DCST d, $|Q|$ is the number of states of the LR(1) automaton, and $|\Sigma|$ is the number of terminal symbols. In practice, we expect that for each node d' in the DAG d, the number of subproblems (d', q, a, b) that are ever considered is independent of the size of the alphabet and of the size of the automaton. We have experimentally confirmed this expectation: for the grammar of Figure 7, we find that in practice this number is at most 3. Therefore, the time and space complexity of this algorithm is effectively linear in $\#d$.

Comparing Settlement Algorithms We have implemented both algorithms and performed a limited evaluation of their performance, based on the grammar of Figure 7 and on randomly-generated arithmetic expressions. As expected, both algorithms exhibit linear time complexity. The eager algorithm appears to be much faster in practice: taking into account both DCST construction time and settling time, the eager algorithm outperforms the memoizing algorithm by a factor of 15. We believe that the high cost of the memoizing

algorithm is due, to a large extent, to memoization itself: even if the cost of solving subproblems is discounted, the cost of storing and fetching data in and out of a large hash table seems very high.

Choosing an Algorithm In summary, the eager algorithm is fastest, but incomplete; the memoizing algorithm is complete, but significantly slower; and the eager algorithm with backtracking is complete and usually fast, but can be exponentially slow in pathological cases. At the time of writing, Menhir offers the eager algorithm only. This could change in the future, based on user feedback.

To what extent could the incompleteness of the eager algorithm be a problem in practice? We do not know yet. For the grammar of Figure 7, it is not a problem: indeed, for this simple grammar, apparent viability implies viability. However, this implication is not obvious. Furthermore, we have constructed more complex grammars where this is not the case. More experience is required to tell whether users are likely to face difficulties caused by incompleteness.

The eager algorithm is not optimal: it does not necessarily produce the smallest viable CST c such that $c \in d$. Instead, in every disjunction node, it favors the left disjunct. The code in Figure 11 relies on this behavior: at line 2, the left disjunct proposes not to insert parentheses, while the right disjunct inserts parentheses. Thus, in this simple example, optimality is achieved anyway. Regarding the memoizing algorithm, we believe that there are two variants of it: a left-biased variant, which we have implemented, and an optimal variant, which we have not implemented, and which we imagine will be slightly more expensive.

4 Formatting and Rendering

Once a (viable) CST has been constructed, there remains to transform it into a printable form, such as string or a `PPrint` document. This corresponds to steps 6 and 7 in Figure 1.

To allow this, we could expose an API that allows deconstructing CSTs, and let the user implement this transformation entirely on her own. We could for instance present the types `CST.expr` and `CST.main` as *private* algebraic data types, that is, algebraic data types whose values can be inspected via **`match`** constructs, but cannot be constructed. However, this approach would have two drawbacks. First, it would leave to the user the rather dreary task of implementing step 6, whereas one might expect this task to be performed by generated code. Second, it would force us to expose a grammar-specific representation of CSTs in memory, thereby preventing us from internally using a different, grammar-independent representation, or forcing us to transform one representation into the other.

Visitors to the Rescue Our solution to these problems is to present `CST.expr` and `CST.main` as *abstract types* and to generate code for step 6 in such a form that the behavior of this code can be customized. Indeed, not everything about step 6 is mechanical. For one thing, we wish to let the user pick the return type of this transformation: strings, `PPrint` documents, or some other type of her choosing. Furthermore, should the user decide to produce `PPrint` documents, we wish to let her customize the layout of these documents: she must decide where and how to use the document-construction combinators offered by the `PPrint` library, which offer fine-grained control over indentation, line breaks, and so on. To allow customization, we generate a *visitor class* whose methods reduce a CST to a printable representation. For the grammar of Figure 7, this API is shown in Figure 13.

The module `CST`, whose code is generated by Menhir, advertises `expr` and `main` as abstract types. Then, it advertises a visitor class, `reduce`. This class is parameterized by a type `'r`, which is the result type of every method: it is the type of the printable form to which every tree must be reduced. The user is expected to provide implementations for the three virtual methods `zero`, `cat`, and `text`. In short, `zero` is the empty printable thing; `cat` concatenates

```
class print = object
  inherit [string] CST.reduce
  method zero = ""
  method cat = (^)
  method text s = s
  method visit_INT i = Printf.sprintf "%d" i
  method visit_EOL = "\n"
end

let main (m : CST.main) : string =
  (new print)#visit_main m
```

Figure 15. Non-stratified grammar: translating CSTs to strings

```
open PPrint
let lparen = lparen ^^ ifflat empty space
let rparen = ifflat empty hardline ^^ rparen
class print = object (self)
  inherit [document] CST.reduce as super
  method zero = empty
  method cat = (^^)
  method text = string
  method visit_INT i = utf8format "%d" i
  method! visit_ADD = space ^^ plus ^^ break 1
  method! visit_MUL = space ^^ star ^^ break 1
  method visit_EOL = hardline
  method! visit_expr e = group (super#visit_expr e)
  method! case_paren e = nest 2 (lparen ^^ self#visit_expr e) ^^ rparen
end

let main (m : CST.main) : document =
  (new print)#visit_main m
```

Figure 16. Non-stratified grammar: translating CSTs to `PPrint` documents

```
# Printing as a string:
65*((22+38)*69+(24+58))+(84*70+(20+63*83*97+49*(70+0))*(93+89)*(12*15+85+21))
# Printing via PPrint in 20 columns:
65 *
( (22 + 38) * 69 +
  (24 + 58)
) +
( 84 * 70 +
  ( 20 +
    63 * 83 * 97 +
    49 * (70 + 0)
  ) *
  (93 + 89) *
  ( 12 * 15 + 85 +
    21
  )
)
```

Figure 17. Sample output produced by the code in Figures 15 and 16

two printable things; and `text` converts a string to a printable thing. Furthermore, the following methods exist:

- For each terminal symbol, there is a visitor method: these include `visit_INT`, `visit_ADD`, and so on. If the terminal symbol carries no semantic value and if a "token alias" (a concrete string) has been provided by the user for this symbol (Figure 7) then a default implementation of this method is generated by Menhir; this default implementation relies on the method `text`. Otherwise, this method is declared virtual.
- For each nonterminal symbol, there is a visitor method: these are `visit_expr` and `visit_main`. These methods expect a CST as an argument, perform a case analysis of the root node, and apply a suitable `case` method to the children of the root node.
- For each production, there is a visitor method: these include `case_paren`, `case_const`, and so on. These methods expect zero, one, or more arguments. First, each argument is reduced to a printable thing via recursive calls to suitable `visit` methods; then these printable things are concatenated using `zero` and `cat`.

Examples of Use Figure 15 shows how a user may rely on the class `CST.reduce` to convert CSTs to strings. The user defines a new class, `print`, which inherits `reduce` where the type parameter `'r` is instantiated with `string`. Then, the user supplies trivial implementations of the methods `zero`, `cat`, and `text`: the empty string, string concatenation, and the identity function. Finally, the user provides implementations of the virtual methods `visit_INT` and `visit_EOL`.[4] Then, the method call `(new print)#visit_main` converts a CST to a string.[5]

Figure 16 shows how a user may rely on the class `CST.reduce` to convert CSTs to `PPrint` documents. The idea is the same. This time, the type parameter `'r` is instantiated with the type `PPrint.document`. The document construction combinators offered by `PPrint` are used in a few key places. For example, by overriding the method `visit_expr`, the user indicates that a `group` combinator, which offers a choice between flat and non-flat output, should be used at every subexpression. By overriding the method `case_paren`, the user customizes the manner in which parenthesized subexpressions are indented. An example of the output produced by this code appears in Figure 17.

5 Related Work

van den Brand and Visser [vdBV96] describe a tool that generates "formatters" (that is, unparsers and pretty-printers) for ASF+SDF grammars. An ASF+SDF grammar is a context-free grammar, augmented with priority and associativity declarations. It does not have semantic actions, but a production can be marked with the attribute `bracket`, which means that no AST node must be produced. Thus, an unparser must decide where to insert parentheses. van den Brand and Visser spell out the round-trip property that a correct unparser must obey. They provide a brief description of a (linear-time, bottom-up) unparser of arithmetic expressions, but do not describe how they deal with the general case. They separate unparsing and layout in the same way as we do (§3, §4).

Ramsey [Ram98] offers a detailed presentation of a (linear-time, bottom-up) unparser. The design and correctness proof of his algorithm are guided by thinking about the behavior of a shift-reduce parser. This is also the case in this paper: in fact, at runtime, our settlement algorithm relies directly on the parser's tables. Ramsey supports a single syntactic category of

[4] Because Menhir does not allow a token alias to contain a newline character, the token alias `"\n"` cannot be associated with the symbol `EOL`.

[5] This example happens to "just work" because two consecutive tokens can be safely concatenated: for this simple language of arithmetic expressions, *in a grammatically valid sentence*, there is no possibility for the concatenation of two tokens to be confused by the lexer with some other token. In a more complex scenario, greater care would be required. The type of printable things would be more elaborate than `string`, and the `cat` method would sometimes insert a space between two things.

"expressions" with infix, prefix, and postfix operators whose precedence and associativity are specified by the user. We deal with the more general case of an arbitrary LR(1) automaton, yet our eager settlement algorithm is arguably simpler than Ramsey's unparser. Our eager algorithm is incomplete in general, but we believe that it is complete in the restricted setting considered by Ramsey. We sketch (and have implemented) a slower, complete algorithm.

Bouwers, Bravenboer, and Visser [BBV08] regret that the style of precedence declarations supported by most LR(1) parser generators, including `yacc`, `bison`, and Menhir, does not have a clear semantics. To clarify this semantics, and to help compare two parsers that are based on different methodologies, they propose "a core formalism for defining precedence rules", based on "tree patterns" of bounded depth. They describe a "precedence rule recovery tool", which tests whether a parser, viewed as a black box, can or cannot construct certain tree patterns. Thanks to this tool, they discover discrepancies between several parsers for C99 and PHP5. The question that their tool answers, "can this parser possibly construct a tree of this shape?", is closely related to the question that our settlement algorithm addresses. That said, we believe that the answer to such a question can depend on the current state of the LR(1) automaton: that is, it can be the case that in a certain state q the parser will construct a tree of a certain shape whereas in some other state q' it will refuse to construct a tree of this shape. Another source of worry is Bravenboer *et al.*'s remark that tree patterns of depth 1 are often sufficient, yet depths 2 and 3 are sometimes needed. Thus, it is not clear whether (or under what conditions) Bravenboer *et al.*'s tree pattern formalism is capable of accurately describing the behavior of an LR(1) parser. This open question could be related to the open question of clarifying under what conditions "apparent viability" (§3) implies viability. Indeed, apparent viability is an inspection of the tree at depth 1.

Danielsson [Dan13] discusses the construction of pretty-printers in a dependently-typed programming language. He uses dependent types to express and enforce the end-to-end round-trip property $\mathit{parse}(\mathit{unparse}(t)) = t$. In contrast, we guarantee the more limited property $\mathit{grow}(\mathit{fringe}(c)) = c$, and we rely on (generated) abstract types to ensure that the user can construct only grammatically valid DCSTs and CSTs (see Figures 5 and 6).

Several authors have investigated the idea of producing parsers and pretty-printers out of a single "invertible syntax description" [RO10, ANO12, MW13, ZZK+16, ZKZ+20]. This approach promises to reduce redundancy and guarantee that a round-trip property holds by construction. Our starting point in this paper is an existing parser generator that allows arbitrary semantic actions, so we cannot follow this approach.

6 Conclusion

We have described several new features offered by Menhir to support fast and correct unparsing. Menhir now generates type-safe, grammar-specific DCST construction and CST deconstruction APIs. A settlement algorithm, which transforms DCSTs into CSTs, is implemented once and for all in a library. Because it relies directly on a description of the LR(1) automaton, it supports arbitrary precedence declarations and all LR(1) construction methods. The concept of a DCST, as well as our settlement algorithms, are new.

The most unpleasant aspect of our approach is perhaps the need for the user to manually implement the translation of ASTs to (viable) DCSTs. In the future, it would be desirable to automate this transformation; this should be possible, provided the expressive power of semantic actions is restricted. Another limitation is the incompleteness of our fast settlement algorithm. Whether it is a problem in practice remains to be determined: we do not yet have practical experience with this new tool. In the future, we would like to allow decorating trees with user-specified data, such as comments or source code locations, so as to allow "high-fidelity" program transformations [dJV11, AvdS20].

References

[AJU75] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, 1975.

[ANO12] Reynald Affeldt, David Nowak, and Yutaka Oiwa. Formal network packet processing with minimal fuss: invertible syntax descriptions at work. In *Programming Languages Meets Program Verification (PLPV)*, pages 27–36, 2012.

[AvdS20] Rodin T. A. Aarssen and Tijs van der Storm. High-fidelity metaprogramming with separator syntax trees. In *Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 27–37, January 2020.

[BBV08] Eric Bouwers, Martin Bravenboer, and Eelco Visser. Grammar engineering support for precedence rule recovery and compatibility checking. *Electronic Notes in Theoretical Computer Science*, 203(2):85–101, 2008.

[Dan13] Nils Anders Danielsson. Correct-by-construction pretty-printing. In *Workshop on dependently-typed programming*, pages 1–12, September 2013.

[dJV11] Maartje de Jonge and Eelco Visser. An algorithm for layout preservation in refactoring transformations. In *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 40–59. Springer, July 2011.

[GJ08] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques: a practical guide, second edition*. Monographs in computer science. Springer, 2008.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Computing Science Technical Report 32, Bell Laboratories, 1975.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information & Control*, 8(6):607–639, December 1965.

[KVW10] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 918—-932, 2010.

[MW13] Kazutaka Matsuda and Meng Wang. FliPpr: A prettier invertible printing system. In *European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 101–120. Springer, March 2013.

[Pot23] François Pottier. The PPrint pretty-printing library, 2007–2023. `https://github.com/fpottier/pprint/`.

[PRG23] François Pottier and Yann Régis-Gianas. The Menhir parser generator, 2005–2023. `https://gitlab.inria.fr/fpottier/menhir/`.

[Ram98] Norman Ramsey. Unparsing expressions with prefix and postfix operators. *Software: Practice and Experience*, 28(12):1327–1356, 1998.

[RO10] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *Symposium on Haskell*, pages 1–12, September 2010.

[vdBV96] Mark van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996.

[ZKZ+20] Zirun Zhu, Hsiang-Shang Ko, Yongzhe Zhang, Pedro Martins, João Saraiva, and Zhenjiang Hu. Unifying parsing and reflective printing for fully disambiguated grammars. *New Generation Computing*, 38(3):423–476, 2020.

[ZZK+16] Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. Parsing and reflective printing, bidirectionally. In *Software Language Engineering*, pages 2–14, November 2016.