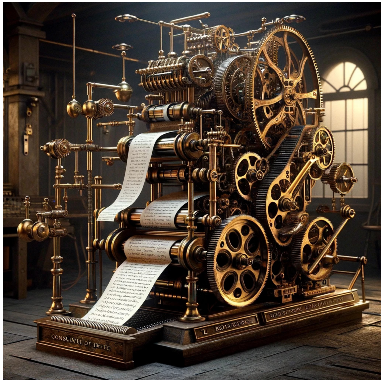
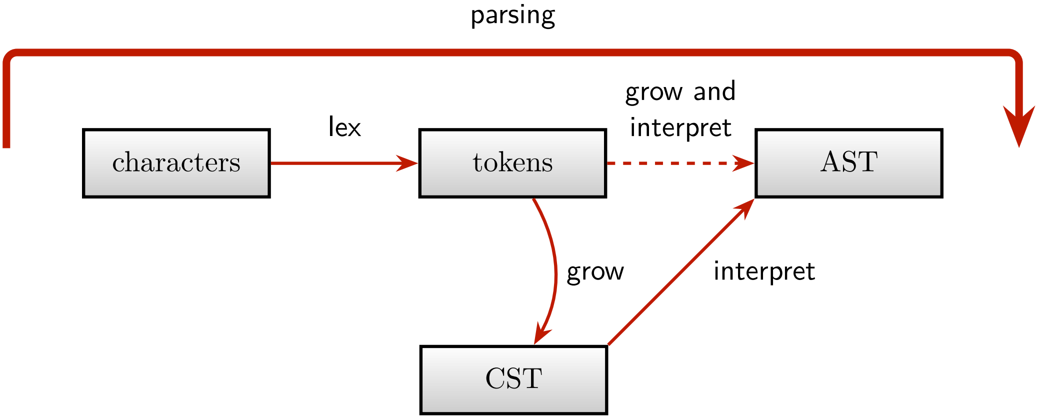


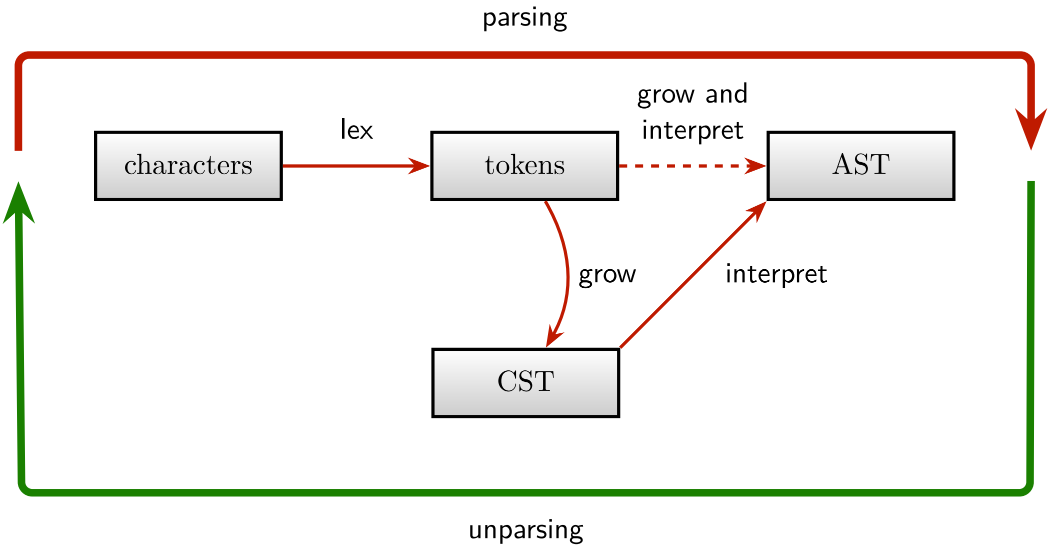
# Correct, Fast LR(1) Unparsing

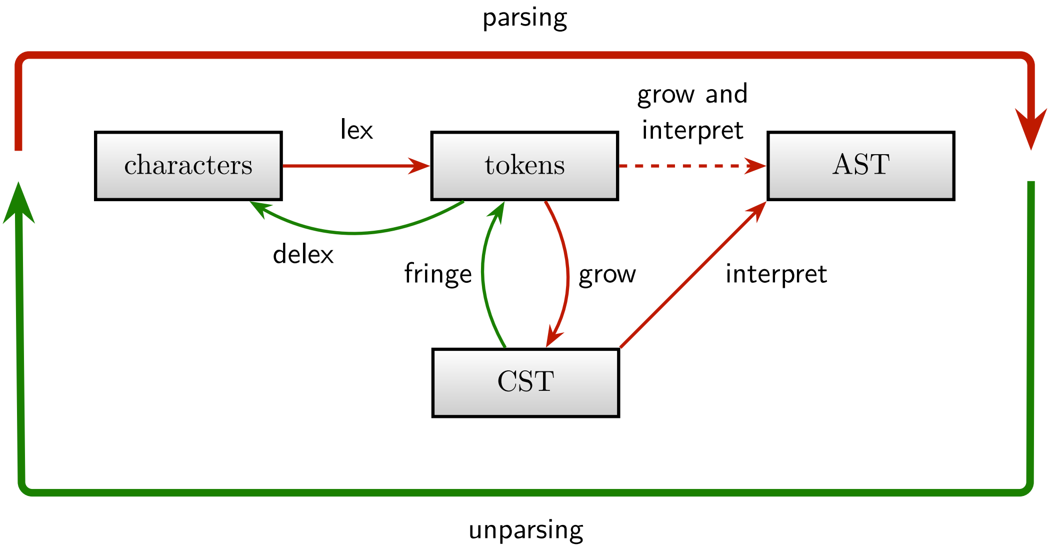

François Pottier

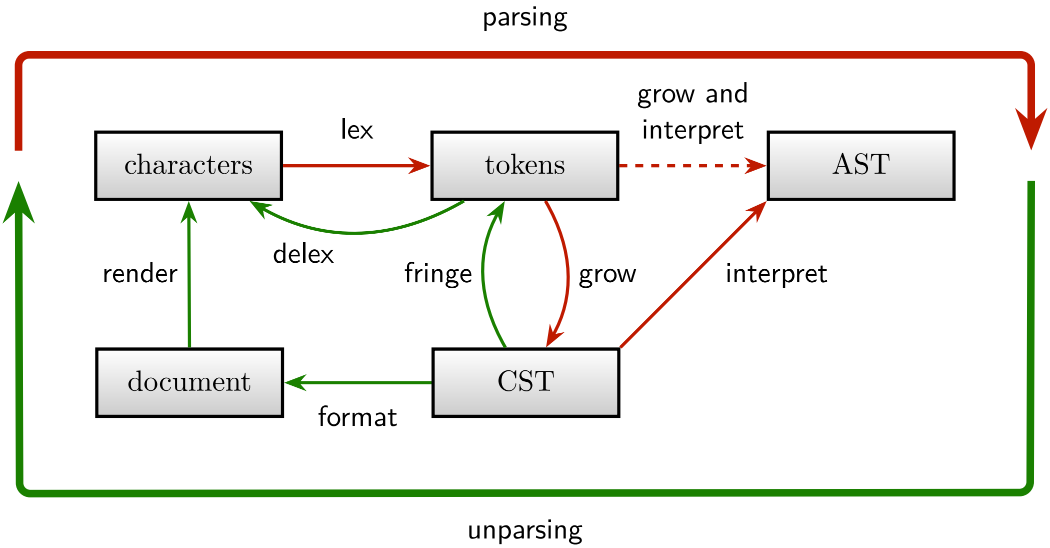


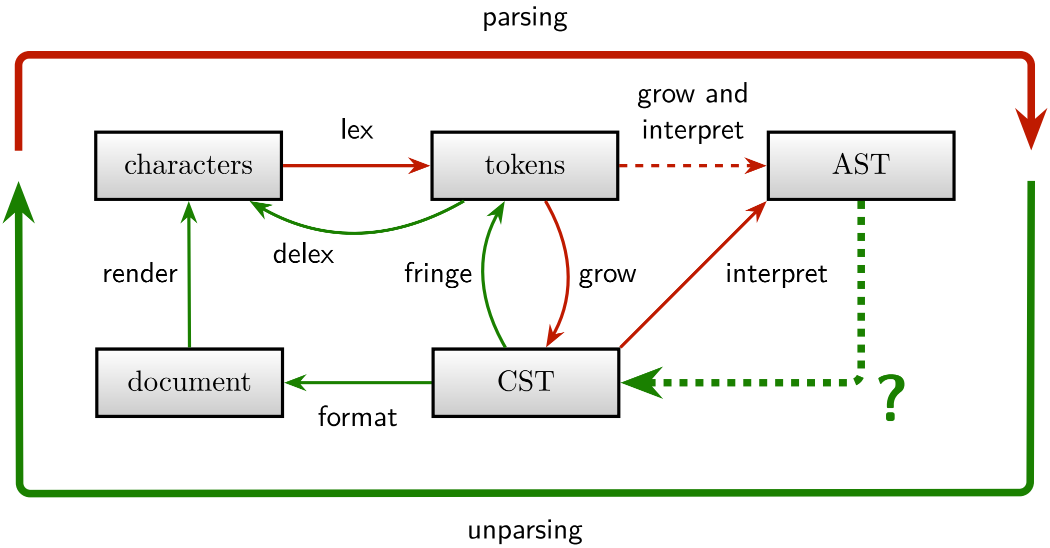
JFLA 2024


parsing
lex
grow and interpret
characters
tokens
AST
grow
interpret
CST


parsing
characters
lex
tokens
grow and
interpret
AST
grow
interpret
CST
unparsing


parsing
characters
lex
tokens
grow and
interpret
AST
delex
fringe
grow
interpret
CST
unparsing


parsing
grow and
interpret
lex
characters
tokens
AST
delex
render
fringe
grow
interpret
document
CST
format
unparsing


parsing
grow and interpret
lex
characters
tokens
AST
delex
render
fringe
grow
interpret
document
CST
?
format
unparsing

Can we **automatically generate** a translation of ASTs to CSTs?

## From ASTs Back to CSTs

Can we **automatically generate** a translation of ASTs to CSTs?

- **No!** Semantic actions are arbitrary OCaml code, so cannot (in general) be inverted.

Can we **let the user** write a translation of ASTs to CSTs?

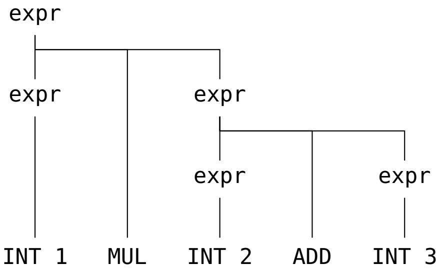
Can we **automatically generate** a translation of ASTs to CSTs?

- **No!** Semantic actions are arbitrary OCaml code, so cannot (in general) be inverted.

Can we **let the user** write a translation of ASTs to CSTs?

- **No!** Some CSTs are **not viable** and must be avoided.

# Not All CSTs Are Viable

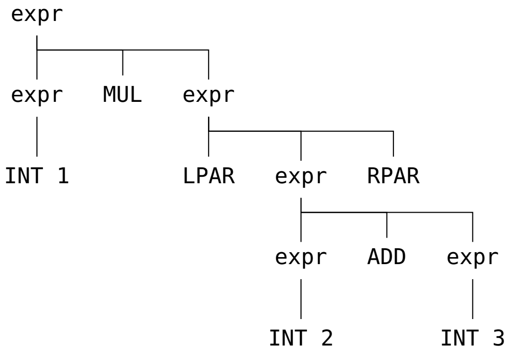



This CST is **not viable**: it does not satisfy $\mathit{grow}(\mathit{fringe}(c)) = c$.

In other words, parsing `1*2+3` does not produce this tree.

In other words, the parser cannot construct this tree.

One should **never** attempt to print this tree!




Here is a **viable** CST whose fringe is `1*(2+3)`.

It represents **the same AST** as the previous non-viable tree.

This is the CST that we wish to print! **Parentheses** are necessary in this example.

Can we **automatically generate** a translation of ASTs to CSTs?

Can we **automatically generate** a translation of ASTs to CSTs?

- No.

Can we **let the user** write a translation of ASTs to CSTs?

Can we **automatically generate** a translation of ASTs to CSTs?

- No.

Can we **let the user** write a translation of ASTs to CSTs?

- No. Guaranteeing that a viable tree is obtained
  can be **difficult** and **error-prone**.
  Maintaining this guarantee as the parser evolves
  seems difficult as well.

## From ASTs Back to CSTs

To escape this conundrum, we propose to **split** this step:

- **let the user** translate an AST to (a description of) a **set** of possible CSTs;
- **generate and/or provide** an algorithm that selects a viable CST among this set.

To escape this conundrum, we propose to **split** this step:

- **let the user** translate an AST to (a description of) a **set** of possible CSTs;
- **generate and/or provide** an algorithm that selects a viable CST among this set.

Thus,

- the user deals with the problem of **inverting the semantic actions**;
- the user indicates **where parentheses may be inserted**;
- the tool decides **where to actually insert parentheses**.

# From ASTs Back to CSTs

To escape this conundrum, we propose to **split** this step:

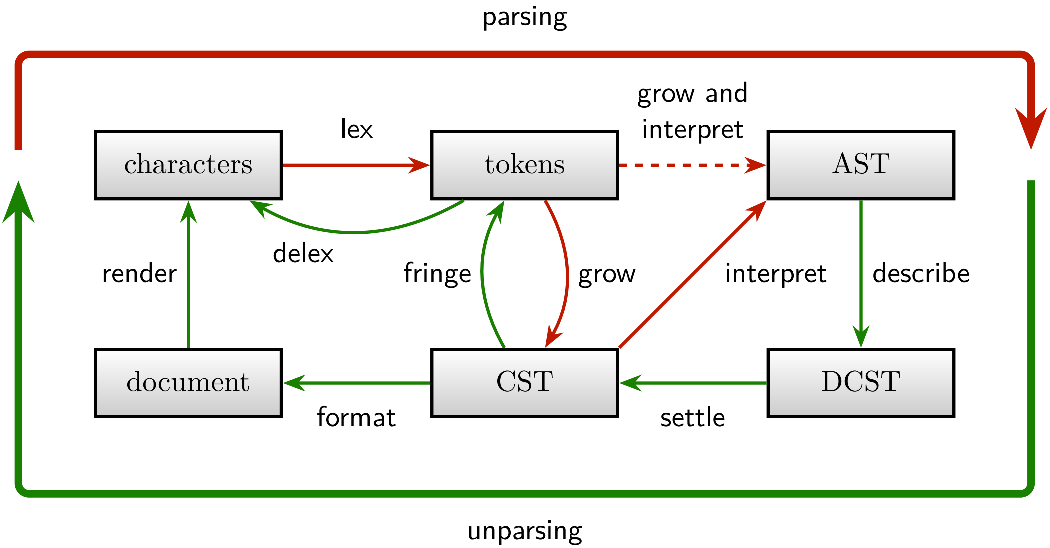
- **let the user** translate an AST to (a description of) a **set** of possible CSTs;
- **generate and/or provide** an algorithm that selects a viable CST among this set.

Thus,

- the user deals with the problem of **inverting the semantic actions**;
- the user indicates **where parentheses may be inserted**;
- the tool decides **where to actually insert parentheses**.

A DCST resembles a CST but can contain binary **disjunction** nodes. It is usually a **DAG**.

# The Big Picture



Menhir can now:

- generate **abstract types of DCSTs**
  and **a DCST construction API**
  so the user can translate ASTs to DCSTs.
- generate **abstract types of CSTs**
  and **a CST deconstruction API**
  so the user can translate CSTs to documents or strings.
- provide a **translation of DCSTs to CSTs**
  whose correctness is guaranteed,
  even if the grammar has conflicts and uses `%left`, `%right`, `%nonassoc`, `%prec`.

Only **viable CSTs** can ever be constructed.

**Two** DCST-to-CST translations have been implemented:

- one is fast but **incomplete**: in certain (unlikely?) situations, it can fail to find a viable CST even though there exists one.
- the other is complete but can be 15x **slower**, due to memoization.

**Two** DCST-to-CST translations have been implemented:

- one is fast but **incomplete**: in certain (unlikely?) situations, it can fail to find a viable CST even though there exists one.
- the other is complete but can be 15x **slower**, due to memoization.

This new facility has **no known users** yet...

# How Unparsing Is Used, and How It Works

Here are **abstract syntax trees** for arithmetic expressions:

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

AST.ml

As usual, the tokens are defined first:

```
%token<int> INT                  (* Tokens *)
%token      ADD  "+"
%token      MUL  "*"
%token      LPAR "("
%token      RPAR ")"
%token      EOL
```

parser.mly

Then, **precedence declarations** are provided:

```
%left       ADD                    (* Priority levels: weakest to strongest *)
%left       MUL
```

parser.mly

Then, an **unstratified** syntax of expressions is given:

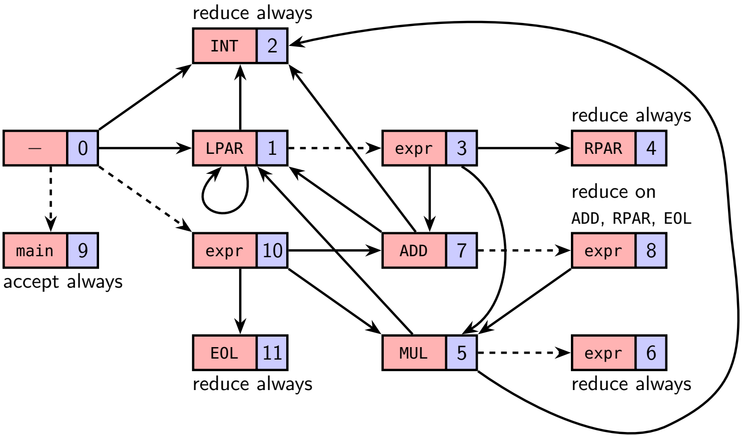
```
%inline op:
| ADD                              { BAdd }                [@name add]
| MUL                              { BMul }                [@name mul]

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]
```

parser.mly

The `[@name]` attributes influence the generated CST and DCST APIs.




A shift/reduce conflict on `MUL` in state 8 is resolved in favor of shifting.

A shift/reduce conflict on `ADD` in state 6 is resolved in favor of reduction.

The generated parser contains this submodule:

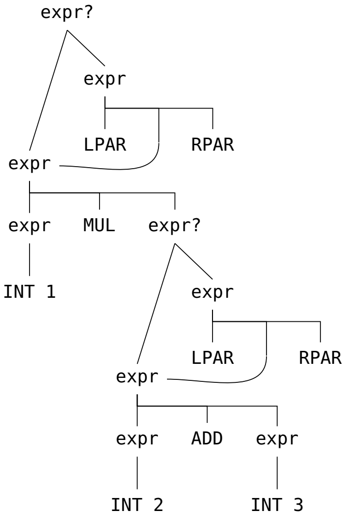
```
module DCST : sig
  type expr
  type main
  val expr_choice: expr -> expr -> expr        (* Constructors for [expr] *)
  val paren: expr -> expr
  val const: (int) -> expr
  val add: expr -> expr -> expr
  val mul: expr -> expr -> expr
  val main_choice: main -> main -> main        (* Constructors for [main] *)
  val eol: expr -> main
end                                                                parser.mli
```

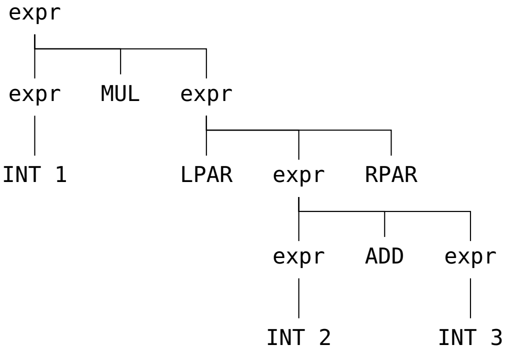
The user exploits the DCST construction API as follows:

```
let possibly_paren (e : DCST.expr) : DCST.expr =
  DCST.expr_choice e (DCST.paren e)            (* [e] is shared: a DAG is built *)

let rec expr (e : AST.expr) : DCST.expr =
  possibly_paren @@                  (* at every node, parentheses may be inserted *)
  match e with
  | EConst i             -> DCST.const i
  | EBinOp (e1, BAdd, e2) -> DCST.add (expr e1) (expr e2)
  | EBinOp (e1, BMul, e2) -> DCST.mul (expr e1) (expr e2)

and main       : AST.main -> DCST.main = function
  | e                    -> DCST.eol (expr e)                          AST2DCST.ml
```


expr?
expr
LPAR
RPAR
expr
expr
MUL
expr?
INT 1
expr
LPAR
RPAR
expr
expr
ADD
expr
INT 2
INT 3


expr
expr
MUL
expr
INT 1
LPAR
expr
RPAR
expr
ADD
expr
INT 2
INT 3

The generated parser contains this submodule:

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

parser.mli

## DCST to CST Conversion: Key Insights

Suppose you have access to the parse tables.

To **check** that a CST is viable, **run the parser** on its fringe.
Verify that the parser succeeds and produces this tree.

# DCST to CST Conversion: Key Insights

Suppose you have access to the parse tables.

To **check** that a CST is viable, **run the parser** on its fringe.
Verify that the parser succeeds and produces this tree.

- In reality, viability depends on the parser's state and on the lookahead symbol.

# DCST to CST Conversion: Key Insights

Suppose you have access to the parse tables.

To **check** that a CST is viable, **run the parser** on its fringe.
Verify that the parser succeeds and produces this tree.

- In reality, viability depends on the parser's state and on the lookahead symbol.

To **transform** a DCST into a viable CST, **run the parser** on its fringe.
At disjunction nodes, choose a viable child:

Suppose you have access to the parse tables.

To **check** that a CST is viable, **run the parser** on its fringe.
Verify that the parser succeeds and produces this tree.

- In reality, viability depends on the parser's state and on the lookahead symbol.

To **transform** a DCST into a viable CST, **run the parser** on its fringe.
At disjunction nodes, choose a viable child:

- by trying both children and **backtracking** (complete; **exponentially slow**), or

Suppose you have access to the parse tables.

To **check** that a CST is viable, **run the parser** on its fringe.
Verify that the parser succeeds and produces this tree.

- In reality, viability depends on the parser's state and on the lookahead symbol.

To **transform** a DCST into a viable CST, **run the parser** on its fringe.
At disjunction nodes, choose a viable child:

- by trying both children and **backtracking** (complete; **exponentially slow**), or
- by trying both children and **memoizing** shared subgoals (complete; **slow**), or

## DCST to CST Conversion: Key Insights

Suppose you have access to the parse tables.

To **check** that a CST is viable, **run the parser** on its fringe.
Verify that the parser succeeds and produces this tree.

- In reality, viability depends on the parser's state and on the lookahead symbol.

To **transform** a DCST into a viable CST, **run the parser** on its fringe.
At disjunction nodes, choose a viable child:

- by trying both children and **backtracking** (complete; **exponentially slow**), or
- by trying both children and **memoizing** shared subgoals (complete; **slow**), or
- by **committing** to the first child if it seems **apparently viable** (**incomplete**; fast).

The generated parser contains this submodule:

```
module CST : sig
  type expr
  type main
  class virtual ['r] reduce : object
    method virtual zero : 'r             (* Document construction methods *)
    method virtual cat : 'r -> 'r -> 'r
    method virtual text : string -> 'r
    method visit_expr : expr -> 'r       (* Visitor methods *)
    method case_paren : expr -> 'r
    method case_add : expr -> expr -> 'r
    method case_mul : expr -> expr -> 'r
      (* ... more visitor methods and case methods ... *)
  end
end                                                                        parser.mli
```

The user instantiates (just) the virtual methods:

```
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
```

CST2String.ml

This code makes no decisions regarding parenthesization. It is **just a printer**.

This kind of output is produced:

```
65*((22+38)*69+(24+58))+(84*70+(20+63*83*97+49*(70+0))*(93+89)*(12*15+85+21))
```

The user instantiates the virtual methods and overrides a few other methods:

```
open PPrint
class print = object (self)
  inherit [document] CST.reduce as super
  method! visit_ADD = space ^^ plus ^^ break 1
  method! visit_MUL = space ^^ star ^^ break 1
  method! visit_expr e = group (super#visit_expr e)
  method! case_paren e = nest 2 (lparen ^^ self#visit_expr e) ^^ rparen
    (* ... a few more methods ... *)
end

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

CST2Document.ml

Again, this code makes no decisions regarding parenthesization.

This kind of output is produced:

```
65 *
( (22 + 38) * 69 +
  (24 + 58)
) +
( 84 * 70 +
  ( 20 +
    63 * 83 * 97 +
    49 * (70 + 0)
  ) *
  (93 + 89) *
  ( 12 * 15 + 85 +
    21
  )
)
```

À **vous** de jouer!