MP 8 – A Parser for PicoML

CS 342
Revision 1.0

Assigned Mar 26, 2024
Due Mar 26, 2024
Extension 96 hours (20% penalty)

1 Change Log

1.1 Update to disallow associativity and precedence annotations.

1.0 Initial Release.

2 Overview

In this MP, we will deal with the process of converting PicoML code into an abstract syntax tree using a parser. We will use the *ocamlyacc* tool to generate our parser from a description of the grammar. This parser will be combined with the lexer and type inferencer from previous MLs to make an interactive PicoML interpreter (well, it does not interpret yet), where you can type in PicoML expressions and see a proof tree of the expression's type:

```
Welcome to the Student parser

> let x = 5;;
val x : int


final environment:

{}

proof:

  {} |= let x = 5 : {x : int}
  |--{} |= 5 : int
```

To complete this MP, you will need to be familiar with describing languages with BNF grammars, adding attributes to return computations resulting from the parse, and expressing these attribute grammars in a form acceptable as input to *ocamlyacc*.

3 Given Files

mp8.mly: You should modify the file **mp8.mly**. You will be given slices of this file touching on most of the difficulties embodied in this assignment, but with far few constructs and strata. The skeleton contains some pieces of code that we have started for you, with triple dots indicating places where you should add code.

picomlIntPar.ml: This file contains the main body of the PicoML executable. It essentially connects your lexer, parser, and type inference code and provides a friendly prompt to enter PicoML expressions.

picomllex.mll: This file contains the ocamllex specification for the lexer. It is a modest expansion to the lexer you wrote for MP7.

common.ml: This file includes the types of expressions and declarations. It also contains the type inferencing code. Appropriate code from this file will automatically be called by the interactive loop defined in **picomlIntPar.ml**. You will want to use the types defined in this file, and probably some of the functions, when you are creating your attributes in **mp8.mly**.

4 Overview of `ocamlyacc`

Take a look at the given **mp8.mly** file. The grammar specification has a similar layout to the lexer specification of MP7. It begins with a header section (where you can add raw OCaml code), then has a section for directives (these start with a `%` character), then has a section that describes the grammar (this is the part after `%%`). You will only need to add to the last section.

4.1 Example

The following is the `exp` example from class.

```
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
exp:
   term                                    { Term_as_Expr $1 }
 | term Plus_token exp                     { Plus_Expr ($1, $3) }
 | term Minus_token exp                    { Minus_Expr ($1, $3) }
term:
   factor                                  { Factor_as_Term $1 }
 | factor Times_token term                 { Mult_Term ($1, $3) }
 | factor Divide_token term                { Div_Term ($1, $3) }
factor:
   Id_token                                { Id_as_Factor $1 }
 | Left_parenthesis exp Right_parenthesis  { Parenthesized_Expr_as_Factor $2 }
main:
 exp EOL                                   { $1 }
```

Recall from lecture that the process of transforming program code (i.e, as ASCII text) into an *abstract syntax tree* (AST) has two parts. First, the *lexical analyzer* (lexer) scans over the text of the program and converts the text into a sequence of *tokens*. The type of tokens in general may be a preexisting OCaml type, or a user-defined type created for the purpose. In the case where *ocamlyacc* is used, the type should be named `token` and the datatype `token` is created implicitly by the `%token` directives. These tokens are then fed into the *parser* created by entry points in your input, which builds the actual AST.

The first five lines in the example above define the sorts of tokens of the language. These directives are converted by *ocamlyacc* into an OCaml disjoint type declaration defining the type `token`. Notice that the `Id_token` token has data associated with it (this corresponds to writing `type token = ...| Id_token of string` in OCaml). The sixth line says that the start symbol for the grammar is the nonterminal called `main`. After the `%%` directive comes the important part: the productions. The format of the productions is fairly self-explanatory. The above specification describes the following extended BNF grammar:

$$
\begin{array}{lll}
S ::= E \textbf{ eol} & & \\
E ::= T & \mid T + E & \mid T - E \\
T ::= F & \mid F * T & \mid F/T \\
F ::= id & \mid (E) &
\end{array}
$$

An important fact about *ocamlyacc* is that **each production returns a value** that is to be put on the stack. We call this the *semantic value* of the production. It is described in curly braces by the *semantic action*. The semantic action is actual OCaml code that will be evaluated when this parsing algorithm reduces by this production. The result of this code is the semantic value, and it is placed on the stack to represent the nonterminal.

What do $1, $2, etc., mean? These refer to the positional values on the stack, and are replaced in the OCaml code by the semantic values of the subexpressions on the right-hand side of the production. Thus, the symbol $1 refers to the semantic value of the first subexpression on the right-hand side, and so on. As an example, consider the following production:

```
exp:
  ...
  | term Plus_token exp                    { Plus_Expr ($1, $3) }
```

When the parser reduces by this rule, $1 holds the semantic value of the `term` subexpression, and $3 holds the value of the `exp` subexpression. The semantic rule generates the AST representing the addition of the two, and the result becomes the semantic value for this production and is put on the stack to replace the top three items.

Also note that when tokens have associated data (like `Id_token`, which has a string), that associated data is treated as the semantic value of the token:

```
factor:
   Id_token                                { Id_as_Factor $1 }
```

Thus, the above $1 corresponds to the string component of the token, and not the token itself.

4.2 More Information

Here is a website you should check out if you would like more information or an alternate explanation of *ocamlyacc* usage:

- `http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`

5 Compiling

A `Makefile` is provided for this MP. After you make changes to `mp8.mly`, all you have to do is type `make` (or possibly `gmake` if you are using a non-linux machine) and the two needed executables will be rebuilt.

5.1 Running PicoML

The given `Makefile` builds executables called `picomlIntPar` and `picomlIntParSol`. The first is an executable for an interactive loop for the parser built from your solution to the assignment, and the second is one built from the standard solution. If you run `./picomlIntPar` or `./picomlIntParSol`, you will get an interactive screen, much like the OCaml interactive screen. You can type in PicoML expressions followed by a double semicolon, and they will be parsed and their types inferred and displayed:

```
Welcome to the Solution parser

> 3;;
```

```
val _ : int

final environment:

{}

proof:

  {} |= 3 : int

> let x = 3 + 4;;
val x : int

final environment:

{x : int}

proof:

  {} |= let x = 3  +  4 in x : int
  |--{} |= 3  +  4 : int
  | |--{} |= 3 : int
  | |--{} |= 4 : int
  |--{x : int} |= x : int

> let f = fun y -> y * x;;
val f : int -> int

final environment:

{f : int -> int,x : int}

proof:

  {x : int} |= let f = fun y -> y  *  x in f : int -> int
  |--{x : int} |= fun y -> y  *  x : int -> int
  | |--{y : int,x : int} |= y  *  x : int
  |   |--{y : int,x : int} |= y : int
  |   |--{y : int,x : int} |= x : int
  |--{f : int -> int,x : int} |= f : int -> int

> f 5;;
val _ : int

final environment:

{f : int -> int,x : int}

proof:

  {f : int -> int,x : int} |= f 5 : int
  |--{f : int -> int,x : int} |= f : int -> int
  |--{f : int -> int,x : int} |= 5 : int
```

Note: your output might have different type variables than those shown in subsequent examples.

Notice the accumulation of values in the (type) environment as expressions are entered. To reset the environment, you must quit the program (with CTRL+C) and start again.

6 Important Notes

- The BNF below for PicoML's grammar is ambiguous, and it is just a description of the *concrete* syntax of PicoML. You are also provided with a table listing the associativity/precedence attributes of the various language constructs. You are supposed to use the information given in this table in order to create a grammar that generates the same language as the given one, but that is unambiguous and enforces the constructs to be specified as in the table. Your actual *ocamlyacc* specification will consist of the latter grammar.
- **The BNF does not show the stratification needed to eliminate ambiguity. That is your job!** This will likely involve reorganizing things.
- *ocamlyacc* has some shortcut directives (`%left`, `%right`) for defining operator precedence, but **you may not use them in this MP**. Ambiguity should be resolved by stratifying the grammar, and our solution takes this approach.
- Even though the work in this MP is split into several problems, you should really have the overall view on how the disambiguated grammar will look because precedence makes the choices for the productions corresponding to the language constructs conceptually interdependent. You might want to read through all the expression types first and try to organize your stratification layers before starting. 90 percent of your intellectual effort in this MP will consist of disambiguating the grammar properly.
- **You will lose points for shift/reduce and reduce/reduce conflicts in your grammar**. Reduce/reduce conflicts will be penalized more heavily. `ocamlyacc -v mp8.mly` will inform you of any conflicts in your grammar, and produce a file `mp8.output` giving the action and goto table information and details of how conflicts arise and which productions are involved.

Stratification means breaking something up into layers. In the example 4.1 (page 2), we could have expressed the grammar more succinctly by

$$S ::= E \textbf{ eol}$$
$$E ::= id \mid E + E \mid E - E \mid E * E \mid E/E \mid (E)$$

This grammar, while compact, and comprehensible to humans, is highly ambiguous for the purposes of parsing. To render it unambiguous, we introduced intermediate non-terminals (layers, or strata) to express associativity and precedence of operators. You will need to perform similar transformations on the description given here to remove ambiguities and avoid shift-reduce or reduce-reduce conflicts.

7 Problem Setting

The concrete syntax of PicoML that you will need to parse is the following:

```
<main> ::= <exp> ;;
         | let IDENT = <exp> ;;
         | let rec IDENT = <exp> ;;

<exp> ::= IDENT
         | TRUE | FALSE | INT | FLOAT | STRING | NIL | UNIT
         | ( )
         | ( <exp> )
         | ( <exp> , <exp> )
```

```
        | let IDENT = <exp> in <exp>
        | let rec IDENT IDENT = <exp> in <exp>
        | <exp> <binop> <exp>
        | <monop> <exp>
        | <exp> && <exp>
        | <exp> || <exp>
        | [ ]
        | [ <list_contents> ]   /* sugar for non-empty lists, extra credit */
        | if <exp> then <exp> else <exp>
        | <exp> <exp>
        | fun IDENT -> <exp>
        | raise <exp>
        | try <exp> with n1 -> e1 | ...   /* extra credit */

<binop> :: + | - | * | / | +. | -. | *. | /. | ^ | :: | < | >
         | = | >= | <= | <> | mod | **

<monop> ::= fst | snd | hd | tl | print_string | ~

<list_contents> ::= <nonempty sequence of expressions separated by semicolons>
```

`IDENT` refers to an identifier token (only one token, takes a string as argument). `<binop>` refers to some infix identifier token (one for each infix operator). Similarly, `<monop>` are the unary operators. The nonterminals in this grammar are `main`, `exp`, `binop`, `monop`, and `list_contents`, with `main` being the start symbol.

The rest of the symbols are terminals, and their representations in OCaml are elements of the type `token`, defined at the beginning of the file `mp8.mly`. Our OCaml representation of terminals is not always graphically identical to the one shown in the above gramar; we have used concrete syntax in place of tokens for the terminals. For example, `::` is represented by `DCOLON` and + by `PLUS`. Our OCaml representation of the identifier tokens (`IDENT`) is achieved by the constructor `IDENT` that takes a string and yields a token, as constructed by the lexer from MP7.

Some of productions in the above grammar do not have a corresponding direct representation in abstract syntax. These language constructs are syntactic sugar and are summarized in the following table.

Expression	*Desugared form*	*Notes*
x `&&` y	`if` x `then` y `else false`	
x `||` y	`if` x `then true else` y	
x < y	(y > x)	with further expansion of ——
x <= y	(y > x) `||` (x = y)	
x >= y	(x > y) `||` (x = y)	
x <> y	`if` (x = y) `then false else true`	
`[]`	`NIL`	

Recall that identifying the tokens of the language is the job of lexer, and the parser (that you have to write in this MP) takes as input a *sequence of tokens*, such as `(INT 3) PLUS (INT 5)` and tries to make sense out of it by transforming it into an abstract syntax tree, in this case `BinOpAppExp(IntPlusOp, ConstExp(IntConst 3), ConstExp(IntConst 5))`.

The abstract syntax trees into which you have to parse your sequences of tokens are given by the following OCaml types (metatypes, to avoid confusion with PicoML types), present in the file `common.ml`:

```
type const = BoolConst of bool | IntConst of int | FloatConst of float
           | StringConst of string | NilConst | UnitConst

type mon_op = HdOp | TlOp | PrintOp | IntNegOp | FstOp | SndOp

type bin_op = IntPlusOp | IntMinusOp | IntTimesOp | IntDivOp
           | FloatPlusOp | FloatMinusOp | FloatTimesOp | FloatDivOp
```

```
            | ConcatOp | ConsOp | CommaOp | EqOp | GreaterOp
            | ModOp | ExpoOp

type exp =  (* Exceptions will be added in later MPs *)
    | VarExp of string                    (* variables *)
    | ConstExp of const                   (* constants *)
    | MonOpAppExp of mon_op * exp         (* % e1 for % is a builtin monadic operator *)
    | BinOpAppExp of bin_op * exp * exp   (* e1 % e2 for % is a builtin binary operator *)
    | IfExp of exp * exp * exp            (* if e1 then e2 else e3 *)
    | AppExp of exp * exp                 (* e1 e2 *)
    | FunExp of string * exp              (* fun x -> e1 *)
    | LetInExp of string * exp * exp      (* let x = e1 in e2 *)
    | LetRecInExp of string * string * exp * exp (* let rec f x = e1 in e2 *)
    | RaiseExp of exp                           (* raise e *)
    | TryWithExp of (exp * int option * exp * (int option * exp) list)
                                  (* try e with i -> e1 | j -> e1 | ... | k -> en *)

type dec =
     Anon of exp
    | Let of string * exp                 (* let x = exp *)
    | LetRec of string * string * exp     (* let rec f x = exp *)
```

Thus each sequence of tokens should either be interpreted as an element of metatype `exp` or `dec`, or should yield a parse error. Note that the metatypes `exp` and `dec` contain abstract, and not concrete syntax.

Notice that here we discuss how to parse *sequences of items*, and not the original text that the PicoML programmer writes - thus, while programming in PicoML one writes `3 + 5`, which will be lexed to `(INT 3) PLUS (INT 5)`, and then parsed to the form of abstract syntax tree as shown above.

If we do not specify the precedence and associativity of our language constructs and operators, the parsing function is not well-defined. For instance, how should `if true then 3 else 2 + 4` be parsed? Depending on how we "read" the above sequence of tokens, we get different results:

- If we read it as the sum of a conditional and a number, we get the same thing as if it were: `(if true then 3 else 2) + 4`
- If we read it as a conditional having a sum in its false branch, we get `if true then 3 else (2 + 4)`

The question is really which of the sum and the conditional binds its arguments tighter; that is, which one has a higher precedence (or which one has precedence over the other). In the first case, the conditional construct has a higher precedence whereas in the second, the sum operator has a higher precedence.

Another source of ambiguity arises from associativity of operators: how should `true && true && false` be parsed?

- If we read it as the conjunction between true and a conjunction, we get `true && (true && false)`
- If we read it as a conjunctions between a conjunction and false, we get `(true && true) && false`

In the first case, `&&` is right-associative; in the second, it is left-associative.

The desired precedence and associativity of the language constructs and operators (which impose a unique parsing function) are given below, where a left-associative operator is preceded by "left", a right-associative operator by "right", and precedence decreases downwards on the lines (thus two items listed on the same line have the same precedence).

```
fst _   snd _   hd _   tl _   print_string _  ~ _
left _ _    (application is left associative, and binds tighter than anything
             else, except monop application)
raise_
```

```
right **
left *  left *.  left /  left /. left mod
left +  left +.  left -  left -.  left ^
right ::   (:: is right associative, binds tighter than anything but the above)
left =  left <  left >  left <=  left >=  left <>
left _&&_
left _||_
if_then_else_
fun_->_
let_=_in_
let rec_=_in_
try_with_->_|_|_ ..., where | is right associative
```

Above, the underscores are just a graphical indication of the places where the various syntactic constructs expect their "arguments". For example, the conditional has three underscores, the first for the condition, the second for the `then` branch, and the third for the `else` branch.

8 Problems

At this point, your assignment for this MP should already be fairly clear. The following problems just break your assignment into pieces and are meant to guide you towards the solution. A word of warning is however in order here: The problem of writing a parser is *not* a modular one because the parsing of each language construct depends on all the other constructs. Adding a new syntactic category may well force you to go back and rewrite all the categories already present. Therefore you should **approach the set of problems as a whole**, and always keep in mind the precedences and associativities given for the PicoML constructs.

You are allowed, and even encouraged, **to add to your grammar new nonterminals (together with new productions)** in addition to the one that we require (`main`). In addition, you may find it desirable to **rewrite or reorganize the various productions we have given you**. The productions given are intended only to be enough to allow you to start testing your additions.

Also, **it is allowed that you define the constructs in an order that is different from the one we have given here**. For instance, we have gathered the requirements according to overall syntactic and semantic similarities (e.g., grouping arithmetic operators together); you may rather want to group the constructs according to their precedence; you are absolutely free to do that. However, we require that the non-terminal `main` that we introduced in the problem statement be present in your grammar and that it produces exactly the same set of strings as described by the grammar in Section 7, obeying the precedences and associativities also described in that section.

In between each of these examples we have reset the environment.

1. (5 pts) In the file `mp8.mly`, add the integer, unit, boolean, float, and string constants.

```
> "hi";;


final environment:

{}

proof:

  {} |= let _ = "hi" : {}
  |--{} |= "hi" : string
```

2. (5 pts) Add parentheses.

```
> ("hi");;


final environment:

{}

proof:

  {} |= let _ = "hi" : {}
  |--{} |= "hi" : string
```

3. (5 pts) Add pairs. Note that unlike OCaml, PicoML requires opening and closing parentheses around pairs.

```
> ("hi",3);;


final environment:

{}

proof:

  {} |= let _ = ("hi" , 3) : {}
  |--{} |= ("hi" , 3) : string * int
    |--{} |= "hi" : string
    |--{} |= 3 : int
```

4. (8 pts) Add unary operators. These should be treated for precedence and associativity as an application of single argument functions.

```
> hd [];;


final environment:

{}

proof:

  {} |= let _ = hd [] : {}
  |--{} |= hd [] : 'c
    |--{} |= [] : 'c list
```

5. (12 pts) Add comparison operators.

```
> 3 < 5;;


final environment:

{}

proof:

  {} |= let _ = 5  >  3 : {}
  |--{} |= 5  >  3 : bool
    |--{} |= 5 : int
    |--{} |= 3 : int
```

6. (12 pts) Add infix operators. You will need to heed the precedence and associativity rules given in the table above.

```
> 3 + 4 * 8;;


final environment:

{}

proof:

  {} |= let _ = 3  +  (4  *  8) : {}
  |--{} |= 3  +  (4  *  8) : int
    |--{} |= 3 : int
    |--{} |= 4  *  8 : int
      |--{} |= 4 : int
      |--{} |= 8 : int
```

7. (10 pts) Add :: (list consing).

```
> 3 :: 2 :: 1 :: [];;


final environment:

{}

proof:

  {} |= let _ = 3  ::  (2  ::  (1  ::  [])) : {}
  |--{} |= 3  ::  (2  ::  (1  ::  [])) : int list
    |--{} |= 3 : int
    |--{} |= 2  ::  (1  ::  []) : int list
      |--{} |= 2 : int
      |--{} |= 1  ::  [] : int list
```

```
        |--{} |= 1 : int
        |--{} |= [] : int list
```

8. (8 pts) Add `let_in_` and `let_rec_in`.

```
> let rec f x = 3 :: x :: (f x) in f 8;;


final environment:

{}

proof:

  {} |= let _ = let rec f x = 3  ::  (x  ::  (f x)) in f 8 : {}
  |--{} |= let rec f x = 3  ::  (x  ::  (f x)) in f 8 : int list
    |--{x : int, f : int -> int list} |= 3  ::  (x  ::  (f x)) : int list
    | |--{x : int, f : int -> int list} |= 3 : int
    | |--{x : int, f : int -> int list} |= x  ::  (f x) : int list
    |   |--{x : int, f : int -> int list} |= x : int
    |   |--{x : int, f : int -> int list} |= f x : int list
    |     |--{x : int, f : int -> int list} |= f : int -> int list
    |     |--{x : int, f : int -> int list} |= x : int
    |--{f : int -> int list} |= f 8 : int list
      |--{f : int -> int list} |= f : int -> int list
      |--{f : int -> int list} |= 8 : int
```

9. (20 pts) Add `fun__->__` and `if__then__else__`.

```
> if true then fun x -> 3 else fun x -> 4;;


final environment:

{}

proof:

  {} |= let _ = if true then fun x -> 3 else fun x -> 4 : {}
  |--{} |= if true then fun x -> 3 else fun x -> 4 : 'c -> int
    |--{} |= true : bool
    |--{} |= fun x -> 3 : 'c -> int
    | |--{x : 'c} |= 3 : int
    |--{} |= fun x -> 4 : 'c -> int
      |--{x : 'c} |= 4 : int
```

10. (10 pts) Add application.

```
> (fun x -> x + x + 3) 4;;


final environment:

{}

proof:

  {} |= let _ = (fun x -> (x  +  x)  +  3) 4 : {}
  |--{} |= (fun x -> (x  +  x)  +  3) 4 : int
    |--{} |= fun x -> (x  +  x)  +  3 : int -> int
    | |--{x : int} |= (x  +  x)  +  3 : int
    |   |--{x : int} |= x  +  x : int
    |   | |--{x : int} |= x : int
    |   | |--{x : int} |= x : int
    |   |--{x : int} |= 3 : int
    |--{} |= 4 : int
```

11. (11 pts) Add `&&` and `||`.

```
> true || false && true;;


final environment:

{}

proof:

  {} |= let _ = if true then true else if false then true else false : {}
  |--{} |= if true then true else if false then true else false : bool
    |--{} |= true : bool
    |--{} |= true : bool
    |--{} |= if false then true else false : bool
      |--{} |= false : bool
      |--{} |= true : bool
      |--{} |= false : bool
```

12. (5 pts) Add `raise`.

```
> raise (fun x -> x) 4 + 3;;


final environment:

{}

proof:
```

```
{} |= let _ = (raise (fun x -> x) 4)  +  3 : {}
|--{} |= (raise (fun x -> x) 4)  +  3 : int
  |--{} |= raise (fun x -> x) 4 : int
  | |--{} |= (fun x -> x) 4 : int
  |   |--{} |= fun x -> x : int -> int
  |   | |--{x : int} |= x : int
  |   |--{} |= 4 : int
  |--{} |= 3 : int
```

13. (5 pts) Add syntactic sugar for lists to your expressions. More precisely, add the following expressions to the grammar:

    - `exp` → [*list_contents*]

    where *list_contents* is a non-empty sequence of expressions separated by semicolons. It has to be the case that semicolon binds less tightly than any other language construct or operator.

```
> [1; 2; 3];;


final environment:

{}

proof:

  {} |= let _ = 1  ::  (2  ::  (3  ::  [])) : {}
  |--{} |= 1  ::  (2  ::  (3  ::  [])) : int list
    |--{} |= 1 : int
    |--{} |= 2  ::  (3  ::  []) : int list
      |--{} |= 2 : int
      |--{} |= 3  ::  [] : int list
        |--{} |= 3 : int
        |--{} |= [] : int list
```

9 Additional Tests

1. Can you pass this test? Make sure your parser parses the expression as in the example.

```
> 3 - 4 - 2 * 9 < 10 && true ;;


final environment:

{}

proof:
```

```
{} |= let _ = if 10  <  ((3  -  4)  -  (2  *  9)) then true else false : {}
|--{} |= if 10  <  ((3  -  4)  -  (2  *  9)) then true else false : bool
  |--{} |= 10  <  ((3  -  4)  -  (2  *  9)) : bool
  | |--{} |= 10 : int
  | |--{} |= (3  -  4)  -  (2  *  9) : int
  |   |--{} |= 3  -  4 : int
  |   | |--{} |= 3 : int
  |   | |--{} |= 4 : int
  |   |--{} |= 2  *  9 : int
  |     |--{} |= 2 : int
  |     |--{} |= 9 : int
  |--{} |= true : bool
  |--{} |= false : bool
```

2. This one?

```
> if true then 1 else 0 + 2;;


final environment:

{}

proof:

  {} |= let _ = if true then 1 else 0  +  2 : {}
  |--{} |= if true then 1 else 0  +  2 : int
    |--{} |= true : bool
    |--{} |= 1 : int
    |--{} |= 0  +  2 : int
      |--{} |= 0 : int
      |--{} |= 2 : int
```

3. How about this one?

```
> (fun x -> ()) 3;;


final environment:

{}

proof:

  {} |= let _ = (fun x -> ()) 3 : {}
  |--{} |= (fun x -> ()) 3 : unit
    |--{} |= fun x -> () : int -> unit
    | |--{x : int} |= () : unit
    |--{} |= 3 : int
```