DAY 5 FP

```
(* main.ml *)
let lexbuf outchan l = (* Compilation: input=lexical buffer, output=outchan (caml2html:
  ...... .......
    (..........
       (......
          (.........
             (.........
                (.... !.....
                   (.......
                      (.........
                         (.........
                            (Parser.exp Lexer.token l)))))))))
```

Compile from the last line

body

```
let rec fib n = if n < 2 then ... : Lexing.lexbuf (* Source code *)

[Parser.LET; Parser.REC;
 Parser.IDENT("fib"); Parser.IDENT("n"); Parser.EQUAL;
 Parser.IF; Parser.IDENT("n"); Parser.INT(2); Parser.THEN;
 ...] : Parser.token list (* Tokens *)

Syntax.LetRec({name = ("fib", Var {contents = None}),mincaml doesnt require var type
               args = [("n", Var {contents = None})],
               body = Syntax.If(Syntax.LE(Syntax.Int(2),
                                          Syntax.Var("n")),
                                then_expression,
                                else_expression))}) : Syntax.t (* Abstract Syntax Tree
```

Compile a fib function: from “let” “rec” defined in parser module -> IDENT (identifier) “fib” “n”
Parser accepts this kind of tokens and convert them to syntax tree

```
akaishuushandeMacBook-Pro:min-caml akaishuushan$ head .ocamlinit
(* Save this file as ".ocamlinit" under the directory, where you have min-caml.top *)

open Alpha;; open Asm;; open Assoc;; open Beta;; open Closure;; open ConstFold;;
open Elim;; open Emit;; open Id;; open Inline;; open KNormal;; open Lexer;; open M;;
open Main;; open Parser;; open RegAlloc;; open S;; open Simm;; open Syntax;;
open Type;; open Typing;; open Virtual;;

let compose f g x = (* Function composition *)
  f (g x)
```

```
akaishuushandeMacBook-Pro:min-caml akaishuushan$ head -20 .ocamlinit
(* Save this file as ".ocamlinit" under the directory, where you have min-caml.top *)

open Alpha;; open Asm;; open Assoc;; open Beta;; open Closure;; open ConstFold;;
open Elim;; open Emit;; open Id;; open Inline;; open KNormal;; open Lexer;; open M;;
open Main;; open Parser;; open RegAlloc;; open S;; open Simm;; open Syntax;;
open Type;; open Typing;; open Virtual;;

let compose f g x = (* Function composition *)
  f (g x)

let fib_p = "let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 1) in print(fib 5)"

let knf_p = "print(1 + 2 + 3)"
let if_p  = "print(if 1 > 0 then 1 else 0)"

let alpha_p = "print(let x = 1 in let x = 2 in x)"

let lex     = Lexing.from_string
let ast     = compose (Parser.exp Lexer.token) Lexing.from_string
let typing  = compose Typing.f ast
```

进入min-caml

```
akaishuushandeMacBook-Pro:min-caml akaishuushan$ ./min-caml.top
        OCaml version 4.07.0
```

fib_p is the predefined var string:

```
# fib_p
  ;;
- : string =
"let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 1) in print(fib 5)"
```

lex fib_p create a buffer (????

```
# lex fib_p;;
- : Lexing.lexbuf =
{Lexing.refill_buff = <fun>;
 lex_buffer =
  Bytes.of_string "let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 1) in print(fib 5)
";
 lex_buffer_len = 78; lex_abs_pos = 0; lex_start_pos = 0; lex_curr_pos = 0;
 lex_last_pos = 0; lex_last_action = 0; lex_eof_reached = true;
 lex_mem = [||];
 lex_start_p =
  {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0};
 lex_curr_p =
  {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0}}
```

```
# let b = lex fib_p;;
val b : Lexing.lexbuf =
  {Lexing.refill_buff = <fun>;
   lex_buffer =
    Bytes.of_string "let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 1) in print(fib
5)";
   lex_buffer_len = 78; lex_abs_pos = 0; lex_start_pos = 0; lex_curr_pos = 0;
   lex_last_pos = 0; lex_last_action = 0; lex_eof_reached = true;
   lex_mem = [||];
   lex_start_p =
    {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0};
   lex_curr_p =
    {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0}}
```

iterating **Lexer.token b** convert the strings into tokens then syntax tree:

```
# Lexer.token b;;
- : Parser.token = LET
# Lexer.token b;;
- : Parser.token = REC
# Lexer.token b;;
- : Parser.token = IDENT "fib"
# Lexer.token b;;
- : Parser.token = IDENT "n"
# Lexer.token b;;
- : Parser.token = EQUAL
# Lexer.token b;;
- : Parser.token = IF
# ast;;
- : string -> Syntax.t = <fun>
# ast fib_p;;
- : Syntax.t =
LetRec
 ({name = ("fib", Var {contents = None});
   args = [("n", Var {contents = None})];
   body =
    If (Not (LE (Syntax.Int 2, Syntax.Var "n")), Syntax.Int 1,
     Add (App (Syntax.Var "fib", [Sub (Syntax.Var "n", Syntax.Int 1)]),
      App (Syntax.Var "fib", [Sub (Syntax.Var "n", Syntax.Int 1)])))},
 App (Syntax.Var "print", [App (Syntax.Var "fib", [Syntax.Int 5])]))
```

G(rammar) = (T(erminal), N(onterminal), S(tarting), R(ules))

parser.mly

```
| INT
    { Int($1) }
```

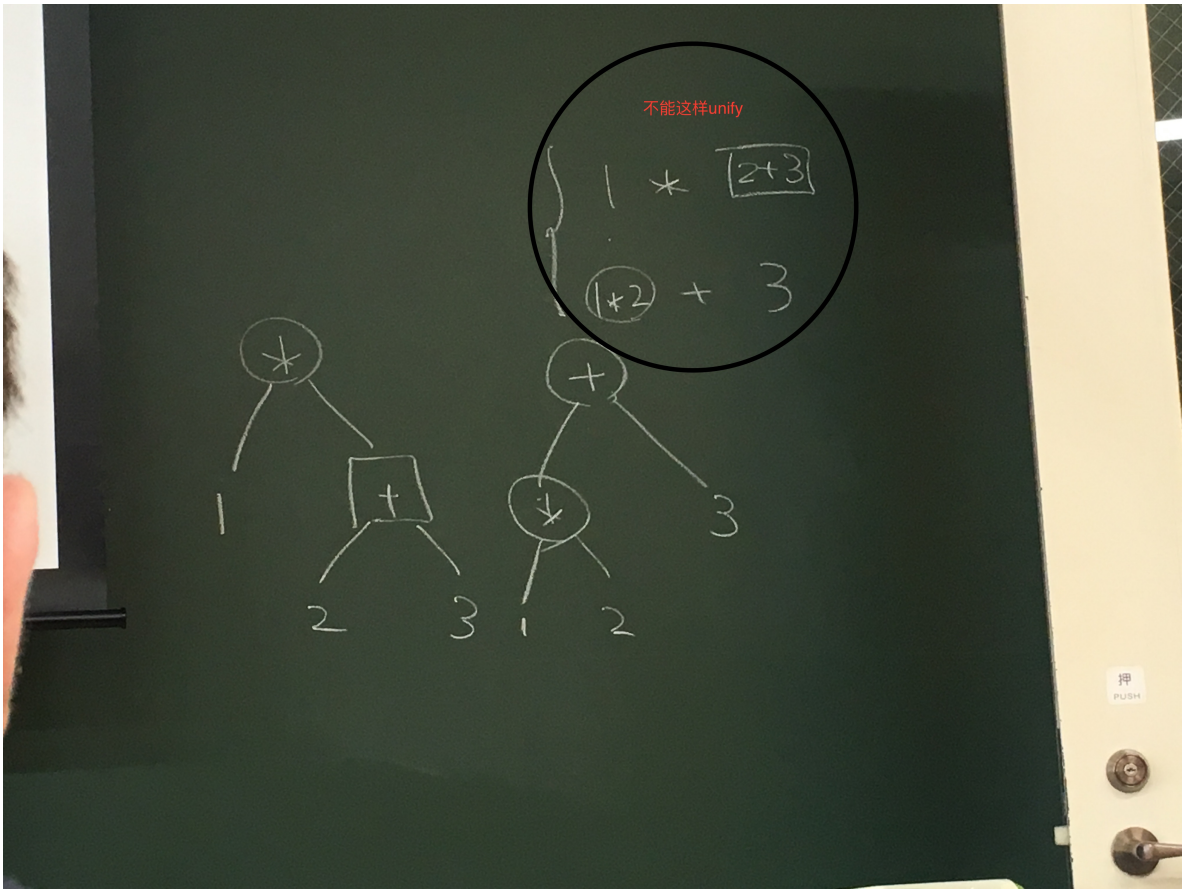
INT: token
Int: member of syntax tree

If want to add new token -> in **paser.mly**

*Parser generation theory
*Finite state automata

Pattern mathing is to match pattern to a concrete data structure
unification question contains some unknowns and copy paste the corresponding known parts to each other.

The target of unification is in a tree form


不能这样unify
1 * 2+3
1*2 + 3
*
+
1
+
2
3
+
*
3
1
2