```
%{
```

Definitions of variables/functions/types used by the generated parser.

```
(* parserが利用する変数、関数、型などの定義 *)
open Syntax
let addtyp x = (x, Type.gentyp ())
%}

```

Data type definitions of lexical elements

```
/* (* 字句を表すデータ型の定義 (caml2html: parser_token) *) */
%token <bool> BOOL
%token <int> INT
%token <float> FLOAT
%token NOT
%token MINUS
%token PLUS
%token MINUS_DOT
%token PLUS_DOT
%token AST_DOT
%token SLASH_DOT
%token EQUAL
%token LESS_GREATER
%token LESS_EQUAL
%token GREATER_EQUAL
%token LESS
%token GREATER
%token IF
%token THEN
%token ELSE
%token <Id.t> IDENT
%token LET
%token IN
%token REC
%token COMMA
%token ARRAY_CREATE
%token DOT
%token LESS_MINUS
%token SEMICOLON
%token LPAREN
```

```
%token RPAREN
%token EOF

```

Definitions of operator priority and associativity
Lower priority first

```
/* (* 優先順位とassociativityの定義（低い方から高い方へ） (caml2html: parser_prior) *) */
%right prec_let
%right SEMICOLON
%right prec_if
%right LESS_MINUS
%left COMMA
%left EQUAL LESS_GREATER LESS GREATER LESS_EQUAL GREATER_EQUAL
%left PLUS MINUS PLUS_DOT MINUS_DOT
%left AST_DOT SLASH_DOT
%right prec_unary_minus
%left prec_app
%left DOT

```

Declaration of the starting symbol for MinCaml CFG (context-free grammar)

```
/* (* 開始記号の定義 *) */
%type <Syntax.t> exp
%start exp

%%

```

We can place a simple expression at the argument position of a function. Compound expression requires the use of parentheses.

```
simple_exp: /* (* 括弧をつけなくても関数の引数になれる式 (caml2html: parser_simple) *) */
| LPAREN exp RPAREN
    { $2 }
| LPAREN RPAREN
    { Unit }
| BOOL
    { Bool($1) }
| INT
    { Int($1) }
| FLOAT
    { Float($1) }
| IDENT
```

The detail of *.mly file format and ocamlyacc is found in sections 12.3 and 12.4 of ocaml-4.03-refman.pdf.

```
  { Var($1) }
| simple_exp DOT LPAREN exp RPAREN
  { Get($1, $4) }
```

General expression (including simple expression)

```
exp: /* (* 一般の式 (caml2html: parser_exp) *) */
| simple_exp
  { $1 }
| NOT exp
  %prec prec_app
  { Not($2) }
| MINUS exp
  %prec prec_unary_minus
  { match $2 with
```

For the sake of handling negative FP-values

```
  | Float(f) -> Float(-.f) (* -1.23などは型エラーではないので別扱い *)
  | e -> Neg(e) }
```

Addition

```
| exp PLUS exp /* (* 足し算を構文解析するルール (caml2html: parser_add) *) */
  { Add($1, $3) }
| exp MINUS exp
  { Sub($1, $3) }
| exp EQUAL exp
  { Eq($1, $3) }
| exp LESS_GREATER exp
  { Not(Eq($1, $3)) }
| exp LESS exp
  { Not(LE($3, $1)) }
| exp GREATER exp
  { Not(LE($1, $3)) }
| exp LESS_EQUAL exp
  { LE($1, $3) }
| exp GREATER_EQUAL exp
  { LE($3, $1) }
| IF exp THEN exp ELSE exp
  %prec prec_if
  { If($2, $4, $6) }
| MINUS_DOT exp
  %prec prec_unary_minus
```

```
  { FNeg($2) }
| exp PLUS_DOT exp
  { FAdd($1, $3) }
| exp MINUS_DOT exp
  { FSub($1, $3) }
| exp AST_DOT exp
  { FMul($1, $3) }
| exp SLASH_DOT exp
  { FDiv($1, $3) }
| LET IDENT EQUAL exp IN exp
  %prec prec_let
  { Let(addtyp $2, $4, $6) }
| LET REC fundef IN exp
  %prec prec_let
  { LetRec($3, $5) }
| exp actual_args
  %prec prec_app
  { App($1, $2) }
| elems
  { Tuple($1) }
| LET LPAREN pat RPAREN EQUAL exp IN exp
  { LetTuple($3, $6, $8) }
| simple_exp DOT LPAREN exp RPAREN LESS_MINUS exp
  { Put($1, $4, $7) }
| exp SEMICOLON exp
  { Let((Id.gentmp Type.Unit, Type.Unit), $1, $3) }
| ARRAY_CREATE simple_exp simple_exp
  %prec prec_app
  { Array($2, $3) }
| error
  { failwith
  (Printf.sprintf "parse error near characters %d-%d"
     (Parsing.symbol_start ())
     (Parsing.symbol_end ())) }

fundef:
```

```
| IDENT formal_args EQUAL exp
    { { name = addtyp $1; args = $2; body = $4 } }

formal_args:
| IDENT formal_args
    { addtyp $1 :: $2 }
| IDENT
    { [addtyp $1] }

actual_args:
| actual_args simple_exp
    %prec prec_app
    { $1 @ [$2] }
| simple_exp
    %prec prec_app
    { [$1] }

elems:
| elems COMMA exp
    { $1 @ [$3] }
| exp COMMA exp
    { [$1; $3] }

pat:
| pat COMMA IDENT
    { $1 @ [addtyp $3] }
| IDENT COMMA IDENT
    { [addtyp $1; addtyp $3] }
```