

MiniML Grammar Spec

Brahima,Yukai,Zaid

2 fevrier, 2023

Contents

1	Change Log	2
2	Notes	2
2.1	Todo	2
3	Lexing Tokens	3
3.1	Separators	3
3.2	Mots-Clefs	3
3.3	Types	3
3.4	Operators	3
3.5	Valeurs_Atomiques	3
3.6	Identificateur	3
3.6.1	Constructeurs	3
4	Grammaire	4
4.1	Definitions	4
4.2	Expressions	4
4.3	Filtrage et Motifs	5
4.4	Types	5
5	Traduction	6
5.1	Programmes	6
5.2	Suites de commandes	6
5.3	Définitions	6
5.4	Constructeurs	6
5.5	Litteraux et Expressions	6
5.6	Motifs et Filtrage	7

1 Change Log

- 2 fevrier, 2023 Première Version
- 2 fevrier, 2023 Première Correction
 - Ajout de Unit
 - Ajout des patterns
 - Rename Value -> Litteral
 - Retrait Operators/Type de Base
 - Retrait Sucre Syntaxique pour le moment
- 7 fevrier, 2023 Deuxième Correction
 - Simplification (des _LS)
 - Ajout des constructeurs infixes
 - Fix des Match Patterns
 - Fix Definition Globales
 - Reintroduction du Parsing Operators/Type de Base
- 11 fevrier, 2023 Post Reunion
 - Ajout et compréhension des vartypes
 - Ajout du keyword rec
 - Ajout des types paramettrer

2 Notes

2.1 Todo

- Crée du Sucre Syntaxique. # Plus Tard

3 Lexing Tokens

3.1 Separators

`{ } [] () ; , * -> | =`

3.2 Mots-Clefs

`let fun in match with type of rec`

3.3 Types

`int bool unit`

3.4 Operators

`+ - % / & | ~ :: && || *`

3.5 Valeurs_Atomiques

`integer := ('-')?['0'-'9']*
boolean := ("true"|"false")`

3.6 Identificateur

`alphanum := ['a'-'z' 'A'-'Z' '0'-'9' '_']*
basic_ident := ['a'-'z' '_'] alphanum
vartype := ['`t'] [0..9]*`

3.6.1 Constructeurs

`constructeur_ident := ['A'-'Z'] alphanum
constructeur_infixes := [":" ' ', ']`

4 Grammaire

```
Prog := | Def
      | Expr
      | Prog ;; Prog
```

4.1 Definitions

```
Def := | let basic_ident = Expr
      | let basic_ident list = Expr
      | let rec basic_ident list = Expr
      | type vartype list basic_ident = NewConstructor_Case #TypeDef
```

```
NewConstructor_Case := | constructeur_ident
                      | constructeur_ident of Type
                      | NewConstructor_Case '|' NewConstructor_Case
```

4.2 Expressions

```
Litteral := | integer
            | boolean
            | ( ) # Unit
```

```
Expr := | ( Expr )
        | Litteral
        | basic_ident
        | UnaryOperator Expr
        | Expr BinaryOperator Expr
        | Expr Expr # Call
        | Expr ; Expr # Sequence
        | let basic_ident = Expr in Expr # Binding
        | fun basic_ident list -> Expr # Lambda
        | Expr constructeur_infixes Expr
        | constructeur_ident Expr # Built Expr
        | constructeur_ident # Avoid Nil ()
        | let basic_ident list = Expr in Expr
        | let rec basic_ident basic_ident list = Expr in Expr
        | match Expr with Match_Case
```

```
UnaryOperator := | ~
                 | -
```

```
BinaryOperator := | &
```

```

| &&
| ||
| +
| -
| /
| %
| *

```

4.3 Filtrage et Motifs

```

Match_Case := | Pattern -> Expr
              | Pattern -> Expr '|' Match_Case

```

```

Pattern :=   | ( Pattern )
              | Litteral
              | basic_ident
              | _
              | constructeur_ident
              | constructeur_ident Pattern
              | Pattern constructeur_infixes Pattern

```

4.4 Types

```

Type      := | (Type)
              | int
              | bool
              | unit
              | Type * Type # Tuple_Type
              | Type -> Type # Lambda_Type
              | vartype # 'a
              | basic_ident # defined type
              | Type List # Parametred Type (EXEMPLE : int list option)

```

5 Traduction

5.1 Programmes

(PROG) si $\vdash \text{pi} \rightarrow \omega$
alors $\vdash [\text{pi}] \rightarrow \text{Prog}(\omega)$

5.2 Suites de commandes

(DEFS) si $d \in \text{DEF}$, si $\vdash d \rightarrow \omega$ et si $\vdash \text{pi} \rightarrow \omega'$
alors $\vdash (\text{Def}(d), \text{pi}) \rightarrow (\omega, \omega')$
(BLOCK) si $b \in \text{BLOCK}$, si $\vdash b \rightarrow \omega$ et si $\vdash \text{pi} \rightarrow \omega'$
alors $\vdash (\text{Expr}(b), \text{pi}) \rightarrow (\text{Do}(\omega), \omega')$

5.3 Définitions

(VALDEF) si $\vdash v \rightarrow v'$, si $\vdash e \rightarrow e'$ et si $\vdash \text{pi} \rightarrow \omega'$
alors $\vdash (\text{VariableDef}(d), \text{pi}) \rightarrow (\omega, \omega')$
(FUNREC)
(TYPDEF) si $\text{td} \in \text{CONSTR}$, si $\vdash \text{td} \rightarrow \omega$ et si $\vdash \text{pi} \rightarrow \omega'$
alors $\vdash (\text{TypeDef}(n, [t_1, \dots, t_n], \text{td}), \text{pi})$
 $\rightarrow (\text{Typ_Def}(n, [t_1, \dots, t_n], \omega), \omega')$

5.4 Constructeurs

(SYNON) si
(DATYP)
(COMPUT)

To be done

5.5 Litteraux et Expressions

(INT) si $i \in \text{NUM}$
alors $\vdash \text{Integer}(i) \rightarrow \text{Expr_Int}(i)$
(TRUE) si $\vdash b \rightarrow \text{true}$
alors $\vdash \text{Boolean}(b) \rightarrow \text{Expr_Constructor}(\text{True}, [])$
(FALSE) si $\vdash b \rightarrow \text{false}$
alors $\vdash \text{Boolean}(b) \rightarrow \text{Expr_Constructor}(\text{False}, [])$
(TUPLE) si $\vdash e_1 \rightarrow e_1, \dots, \vdash e_N \rightarrow e_N$
alors $\vdash \text{Tuple}([e_1, \dots, e_N])$
 $\rightarrow \text{Expr_Constructor}(\text{Tuple}, [e_1, \dots, e_N])$
(CONSTR) si $\vdash e \rightarrow e'$

```

    alors  $\vdash$  Construct((c,e))
             $\rightarrow$  Expr_Constructor(Cons_Named c, [e'])
(BIND) si  $\vdash i \rightarrow i'$  et si  $\vdash c \rightarrow c'$ 
    alors  $\vdash$  Binding((v,i,c))
             $\rightarrow$  Expr_Block(Blk([Ins_Let (v, i')], c'))
(MATCH) si  $\vdash m \rightarrow m'$ , si  $m_1 \in \text{CASE}, \dots, m_N \in \text{CASE}$ ,
    si  $\vdash m_1 \rightarrow m_1, \dots$  et si  $\vdash m_N \rightarrow m_N$ 
    alors  $\vdash$  Match((m,[m1,...,mN]))
             $\rightarrow$  Expr_Match(m', [m_1,...,m_N])

```

5.6 Motifs et Filtrage

```

(LITPAT1) si  $l = \text{Integer}(l')$  et  $\vdash e \rightarrow e'$ 
    alors  $\vdash$  Case(LitteralPattern(l), e)  $\rightarrow$ 
             $\rightarrow$  MatchPat(Int_litt l', [], e')
(LITPAT2) si  $l = \text{Boolean}(\_)$ , si  $\vdash l \rightarrow l'$  et  $\vdash e \rightarrow e'$ 
    alors  $\vdash$  Case(LitteralPattern(l), e)
             $\rightarrow$  MatchPat(l', [], e')
(LITPAT3) si  $l = \text{Unit}$  et  $\vdash e \rightarrow e'$ 
    alors  $\vdash$  Case(LitteralPattern(l), e)
             $\rightarrow$  MatchPat(Unit, [], e')
(TUPAT) si  $p_1 \in \text{CASE}, \dots, p_N \in \text{CASE}$ ,
    si  $\vdash p_1 \rightarrow p_1, \dots, \vdash p_N \rightarrow p_N$  et  $\vdash e \rightarrow e'$ 
    alors  $\vdash$  Case(TuplePattern([p1,...,pN]), e)
             $\rightarrow$  MatchPat(Tuple, [p_1,...,p_N], e')
(CONSPAT) si  $c \in \text{CASE}$ , si  $\vdash c \rightarrow c'$  et  $\vdash e \rightarrow e'$ 
    alors  $\vdash$  Case(ConstructorPattern((n,c)), e)
             $\rightarrow$  MatchPat(Cons_Named(n), c', e')

```