MiniML Grammar Spec

Brahima,Yukai,Zaid

2 fevrier, 2023

Contents

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 parametrer

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

```
# For Type Inference
Variable :=     | basic_ident
                | basic_ident : Type

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

4.1 Definitions

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

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

4.2 Expressions

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

Expr    :=  | ( Expr )
            | Litteral
            | Variable
            | UnaryOperator Expr
            | Expr BinaryOperator Expr
            | Expr Expr # Call
            | Expr ; Expr # Sequence
            | let Variable = Expr in Expr # Binding
            | fun Variable list -> Expr # Lambda
            | Expr constructeur_infixes Expr
            | constructeur_ident Expr # Built Expr
            | constructeur_ident # Avoid Nil ()
            | let basic_ident Variable Variable list = Expr in Expr
            | let rec basic_ident Variable Variable 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  ⊢ pi -> ω
    alors  ⊢ [pi] -> Prog(ω)
```

5.2 Suites de commandes

```
(DEFS) si d ∈ DEF, si ⊢ d -> ω et si ⊢ pi -> ω'
    alors  ⊢ (Def(d), pi) -> (ω, ω')
(BLOCK) si b ∈ BLOCK, si ⊢ b -> ω et si ⊢ pi -> ω'
    alors  ⊢ (Expr(b), pi) -> (Do(ω), ω')
```

5.3 Définitions

```
(VALDEF) si ⊢ v -> v',si  ⊢ e -> e' et si ⊢ pi -> ω'
    alors  ⊢ (VariableDef(d), pi) -> (ω, ω')
(FUNREC)
(TYPDEF) si td ∈ CONSTR, si ⊢ td -> ω et si ⊢ pi -> ω'
    alors  ⊢ (TypeDef(n, [t1,...,tn], td), pi)
            -> (Typ_Def(n, [t1,...,tn], ω), ω')
```

5.4 Constructeurs

```
(SYNON) si
(DATYP)
(COMPUT)
```

To be done

5.5 Litteraux et Expressions

```
(INT) si i ∈ NUM
    alors ⊢ Integer(i) -> Expr_Int(i)
(TRUE) si ⊢ b -> true
    alors ⊢ Boolean(b) -> Expr_Constructor(True,[])
(FALSE) si ⊢ b -> false
    alors ⊢ Boolean(b) -> Expr_Constructor(False,[])
(TUPLE) si ⊢ e1 -> e_1, ..., si ⊢ eN -> e_N
    alors ⊢ Tuple([e1,...,eN])
            -> Expr_Constructor(Tuple, [e_1,...,e_N])
(CONSTR) si ⊢ e -> e'
```

```
    alors ⊢ Construct((c,e))
            -> Expr_Constructor(Cons_Named c, [e'])
(BIND) si ⊢ i -> i' et si ⊢ c -> c'
    alors ⊢ Binding((v,i,c))
        -> Expr_Block(Blk([Ins_Let (v, i')], c'))
(MATCH) si ⊢ m -> m', si m1 ∈ CASE,..., si mN ∈ CASE,
        si ⊢ m1 -> m_1, ... et si ⊢ mN -> m_N
    alors ⊢ Match((m,[m1,...,mN]))
        -> Expr_Match(m', [m_1,...,m_N])
```

5.6 Motifs et Filtrage

```
(LITPAT1) si l = Integer(l') et ⊢ e -> e'
    alors ⊢ Case(LitteralPattern(l), e) ->
        -> MatchPat(Int_litt l', [], e')
(LITPAT2) si l = Boolean(_), si ⊢ l -> l' et ⊢ e -> e'
    alors ⊢ Case(LitteralPattern(l), e)
        -> MatchPat(l', [], e')
(LITPAT3) si l = Unit et ⊢ e -> e'
    alors ⊢ Case(LitteralPattern(l), e)
        -> MatchPat(Unit, [], e')
(TUPAT) si p1 ∈ CASE,..., si pN ∈ CASE,
        si ⊢ p1 -> p_1, ..., si ⊢ pN -> p_N et ⊢ e -> e'
    alors ⊢ Case(TuplePattern([p1,...,pN]), e)
        -> MatchPat(Tuple, [p_1,...,p_N], e')
(CONSPAT)  si c ∈ CASE, si ⊢ c -> c' et ⊢ e -> e'
    alors ⊢ Case(ConstructorPattern((n,c)), e)
        -> MatchPat(Cons_Named(n), c', e')
```