# MiniML Spec

Fazazi Zeid Luo Yukai Dibassi Brahima

2023-05-06

# Table des matières

# 1 Grammaire

Voici la grammaire BNF du language MiniML. Les extensions non compatibles avec Ocaml mais compatible avec **Autobill** sont en *Orange* .

## 1.1 Elements Nommés

<Id> ::= [ 'a - z' 'A - Z' '0 - 9' '_' ]+
<ConstructeurId> ::= [ 'A - Z' ] <Id>
<Vartype> ::= ''' [ 'a - z' ] [ '0 - 9' ]*

## 1.2 Programmes

<Prog> ::= | <Expr>
| <Def> ';;' <Prog>

## 1.3 Definitions

<Def> ::= | 'let' <Id> '=' <Expr>
| 'type' <Vartype>* <Id> '=' <NewContructors>

<NewContructors> ::= | <ConstructeurId> 'of' <Type>
| <NewContructors> '|' <NewContructors>
*Debut Extension* | <ConstructeurId> 'of' <Type> 'with' <ParamAssigns>

<ParamAssigns> ::= | <VarType> '=' <ParmExpr>
| <ParamAssigns> ',' <ParamAssigns>
<ParmExpr> ::= | '1'
| <VarType>
| <ParmExpr> '+' <ParmExpr>
*Fin Extension* | <ParmExpr> '*' <ParmExpr>

## 1.4 Expressions

```
<Litteral>      ::=  |  [ '0 - 9' ]+
                     |  [ 'true' | 'false' ]
                     |  '(' ')'

<Expr>          ::=  |  <Litteral>
                     |  <Id>
                     |  <UnaryOperator> <Expr>
                     |  <Expr> <BinaryOperator> <Expr>
                     |  <Expr> <Expr>
                     |  <Expr> ',' <Expr>
                     |  'let' <Id> '=' <Expr> 'in' <Expr>
                     |  'fun' <Id> '->' <Expr>
                     |  'fun' 'rec' <Id> <Id> '->' <Expr>
                     |  <ConstructeurId> <Expr>
                     |  'match' <Expr> 'with' <MatchCase>

<UnaryOperator>  ::=  'not'
<BinaryOperator> ::=  [ 'and' | 'or' | '+' | '-' | '/' | '%' | '*' | '<' | '>' | '=' ]
```

## 1.5 Filtrage et Motifs

```
<MatchCase>     ::=  |  <Pattern> '->' <Expr>
                     |  <MatchCase> '|' <MatchCase>

<Pattern>       ::=  |  <Litteral>
                     |  <Id>
                     |  <Pattern> ',' <Pattern>
                     |  <ConstructeurId>  <Pattern>
```

### 1.6 Types

| | | |
|---|---|---|
| <Type> | ::= | \| <Vartype> |
| | | \| <Id> |
| | | \| <Type> <Type> |
| | | \| <Type> '->' <Type> |
| | | \| <Type> '*' <Type> |

# 2 Semantique de traduction

## 2.1 Notation

$[\![let\ v\ =\ e_1\ in\ e_2]\!]_{Expr} \rightarrow let\ v =\ [\![e_1]\!]\ in\ [\![e_2]\!]$

- $[\![\_]\!]_{Expr} \rightarrow$ est la traduction d'un noeud expr du langage MiniML vers le langage LCBPV
  - A l'interieur des $[\![]\!]$ les elements propre au langage MiniML qui sont traduit vers le langage LCBPV
  - A l'exterieur des $[\![]\!]$ les elements propre au langage LCBPV
- $X_n$ est le n-ième sous-noeud de l'arbre de syntaxe abstrait
  Selon les cas $X$ peut être :
  - $e$ pour les expressions
  - $p$ pour les motifs
  - $t$ pour les types
  - $d$ pour les définitions
    - $dc$ pour les définitions de constructeurs
  - $v$ pour les elements nommées
- $X_1 \dots X_n$ est la liste des sous-noeuds de type $X$ allant de 1 à $n$

## 2.2 Programmes

Un programme MiniML est une suite de taille arbitraire de définitions suivie d'une expression.

$$[\![d_1 \dots d_n\ e]\!]_{Prog} \rightarrow [\![d_1 \dots d_n]\!]\ return\ [\![e]\!]$$

## 2.3 Définitions

On définit l'operation de traduction $[\![\_]\!]_{Def}$ selon les cas de construction des définitions précisées par la règle de grammaire : <Def>

On distingue deux noeuds de types <Def> :

- Les définitions de **Variables Globales**,
- Les définitions de **Types**.

$$\text{(VARDEF)} \quad [\![let\, v\, =\, e]\!]_{Def} \rightarrow let\, v\, =\, [\![e]\!]$$

$$\begin{aligned}\text{(TYPDEF)} \quad & [\![type\ v_1 \ldots v_N\ =\ dc_1 \ldots dc_N]\!]_{Def} \\ & \rightarrow data[\![v_1]\!][\![\ldots v_N]\!] : + = [\![dc_1 \ldots dc_N]\!]\end{aligned}$$

### 2.3.1 Définitions de Constructeurs

On définit l'operation de traduction $[\![\_]\!]_{DefContructors}$ selon les cas de construction précisées par la règle de grammaire : <NewContructors>

On distingue deux noeuds de types <NewContructors> :

- Les définitions de **Constructeurs Classiques** qui sont compatibles avec OcamL
- Les définitions de **Constructeurs Equationnels** qui sont une extension spécifique pour **Autobill** dont la traduction n'est pas encore mise en place

$$\text{(MLCONSTRUCTDEF)} \quad [\![v\ of\ t]\!]_{DefContructors} \rightarrow v[\![t]\!]$$

## 2.4 Types

On définit l'operation de traduction $[\![\_]\!]_{Type}$ selon les cas de construction des types précisées par la règle de grammaire : **<Type>**

On distingue 4 noeuds d'interêt de types $Type$:

- Les **Variables de types**.
- Les **Applications de types**.
- Les **Lambda**
- Les **Tuples**.

$$
\begin{aligned}
\text{(TVAR)}\ & [\![v]\!]_{Type} \rightarrow v \\
\text{(TAPP)}\ & [\![t_1 t_2]\!]_{Type} \rightarrow [\![t_1]\!]\ [\![t_2]\!] \\
\text{(TCLOS)}\ & [\![t_1 \text{->} t_2]\!]_{Type} \rightarrow [\![t_1]\!] \text{->} [\![t_2]\!] \\
\text{(TTUPLE)}\ & [\![t_1 * t_2]\!]_{Type} \rightarrow [\![t_1]\!]\ ,\ [\![t_2]\!]
\end{aligned}
$$

## 2.5 Expressions

On définit l'operation de traduction $[\![\_]\!]_{Expr}$ selon les cas de construction des expressions précisées par la règle de grammaire : **<Expr>**

On distingue 8 noeuds d'Intérêt de types **<Expr>** :

- Les **Tuples**.
- Les **Variables**.
- Les **Appels de fonctions**.
- Les **Fixation**.
- Les **Lambda**.
- Les **Fonctions Recursive**.
- Les **Constructions**.
- Les **Correspondance de motifs**.

$$
\begin{aligned}
\text{(TUPLE)}\ & [\![e_1\ ,\ e_2]\!]_{Expr} \rightarrow \text{Tuple}([\![e_1]\!]\ ,\ [\![e_2]\!]) \\
\text{(VAR)}\ & [\![v]\!]_{Expr} \rightarrow v \\
\text{(LAMBDA)}\ & [\![fun\ v\ \rightarrow\ e]\!]_{Expr} \rightarrow exp(\ get \mid call(v) \text{->} thunk[\![e]\!]) \\
\text{(FUN REC)}\ & [\![fun\ rec\ v_1\ v_2\ \rightarrow\ e]\!]_{Expr} \rightarrow exp(\ rec\ v_1\ is\ get \mid call(v_2) \text{->} thunk[\![e]\!]) \\
\text{(CALL)}\ & [\![e_1 e_2]\!]_{Expr} \rightarrow \{ \\
& \qquad open\ exp\ v_1 = [\![e_1]\!] \\
& \qquad force\ thunk(v_2) = (v_1).call[\![e_2]\!] \\
& \qquad return\ v_2 \\
& \}
\end{aligned}
$$

$$
\begin{array}{rl}
\text{(BIND)} & [\![let\ v\ =\ e_1\ in\ e_2]\!]_{Expr} \rightarrow \{ \\
 & \qquad let\ v = [\![e_1]\!] \\
 & \qquad return\ [\![e_2]\!] \\
 & \} \\
\text{(CONSTRUCT)} & [\![v\ e]\!]_{Expr} \rightarrow v([\![e]\!]) \\
\text{(MATCH)} & [\![match\ e\ with\ c_1 \ldots c_N]\!]_{Expr} \rightarrow match\ [\![e]\!]\ with\ [\![c_1]\!] \ldots [\![c_N]\!]\ end
\end{array}
$$

## 2.6 Motifs et Filtrage

$$Case(p, e) \rightarrow [\![Case(p, e)]\!]_{Case} \rightarrow \alpha$$

On définit la relation $Case$ selon les cas de construction des motif de correspondance. Les cas de construction des motif de correspondance sont donnés par les clauses des règles syntaxiques. Un motif de correspondance est dit traduisible si chacune de ses clauses peut être traduite.

- $p$ est un motif
- $e$ est l'expression qui sera évaluée si le motif est vérifié

$$
\begin{array}{rl}
\text{(PATTAG)} & Case(ConstructorPattern((n, c)), e) \\
 & \quad \rightarrow MatchPatTag(ConsNamed(n), [\![c]\!]_{Case}, [\![e]\!]_{Expr})
\end{array}
$$

$$\text{(PATVAR)} \quad Case(VarPattern(x), e, l) \rightarrow MatchPatVar((x, l), [\![e]\!]_{Expr}, l)$$