

MiniML Grammar Spec

Fazazi Zeid

Luo Yukai

Dibassi Brahima

28 avril, 2023

Table des matières

1	Syntaxe MiniML	3
1.1	Lexing Tokens	3
1.1.1	Separators	3
1.1.2	Mots Clefs	3
1.1.3	Types	3
1.1.4	Operateurs	3
1.1.5	Valeurs Atomiques	3
1.1.6	Identificateur	3
1.2	Grammaire	4
1.2.1	Définitions	4
1.2.2	Expressions	4
1.2.3	Filtrage et Motifs	5
1.2.4	Types	6
2	Semantique de traduction	7
2.1	Programmes	7
2.2	Suites de commandes	7
2.3	Définitions	8
2.4	Types	8
2.5	Littéraux et Expressions	9
2.6	Motifs et Filtrage	10

1 Syntaxe MiniML

1.1 Lexing Tokens

1.1.1 Separators

$\{ \}$ $[\]$ $(\)$ $;$ $,$ $*$ \rightarrow $|$ $=$

1.1.2 Mots Clefs

let fun in match with type of rec if then else

1.1.3 Types

int bool unit

1.1.4 Opérateurs

$+$ $-$ $\%$ $/$ $::$ $\&\&$ $\|$ $*$ $<$ $>$ \leq \geq $=$

1.1.5 Valeurs Atomiques

integer = $(-) ? [0 - 9]$

boolean = $(true \mid false)$

1.1.6 Identificateur

alphanum = $[a - z \ A - Z \ 0 - 9 \ _] *$

basic_ident = $[a - z \ _] alphanum$

vartype = $['a'] [0 .. 9] *$

constructeur_ident = $[A - Z] alphanum$

constructeur_infixes = $[:: \ ,]$

1.2 Grammaire

$$\begin{array}{lcl} \text{Prog} := & | & \text{Def} \\ & | & \text{Expr} \\ & | & \text{Prog} ;; \text{Prog} \end{array}$$

1.2.1 Definitions

$$\begin{array}{lcl} \text{Def} := & | & \text{let basic_ident} = \text{Expr} \quad \text{Def Variable} \\ & | & \text{let basic_ident}_{\text{list}} = \text{Expr} \quad \text{Def Function} \\ & | & \text{let rec basic_ident}_{\text{list}} = \text{Expr} \quad \text{Def FunctionRec} \\ & | & \text{type vartype}_{\text{list}} \text{ basic_ident} = \text{NewConstructor_Case} \quad \text{Def Type} \end{array}$$

$$\begin{array}{lcl} \text{NewConstructor_Case} := & | & \text{constructeur_ident} \\ & | & \text{constructeur_ident of Type} \\ & | & \text{NewConstructor_Case '}' \text{ NewConstructor_Case} \end{array}$$

1.2.2 Expressions

$$\begin{array}{lcl} \text{Litteral} & := & | \text{integer} \\ & & | \text{boolean} \\ & & | () \quad \text{Unit} \\ \\ \text{Expr} & := & | (\text{Expr}) \\ & & | \text{Litteral} \\ & & | \text{basic_ident} \\ & & | \text{UnaryOperator Expr} \\ & & | \text{Expr BinaryOperator Expr} \\ & & | \text{Expr Expr} \quad \text{Call} \\ & & | \text{Expr}; \text{Expr} \quad \text{Sequence} \\ & & | \text{let basic_ident} = \text{Expr in Expr} \quad \text{Binding} \\ & & | \text{fun basic_ident}_{\text{list}} \rightarrow \text{Expr} \quad \text{Lambda} \\ & & | \text{Expr constructeur_infixes Expr} \\ & & | \text{constructeur_ident Expr} \quad \text{Construction} \\ & & | \text{constructeur_ident} \end{array}$$

		<i>let</i> <i>basic_ident</i> _{list} = Expr <i>in</i> Expr	<i>Fonction</i>
		<i>let rec</i> <i>basic_ident</i> <i>basic_ident</i> _{list} = Expr <i>in</i> Expr	<i>Fonction Recursive</i>
		<i>match</i> Expr <i>with</i> Match_Case	
		<i>if</i> Expr <i>then</i> Expr <i>else</i> Expr	
UnaryOperator	:=	<i>not</i>	
BinaryOperator	:=	&&	
		+	
		−	
		/	
		%	
		*	
		<	
		>	
		≤	
		≥	
		=	

1.2.3 Filtrage et Motifs

Match_Case	:=	Pattern → Expr
		Pattern → Expr ' ' Match_Case
Pattern	:=	(Pattern)
		Litteral
		<i>basic_ident</i>
		—
		<i>constructeur_ident</i>
		<i>constructeur_ident</i> Pattern

1.2.4 Types

Type	:=		(Type)	
			<i>int</i>	
			<i>bool</i>	
			<i>unit</i>	
			Type * Type	<i>Tuple Type</i>
			Type → Type	<i>Lambda Type</i>
			<i>vartype</i>	<i>'a</i>
			<i>basic_ident</i>	<i>Defined Type</i>
			Type_{List}	<i>App Type</i>

2 Semantique de traduction

2.1 Programmes

$$\vdash_{Prog} Prog[cs] \rightarrow Prog'(x)$$

$$\begin{array}{l} \text{si } \vdash_{Cmds} cs \rightarrow (g, \omega, v) \\ \text{alors } \vdash_{Cmds} Prog[cs] \rightarrow Prog'(\omega; Do(g, v)) \end{array}$$

2.2 Suites de commandes

$$\vdash_{Cmds} cs \rightarrow (\gamma, \omega, v)$$

$$\begin{array}{l} \text{(VAR DEFS) si } d \in DEF, \\ \text{et si } \vdash_{Cmds} cs \rightarrow (\gamma, \omega, v) \\ \text{et si } \vdash_{Def} d \rightarrow \pi \\ \text{et si } \pi \in GLB \\ \text{alors } \vdash_{Cmds} (Def(d); cs) \rightarrow ((\gamma; \pi), \omega, v) \end{array}$$

$$\begin{array}{l} \text{(TYPE DEFS) si } d \in DEF, \\ \text{et si } \vdash_{Cmds} cs \rightarrow (\gamma, \omega, v) \\ \text{et si } \vdash_{Def} d \rightarrow \pi \\ \text{et si } \pi \in TYPE \\ \text{alors } \vdash_{Cmds} (Def(d); cs) \rightarrow (\gamma, (\omega; \pi), v) \end{array}$$

$$\begin{array}{l} \text{(GLB EXPR) si } b \in EXPR, \\ \text{et si } \vdash_{Cmds} cs \rightarrow (\gamma, \omega, v) \\ \text{et si } \vdash_{Expr} b \rightarrow v' \\ \text{alors } \vdash_{Cmds} (Expr(b), cs) \rightarrow (\gamma, \omega, v') \end{array}$$

2.3 Définitions

$$\vdash_{\text{Def}} d \rightarrow \pi$$

$$\begin{aligned} (\text{VARDEF}) \text{ si } \vdash_{\text{Expr}} e \rightarrow e' \\ \text{alors } \vdash_{\text{Def}} \text{VariableDef}(v, e) \rightarrow \text{GLB}(\text{InsLet}(v, e')) \end{aligned}$$

$$\begin{aligned} (\text{TYPDEF}) \text{ si } \vdash c1 \rightarrow c1' \dots \text{ si } \vdash cN \rightarrow cN' \\ \text{alors } \vdash_{\text{Def}} \text{TypeDef}(n, [t1, \dots, tn], [c1, \dots, cN]) \\ \rightarrow \text{TYPE}(\text{Typ_Def}(n, [t1, \dots, tn], \text{Def_Datatype}[c1', \dots, cN'])) \end{aligned}$$

2.4 Types

$$\vdash_{\text{Type}} t \rightarrow t'$$

$$\begin{aligned} (\text{TINT}) \vdash_{\text{Type}} \text{TypeInt} \rightarrow \text{TypInt} \\ (\text{TBOOL}) \vdash_{\text{Type}} \text{TypeBool} \rightarrow \text{TypBool} \\ (\text{TUNIT}) \vdash_{\text{Type}} \text{TypeUnit} \rightarrow \text{TypUnit} \\ (\text{TDEF}) \vdash_{\text{Type}} \text{TypeDefined}(id) \rightarrow \text{TypVar}(id) \\ (\text{TVAR}) \vdash_{\text{Type}} \text{TypeVar}(id) \rightarrow \text{TypVar}(id) \\ \\ (\text{TTUPLE}) \text{ si } \vdash_{\text{Type}} t1 \rightarrow t_1, \dots \text{ et } \vdash_{\text{Type}} tN \rightarrow t_N \\ \text{alors } \vdash_{\text{Type}} \text{TypeTuple}([t1, \dots, tN]) \rightarrow \text{TypTuple}[t_1, \dots, t_N] \\ \\ (\text{TCONS}) \text{ si } \vdash_{\text{Type}} t \rightarrow t' \\ \text{ si } \vdash_{\text{Type}} p1 \rightarrow p_1, \dots \text{ et } \vdash_{\text{Type}} pN \rightarrow p_N \\ \text{alors } \vdash_{\text{Type}} \text{TypeConstructor}(t, [p1, \dots, pN]) \rightarrow \text{TypApp}(t, [p_1, \dots, p_N]) \\ \\ (\text{TLAMB}) \text{ si } \vdash_{\text{Type}} a \rightarrow a' \\ \text{ et si } \vdash_{\text{Type}} \text{ret} \rightarrow \text{ret}' \\ \text{alors } \vdash_{\text{Type}} \text{TypeLambda}(a, \text{ret}) \\ \rightarrow \text{TypClosure}(\text{Exp}, (\text{TypFun}(\text{TypThunk}(\text{ret}'), a')))) \end{aligned}$$

2.5 Litteraux et Expressions

$$\vdash_{\text{Expr}} e \rightarrow e'$$

(INT) si $i \in \text{NUM}$

alors $\vdash_{\text{Expr}} \text{Integer}(i) \rightarrow \text{ExprInt}(i)$

(TRUE) $\vdash_{\text{Expr}} \text{Boolean}(\text{true}) \rightarrow \text{ExprConstructor}(\text{True}, [])$

(FALSE) $\vdash_{\text{Expr}} \text{Boolean}(\text{false}) \rightarrow \text{ExprConstructor}(\text{False}, [])$

(UNIT) $\vdash_{\text{Expr}} \text{Unit} \rightarrow \text{ExprConstructor}(\text{Unit}, [])$

(TUPLE) si $\vdash_{\text{Expr}} e1 \rightarrow e_1, \dots$ si $\vdash_{\text{Expr}} eN \rightarrow e_N$

alors $\vdash_{\text{Expr}} \text{Tuple}([e1, \dots, eN]) \rightarrow \text{ExprConstructor}(\text{Tuple}, [e_1, \dots, e_N])$

(UNARY1) si $\vdash_{\text{Expr}} a \rightarrow a'$

alors $\vdash_{\text{Expr}} \text{CallUnary}(\text{op}, [a]) \rightarrow \text{ExprMonPrim}(\text{op}, a')$

(UNARY0) si $\vdash_{\text{Expr}} \text{Lambda}(a, \text{CallUnary}(\text{op}, [a])) \rightarrow \omega$

alors $\vdash_{\text{Expr}} \text{CallUnary}(\text{op}, []) \rightarrow \omega$

(BINARY2) si $\vdash_{\text{Expr}} a1 \rightarrow a_1$ et $\vdash_{\text{Expr}} a2 \rightarrow a_2$

alors $\vdash_{\text{Expr}} \text{CallBinary}(\text{op}, [a1 : a2]) \rightarrow \text{ExprBinPrim}(\text{op}, a_1, a_2)$

(BINARY1) si $\vdash_{\text{Expr}} \text{Lambda}(b, \text{CallUnary}(\text{op}, [a, b])) \rightarrow \omega$

alors $\vdash_{\text{Expr}} \text{CallBinary}(\text{op}, [a]) \rightarrow \omega$

(BINARY0) si $\vdash_{\text{Expr}} \text{Lambda}(a, \text{Lambda}(b, \text{CallUnary}(\text{op}, [a, b]))) \rightarrow \omega$

alors $\vdash_{\text{Expr}} \text{CallBinary}(\text{op}, []) \rightarrow \omega$

(CONSTR) si $\vdash_{\text{Expr}} e \rightarrow e'$

alors $\vdash_{\text{Expr}} \text{Construct}(c, e) \rightarrow \text{ExprConstructor}(\text{ConsNamed}(c), e')$

(BIND) si $\vdash_{\text{Expr}} i \rightarrow i'$

et si $\vdash_{\text{Expr}} c \rightarrow c'$

alors $\vdash_{\text{Expr}} \text{Binding}(v, i, c) \rightarrow \text{ExprBlock}(\text{Blk}([\text{InsLet}(v, i')], c'))$

(MATCH) si $\vdash_{\text{Expr}} m \rightarrow m'$

si $m1 \in \text{CASE} \dots$ et $mN \in \text{CASE}$

si $\vdash_{\text{Case}} m1 \rightarrow m_1, \dots$ et $\vdash_{\text{Case}} mN \rightarrow m_N$

alors $\vdash_{\text{Expr}} \text{Match}(m, [m1, \dots, mN]) \rightarrow \text{ExprMatch}(m', [m_1, \dots, m_N])$

(SEQ) si $\vdash_{Expr} e1 \rightarrow InsLet(x1, e1), \dots$ et $\vdash_{Expr} eN - 1 \rightarrow InsLet(xN - 1, e_{N-1})$
 et si $\vdash_{Expr} eN \rightarrow e_N$
 alors $\vdash_{Expr} Sequence([e1, \dots, eN])$
 $\rightarrow ExprBlock(Blk([InsLet(x1, e1); \dots; InsLet(xN - 1, e_{N-1})], e_N))$

(CALL) si $\vdash_{Expr} a \rightarrow a'$
 et si $\vdash_{Expr} f \rightarrow f'$
 alors $\vdash_{Expr} Call(f, a)$
 $\rightarrow ExprBlock(Blk([InsOpen(Exp, f'), InsForce(ExprMethod(Call, [a'])))])$

(LAMBDA) si $\vdash_{Expr} a \rightarrow a'$
 et si $\vdash_{Expr} b \rightarrow b'$
 alors $\vdash_{Expr} Lambda(a, b)$
 $\rightarrow ExprClosure(Exp, ExprGet([GetPatTag(Call, [a'], ExprThunk(b'))]))$

(REC) si $\vdash_{Expr} a \rightarrow a'$
 si $\vdash_{Expr} v \rightarrow v'$ et si $\vdash_{Expr} b \rightarrow b'$
 alors $\vdash_{Expr} FunctionRec(v, a, b)$
 $\rightarrow ExprClosure(Exp, ExprRec(v', ExprGet([GetPatTag(Call, [a'], ExprThunk(b'))])))$

2.6 Motifs et Filtrage

$$\vdash_{Case} Case(p, e) \rightarrow \alpha$$

(INTPAT) si $l \in NUM$ et $\vdash_{Expr} e \rightarrow e'$
 alors $\vdash_{Case} Case(LitteralPattern(l), e) \rightarrow MatchPatTag(IntLitt l, [], e')$
 (BOOLPAT) si $l = Boolean(_)$,
 si $\vdash l \rightarrow l'$ et $\vdash_{Expr} e \rightarrow e'$
 alors $\vdash_{Case} Case(LitteralPattern(l), e) \rightarrow MatchPatTag(l', [], e')$
 (UNITPAT) si $l = Unit$ et $\vdash_{Expr} e \rightarrow e'$
 alors $\vdash_{Case} Case(LitteralPattern(l), e) \rightarrow MatchPatTag(Unit, [], e')$

(TUPAT) si $p1 \in CASE, \dots$ si $pN \in CASE$

si $\vdash p1 \rightarrow p_1, \dots$, si $\vdash pN \rightarrow p_N$

et $\vdash_{Expr} e \rightarrow e'$

alors $\vdash_{Case} Case(TuplePattern([p1, \dots, pN]), e) \rightarrow MatchPatTag(Tuple, [p1, \dots, pN], e')$

(CONSPAT) si $c \in CASE$

si $\vdash c \rightarrow c'$

et $\vdash_{Expr} e \rightarrow e'$

alors $\vdash_{Case} Case(ConstructorPattern((n, c)), e) \rightarrow MatchPatTag(ConsNamed(n), c', e')$

(VARPAT) si $\vdash_{Expr} e \rightarrow e'$

alors $\vdash_{Case} Case(VarPattern(x), e, l) \rightarrow MatchPatVar((x, l), e', l)$

(WILDPAT) si $\vdash_{Expr} e \rightarrow e'$

alors $\vdash_{Case} Case(WildcardPattern(), e, l) \rightarrow MatchPatVar((n, l), e', l)$