
Rapport sur le transpilateur MiniML vers C

Name
23 avril 2018

TABLE DES MATIÈRES

1	Analyses	2
2	Analyse lexicale	3
2.1	Tokens et lexèmes	3
2.2	Commentaires	3
3	Analyse syntaxique	4
3.1	Structure d'AST	4
3.2	Bison et construction de l'AST	5
4	Évaluation d'une expression	6
5	Inférence de types	7
6	Fonctions natives	8
7	Génération de code	9
8	Annexe 1 : Grammaire	10
8.1	Grammaire BNF	10
8.2	Terminaux	11

1 ANALYSES

La compilation est divisée en quatre étapes :

1. l'analyse lexicale (transformation du texte en flot de lexèmes)
2. l'analyse syntaxique (transformation du flot de lexèmes en arbre syntaxique abstrait)
3. l'analyse sémantique (vérification de la validité des expressions)
4. l'exécution ou la génération du code

Chaque section traitera un type d'analyse.

NB : Le fichier README.md contient des informations plus pratiques concernant la compilation et l'utilisation du compilateur.

2 ANALYSE LEXICALE

2.1 TOKENS ET LEXÈMES

L'implémentation présente permet 6 types de tokens différents (association d'un lexème avec une valeur éventuelle).

LES LITÉRAUX : Les nombres entiers, les nombres flottants et les chaînes de caractères ont leur propre catégorie. Les lexèmes `true` et `false` sont associés dans une même catégorie aux valeurs 1 et 0 respectivement.

LES IDENTIFIANTS : Les identifiants et les opérateurs appartiennent à la même catégorie des noms. En effet, par la nature fonctionnelle des langages ML, les opérateurs sont considérés comme des fonctions infixes, et donc des identifiants spéciaux.

Une table des noms est mise en place pour associer chaque identifiant à un entier unique afin de simplifier la comparaison d'identifiants. Cette table est implémentée par une table d'association bijective.

Chaque nouveau nom est associé à un entier unique strictement supérieur à 0. L'entier 0 est utilisé lorsqu'un identifiant ne peut pas exister (p.ex. une fonction anonyme).

LES AUTRES : Les autres mots-clés et lexèmes restants ne portant pas de valeur particulière, ils forment la dernière catégorie de tokens. Les caractères blancs sont ignorés par l'analyseur, et les caractères invalides sont passés à l'analyseur syntaxique tels quels, où ils provoqueront une erreur de syntaxe.

2.2 COMMENTAIRES

L'analyseur lexical ignore les commentaires, lesquels peuvent être imbriqués.

Le contexte devient `comment` en lisant un commentaire. Un compteur est incrémenté et décrementé en cas d'ouverture et de fermeture de commentaire respectivement. Quand le compteur est nul, le contexte redevient le contexte initial.

Les spécificités dans l'implémentation de cette règle permettent d'optimiser la lecture quand les caractères des balises de commentaire apparaissent dans son contenu.

REMARQUE : Il peut être plus efficace de créer une fonction `C` pour analyser les commentaires. En effet, en pratique, le tampon de lecture de Flex a une capacité limitée, et si la longueur du commentaire dépasse cette capacité, le comportement du programme peut être indéterminé.

3 ANALYSE SYNTAXIQUE

3.1 STRUCTURE D'AST

La structure d'AST utilisée contient un champ d'énumération de type et une union anonyme selon ce champ.

Type	Champs
unit	-
integer	value: Int
float	value: Float
boolean	value: Bool
string	value: String
variable	name: Int
block	expr: AST
seq	seq: Seq[AST]
application	function: AST, args: List[AST]
let	names: List[Name], rec: Bool, params: List[Name], expr: AST, block: AST
if	cond: AST, if: AST, else: AST
tuple	elems: List[AST]

Voici une explication détaillée pour la construction let :

Le champ names est une liste des noms liés par cette expression. Cette liste est vide si et seulement si l'expression est une déclaration de fonction anonyme avec function. La liste contient plus d'un nom dans le cas où l'expression définit un tuple de noms.

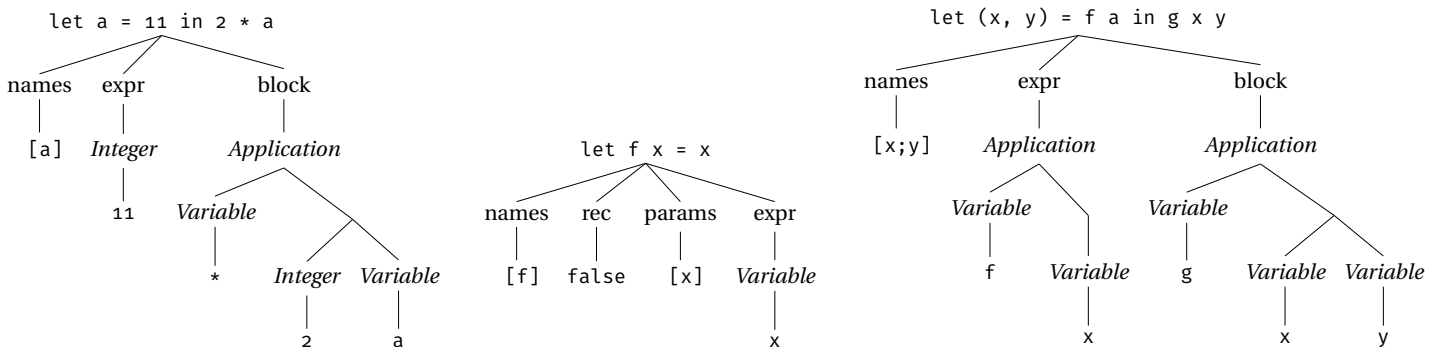
Le champ rec est une valeur booléenne. Elle n'a de sens qu'en déclaration de fonction. Le champ vaut true si et seulement si la définition est récursive.

Le champ params est la liste des noms des paramètres de la fonction déclarée. Cette liste est vide si et seulement si s'agit d'une déclaration de variable.

Le champ expr est l'expression du corps de la déclaration.

Le champ block est l'expression suivant cette définition. Il correspond au bloc in dans la construction let-in.

Et quelques exemples :



On remarquera que l'opération $2 * a$ est équivalente à l'application $(*) 2 a$, comme décrit avant.

3.2 BISON ET CONSTRUCTION DE L'AST

La grammaire du langage est décrite en annexe dans le méta-langage BNF. Les modifications suivantes ont été effectuées pour l'adapter au générateur Bison.

La récupération sur erreur s'effectue par instruction : en cas d'erreur de syntaxe, l'instruction en cours d'analyse est ignorée et celle-ci reprend à la fin de l'instruction erronée.

La grammaire décrite contient également des conflits décalage/réduction. Ceux-ci ont été résolus en appliquant des règles de priorité et d'associativité sur certaines productions et terminaux.

Les priorités des opérations arithmétiques sont définies explicitement, en créant des règles intermédiaires pour former `simple-expr`, en suivant l'ordre conventionnel des opérations.

Voici, pour illustrer, la réduction de l'expression `1 = 2 + 3 * 6` :

$$\begin{aligned}
 \langle expr \rangle &\Rightarrow \langle or \rangle \Rightarrow \langle and \rangle \Rightarrow \langle equ \rangle \\
 &\Rightarrow \langle rel \rangle = \langle rel \rangle \\
 &\Rightarrow \langle add \rangle = \langle add \rangle \\
 &\Rightarrow \langle mul \rangle = (\langle add \rangle + \langle mul \rangle) \\
 &\Rightarrow \langle un \rangle = (\langle mul \rangle + (\langle un \rangle * \langle un \rangle)) \\
 &\Rightarrow 1 = (\langle un \rangle + (3 * 6)) \\
 &\Rightarrow 1 = (2 + (3 * 6)).
 \end{aligned}$$

La priorité des non-opérateurs est définie dans le fichier source Bison, grâce aux instructions de priorité/associativité.

Le tableau de priorité/associativité est le suivant, par ordre de priorité de la plus élevée à la plus faible :

Production ou symbole	Associativité
!	droite
application	gauche
+ - +. -. (préfixe)	droite
* / *. /.	gauche
+ - +. -.	gauche
= <> < <= > >=	gauche
&&	droite
	droite
,	-
else	droite
if	-
;	droite
let function	-

4 ÉVALUATION D'UNE EXPRESSION

Le compilateur peut également évaluer directement l'entrée sans produire de fichier C. Le fonctionnement du code généré et du mode interpréteur est strictement identique à partir de l'évaluation.

L'évaluation d'une expression se fait en deux étapes distinctes : il faut évaluer le type de l'expression, puis sa valeur. Le typage, étant caractéristique des langages ML, sera détaillé dans une section suivante.

On rappelle qu'un programme MiniML est une suite d'instructions, où une instruction est soit une expression, soit une déclaration globale.

L'environnement d'exécution est défini par une liste de définitions. Une définition porte un identifiant et un type, une valeur, ou les deux.

Les structure de type et de valeur sont définies, comme les AST, avec une énumération et une union anonyme. Leur contenu est décrit dans le tableau ci-dessous.

Énumération	Type	Valeur
unit	-	-
int	-	value: Int
float	-	value: Float
bool	-	value: Bool
string	-	value: String
nativefunc	descriptor: NativeDescriptor	
function	-	env: Environment, params: List[Name], body: AST
tuple	types: List[Type]	elems: List[Value]
polymorph	id: Int	-
error	-	-

L'évaluation d'une expression s'effectue récursivement sur l'AST avec un pattern visiteur. Les détails de l'implémentation sont dans le sous-dossier eval des fichiers sources.

5 INFÉRENCE DE TYPES

Le système de typage utilise le système de typage Hindley-Milner pour inférer le type d'une expression.

L'inférence du type d'une expression s'effectue en quatre étapes : annotation, génération des contraintes, substitution et unification, application.

ANNOTATION L'AST est annoté d'un type (définition des structures dans la section précédente), et ce récursivement. Une expression est annotée d'un type concret si possible, sinon un type générique polymorphe.

Exemple : `let x = f 1 2.;;` est annoté en `<'a>let x = <'b>f <int>1 <float>2.;;`

Exemple : `let f y = y + 1;;` est annoté en `<'a>let f y = <int->int>(+) <'b>y <int>1;;`

GÉNÉRATION DES CONTRAINTES L'AST annoté est parcouru une deuxième fois, afin de collecter des contraintes sur les différentes annotations de type.

Exemple : `<'a>let x = <'b>f <int>1 <float>2.;;` génère les contraintes suivantes :

`'b \iff int -> float -> 'c`

`'a \iff 'c`

SUBSTITUTION ET UNIFICATION Les contraintes sont transformées en listes de substitutions concrètes pour les types polymorphes.

Exemple : La contrainte `'a -> 'b \iff float -> int` génère les substitutions suivantes :

`('a := float)`

`('b := int)`

APPLICATION Cette étape consiste simplement à parcourir l'AST annoté d'origine et appliquer les substitutions générées aux types annotés.

Le type de l'expression racine correspond au type final.

Les détails de l'implémentation sont dans le sous-dossier `infer` du code source.

6 FONCTIONS NATIVES

En raison de la nature fonctionnelle des langages ML, il est indispensable de disposer de certaines fonctions nativement, et plus que cette implémentation considère les opérateurs infixes comme des fonctions.

Le système a été conçu afin de simplifier l'ajout et la modification de nouvelles fonctions.

Une liste de descripteurs est construite au lancement, avec :

- l'indice du nom de la fonction dans la table des noms;
- un pointeur vers la fonction C correspondant à la fonction ML;
- la liste des types des arguments de la fonction (polymorphes si nécessaire);
- le type de retour de la fonction (polymorphe si nécessaire);
- une liste des contraintes éventuelles à appliquer aux types (voir section sur l'inférence)

A l'étape d'annotation, les descripteurs de fonctions natives sont concrétisés en remplaçant les types par des types concrets (ou polymorphes si nécessaire), et de même pour la génération des contraintes.

Cette approche pose un certain problème pour les fonctions polymorphes, comme `compare`. Cette fonction est traitée comme une fonction de type `'a -> 'a -> int` par l'analyse sémantique, mais la valeur de l'identifiant `compare` est `'a -> 'b -> int`.

Voici la liste des fonctions natives implémentées :

Identifiant	Type fonctionnel
<code>+</code>	<code>int -> int -> int</code>
<code>-</code>	<code>int -> int -> int</code>
<code>*</code>	<code>int -> int -> int</code>
<code>/</code>	<code>int -> int -> int</code>
<code>+. </code>	<code>float -> float -> float</code>
<code>-. </code>	<code>float -> float -> float</code>
<code>*. </code>	<code>float -> float -> float</code>
<code>/. </code>	<code>float -> float -> float</code>
<code>compare</code>	<code>'a -> 'a -> int</code>
<code>=</code>	<code>'a -> 'a -> bool</code>
<code><</code>	<code>'a -> 'a -> bool</code>
<code>></code>	<code>'a -> 'a -> bool</code>
<code><=</code>	<code>'a -> 'a -> bool</code>
<code>>=</code>	<code>'a -> 'a -> bool</code>
<code>&&</code>	<code>bool -> bool -> bool</code>
<code> </code>	<code>bool -> bool -> bool</code>
<code>print_string</code>	<code>string -> unit</code>
<code>print_int</code>	<code>int -> unit</code>
<code>print_float</code>	<code>float -> unit</code>
<code>print_bool</code>	<code>bool -> unit</code>
<code>int_of_float</code>	<code>float -> int</code>
<code>float_of_int</code>	<code>int -> float</code>

7 GÉNÉRATION DE CODE

Le compilateur possède deux modes de sortie : interprète et compilateur.

En mode interprète, le typage et l'évaluation s'effectuent en même temps, tandis que seulement l'inférence de type est exécutée en mode compilateur. Le but est ici d'effectuer une vérification statique du code ML d'entrée avant de commencer à générer le code C de sortie.

La génération de code en elle-même est très rudimentaire : tous les fichiers sources d'origine sont inclus dans le bon ordre en utilisant le préprocesseur C, et seule la fonction `main` varie selon le code d'entrée.

Le compilateur génère un AST statique correspondant à l'expression entière du code ML, puis exécute les mêmes instructions que le mode interprète.

Les sources communes sont incluses directement dans le code, par un fichier `.xxd`, qui contient une définition statique du contenu binaire du fichier source (voir règles `.xxd` et `.pre` du Makefile).

Les codes sources du compilateur utilisent la bibliothèque `libcalg`, ainsi partager ces sources signifie que le programme cible doit être lié contre `libcalg`.

8 ANNEXE 1 : GRAMMAIRE

8.1 GRAMMAIRE BNF

$\langle \text{program} \rangle$	$::= \langle \text{program} \rangle \langle \text{instruction} \rangle ';;'$ $\langle \text{program} \rangle ';;'$ $\langle \text{instruction} \rangle ';;'$ $';;'$
$\langle \text{instruction} \rangle$	$::= \langle \text{expr} \rangle \mid \langle \text{let-binding} \rangle$
$\langle \text{expr} \rangle$	$::= \langle \text{simple-expr} \rangle$ $\langle \text{atom} \rangle \langle \text{atom-list} \rangle$ $\langle \text{let-binding} \rangle \text{'in'} \langle \text{expr} \rangle$ $\text{'function'} \langle \text{parameter-list} \rangle \text{'->'} \langle \text{expr} \rangle$ $\langle \text{if-expr} \rangle$
$\langle \text{simple-expr} \rangle$	$::= \langle \text{simple-expr} \rangle \text{' '} \langle \text{simple-expr} \rangle$ $\langle \text{simple-expr} \rangle \text{'\&\&'} \langle \text{simple-expr} \rangle$ $\langle \text{simple-expr} \rangle \langle \text{equ-op} \rangle \langle \text{simple-expr} \rangle$ $\langle \text{simple-expr} \rangle \langle \text{rel-op} \rangle \langle \text{simple-expr} \rangle$ $\langle \text{simple-expr} \rangle \langle \text{add-op} \rangle \langle \text{simple-expr} \rangle$ $\langle \text{simple-expr} \rangle \langle \text{mul-op} \rangle \langle \text{simple-expr} \rangle$ $\langle \text{unary-op} \rangle \langle \text{simple-expr} \rangle$ $\langle \text{simple-expr} \rangle$
$\langle \text{if-expr} \rangle$	$::= \text{'if'} \langle \text{expr} \rangle \text{'then'} \langle \text{expr} \rangle$ $\text{'if'} \langle \text{expr} \rangle \text{'then'} \langle \text{expr} \rangle \text{'else'} \langle \text{expr} \rangle$
$\langle \text{let-binding} \rangle$	$::= \text{'let'} \langle \text{let-pattern} \rangle \text{'='} \langle \text{expr} \rangle$ $\text{'let'} \text{'rec'} \langle \text{let-pattern} \rangle \text{'='} \langle \text{expr} \rangle$
$\langle \text{let-pattern} \rangle$	$::= \langle \text{name} \rangle$ $\text{'('} \langle \text{operator} \rangle \text{'}'$ $\text{'('} \langle \text{tuple-name-list} \rangle \text{'}'$
$\langle \text{atom} \rangle$	$::= \langle \text{integer} \rangle \mid \langle \text{float} \rangle$ $\langle \text{boolean} \rangle \mid \langle \text{string} \rangle$ $\langle \text{name} \rangle$ $\text{'('} \text{'}'$ $\text{'('} \langle \text{operator} \rangle \text{'}'$ $\text{'('} \langle \text{expr-list} \rangle \text{'}'$ $\text{'begin'} \text{'end'}$ $\text{'begin'} \langle \text{expr-list} \rangle \text{'end'}$ $\text{'('} \langle \text{tuple-expr-list} \rangle \text{'}'$
$\langle \text{atom-list} \rangle$	$::= \langle \text{atom} \rangle$ $\langle \text{atom-list} \rangle \langle \text{atom} \rangle$
$\langle \text{expr-list} \rangle$	$::= \langle \text{expr} \rangle$ $\langle \text{expr-list} \rangle \langle \text{expr} \rangle$

$\langle parameter-list \rangle$	$::= \langle name \rangle$ $\langle parameter-list \rangle \langle name \rangle$
$\langle tuple-name-list \rangle$	$::= \langle name \rangle$ $\langle tuple-name-list \rangle \text{ ',' } \langle name \rangle$
$\langle tuple-expr-list \rangle$	$::= \langle expr \rangle$ $\langle tuple-expr-list \rangle \text{ ',' } \langle expr \rangle$
$\langle equ-op \rangle$	$::= \text{'='} \text{ ' < >'}$
$\langle rel-op \rangle$	$::= \text{'>' '>=' '<' '<='}$
$\langle add-op \rangle$	$::= \text{'+' '-' '+.' '-.'}$
$\langle mul-op \rangle$	$::= \text{'*' '/' '*.' '/.'}$
$\langle unary-op \rangle$	$::= \text{'!' '+' '-' '+.' '-.'}$

8.2 TERMINAUX

INTEGER : les nombres entiers.

FLOAT : les nombres flottants, avec décimales et/ou exposant. (ex : 5., 3.1415, 9e+5 ou 1.2e-67)

BOOLEAN : les valeurs de vérité true et false.

STRING : les chaînes littérales, avec les échappements suivants : le guillemet (\"), l'apostrophe (\'), le retour à la ligne (\n), le retour arrière (\b), la tabulation (\t), le retour chariot (\r), l'espace (\).

NAME : les identifiants, commencent par une lettre et constitués de lettres, chiffres et/ou tiret du bas.

OPERATOR : les terminaux des opérateurs, @, || et les autres terminaux qui apparaissent dans les productions *-op.