Rapport sur le transpilateur MiniML vers C

Name
21 février 2018

TABLE DES MATIÈRES

1 Grammaire MiniML et Bison

La grammaire employée pour ce langage est décrite en annexe grâce au méta-langage BNF. Certaines adaptations ont été effectuées afin de rendre la grammaire compatible avec Bison.

La récupération sur erreur s'effectue par instruction, c'est-à-dire qu'une instruction est ignorée si une erreur d'analyse survient, ainsi que les symboles restants jusqu'à une nouvelle instruction.

La grammaire en annexe contient des conflits décalage/réduction. Ces conflits ont été résolus en appliquant des règles de priorité. La priorité des opérateurs est définie explicitement, en imbriquant des règles intermédiaires pour former `simple-expr`. La priorité des non-opérateurs est définie dans le fichier source Bison, grâce aux instructions de priorité/associativé.

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`	-

2 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 programme final doit être compilé et lié avec libcalg (à compiler à partir des sources). Il s'agit d'une bibliothèque de structures de données, utilisée dans le but de simplifier les implémentations, i.e. le code relatif aux structures des données de base est abstrait par libcalg.

3 Analyse lexicale

3.1 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.

3.2 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.

4 Analyse syntaxique

4.1 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`
`list`	`list: List[AST]`
`application`	`function: AST, args: List[AST]`
`let`	`names: List[AST], rec: Bool, params: List[AST], expr: AST, block: AST`
`if`	`cond: AST, if: AST, else: AST`
`tuple`	`list: 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.

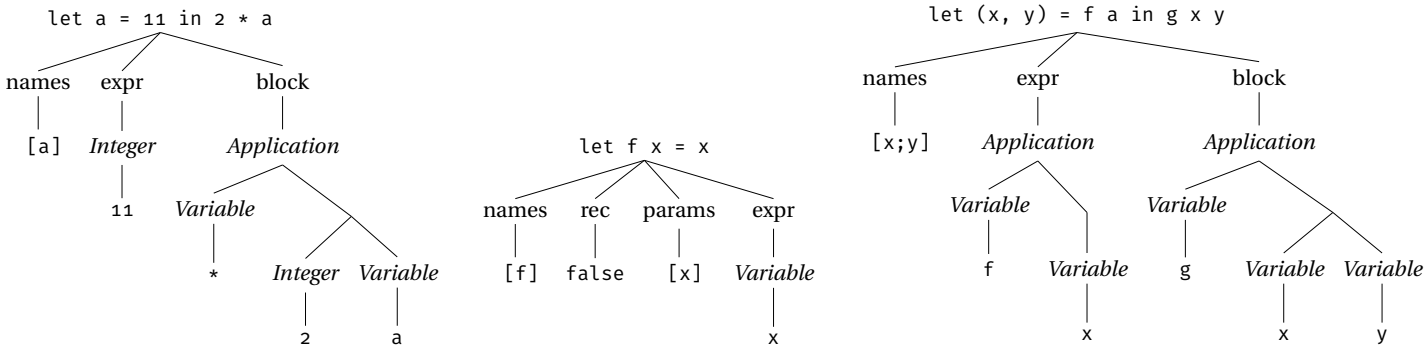
Le champ `rec` est une valeur booléenne. Elle n'a de sens qu'en déclaration de fonction. Le champ vaut `true` si et seulement si la définition est récursive.

Le champ `params` est la liste des noms des paramètres de la fonction déclarée. Cette liste est vide si et seulement s'il 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.

5 Annexe 1 : Grammaire

5.1 Grammaire BNF

⟨*program*⟩ ::= ⟨*program*⟩ ⟨*instruction*⟩ ';;'
| ⟨*program*⟩ ';;'
| ⟨*instruction*⟩ ';;'
| ';;'

⟨*instruction*⟩ ::= ⟨*expr*⟩ | ⟨*let-binding*⟩

⟨*expr*⟩ ::= ⟨*simple-expr*⟩
| ⟨*atom*⟩ ⟨*atom-list*⟩
| ⟨*let-binding*⟩ 'in' ⟨*expr*⟩
| 'function' ⟨*parameter-list*⟩ '->' ⟨*expr*⟩
| ⟨*if-expr*⟩

⟨*simple-expr*⟩ ::= ⟨*simple-expr*⟩ '||' ⟨*simple-expr*⟩
| ⟨*simple-expr*⟩ '&&' ⟨*simple-expr*⟩
| ⟨*simple-expr*⟩ ⟨*equ-op*⟩ ⟨*simple-expr*⟩
| ⟨*simple-expr*⟩ ⟨*rel-op*⟩ ⟨*simple-expr*⟩
| ⟨*simple-expr*⟩ ⟨*add-op*⟩ ⟨*simple-expr*⟩
| ⟨*simple-expr*⟩ ⟨*mul-op*⟩ ⟨*simple-expr*⟩
| ⟨*unary-op*⟩ ⟨*simple-expr*⟩
| ⟨*simple-expr*⟩

⟨*if-expr*⟩ ::= 'if' ⟨*expr*⟩ 'then' ⟨*expr*⟩
| 'if' ⟨*expr*⟩ 'then' ⟨*expr*⟩ 'else' ⟨*expr*⟩

⟨*let-binding*⟩ ::= 'let' ⟨*let-pattern*⟩ '=' ⟨*expr*⟩
| 'let' 'rec' ⟨*let-pattern*⟩ '=' ⟨*expr*⟩

⟨*let-pattern*⟩ ::= ⟨*name*⟩
| '(' ⟨*operator*⟩ ')'
| '(' ⟨*tuple-name-list*⟩ ')'

⟨*atom*⟩ ::= ⟨*integer*⟩ | ⟨*float*⟩
| ⟨*boolean*⟩ | ⟨*string*⟩
| ⟨*name*⟩
| '(' ')'
| '(' ⟨*operator*⟩ ')'
| '(' ⟨*expr-list*⟩ ')'
| 'begin' 'end'
| 'begin' ⟨*expr-list*⟩ 'end'
| '(' ⟨*tuple-expr-list*⟩ ')'

⟨*atom-list*⟩ ::= ⟨*atom*⟩
| ⟨*atom-list*⟩ ⟨*atom*⟩

⟨*expr-list*⟩ ::= ⟨*expr*⟩
| ⟨*expr-list*⟩ ⟨*expr*⟩

⟨*parameter-list*⟩ ::= ⟨*name*⟩
| ⟨*parameter-list*⟩ ⟨*name*⟩

⟨*tuple-name-list*⟩ ::= ⟨*name*⟩
| ⟨*tuple-name-list*⟩ ‘`,`’ ⟨*name*⟩

⟨*tuple-expr-list*⟩ ::= ⟨*expr*⟩
| ⟨*tuple-expr-list*⟩ ‘`,`’ ⟨*expr*⟩

⟨*equ-op*⟩ ::= ‘`=`’ | ‘`<>`’

⟨*rel-op*⟩ ::= ‘`>`’ | ‘`>=`’ | ‘`<`’ | ‘`<=`’

⟨*add-op*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`+.`’ | ‘`-.`’

⟨*mul-op*⟩ ::= ‘`*`’ | ‘`/`’ | ‘`*.`’ | ‘`/.`’

⟨*unary-op*⟩ := ‘`!`’ | ‘`+`’ | ‘`-`’ | ‘`+.`’ | ‘`-.`’

5.2 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 litérales, 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`.