Ali Baydoun
Romain Bossy
Elie Carrot
Jorane Guidetti
Seryozha Hakobyan
Noémie Pellegrin

Université Grenoble Alpes

Projet — Master 1 Informatique

Compilateur pour le language MinCaml
Documentation technique

LesPerdus – Janvier 2024

Table des matières

1 Présentation générale du projet

Cette partie et les suivantes risquent de répéter nombre d'éléments présents dans la documentation fournie et particulièrement dans ce papier : `https://esumii.github.io/min-caml/paper.pdf`.

1.1 Outils nécessaires

Afin de lancer le compilateur, plusieurs librairies et programmes sont nécessaires et doivent être installés. Ces derniers sont :

- Le langage **OCaml** (`https://v2.ocaml.org/docs/install.fr.html`) afin de pouvoir le compiler. Il est conseillé d'utiliser le gestionnaire de paquets **Opam** (`https://opam.ocaml.org/`) pour installer facilement Ocaml et les autres packages nécessaires au projet ;
- Le package **ocamlbuild** (`https://opam.ocaml.org/packages/ocamlbuild/`) qui contient un ensemble de librairies utilisées pour l'exécution du compilateur ;
- Le programme **qemu-user** (`https://www.qemu.org/download/`) et l'ensemble d'outils GNU ARM **gcc-arm-linux-gnueabi** (`https ://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads/11-2-2022-02`) afin de transformer un fichier assembleur en un fichier exécutable. Ce dernier est lancé par QEMU et donne le résultat de l'exécution du programme MinCaml écrit initialement.

1.2 Architecture globale du compilateur

N.B. Une étape de `let`-reduction a été rajoutée avant le passage en closures car certaines optimisations peuvent engendrer des « `let` » imbriqués et l'étape de conversion en closures ne gère pas ce cas de figure.

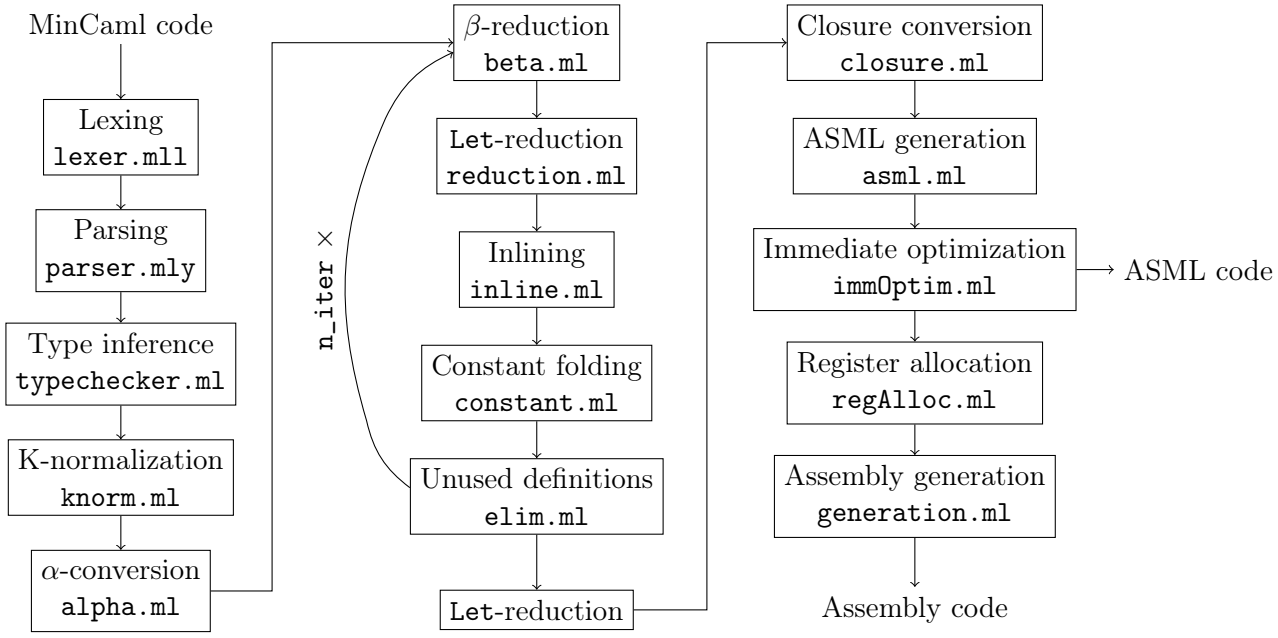

FIGURE 1 – Architecture générale du compilateur

1.3 Organisation de la structure du code

L'ensemble des fichiers constituant le projet mincaml-compiler sont répartis dans plusieurs dossiers comme suit :

ARM contient la librairie standard de mincaml pour la compilation de fichiers assembleurs (ARM) ;
asml présente des exemples de fichiers écrits en code intermédiaire asml ;
mincaml possède plusieurs exemples de programmes écrits en langage mincaml ;
ocaml regroupe tous les fichiers nécessaires au fonctionnement du compilateur construit ;
scripts contient tous les scripts de tests automatisés ;
tests possède tous les programmes mincaml sur lesquels les tests sont exécutés ;
tools détient l'interpréteur asml permettant d'exécuter des fichiers asml.

2 Présentation du code et éléments théoriques

2.1 Arbre syntaxique abstrait en MinCaml

La définition que l'on redonne ici est présente dans le fichier `syntax.ml`. L'arbre syntaxique d'un programme MinCaml est défini par :

$$
\begin{aligned}
e ::= \; & () \mid \text{bool} \mid \text{int} \mid \text{float} \mid \neg e \mid -e \mid e_1 + e_2 \mid e_2 - e_2 \\
& \mid e_1 +. \; e_2 \mid e_1 -. \; e_2 \mid e_1 \; *. \; e_2 \mid e_1 \; /. \; e_2 \\
& \mid e_1 = e_2 \mid e_1 \leq e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
& \mid \text{let } x = e_1 \text{ in } e_2 \mid x \\
& \mid \text{let rec } f \; x_1 \; ... \; x_n = e_1 \text{ in } e_2 \mid e \; e_1 \; ... \; e_n \quad \texttt{(* appel d'une fonction *)} \\
& \mid \text{let } (x_1, ..., x_n) = e_1 \text{ in } e_2 \mid (e_1, ..., e_n) \\
& \mid \text{Array.create } e_1 \; e_2 \mid e_1.(e_2) \mid e_1.(e_2) \leftarrow e_3
\end{aligned}
$$

2.2 Analyse et inférence de types

2.2.1 Définition des types

L'ensemble des types disponibles pour MinCaml est défini dans le fichier `type.ml` comme suit :

$$
\begin{aligned}
t ::= \; & \text{Unit} \mid \text{Bool} \mid \text{Int} \mid \text{Float} \\
& \mid \text{Fun}((t_1, ..., t_n), t) \\
& \mid \text{Tuple}(t_1, ..., t_n) \\
& \mid \text{Array}(t) \\
& \mid \alpha, \beta, \gamma, ...
\end{aligned}
$$

Pour le type Fun, le tuple $(t_1, ..., t_n)$ indique le type de chaque paramètre en entrée et le type t indique la valeur de retour de la fonction.

Les types $\alpha, \beta, \gamma, ...$ sont des types inconnus qui servent pour l'inférence de type. Par exemple, pour l'expression `let x = a in expr`, on commence par dire que `x` est de type α avant de résoudre le type par résolution d'équation. Le parseur réalise cette association sur les définitions de variables, de fonctions et de tuples.

2.2.2 Construction du système d'équations

On détermine l'ensemble des équations en parcourant la structure de l'arbre syntaxique représentant le programme. On définit ainsi la fonction $\text{genEq}_E : e \times t \to \mathcal{P}(t \times t)$ qui a une expression et un type attendu retourne les équations associées. Ici, E représente l'environnement, c'est-à-dire l'ensemble des noms de variables associées à leur type. On a alors :

$$
\begin{aligned}
\text{genEq}_E((), t) &= \{t = \text{Unit}\} & \qquad \text{genEq}_E(\text{bool}, t) &= \{t = \text{Bool}\} \\
\text{genEq}_E(\text{int}, t) &= \{t = \text{Int}\} & \qquad \text{genEq}_E(\text{float}, t) &= \{t = \text{Float}\} \\
\text{genEq}_E(\neg e, t) &= \{t = \text{Bool}\} \cup \text{genEq}(e, \text{Bool}) & \qquad \text{genEq}_E(-e, t) &= \{t = \text{Int}\} \cup \text{genEq}_E(e, \text{Int})
\end{aligned}
$$

$$
\begin{aligned}
\text{genEq}_E(e_1 + e_2, t) &= \{t = \text{Int}\} \cup \text{genEq}_E(e_1, \text{Int}) \cup \text{genEq}(e_2, \text{Int}) \\
&\vdots \\
\text{genEq}_E(e_1 = e_2, t) &= \{t = \text{Bool}\} \cup \text{genEq}_E(e_1, t') \cup \text{genEq}_E(e_2, t') \\
\text{genEq}_E(e_1 \leq e_2, t) &= \{t = \text{Bool}\} \cup \text{genEq}_E(e_1, t') \cup \text{genEq}_E(e_2, t') \\
\text{genEq}_E(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, t) &= \text{genEq}_E(e_1, \text{Bool}) \cup \text{genEq}_E(e_1, t) \cup \text{genEq}_E(e_2, t) \\
\text{genEq}_E(x, t) &= \{t = t', (x, t') \in E\} \\
\text{genEq}_E(\text{let } x = e_1 \text{ in } e_2, t) &= \text{genEq}_E(e_1, \alpha) \cup \text{genEq}_{E \cup \{(x,\alpha)\}}(e_2, t) \\
\text{genEq}_E(f \; x_1 \; ... \; x_n, t) &= \text{genEq}_E(f, \text{Fun}((\alpha_1, ..., \alpha_n), t)) \bigcup_{i=1}^{n} \text{genEq}_E(x_i, \alpha_i) \\
\text{genEq}_E(\text{let rec } f \; x_1 \; ... \; x_n = e_1 \text{ in } e_2, t) &= \text{genEq}_{E \cup \{(x_1,\alpha_1),...,(x_n,\alpha_n)\}}(e_1, \beta) \cup \text{genEq}_{E \cup \{(f,\beta)\}}(e_2, t) \\
&\quad \cup \{(\beta, \text{Fun}((\alpha_1, ..., \alpha_n), \gamma)\}
\end{aligned}
$$

$$
\begin{array}{rcl}
\text{genEq}_E((x_1,...,x_n),t) & = & \{t = \text{Tuple}(\alpha_1,...,\alpha_n),\ \text{avec}\ (x_i,\alpha_i) \in E\} \\
\text{genEq}_E(\text{let}\ (x_1,...,x_n) = e_1\ \text{in}\ e_2, t) & = & \text{genEq}_E(e_1, \text{Tuple}(\alpha_1,...,\alpha_n)) \\
 & & \cup\ \text{genEq}_{E\cup\{(x_1,\alpha_1),...,(x_n,\alpha_n)\}}(e_2,t) \\
\text{genEq}_E(\text{Array.create}\ e_1\ e_2, t) & = & \{t = \text{Array}(\alpha)\} \cup \text{genEq}_E(e_1,\text{Int}) \cup \text{genEq}_E(e_2,\alpha) \\
\text{genEq}_E(e_1.(e_2),t) & = & \text{genEq}_E(e_1,\text{Array}(t)) \cup \text{genEq}_E(e_2,\text{Int}) \\
\text{genEq}_E(e_1.(e_2) \leftarrow e_3, t) & = & \{t = \text{Unit}\} \cup \text{genEq}_E(e_1,\text{Array}(\alpha)) \cup \text{genEq}_E(e_2,\text{Int}) \\
 & & \cup\ \text{genEq}_E(e_3,\alpha)
\end{array}
$$

On a comme environnement par défaut Predef que l'on donne ici :

$$
\begin{array}{rll}
\text{Predef} = \{ & (\texttt{print\_int}, \text{Fun}((\text{Int}),\text{Unit})), & (\texttt{print\_newline}, \text{Fun}((\text{Unit}),\text{Unit})), \\
 & (\texttt{print\_float}, \text{Fun}((\text{Float}),\text{Unit})), & (\texttt{abs}, \text{Fun}((\text{Int}),\text{Int})), \\
 & (\texttt{abs\_float}, \text{Fun}((\text{Float}),\text{Float})), & (\texttt{sin}, \text{Fun}((\text{Float}),\text{Float})), \\
 & (\texttt{cos}, \text{Fun}((\text{Float}),\text{Float})), & (\texttt{sqrt}, \text{Fun}((\text{Float}),\text{Float})), \\
 & (\texttt{int\_of\_float}, \text{Fun}((\text{Float}),\text{Int})), & (\texttt{truncate}, \text{Fun}((\text{Float}),\text{Int})), \\
 & (\texttt{float\_of\_int}, \text{Fun}((\text{Int}),\text{Float})) & \}
\end{array}
$$

On génère ainsi les équations d'un programme a par l'appel à $\text{genEq}_{\text{Predef}}(a, \text{Unit})$. Le corollaire est que tout programme doit être une expression de type Unit.

2.2.3 Résolution des équations

Pour la résolution des équations, on utilise l'algorithme d'unification qu'on peut retrouver ici : `https://wackb.gricad-pages.univ-grenoble-alpes.fr/inf402/Poly-inf402.pdf`, section 5.3.2 (page 97). On en donne ainsi la version adaptée au type t défini ci-dessus.

La fonction `is_same t1 t2 : bool` du fichier `type.ml` indique si deux types sont identiques. La fonction `occur t1 t2 : bool` indique si t_1 apparaît (est contenu) dans t_2. Enfin `replace` remplace dans une liste d'équations un type t_1 par un type t_2. Ainsi :

$$
\begin{array}{rcl}
\texttt{(* Supprimer les équations inutiles *)} & & \\
\text{resolution}(\{t = t\} \cup E) & = & \text{resolution}(E) \\
\texttt{(* Décomposition *)} & & \\
\text{resolution}(\{\text{Fun}((t_1,...,t_n),t) = \text{Fun}((t'_1,...,t'_n),t')\} \cup E) & = & \text{resolution}(\bigcup_{i=1}^{n}\{(t_i = t'_i\} \cup \{t = t'\} \cup E) \\
\text{resolution}(\{\text{Tuple}(t_1,...,t_n) = \text{Tuple}(t'_1,...,t'_n)\} \cup E) & = & \text{resolution}(\bigcup_{i=1}^{n}\{(t_i = t'_i\} \cup E) \\
\text{resolution}(\text{Array}(t) = \text{Array}(t')) & = & \text{resolution}(\{t = t'\} \cup E) \\
\texttt{(* Elimination *)} & & \\
\text{resolution}(\{\alpha = t\} \cup E) & = & \{\alpha = t\} \cup \text{resolution}(E) \\
 & & \text{si } \alpha \text{ n'apparaît pas dans t} \\
 & & \text{sinon : } \texttt{Echec de l'élimination} \\
\texttt{(* Orientation *)} & & \\
\text{resolution}(\{t = \alpha\} \cup E) & = & res(\{\alpha = t\} \cup E) \\
\texttt{(* Dans les autres cas : échec de la décomposition *)} & &
\end{array}
$$

On retranscrit ensuite les types inférés avec resolution dans l'arbre syntaxique et on passe à la k-normalisation.

2.3 k-normalisation et α-conversion

2.3.1 k-normalisation

La k-normalisation permet de décomposer une expression et de stocker les résultats intermédiaires dans une variable temporaire.

On redéfinit le type pour que mieux comprendre comme suit : on a fait le choix ici de convertir les booléen en entier (0 pour `false` et 1 pour `true`).

$$
\begin{array}{rl}
k ::= & () \mid \text{int} \mid \text{float} \mid x \mid -x \mid x_1 + x_2 \mid x_2 - x_2 \\
& \mid x_1 +. \ x_2 \mid x_1 -. \ x_2 \mid x_1 *. \ x_2 \mid x_1 /. \ x_2 \\
& \mid \text{if } x_1 = x_2 \text{ then } k_1 \text{ else } k_2 \mid \text{if } x_1 \leq x_2 \text{ then } k_1 \text{ else } k_2 \\
& \mid \text{let } x = k_1 \text{ in } k_2 \\
& \mid \text{let rec } f\ x_1\ ...\ x_n = k_1 \text{ in } k_2 \\
& \mid x\ x_1\ ...\ x_n \texttt{ (* Appel de fonction *)} \\
& \mid \text{let } (x_1, ..., x_n) = x \text{ in } k \mid (x_1, ..., x_n) \\
& \mid \text{Array.create } x_1\ x_2 \mid x_1.(x_2) \mid x_1.(x_2) \leftarrow x_3
\end{array}
$$

On redonne ici une version complète de l'algorithme très largement inspiré de celui présenté dans l'article mentionné plus haut.

$$
\begin{array}{rcl}
\mathcal{K}(()) & = & () \\
\mathcal{K}(b), b \in \text{bool} & = & \begin{cases} 1 \text{ si } b = \text{true} \\ 0 \text{ si } b = \text{false} \end{cases} \\
\mathcal{K}(i), i \in \text{int} & = & i \\
\mathcal{K}(f), f \in \text{float} & = & f \\
\mathcal{K}(\neg e) & = & \mathcal{K}(\text{if } e = \text{true then false else true}) \\
\mathcal{K}(-e) & = & \text{let } x = \mathcal{K}(e) \text{ in } -x \\
\mathcal{K}(e_1 + e_2) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in } x_1 + x_2 \\
\mathcal{K}(e_1 - e_2) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in } x_1 - x_2 \\
\mathcal{K}(e_1 +. \ e_2) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in } x_1 +. \ x_2 \\
& \vdots & \\
\mathcal{K}(e_1 /. \ e_2) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in } x_1 /. \ x_2 \\
\mathcal{K}(\text{let } x = e_1 \text{ in } e_2) & = & \text{let } x = \mathcal{K}(e_1) \text{ in } \mathcal{K}(e_2) \\
\mathcal{K}(x) & = & x \\
\mathcal{K}(e_1 = e_2) & = & \mathcal{K}(\text{if } (e_1 = e_2) \text{ then true else false}) \\
\mathcal{K}(e_1 \leq e_2) & = & \mathcal{K}(\text{if } (e_1 \leq e_2) \text{ then true else false}) \\
\mathcal{K}(\text{if } (e_1 = e_2) \text{ then } e_3 \text{ else } e_4) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in (if } x_1 = x_2 \text{ then } \mathcal{K}(e_3) \text{ else } \mathcal{K}(e_4)) \\
\mathcal{K}(\text{if } (e_1 \leq e_2) \text{ then } e_3 \text{ else } e_4) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in (if } x_1 \leq x_2 \text{ then } \mathcal{K}(e_3) \text{ else } \mathcal{K}(e_4)) \\
\mathcal{K}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) & = & \mathcal{K}((\text{if } e_1 = \text{false then } e_3 \text{ else } e_2) \\
\mathcal{K}(\text{let rec } f\ x_1\ ...\ x_n = e_1 \text{ in } e_2) & = & \text{let rec } f\ x_1\ ...\ x_n = \mathcal{K}(e_1) \text{ in } \mathcal{K}(e_2) \\
\mathcal{K}(e\ e_1\ ...\ e_n) & = & \text{let } x = \mathcal{K}(e) \text{ in let } x_1 = \mathcal{K}(e_1) \text{ in} \\
& & \vdots \\
& & \text{let } x_n = \mathcal{K}(e_n) \text{ in } x\ x_1\ ...\ x_n \\
\mathcal{K}(\text{let } (x_1, ..., x_n) = e_1 \text{ in } e_2) & = & \text{let } x = \mathcal{K}(e_1) \text{ in let } (x_1, ..., x_n) = x \text{ in } \mathcal{K}(e_2) \\
\mathcal{K}((e_1, ..., e_n)) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in} \\
& & \vdots \\
& & \text{let } x_n = \mathcal{K}(e_n) \text{ in } (x_1, ..., x_n) \\
\mathcal{K}(\text{Array.create } e_1\ e_2) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in Array.create } x_1\ x_2 \\
\mathcal{K}(e_1.(e_2)) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in } x_1.(x_2) \\
\mathcal{K}(e_1.(e_2) \leftarrow e_3) & = & \text{let } x_1 = \mathcal{K}(e_1) \text{ in let } x_2 = \mathcal{K}(e_2) \text{ in let } x_3 = \mathcal{K}(e_3) \text{ in } x_1.(x_2) \leftarrow x_3
\end{array}
$$

Remarque : L'ajout d'un `let` nécessite de mettre le type de l'expression. Cela explique l'ajout de la fonction `get_type` qui prend un AST k-normalisé et l'environnement qui, à un nom de variable associe son type et renvoie le type de l'AST en entrée.

2.3.2 α-conversion

L'α-conversion consiste à donner un nom unique à chaque variable. Pour cela dès qu'on croise un `let` on utilise la fonction `Id.make_unique` et on met à jour la fonction ϵ en ajoutant la nouvelle association.

Voici les parties principales du fichier `alpha.ml`.

```
(* la fonction epsilon est représenté par un tableau d'association *)
type espilon = (Id.t * Id.t) list

(* Application de la fonction espilon *)
let apply (eps:epsilon) (var:Id.t) : Id.t =
  try snd (List.find (fun (x,y) -> x = var) eps)
  with Not_found -> var

...
(* Utilisation sur Var s et Add (x, y) *)
| Var s -> Var (eps_apply eps s)
| Add (x, y) -> Add (eps_apply eps x, eps_apply eps y)

...
(* Mise à jour lors d'un let et let rec *)
| Let ((x,t), e1, e2) ->
  let newvar = Id.make_unique x in (* rendre x unique *)
    let eps2 = (s, newvar)::eps in (* Mise à jour d'epsilon *)
      let e12 = a_conversion eps e1 and e22 = a_conversion eps2 e2 in
        Let ((newvar, t), e12, e22)

| LetRec (fd, e) ->
  (* on rend unique le nom de la fonction et ses arguments  *)
  let newname = Id.make_unique (fst fd.name) in
  let newargs = List.map (fun x -> Id.make_unique (fst x)) fd.args in
    (* Mise à jour d'epsilon *)
    let eps2 = (fst fd.name, newname)::(List.map2 (fun x y -> (fst x,y)) fd.
    args newargs) @ env in
      (* Application d'epsilon *)
      let newbody = alpha_conversion env2 fd.body in
      let e2 = alpha_conversion ((fst fd.name, newname)::env) e in
        (* Mise à jour de l'arbre *)
        let args = List.map2 (fun (s1,t) s2 -> (s2,t)) fd.args newargs in
          LetRec ({name = (newname, snd fd.name); args = args; body = newbody
    }, e2)
```

CODE 1 – Extrait du code pour l'α-conversion

2.4 Optimisations

2.4.1 β-réduction

Cette optimisation consiste à supprimer les alias de variables. Par exemple : `let a = 1 in let b = a in a + b` donne `let a = 1 in a + a`. Pour cela on utilise le module `Alpha` avec, dans ce cas, la fonction ϵ qui associe à un alias, la variable d'origine correspondante. Voici une partie des équations :

$$
\begin{aligned}
\beta_\epsilon(i), i \in \text{int} &= i \\
&\vdots \\
\beta_\epsilon(x) &= \epsilon(x) \\
\beta_\epsilon(x_1 + x_2) &= \epsilon(x_1) + \epsilon(x_2) \\
&\vdots \\
\beta_\epsilon(\text{if } x_1 = x_2 \text{ then } k_1 \text{ else } k_2) &= \text{if } \epsilon(x_1) = \epsilon(x_2) \text{ then } \beta_\epsilon(k_1) \text{ else } \beta_\epsilon(k_2) \\
\beta_\epsilon(\text{let rec } f\ x_1\ \ldots\ x_n = k_1 \text{ in } k_2) &= \text{let rec } f\ x_1\ \ldots\ x_n = \beta_\epsilon(k_1) \text{ in } \beta_\epsilon(k_2) \\
&\vdots \\
\beta_\epsilon(\text{let } x_1 = k_1 \text{ in } k_2) &= \begin{cases} \beta_{\epsilon, x \mapsto x'}(k_2), & \text{si } \beta_\epsilon(k_1) = x' \\ \text{let } x = \beta_\epsilon(k_1) \text{ in } \beta_\epsilon(k_2) & \text{sinon} \end{cases}
\end{aligned}
$$

2.4.2 Réduction des expressions imbriquées

Cette optimisation consiste à « aplatir » les expressions `let` imbriquées. Par exemple l'expression `let a = let b = 5 in b + 1 in a + 2` devient `let b = 5 in let a = b + 1 in a + 2`.

```
let rec insert (x:Id.t*Type.t) (e1:knorm_t) (e2:knorm_t) : knorm_t =
  match e1 with
  | Let (y, e3, e4) -> Let (y, e3, insert x e4 e2)
  | LetRec (fundef, e) -> LetRec (fundef, insert x e e2)
  | LetTuple (l1, v, e) -> LetTuple (l1, v, insert x e e2)
  | _ -> Let (x, e1, e2)

let rec reduction (ast:knorm_t) : knorm_t =
  match ast with
  | IfEq (b, e2, e3) -> IfEq (b, reduction e2, reduction e3)
  | IfLE (b, e2, e3) -> IfLE (b, reduction e2, reduction e3)
  | Let (x, e1, e2) -> insert x (reduction e1) (reduction e2)
  | LetRec (fd, e) ->
    LetRec ({name = fd.name; args = fd.args; body = reduction fd.body},
   reduction e)
  | LetTuple(l1, l2, e) -> LetTuple(l1, l2, reduction e)
  | _ -> ast_norm
```

Code 2 – Fichier reduction.ml

La fonction `insert` déplace les affectations en appliquant autant que possible les équivalences suivantes :

$$\begin{aligned} \text{let } x = (\text{let } y = e_3 \text{ in } e_4) \text{ in } e_2 &\equiv \text{let } y = e_3 \text{ in } (\text{let } x = e_4 \text{ in } e_2) \\ \text{let } x = (\text{let rec } f\ x_1 \ldots x_n = e' \text{ in } e) \text{ in } e_2 &\equiv \text{let rec } f\ x_1 \ldots x_n = e' \text{ in } (\text{let } x = e \text{ in } e_2) \\ \text{let } x = (\text{let } (x_1, \ldots, x_n) = v \text{ in } e) \text{ in } e_2 &\equiv \text{let } (x_1, \ldots, x_n) = v \text{ in } (\text{let } x = e \text{ in } e_2) \end{aligned}$$

2.4.3 Suppression d'appels de fonctions ou « inlining »

Cette optimisation se déroule comme suit :

— la fonction `Inline.depth` permet de calculer la taille d'une fonction. En pratique, cela correspond approximativement au nombre d'instructions. Une fonction récursive a une taille arbitrairement supérieure à `max_depth`

— A chaque définition de fonction, si sa taille est plus petite qu'une constante (ici `max_depth`) on ajoute la définition (de type `fundef`) de la fonction dans l'environnement.

— Lorsqu'on a un appel de fonction qui est dans l'environnement, on remplace l'appel par le corps en remplaçant les arguments (de la fonction) par les arguments effectifs (ceux donnés lors de l'appel) via la fonction `Inline.replace` qui réutilise la fonction `Alpha.apply`.

— Afin de garantir qu'aucune variable distincte n'ait le même nom, on applique l'α-conversion sur la partie ainsi substituée. En effet, si la fonction est appelée plusieurs fois, il ne faut pas que les variables locales à cette fonction gardent le même nom.

Voici l'extrait du programme pour l'application d'une fonction :

```
(* Extrait du code pour un noeud correspondant à un appel *)
| App (f, vars) -> (try
  (* On cherche si la fonction est dans l'environnement *)
  let fd = List.find (fun fd -> fst fd.name = f) env in
    (* On remplace les paramètres par ceux utilisé à l'appel dans le corps *)
    let expr = replace fd.body
      (List.map2 (fun x y -> (x,y)) (List.map fst fd.args) vars)
    in (* On applique l'alpha-conversion au résultat *)
      Alpha.conversion expr
(* si la fonction n'est pas dans l'environnement : on laisse l'appel *)
with Not_found -> a)
```

Code 3 – Extrait de `Inlime.ml`

Voici un petit exemple d'application :

```
let f x = let a = 2 in x + a in        let f x = let a = 2 in x + a in
let y = 5 in                           let y = 5 in
let u = f y in                         let u = (let a1 = 2 in y + a1) in
let v = f u in ()                      let v = (let a2 = 2 in u + a2) in ()
```

N.B. Cette optimisation peut engendrer des `let`-imbriqués, des alias de variables.

2.4.4 Propagation de constantes

Comme son nom l'indique, la propagation de constante sert à remplacer une variable par sa valeur. Par exemple, la ligne de code : `let x = 3 in let y = 7 in x + y` est remplacée par `let x = 3 in let y = 7 in 10`.

Cette optimisation est effectuée en mémorisant dans une liste l'association entre une variable et sa valeur (via : `type const = Ent of int | Floatt of float | Tuplet of Id.t list | None`). Cette liste permet au compilateur, lorsqu'il observe une expression, de vérifier si les éléments de l'expression sont des constantes ou non et, le cas échéant, d'appliquer les opérations (`+, -, +., -., /., *.`) effectuées sur ces constantes et de remplacer les éléments par leur valeur.

De plus, lorsque le compilateur rencontre une définition de variable (`let a = b in ...`), il ajoute si possible dans la liste l'association entre a et la valeur associée à b. Dans le cas où une expression n'est ni une constante, ni le résultat d'une opération simple, cette dernière est renvoyée telle quelle. La fonction `constant_folding` s'assure d'effectuer la propagation de constante quelque soit l'expression rencontrée (`if ... then ... else`, `let rec f x1 x2 ...` et `let (x1, x2, ..., xn)`) en effectuant des appels récursifs.

Cette fonction propage aussi les tuples. Ainsi :

```
let x = (u,v) in        devient    let x = (u,v) in
let (a,b) = x in                   let a = u in let b = v in
a + b                              a + b
```

N.B. Cette optimisation peut engendrer des alias de variables ainsi que des définitions inutilisées.

2.4.5 Suppression des définitions inutiles

La dernière optimisation avant la conversion en closures consiste à supprimer les expressions « `let` » inutiles. Si la définition est un appel de fonction ou une écriture dans un tableau ($a.(i) \leftarrow x$), on ne peut pas supprimer la définition car elle a des effets de bord.

En pratique, la fonction `Elim.has_side_effets`, prenant en paramètres une expression et la liste des fonctions avec des effets de bord, retourne vrai si l'expression a des effets de bord. Par défaut, les fonctions `print_int`, `print_float` et `print_newline` ont des effets de bords.

Pour savoir si une définition est utile, à chaque fois qu'une variable, une fonction ou un tuple est déclaré, on ajoute le(s) nom(s) de variable associé(s) à un entier (0 par défaut) dans l'environnement. A chaque fois, qu'une variable est utilisée, cet entier est incrémenté.

```
1 type environment = (Id.t * int ref) list
2
3 (* Incrémente l'entier associé à la variable v *)
4 let rec env_incr (env:environment) (v:Id.t) : unit =
5
6 (* Récupère l'entier à la variable v *)
7 let rec env_get (env:environment) (v:Id.t) : int =
```

Code 4 – Interface de l'environnement pour l'élimination de définitions

La fonction `Elim.elim` prend en paramètre une expression (k-normalisée et α-convertie) et deux environnement. Le premier (le plus important) est celui décrit ci-avant, le deuxième correspond à la liste des fonctions avec des effets de bord.

Le code ci-dessous donne la partie principale du programme.

```
let rec elim (a:knorm_t) (env:environment) (env2:Id.t list) : knorm_t =
(* la fonction elim incremente le nombre d'utilisation de chaque variable.
   Par exemple : *)
| App (f, vars) ->
  let _ = env_incr env f; List.iter (env_incr env) vars in a

(* Cas d'une déclaration de variable *)
| Let ((x,t), e1, e2) ->
  let env' = (x, ref 0)::env in (* on ajoute x à l'environnement *)
    let e2' = elim e2 env' env2 in (* on applique l'élimination sur e2 *)
      (* si x est utilisé dans e2 ou si e1 a des effets de bord *)
      if (env_get env' x) <> 0 || has_side_effects e1 env2 then
        (* on élimine les déf. inutiles dans e1 et on garde le définition *)
        let e1' = elim e1 env env2 in Let((x,t), e1', e2')
      else e2' (* sinon on supprime la définition *)

(* Cas d'une déclaration de fonction *)
| LetRec (fd, e) ->
  (* on ajoute le nom de la fonction au 1er environnement *)
  let env' = (fst fd.name, ref 0)::env in
    (* si la fonction a des effets de bord on l'ajoute au 2e environnement *)
    let env22 =
      if has_side_effects fd.body env2 then (fst fd.name)::env2 else env2
    in
      (* en élimine dans e avec les nouveaux environnements *)
      let e' = elim e env' env22 in
        (* si la fonction est utilisé *)
        if (env_get env' (fst fd.name)) <> 0 then
          (* on élimine dans le corps de la fonction et on garde la def. *)
          let b = elim fd.body env env2 in
            LetRec({name= fd.name; args=fd.args; body=b}, e')
        else e' (* sinon on supprime la déclaration *)

(* Pour les tuples on procède de même en ajoutant toutes les variables à env
   et regardant si la somme des utilisations de chacune est nulle *)
```

CODE 5 – Gestion des définitions dans la suppression des celles inutiles

N.B. Cette optimisation peut engendrer de nouvelles définitions inutilisées d'où l'itération de la figure 1

2.5 Conversion en closures et génération de l'ASML

2.5.1 Conversion en closures

Durant cette étape :

— les définitions des fonctions sont séparées du reste du code qui constitue le code principal ;
— Les définitions de fonctions imbriquées sont séparées les unes des autres ;
— Pour chaque fonction on identifie ses paramètres et ses variables libres ;
— On "supprime" les fonctions de haut-niveau, c'est à dire qu'une fonction ne retourne plus une fonction et ne prend plus de fonctions en paramètre.

Les variables libres d'une fonction sont l'ensemble des variables déclarées en dehors de la fonction, qui sont utilisées dans le code de celle-ci sans faire parti de ses paramètres.

Par exemple dans le code suivant, la fonction f a pour variable libre la variable a.

```
let a = 12 in
let rec f b = a + b in
print_int (f 12)
```

CODE 6 – Code exemple avec variables libres

Les fonctions ne peuvent plus retourner une fonction et ne peuvent plus avoir de fonctions en paramètre. Il devient alors nécessaire d'ajouter une closure dans les cas suivants :

1. Appel d'une fonction qui a des variables libres
2. Une fonction retourne une autre fonction
3. Une fonction est utilisée en paramètre d'une autre fonction

Chaque closure contient le label de la fonction ainsi que l'ensemble des valeurs des variables libres de cette fonction. La création d'une closure se fait grâce à une nouvelle instruction `make_closure(l, frees)`. L'appel d'une fonction dans une closure se fait via l'instruction `apply_closure(c, args)` où *c* est le nom de la variable qui contient la closure de la fonction et *args* sont les valeurs des paramètres de la fonction.

Voici un exemple complet d'un programme mincaml et de son état après la closure conversion :

```
let a = 12 in
let rec double x =
    x + x
in
let rec make_adder f x =
    let b = f x in
    let rec add y =
    x + y
    in
add
in
print_int ((make_adder double 10) 5)
```

Code 7 – Exemple complet pour la closure conversion

```
label: _make_adder8
parameters: f9 x10
frees variables: None
code:
  let b11 = apply_closure(f9, x10) in
  let t18 = make_closure(_add12, x10) in
  t18

label: _add12
parameters: y13
frees variables: x10
code:
  x10 + y13

label: _double6
parameters: x7
frees variables: None
code:
  x7 + x7

MAIN:
let a5 = 12 in
let t116 = 10 in
let t19 = make_closure(_double6) in
let t215 = apply_direct(_make_adder8, t19, t116) in
let t317 = 5 in
let t414 = apply_closure(t215, t317) in
apply_direct(print_int, t414)
```

Code 8 – Résultat après la conversion en closures

2.5.2 Génération de l'ASML

Cette étape est la dernière étape obligatoire dans la partie front-end du compilateur. Durant celle-ci, le code produit lors de la closure conversion est converti dans un langage appelé ASML.

Le langage ASML est un assembleur de haut-niveau, ce qui signifie que :

— chaque programme utilise un nombre infini de variables (alors qu'en assembleur un programme utilise un nombre fini de registres) ;
— les éléments de "haut-niveau" comme les if-then-else et les appels de fonctions sont toujours présents (en assembleur ils sont remplacés par des comparaisons, des étiquettes et des branchements).

Il fait office d'intermédiaire entre la partie front-end et la partie back-end du compilateur. Un programme ASML est structuré de la façon suivante :

1. Déclarations des flottants
2. Définitions des fonctions
3. La fonction main

Chaque fonction peut avoir un nombre différent de paramètres et de variables libres, à l'exception de la fonction main qui n'a aucun paramètre ni aucune variable libre. Si une fonction a des variables libres, celles-ci sont chargées depuis la mémoire au début du corps de la fonction en utilisant un pointeur spécial appelé `%self` qui pointe sur l'adresse en mémoire de la closure avec laquelle la fonction a été appelée.

Les accès en mémoire se font via l'instruction `mem(addr + offset)` où `addr` est une adresse en mémoire et `offset` une valeur immédiate entière ou une variable contenant une valeur entière. En ASML les tuples et les tableaux sont en mémoire, il faut donc utiliser l'instruction `mem` pour pouvoir les manipuler.

Contrairement aux entiers, les flottants ne peuvent pas être manipulés directement. Ils sont donc déclarés au début du programme et placés dans des labels. Ils sont alors considérés comme étant en mémoire doivent être chargés pour pouvoir être manipulés.

Voici un programme mincaml et le code ASML généré pour celui-ci :

```
let rec double x =  x + x in
let rec fdouble x = x +. x in
let a = 1 in
let b = 2.0 in
print_int (double a);
print_float (fdouble b)
```

Code 9 – Un programme mincaml exemple

```
let _b9 = 2.

let _fdouble6 x7 =
  (fadd x7 x7)

let _double4 x5 =
    add x5 x5

let _ =
  let a8 = 1 in
  let t14 = _b9 in
  let b9 = mem(t14 + 0) in
  let t311 = call _double4 a8 in
  let t110 = call _min_caml_print_int t311 in
  let t212 = call _fdouble6 b9 in
  let t13 = call _min_caml_print_float t212 in
  t13
```

Code 10 – Code ASML généré pour le programme précédent

2.5.3 Optimisation de valeurs immédiates sur l'ASML

En assembleur, il est possible d'utiliser des valeurs immédiates (par exemple `add r0, r0, #15`. La figure suivante donne l'utilisation d'une valeur immédiate dans une instruction.

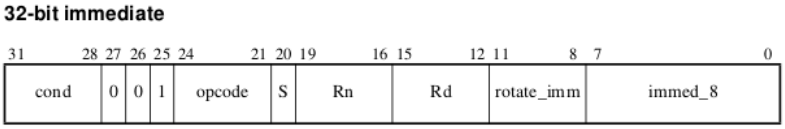

FIGURE 2 – Extrait de la documentation ARM

Ainsi, tous les entiers ne peuvent pas être des valeurs immédiates. En ARM (v5 et antérieure), une valeur immédiate est sur 12 bits. Cependant, sur ces 12 bits seuls 8 bits correspondent à la partie principale et les 4 bits restants sont multipliés par 2 pour faire une rotation des 8 bits précédents.

Simplement une valeur est immédiate si elle contient 12 zéros consécutifs modulo 16 dans son écriture quaternaire (en base 4).

```
1  (* donne sous forme de tableau l'écriture en base 4 de i *)
2  let bits (i:int) : int array
3
4  (* Détermine le nombre de zeros successifs à partir de k (modulo 16) *)
5  let nb_succ (a:int array) (k:int): int
6
7
8  (* Determine si un entier entre sur 8 bits moyennant une rotation *)
9  (* On cherche 12 chiffres à zéro consécutifs (modulo 16) *)
10 let can_be_imm (i:int) : bool
11
12 (* Propage les valeurs immédiates selon un algorithme similaire mais
13    simplifié de celui de la propagation de constantes *)
14 let propagation_imm (asml: Asml.asml) : Asml.asml
15
16 (* Elimination des déf. inutiles dans la même idée que le module Elim *)
17 let elimination_imm (asml: Asml.asml) : Asml.asml
18
19 (* Fonction principale : composition des deux ci-dessus *)
20 let optim (asml:Asml.asml) : Asml.asml =
21   elimination_imm (propagation_imm asml)
```

CODE 11 – Interface de `immOpt.ml`

2.6 Allocation des registres

Initialement, le choix d'un algorithme d'allocation de registres a été fait en faveur de *Linear Scan Algorithm*. Après de nombreuses tentatives d'implémentation, une variante de celui-ci a été mise au point. La différence consiste à ne garder qu'un nombre de variables actives équivalent au nombre de registres utilisables.

Ainsi, les variables actives sont les n prochaines variables utilisées, indépendamment du fait qu'elles soient chargées dans un registre ou non. Seules les noms de variables sont gardés dans une liste référencée `actives` qui est mise à jour par la fonction `get_interval_i` à chaque instruction.

Les variables qui sont chargées dans des registres sont sauvegardées dans une hashmap reliant `'nom de variable' -> 'registre' (string, string)`. Pour les variables présentes dans la hashmap mais absentes de la liste, le contenu du registre `r` est enregistré sur la pile à l'adresse dédiée, `r` est libéré et une nouvelle variable présente dans la liste `active` et absente de la hashmap occupera `r`. Après l'avoir chargée, on vérifie si la variable a une adresse allouée dans la pile, si elle n'en a pas une on lui réserve une case mémoire.

Regardons l'exemple de la table 1. A la ligne 6, `z` n'est pas présente dans la liste `active` donc elle est `store`, et `y` est présente dans la liste `active` et non dans la hashmap, donc elle est `load`.

i	Ligne de code	Liste active	Hashmap précédente	Hashmap actuelle
1	`k = 1`	`(k ; x ; y)`	$\emptyset$	{`k : r4 ; x : r5 ; y : r6`}
2	`x = k + 3`	`(k ; x ; y)`	{`k : r4 ; x : r5 ; y : r6`}	{`k : r4 ; x : r5 ; y : r6`}
3	`y = x + k`	`(k ; x ; y)`	{`k : r4 ; x : r5 ; y : r6`}	{`k : r4 ; x : r5 ; y : r6`}
4	`v = 3`	`(v ; z ; i)`	{`k : r4 ; x : r5 ; y : r6`}	{`v : r4 ; z : r5 ; i : r6`}
5	`z = 6`	`(v ; z ; i)`	{`v : r4 ; z : r5 ; i : r6`}	{`v : r4 ; z : r5 ; i : r6`}
6	`i = v + y`	`(v ; y ; i)`	{`v : r4 ; z : r5 ; i : r6`}	{`v : r4 ; y : r5 ; i : r6`}
7	`a = k + z`	`(a ; k ; z)`	{`v : r4 ; y : r5 ; i : r6`}	{`a : r4 ; k : r5 ; z : r6`}

TABLE 1 – Exemple d'allocation de registre avec 3 registres de disponibles

L'algorithme est codé dans le fichier `regAlloc.ml`. Il reçoit comme entrée un programme de type `asml` et renvoie une structure de type `letregdef` qui sert pour la génération du code assembleur.

Le nouveau type indique explicitement quelles variables doivent être chargées dans quels registres ainsi que les adresses mémoires des variables qui doivent être stockées. L'intérêt de ce nouveau type est de diviser le problème en deux et, donc, de pouvoir le partager le travail entre plusieurs personnes.

2.7 Génération du code assembleur

Lors de la récupération de la structure de type `letregdef`, des traitements spécifiques sont effectués en fonction du type des variables. En effet, si des constantes supérieures à 2^8 sont présentes, elles sont placées dans une liste pour gérer le calcul du début de `sp`. De manière similaire, les labels des flottants sont stockés dans une autre liste car, dans le code, les flottants apparaissent avant les fonctions.

La première fonction est toujours la principale, elle porte le nom de `main`, les autres sont placées dans l'ordre qui figure dans `letregdef`. Le prologue de chaque fonction ARM est calculé dans la fonction `generate_asm_fun_internal`, afin de déterminer l'adresse de la pile (relativement à `fp`) la plus éloignée atteinte par celle-ci. La fonction itère sur les `store` pour prendre en compte les adresses (de la pile) ainsi que sur les `if` afin d'examiner les branchements. L'adresse la plus éloignée est ensuite mise dans une variable. Dans le cas où les paramètres d'une autre fonction sont `push` à la suite de `sp`, la variable est utilisée pour remettre le `sp` à la bonne adresse.

Une fois le prologue déterminé, la traduction vers l'assemblage ARM commence. En parallèle, des optimisations sont appliquées comme la suppression des `mov` et `add` inutiles (`mov r0, r0` ; `add r0, r0, #0`).

L'attention a également été portée sur l'utilisation de bonnes instructions en fonction du type de registre utilisé. En effet, il faut différencier les cas où le résultat ou la valeur est de type `int` ou `float`. Dans le cas d'un entier, les instructions principales à utiliser sont : `mov, add, sub, ldr, str`, alors que celles des nombres flottants (sur 32 bits) sont : `vmov.f32, vadd.f32, vsub.f32, vdiv.f32, vldr.f32, vstr.f32`.

3 Présentation des tests et des scripts de tests

Les tests et les scripts de tests ont été conçus de manière à identifier la fonctionnalité et/ou l'itération de développement correspondante testée. Ceci se retrouve dans leur nom et/ou le dossier auquel ils appartiennent. Chaque script calcule le nombre de tests réussis ou ayant échoués à l'aide de deux variables : `passed` et `failed` qui sont incrémentées pour chaque test réussi (respectivement échoué). Elles permettent de faire une synthèse succincte de tous les tests effectués et leurs résultats lors de l'appel à la commande `make test` dans le dossier `ocaml/`.

3.1 Tests de syntaxe

Le script `mincaml-test-parser.sh` ainsi que les programmes mincaml situés dans les sous-dossiers `tests/syntax/invalid` et `tests/syntax/valid` sont ceux fournis au démarrage du projet. Ils permettent de vérifier que la syntaxe d'un programme mincaml est correct par rapport aux règles du langage. Pour les programmes situés dans le dossier `invalid`, le test est réussi si le compilateur renvoie une erreur ou échoue à compiler. Un exemple de programme ayant une syntaxe incorrecte est :

```
let y = print_int
```

Code 12 – Exemple de code ayant une syntaxe invalide

En effet, la structure d'assignation est : `let x = ... in ...`

3.2 Tests d'analyse de type

Pour l'analyse de type, le script `mincaml-test-typechecking.sh` et les fichiers mincaml situés dans les sous-dossiers `tests/typechecking/invalid` et `tests/typechecking/valid` sont utilisés.

Ces tests se concentrent sur la deuxième étape du compilateur, à savoir s'assurer que le programme fourni par un utilisateur est correct par rapport à la correspondance entre les types des variables et des fonctions utilisées. Comme Mincaml est un langage fortement typé, ce dernier contraint l'utilisation des opérations ou des fonctions pour un certain type de variable. De ce fait, le compilateur ne doit pas laisser passer des programmes ne respectant pas la signature des opérations ou fonctions utilisées. Entre autres, le programme suivant n'est pas valide du point de vue du compilateur :

```
let x = 3 in
  if x then
    print_int(x)
  else
    print_int(0)
```

Code 13 – Exemple de programme invalide au niveau du typage

La variable `x` est un entier, pourtant elle est utilisée comme condition dans le `if`, ce qui n'est pas cohérent avec le type booléen, normalement attendu pour une condition.

L'organisation des programmes mincaml dans les différents sous-dossiers correspond à l'itération de développement des fonctionnalités suivie lors du projet (opérations arithmétiques simples, fonctions prédéfinies, if then else, fonctions, tuples/tableaux, closure, flottants). Cette dernière permet de distinguer les itérations du langage qui sont fonctionnelles par rapport à l'étape d'analyse de type. Ainsi, le script shell parcourt l'ensemble des sous-dossiers de `valid` et `invalid`, écrit le numéro et le nom de l'itération correspondants à chaque sous-dossier et essaye de lancer la commande `./mincamlc nom_du_fichier.ml -t` sur chacun des programmes contenus dans les sous-dossiers. Si cette commande réussit et que le test appartient au dossier `valid` alors il est déclaré comme réussi, à l'inverse les programmes situés dans le dossier `invalid` sont réussis si la commande précédente échoue.

3.3 Tests des étapes front-end (de la k-normalisation à la génération du code ASML)

Ensuite, les étapes du compilateur situées entre l'analyse de type et la génération du code intermédiaire ASML (inclue) sont testées à l'aide du script `mincaml-test-front-end.sh` et des programmes situés dans le dossier `gen-code`.

Ces tests permettent de contrôler que les transformations appliquées par les différentes optimisations soient correctement effectuées et renvoient un code ASML valide. Pour cela, le fichier asml résultant des optimisations est exécuté à l'aide de l'interpréteur asml situé dans le dossier `tools/`. Si le résultat de l'exécution du fichier asml est égal à celui attendu dans le fichier `.expected` du même nom, le test est réussi. Sinon il a échoué et la différence entre le résultat attendu et le résultat effectif est affiché par la commande `diff`. Le calcul du nombre de tests réussis et échoués suit la même logique que pour les tests précédents à l'aide des variables `passed` et `failed`.

```
         - Iteration 4 : functions -
Test on: 4-definition_function.ml ... OK
Test on: 4-double_functions.ml ... OK
Test on: 4-parameters_from_stack.ml ... OK

```

```
6           - Iteration 5 : arrays and tuples -
7  Test on: 5-change_value_in_array.ml ...1c1
8  < 0 0
9  ---
10 > 4 10
11  KO
```

Code 14 – Aperçu des résultats du lancement du script de test mincaml-test-front-end

Le fichier intermédiaire asml est obtenu en appelant la commande `./mincamlc nom_du_fichier.ml -asml` sur chaque programme mincaml. Cette commande applique toutes les étapes de compilation (primaires : k-normalisation, α-conversion,... et secondaires : β-réduction, propagation de constantes, ...) sur le fichier mincaml fourni en entrée.

N.B. Le nom des programmes est déterminé en fonction de l'itération de développement des fonctionnalités (cf. Table 2 à laquelle ils appartiennent. Ceci s'illustre dans le code ci-dessus.
De ce fait, si vous souhaitez ajouter de nouveaux fichiers de tests, il est impératif de les écrire sous la forme :

`numéro_iteration-(front- ou back- ou '')nom_explicite_de_test.ml`

Il faut qu'il soit accompagné de son fichier de résultats attendus `nom_du_fichier_test.expected` pour que les tests automatisés s'effectuent correctement sur votre nouveau fichier de test.

Numéro itération	Nom de l'itération
1	Opérations arithmétiques
2	Fonctions prédéfinies
3	If Then Else
4	Fonctions (`let rec`)
5	Tableaux (arrays) et tuples
6	Closures
7	Flottants

Table 2 – Tableau des itérations de développement du projet

3.4 Tests des étapes back-end (allocation des registres et génération du code ARM)

Enfin, l'allocation des registres et la génération du code ARM sont testés via le script `mincaml-test-back-end.sh` sur les mêmes programmes que pour les tests des étapes front-end, à l'exception des fichiers ayant le préfixe `front`.

Ici, les tests vérifient que le code assembleur généré via un fichier mincaml fourni en entrée, compilé en programme arm par la commande `arm-linux-gnueabi-gcc` et exécuté par le programme `qemu-user` renvoie le résultat attendu par l'exécution du code mincaml initial. Le choix d'utiliser la libC pour générer le fichier arm (avec `arm-linux-gnueabi-gcc`) est justifié par la présence de fonctions prédéfinies, telles que `create_array` ou `print_float`, qui n'étaient pas dans le fichier `libmincaml.S` fourni à l'origine du projet.

Le calcul du nombre de tests réussis ou échoués est effectué de manière identique aux tests pour la partie front-end et la présentation des résultats des tests automatisés est également similaire. Seulement une manipulation supplémentaire est effectuée pour les tests sur les flottants. En effet, lorsque des flottants sont manipulés en arm, un certain nombre de chiffres après la virgule sont rajoutés, créant des 0 superflus. Par exemple, le chiffre 2.5 peut être renvoyé sous la forme 2.500000. De ce fait, la commande perl située dans le script `mincaml-test-back-end.sh` est appliquée pour supprimer les 0 supplémentaires et garantir la comparaison correcte entre le fichier de résultats attendus et effectifs.

N.B. Les optimisations secondaires ne sont pas appliquées pour la génération du code ARM, elles peuvent être ajoutées en modifiant le chiffre de la variable `OPTION` à 1 ou une autre chiffre supérieur à 1.

4 Travail en cours de développement

4.1 Problèmes existants

Le présent compilateur fonctionne sur tous les programmes mincaml, toutefois il ne gère pas les cas suivants :

— Les fonctions appelées qui créent une `closure` et la renvoient à la fonction appelante avec des variables libres. En effet, le pointeur de la closure est placé dans le registre de retour, mais la closure est placée dans la pile après l'appel de la fonction appelée, le problème est qu'elle est ensuite dépilée et devient inaccessible ;
— Les programmes qui utilisent simultanément des entiers et des flottants, car on ne connaît pas les types des variables dans le type `ASML` ;
— De la même manière, les programmes ayant des tableaux de flottants. Les tableaux sont fonctionnels dans la fonction principale (`Main`), mais lorsqu'ils sont passés en paramètres, l'information sur le type est perdue.

4.2 Fonctionnalités non implémentées

La transmission d'une table des symboles de l'étape d'analyse syntaxique jusqu'à la génération du code assembleur n'est pas effectuée. Néanmoins, l'information du type des variables et fonctions est maintenue jusqu'à l'étape de closure (non inclue).

4.3 Améliorations futures

L'ajout d'une table des symboles générée avec l'ASML permettrait de faciliter l'allocation des registres et la génération du code assembleur.

De plus, une liste de fonctionnalités envisageable pour le futur considère :

— La gestion des boucles (`while` et `for` par exemple) ;
— La gestion des listes ;
— L'ajout d'un *garbage collector* pour améliorer la gestion de la mémoire des programmes MinCaml (essentiellement pour les tableaux).