

Travaux pratiques de programmation

Implémentation des langages de haut niveau

Agrégation externe d'informatique 2026

Le sujet est composé de trois parties complètement indépendantes. Il est attendu du candidat qu'il aborde les trois parties.

Table des matières

1	Fermeture d'un code fonctionnel – programmation OCaml	1
1.1	Introduction	1
1.2	Organisation du code fourni	5
1.3	Questions	7
1.3.1	Prise en main	7
1.3.2	Calcul des variables libres	7
1.3.3	Fonctions et fonctions récursives	8
1.3.4	Gestion des opérateurs prédéfinis	8
2	Gestion automatique de la mémoire – programmation C	8
2.1	Organisation de la mémoire	9
2.2	Allocation	10
2.2.1	Il reste de la place dans <code>tas.donnees</code>	11
2.2.2	Il n'y a plus de place dans <code>tas.donnees</code>	11
2.3	Annexe	15
3	Revue de code	18

1 Fermeture d'un code fonctionnel – programmation OCaml

1.1 Introduction

L'*explicitation des fermetures* est une transformation de programme qui concerne les langages de programmation fonctionnels. Elle consiste à traduire un langage doté de fonctions anonymes dont le corps peut contenir des variables libres, comme par exemple OCaml, en un langage où toutes les fonctions sont nommées et dont le corps est clos, c'est-à-dire sans variable libre, comme par exemple C. Le langage d'arrivée étant plus proche du langage machine que le langage de départ, il s'agit d'une étape classique du processus de compilation des langages fonctionnels.

L'objectif de cette partie du sujet est d'implémenter l'explicitation des fermetures au sein d'un compilateur miniature écrit en OCaml. Les langages de départ et d'arrivée de la transformation que vous allez écrire sont tous deux fixés. On commence par décrire ces langages et les grandes lignes de la transformation de programme avant de présenter l'architecture du code fourni et le format attendu pour votre contribution.

Langage de départ. Le langage de départ, nommé Skimmel, est une variante grandement simplifiée d'OCaml. Sa grammaire est présenté en forme de Backus-Naur à la figure 1a. On désigne par les lettres x, y, f des identifiants quelconques jouant le rôle de variables.

$e ::=$	Expressions	$e ::=$	Expressions
x	Variable	x	Variable
n	Constante littérale	n	Constante littérale
op	Opérateur prédéfini	$op(e^*)$	Opérateur
$\text{fun } x \rightarrow e$	Fonction anonyme	$e e$	Application
$\text{fun } x \text{ rec } f \rightarrow e$	Fonc. anon. récursive	$\text{let } e = e \text{ in } x$	Définition locale
$e e$	Application	$e ; e$	Séquence
$\text{let } e = e \text{ in } x$	Définition locale	$\text{if } e \text{ then } e \text{ else } e$	Conditionnelle
$e ; e$	Séquence	(e, e)	Appairage
$\text{if } e \text{ then } e \text{ else } e$	Conditionnelle	$\text{fst } e$	Projection gauche
(e, e)	Appairage	$\text{snd } e$	Projection droite
$\text{fst } e$ $\text{snd } e$	Projections	$\text{alloc } e$	Allocation
		$e.(e)$	Lecture indicée
		$e.(e) \leftarrow e$	Écriture indicée
		$p ::=$	Programmes
		$\text{let rec } f x^* = e \text{ in } p$	Définition de fonction
		e	Expression principale

(a) Syntaxe abstraite de Skimmel

(b) Syntaxe abstraite de Closed

FIGURE 1 : Syntaxes abstraites des langages de départ et d'arrivée

La seule catégorie syntaxique est celle des expressions. Celles-ci comprennent les variables, constantes littérales entières, les opérateurs prédéfinis, les fonctions anonymes (récursives ou non) et leur application, les définitions locales de variables, les conditionnelles et les opérations relatives aux paires. Quelques points sont à noter.

1. Le langage ne manipule que des entiers machine signés comme valeurs de base, les valeurs logiques étant représentées comme en langage C : l'entier 0 représente le "faux" logique tandis que toute valeur non-nulle représente le "vrai".
2. Les fonctions anonymes peuvent être récursives via une construction *ad-hoc* notée $\text{fun } x \text{ rec } f \rightarrow e$ qui indique que les appels récursifs peuvent être faits dans le corps e en appliquant f . Ce choix simplificateur éloigne Skimmel d'OCaml, où la récursion est liée à la définition de fonctions nommées via le mot-clef **let rec**.
3. Les opérateurs prédéfinis sont des fonctions. Ils comprennent les opérations arithmétiques et logiques de base, les comparaisons d'entiers, ainsi qu'une opération permettant d'afficher un entier sur la sortie standard du programme. Les expressions Skimmel peuvent donc réaliser des effets de bords.

Le langage impose un ordre d'évaluation de droite à gauche pour fixer le sens des expressions contenant des effets de bords. Par exemple, l'expression $(\text{print_int } 1 ; \text{fun } x \rightarrow x) (\text{print_int } 2 ; 42)$ s'évalue vers l'entier 42 en affichant d'abord 2 puis 1 sur la sortie standard.

L'analyseur lexical et syntaxique de Skimmel implémente plusieurs raccourcis notationnels qui facilitent l'écriture des programmes. Il est par exemple possible d'écrire dans un code source l'expression **let rec** $f \ x_1 \dots x_K = e_1 \text{ in } e_2$, qui n'existe pas dans l'arbre de syntaxe de Skimmel mais sera développée vers l'expression **let** $f = \text{fun } x_1 \text{ rec } f \rightarrow \dots \rightarrow \text{fun } x_K \rightarrow e_1 \text{ in } e_2$ qui, elle, existe dans l'arbre de syntaxe.

Langage d'arrivée Le langage d'arrivée, nommé *Closed*, est très proche de *Skimmel*. Sa grammaire est présentée en forme de Backus-Naur à la figure 1b. Cette grammaire utilise quelques raccourcis de notation : la notation C^* désigne une suite de longueur finie potentiellement vide de symboles (terminaux ou non terminaux) appartenant à la catégorie syntaxique C (par exemple celle des expressions e ou celle des identifiants x), et la notation $C^?$ désigne un symbole de la catégorie syntaxique C optionnel.

Il s'agit toujours d'un langage fonctionnel, au sens où les fonctions sont des valeurs de première classe, mais à l'inverse de *Skimmel* toutes les fonctions y sont nommées et closes. On y distingue par conséquent la catégorie syntaxique des expressions de celle des *programmes*. Un programme est essentiellement une liste de définitions de fonctions nommées, chacune étant possiblement récursive (comme indiqué par le mot-clef optionnel **rec**), précédant une expression principale dont la valeur et les effets de bords détermineront ceux du programme complet.

Le langage des expressions de *Closed* ne comprend plus de fonctions anonymes, mais il propose de nouvelles constructions issues de la programmation impérative. Celles-ci manipulent des *blocs*, c'est-à-dire des tableaux hétérogènes de taille fixe. Ces tableaux sont hétérogènes au sens où leurs cellules peuvent contenir des données de types différents ; le typage est la charge du programme. De même, les blocs sont alloués non initialisés, et le programme doit s'assurer de ne jamais lire une cellule sans y avoir précédemment écrit. L'expression **alloc** e alloue un bloc de taille e , indicé de 0 à $e - 1$ inclus. L'expression $e_1.(e_2) \leftarrow e_3$ modifie la cellule du bloc e_1 située à l'indice e_2 pour qu'elle contienne e_3 . Enfin, l'expression $e_1.(e_2)$ s'évalue vers la valeur contenue dans la cellule du bloc e_1 située à l'indice e_2 . Comme le reste du langage, ces constructions supposent que les expressions e, e_1, e_2, e_3 soient du type attendu, par exemple e_1 doit s'évaluer vers un bloc tandis que e et e_2 doivent s'évaluer vers des entiers.

Une dernière différence entre *Skimmel* et *Closed*, mineure, est que dans ce dernier les opérateurs prédéfinis ne sont plus traités comme des fonctions. Ils ne peuvent apparaître que totalement appliqués.

L'explicitation des fermetures Pour illustrer le processus d'explicitation des fermetures, nous utiliserons des exemples écrits en OCaml, qui y jouera à la fois le rôle de langage source et de langage cible. On y négligera les contraintes de typage.

Informellement, une *fermeture* est une structure de données simple qui regroupe une référence vers une fonction et un certain nombre de valeurs auquel le corps de la dite fonction doit avoir accès. On utilise les fermetures pour simuler les fonctions dont le corps est ouvert¹ dans un langage qui en est dépourvu. Dans le code *Closed* que vous devez générer, les fermetures seront implémentées par des blocs de taille $n + 1$, où n est le nombre de variables libres de la fonction *Skimmel* à traduire.

Dans les exemples en OCaml de cette section, ces blocs seront simulés par des tableaux. Pour ce faire, on suppose disponible une fonction **alloc** qui alloue un tableau sans l'initialiser, et on relâche le typage d'OCaml pour autoriser les cellules d'un même tableau à contenir des valeurs de types différents, comme c'est le cas en *Closed*.

Considérons par exemple l'expression OCaml donnée à la figure 2a. Elle construit une fonction nommée **f** et l'applique immédiatement à l'entier 40. Le corps de la fonction **f** contient une variable libre, à savoir **x**, définie précédemment comme valant 2. Cet exemple illustre en particulier la gestion *lexicale* de la portée en OCaml : la valeur de **x** pertinente lors de l'application de **f** est celle en vigueur au point de création de la fonction, soit 2, plutôt que celle en vigueur au point d'appel, soit 10. Le résultat de l'expression complète est donc l'entier 42.

Un processus systématique d'explicitation des fermetures peut traduire l'expression précédente vers le programme OCaml présenté à la figure 2b. La sous-expression fonctionnelle **fun** $y \rightarrow y + x$ a donné lieu à deux fragments de code dans le programme généré : la fonction nommée **anon0** et l'expression construisant la fermeture dédiée à **anon0**.

1. La fonction **anon0** a été obtenue via deux modifications.
 - a) On a ajouté au paramètre original de la fonction un nouveau paramètre destiné à recevoir la fermeture. Dans l'exemple, il s'agit du premier paramètre, nommé **self**.
 - b) On a modifié le corps de la fonction pour y lire les valeurs des variables précédemment libres depuis la fermeture reçue via le premier paramètre. Dans l'exemple, il s'agit de la définition

1. On rappelle qu'une expression est *ouverte* lorsqu'elle contient des variables libres.

```

1 let x = 2 in
2 let f = fun y -> y + x in
3 let x = 10 in
4 f 40

```

(a) Code source en Skimmel

```

1 let anon0 self y =
2   let x = self.(1) in y + x
3 in
4 let x = 2 in
5 let f =
6   let clo0 = alloc 2 in
7   clo0.(0) <- anon0;
8   clo0.(1) <- x; (* non typé *)
9   clo0
10 in
11 let x = 10 in
12 f.(0) f 40

```

(b) Code traduit en Closed

FIGURE 2 : Premier exemple d'explicitation des fermetures

```

1 let x = 2 in
2 let sum =
3   fun n rec sum ->
4     if n <= 0 then 0
5     else x + sum (n - 1) in
6 sum 4

```

(a) Code source en Skimmel

```

1 let anon0 self n =
2   let sum = self in
3   let x = self.(1) in
4   if n <= 0
5   then 0
6   else x + sum.(0) sum (n - 1)
7 in
8 let x = 2 in
9 let sum =
10   let clo0 = alloc 2 in
11   clo0.(0) <- anon0;
12   clo0.(1) <- x;
13   clo0
14 in
15 sum.(0) sum 4

```

(b) Code traduit en Closed

FIGURE 3 : Deuxième exemple d'explicitation des fermetures

locale `let x = self.(1) in ...` qui charge la valeur de la variable libre `x` depuis à la deuxième cellule du bloc.

2. La construction de la fermeture alloue un bloc de taille 2, la première cellule étant destinée à contenir une référence vers la fonction `anon0` et la seconde la valeur de la variable libre `x`.

Le deuxième exemple (figures 3a et 3b) illustre une manière de traiter les fonctions récursives. Chaque fonction recevant sa fermeture en argument, il suffit d'utiliser celle-ci pour les appels récursifs. Dans l'exemple, on définit l'identifiant `sum` pour le lier à cette fermeture dans le corps de la fonction. Cette approche permet de ne pas traiter spécialement le corps d'une fonction récursive. Une alternative moins commode serait de renommer `sum` en `self` à la volée dans le corps de la fonction.

Pour Skimmel et Closed et beaucoup d'autres langages fonctionnels, dont OCaml, une fonction à $n > 1$ arguments est traitée comme une fonction à 1 argument qui renvoie une fonction à $n - 1$ arguments, et ainsi de suite récursivement. Le dernier exemple (figures 4a et 4b) est une variation du précédent qui n'introduit pas de nouveau concept mais illustre comment l'explicitation des fermetures interagit avec ce point de vue sur les fonctions à plusieurs arguments. Ici, une fonction à deux arguments est traduite vers la construction d'une fermeture dont l'application au premier argument `x` produit une nouvelle

```

1 let sum =
2   fun x rec sum ->
3   fun n ->
4     if n <= 0 then 0
5     else x + sum x (n - 1)
6 in
7 sum 2 4

```

(a) Code source en Skimmel

```

1 let anon0 self n =
2   let x = self.(1) in
3   let sum = self.(2) in
4   if n <= 0
5   then 0
6   else x +
7     (let f0 = sum.(0) sum x
8      in f0.(0) f0 (n - 1))
9 in
10 let anon1 self x =
11   let sum = self in
12   let clo0 = alloc 3 in
13   clo0.(0) <- anon0;
14   clo0.(1) <- x;
15   clo0.(2) <- sum;
16   clo0
17 in
18 let f1 =
19   (let clo1 = alloc 1 in
20    clo1.(0) <- anon1;
21    clo1)
22 in
23 let f2 = f1.(0) f1 2 in
24 f2.(0) f2 4

```

(b) Code traduit en Closed

FIGURE 4 : Troisième exemple d'explicitation des fermetures

fermeture qui attend le deuxième argument n . La définition de nouvelles variables comme $f0$, $f1$ et $f2$ se révèle commode pour éviter de dupliquer les calculs de fermetures. Une traduction plus naïve pourrait même produire un programme Closed contenant plus de variables que celui montré à la figure 4b.

1.2 Organisation du code fourni

Le code attendu doit manipuler les arbres de syntaxe des deux langages en s'insérant dans une base de code fournie petite mais structurée. Nous allons décrire cette base de code, ainsi que les outils permettant de compiler celle-ci, d'exécuter le projet, de lancer les tests fournis et d'en écrire d'autres de votre cru. Cette base de code est d'une taille relativement importante, et **vous ne pourrez pas la lire entièrement** dans le temps imparti. Il vous faut donc vous concentrer sur la lecture de certains fichiers d'interface (d'extension `.mli`) de modules mentionnés plus bas.

Le code qui vous est fourni définit l'arbre de syntaxe abstraite ainsi qu'un évaluateur pour Skimmel comme pour Closed. Ces évaluateurs définissent la sémantique de chacun des deux langages, et peuvent être utilisés pour tester votre implémentation de l'explicitation des fermetures. De plus, le langage Skimmel est fourni avec un analyseur lexical et syntaxique, ce qui permet de tester l'explicitation des fermetures facilement en lisant du code source Skimmel depuis le système de fichiers.

Architecture logicielle. Le logiciel que vous devez compléter pour cette partie du sujet est fourni dans le répertoire `partie1`, dont nous allons décrire les sous-répertoires.

- Le répertoire `tests` contient les tests fournis, qui servent également de documentation. Le fonctionnement de ces tests est détaillé plus bas.
- Le répertoire `doc` contient les instructions pour le système de compilation permettant de produire la documentation. Vous pouvez ignorer le contenu de ce répertoire.

- Le répertoire `lib` contient le code OCaml fourni ou à compléter.
 - Le fichier `Skimmel.mli` contient la définition de l'arbre de syntaxe du langage `Skimmel`, ainsi que le code qui ne traite que de ce langage. Vous devez lire à minima la définition du type `Skimmel.t`.
 - Le fichier `Closed.mli` contient la définition de l'arbre de syntaxe du langage `Closed`, ainsi que le code qui ne traite que de ce langage. Vous devez lire à minima les définitions de types que ce fichier contient, en particulier `Closed.exp` et `Closed.t`.
 - Les autres fichiers du répertoire `lib` contiennent du code utilitaire indépendant du langage considéré. Vous devez lire à minima le fichier `closed/Id.mli`.
 - Le fichier `TP.ml` est le seul que vous devez modifier. Il contient des fonctions à compléter, ainsi que des tests automatiques qui peuvent vous guider dans cette tâche.
- Le répertoire `bin` contient la fonction principale du programme, qui appelle notamment le code que vous allez écrire. **Vous n'avez pas à vous en soucier.**

Compiler le projet, l'exécuter, lancer les tests. Le projet utilise Dune, l'outil logiciel qui constitue le standard de fait pour la compilation des projets OCaml. Celui-ci permet de compiler le projet, produire la documentation, exécuter le compilateur et exécuter les tests. **Les explications qui suivent supposent que vous lanciez les commandes spécifiées depuis le répertoire `partie1`. Celui-ci contient un fichier `.ocamlinit` qui ouvre la bibliothèque `Lib` ce qui est nécessaire aux fragments de code qui suivent.**

- La commande `dune build` lance la compilation du projet. Celle-ci produit en particulier le binaire exécutable du compilateur, qu'on peut exécuter par la commande `./bin/main.exe`.
- La commande `dune build @doc-new` produit la documentation du projet et de ses dépendances au format HTML². Par souci de commodité, nous vous fournissons une version préconstruite de cette documentation dans le dossier `html-doc`. Vous pouvez utiliser n'importe quel navigateur web pour la consulter, par exemple via la commande shell `firefox html-doc/index.html`. La documentation du code du projet est accessible sous le libellé *Private libraries* depuis ce fichier HTML.
- La commande `dune runtest` lance les tests situés à la fois dans le code OCaml du fichier `lib/TP.ml`, et dans le répertoire `tests`. Ces tests **n'affichent rien en cas de succès**.
- La commande `dune utop` lance une boucle interactive OCaml dans laquelle les bibliothèques du projet sont chargées. Cette commande peut vous servir à expérimenter avec le code que vous avez fourni ou que vous avez écrit.
- La commande `dune clean` efface tous les fichiers générés. Son usage n'est *a priori* pas nécessaire. Vous êtes encouragés à expérimenter avec vos propres tests.

Les tests. Le projet vient accompagner de tests unitaires et de tests d'intégration, tous lancés par la commande `dune runtest`.

Les tests unitaires sont contenus dans le fichier `TP.ml`, après les fonctions que vous avez à compléter. Ils spécifient les résultats attendus de vos fonctions sur quelques exemples.

Les tests d'intégration fournis sont spécifiés au format *cram* et contenus dans les fichiers `run.t` des sous-répertoires du répertoire `tests`. Les tests au format *cram* sont des fichiers texte contenant :

1. du texte arbitraire, à des fins de documentation ;
2. des commandes shell ;
3. la sortie attendue des dites commandes.

La commande `dune runtest` exécute les commandes shell contenues dans chaque fichier et compare leur sortie à la sortie attendue spécifiée dans le test *cram*. Toute différence est considérée comme une erreur et affichée en couleurs sur la sortie standard.

Remarquons pour terminer que si les tests et explications attenantes sont fournis pour vous guider, **passer tous les tests ne garantit pas la correction complète de votre solution.**

2. Cette commande peut éventuellement afficher des erreurs liées à l'absence de documentation pour la bibliothèque standard d'OCaml. Celles-ci sont sans gravité.

```

val singleton : Id.t -> Id.Set.t
val empty : Id.Set.t
val union : Id.Set.t -> Id.Set.t -> Id.Set.t
val remove : Id.t -> Id.Set.t -> Id.Set.t

```

FIGURE 5 : Quelques fonctions utiles du module `Id.Set`

1.3 Questions

On attend que votre code soit contenu dans le fichier `lib/TP.ml`. Il n'est *a priori* pas nécessaire de modifier le reste des fichiers fournis.

1.3.1 Prise en main

Prenez le temps de vous familiariser avec l'architecture logicielle du projet, en relisant la section 1.2 tout en parcourant l'arborescence de fichiers et la documentation générée. Il est particulièrement important de comprendre les définitions contenues dans les fichiers `Skimmel.mli` et `Closed.mli`.

- Q1)** Évaluer l'expression OCaml ci-dessous dans la boucle d'interaction `utop` (lancée en exécutant la commande shell `dune utop` comme expliqué à la section 1.2). Que représente la valeur affichée ?
`utop # let e = Parse.string "let f = fun x -> x + 1 in f 40";;`
- Q2)** Évaluer l'expression `Skimmel` obtenue à la question précédente via l'infrastructure du projet, par exemple via la commande suivante lancée dans la même session de la boucle d'interaction.
`utop # Skimmel.Eval.eval Skimmel.Eval.initial_env e;;`
- Q3)** Donner les valeurs OCaml de type `Skimmel.t` qui représentent les codes des figures 2a et 3a en utilisant le module `Parse`, à la manière de la première question.
- Q4)** Lister les constructeurs du type `Closed.t` qui sont utilisés pour représenter le code de la figure 2b. Vous n'avez pas à construire l'arbre de syntaxe complet.
- Q5)** Écrire les codes sources des figures 2a et 3a dans des fichiers sur disque, puis les modifier pour afficher leur résultat via l'opérateur prédéfini `print_int`. Utiliser ensuite l'infrastructure du projet pour évaluer les programmes en question, par exemple en exécutant la commande suivante dans votre shell.
`$ ./bin/main.exe -e exemple_2a.skm exemple_3a.skm`
Vérifier que les résultats affichés sont, comme attendu, les entiers 42 et 8.

1.3.2 Calcul des variables libres

L'explicitation des fermetures doit calculer les identifiants libres d'une expression pour plusieurs usages, notamment déterminer la taille du bloc implémentant la fermeture et calculer quelles valeurs doivent être relues depuis celle-ci. Pour représenter un ensemble fini d'identifiant, on utilisera le module `Common.Id.Set` dont le type `t` représente des ensembles finis d'identifiants et qui offre des fonctions permettant de manipuler ce type. Les plus utiles d'entre elles, aux noms auto-descriptifs, sont rappelées à la figure 5.

- Q6)** Compléter le code des fonctions `free_variables` et `free_variables_scoped` qui calculent respectivement l'ensemble des variables libres d'une expression `Skimmel` et celui d'une expression `Skimmel` placée sous une variable liée. Vous pouvez tester votre code via la boucle d'interaction, le listing ci-dessous vous fournissant un exemple.
`utop # TP.free_variables (Parse.string "(fun x -> x + y) z");;`
`- : Common.Id.Set.t = { y, z }`
`utop # TP.free_variables (Parse.string "(fun x -> x + y) x");;`
`- : Common.Id.Set.t = { x, y }`

```

utop # TP.free_variables (Parse.string "let x = y in x + z");;
- : Common.Id.Set.t = { y, z }

```

1.3.3 Fonctions et fonctions récursives

Le squelette du code qui implémente l'explicitation des fermetures vous est fourni sous la forme de la fonction `close_exp`. Il s'agit de la fonction récursive qui parcourt l'arbre de syntaxe. Vous devez lire celle-ci avec attention avant de répondre aux questions suivantes.

Q7) Compléter le code de la fonction `apply_closure`. Celle-ci manipule des expressions `Closed` pour réaliser l'application d'une fermeture selon les principes décrits à la section 1.1. Elle reçoit deux expressions `e1` et `e2`, où `e1` s'évalue vers une fermeture et `e2` vers son argument, et doit construire l'expression `apply_closure e1 e2` qui représente l'application de `e1` à `e2`.

Dans le reste du code qui vous est fourni, `e1` et `e2` seront les résultats de la traduction d'expression Skimmel. Appliquée à l'arbre de syntaxe Skimmel correspondant au code source donné à la figure 2, la fonction `close_exp` appellera votre fonction `apply_closure` une seule fois en lui passant comme arguments `e1` et `e2` les arbres de syntaxe `Closed` représentant respectivement `f` et `40`. Votre fonction pourra donc produire l'arbre de syntaxe correspondant au code source `f.(0) f 40`. Notez que traiter des cas plus généraux, notamment ceux où `e1` et `e2` comprennent des effets de bords, peut nécessiter une implémentation un peu plus complexe.

Q8) Compléter le code de la fonction `assign_positions` qui calcule les positions d'une liste de variables libres dans la fermeture. Vous pouvez suivre la spécification fournie dans le code source.

Q9) Compléter le code de la fonction `build_closure` qui calcule l'expression construisant une fermeture pour une fonction déjà nommée. Vous pouvez suivre la spécification fournie dans le code source. À titre d'exemple, cette fonction doit produire l'arbre de syntaxe correspondant aux lignes 6 à 9 de la figure 2b.

Q10) Compléter le code de la fonction `build_named_function` qui construit la fonction nommée correspondant à une fonction anonyme. Vous pouvez suivre la spécification fournie dans le code source. À titre d'exemple, cette fonction doit produire l'arbre de syntaxe correspondant aux lignes 1 à 3 de la figure 2b. Vous pouvez négliger le cas des fonctions récursives dans un premier temps en supposant que le paramètre `self` vaut toujours `None`.

1.3.4 Gestion des opérateurs prédéfinis

Même après avoir implémenté les fonctions des questions précédentes, votre code ne traite a priori pas les applications partielles d'opérateurs prédéfinis. Celles-ci sont difficiles à prendre en compte dans la fonction `close_exp` sans y introduire de cas particuliers désagréables. Une solution simple consiste à les éliminer par une réécriture préalable de l'arbre de syntaxe Skimmel, ce qui évitera d'avoir à modifier `close_exp`.

Q11) Implémenter la fonction `eta_expand_builtins` qui réécrit une expression Skimmel pour s'assurer que tous les opérateurs prédéfinis y sont appliqués totalement.

2 Gestion automatique de la mémoire – programmation C

Dans les langages de haut niveau, la libération de la mémoire dynamique (pour les données dont la durée de vie n'est pas connue à la compilation, comme les fermetures ou les listes chaînées) n'est pas à la charge du programmeur : un *ramasse-miettes* (ou *garbage collector* en anglais) nettoie la mémoire non utilisée quand nécessaire. Le but de cette partie est d'écrire en langage C une simulation de ramasse-miettes.

À noter que les langages possédant des ramasse-miettes ne contiennent pas d'arithmétique des pointeurs, ce qui interdit en particulier de stocker l'adresse d'une case de milieu de tableau. On se place donc dans ce cadre pour la simulation (mais on ne s'interdit pas d'utiliser l'arithmétique des pointeurs dans le code à écrire).

On suppose pour simplifier que les fonctions dont on simule l'appel prennent exactement un paramètre et ne renvoient rien.

Tout le code fourni est déclaré dans le fichier `code.h` et défini dans le fichier `code.c`. Les constantes macro-définies (et uniquement elles) sont écrites en majuscules.

Le tableau ci-dessous donne une partie des fonctions fournies.

fonction	rôle
<code>void init_mem_process();</code>	prépare la mémoire simulée
<code>void entree_appel(donnee_tt x);</code>	simule un nouvel appel de fonction
<code>void sortie_appel();</code>	simule la sortie de l'appel courant
<code>void print_pile();</code>	affiche le contenu de la pile simulée
<code>void print_tas(bool reserve);</code>	affiche le contenu du tas simulé

L'annexe contient des exemples de fonctions `main` et d'affichages associés pour clarifier le propos. Les sources de ces exemples sont disponibles dans le fichier `exemples.c`. N'hésitez pas à aller les consulter pour mieux comprendre ce qui est demandé. L'énoncé indique explicitement quand une aide visuelle est disponible avec un encadré, comme celui-ci : `main1`.

2.1 Organisation de la mémoire

Dans la simulation que nous allons implémenter, nous disposons de deux emplacements mémoire alloués au début de l'exécution par la fonction `void init_mem_process();` et qui sont représentés par des variables globales :

- un emplacement `pile` qui nous servira à représenter la pile d'appel du processus;
- un emplacement `tas` qui nous servira à gérer la mémoire dynamique, et qui est découpé en deux zones mémoires `tas.donnees` et `tas.non_utilise`.

La fonction `init_mem_process` lance également le premier appel courant (simulant ainsi la fonction `main`). `main1`

Dans votre code il est strictement interdit d'utiliser les fonctions d'allocation et de libération de mémoire fournies par le langage C.

Notre simulation gère des données d'un seul type : `donnee_tt`.

Le type `donnee_tt` est un entier non signé et sert à représenter aussi bien des entiers que des emplacements dans `pile` et `tas`. À l'instar de ce qui se passe en OCaml, nous utilisons un des bits du type `donnee_tt` pour distinguer les entiers des emplacements. Ici c'est le bit de poids fort selon la règle définie dans la table ci-dessous (où `ADRESSE` désigne le plus grand entier non signé de type `donnee_tt` possédant un seul bit à 1).

sémantique	valeur minimale	valeur maximale
entier	0	<code>ADRESSE - 1</code>
dans <code>pile</code>	<code>ADRESSE</code>	<code>ADRESSE (TAILLE_PILE - 1)</code>
dans <code>tas</code>	<code>ADRESSE DEBUT_TAS</code>	<code>ADRESSE (DEBUT_TAS + TAILLE_TAS - 1)</code>

Une instance de `donnee_tt` représentant un entier a pour valeur elle-même.

Une instance de `donnee_tt` représentant une adresse dans `tas` est interprétée de la manière suivante :

- on translate la valeur de sorte que l'image de la valeur minimale d'un emplacement de `tas` soit ramenée à 0,
- on obtient alors un numéro de case dans `tas.donnees` qui est l'emplacement recherché.

La *pile* représente une pile d'instances de `struct bloc`. Pour simplifier, nous supposons que tous les blocs ont même taille, le nombre de blocs maximal autorisé dans la pile étant `NB_BLOCS`.

Un bloc est composé d'un champ `local` qui permet de stocker l'équivalent simulé de variables locales et contient `TAILLE_BLOC` cases. Lors de la simulation d'un appel de fonction, l'argument fourni à `entree_appel` (qui est supposé être celui fourni à la fonction simulée) sera stocké dans la première case de ce champ, les autres champs recevant la valeur 0.

Une instance de `donnee_tt` représentant une adresse dans *pile* est interprétée de la manière suivante :

- on translate la valeur de sorte que l'image de la valeur minimale d'un emplacement de *pile* soit ramenée à 0,
- le quotient de cette valeur par `TAILLE_BLOC` donne l'indice du bloc concerné dans `pile.b`,
- le reste fournit un indice pour le champ `local` de ce bloc.

Pour éviter toute ambiguïté de vocabulaire, dans la suite du sujet, de telles adresses seront appelées des *positions*, le mot *adresse* étant réservé aux adresses en C. Une position est donc une valeur de type `donnee_tt` dont le bit de poids fort est à 1 et qui permet d'identifier un emplacement situé dans *pile* ou dans *tas*, en utilisant les règles exposées ci-dessus.

Q12) Écrire des fonctions

```
void set_valeur_pile(uint32_t i, donnee_tt val);
donnee_tt get_valeur_pile(uint32_t i);
```

qui permettent respectivement de donner la valeur `val` à la variable locale d'indice `i` du bloc d'appel courant ou de récupérer la valeur de cette même variable. Si `i` ne correspond pas à l'indice d'une variable locale, ces fonctions doivent afficher un message `débordement de pile` et l'exécution doit s'arrêter. `main2`

Q13) Écrire une fonction `uint32_t type_donnee(donnee_tt x)`; qui permet de déterminer si l'argument qui lui est transmis représente une position et si cette position se trouve dans *pile* ou dans *tas*.

La valeur renvoyée par cette fonction doit être une des valeurs suivantes : `ETAS`, `EPILE`, `ENOMEM` et `ENOVALID`. La sémantique de ces constantes est explicitée dans le fichier `code.h`

Q14) Écrire une fonction `void set_contenu(donnee_tt x, donnee_tt val, bool tas_courant)`; qui prend en argument une donnée `x`, une valeur `val` et un booléen `tas_courant`, et :

- si `x` n'est pas une position dans *pile* ou dans *tas*, alors l'exécution doit s'arrêter avec un message `erreur de segmentation`;
- si `x` est une position dans la pile, alors la fonction y stocke la valeur `val`;
- si `x` est une position dans le tas, alors si `tas_courant` vaut `true`, la valeur doit être stockée dans `tas.donnees`, et sinon dans le `tas.non_utilise`.

(Cette dernière fonctionnalité peut paraître artificielle, mais elle pourra être utile au moment de l'implémentation du ramasse-miettes.) `main3`

2.2 Allocation

Pour éviter toute ambiguïté, on dira qu'une position dans le tas est *réservée* par notre simulation, le mot *alloué* restant dédié aux adresses que renvoient les fonctions de la famille de `malloc`.

Les deux zones `tas.donnees` et `tas.non_utilise` de `tas` ont même taille : `TAILLE_TAS`.

La zone `tas.donnees` est celle qui est utilisée. L'autre zone sert au moment du passage du ramasse-miettes. Le remplissage du tas se fait par positions croissantes, la valeur `tas.taille` donne le nombre de cases de `tas.donnees` qui ont déjà été réservées par notre simulation. La valeur `tas.taille_nu` donne le nombre de cases de `tas.non_utilise` qui étaient réservées avant le dernier passage du ramasse-miettes (et sert essentiellement à la fonction d'affichage qui vous est fournie), elle est initialement vide.

2.2.1 Il reste de la place dans `tas.donnees`

Quand l'utilisateur demande un emplacement mémoire en spécifiant le nombre de cases souhaité sous forme d'un entier non signé `nb_cases`, et qu'il reste suffisamment de place dans `tas.donnees`, on doit prévoir de la place pour `nb_cases+2` cases contenant chacune une valeur de type `donnee_tt` :

- dans la première case, on met la valeur `nb_cases` ;
- dans la dernière case, on met 0 ; cette case servira plus tard pour le ramasse-miettes, c'est la case de *redirection* ;
- on renvoie la position `p` (et non l'indice) de la première case où l'utilisateur pourra stocker des données (la case qui suit celle où on a stocké `nb_cases`).

Nous appellerons cet objet la *table* (de taille `nb_cases`) à la position `p`.

Q15) Écrire une fonction `donnee_tt alloc_si_place(uint32_t nb_cases)` ; qui effectue l'opération décrite ci-dessus, sans avoir à vérifier qu'il y a effectivement suffisamment de place dans `tas.donnee`. main4

2.2.2 Il n'y a plus de place dans `tas.donnees`

S'il n'y a plus assez de place dans `tas` pour l'allocation demandée, il faut faire passer le ramasse-miettes. Nous rappelons que dans les langages que nous simulons, il n'est pas possible de considérer la position d'une case particulière dans une table, dans le tas les seules positions manipulables (et stockables) sont les positions des tables elles-mêmes, telles que définies plus haut.

Il s'agit ici d'implémenter l'algorithme de Cheney.

On considère un graphe orienté dans lequel les sommets sont les positions dans `pile` et `tas`, et il y a un arc $s \rightarrow t$ si et seulement si la valeur à la position `s` est la position `t`. Les positions qui ne sont pas atteignables dans ce graphe à partir d'une position de `pile` sont devenues inutiles.

L'algorithme de Cheney consiste à faire un parcours en largeur de ce graphe à partir des positions dans `pile` et à recopier dans `tas.non_utilise` les cases atteignables de `tas.donnees`, dans l'ordre des positions croissantes. Quand toutes les tables atteignables ont été copiées, on inverse les rôles des deux moitiés du tas. De façon plus précise (et illustrée en partant de la situation décrite en Figure 6 main5) :

- On parcourt les valeurs `x` contenues dans les cases des blocs de la pile (voir Figure 7) :
 - si `x` est une position dans le tas :
 - soit la redirection a été mise en place (elle est non nulle), et on met à jour la valeur dans la pile avec la redirection,
 - soit ce n'est pas le cas, alors on recopie la table correspondante dans `tas.non_utilise` (y compris la taille) avec une redirection nulle, et on met en place la redirection dans `tas.donnees` pour cette table.
- Tant que c'est possible, on parcourt les valeurs contenues dans les tables de `tas.non_utilise` et on effectue les mêmes opérations que ci-dessus (voir Figure 8).
- On inverse les rôles de `tas.donnees` et `tas.non_utilise` (voir Figure 9).

Q16) Écrire une fonction `void ramasse_miettes()` ; qui implémente l'algorithme de Cheney. main5

Q17) Écrire une fonction `donnee_tt alloc(uint32_t nb_cases)` ; qui effectue l'allocation demandée, quitte à lancer le ramasse-miettes si besoin. S'il n'y a pas assez de place dans `tas`, même après le passage du ramasse-miettes, la fonction doit renvoyer `NUL`.

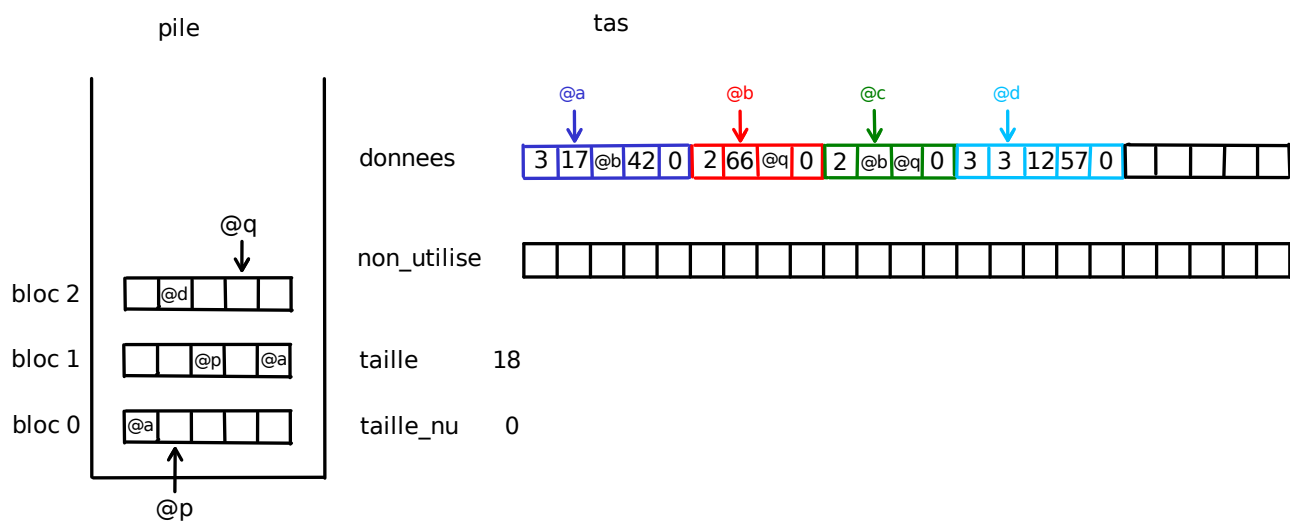
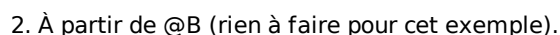


FIGURE 6 : État de la mémoire avant le passage du ramasse-miettes : les notations de type `@a` indiquent une position dans *pile* ou dans *tas.donnees*, celles du type `@A` indiquent une position dans *tas.non_utilise*.

1. À partir de @A, puis de @D.



III - Finalisation (changements en **turquoise**) : inversion des rôles + mise à jour de la taille



14

2.3 Annexe

Tous les exemples sont écrits avec TAILLE_BLOC à 5u, TAILLE_PILE suffisante et TAILLE_TAS à 23u. Ils sont disponibles dans le fichier `exemples.c`.

main1

```
1 int main1(void){
2     init_mem_process();
3     print_pile();
4     print_tas(false);
5     return 0;
6 }

***pile***
* bloc 0 @2147483648 : 0 | 0 | 0 | 0 | 0 | 0 |
###tas courant###
```

main2

```
1 int main2(void){
2     init_mem_process();
3     entree_appel(5);
4     print_pile();
5     print_tas(false);
6     set_valeur_pile(0, 10);
7     set_valeur_pile(1, 20);
8     set_valeur_pile(3, 36);
9     set_valeur_pile(3, 43);
10    print_pile();
11    print_tas(false);
12    printf("\nbloc courant : ");
13    for(uint32_t i=0; i<TAILLE_BLOC; i++)
14        printf("%u ", get_valeur_pile(i));
15    printf("\n");
16    return 0;
17 }

***pile***
* bloc 0 @2147483648 : 0 | 0 | 0 | 0 | 0 | 0 |
* bloc 1 @2147483653 : 5 | 0 | 0 | 0 | 0 | 0 |
###tas courant###

***pile***
* bloc 0 @2147483648 : 0 | 0 | 0 | 0 | 0 | 0 |
* bloc 1 @2147483653 : 10 | 20 | 0 | 43 | 0 | 0 |
###tas courant###

bloc courant : 10 20 0 43 0
```

main3

```

1  int main3(void){
2      init_mem_process();
3      set_contenu(ADRESSE+2, 42, true);
4      entree_appel(5);
5      print_pile();
6      print_tas(false);
7      set_contenu(ADRESSE+TAILLE_BLOC+1, 66, true);
8      donnee_tt d1 = alloc(3);
9      set_valeur_pile(2, d1);
10     set_contenu(d1, 17, true);
11     print_pile();
12     print_tas(false);
13     return 0;
14 }
```

```

***pile***
* bloc 0 @2147483648 : 0 | 0 | 42 | 0 | 0 |
* bloc 1 @2147483653 : 5 | 0 | 0 | 0 | 0 |

##tas courant##

***pile***
* bloc 0 @2147483648 : 0 | 0 | 42 | 0 | 0 |
* bloc 1 @2147483653 : 5 | 66 | 2147483857 | 0 | 0 |

##tas courant##
# @2147483857 (3 cases) : 17 | 0 | 0 | redirection -> 0
```

main4

```

1  int main4(void){
2      init_mem_process();
3      entree_appel(42);
4      set_valeur_pile(0, alloc_si_place(5));
5      set_valeur_pile(1, alloc_si_place(4));
6      print_pile();
7      print_tas(false);
8      return 0;
9 }
```

```

***pile***
* bloc 0 @2147483648 : 0 | 0 | 0 | 0 | 0 |
* bloc 1 @2147483653 : 2147483857 | 2147483864 | 0 | 0 | 0 |

##tas courant##
# @2147483857 (5 cases) : 0 | 0 | 0 | 0 | 0 | redirection -> 0
# @2147483864 (4 cases) : 0 | 0 | 0 | 0 | redirection -> 0
```

main5

```

1  int main5(void){
2      init_mem_process();
3      donnee_tt a = alloc(3);
4      donnee_tt b = alloc(2);
5      donnee_tt c = alloc(2);
6      donnee_tt d = alloc(3);
7      set_contenu(a, 17, true);
8      set_contenu(a+1, b, true);
9      set_contenu(a+2, 42, true);
10     set_contenu(b, 66, true);
11     set_contenu(d, 3, true);
12     set_contenu(c, b, true);
13     set_contenu(d+1, 12, true);
14     set_contenu(d+2, 57, true);
15     set_contenu(ADRESSE, a, true);
16     entree_appel(0);
17     set_valeur_pile(2, ADRESSE|1);
18     set_valeur_pile(4, a);
19     entree_appel(0);
20     set_valeur_pile(1, d);
21     set_contenu(b+1, ADRESSE|(2*TAILLE_BLOC+3), true);
22     set_contenu(c+1, ADRESSE|(2*TAILLE_BLOC+3), true);
23     print_pile();
24     print_tas(true);
25     printf("\n...passage du ramasse-miettes...\n");
26     ramasse_miettes();
27     print_pile();
28     print_tas(true);
29     return 0;
30 }
```

```

***pile***
* bloc 0 @2147483648 : 2147483857 | 0 | 0 | 0 | 0 |
* bloc 1 @2147483653 : 0 | 0 | 2147483649 | 0 | 2147483857 |
* bloc 2 @2147483658 : 0 | 2147483870 | 0 | 0 | 0 |

###tas courant###
# @2147483857 (3 cases) : 17 | 2147483862 | 42 | redirection -> 0
# @2147483862 (2 cases) : 66 | 2147483661 | redirection -> 0
# @2147483866 (2 cases) : 2147483862 | 2147483661 | redirection -> 0
# @2147483870 (3 cases) : 3 | 12 | 57 | redirection -> 0

###tas en reserve###
...passage du ramasse-miettes...

***pile***
* bloc 0 @2147483648 : 2147483857 | 0 | 0 | 0 | 0 |
* bloc 1 @2147483653 : 0 | 0 | 2147483649 | 0 | 2147483857 |
* bloc 2 @2147483658 : 0 | 2147483862 | 0 | 0 | 0 |

###tas courant###
# @2147483857 (3 cases) : 17 | 2147483862 | 42 | redirection -> 0
# @2147483862 (3 cases) : 3 | 12 | 57 | redirection -> 0
# @2147483867 (2 cases) : 66 | 2147483661 | redirection -> 0

###tas en reserve###
# @2147483857 (3 cases) : 17 | 2147483862 | 42 | redirection -> 2147483857
# @2147483862 (2 cases) : 66 | 2147483661 | redirection -> 2147483867
# @2147483866 (2 cases) : 2147483867 | 2147483661 | redirection -> 0
# @2147483870 (3 cases) : 3 | 12 | 57 | redirection -> 2147483862
```

3 Revue de code

```
1 def trace(M):
2     '''Calcule la somme des éléments diagonaux de la matrice M.'''
3     if len(L) != len(L[0]):
4         return
5     for i in range(M):
6         for j in range(M[i]):
7             if i == j:
8                 res += M[i][j]
9     return res
```

```
1 def occurrences(L):
2     Occ = [ [False] * len(L) ] * len(L)
3     for i in L:
4         j = L[i]
5         Occ[i][j] = True
6         i += 1
7     return Occ
```