Projet : Construction d'un graphe de flot de contrôle

1 Contexte

1.1 Motivation

Le but de ce projet est de mettre en oeuvre des techniques de traitement de langages de programmation en Caml avec la

- formalisation de la syntaxe d'un langage impératif en Caml
- vérification de types de ce langage
- conception d'un assembleur simple, avec une sémantique de cet assembleur
- traduction du langage impératif vers l'assembleur
- génération et l'affichage d'un graphe de flot de contrôle

Le langage d'entrée est un sous-ensemble réaliste, mais restreint, du langage C. Vous pourrez donc compiler les programmes source de ce langage aussi avec des compilateurs C traditionnels (par exemple `gcc`), pour vérifier leur correction syntaxique et pour tester le bon fonctionnement de votre traducteur.

Le traducteur produit un assembleur qui sera utilisé de deux manières :

- pour être exécuté par une machine virtuelle spécifique que vous écrirez ;
- pour être affiché comme un graphe de flot de contrôle.

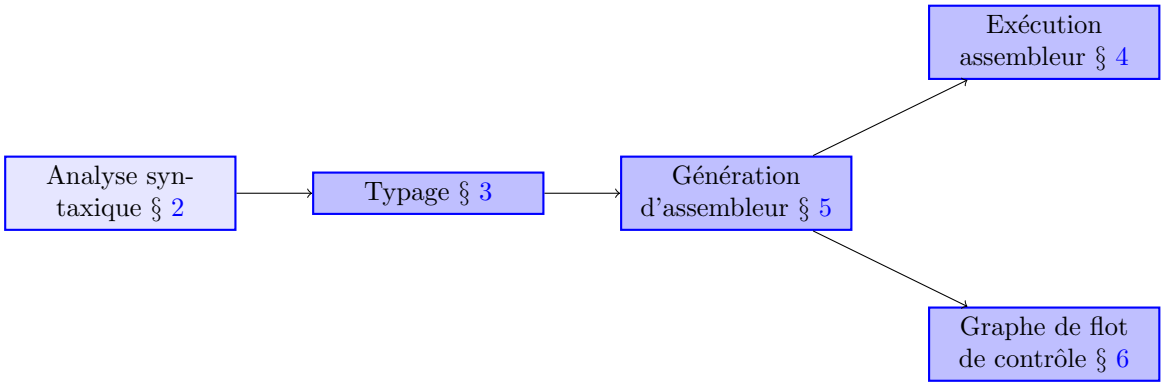

FIGURE 1 – Étapes principales du projet

Les analyses lexicale et syntaxique sont fournies, vous pourrez donc appeler un parser qui prend comme entrée un fichier contenant un programme C et qui construit un arbre syntaxique de ce programme – ou le rejette s'il est mal formé. Mis à part l'analyse syntaxique, qui pourra être réalisée à la fin du projet, il semble judicieux de procéder dans l'ordre suggéré par la numérotation des tâches. Néanmoins, elles sont largement indépendantes l'une de l'autre, et vous pouvez procéder différemment.

1.2 Travail demandé

Dates à retenir :

1. Le projet est à rendre le **mercredi 6 avril 2022 à 23h** au plus tard.

2. Une présentation par les équipes du projet aura lieu le **mercredi 13 avril 2022** dans l'après-midi. L'organisation de cette présentation se fera plus tard.

Le travail demandé est précisé dans les boîtes "Travail à réaliser" de ce document. Dans la plupart des cas, il s'agit d'écrire du code Caml dans les fichiers désignés. Parfois, il s'agit d'exercices d'évaluation ou de réflexion. Dans ce cas, mettez vos observations comme commentaire dans le fichier Caml ou dans le `README`.

Format des fichiers à rendre : Vous avez deux possibilités :

1. Vous pouvez déposer une archive (`tarfile` ou `zip`, pas de format `rar` ou autres formats propriétaires) sur Moodle, *avant la date limite*.
2. Pour vous inciter à travailler avec des systèmes de gestion de version, vous pouvez aussi nous communiquer les coordonnées d'un dépôt (seuls formats admis : `git` ou Mercurial) où nous pouvons récupérer votre code. L'adresse du dépôt doit être envoyée à martin.strecker@irit.fr avant la date limite, et le dépôt doit rester stable pendant au moins 24h après la date limite, pour nous permettre de récupérer le contenu. Évidemment, le dépôt doit être accessible en mode lecture. Si vous avez des doutes, envoyez l'adresse du dépôt au moins 24h avant la date limite pour permettre des tests. Nous nous engageons à évaluer le code uniquement après la date limite.

Le travail peut être effectué en équipes de trois personnes maximum. Un travail individuel est aussi possible. Si vous travaillez en équipe, votre `README` doit en indiquer les membres. Votre projet ne doit pas être la copie (partielle ou intégrale) du code d'une autre équipe.

Fichiers à rendre : Déposez une archive (`tarfile` ou `zip`, pas de format `rar` ou autres formats propriétaires) sur Moodle, *avant la date limite*. L'archive devra contenir :

— Les fichiers source, qui doivent compiler (par un simple `make` à la console).
— un fichier `README` contenant une description courte du travail effectué, éventuellement des difficultés rencontrées, et aussi des tests que vous avez faits, éventuellement avec vos observations (par exemple sur la taille du code, son optimalité etc.).

Un projet qui ne compile pas et qui ne peut pas être exécuté sur des exemples (même modestes) sera d'emblée exclu de toute évaluation.

1.3 Fichiers fournis

L'archive fournie contient les fichiers suivants :

— `lexer.ml` et `parser.ml` : Le lexer/parser pour le langage source. Voir aussi § 2.
— `code2graph.ml` (actuellement presque vide) : conversion du code assembleur vers le graphe de flot de contrôle, voir § 6.
— `comp.ml` : le fichier principal, contenant la fonction `main` qui permet d'exécuter le code de la console, voir annexe A.2.
— `graph.ml` : types de graphes étiquetés et de fonctions pour l'affichage de graphes, voir § 6.
— `lang.ml`, qui contient les types du langage source. Vous avez le droit de modifier ces types, mais il n'est pas nécessaire de le faire, et si vous le faites, il faut documenter et motiver ces modifications.
— `instrs.ml` : le type de donnés des instructions du langage cible (assembleur).
— `interf.ml` : une interface pour interagir avec les fonctions en Caml, voir annexe A.2.
— `gen.ml` (actuellement presque vide) : le fichier qui devra contenir vos fonctions de génération de code, voir § 5.
— `typing.ml` (actuellement presque vide), contenant des fonctions de typage, voir § 3.
— `use.ml` : Permet de charger les fichiers compilés, dans l'interpréteur de Caml, voir annexe A.2.
— un `Makefile` pour compiler le projet. Il faut décommenter la partie marquée `TODO` quand vous faites l'intégration du lexer / parser. Du reste, ne modifiez pas ce fichier sauf si vous savez exactement ce que vous faites.

— dans le sous-répertoire `Tests` : `even.c` et `fac.c`, des exemples de programmes que vous pourrez utiliser comme cas de test. Ces programmes sont aussi acceptés par des compilateurs C. Pour les compiler en C,
  — dé-commentez la fonction `main()`
  — compilez le code, par exemple avec `gcc -o even even.c`
  — exécutez le programme, par exemple avec `./even`

  Notre parser n'accepte qu'une seule fonction dans un fichier, avant de l'utiliser, pensez à commenter `main()`.

Souvent, les fichiers contiennent des définitions incomplètes, et qui existent uniquement pour permettre une compilation des sources Ocaml sans erreur.

2 Analyse syntaxique

L'analyse syntaxique est un travail à fournir par les participants du cours "Langages et Automates" et compte à titre de ce cours. [1]

Lors de la phase initiale du projet, un lexer / parser est mis à votre disposition (fichiers `lexer.ml` et `parser.ml`). Il s'agit des fichiers générés par les outils `ocamllex` et `ocamlyacc`. Vous pourrez les remplacer ultérieurement par votre propre lexer / parser.

L'analyse syntaxique doit reconnaître des programmes écrits en C, plus précisément : le parser doit être capable de reconnaître une définition de fonction écrite en C. La syntaxe de C est suffisamment proche de Java pour vous permettre d'écrire des programmes vous-mêmes, de les compiler et exécuter comme décrit en § 1.3.

Le langage C a été normalisé par l'ISO et est décrit dans un standard très volumineux [2]. L'annexe A du standard donne la syntaxe complète que nous n'allons évidemment pas reproduire en intégralité. Les principes suivants doivent guider votre définition du langage en Ocamllex / Ocamlyacc :

— votre définition doit être *correcte* au sens que tout programme accepté par votre analyse syntaxique doit aussi être accepté par un compilateur qui implante le standard (comme `gcc`).
— votre définition n'est pas forcément *complète* et peut rejeter des programmes C valides. Par contre, l'analyse doit admettre une syntaxe concrète pour tous les constructeurs des expressions et commandes définies dans `lang.ml`.

Travail à réaliser : Écrivez un lexer (Fichier `lexer.mll`) et un parser (fichier `parser.mly`) pour le langage C.

Une introduction approfondie à Ocamllex, Ocamlyacc et leur interaction sera fournie dans les TP de Théorie des langages dans les semaines à venir, et il ne sera pas judicieux de se lancer dans cette tâche maintenant. Pour ce projet, on vous demandera de finir le travail commencé en TP, mais pas de l'accomplir en toute indépendance.

3 Typage

Le travail à réaliser ici correspond, *grosso modo*, à la vérification de types déjà effectuée en TP, avec deux différences notables : le langage étudié est impératif, et le typage modifie les termes du langage au lieu de calculer uniquement un type.

Avant d'entrer dans les détails, voici quelques remarques sur le langage, voir aussi le fichier `lang.ml`. Nous recommandons aussi de parser les fichiers C fournis comme exemples et de comparer la syntaxe abstraite et la syntaxe concrète. Il y a deux types de termes :

1. Les étudiants de la DLMI sont donc exemptés de cette tâche.
2. https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf

— `expr` (expressions) qui représentent des valeurs, par exemple (en syntaxe concrète) : `2 + x`. Les constructeurs sont : des constantes, variables, opérations binaires et une expression conditionnelle.
— `stmt` (commandes, *statements*) [3], qui induisent un changement d'état, à savoir `Skip` (qui ne fait rien), une affectation (`Assign`), une séquence `Seq` de deux commandes, une commande conditionnelle `Cond`, une boucle `While` et un `Return` pour terminer une fonction et renvoyer un résultat.

Nous attirons l'attention sur la différence entre l'expression et la commande conditionnelle : La première, `IfThenElse`, est composée d'une expression (condition) et deux expressions pour les cas si / sinon, qui doivent avoir le même type. Un exemple en C est `(2 < 3 ? 4 : 5)`. Si la condition `2 < 3` est vraie, on renvoie `4`, sinon `5`. La commande `Cond` comporte une condition et deux commandes.

Il n'y a pas un constructeur d'expression par opérateur binaire, mais les opérateurs ont été regroupés (arithmétiques et de comparaison) et sont un argument du constructeur `BinOp`. Par souci de simplicité, il n'y a pas d'opérateurs unaires, qui se codent facilement avec des opérateurs binaires.

Les types de nos programmes C sont limités : `int` et `bool`, [4] représentés en Caml par :

```
type tp = BoolT | IntT
```

Pour faire la vérification de types, nous utilisons un *environnement*, représenté en Caml par

```
type environment =
    {localvar: (vname * tp) list;
     returntp: tp}
```

Il est plus complexe que les environnements vus précédemment : outre la liste d'association entre variables et types, il doit aussi tenir compte du type de retour attendu par la fonction.

Travail à réaliser : Dans le fichier `typing.ml`, implantez des fonctions de vérification de type pour expressions et commandes `tp_expr` et `tp_stmt`. Vous avez évidemment le droit d'utiliser des fonctions auxiliaires. Ainsi, une fonction qui récupère le type (donc le premier argument des constructeurs) s'avère utile.

Pour vérifier une définition de fonction (`fundefn`), il faut assurer que
— les paramètres et variables locales des fonctions ont des noms disjoints ;
— tout `return` de la fonction renvoie un résultat du type attendu par la fonction ;
— la fonction renvoie un résultat en toute circonstance ; autrement dit, qu'un programme ne termine pas avec un résultat indéfini (voir cours pour plus de détails). Par exemple, une fonction dont le corps est :

```
if (x == 0) {
  return (42); }
else {
  x = x + 1; }
```

ne renvoie pas de résultat défini au cas où $x \neq 0$.

Travail à réaliser : Dans le fichier `typing.ml`, écrivez la fonction `tp_fundefn` qui prend un argument de type `fundefn`, effectue les vérifications mentionnées, et ensuite construit un environnement initial et vérifie la commande (avec `tp_stmt`) qui constitue le corps de la fonction.

Regardez § B.2 pour un exemple de code typé.

3. Pour éviter toute confusion terminologique, nous parlons de *commandes* pour le langage source et d'*instructions* pour le langage cible

4. En C, on peut bien utiliser un type `bool`, à l'aide de la librairie `stdbool` ; voir le fichier `even.c` joint comme test.

4 Exécution assembleur

Le but de cette section est de décrire le langage cible de la traduction, un mini-assembleur, et de préciser sa sémantique. Ceci vous permet d'écrire une machine virtuelle qui exécute le code assembleur.

Le mini-assembleur considéré ici n'a que quatre instructions :

— `Store`, pour calculer et stocker une valeur dans une variable ;
— `Goto`, pour un branchement inconditionnel vers une adresse ;
— `Branch`, pour un branchement conditionnel selon la valeur d'une variable ;
— `Exit`, pour terminer l'exécution et renvoyer un résultat.

Le type `('l, 'v) instr` des instructions de l'assembleur est paramétré par un type d'étiquettes (*label*) `'l` et un type de variables `'v`. Nous verrons différentes instanciations plus tard.

Pour décrire plus précisément la *sémantique* des instructions, c'est à dire leur comportement souhaité, il faut spécifier quelles sont les *configurations* du programme et comment elles sont impactées par les instructions. Ici, une configuration est un triplet :

— une liste d'instructions (le code du programme) ;
— une liste de variables (les variables du programme) ;
— l'étiquette de l'instruction courante, aussi appelée *program counter* (*pc*).

Pour des raisons de performance, nous utiliserons des tableaux (d'instructions / variables) au lieu de listes.

Un exemple d'une configuration est donné dans la fig. 2.

Code :

0	1	...	5	...	9
`Store(x, y + z)`	`Goto(5)`	...	`Branch(y, 0, 9)`	...	`Exit(x)`

Vars :

`x`	`y`	`z`
30	40	2

pc : 0

FIGURE 2 – Exemple de configuration d'un programme assembleur

Voici donc une sémantique plus précise. Notons que la liste d'instructions n'est pas modifiée par l'exécution d'instructions.

— `Store(`v`,`se`)` évalue l'expression simple se et remplace la variable v dans la liste des variables par la valeur obtenue. Le *pc* est avancé à l'instruction suivante.
Exemple (voir fig. 2) : Le $pc = 0$ pointe sur l'instruction `Store(x, y + z)`. L'évaluation de l'expression `y + z` donne `42`. Le tableau des variables est donc modifié : | `x` | `y` | `z` | / | 42 | 40 | 2 | et le *pc* avance à la position 1.
— `Goto(`l`)` : Le *pc* avance à la position l, tout le reste est inchangé.
Exemple (continué) : Le *pc* avance à la position 5.
— `Branch(`v`, `l_1`, `l_2`)` : Si la variable v n'est pas zéro, le *pc* devient l_1, sinon l_2 ; tout le reste est inchangé.
Exemple (continué) : La variable `y` a une valeur non-zéro, et le *pc* devient 9.
— `Exit(`v`)` : Le programme renvoie la valeur associée à la variable v et s'arrête ; il n'y a pas de configuration suivante.
Exemple (continué) : La variable `x` a la valeur 42, et le programme s'arrête avec cette valeur.

Travail à réaliser : Dans le fichier `instrs.ml`, implantez la fonction `exec_instr` qui prend comme arguments un tableau d'instructions, un tableau de variables et le compteur de programmes *pc*. Cette fonction exécute donc une seule instruction, à savoir l'instruction désignée par le *pc* dans le tableau d'instructions. Sur cette base, définissez la fonction `run_code` qui prend comme argument une configuration et l'exécute jusqu'à la fin, pourvu que le code termine. Au cas contraire, `run_code` peut boucler à l'infini.

Dans la suite, nous fournissons quelques compléments d'information pour réaliser ces fonctions.

Expressions simples Pour la fonction `exec_instr`, vous avez besoin d'une fonction d'évaluation d'*expressions simples*. Le type `simple_expr` est prédéfini dans `instr.ml`. Les expressions simples sont : des constantes ; des variables ; ou l'application d'un opérateur binaire à deux variables.

Adressage d'instructions Nous avons prédéfini deux types d'adressage :

— L'adressage relatif `rel_jump` définit la cible d'un branchement par rapport à la position actuelle d'une instruction. Par exemple, l'instruction `Goto (RelJump -3)` qui se trouve à la position 12 du code se réfère à l'instruction à la position 9. L'adressage relatif est plus approprié lors de la génération de code, voir § 5.

— L'adressage absolu `abs_jump` définit la cible d'un branchement en termes absolus. Ainsi, `Goto (AbsJump 9)` se réfère à l'instruction à la position 9 indépendamment de la position du `Goto`. Dans un code bien formé, uniquement des adresses entre 0 et longueur du code -1 sont possibles.

Vous avez le choix du mode d'adressage qui vous convient le mieux. Dans le premier cas, la fonction `run_code` a le type `(rel_jump, instr_index) instr array * int array * int -> int` ; dans le deuxième cas, le type est `(abs_jump, instr_index) instr array * int array * int -> int`.

Adressage de variables Pareil, nous avons prédéfini deux types d'adressage pour les variables :

— `instr_var` permet d'adresser des variables par nom ou par nombre :

```
type instr_var =
  IVarNamed of vname
| IVarNum of int
```

Les variables nommés (`IVarNamed`) proviennent directement du programme source ; les variables numérotées (`IVarNum`) sont introduites lors de la compilation pour stocker des valeurs intermédiaires.

— `instr_index` adresse les variables par index (un entier positif).

```
type instr_index = IIndex of int
```

C'est le mode d'adressage choisi quand les valeurs des variables sont stockées dans un tableau et on y accède par position.

Notez que l'exemple de la fig. 2 utilise une notation "enjolivée". Dans un tableau du type `(abs_jump, instr_index) instr array`, on trouverait :

— au lieu de l'instruction `Store(x, y + z)`, l'instruction `Store (IIndex 0, BinOpSE (BArith BAadd, IIndex 1, IIndex 2))`, et le tableau

x	y	z
30	40	2

s'écrirait en fait

0	1	2
30	40	2

.

— au lieu de l'instruction `Goto(5)`, l'instruction `Goto(AbsJump 5)`.

Les différents modes d'adressage seront approfondis en § 5.

5 Génération d'assembleur

La génération de code se décompose en plusieurs étapes. Nous proposons de procéder comme suit – vous avez toutefois la liberté de choisir une autre démarche :

1. Génération de code avec des adresses relatives et variables nommées. Algorithmiquement, cette étape est la plus complexe ; elle sert à convertir l'arbre syntaxique issu de l'analyse syntaxique et du typage (voir § B.2 pour un exemple) en une liste d'instructions de l'assembleur (voir § B.3).
2. Transformation en code avec des adresses absolues : une traversée facile des instructions de la phase précédente.
3. Transformation en code avec des indices au lieu de variables nommées : également une transformation facile du code de la phase précédente.

Le code obtenu après enchaînement de ces trois phases peut être exécuté comme décrit en § 4.

5.1 Code avec adresses relatives, variables nommées

La génération de code repose essentiellement sur deux fonctions que nous appelons `gen_expr` et `gen_stmt`. La première traduit une expression, la deuxième un *statement* en une liste d'instructions.

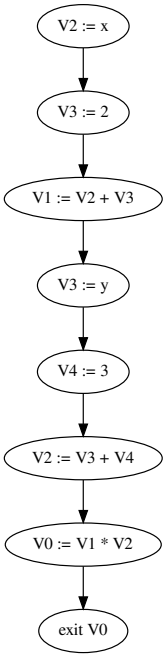

FIGURE 3 – `return((x+2)*(y+3))`

En première approximation, la fonction `gen_instr` a donc le type `tp expr -> (rel_jump, instr_var) instr list`. Il faut maintenant clarifier quel est le rapport entre le code source et le code généré. Nous rappelons qu'une expression représente une valeur, possiblement en fonction des valeurs des variables qu'elle contient. Ainsi, $(x + 2) * (y + 3)$ vaut 24 pour $x = 2$ et $y = 3$. Par convention, nous ferons en sorte que le résultat soit stocké dans une variable. Pour identifier cette variable, nous introduisons un paramètre supplémentaire, la *variable résultat*. La fonction `gen_instr` a maintenant le type `instr_var -> tp expr -> (rel_jump, instr_var) instr list`. Il s'avère qu'il faut introduire des variables auxiliaires lors de la traduction, pour stocker des valeurs intermédiaires. Les variables auxiliaires seront numérotés (`IVarNum`, voir § 4), contrairement aux variables du code source qui sont nommées (`IVarNamed`). Pour les variables auxiliaires, il faut connaître le prochain numéro de variable qui n'est pas encore utilisé ; ce sera le premier paramètre de `gen_instr` qui a donc finalement le type `int -> instr_var -> tp expr -> (rel_jump, instr_var) instr list`.

Nous illustrons la démarche avec la traduction de l'expression $(x + 2) * (y + 3)$, dont le résultat doit être stocké dans la variable auxiliaire 0 (nous l'appelons $V0$). Pour le faire, nous traduirons récursivement $x+2$ dont le résultat sera stocké dans une nouvelle variable $V1$, et $y+3$ dont le résultat sera stocké dans une nouvelle variable $V2$. A la fin, on calculera $V0 := V1 * V2$.[5] Récursivement, pour compiler $x + 2$ et mettre le résultat dans $V1$, on va créer $V2$ qui recevra la valeur de x, $V3$ qui recevra la valeur de 2, et on calcule $V1 = V2 + V3$. On procédera pareil avec l'expression $y + 3$. Le code pour cette expression (plus précisément pour `return((x + 2) * (y + 3))`) se trouve dans la fig. 3.

5. On utilisera la notation mnémonique $v := e$ pour `Store(`v`, `e`)`.

On note que notre génération de code n'est pas optimisée (l'affectation d'une variable à une variable auxiliaire ne serait pas nécessaire), mais la compilation est plus uniforme de cette manière. Plus important, on pourra réutiliser des variables, mais uniquement quand leur valeur initiale est obsolète. Ainsi, la variable $V3$ reçoit d'abord la constante 2. Cette valeur est utilisée dans l'affectation $V1 := V2 + V3$, ce qui libère $V3$ pour une nouvelle affectation $V3 := y$.

Travail à réaliser : Écrivez la fonction `gen_expr` (fichier `gen.ml`).

Pour la traduction de l'expression `IfThenElse`, regardez aussi la traduction de la commande conditionnelle `Cond` plus bas.

A plusieurs égards, la fonction `gen_stmt` pour la génération de *statements* du code source est plus facile à réaliser. Cette fonction a le type `tp stmt -> (rel_jump, instr_var) instr list`. Le seul point délicat est le branchement induit par les commandes conditionnelles et les boucles. Pour illustrer le problème, considérons la boucle `while`. Après avoir exécuté le corps de la boucle, il faut retourner (inconditionnellement, avec un `Goto`) au début de la condition de la boucle pour la réévaluer et possiblement effectuer un autre tour de boucle. Il est le plus facile d'utiliser un adressage relatif pour ce branchement, et l'adresse cible se calcule en fonction de la longueur du code pour la condition et le corps de la boucle.

Il faut encore traiter un phénomène fâcheux : Dans le cas d'une commande conditionnelle, on voudra faire un branchement `Goto` vers l'instruction qui suit le `if .. then .. else`. Par exemple, dans

```
if (x < 3) {
  y = 1;
}
else {
  y = 2;
}
x = x + 1;
```

après les affectations `y = 1` et `y = 2`, on branche vers l'instruction `x = x + 1`. Or, si les deux branches se terminent avec un `return`, la conditionnelle n'aura pas d'instruction suivante, et on risque de brancher vers une instruction inexistante.

Travail à réaliser : Écrivez la fonction `gen_stmt` (fichier `gen.ml`). Trouvez un moyen pour éviter le problème de branchement vers des adresses inexistantes.

5.2 Code avec adresses absolues, variables nommées

Travail à réaliser : Traduisez les adresses relatives vers des adresses absolues (dans `instrs.ml`).

On passe donc d'un code de type `(rel_jump, instr_var) instr list` à un code de type `(abs_jump, instr_var) instr list`, voir l'exemple § B.3.2.

Ainsi, l'instruction à la position 4, `Goto (RelJump 5)`, devient `Goto (AbsJump 9)`. Cette traduction est très facile à réaliser : vous avez besoin d'une fonction qui traduit une seule instruction (étant donné la position actuelle de l'instruction dans la liste des instructions), et une autre fonction qui applique cette fonction à une liste d'instructions – une ligne si vous utilisez la fonction prédéfinie `List.mapi` [6].

5.3 Code avec adresses absolues, variables indexées

Cette traduction est légèrement plus complexe (mais toujours de l'ordre de 20 lignes de Caml) : on traduit les variables nommées en variables numérotées par des indices, on passe donc d'un code de type

6. https://ocaml.org/api/List.html#VALmapi

`(abs_jump, instr_var) instr list` à un code de type `(abs_jump, instr_index) instr list`, voir l'exemple § B.3.3.

Nous illustrons la démarche avec un exemple : supposons que votre code a trois variables nommées `IVarNamed "x", IVarNamed "y", IVarNamed "z"` et cinq variables numérotées `IVarNum 0`, ..., `IVarNum 4`. On va stocker les valeurs de ces variables dans un tableau de taille 8. Les variables x, y, z se trouveront au début de ce tableau et deviendront les variables indexées `IIndex 0`, ..., `IIndex 2`, les variables numérotées `IVarNum 0`, ..., `IVarNum 4` deviendront les variables indexées `IIndex 3`, ..., `IIndex 7`.

Travail à réaliser : Traduisez les variables nommées et numérotées vers des variables indexées (dans `instrs.ml`).

5.4 Génération et exécution du code

Il reste à relier les étapes précédentes, voir aussi § B.4, pour obtenir la fonction `run_test` qui prend comme argument le nom d'un fichier (contenant une fonction écrite en C) et une liste de valeurs (les arguments de la fonction) et renvoie le résultat de l'application de la fonction C aux arguments.

Pour cela, il est nécessaire :

1. de combiner les étapes précédentes : analyse syntaxique, vérification de types, traduction de l'arbre syntaxique typé vers une liste d'instructions assembleur avec adresses absolues et variables indexées ;
2. de créer une configuration initiale avec ce code et un tableau de variables initialisé avec les arguments de la fonction ;
3. d'exécuter ce code avec la machine virtuelle, voir § 4.

Nous rappelons qu'une configuration de la machine virtuelle est composée du code à exécuter, du tableau des variables et du compteur de programme. La création de la configuration initiale n'est pas difficile, mais demande quelque attention : il faut correctement dimensionner le tableau des variables. Sa taille est déterminée par l'index maximal utilisé dans le code (surtout : n'utilisez pas une constante arbitraire). Écrivez une fonction qui calcule cet index maximal.

Travail à réaliser : Écrivez la fonction `run_test` (dans `interf.ml`).

6 Graphe de flot de contrôle

Nous générons un graphe de flot de contrôle qui peut être visualisé comme dans les figures 3 et 4. Pour l'affichage des graphes, nous utilisons la librairie Graphviz [7]. Graphviz prend comme entrée un langage de description de graphes (Dot) et le convertit vers différents formats (jpeg, pdf). Certains attributs des graphes sont configurables (forme et couleur des noeuds et arcs), mais la disposition des noeuds sur le plan est calculé par un algorithme de layout.

Pour faciliter encore plus l'interaction avec Graphviz / Dot, nous fournissons dans le fichier `graph.ml` une interface avec Caml. Vous y trouvez notamment les types `('n, 'l) node` des noeuds et `('n, 'l) edge` des arcs. Le type `'n` correspond à un type d'identifiant de noeuds et `'l` a un type de *label* (types d'étiquettes des noeuds et d'arcs, qui peuvent être différents). Par conséquent, le type `('n, 'nl, 'el) graph` est paramétré par un type d'identifiants de noeuds, d'étiquettes de noeuds et d'étiquettes d'arcs.

Travail à réaliser : Convertissez le code obtenu en § 5.2 en graphe. Pour cela, il suffit de compléter la définition de `graph_of_code` (fichier `code2graph.ml`).

7. http://www.graphviz.org/

Vous êtes libres à développer votre propre affichage de graphes. Dans ce qui suit, nous faisons quelques suggestions :

1. Le type de noeuds devrait être `int` : Il y a un noeud pour chaque instruction, et il semble pertinent d'identifier chaque instruction par sa position dans le code (le tableau des instructions assembleur).
2. L'étiquette d'un noeud pourrait être l'instruction correspondante.
3. Vous pouvez choisir l'étiquette d'un arc. Dans un branchement conditionnel (`Branch`), on devrait marquer l'arc qui correspond à un résultat positif et l'arc qui correspond à un résultat négatif du test. Pour ce faire, vous pouvez utiliser ou modifier le type `edge_label`.

Avec ceci, le graphe a le type `(int, (abs_jump, instr_var) instr, edge_label) graph`.

Pour donner un exemple : Soit `Branch (IVarNum 0, AbsJump 13, AbsJump 17)` une instruction à la position 12 du code (voir exemple § B.3.2).

Le noeud créé pour cette instruction est : `Node(12, Branch (IVarNum 0, AbsJump 13, AbsJump 17))` Deux arcs sont créés : `Edge(12, 13, BranchIf true)` et `Edge(12, 17, BranchIf false)`.

Construire le graphe de flot de contrôle ne dit rien sur la manière dont il est affiché. Le but du type `('n, 'nl, 'el) display` est de packager un graphe avec des fonctions qui produisent le code Dot pour
— les noeuds ; ici : `string_of_int` ;
— les étiquettes des noeuds ; ici : `node_label_display` ;
— les étiquettes des arcs ; ici : `edge_label_display`.

Vous trouvez une proposition pour ces fonctions dans le fichier `code2graph.ml`. La fonction `node_label_display` dépend d'une fonction `string_of_instr_IVar` qui est censée afficher les instructions.

> **Travail à réaliser :** Définissez la fonction `string_of_instr_IVar` et adaptez les fonctions d'affichage des noeuds / arcs selon votre goût.

Vous pouvez modifier quelques éléments de style de l'affichage (couleur des noeuds et arcs etc.). Pour cela, il faut se plonger dans la documentation de Dot. Pourtant, ne passez pas trop de temps à enjoliver l'affichage, ce n'est pas très valorisant.

Encore quelques mots sur la fonction `generate_dot_file` (fichier `graph.ml`) : elle prend (le préfixe d')un nom de fichier (disons `"test"`) et une instance d'un `('n, 'nl, 'el) display` et génère deux fichiers :
— un fichier Dot, par exemple `test.dot`
— un fichier PDF, par exemple `test.pdf`

Le fichier Dot est un simple fichier texte que vous pouvez aussi modifier avec un éditeur texte. Pour afficher le graphe qu'il contient, vous pouvez coller le contenu du fichier dans un interpréteur interactif, par exemple sketchviz.com ou graphviz.it [8].

Vous pouvez aussi générer un format graphique, par exemple PDF, à partir de la console, dans ce cas avec la commande `dot test.dot -Tpdf -o test.pdf`. Pour cela, il faut que l'application `dot` soit installée sur votre ordinateur. C'est précisément ce que fait la fonction Caml `generate_dot_file`. Si l'application Dot a un autre nom, vous devez adapter la ligne qui effectue l'appel : `Sys.command ...`

A Quelques astuces d'utilisation de Caml

A.1 Tableaux

Un trait décidément impératif de Caml est le type des tableaux (`array`). Il est documenté en détail dans le manuel de Caml [9] ; nous y résumons l'essentiel.

Contrairement à une liste, un tableau est alloué initialement au début de sa vie ; on peut accéder à un élément par indexe (donc en temps constant) et pas en traversant la liste (donc en temps linéaire).

8. Actuellement inaccessible ? Était longtemps le meilleur interpréteur.
9. https://ocaml.org/api/Array.html

Pareil, une position donnée d'un tableau peut être modifiée destructivement et pas en reconstruisant une nouvelle liste contenant un nouvel élément. La manipulation de tableaux est donc plus efficace que la manipulation de listes, mais comporte le risque d'effets de bord imprévisibles en cas de partage de tableaux.

Les faits essentiels sur la manipulation de tableaux sont :

— *écriture :* par exemple, `[|1; 2; 3|]` est le tableau qui correspond à la liste `[1; 2; 3]`.
— *création* (1ère méthode) : `Array.make n x` crée un tableau de taille `n` initialisé avec la valeur `x`.
— *création* (2ème méthode) : `Array.of_list xs` convertit la liste `xs` en tableau.
— *accès* (lecture) : Pour un tableau `myarr`, on accède à l'élément `i` avec `myarr.(i)`.
— *accès* (modification) : Pour un tableau `myarr`, on modifie l'élément à la position `i` par exemple avec : `myarr.(i) <- 42`.

A.2 Compilation de fichiers

En Caml, vous avez le choix entre
— des fichiers source `*ml`
— des fichiers compilés / objet `*cmo`

Pour lancer votre code à partir de la console, il est impératif de le compiler. Pour le développement de votre code, il est préférable d'interagir avec l'interpréteur de Caml et de travailler avec les fichiers `*ml`. Vous pouvez facilement compiler tous les fichiers avec un `make` dans la console. Dans l'évaluateur Caml,
— pour charger un fichier `f` compilé, lancez `#load "f.cmo";;` [10]
— pour charger un fichier `f` source, lancez `#use "f.ml";;`

Problèmes typiques :
— Modules : après compilation et chargement avec `#load "nom de fichier.cmo";;`, les fonctions sont définis dans un module. Vous pouvez ou bien les appeler avec leur nom complet, par exemple `Typing.tp_expr`, ou bien ouvrir le fichier pour omettre le préfixe : `open Typing;;` et ensuite appeler `tp_expr`.
— En cas de blocage complet : Sortez de l'interpréteur interactif de Caml (avec `Ctrl-d`). Recompilez tous les fichiers avec `make`. Évaluez de nouveau le fichier `use.ml`.

Le fichier `use.ml` regroupe tous les `#load` à effectuer. Pour démarrer avec un état bien défini, vous pourrez lancer Caml et évaluer `#use "use.ml";;`

Utilisation du parser Pour développer votre code progressivement et tester vos fonctions de typage et de génération de code, il est utile d'utiliser le parser en mode interactif, pour récupérer l'arbre syntaxique correspondant au code que vous avez écrit dans un fichier, par exemple `even.c`.

Pour utiliser le parser, procédez comme suit :

1. Assurez vous que les fichiers Caml sont correctement compilés, avec un `make`.
2. Lancez une session interactive de Caml et évaluez le fichier `use.ml`, avec `#use "use.ml";;`
3. Pour avoir accès au parser : `open Interf;;`
4. Ensuite, lancez `parse "Tests/even.c" ;;` (ou choisissez le nom de fichier approprié)
5. Pour éviter le préfixe `Lang` devant les constructeurs : `open Lang;;`

Aussi la génération de code, du graphe de flot de contrôle etc. peuvent se faire à partir de la session interactive.

10. Le dièse # fait partie de la commande

B Exemple complet

B.1 Code source

Dans cette section, nous présentons un exemple complet : la fonction `even` qui calcule si un nombre `n` est pair. Le code n'a pas vocation à être le plus élégant ou concis possible, mais à illustrer plusieurs éléments du langage. Le code se trouve aussi dans le fichier `Tests/even.c` dans l'archive du projet.

```
#include <stdbool.h>

bool even (int n) {

  if (n < 0)
    /* no unary expressions available ... */
    n = 0 - n;

  while (n > 1) {
    n = n - 2;
  }

  return (n == 0) ;
}
```

Nous attirons votre attention sur les aspects suivants :

- Le type `bool` n'est pas prédéfini en C, mais on peut le simuler avec la librairie `stdbool.h` incluse dans la première ligne.
- Notre parser n'admet pas d'opérateurs unaires (par exemple `- n`). Vous êtes libres à les implanter dans votre version du parser.

B.2 Typage

Après analyse syntaxique et typage (voir § 3), vous obtenez le code Caml suivant :

```
# Typing.tp_fundefn (Interf.parse "Tests/even.c") ;;
- : Lang.tp Lang.fundefn =
Fundefn (Fundecl (BoolT, "even", [Vardecl (IntT, "n")]), [],
 Seq
  (Seq
    (Cond
      (BinOp (BoolT, BCompar BClt, VarE (IntT, "n"), Const (IntT, IntV 0)),
      Assign (IntT, "n",
       BinOp (IntT, BArith BAsub, Const (IntT, IntV 0), VarE (IntT, "n"))),
      Skip),
    While
     (BinOp (BoolT, BCompar BCgt, VarE (IntT, "n"), Const (IntT, IntV 1)),
     Assign (IntT, "n",
      BinOp (IntT, BArith BAsub, VarE (IntT, "n"), Const (IntT, IntV 2))))),
  Return
   (BinOp (BoolT, BCompar BCeq, VarE (IntT, "n"), Const (IntT, IntV 0)))))
```

Entre autres, l'expression qui correspond à `n - 2` dans le code source a été annotée avec le type `IntT` : `BinOp (IntT, BArith BAsub, VarE (IntT, "n"), Const (IntT, IntV 2))`, et l'expression qui correspond à `n == 0` avec le type `BoolT` : `BinOp (BoolT, BCompar BCeq, VarE (IntT, "n"), Const (IntT, IntV 0))`

B.3 Génération d'assembleur

B.3.1 Adresses relatives, variables nommées

Comme indiqué dans § 5, il y a plusieurs manières de procéder. La démarche suivie ici est de créer d'abord un code avec des *adresses relatives* et la préservation de *variables nommées* (ici : `IVarNamed "n"`) mais aussi des variables numérotées pour des variables temporaires introduites lors de la compilation (ici : `IVarNum 0` etc.).

```
# Gen.gen_fundefn_rel_addr (Typing.tp_fundefn (Interf.parse "Tests/even.c")) ;;
- : (Instrs.rel_jump, Instrs.instr_var) Instrs.instr list =
[Store (IVarNum 1, VarSE (IVarNamed "n")); Store (IVarNum 2, ConstSE 0);
 Store (IVarNum 0, BinOpSE (BCompar BClt, IVarNum 1, IVarNum 2));
 Branch (IVarNum 0, RelJump 2, RelJump 1); Goto (RelJump 5);
 Store (IVarNum 0, ConstSE 0); Store (IVarNum 1, VarSE (IVarNamed "n"));
 Store (IVarNamed "n", BinOpSE (BArith BAsub, IVarNum 0, IVarNum 1));
 Goto (RelJump 1); Store (IVarNum 1, VarSE (IVarNamed "n"));
 Store (IVarNum 2, ConstSE 1);
 Store (IVarNum 0, BinOpSE (BCompar BCgt, IVarNum 1, IVarNum 2));
 Branch (IVarNum 0, RelJump 1, RelJump 5);
 Store (IVarNum 0, VarSE (IVarNamed "n")); Store (IVarNum 1, ConstSE 2);
 Store (IVarNamed "n", BinOpSE (BArith BAsub, IVarNum 0, IVarNum 1));
 Goto (RelJump (-7)); Store (IVarNum 1, VarSE (IVarNamed "n"));
 Store (IVarNum 2, ConstSE 0);
 Store (IVarNum 0, BinOpSE (BCompar BCeq, IVarNum 1, IVarNum 2));
 Exit (IVarNum 0)]
```

B.3.2 Adresses absolues, variables nommées

De ceci, nous passons aux adresses absolues et gardons les variables nommées, voir aussi § 5.2. Notez, par exemple, que l'instruction à la position 4, `Goto (RelJump 5)`, devient `Goto (AbsJump 9)` si on utilise des adresses absolues. Le code ainsi obtenu est aussi la base pour la construction du graphe de flot de contrôle, voir § B.5.

```
# Gen.gen_fundefn_abs_addr (Typing.tp_fundefn (Interf.parse "Tests/even.c")) ;;
- : (Instrs.abs_jump, Instrs.instr_var) Instrs.instr list =
[Store (IVarNum 1, VarSE (IVarNamed "n")); Store (IVarNum 2, ConstSE 0);
 Store (IVarNum 0, BinOpSE (BCompar BClt, IVarNum 1, IVarNum 2));
 Branch (IVarNum 0, AbsJump 5, AbsJump 4); Goto (AbsJump 9);
 Store (IVarNum 0, ConstSE 0); Store (IVarNum 1, VarSE (IVarNamed "n"));
 Store (IVarNamed "n", BinOpSE (BArith BAsub, IVarNum 0, IVarNum 1));
 Goto (AbsJump 9); Store (IVarNum 1, VarSE (IVarNamed "n"));
 Store (IVarNum 2, ConstSE 1);
 Store (IVarNum 0, BinOpSE (BCompar BCgt, IVarNum 1, IVarNum 2));
 Branch (IVarNum 0, AbsJump 13, AbsJump 17);
 Store (IVarNum 0, VarSE (IVarNamed "n")); Store (IVarNum 1, ConstSE 2);
 Store (IVarNamed "n", BinOpSE (BArith BAsub, IVarNum 0, IVarNum 1));
 Goto (AbsJump 9); Store (IVarNum 1, VarSE (IVarNamed "n"));
 Store (IVarNum 2, ConstSE 0);
 Store (IVarNum 0, BinOpSE (BCompar BCeq, IVarNum 1, IVarNum 2));
 Exit (IVarNum 0)]
```

B.3.3 Adresses absolues, variables indexées

La dernière étape consiste à remplacer les variables nommées par des indices :

```
# Gen.gen_fundefn (Typing.tp_fundefn (Interf.parse "Tests/even.c")) ;;
- : (Instrs.abs_jump, Instrs.instr_index) Instrs.instr list =
[Store (IIndex 2, VarSE (IIndex 0)); Store (IIndex 3, ConstSE 0);
 Store (IIndex 1, BinOpSE (BCompar BClt, IIndex 2, IIndex 3));
 Branch (IIndex 1, AbsJump 5, AbsJump 4); Goto (AbsJump 9);
 Store (IIndex 1, ConstSE 0); Store (IIndex 2, VarSE (IIndex 0));
 Store (IIndex 0, BinOpSE (BArith BAsub, IIndex 1, IIndex 2));
 Goto (AbsJump 9); Store (IIndex 2, VarSE (IIndex 0));
 Store (IIndex 3, ConstSE 1);
 Store (IIndex 1, BinOpSE (BCompar BCgt, IIndex 2, IIndex 3));
 Branch (IIndex 1, AbsJump 13, AbsJump 17);
 Store (IIndex 1, VarSE (IIndex 0)); Store (IIndex 2, ConstSE 2);
 Store (IIndex 0, BinOpSE (BArith BAsub, IIndex 1, IIndex 2));
 Goto (AbsJump 9); Store (IIndex 2, VarSE (IIndex 0));
 Store (IIndex 3, ConstSE 0);
 Store (IIndex 1, BinOpSE (BCompar BCeq, IIndex 2, IIndex 3));
 Exit (IIndex 1)]
```

La variable `n` (dans le code de § B.3.2 : `IVarNamed "n"`) obtient l'indice 0 : `IIndex 0`, et par conséquence, les indices des variables numérotées sont décalés d'une position. Par exemple, `IVarNum 2` devient `IIndex 3`.

B.4 Exécution assembleur

Le code obtenu en § B.3.3 est utilisé par l'environnement d'exécution de l'assembleur.

Le code peut être appelé depuis Caml, comme suit :

```
# run_test "Tests/even.c" [3] ;;
- : int = 0
# run_test "Tests/even.c" [4] ;;
- : int = 1
```

La fonction `run_test` génère le code, construit un environnement initial avec les arguments, l'exécute et renvoie le résultat de l'exécution.

Puisque chaque fichier ne contient qu'une seule fonction, il est inutile de préciser la fonction appelée. Les arguments de la fonction sont fournis dans une liste. Puisque la compilation efface tous les types et l'exécution se fait sur les entiers, le résultat est numérique, où 0 représente le booléen "faux" et une valeur $\neq 0$ le "vrai".

Après avoir compilé le code Caml (avec `make`), vous pouvez l'exécuter depuis la console avec :

```
> ./comp run Tests/even.c 3
0
> ./comp run Tests/even.c 4
1
```

B.5 Graphe de flot de contrôle

Le graphe de flot de contrôle construit à partir du code vu en § B.3.2 est affiché en fig. 4. Il est généré à partir du code Dot suivant. La première partie est la spécification des noeuds et de leurs étiquettes, la deuxième partie la spécification des arcs (en partie avec étiquettes).

```
digraph {
0 [label= "V1 := n"]
1 [label= "V2 := 0"]
2 [label= "V0 := V1 < V2"]
3 [label= "V0?"]
4 [label= "goto "]
5 [label= "V0 := 0"]
6 [label= "V1 := n"]
7 [label= "n := V0 - V1"]
8 [label= "goto "]
9 [label= "V1 := n"]
10 [label= "V2 := 1"]
11 [label= "V0 := V1 > V2"]
12 [label= "V0?"]
13 [label= "V0 := n"]
14 [label= "V1 := 2"]
15 [label= "n := V0 - V1"]
16 [label= "goto "]
17 [label= "V1 := n"]
18 [label= "V2 := 0"]
19 [label= "V0 := V1 = V2"]
20 [label= "exit V0"]
0 -> 1
1 -> 2
2 -> 3
3 -> 5 [label= T]
3 -> 4 [label= F]
4 -> 9
5 -> 6
6 -> 7
7 -> 8
8 -> 9
9 -> 10
10 -> 11
11 -> 12
12 -> 13 [label= T]
12 -> 17 [label= F]
13 -> 14
14 -> 15
15 -> 16
16 -> 9
17 -> 18
18 -> 19
19 -> 20
}
```

Depuis Caml, vous pouvez appeler la fonction comme suit :

```
# graph_test "Tests/even.c" "testgraph" ;;
```

Depuis la console, vous pouvez appeler la fonction comme suit :

```
> ./comp graph Tests/even.c testgraph
```

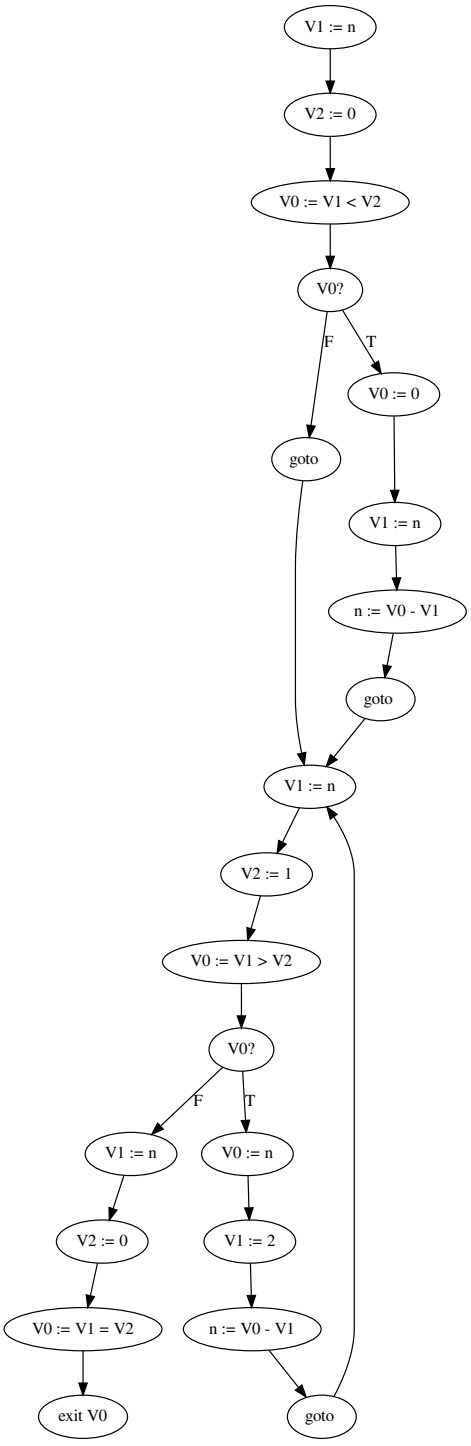

FIGURE 4 – Graphe de flot de contrôle du code de § B.1