TP 3 : Compilation vérifiée

L'objectif de ce TP est d'écrire un compilateur du langage While vers une machine à pile, et de prouver la correction de ce compilateur.

Mise en place du TP

Un squelette de TP est disponible ici :

```
$ git clone git@gitlab-research.centralesupelec.fr:cidre-public/seculog.git
OU
$ git clone https://gitlab-research.centralesupelec.fr/cidre-public/seculog.git

$ git checkout tp3-compil
```

Vous récupérez un répertoire avec les fichiers suivants :

- `_CoqProject` : il s'agit d'un fichier de configuration, à partir duquel on pourra générer un Makefile
- `Tactics.v` : quelques tactiques utiles
- `While.v` : syntaxe et sémantique à grands pas du langage While, adaptée à nos besoins depuis le dernier TP.
- `Stk.v` : syntaxe et sémantique de la machine à pile.
- `While2Stk.v` : compilateur de While vers Stk, et preuve de correction.

Tout d'abord, générez le Makefile avec la commande suivante :

```
$ coq_makefile -f _CoqProject -o Makefile
```

Puis construisez le projet en tapant `make`.

À chaque fois que vous faites une modification dans un fichier, pour que vos changements soient visibles depuis les autres fichiers, pensez à recompiler votre projet en tapant `make` dans le terminal.

1 Le langage While – fichier `While.v`

Ce fichier ressemble à ce que vous avez écrit au dernier TP. On n'a besoin pour ce TP que de la sémantique à grands pas de While. La plus grande différence par rapport au langage While du dernier TP est la nouvelle instruction `Output e`, qui affiche la valeur de l'expression `e`.

Pour modéliser la sémantique de cette instruction, on ajoute un paramètre de type `list Z` à la sémantique : il s'agit de la liste des entiers qu'un programme affiche lors de son exécution. On appellera cette liste la *trace* du programme.

Par exemple le programme `Output (Const 1); Output (Binop Badd (Const 2) (Const 3))` affichera la trace `[1; 5]`.

2 Tactiques utiles – fichier `Tactics.v`

Ce fichier contient notamment la tactique `inv H`, que nous avons déjà rencontrée lors du dernier TP et qui permet de faire une étude de cas sur une hypothèse inductive, en éliminant les cas incohérents.

La tactique `destr` pourra vous être utile : appliquée sur un but qui contient un `match x with ... end`, cette tactique va appeler `destruct x` et donc générer autant de sous-buts que de constructeurs du type de `x`. Cela fonctionne également si le but contient `if x then ... else ...`. Cette tactique permet de ne pas avoir à copier-coller le terme (potentiellement long) sur lequel le `match` ou la condition porte.

Une tactique similaire permet d'appliquer cette stratégie sur le type d'une hypothèse. Ainsi si une hypothèse `H` contient un `match` ou une condition, on peut appeler `destr_in H`, pour le même effet.

On peut utiliser cette tactique avec le combinateur `repeat` qui répète l'application d'une tactique tant qu'elle réussit. Par exemple, `repeat destr_in H; simpl in *; inv H.` permet souvent d'avancer rapidement dans une preuve, et d'éliminer un certain nombre de cas inintéressants.

3 Machine à pile – fichier `Stk.v`

Ce fichier comporte la syntaxe et la sémantique de notre machine à pile, le langage cible de notre compilateur !

Un programme (type `sprog`) est une liste d'instructions (type `instr`). Ces instructions manipulent une pile d'opérandes.

Les instructions sont les suivantes :

- `SConst n` pousse la constante `n` au sommet de la pile.
- `SBinop b` dépile deux entiers de la pile, applique l'opération `b` sur ces deux opérandes, puis pousse le résultat sur la pile.
- `SUnop u` dépile un entier de la pile, applique l'opération `b` sur cette opérande, puis pousse le résultat sur la pile.
- `SLoad v` pousse la valeur de la variable `v` (depuis l'environnement) au sommet de la pile.
- `SStore v` dépile une valeur de la pile, et l'affecte à la variable `v` (dans l'environnement).
- `SOutput` dépile une valeur de la pile, et l'affiche.
- `SLabel l` déclare un label nommé `l` (un entier naturel).
- `SJmp l` dépile une valeur de la pile. Si cette valeur représente le booléen vrai (i.e. un entier non nul, voir `bool_of_Z` dans ce même fichier), alors on saute au label `l`. Sinon on continue l'exécution à la prochaine instruction.
- `SHalt` termine l'exécution du programme.

Les états de programme contiennent un environnement (comme dans le langage WHILE) et une pile (une liste d'entiers). Ces états sont représentés par le type `mstate`, décrits dans le squelette.

Pour l'instruction `SJmp l`, on a besoin de trouver le code corrrespondant à ce label dans le programme : c'est la fonction `find_label` qui effectue cette recherche.

Ensuite, la sémantique des instructions est donnée par la fonction `exec_instr`, commentée avec soin dans le fichier `Stk.v`.

L'évaluation d'un programme entier est donné par le prédicat `eval_sprog`, lui aussi commenté.

On peut maintenant passer au compilateur.

4 Un compilateur prouvé – fichier `While2Stk.v`

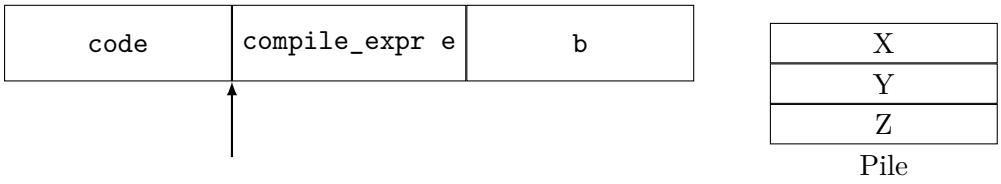
4.1 Compilation des expressions

Commençons par compiler les expressions. Nous vous avons fourni une fonction `compile_binop` qui transforme une opération binaire du langage While en l'opérateur binaire correspondant dans le langage Stk.

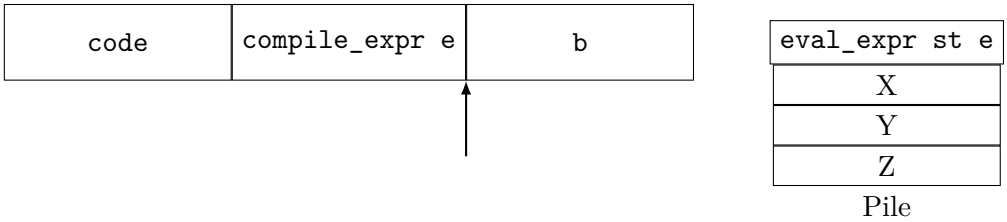
L'objectif de la fonction `compile_expr (e: expr): list instr` est de générer une liste d'instructions `l`, de telle sorte qu'à la fin de l'exécution de ces instructions `l`, on ait la valeur de l'expression `e` au sommet de la pile. Plus précisément, on cherche à prouver le lemme suivant :

```
Lemma compile_expr_correct:
forall e code b st stk tr,
eval_sprog (MState code (compile_expr e ++ b) st stk tr)
(MState (code ++ compile_expr e) b st (eval_expr st e :: stk) tr).
```

Ce lemme dit qu'en partant de cet état :



On arrive à celui-ci :



Avec cet objectif, et la sémantique des instructions de la machine à pile, en tête, c'est parti !

Activité 4.1. Écrivez la fonction `compile_expr`.

Activité 4.2. Prouvez le lemme `compile_expr_correct`.

4.2 Compilation des conditions

On procède de la même manière pour les conditions, sans grande différence.

Activité 4.3. Écrivez la fonction `compile_cond`.

Activité 4.4. Prouvez le lemme `compile_cond_correct`.

4.3 Compilation des instructions

Passons maintenant à la compilation des instructions While en instructions Stk.

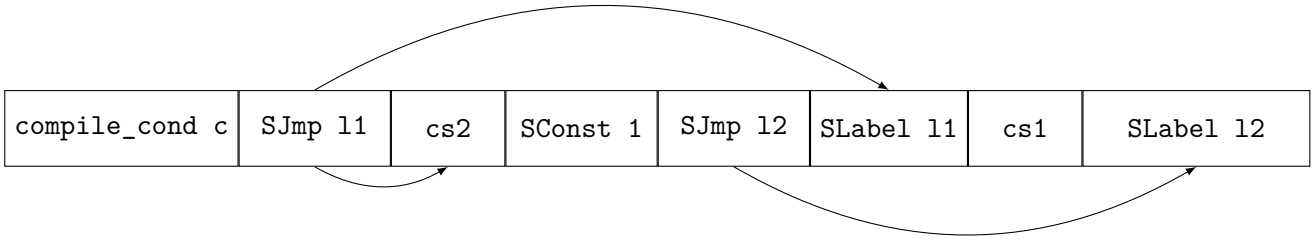
La fonction `compile_stmt (next_lbl: nat) (i: stmt) : (nat * list instr)` prend un paramètre `next_lbl` de type `nat` qui indique le prochain identifiant de label disponible. En effet, il faut générer des labels uniques à chaque fois que l'on en aura besoin. Cette fonction retourne une paire `(nl', l)` où `l` est la liste des instructions Stk correspondant à l'instruction While `i`, et `nl'` est le premier identifiant de label disponible à partir de maintenant. (La compilation de `i` a peut-être utilisé des labels, donc le label `nl` n'est peut-être plus disponible...)

Plusieurs cas vous sont donnés.

Pour l'affectation `Assign x e`, on génère le code suivant :

`compile_expr e`	`SStore x`

Pour la condition `If c s1 s2`, où `cs1` et `cs2` sont les instructions résultant de la compilation de `s1` et `s2`, respectivement.



On suit le chemin du haut si la condition `c` est vraie, le chemin du bas sinon.

Activité 4.5. Complétez la fonction `compile_stmt`.

Dans le fichier, viennent ensuite une série de définitions concernant les labels. Notamment les définitions `labels_below l n` et `labels_above l n` spécifient que tous les labels de `l` sont strictement inférieurs à `n` ou supérieurs ou égaux à `n`, respectivement. Ces prédicats seront utiles pour raisonner sur les labels, et notamment la fonction `find_label`.

La plupart des lemmes qui suivent devraient passer sans problèmes (à moins que vous ayez écrit des bêtises aux questions précédentes.), jusqu'au lemme principal `compile_stmt_correct`. Essayez de comprendre l'énoncé des théorèmes concernant `labels_below`, `labels_above` et `find_label` qui sont prouvés (ou dont on vous donne la preuve en commentaire). Ils vous seront utiles pour la suite.

```
Lemma compile_stmt_correct:
forall i st st' t
       (BS: bigstep st i st' t)
       stk tr nl nl' cs
       (COMPILE: compile_stmt nl i = (nl', cs))
       a b
       (BOUND: labels_below a nl),
```

```
eval_sprog (MState a (cs ++ b) st stk tr)
           (MState (a ++ cs) b st' stk (tr ++ t)).
```

Ce lemme fait le lien entre la sémantique à grands pas de While et la sémantique de Stk. Ce lemme a la même forme que ceux concernant la compilation des expressions et des conditions, vus précédemment.

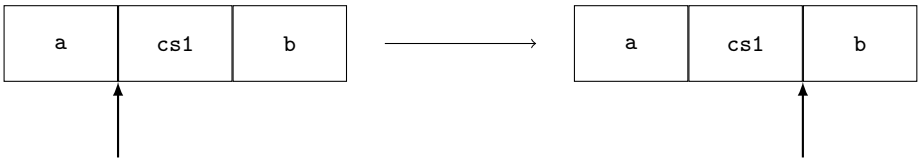
On stipule de plus que les labels contenus dans `a` (le code précédent le point de programme) sont tous inférieux à `nl` (inutile donc d'aller chercher dans `a` un label supérieur ou égal à `nl` !).

4.3.1 Zoom sur un exemple

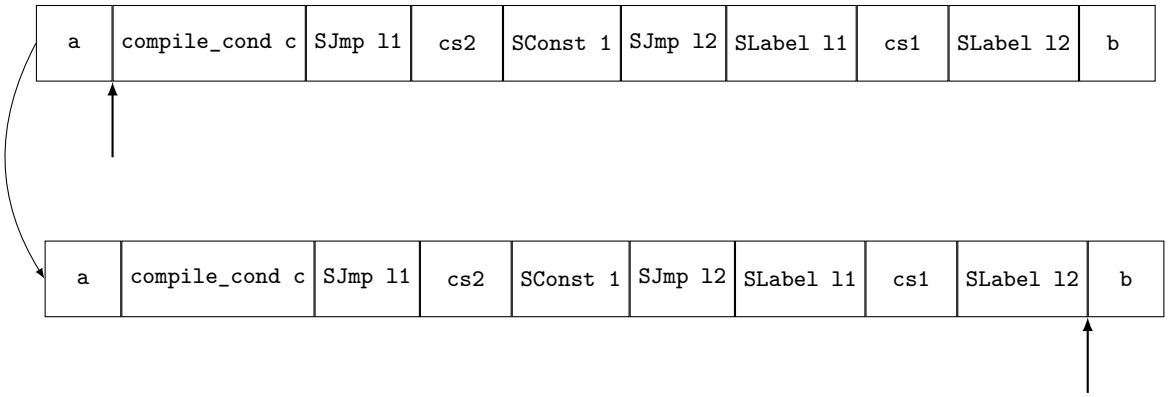
Pour vous mettre sur la piste, le lemme `compile_if_true_correct`, qui correspond au cas de la preuve `compile_stmt_correct` lorsque l'évaluation de la condition est vraie.

Voici, schématiquement, ce que dit ce lemme :

Si on sait cela sur `s1`



Alors :



La preuve de ce lemme est commentée, pour vous expliquer ce que l'on prouve, et vous montrer certaines tactiques et lemmes utiles.

4.3.2 Reste de la preuve

Activité 4.6. Complétez la preuve de `compile_stmt_correct`. Ça va prendre un peu de temps :-)

Pour terminer en douceur, on définit la fonction `compile_prog` qui compile l'instruction `i` et ajoute une instruction `SHalt` pour terminer le programme.

Activité 4.7. Prouvez le lemme `compile_prog_correct` qui établit la correction de la compilation complète d'un programme.