

Conception et implémentation d'un compilateur pour le langage E

1 But de ces séances de travaux pratiques

Au cours de ces séances de travaux pratiques, vous allez réaliser un compilateur pour un petit langage de programmation, le langage E. Le langage E est un dérivé du langage C. Nous nous concentrerons d'abord sur la compilation d'une version basique du langage E que nous enrichirons par la suite sur les créneaux de projet. Votre compilateur sera composé d'un analyseur lexical (*lexer*), d'un analyseur syntaxique (*parser*), puis de plusieurs passes de compilation qui transformeront les programmes E dans des langages de plus en plus bas niveau, jusqu'à la génération de code assembleur RISC-V, qui seront finalement assemblés par des assembleurs existants et qui pourront être exécutés sur vos machines (en utilisant *qemu*).

Comme nous allons le voir, le langage E est relativement petit, pour vous permettre de le réaliser dans le temps de TP qui vous est imparti. Cependant, il permet d'illustrer un grand nombre de concepts fondamentaux de la compilation. Au cas où vous trouveriez que le langage est trop petit, ou bien que les passes de compilation et d'optimisation suggérées ne sont pas suffisantes, nous vous fournirons une liste d'améliorations possibles que vous pourrez implémenter.

La Section 2 vous présente l'architecture du compilateur que vous allez concevoir, notamment les structures de données à utiliser et les différents langages intermédiaires. La Section 3 vous présente l'infrastructure de test qui vous accompagnera pour déboguer votre compilateur. Les sections suivantes décrivent le travail que vous aurez à faire lors des séances de TP. Le découpage en TP est donné à titre indicatif. Si vous n'avez pas fini le travail demandé à la fin d'un TP, vous pourrez utiliser un bout de la séance suivante pour le finir. Essayez de ne pas prendre *trop* de retard.

Vous trouverez le squelette associé à ce TP à l'adresse suivante :

<https://gitlab-research.centralesupelec.fr/cidre-public/compilation/infosec-ecomp>

Si vous voulez travailler sur ce projet dans un dépôt git pour partager votre code avec votre binôme, créez un dépôt vierge sur la plateforme de votre choix (un gitlab de CentraleSupélec, un github, autre chose), puis utilisez la procédure suivante :

```
$ git clone https://gitlab-research.centralesupelec.fr/cidre-public/compilation/infosec-ecomp
$ cd infosec-ecomp
$ git remote rename origin le-remote-d-origine
$ git remote add origin git@votre-nouveau-depot.com/.../votre-depot.git
```

Vous pourrez alors utiliser ce dépôt git normalement (commit / push / pull) comme vous avez l'habitude. Il pourra être pratique, pour que nous vous aidions, que vous nous donniez accès en lecture à votre dépôt git. Peut-être (comprendre *sûrement*) que nous modifierons le squelette au cours des séances de TP, notamment pour mettre à jour ce sujet. À ce moment là, nous vous préviendrons et il faudra committer vos changements sur votre propre dépôt git avant de faire :

```
$ git pull le-remote-d-origine master
```

Ce qui récupérera les changements que nous aurons poussés. (La plupart du temps, ce ne sera pas pour vous embêter mais pour vous fournir du code plus robuste et mieux documenté. Il n'est pas exclus qu'on vous donne un jour la solution du TP sans faire exprès et qu'on vous demande de ne pas la regarder.)

2 Organisation du compilateur

La figure 1 donne un aperçu de la structure du compilateur que vous allez réaliser. À partir d'un fichier source *.e*, l'analyseur lexical (ou *lexer*) générera un flux de lexèmes (ou *tokens*). Ce flux sera donné à l'analyseur syntaxique (ou *parser*) qui devra générer un arbre de syntaxe abstraite (*Abstract Syntax Tree*, ou AST). L'AST sera transformé dans une séquence de langages intermédiaires :

- un programme E, qui simplement une représentation formelle, en OCaml, du programme source ;
- un programme CFG (*Control-Flow Graph*) ;
- un programme RTL (*Register Transfer Language*) ;
- un programme Linear ;
- un programme LTL (*Location Transfer Language*) ;
- un programme Assembleur RISC-V.

Chacun de ces langages intermédiaire est détaillé dans le sujet de TP qui s'y rapporte, et est illustré sur l'exemple de la Figure 2a.

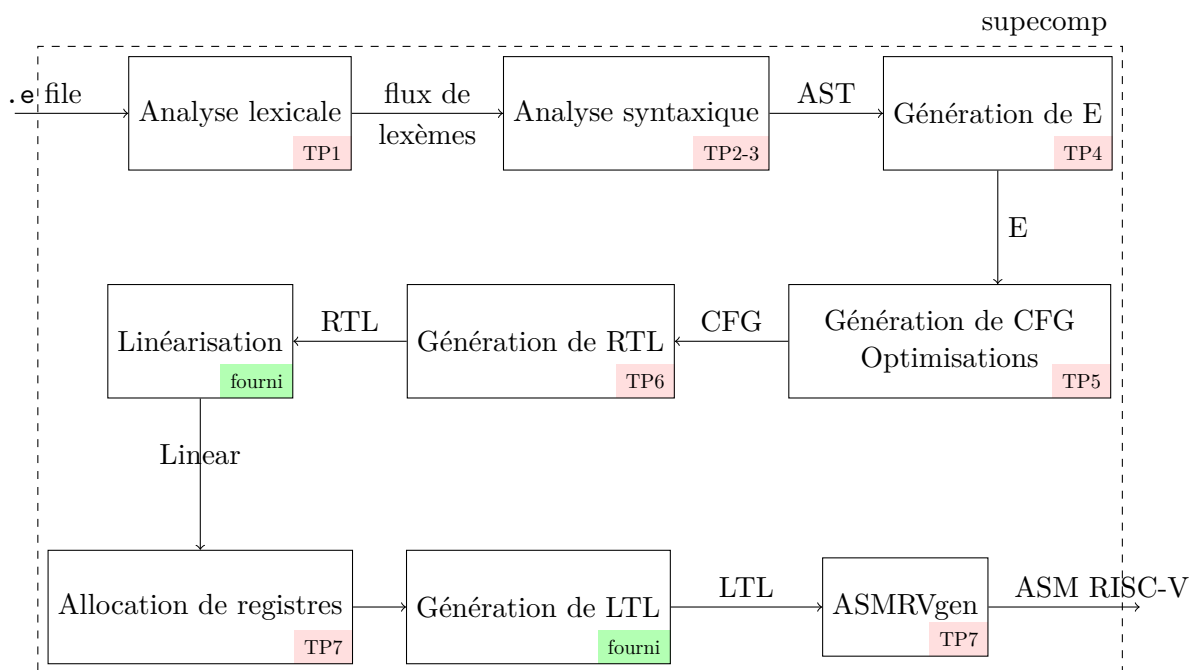
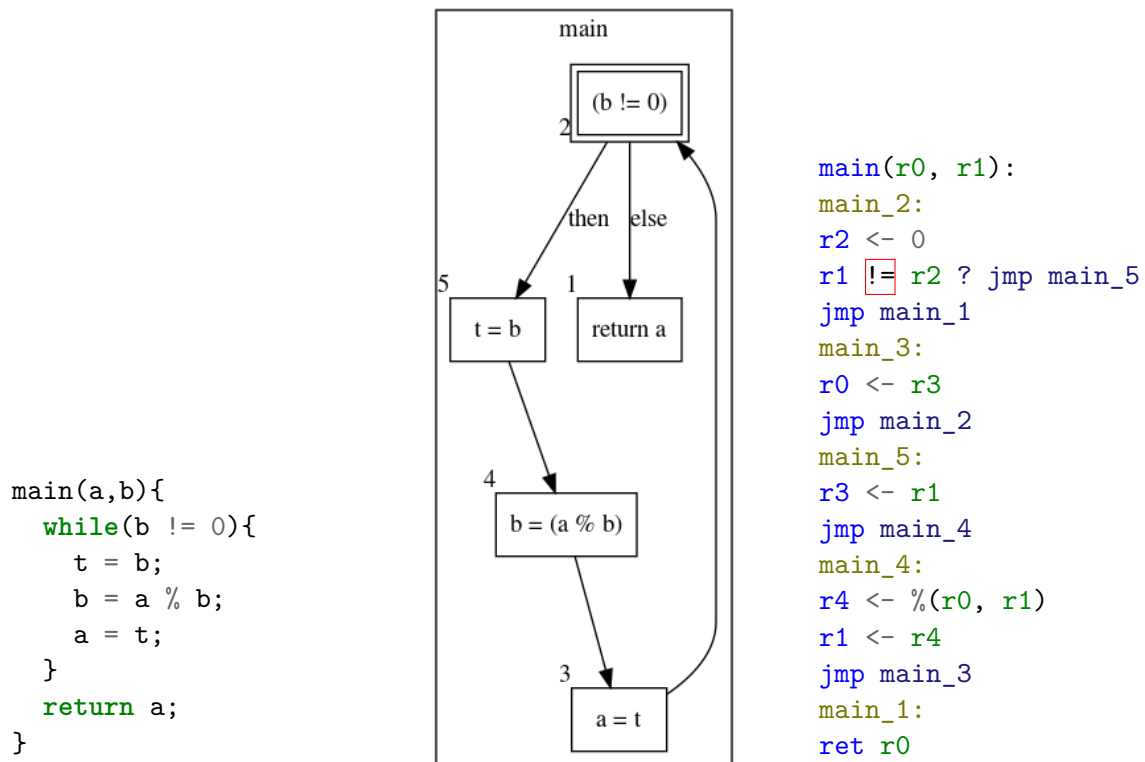


FIGURE 1 – Aperçu de la structure du compilateur



(a) Programme E

(b) CFG correspondant

(c) Programme RTL

```

.global main
main:
addi sp, sp, -8
sd ra, 0(sp)
addi sp, sp, -8
sd s0, 0(sp)
addi sp, sp, -8
sd s1, 0(sp)
addi sp, sp, -8
sd s2, 0(sp)
mv s0, sp
addi sp, sp, 0
main_2:
li s1, 0
bne a1, s1, main_5
j main_1
main_1:
mv a0, a0
j main_6
main_5:
mv s2, a1
remu s1, a0, a1
mv a1, s1
mv a0, s2
j main_2
main_6:
mv sp, s0
ld s2, 0(sp)
addi sp, sp, 8
ld s1, 0(sp)
addi sp, sp, 8
ld s0, 0(sp)
addi sp, sp, 8
ld ra, 0(sp)
addi sp, sp, 8
jr ra

```

(d) Assembleur RISC-V

FIGURE 2 – Les différents langages intermédiaires utilisés lors de la compilation d'un programme

3 Tests

Vous trouverez dans le répertoire `tests` un ensemble d'outils vous permettant de tester votre compilateur. Les dossiers `array`, `basic`, `char`, `funcall`, `globals`, `invader`, `ptr`, `type_basic`, `type_funcall` et `structs` contiennent des programmes E vous permettant de tester les fonctionnalités correspondant au nom du dossier. Durant les séances de TP, nous nous concentrerons exclusivement sur les tests du dossier `basic`.

Lors des séances de projet, vous serez amenés à améliorer votre compilateur pour étendre son langage vers du C. Lors de ces extensions, vous pourrez utiliser les tests fournis dans les différents autres répertoires.

Pour chaque fichier `test.e`, nous vous avons fourni la sortie attendue avec les paramètres 1, 2 et 3 dans `test.e.expect_1_2_3` et avec les paramètres 14, 12, 3, 8 et 12 dans `test.e.expect_14_12_3_8_12`. Vous pouvez tester que votre compilateur est conforme à ce qui est attendu en lançant `make test` depuis la racine de votre projet.

Les résultats des tests seront rassemblés dans le fichier `tests/results.html` que vous pouvez visualiser avec votre navigateur web préféré. Les résultats sont présentés sous forme de tableaux où les lignes correspondent aux programmes testés et les colonnes les résultats obtenus à différentes étapes de la compilation. Les noms des programmes sont cliquables pour avoir plus d'informations sur le déroulement de leur compilation, et leur exécution dans chacun des différents langages intermédiaires.

Pour tester les programmes individuellement vous pouvez utiliser le script `tests/test.py` ou directement le binaire produit (`main.native`) avec `make` :

```
# Commandes utiles
$ tests/test.py -f tests/basic/toto.e
$ tests/test.py -f tests/basic/*.e # équivalent à 'make test'
$ tests/test.py --help
$ ./main.native --help
```

4 TP1 : Analyseur lexical

Le but de cette séance de TP est de réaliser un analyseur lexical pour le langage E. Cette séance est l'occasion de mettre en œuvre l'algorithme vu en cours pour la réalisation d'un analyseur lexical. On vous rappelle qu'il repose sur l'utilisation d'un automate déterministe à états finis. Afin d'accélérer votre développement nous allons vous fournir une partie du code, et vous proposer une organisation de votre code.

4.1 Fonctions utiles de la librairie standard OCaml

La documentation complète est disponible en ligne sur : <https://caml.inria.fr/pub/docs/manual-ocaml/libref>

Par ailleurs, nous utilisons dans le squelette la librairie alternative **Batteries** (pour *OCaml*, *batteries included*, puisque la bibliothèque standard n'est parfois pas aussi complète qu'on pourrait le souhaiter).

Fonctions sur les listes : <https://ocaml-batteries-team.github.io/batteries-included/hdoc2/BatList.html>

- **List.mem** : vérifie si un élément appartient à une liste
- **List.fold_left** : applique une fonction **f** à chaque élément d'une liste **l** en stockant le résultat dans **acc**
- **List.map** : applique une fonction à chaque élément d'une liste
- **List.filter_map** : filtre les éléments d'une liste et applique une fonction aux éléments qui sont conservés.
- **List.mem_assoc** : étant donnée une liste d'association (une liste dont les éléments sont des paires (k, v) où k correspond à une clé et v à une valeur), détermine si une clé est présente dans la liste.
- **List.assoc_opt** : étant donnée une liste d'association, retourne la valeur associée à une clé (ou **None** si cette clé n'est pas définie)
- **List.remove_assoc** : supprime les associations pour une clé donnée, dans une liste d'association
- **List.iter** : itère une fonction de type **unit** sur chaque élément d'une liste
- **List.rev** : renverse une liste (les premiers éléments deviennent les derniers)

Fonctions sur les ensembles **Set** : <https://ocaml-batteries-team.github.io/batteries-included/hdoc2/BatSet.html>

- **Set.union** : union de deux ensembles
- **Set.add** : ajout d'un élément à un ensemble
- **Set.mem** : vérifie si un élément appartient à un ensemble
- **Set.empty** : l'ensemble vide
- **Set.is_empty** : vérifie si un ensemble est vide
- **Set.singleton** : construit un ensemble contenant un seul élément
- **Set.fold** : comme **List.fold_left**, mais ici l'ordre des éléments ne peut pas être pris en compte (pas d'ordre dans un ensemble)
- **Set.exists** : vérifie s'il existe un élément d'un ensemble satisfaisait un prédicat.

Fonctions sur les tables de hachage **Hashtbl** : <https://ocaml-batteries-team.github.io/batteries-included/hdoc2/BatHashtbl.html>

- **Hashtbl.find_option** : cherche la valeur associée à une clé dans une table de hachage.
- **Hashtbl.replace** : modifie ou crée une association clé/valeur dans une table de hachage.
- **Hashtbl.create** : crée une table de hachage vide
- **Hashtbl.keys** : la liste des clés pour lesquelles une association est connue dans une table de hachage

4.2 Tests

Pour tester le code des différentes fonctions de l'analyseur lexical, indépendamment du reste du compilateur, vous pouvez lancer la commande suivante, depuis la racine de votre projet :

```
$ make -C src test_lexer
make : on entre dans le répertoire «<PATH>/src»
ocamlbuild -use-ocamlfind test_lexer.native
Finished, 22 targets (22 cached) in 00:00:00.
./test_lexer.native
0: ([w]).([h]).([i]).([l]).([e]).(Eps))
1: ([i]).([f]).(Eps)
2:
↪ ([ABCDEFGHGIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]).([0123456789ABCDEFGHIJKLMN
==== NFA
States :
Initial states :
Final states :

[OK] epsilon_closure 1 : got , expected 1_2_3
[OK] epsilon_closure 2 : got , expected 2
[OK] epsilon_closure 3 : got , expected 2_3
[OK] epsilon_closure 4 : got , expected 4
[OK] dfa_initial_state : got , expected 1_2_3
[OK] min_priority 1 : got None, expected Some (SYM_WHILE)
[OK] min_priority 2 : got None, expected Some (SYM_IDENTIFIER(bla))
[OK] min_priority 3 : got None, expected Some (SYM_WHILE)
[OK] min_priority 4
[OK] dfa states : got {}, expected {{1,2,3}, {2,4}, {2}}
dot -Tsvg /tmp/dfa.dot -o /tmp/dfa.svg
dot -Tsvg /tmp/nfa.dot -o /tmp/nfa.svg
make : on quitte le répertoire «<PATH>/src»
```

Cela a pour effet de compiler le fichier `src/test_lexer.ml` et de le lancer. Ce fichier appelle les fonctions que vous définirez au fur et à mesure du TP dans `src/lexer_generator.ml` et compare leur résultat avec le résultat attendu.

Ce test génère des graphes au format dot, puis les convertit en SVG pour les afficher d'une manière plus ou moins compréhensible par un humain. Si ce n'est pas déjà le cas, installez le paquet **graphviz** qui contient l'utilitaire **dot**. Le compilateur en aura par ailleurs besoin par la suite. Vous pouvez examiner les automates non-déterministe et déterministe générés en ouvrant les fichiers `/tmp/nfa.svg` et `/tmp/dfa.svg`, par exemple avec votre navigateur web.

Pour compiler et tester votre code au sein du compilateur, il suffit de lancer `make test` dans le répertoire racine de votre projet. Les résultats des tests sont stockés dans le fichier `tests/results.html`.

Résultats attendus sur un exemple :

<pre>\$ cat tests/basic/just_a_variable_37.e main(){ just_a_variable = 37; return just_a_variable; }</pre>	<pre># À la fin du TP, results.html: SYM_IDENTIFIER(main) SYM_LPARENTHESIS SYM_RPARENTHESIS SYM_LBRACE SYM_IDENTIFIER(just_a_variable) SYM_ASSIGN SYM_INTEGER(37) SYM_SEMICOLON SYM_RETURN SYM_IDENTIFIER(just_a_variable) SYM_SEMICOLON SYM_RBRACE SYM_EOF</pre>
<pre># Au début du TP dans results.html: Lexing error: Lexer failed to recognize string starting with ↪ 'main(){ just_a_var'</pre>	

Lorsque vous appelez `make test`, votre compilateur est lancé sur 38 fichiers de tests (les fichiers `tests/basic/*.e`). Pour le moment, le fichier `tests/results.html` indique que tous ces tests échouent à l'analyse lexicale puisque votre analyseur n'est pas encore écrit. Au fur et à mesure des séances de TP, ce fichier vous donnera de plus en plus d'information, notamment le résultat de l'analyse lexicale, syntaxique ainsi que le résultat de l'exécution de chacun des programmes de test à différents niveaux dans la chaîne de compilation. Cela sera un bon moyen de valider la correction de vos passes de compilation.

Vous pouvez aussi lancer le compilateur « à la main », c'est-à-dire sans passer par le `make test` :

```
$ make
$ ./main.native -f tests/basic/just_a_variable_37.e -show-tokens -
```

pour lancer le compilateur sur le fichier `tests/basic/just_a_variable_37.e` et afficher les tokens reconnus. (Le « - » à la fin de la ligne de commande indique qu'on souhaite afficher les tokens sur la sortie standard. Si on veut les écrire dans un fichier, on remplacera ce « - » par le nom du fichier.)

4.3 Travail à effectuer

Le développement de notre analyseur lexical se déroule en trois étapes.

1. Premièrement, la spécification des expressions régulières permettant de reconnaître les termes du langage E.
2. Ensuite, un NFA (*Non-deterministic Finite Automaton*) pourra être généré pour ces expressions régulières.
3. Finalement, ce NFA sera transformé en DFA (*Deterministic Finite Automaton*) qui sera utile à l'analyseur pour reconnaître les termes du langage E et les associer au bon lexème.

Le travail que vous réaliserez au cours de cette séance se déroulera dans les fichiers `src/lexer_generator.ml` et `src/e_regex.ml`.

4.3.1 Expressions régulières du langage E

Un premier travail est de donner à l'analyseur différentes expressions régulières permettant d'identifier les mots-clés et noms de variables du langage E. Pour vous familiariser avec le langage

E, n'hésitez pas à parcourir le répertoire `tests/basic`, où une trentaine d'exemples vous sont donnés.

Ouvrez le fichier `src/e_regex.ml`. On y définit le type des expressions rationnelles (ou régulières) `regex`. Les différents variants correspondent aux différentes constructions d'expressions régulières vues en cours. (À l'exception du variant `Charset s`, qui représente directement un caractère parmi un ensemble, plutôt que d'utiliser beaucoup de `Alt (.,.)`, pour des raisons de praticité, et de performance.)

On y définit ensuite un alphabet, constitué de toutes les lettres minuscules et majuscules, les chiffres, ainsi qu'un certain nombre de symboles non-alphanumériques.

Ensuite vient la définition de `list_regex`, la liste des expressions régulières, chacune correspondant à un lexème de notre langage E. Cette liste est de type `(regex * (string -> token option)) list`. Chaque élément de la liste est donc une paire (r, t) où r est une expression régulière (type `regex`) et t est une fonction qui construit, à partir de la chaîne de caractère capturée par l'expression régulière, un lexème (type `token`). Pour les lexèmes de type mot-clé, qui sont constants, la chaîne de caractère passée en paramètre à la fonction t n'aura pas d'incidence. En revanche, pour les lexèmes comme `SYM_IDENTIFIER`, la chaîne de caractère correspondra à l'identifiant à proprement parler.

Un certain nombre d'expressions régulières associées à des lexèmes est déjà construit pour vous. Il vous reste à compléter cette liste pour un certain nombre de mots-clés (là où l'expression régulière `Eps` est utilisée, en attendant que vous ne complétiez le code).

Question 4.1. Compléter la fonction `list_regex` du fichier `src/e_regex.ml` en remplaçant les regex `Eps` par une expression régulière adéquate. À noter que la liste des lexèmes (type `token`) est disponible dans le fichier `src/symbols.ml`.

4.3.2 Expressions régulières en NFAs

Nous souhaitons maintenant produire le NFA correspondant aux expressions régulières utilisées pour analyser le langage.

Dans le fichier `src/lexer_generator.ml`, le type `nfa` décrit ce qu'est un automate fini non-déterministe (NFA). Le champ `nfa_final` est une liste de paires (q, t) où q est un état de l'automate, qui est déclaré comme étant final, et t est une fonction de type `string -> token option` (le même type que celui que l'on vient de voir dans le fichier `src/e_regex.ml`). Cette fonction construit, lorsque c'est possible, un lexème à partir de la chaîne de caractères reconnue par l'automate.

Question 4.2. Écrire les fonctions `cat_nfa`, `alt_nfa` et `star_nfa` du fichier `src/lexer_generator.ml`. Ces fonctions permettent respectivement la concaténation, l'union et la répétition d'automates `nfa`.

Le type `nfa` est décrit et commenté au début du fichier.

Ces fonctions doivent reproduire la méthode de traduction des expressions rationnelles en automates finis non-déterministes présentée en cours. La fonction `star_nfa` est paramétrée par une fonction t de type `string -> token option`, qui doit être associée à l'état final de l'automate construit.

Question 4.3. Compléter la fonction `nfa_of_regex` qui produit un NFA à partir d'une expression régulière. Les différents paramètres de cette fonction sont décrits dans le squelette du TP.

Les cas `Eps` et `Charset` `c` sont donnés en exemple. Traiter les variantes restantes de `regex`.

Vous pouvez tester votre implémentation ici en lançant, depuis le répertoire `src` :

```
$ make test_lexer
```

Cela va afficher un certain nombre d'informations, la plupart n'étant pas pertinentes pour le moment. Le fichier `"/tmp/nfa.dot"` a du cependant être généré, ainsi qu'un fichier `"/tmp/nfa.svg"`. Vous pouvez ouvrir ce dernier fichier et vous assurer que ce que vous visualisez correspond à ce que vous attendiez, pour la liste d'expressions régulières définie au début du fichier `src/test_lexer.ml`, à savoir un NFA qui reconnaît les mots-clés `if` et `while`, ainsi que des identifiants.

4.3.3 Déterminisation d'un NFA en DFA

Dans cette partie, on s'intéresse à transformer notre NFA en un DFA, qui permettra de construire notre analyseur lexical, en suivant les étapes décrites en cours. Le type `dfa` utilisé dans cette partie est défini et commenté dans le fichier `src/lexer_generator.ml`.

Question 4.4. Compléter la fonction `epsilon_closure` qui retourne l'ensemble des états accessibles par ϵ -transitions à partir d'un état d'un automate `nfa`.

Il faudra notamment écrire la fonction interne `traversal` (`visited: nfa_state set`) (`s: nfa_state`) : `nfa_state set`, qui parcourt l'automate en ne suivant que les ϵ -transitions. Le paramètre `visited` de cette fonction contient l'ensemble des états déjà visités et `s` est l'état à partir duquel on souhaite commencer le parcours.

Les quelques tests relatifs à la fonction `epsilon_closure` devraient afficher `OK`, après un `make lexer_test`.

Question 4.5. Compléter la fonction `epsilon_closure_set` qui retourne l'ensemble des états accessibles par ϵ -transitions à partir d'un ensemble d'états d'un automate `nfa`.

Question 4.6. Écrire la fonction `dfa_initial_state` qui construit l'état initial du DFA correspondant au NFA `n`.

Nous avons maintenant l'état initial de notre DFA. Nous allons désormais construire sa table de transition. Cette table est construite récursivement grâce à la fonction `build_dfa_table`. Cette fonction prend en paramètre une table partiellement construite `table`. Cette table est une table de hachage dont les clés sont des `dfa_state` et les valeurs sont des listes de paires (c, s) où `c` est un caractère de l'alphabet et `s` est l'état dans lequel on transite en lisant ce caractère. Les autres paramètres de la fonction sont `n`, le NFA que l'on souhaite déterminer, et `ds`, l'état du DFA en cours de construction à partir duquel on souhaite ajouter des transitions.

Une partie du code de cette fonction vous est fournie. Si l'état `ds` a déjà été traité, on renvoie `()`, *i.e.* on ne fait rien ; on a terminé ! Dans le cas contraire, on doit calculer l'ensemble des transitions

possibles depuis cet état `ds`. La procédure de construction de ces transitions a été vue en cours, et est détaillée dans un commentaire au-dessus de cette fonction. Un certain nombre de fonctions auxiliaires, qui vous seront utiles, ont été définies pour vous.

Question 4.7. Construire l'expression `transitions` de la fonction `build_dfa_table`. Cette fonction permet de construire la table de transitions d'un `dfa` à partir d'un `nfa`. Les différentes étapes permettant de construire cette table de transitions sont spécifiées dans les commentaires situées au-dessus de la fonction.

Nous avons à présent l'état initial de notre DFA, ainsi que sa table de transitions. Il ne nous reste plus qu'à déterminer les états finaux, et notre DFA sera terminé!

Question 4.8. Compléter les fonctions `min_priority` et `dfa_final_states` permettant de définir les états finaux de notre `dfa`. Les états finaux d'un `dfa` correspondent aux états qui contiennent au moins un état final d'un `nfa`. La fonction de conversion associée à un état final est obtenue en faisant usage de `min_priority`. L'utilité de `min_priority` est expliquée dans un commentaire du squelette.

Question 4.9. Compléter la fonction de transition `make_dfa_step` en utilisant la table de transition construite précédemment avec `build_dfa_table`. Cette fonction devrait être très simple, en se basant sur la table construite précédemment.

La fonction `dfa_of_nfa`, qui construit un DFA à partir d'un NFA, vous est offerte. Il s'agit simplement d'utiliser toutes les fonctions que vous avez écrites jusqu'à maintenant.

Vous pouvez à présent relancer `make test_lexer` et observer le fichier `/tmp/dfa.svg` qui aura été généré. Cela devrait ressembler au DFA obtenu en toute fin de cours.

4.3.4 Obtention de lexèmes à l'aide du DFA

Nous avons maintenant obtenu un DFA capable de reconnaître les mots-clés et variables du langage E. Nous souhaitons maintenant que notre DFA décompose les chaînes de caractères d'un programme E en une série de lexèmes/jetons définis dans le fichier `src/symbols.ml`.

Question 4.10. Complétez la fonction `tokenize_one`. Celle-ci contient une fonction récursive `recognize` qui effectue des transitions dans le DFA tant que possible et retourne un jeton lorsqu'il aboutit.

Comme précédemment, de nombreux détails vous sont donnés dans les commentaires du fichier `src/lexer_generator.ml`.

Note : vous aurez besoin de la fonction `string_of_char_list` qui transforme un `char list` en `string`

Nous vous offrons les dernières étapes, à savoir

- la fonction `tokenize_all` qui répète `tokenize_one` tant qu'il y a des lexèmes à lire,
- la fonction `tokenize_file` qui transforme un nom de fichier en la liste des lexèmes qui sont reconnus dans ce fichier,
- les morceaux de code dans `main.ml` qui appellent le lexer.

Si tout va bien, un `make test` maintenant vous affiche plein d'erreurs, mais de syntaxe seulement :-)

5 TP2 : Analyseur syntaxique

Lors du TP précédent, vous avez écrit un analyseur lexical pour le langage E, et avez donc obtenu, à partir d'un fichier source `.e` un flux de lexèmes. Le but de ce TP est de construire un analyseur syntaxique. Pour ce faire, vous allez devoir écrire la grammaire du langage E dans un format spécifique et utiliser un générateur d'analyseur syntaxique.

5.1 ALPAGA : An LL(1) PARser GenerAtor

Il existe un certain nombre de générateurs d'analyseurs syntaxiques, les plus connus étant `yacc` (*yet another compiler compiler*) et `bison` (qui génèrent du code C), leur cousin `ocamlyacc` (qui génère du code Ocaml), `menhir` (qui génère aussi du code Ocaml, mais également du Coq!), ANTLR (ANother Tool for Language Recognition, écrit en Java et qui génère du Java, C#, python, JavaScript, Go, C++, et du Swift). Tous ces outils acceptent une grammaire en entrée, dans un format particulier, et produisent du code source qui parcourt le flux de lexèmes et produisent un arbre de syntaxe abstraite.

Afin d'avoir un contrôle fin sur le parser généré, et pour pouvoir exporter un certain nombre d'informations utiles lors de l'écriture de la grammaire, nous avons choisi de construire notre propre outil, et ainsi ajouter ALPAGA (*An LL(1) PARser GenerAtor*) au bestiaire des générateurs de parsers. ALPAGA est écrit en OCaml et produit du code OCaml (une version qui produit du C a aussi existé pour vos camarades de feu la majeure SIS) qui pourra être intégré à votre compilateur.

En plus du code de l'analyseur syntaxique, ALPAGA produit un fichier HTML qui contient un certain nombre d'informations intéressantes. Regardons par exemple la Figure 3, le fichier généré par une toute petite grammaire.

Grammaire

(1)S	-> <u>EXPR</u> SYM_EOF
(2)EXPR	-> <u>IDENTIFIER</u>
(3)	-> <u>INTEGER</u>
(4)	-> <u>EXPR</u> SYM_PLUS <u>EXPR</u>
(5)	-> <u>EXPR</u> SYM_ASTERISK <u>EXPR</u>
(6)INTEGER	-> SYM_INTEGER
(7)IDENTIFIER	-> SYM_IDENTIFIER

(a) La grammaire (cliquable)

Table First

Non-terminal	First
S	SYM_IDENTIFIER SYM_INTEGER
EXPR	SYM_IDENTIFIER SYM_INTEGER
INTEGER	SYM_INTEGER
IDENTIFIER	SYM_IDENTIFIER

(b) La relation First

Table LL

	SYM_EOF	SYM_IDENTIFIER	SYM_INTEGER	SYM_PLUS	SYM_ASTERISK
S		<u>1</u>	<u>1</u>		
EXPR		<u>2 4 5</u>	<u>3 4 5</u>		
INTEGER			<u>6</u>		
IDENTIFIER		<u>7</u>			

(c) La table LL

FIGURE 3 – Fichier généré pour une petite grammaire d'expressions

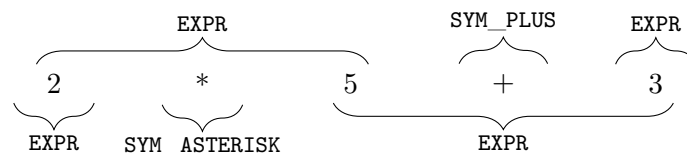
On peut donc voir (Figure 3a) la grammaire que l'on a écrite, où chaque règle est numérotée, et chaque non-terminal de la grammaire est un lien vers l'ensemble des règles associées à ce non-

terminal. Cela paraît anecdotique pour cette toute petite grammaire, mais votre grammaire ne tiendra pas sur un seul écran et cette fonctionnalité sera alors très appréciée.

La Figure 3b donne pour chaque non-terminal, l'ensemble des terminaux qui peuvent commencer ce non-terminal. On voit que les non-terminaux **S** et **EXPR** acceptent un identifiant ou un entier, comme premier lexème. Des tables similaires existent pour les fonctions **Null** et **Follow**, vues en cours.

Finalement, la Figure 3c montre la table de prédiction qui va servir de matrice au parser généré. Chaque ligne correspond à un non-terminal que l'on souhaite parser. **S** est un nom classique pour désigner la règle de départ d'une grammaire, qu'on appelle l'*axiome* de la grammaire. Chaque colonne correspond à un terminal (lexème, token, symbole) qui vient du lexer. Si la case (NT, T) est vide, cela signifie que le non terminal NT ne peut pas commencer par le terminal T, autrement dit $T \notin First(NT)$. Si la case contient un numéro n , cela signifie qu'il faut appliquer la règle portant ce numéro. Si la case contient plusieurs numéros, il y a un conflit ; notre grammaire est ambiguë.

Effectivement, comment doit-on analyser l'expression $2 * 5 + 3$?



Les deux dérivations sont conformes à la grammaire, mais donnent deux résultats différents : $(2 * 5) + 3$ d'un côté, qui vaut 13, et $2 * (5 + 3)$ de l'autre, qui vaut 16. Bien sûr, vous avez appris à l'école que l'addition est plus prioritaire que la multiplication et que c'est donc la première solution qui est la bonne. Pour expliquer cela à notre grammaire, comme vu en cours, il faut écrire la grammaire autrement, comme dans la figure 4. Dans la table LL (Figure 4c), les numéros de règle en bleu correspondent aux règles qui peuvent être appliqués car le terminal en question peut commencer ce non-terminal ($t \in First(nt)$) ; les numéros en rouge correspondent aux règles qui peuvent être appliquées lorsque le non-terminal est *nullable* et que le terminal peut *suivre* ce non-terminal ($Null(nt) \wedge t \in Follow(nt)$).

Comme on le voit, la grammaire est plus compliquée à écrire, moins naturelle, mais non-ambiguë et la table générée est sans conflits.

5.1.1 Format de la grammaire

Voici le fichier de grammaire donné à ALPAGA pour l'exemple précédent.

```
tokens SYM_EOF SYM_IDENTIFIER<string> SYM_INTEGER<int> SYM_PLUS SYM_ASTERISK
non-terminals S EXPR TERM TERMS FACTOR FACTORS INTEGER IDENTIFIER
axiom S
rules
S -> EXPR SYM_EOF
IDENTIFIER -> SYM_IDENTIFIER
INTEGER -> SYM_INTEGER
EXPR -> TERM TERMS
TERM -> FACTOR FACTORS
TERMS -> SYM_PLUS TERM TERMS
TERMS ->
FACTOR -> IDENTIFIER
```

Grammaire

(1) S	-> <u>EXPR</u> <u>SYM</u> <u>EOF</u>
(2) EXPR	-> <u>TERM</u> <u>TERMS</u>
(3) TERM	-> <u>FACTOR</u> <u>FACTORS</u>
(4) TERMS	-> <u>SYM_PLUS</u> <u>TERM</u> <u>TERMS</u>
(5)	-> ϵ
(6) FACTOR	-> <u>IDENTIFIER</u>
(7)	-> <u>INTEGER</u>
(8) FACTORS	-> <u>SYM_ASTERISK</u> <u>FACTOR</u> <u>FACTORS</u>
(9)	-> ϵ
(10) INTEGER	-> <u>SYM</u> <u>INTEGER</u>
(11) IDENTIFIER	-> <u>SYM</u> <u>IDENTIFIER</u>

(a) La grammaire (cliquable)

Table First

Non-terminal	First
S	SYM IDENTIFIER SYM INTEGER
EXPR	SYM IDENTIFIER SYM INTEGER
TERM	SYM IDENTIFIER SYM INTEGER
TERMS	SYM PLUS
FACTOR	SYM IDENTIFIER SYM INTEGER
FACTORS	SYM ASTERISK
INTEGER	SYM INTEGER
IDENTIFIER	SYM IDENTIFIER

(b) La relation First

Table LL

	SYM_EOF	SYM_IDENTIFIER	SYM_INTEGER	SYM_PLUS	SYM_ASTERISK
S		<u>1</u>	<u>1</u>		
EXPR		<u>2</u>	<u>2</u>		
TERM		<u>3</u>	<u>3</u>		
TERMS	<u>5</u>			<u>4</u>	
FACTOR		<u>6</u>	<u>7</u>		
FACTORS	<u>9</u>			<u>9</u>	<u>8</u>
INTEGER			<u>10</u>		
IDENTIFIER		<u>11</u>			

(c) La table LL

FIGURE 4 – Fichier généré pour une petite grammaire d'expressions

```
FACTOR -> INTEGER
FACTORS -> SYM_ASTERISK FACTOR FACTORS
FACTORS ->
```

Le format du fichier est donc le suivant :

- Déclarations de terminaux (tokens). On déclare les terminaux qui vont être utilisés. Dans notre cas, il s'agira de l'ensemble des symboles (éléments du type `token` du fichier `src/symbols.ml`) générés au TP précédent. On place le mot clé `tokens` suivi de la liste des tokens. On peut donner plusieurs telles lignes.
À noter que pour les lexèmes non-constants (`SYM_IDENTIFIER` et `SYM_INTEGER`) on spécifie le type du paramètre entre chevrons : `<string>` ou `<int>`.
- Déclarations de non-terminaux. De manière similaire, on déclare la liste des non-terminaux que l'on va reconnaître, les uns à la suite des autres, après le mot-clé `non-terminals`.
- Déclaration de l'axiome de la grammaire : `axiom S`
- Le mot-clé `rules`. Cela délimite les déclarations de terminaux et non-terminaux de la suite du fichier.
- Une suite de règles, composées de :
 - un non-terminal
 - une flèche (`->`)
 - une suite (éventuellement vide) de terminaux et non-terminaux

Par exemple, la règle `TERM -> FACTOR FACTORS` signifie que le non-terminal `TERM` peut être reconnu en reconnaissant d'abord le non-terminal `FACTOR`, puis le non-terminal `FACTORS`. Autre exemple, la règle `FACTORS ->` signifie que le non-terminal `FACTORS` peut être reconnu par une suite vide de symboles.

5.1.2 Options d'ALPAGA

Le programme ALPAGA répond gentiment à l'option `--help` :

```
$ alpaga/alpaga --help
Usage:
  -g Input grammar file (.g)
  -t Where to output tables (.html)
  -pml Where to output the parser code (.ml)
  -help Display this list of options
  --help Display this list of options
$ alpaga/alpaga -g fichier-grammaire.g -t table.html -pml mon-parser.ml
```

Cela lira le fichier de grammaire `fichier-grammaire.g` et générera le code du parser dans `mon-parser.ml`. Une autre sortie du générateur est un fichier `table.html`. Ce fichier, que vous pouvez ouvrir dans un navigateur web, vous montrera votre grammaire dans un format cliquable, les tables Null, First et Follow nécessaires à la création du parser, et finalement la table LL obtenue. Notamment, les cellules de la table s'afficheront en fond rouge si un conflit a été détecté. Si tel est le cas, il faudra réécrire votre grammaire pour lever les ambiguïtés.

Note : Votre compilateur s'attend à trouver le parser généré dans le fichier `src/generated_parser.ml`. Faites en sorte de ne pas le contrarier. (Ou plutôt : ne lancez pas ALPAGA à la main, le Makefile s'occupera de ça très bien pour vous.)

Le fichier `expr_grammar_action.g` contient un début de grammaire avec une règle qui comporte une action (du code entre accolades). Ignorez ce bout de code pour le moment, on y reviendra dans les questions suivantes.

Grammaire informelle de E. Afin de vous guider dans l'écriture de la grammaire du langage E, voici une grammaire informelle du langage.

- Un programme est constitué d'une liste de déclarations de fonctions (pour le moment, une seule fonction, qui doit s'appeler `main`).
- Une déclaration de fonction est composée du nom de la fonction suivi, entre parenthèse d'une liste d'arguments séparés par des virgules, suivi enfin du corps de la fonction.
- Le corps d'une fonction est composée d'instructions. Les instructions peuvent être :
 - un bloc d'instructions à l'intérieur d'accolades
 - une conditionnelle de la forme `if (c) { i1 } else { i2 }`, où `c` est une expression, et `i1` et `i2` sont des séquences d'instructions. La partie `else` est optionnelle.
 - une boucle de la forme `while (c) i`, où `c` est une expression, et `i` est une instruction.
 - une instruction `return e`; où `e` est une expression.
 - une instruction `print (e)`; où `e` est une expression.
- Les expressions sont similaires à celles du C, où l'on se restreint aux opérateurs `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `%`, ainsi que l'opérateur unaire `-`. Les expressions *de base* sont les entiers,

les identifiants de variables. On peut également parenthéser des sous-expressions. On pourra s'inspirer de la priorité des opérateurs du C¹.

Question 5.1. Complétez le fichier `expr_grammar_action.g` la grammaire pour le langage E, dans le format attendu par ALPAGA. Votre analyseur devrait consommer les lexèmes mais ne pas produire d'AST (vous n'avez rien fait pour cela encore).

Au fur et à mesure de l'écriture de votre grammaire, lancez `make` à la racine de votre projet et examinez le fichier `grammar.html` afin de vous assurer que vous n'introduisez pas de conflits.

Cet analyseur simple vous permettra de vous assurer que votre grammaire est correcte, indépendamment des actions que vous écrirez par la suite. Ainsi, sur un programme syntaxiquement correct, votre analyseur devrait réussir silencieusement. Pour un programme syntaxiquement incorrect, vous devriez obtenir une erreur de syntaxe.

Votre analyseur devrait réussir sur tous les fichiers `.e` présents dans le répertoire `tests`, à l'exception des fichiers `syntax_error*.e`. Plus précisément, la table du fichier `tests/results.html` devrait contenir une erreur de génération de E, et non plus une erreur de syntaxe.

À ce stade, vous savez donc reconnaître des programmes syntaxiquement valides, mais ne construisez pas d'arbre de syntaxe abstraite. Intéressons-nous maintenant à ce problème.

ALPAGA permet de spécifier, pour chaque règle de la grammaire, une **action**, *i.e.* une expression OCaml qui construit *quelque chose* pour cette règle. Par défaut, lorsqu'aucune action n'est explicitement spécifiée, l'action utilisée est `()` (la constante de type `unit`). Pour spécifier une action, il suffit d'ajouter à la fin de la ligne correspondant à une règle, du code OCaml entre accolades.

Regardons par exemple la grammaire, avec actions, ci-dessous :

```
tokens SYM_EOF SYM_IDENTIFIER<string> SYM_INTEGER<int> SYM_PLUS SYM_ASTERISK
non-terminals S EXPR TERM FACTOR INTEGER IDENTIFIER
non-terminals TERMS FACTORS
{
  let resolve_associativity term terms = ...
}
rules
S -> EXPR SYM_EOF { $1 }
IDENTIFIER -> SYM_IDENTIFIER { StringLeaf($1) }
INTEGER -> SYM_INTEGER { IntLeaf($1) }
EXPR -> TERM TERMS { resolve_associativity $1 $2 }
TERM -> FACTOR FACTORS { resolve_associativity $1 $2 }
TERMS -> SYM_PLUS TERM TERMS { (Tadd, $2) :: $3 }
TERMS -> { [] }
FACTOR -> IDENTIFIER { $1 }
FACTOR -> INTEGER { $1 }
FACTORS -> SYM_ASTERISK FACTOR FACTORS { (Tmul,$2) :: $3 }
FACTORS -> { [] }
```

Quelques remarques sur cette grammaire :

1. https://en.cppreference.com/w/c/language/operator_precedence

- On a inséré un bloc de code, entre accolades, juste avant la ligne 'rules'. Ce bloc de code sera inséré dans le code du parser généré, au tout début du fichier. On peut donc y définir des fonctions, importer des modules (`open Symbols`), qui seront utilisables dans les actions de la grammaire.
- Les règles pour les non-terminaux IDENTIFIER et INTEGER construisent des feuilles de l'arbre de syntaxe abstraite. Le type des arbres est défini dans `src/ast.ml`.
- On peut utiliser dans les actions des variables spéciales \$1, \$2, ... La variable \$i correspond au terme OCaml renvoyé par le i-ième élément de la règle. Concrètement, dans la règle :

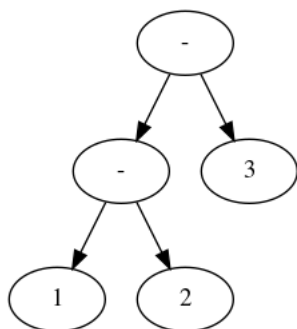
```
TERMS -> SYM_PLUS TERM TERMS { (Tadd, $2) :: $3 }
```

la variable \$2 correspond au résultat renvoyé par TERM et la variable \$3 correspond au résultat renvoyé par TERMS.

- Sur des terminaux, la variable \$i renvoie la chaîne de caractères qui a servi à reconnaître ce terminal. Utilisé notamment pour les terminaux SYM_IDENTIFIER et SYM_INTEGER qui sont respectivement de type `string` et `int`.
- Dans la règle `EXPR -> TERM TERMS { resolve_associativity $1 $2 }`, on appelle une fonction sur les éléments produits par les non-terminaux TERM et TERMS. En regardant un peu les autres règles, on comprend qu'un appel à `resolve_associativity` pourrait être effectué avec les paramètres :

```
resolve_associativity (IntLeaf(1)) [(Tsub, IntLeaf 2); (Tsub, IntLeaf 3)]
```

ce qui correspond à l'analyse syntaxique de la phrase $1 - 2 - 3$. Le but de la fonction `resolve_associativity` est de construire l'arbre suivant :



```
Node (Tsub, [
  Node (Tsub, [IntLeaf 1, IntLeaf 2]);
  IntLeaf 3
])
```

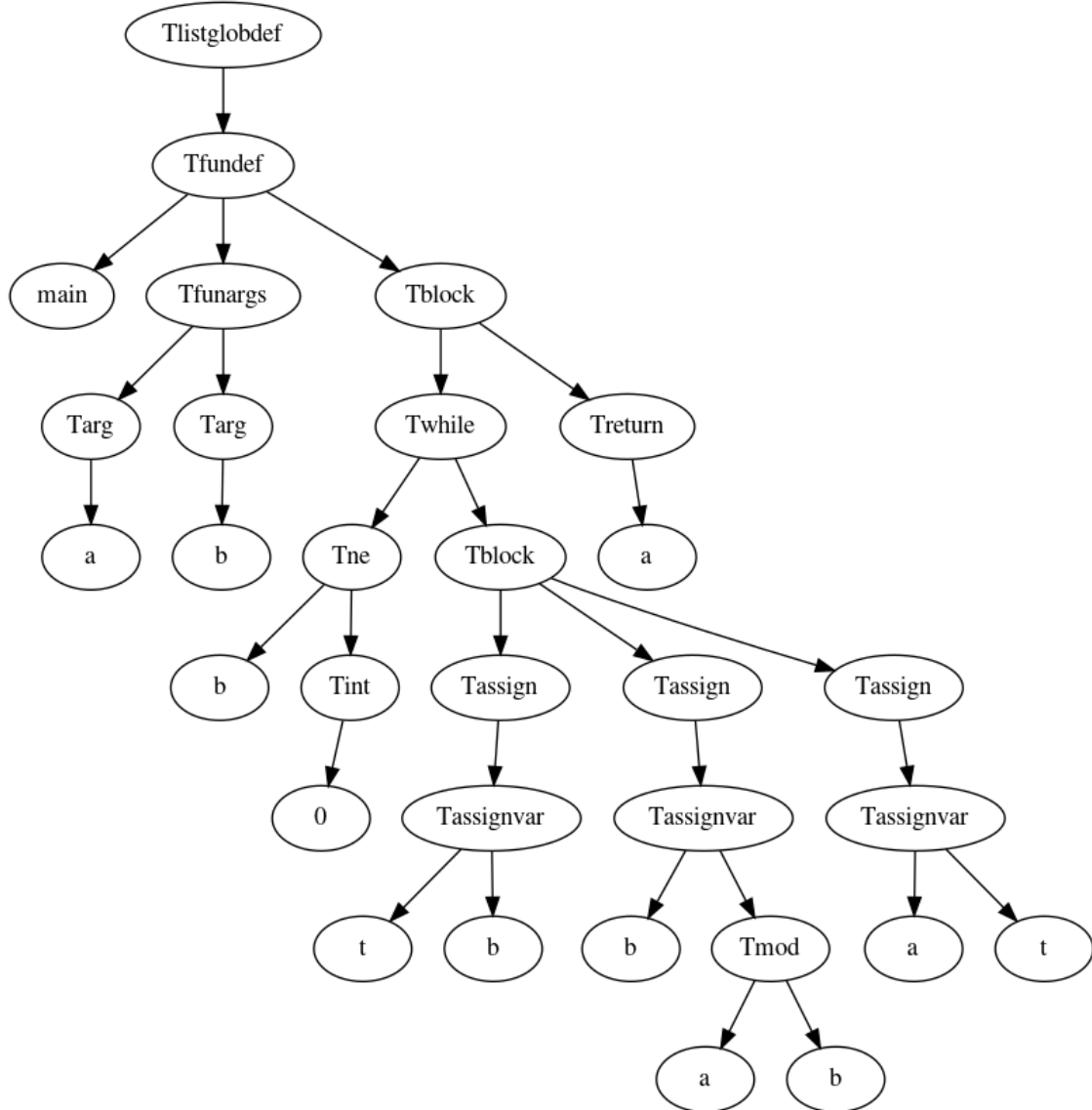
Question 5.2 (Action!). Ajouter des actions à votre grammaire afin de construire un AST similaire à celui montré ci-dessus. Référez-vous au fichier `src/ast.ml` qui documente le type des arbres de syntaxe abstraite.

Il vous faudra notamment écrire le corps de la fonction `resolve_associativity`, dont un squelette se trouve dans le fichier `expr_grammar_action.g`.

Quand vous aurez écrit les actions, pour tester, relancer `make test` depuis la racine de votre projet pour que :

- ALPAGA régénère le parseur dans le fichier `src/generated_parser.ml` ;
- tout le compilateur soit recompilé (wow!) ;
- les tests soient relancés.

Après cela, dans la table de `tests/results.html`, la page accessible depuis chaque lien correspondant à un nom de fichier devrait contenir un arbre de syntaxe abstraite (une représentation graphique de l'arbre que vous aurez construit). Ce qui devrait ressembler à cela (pour le programme de test `tests/basic/gcd.e`) :



6 TP3 : Génération et interprétation de programmes E

À partir de l'AST, vous allez maintenant devoir générer un programme E, et écrire un interpréteur pour ces programmes. La représentation OCaml des programmes E est définie dans le fichier `elang.ml`, décrit en cours.

Le fichier `elang_print.ml` contient quelques fonctions d'affichage d'expressions, instructions et programmes E. Jetez-y un œil pour déboguer vos programmes.

6.1 Génération de E

L'objectif est de générer un programme E à partir d'un AST. Les fonctions suivantes sont à compléter dans le fichier `src/elang_gen.ml`.

Question 6.1. Écrivez la fonction `make_eexpr_of_ast (a: tree) : expr res` qui transforme un sous-arbre `a` en une expression E. Cette fonction renvoie une erreur si l'arbre ne répond pas au format attendu.

Question 6.2. Écrivez la fonction `make_einstr_of_ast (a: tree) : instr res` qui transforme un sous-arbre `a` en une instruction E. Cette fonction renvoie une erreur si l'arbre ne répond pas au format attendu.

Question 6.3. Complétez la fonction `make_fundef_of_ast (a: tree) : (string * efun) res` qui transforme un sous-arbre `a` en une définition de fonction E. Cette fonction renvoie une erreur si l'arbre ne répond pas au format attendu.

Question 6.4. Complétez la fonction `make_eprog_of_ast (a: tree) : eprog res` qui transforme un sous-arbre `a` en un programme E. Cette fonction renvoie une erreur si l'arbre ne répond pas au format attendu.

6.2 Interprétation de E

Un moyen de vérifier que votre construction d'AST et transformation en E est correcte, est d'interpréter vos programmes, *i.e.* de les exécuter ! Pour cela, vous allez écrire une fonction qui prend un programme E et une liste d'arguments, et qui rend la valeur retournée par ce programme.

Vous allez avoir besoin d'un **état de programme** : une structure de données qui mémorise la valeur de chaque variable tout au long de l'exécution de votre programme. Le fichier `prog.ml` contient notamment le type `'a state`.

```
type 'a state = {  
  env: (string, 'a) Hashtbl.t;  
  mem: Mem.t  
}
```

```
let init_state memsize =
{
  mem = Mem.init memsize;
  env = Hashtbl.create 17;
}
```

Nous reviendrons sur la composante “mémoire” plus tard, lorsque l’on en aura besoin. Pour le moment, concentrons-nous sur l’environnement. Il s’agit d’une table de hachage où les clés sont des noms de variables et les valeurs sont de type `'a`. Dans un premier temps, oninstanciera `'a` avec `int`. Plus tard, on pourra enrichir avec le type des variables.

On utilisera donc les fonctions `Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` et `Hashtbl.find_option : ('a, 'b) Hashtbl.t -> 'a -> 'b option` pour écrire et lire dans l’environnement.

Question 6.5 (Passage de paramètres). Dans la fonction `eval_eprog : (oc: formatter) (ep: eprog) (memsize : int) (params: int list) : int option res`, construisez l’état initial du programme avec la liste d’arguments passés en paramètres. Notez que la liste ne contient que les valeurs; les noms des paramètres sont dans le programme lui-même (dans `f.funargs`).

Lancez votre compilateur avec l’option `-e-run` suivi des paramètres que vous voulez passer, et affichez l’état de votre programme pour vérifier que tout se passe bien.

Note : ignorez les paramètres superflus de la ligne de commande (par exemple, si votre programme attend 2 paramètres et que vous en fournissez 4, ignorez les deux derniers).

Passons maintenant à l’évaluation des expressions !

Question 6.6. Écrivez le corps des fonctions `eval_unop : unop -> int -> int` et `eval_binop : binop -> int -> int -> int`, qui prennent un opérateur (unaire ou binaire) et leurs paramètres entiers, et renvoient le résultat.

Question 6.7. Écrivez la fonction `eval_eexpr : int state -> expr -> int res` qui évalue une expression `e`, étant donné un état `s` (de type `int state`).

Question 6.8. Écrivez la fonction `eval_einstr (oc: formatter) (st: int state) (ins: instr): (int option * int state) res` qui exécute l’instruction `i` avec l’état `s`.

Si l’instruction à exécuter contient un `return`, une valeur de retour peut être retournée : c’est la première composante du résultat (`int option`).

L’état peut évoluer, notamment lors d’affectations : la seconde composante du résultat (`int state`) donne le nouvel état.

Finalement, tout ça peut échouer (lecture d’une variable non initialisée), d’où le type `res`.



Question 6.9. Appelez correctement `eval_einstr` dans la fonction `eval_eprog`, renvoyez la valeur finale du programme, ou une erreur si aucune valeur n'est renvoyée.

7 TP4 : Analyse de vivacité et élimination de code mort

L'objectif de cette séance est de programmer une optimisation pour notre compilateur : l'élimination des affectations mortes. Cette optimisation repose sur une analyse préalable du code du programme, qui sera facilitée par l'utilisation d'un langage intermédiaire approprié : CFG. Le langage CFG, utilise les mêmes expressions que le langage E, et un sous-ensemble des mêmes instructions. Les différences majeures sont les suivantes :

- un programme est un graphe de flot de contrôle (d'où le nom du langage : CFG pour Control Flow Graph) ;
- les instructions de branchement (IF et WHILE) sont encodées dans la structure du graphe directement.

Le langage CFG est décrit dans le fichier `src/cfg.ml`.

On y trouve notamment le type des expressions `expr`. Ce sont les mêmes expressions qu'en E. Cependant, lors des extensions de votre compilateur, les expressions E et les expressions CFG sont susceptibles d'évoluer différemment, c'est pourquoi nous avons préféré dupliquer ce type dès le début.

Une fonction CFG (type `cfg_fun`) est composée d'une liste d'arguments (`cfgfunargs`), d'une table de hachage des nœuds dont les clés sont les identifiants (entiers) des nœuds et la valeur associée est un `cfg_node`, et le point d'entrée d'un CFG est l'identifiant d'un nœud particulier du CFG. Les instructions sur chaque nœud du CFG sont du type `cfg_node` :

- `Cassign(v,e,s)` est une affectation `v := e`. `s` est l'identifiant du nœud successeur, *i.e.* à quel nœud doit-on sauter ensuite.
- `Creturn(e)` est un retour de fonction, en renvoyant la valeur de l'expression `e`. Cette instruction n'a pas de successeur.
- `Cprint(e,s)` affiche la valeur de l'expression `e`, puis continue l'exécution au nœud `s`.
- `Ccmp(e, s1, s2)` évalue l'expression `e`. Si sa valeur est vraie (pas zéro), alors l'exécution continue au nœud `s1` ; sinon au nœud `s2`.
- `Cnop s` : n'effectue aucune opération, puis saute au nœud `s`.

Les fonctions `succs` et `preds` donnent respectivement les successeurs et les prédécesseurs d'un nœud dans un CFG.

Nous vous fournissons le code pour la passe de transformation qui génère un programme CFG à partir d'un programme E (`cfg_gen.ml`), un interpréteur pour le langage CFG (`cfg_run.ml`) ainsi qu'un afficheur pour les programmes CFG (`cfg_print.ml`).

Vous pouvez utiliser l'afficheur en lançant votre compilateur comme suit :

```
$ ./main.native -f mon-fichier.e -cfg-dump mon-fichier.dot
$ dot mon-fichier.dot -Tpng -o mon-fichier.png
```

Vous pouvez utiliser l'interpréteur comme ceci :

```
$ ./main.native -f mon-fichier.e -cfg-run -- 4 12
```

La sortie du compilateur est un objet JSON qui donne des statistiques sur les différentes passes de compilation ("`compstep`") et exécutions ("`runstep`"). Dans notre cas, nous devrions trouver un champ "`runstep`":"CFG" qui contient un attribut "`retval`" (la valeur de retour du programme), un attribut "`output`" (la sortie du programme via la fonction `print`) et un attribut "`error`" qui contient les éventuelles erreurs survenues.

Lorsque vous lancez `make test`, la page HTML générée pour chaque fichier de tests contiendra une représentation graphique du programme CFG.

L'optimisation à laquelle on s'intéresse est **l'élimination des affectations mortes**. Cette optimisation permet de supprimer du programme les affectations `v := e` telles que la valeur de `v` n'est jamais lue après cette affectation. Il est aisé de comprendre que dans ce cas, cette affectation peut être supprimée sans modifier le comportement du programme.²

Pour déterminer quelles affectations peuvent être éliminées, nous allons procéder à une analyse de vivacité des variables.

7.1 Analyse de vivacité

Cette partie se passe dans le fichier `cfg_liveness.ml`. Le but de cette analyse est d'obtenir, pour chaque nœud du programme, l'ensemble des variables qui sont *vivantes* avant et après ce nœud. Il nous suffit en fait de calculer l'ensemble des variables vivantes **avant** chaque nœud ; on pourra calculer, au besoin, à partir de cela les variables vivantes après chaque nœud.

Pour stocker l'ensemble des variables vivantes avant chaque nœud, on utilisera une table de hachage indexée par des entiers (les identifiants des nœuds du CFG) : `(int, string Set.t) Hashtbl.t`.

L'analyse se déroulera en calculant le point fixe des équations de flot de données vues en cours, jusqu'à ce qu'une solution stable soit trouvée (*i.e.* jusqu'à ce qu'on n'ajoute plus de variable vivante à aucun point de programme).

Chaque itération de l'analyse mettra à jour l'état de l'analyse, *i.e.* le mapping entre les identifiant de nœuds et la liste des variables vivantes avant ce nœud. Pour chaque nœud, on commencera par calculer l'ensemble des variables vivantes après ce nœud : il s'agit de l'union des variables vivantes avant chacun de ces successeurs. À partir de cet ensemble, on calculera l'ensemble des variables vivantes avant ce nœud : les variables lues deviennent vivantes et les variables écrites deviennent mortes.

Question 7.1. Écrivez une fonction `vars_in_expr : expr -> string Set.t` qui renvoie l'ensemble des variables utilisées par une expression.

Nous vous fournissons, dans le fichier `cfg.ml`, les fonctions `succs : (int, cfg_node) Hashtbl.t -> int -> int Set.t` et `preds : (int, cfg_node) Hashtbl.t -> int -> int Set.t` qui renvoient respectivement les successeurs et prédécesseurs d'un nœud dans un CFG.

Question 7.2. Écrivez une fonction `live_after_node : (int, cfg_node) Hashtbl.t -> int -> (int, string Set.t) Hashtbl.t -> string Set.t`. Plus précisément, `live_after_node cfg n lives` calcule l'ensemble des variables vivantes après exécution du nœud `n` dans le graphe `cfg`, étant donné l'ensemble des variables vivantes avant chaque nœud donné par la table `lives`.

Question 7.3. Écrivez une fonction `live_cfg_node : cfg_node -> string Set.t -> string Set.t`. Par exemple, `live_cfg_node node live_after` doit renvoyer l'ensemble des

2. Attention, dans des langages plus riches que le langage E, comme C par exemple, cela n'est valable que si l'expression `e` n'a pas d'effets de bord.

variables vivantes avant le nœud `n`, où `liv_after` est l'ensemble des variables vivantes après ce nœud.

Question 7.4. Utilisez les fonctions précédentes pour écrire une fonction `live_cfg_nodes` (`cfg: (int, cfg_node) Hashtbl.t`) (`lives: (int, string Set.t) Hashtbl.t`) : `bool` qui effectue une itération du calcul de point fixe sur le CFG `cfg` en partant de l'état `lives`.

Cette fonction met à jour la table `lives` et renvoie un booléen qui indique si des changements ont eu lieu sur au moins un nœud.

La fonction `live_cfg_fun : cfg_fun -> (int, string Set.t) Hashtbl.t` est écrite pour vous. Elle calcule l'ensemble des variables vivantes avant chaque nœud du CFG correspondant à une fonction donnée.

7.2 Élimination de code mort

Cette partie se passe dans le fichier `cfg_dead_assign.ml`. Nous allons maintenant utiliser le résultat de notre analyse pour optimiser notre programme. Nous allons parcourir le graphe de flot de contrôle et pour chaque nœud correspondant à une affectation (`Cassign`), si la variable affectée n'est pas vivante après l'affectation, nous allons supprimer cette affectation (plus précisément transformer le nœud en un nouveau nœud de type `Cnop`).

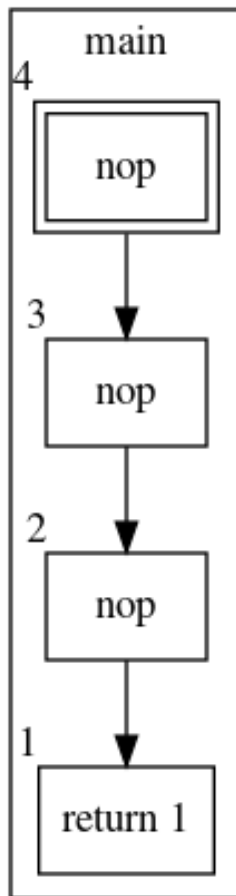
Question 7.5. Écrire une fonction `dead_assign_elimination_fun (f: cfg_fun) : cfg_fun` qui, étant donnée une fonction de CFG, calcule la vivacité des variables et élimine les affectations mortes.

Dans certains cas, le programme transformé peut encore être simplifié par l'application de cette même transformation. En effet, certaines variables pouvaient être vivantes seulement parce qu'elles étaient utilisées dans une affectation concernant une variable elle-même morte. C'est le cas par exemple de l'exemple `useless_assigns.e` présent dans votre répertoire `tests`.

Question 7.6. Modifiez votre code pour appliquer la transformation autant que nécessaire.

7.3 Bonus : élimination de NOPs

Dans la section précédente, on a transformé certains nœuds en `Cnop`, c'est-à-dire `no op` ou `no operation`. On s'intéresse maintenant à éliminer ces NOPs.



Par exemple, le CFG pour le programme `tests/basic/useless-assigns.e` ressemblera au CFG ci-contre.

Un certain nombre de fonctions à compléter sont présentes dans le fichier `src/cfg_nop_elim.ml`.

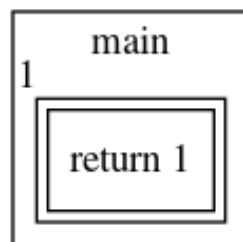
Sur l'exemple ci-contre :

```

— nop_transitions cfg = [(4,3); (3,2); (2,1)]
— follow 4 _ _ = 1
— follow 3 _ _ = 1
— follow 2 _ _ = 1
— follow 1 _ _ = 1
— nop_transitions_closed cfg = [(4,1); (3,1);
    (2,1); (1;1)]

```

Après application de l'optimisation, le CFG devrait ressembler à :



8 TP5 : Génération de programmes RTL

L'objectif de cette séance de TP est de générer des programmes RTL (*Register Transfer Language*). Comme expliqué dans la section 2, RTL est un langage dans lequel le programme est un graphe de flot de contrôle, comme en CFG, mais où les expressions, contrairement à CFG, sont décomposées en opérations élémentaires sur des registres. Il n'y a pas de limites au nombre de registres qu'un programme RTL utilise. En RTL, on perd la notion de variable : toutes les données sont dans des (pseudo-)registres.

Pourquoi choisit-on un tel langage intermédiaire ? Ce langage intermédiaire permet de se rapprocher de l'assembleur qui sera généré *in fine*, on dit qu'il est plus *bas niveau* que les langages précédents. C'est un choix courant dans des compilateurs connus : le compilateur GCC utilise un langage qui s'appelle aussi RTL et qui partage un certain nombre de caractéristiques avec notre langage RTL, la représentation intermédiaire IR de Clang y ressemble aussi, et CompCert utilise également un langage RTL.

La description du langage RTL se trouve dans le fichier `src/rtl.ml`. On y trouve notamment le type des registres `reg` (qui est un synonyme pour `int`).

Les instructions RTL (type `rtl_instr`) sont :

- Opération binaire `RBinop`(`b`, `rd`, `rs1`, `rs2`) : on effectue l'opération binaire `b` sur les registres *source* `rs1` et `rs2`, et on stocke le résultat dans le registre *destination* `rd`.
- Opération unaire `RUnop`(`u`, `rd`, `rs`) : on effectue l'opération unaire `u` sur le registre *source* `rs`, et on stocke le résultat dans le registre *destination* `rd`.
- Opération `Rconst`(`r`, `i`) : on stocke la constante `i` dans le registre `r`.
- Opération `Rmov`(`rd`, `rs`) : on copie la valeur contenue dans le registre *source* `rs` dans le registre *destination* `rd`.
- Opération « label » `Rlabel` `l` : on définit un *label* à cet endroit du code. On pourra y sauter par la suite, mais cette opération ne fait rien.
- Opération de branchement `Rbranch`(`cmp`, `rs1`, `rs2`, `l`) : on évalue la comparaison `cmp` (de type `rtl_cmp` défini plus haut) sur les registres `rs1` et `rs2`. Si la comparaison s'évalue en *vrai* (pas zéro), on saute au *label* `l` ; sinon on continue l'évaluation normalement.
- Opération de saut inconditionnel `Rjmp` `l` : on saute au *label* `l`.
- Opération `Rprint` `r` : affichage du contenu du registre `r`.
- Opération `Rret` `r` : on retourne la valeur contenue dans le registre `r`, et on arrête l'exécution de la fonction courante.

8.1 Génération de programmes RTL

Voici le plan d'attaque pour la transformation de programmes CFG en programmes RTL.

1. Tout d'abord, il nous faut un moyen de générer des nouveaux noms de registres. Les fonctions que nous allons écrire prendront donc en paramètre un couple (`next_reg`, `var2reg`), où :
 - `next_reg` est le numéro du prochain registre disponible (pas encore alloué à une variable, ou utilisé comme registre intermédiaire) ;
 - `var2reg` : (`string * int`) `list` est une liste d'association qui contient une paire (`v`, `r`) si le registre RTL `r` est associé à la variable CFG dont le nom est `v`.
2. Équipés de ces paramètres, il nous faudra transformer les expressions et instructions CFG en opérations RTL.

Le travail à effectuer se trouve dans le fichier `src/rtl_gen.ml`.

La fonction `find_var (next_reg, var2reg) v`, écrite pour vous, renvoie un triplet `(r, next_reg', var2reg')` où `r` correspond au registre correspondant à la variable `v`, et `next_reg'` et `var2reg'` sont les paramètres d'entrée mis à jour.

Passons maintenant à la compilation des expressions. La compilation d'une expression CFG va produire une liste d'opérations RTL qui vont avoir pour effet de calculer l'expression en question et de stocker sa valeur dans un registre. La fonction `rtl_instrs_of_cfg_expr (next_reg, var2reg) e` effectue cette transformation et doit renvoyer un quadruplet `(r, l, next_reg', var2reg')` tel que :

- le résultat de l'expression CFG `e` est stocké dans le registre `r` ;
- la liste d'instructions RTL correspondant au calcul de ce résultat est `l` ;
- `next_reg'` et `var2reg'` sont les variables `next_reg` et `var2reg` passées en paramètre, éventuellement mises à jour.

Par exemple, pour compiler l'expression `a + 2 * b`, en supposant que `next_reg` vaut 0 et `var2reg` est initialement vide :

- on commence par compiler la sous-expression `a`. On va donc appeler la fonction `find_var` qui nous renverra un triplet `(0, 1, [("a", 0)])`.

Le résultat de la compilation de cette première sous-expression donnera le quadruplet `(0, [], 1, [("a", 0)])`. (Aucune opération n'est requise.)

- on compile ensuite la sous-expression `2 * b`.

Tout d'abord, la compilation de la sous-expression `2` nous renverra le quadruplet :

`(1, [Rconst(1, 2)], 2, [("a", 0)])`.

Puis, la compilation de la sous-expression `b` nous renverra le quadruplet :

`(2, [], 3, [("a", 0); ("b", 2)])`.

Finalement, on obtiendra pour la sous-expression `2 * b` :

`(3, [Rconst(1,2); Rbinop(Emul, 3, 1, 2)], 4, [("a", 0); ("b", 2)])`.

- On combine ces deux sous-résultats : il nous faut un nouveau registre pour stocker le résultat de l'expression en entier :

`(4, [Rconst(1,2); Rbinop(Emul, 3, 1, 2); Rbinop(Eadd, 4, 1, 3)], 5, [("a", 0); ("b", 2)])`

Question 8.1. Écrivez la fonction `rtl_instrs_of_cfg_expr (next_reg, var2reg) e`.

Les expressions sont maintenant compilées, passons aux nœuds du CFG. Ceux-ci sont transformés en des nœuds RTL, qui partageront le même identifiant. Seulement, au lieu de contenir une « instruction » CFG, ils contiendront une liste d'opérations RTL.

Question 8.2. Écrivez la fonction `rtl_instrs_of_cfg_node (next_reg, var2reg) (c: cfg_node)` qui renvoie, étant donné un nœud CFG, un triplet `(l, next_reg', var2reg')` où `l` est la liste des opérations RTL correspondant à un nœud CFG `c` et `next_reg'` et `var2reg'` ont le même sens que précédemment.

Vous noterez la présence de la fonction `rtl_cmp_of_cfg_expr` : nous vous laissons inter-

prêter ce qu'elle fait et comment l'utiliser... Sans doute pour les opérations de branchement ?

La fonction suivante, `rtl_instrs_of_cfg_fun`, est écrite entièrement pour vous et ne fait qu'utiliser les fonctions précédentes pour construire une fonction RTL. Jetez-y un œil pour comprendre ce qui s'y passe.

Vous avez désormais des programmes RTL !

Un afficheur et un interpréteur de RTL vous sont fournis, à utiliser comme ci-dessous :

```
$ ./main.native -f tests/basic/gcd.e -rtl -
main(r0, r1):
main_2:
r2 <- 0
r1 != r2 ? jmp main_5
jmp main_1
main_3:
r0 <- r3
jmp main_2
main_5:
r3 <- r1
jmp main_4
main_4:
r4 <- %(r0, r1)
r1 <- r4
jmp main_3
main_1:
ret r0
$ ./main.native -f tests/basic/gcd.e -rtl-run -- 21 14
[
  { "compstep": "Lexing", "error": null, "data": [] },
  { "compstep": "Parsing", "error": null, "data": [] },
  # ...
  { "runstep": "RTL", "retval": 7, "output": "", "error": null },
  # ...
]
```

8.2 Linéarisation des programmes RTL

Les programmes RTL sont des graphes de flot de contrôle. La phase suivante est la linéarisation des programmes, c'est-à-dire transformer le graphe en un programme du langage Linear (définis dans le fichier `src/linear.ml`), qui ne sont qu'une liste d'instructions RTL, incluant des sauts explicites.

Une linéarisation naïve est implémentée dans le fichier `src/linear_gen.ml`. Cette linéarisation est naïve car les différents *blocs* d'instructions RTL sont mis les uns à la suite des autres sans considération du graphe de flot de contrôle, et donc sans doute que trop de sauts sont utilisés.

Vous avez donc une linéarisation qui fonctionne, mais qui pourrait être améliorée. Voici comment :

- on pourrait ordonner les nœuds du graphe de flot de contrôle original selon un ordre topologique, c'est-à-dire un parcours en profondeur du graphe.
- si deux blocs d'instructions sont ordonnées l'un juste après l'autre, on peut éviter un saut. Autrement dit, on n'a pas besoin de la suite d'instructions `[Rjmp 1; Rlabel 1]` (sauter à l'instruction suivante).



- après la transformation précédente, on peut éliminer les labels auxquels aucune instruction ne saute.

Cette section étant optionnelle, peu de détails sont fournis et il vous appartient d'explorer comment mettre en place les différentes optimisations présentées ci-dessus.

9 TP6 : Allocation de registres

Le but de cette séance de travaux pratiques est de réaliser l'allocation de registre, qui permettra de produire du code assembleur qui pourra être assemblé et exécuté par un processeur RISC-V.

L'allocation de registres consiste à spécifier, pour chaque pseudo-registre RTL (ou Linear), à quel emplacement matériel on va le stocker. Concrètement, on associe à chaque pseudo-registre RTL soit un registre matériel, soit un emplacement sur la pile.

Nous vous fournissons un « allocateur de registre » très simple qui alloue tous les pseudo-registres RTL sur la pile et n'utilise donc aucun registre matériel. C'est assurément inefficace puisque cela va générer un grand nombre de lectures/écritures dans la mémoire, mais ça a le mérite de fonctionner.

Vous avez donc déjà un compilateur qui génère des programmes RISC-V exécutables !

```
$ ./main.native -f tests/basic/gcd.e
[
  { "compstep": "Lexing", "error": null, "data": [] },
  { "compstep": "Parsing", "error": null, "data": [] },
  { "compstep": "CFG", "error": null, "data": [ { "main": 14 } ] },
  { "compstep": "CFG loops", "error": null, "data": [ { "main": 15 } ] },
  { "compstep": "Constprop", "error": null, "data": [ { "main": 15 } ] },
  { "compstep": "DeadAssign", "error": null, "data": [ { "main": 15 } ] },
  { "compstep": "NopElim", "error": null, "data": [ { "main": 14 } ] },
  { "compstep": "DSE", "error": null, "data": [] }
]
$ tests/basic/gcd.exe 54 24

6
$ $ ./main.native -f tests/basic/prime.e
[
  { "compstep": "Lexing", "error": null, "data": [] },
  { "compstep": "Parsing", "error": null, "data": [] },
  { "compstep": "CFG", "error": null, "data": [ { "main": 46 } ] },
  { "compstep": "CFG loops", "error": null, "data": [ { "main": 48 } ] },
  { "compstep": "Constprop", "error": null, "data": [ { "main": 48 } ] },
  { "compstep": "DeadAssign", "error": null, "data": [ { "main": 48 } ] },
  { "compstep": "NopElim", "error": null, "data": [ { "main": 46 } ] },
  { "compstep": "DSE", "error": null, "data": [] }
]
$ tests/basic/prime.exe 54
54
2
3
3
3
0
```

On sait calculer le PGCD de deux entiers et décomposer un entier en facteurs premiers !

De plus, le fichier HTML généré pour chacun des fichiers source (par exemple `tests/basic/gcd.e.html`) contient, dans la section **LTL**, l'allocation de chaque pseudo-registre RTL. Par exemple, pour le test `tests/basic/gcd.e`, on a l'allocation suivante :

```
// In function main
// LinReg 1 allocated to a1
// LinReg 0 allocated to a0
```

```
// LinReg 4 allocated to stk(-32)
// LinReg 3 allocated to stk(-24)
// LinReg 2 allocated to stk(-16)
```

Vous remarquerez que certains pseudo-registres (1 et 0) sont alloués dans des vrais registres machine (**a1** et **a0**) tandis que les autres (2, 3 et 4) sont alloués sur la pile, aux emplacements -16, -24 et -32.

L'objectif de la suite de cette séance est donc d'écrire un allocateur de registres plus intelligent, qui utilisera mieux les registres disponibles en RISC-V.

9.1 Crash course : assembleur RISC-V

RISC-V est une architecture de jeu d'instruction (ISA) ouverte et libre. C'est une architecture de type RISC (*Reduced Instruction Set Computer*) avec relativement peu d'instructions et un plus grand nombre de registres (comparé à x86 par exemple qui est de type CISC (*Complex Instruction Set Computer*) et qui a énormément d'instructions et peu de registres). Nous allons présenter ici un sous-ensemble des instructions RISC-V – celles dont on a besoin pour écrire notre compilateur. À noter que nous vous fournissons le code pour la génération de l'assembleur RISC-V (dans `src/riscv.ml`). Les descriptions qui suivent vous serviront donc pour votre culture, ainsi que pour comprendre le code qui vous est fourni lorsque vous devrez l'étendre.

9.1.1 Registres RISC-V

RISC-V existe en 32 bits ou 64 bits (et même 128 bits). Nous allons écrire du code pour RISC-V 32 et 64. Rassurez-vous, la plupart du code peut se factoriser aisément. Dans cette architecture, RISC-V donne accès à 32 registres de 32 ou 64 bits chacun (selon l'architecture), dont voici la liste :

Numéro	Nom	Commentaire	Numéro	Nom	Commentaire
x0	zero	vaut toujours zéro	x16	a6	arguments
x1	ra	adresse de retour	x17	a7	
x2	sp	pointeur de pile	x18	s2	registres <i>callee-save</i>
x3	gp	<i>global pointer</i>	x19	s3	
x4	tp	<i>thread pointer</i>	x20	s4	
x5	t0	temporaires	x21	s5	
x6	t1		x22	s6	
x7	t2		x23	s7	
x8	s0/fp	<i>frame pointer</i>	x24	s8	
x9	s1	registre <i>callee-save</i>	x25	s9	
x10	a0	argument / valeur de retour	x26	s10	
x11	a1	arguments	x27	s11	
x12	a2		x28	t3	temporaires
x13	a3		x29	t4	
x14	a4		x30	t5	
x15	a5		x31	t6	

Le registre **zero** contient toujours la valeur 0.

Le registre **ra** est utilisé pour stocker l'adresse de retour lors d'un appel de fonction.³

3. Notons que ce n'est qu'une convention et que l'on pourrait stocker l'adresse de retour dans n'importe quel registre.

Le registre **sp** contient le pointeur de pile (similaire à **esp** en x86).

Le registre **gp** est le *global pointer*. Il contient l'adresse de la zone de mémoire où sont stockées les variables globales. Nous ne l'utiliserons pas dans notre compilateur.

Le registre **tp** est le *thread pointer*. Il contient l'adresse de la zone de mémoire propre au thread courant. Nous ne l'utiliserons pas dans notre compilateur.

Les registres **t0** à **t6** sont des registres temporaires qui n'ont pas de vocation particulière.

Le registre **s0**, aussi appelé **fp** pour *frame pointer*, contient l'adresse du début de la trame de pile de la fonction courante. C'est l'analogue du registre **ebp** en x86.

Les registres **s1** à **s11** sont des registres temporaires sans vocation particulière.

Les registres **a0** à **a7** sont utilisés pour passer les arguments lors d'un appel de fonction (premier argument dans **a0**, second dans **a1**...). De plus, la valeur de retour des fonctions est passée dans le registre **a0**.

Registres caller- et callee-save. Les registres **ra**, **t0-t6** et **a0-a7** sont *caller-save*, c'est-à-dire que c'est la responsabilité de la fonction appelante (le *caller*) de sauvegarder ces registres avant de faire un appel de fonction si leur valeur est importante.

Les registres **sp**, **s0-s11** sont *callee-save*, c'est-à-dire que c'est la responsabilité de la fonction appelée (le *callee*) de sauvegarder ces registres si la fonction appelée les modifie. On peut donc faire l'hypothèse, lorsqu'on appelle une fonction, que la valeur de ces registres sera la même après l'appel.

9.1.2 Instructions.

Contrairement à x86, où les instructions acceptent plusieurs modes d'adressage pour les opérandes (registres, valeurs immédiates, emplacements mémoire), les instructions RISC-V ont en général un unique mode d'adressage.

La plupart des instructions sont de type « 3 adresses » et attendent des registres comme opérandes. Deux schémas principaux émergent :

- **op rd, rs1, rs2** où **op** est une opération (**add**, **mul**, **div**, **remu** (pour *remainder unsigned*, a.k.a. modulo)...), **rd** est le registre de destination, et **rs1** et **rs2** sont les registres source.
- **op rd, rs** où **op** est une opération (**neg**, ...), **rd** est le registre de destination, et **rs** est le registre source.

Pour compiler les conditionnelles (par exemple **r1 <= r2 <= r3**), on a besoin des instructions supplémentaires suivantes :

- **slt rd, rs1, rs2** : **rd** vaut 1 si **rs1 < rs2**, 0 sinon
- **seqz rd, rs** : **rd** vaut 1 si **rs** vaut zéro, 0 sinon
- **snez rd, rs** : **rd** vaut 0 si **rs** vaut zéro, 1 sinon

Les instructions de branchement.

- Saut inconditionnel : **j <label>**
- Appel de fonction : **jal ra, <label>**
- Saut si deux registres sont égaux : **beq rs1, rs2, <label>**
- Saut si **rs1 < rs2** (en *unsigned* ou pas) : **blt[u] rs1, rs2, <label>**
- Saut si **rs1 >= rs2** (en *unsigned* ou pas) : **bge[u] rs1, rs2, <label>**
- Saut si **rs != 0** : **bnez rs, <label>**

Accès à la mémoire

- Lire depuis la mémoire vers un registre : `ld rd, ofs(rs)`. Cette instruction lit la mémoire à l'adresse `rs + ofs` et stocke le résultat dans le registre `rd`.
`ld` veut dire *load double-word*, i.e. 64 bits. Pour ne lire que 32 bits, on aurait utilisé `lw` pour *load word*; pour 16 bits `lh` pour *load half-word* et pour 8 bits `lb` pour *load byte*.
- Écrire la valeur d'un registre dans la mémoire : `sd rs, ofs(rd)`. Cette instruction écrit la valeur du registre `rs` dans la mémoire à l'adresse `rd + ofs`.
`sd` veut dire *store double-word*, i.e. 64 bits. Pour n'écrire que 32 bits, on aurait utilisé `sw` pour *store word*; pour 16 bits `sh` pour *store half-word* et pour 8 bits `sb` pour *store byte*.

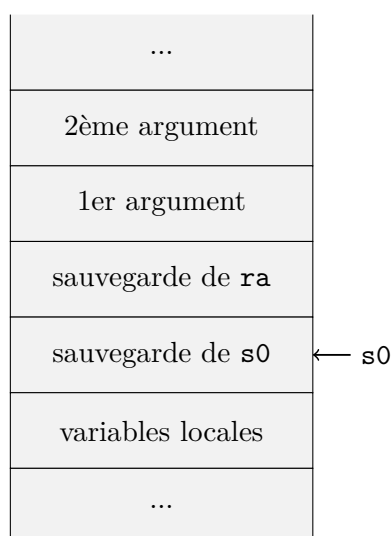
Valeurs immédiates Pour écrire une constante dans un registre : `li rd, imm`, où `rd` est le registre de destination et `imm` est la constante que l'on souhaite stocker.

9.1.3 Code fourni

Nous vous fournissons un allocateur de registres « naïf », dans le fichier `src/regalloc.ml`. Plus précisément, l'allocateur de registres `regalloc_on_stack_fun` prend en entrée une fonction `linear_fun` et renvoie un couple `((reg, loc) Hashtbl.t * int)`. Le premier élément du couple est une table de hachage qui associe à chaque pseudo-registre un emplacement (*location* en anglais) de type `loc` définie comme suit :

```
type loc = Reg of int | Stk of int
```

Un emplacement est donc soit un registre physique du processeur cible (ici le processeur RISC-V), soit un slot sur la pile. Un tel slot est un décalage par rapport au registre `fp` (*frame pointer* ou cadre de pile). Notez que puisque sur RISC-V (comme sur x86 d'ailleurs) le sommet de pile (le registre `sp`) décroît vers des adresses plus petites lorsque l'on empile des données sur celle-ci, le décalage entre une variable locale évincée sur la pile et le registre de cadre de pile `fp` est toujours un entier négatif, comme cela apparaît sur la figure :



L'allocateur fourni fonctionne de manière très simple en parcourant l'ensemble des instructions d'une fonction en langage Linear afin de déterminer les pseudo-registres qu'elle utilise. Pour cela, on fait appel aux fonctions `gen_live` et `kill_live` qui vous sont fournies dans le fichier

`src/linear_liveness.ml` dont le but est précisément de calculer la vivacité des pseudo-registres en langage Linear. On ne vous demande pas de coder de nouveau cette analyse de vivacité puisqu'elle a déjà fait l'objet d'un TP. En faisant l'union de tous les pseudo-registres utilisés par une fonction, il reste plus qu'à itérer sur la liste de ces registres pour allouer à chacun un slot (un entier négatif) qui décroît de 1 pour chacun d'eux (puisque pour le moment toutes les variables sont de type entier de longueur égale à celle de l'architecture sous-jacente).

9.2 Allocation de registres

Bien qu'on ait à présent un compilateur complet (dans le sens où on génère des fichiers exécutables), on peut apporter un grand nombre d'améliorations à notre compilateur. Dans cette partie, nous allons écrire un allocateur de registres afin de profiter de la multitude de registres disponibles en RISC-V ; et minimiser le nombre d'instructions qui accèdent à la mémoire.

Nous allons écrire un algorithme basé sur la coloration de graphes, comme vu en cours. Pour cela il est nécessaire de procéder au préalable à une analyse de vivacité des pseudo-registres utilisés dans le langage Linear. Comme expliqué plus tôt, cette analyse vous est fournie (fichier `src/linear_liveness.ml`).

Lorsque vous lancez un test sur un exemple, vous pouvez consulter le résultat de cette phase d'analyse de vivacité dans le fichier `.linear` associé au code compilé :

```
$ ./test.py -f basic/gcd.e
1 threads running (['basic/gcd.e']).
All threads terminated!
1/1 OK.
0/1 KO : []
$ cat basic/gcd.linear
main(r0, r1):
// Live before : { 0, 1 }
main_2:
// Live after : { 0, 1 }
// Live before : { 0, 1 }
r2 <- 0
// Live after : { 0, 1, 2 }
// Live before : { 0, 1, 2 }
r1 != r2 ? jmp main_5
// Live after : { 0, 1 }
// Live before : { 0 }
jmp main_1
// Live after : { 0 }
// Live before : { 0, 1 }
main_5:
// Live after : { 0, 1 }
// Live before : { 0, 1 }
r3 <- r1
// Live after : { 0, 1, 3 }
// Live before : { 0, 1, 3 }
r4 <- %(r0, r1)
// Live after : { 3, 4 }
// Live before : { 3, 4 }
r1 <- r4
// Live after : { 1, 3 }
// Live before : { 1, 3 }
r0 <- r3
// Live after : { 0, 1 }
```

```
// Live before : { 0, 1 }
jmp main_2
// Live after : { 0, 1 }
// Live before : { 0 }
main_1:
// Live after : { 0 }
// Live before : { 0 }
ret r0
// Live after : { }
```

À partir de cette analyse, nous allons construire un graphe d'interférences : les sommets sont les différents pseudo-registres utilisés dans le programme et on a une arête entre deux pseudo-registres si ceux-ci sont vivants en même temps. Intuitivement, deux pseudo-registres vivants en même temps ne pourront pas être associés au même registre machine. Le problème d'allocation de registre devient alors un problème de coloration de graphes : on souhaite associer une couleur à chaque sommet de sorte que deux sommets voisins dans le graphe sont associés à des couleurs différentes.

9.2.1 Allocation de registres par coloration de graphes

Passons maintenant à la construction du graphe d'interférence et à sa coloration. Cela se passe dans le fichier `src/regalloc.c`.

Construction du graphe d'interférence. La construction du graphe d'interférence sera effectuée par la fonction `build_interference_graph` (`live_out` : (`int`, `reg Set.t`) `Hashtbl.t`) : (`reg`, `reg Set.t`) `Hashtbl.t` qui renvoie le graphe d'interférence à partir d'une analyse de vivacité. L'analyse de vivacité est passée sous la forme d'une table de hachage qui associe à chaque instruction du code en langage Linear (identifiée par un entier : sa position dans le programme) un ensemble de registres vivants à la sortie de cette instruction. Cette fonction retourne un graphe d'interférence sous la forme d'une table de hachage qui associe un arc (dirigé) entre chaque registre et les registres avec lequel ce registre est en interférence et donc adjacent dans le graphe.

C'est à vous de construire le graphe, à partir du résultat de vivacité. Vous devez ajouter des arêtes entre deux sommets si à au moins un point de programme, les pseudo-registres associés à ces deux sommets sont vivants en même temps.

Question 9.1. Écrivez une fonction `add_interf` (`rig` : (`reg`, `reg Set.t`) `Hashtbl.t`) (`x`: `reg`) (`y`: `reg`) : `unit` qui met à jour le graphe d'interférence `rig` (pour *register interference graph*) en ajoutant un lien entre les sommets `x` et `y`.

Attention, étant donnée la représentation que l'on a choisi pour le graphe (sous forme de listes d'adjacence), ajouter une arête implique d'ajouter `x` aux voisins de `y` et `y` aux voisins de `x`.

Question 9.2. Écrivez une fonction `make_interf_live` (`rig`: (`reg`, `reg Set.t`) `Hashtbl.t`) (`live` : (`int`, `reg Set.t`) `Hashtbl.t`) : `unit` qui construit le graphe d'interférence.

On vous fournit la fonction `build_interference_graph` (`live_out` : (`int`, `reg Set.t`) `Hashtbl.t`) : (`reg`, `reg Set.t`) `Hashtbl.t` qui s'assure d'initialiser correctement le graphe

d'interférences avec autant de sommets qu'il y a de pseudo-registres utilisé dans le programme Linear. Cette fonction utilise les fonctions que vous avez définies précédemment.

Coloration du graphe L'algorithme d'allocation de registres par coloration de graphes a été initialement proposé dans un article de recherche par Chaitin *et al.* en 1981⁴.

On suppose que l'on dispose de N couleurs (registres machine). L'algorithme se déroule en plusieurs phases :

1. On commence par construire une pile de pseudo-registres en indiquant pour chacun si on va être capable d'y associer une couleur (un registre machine) à coup sûr, ou non (dans ce cas, il devra être alloué sur la pile – on dit qu'il est évincé ou *spilled*).

La construction de cette pile s'effectue comme suit :

Tant que le graphe `rig` n'est pas vide :

- on essaie de trouver un sommet avec strictement moins de N voisins dans `rig`
 - Si un tel sommet `r` existe, retirons-le du graphe `rig` et ajoutons ce registre au sommet de la pile en indiquant qu'on saura lui associer une couleur. Dans notre code on met sur la pile `NoSpill r`.
 - Sinon, on va devoir évincer (*spiller*) un pseudo-registre, c'est-à-dire choisir un pseudo-registre qui sera alloué sur la pile. On peut choisir n'importe quel sommet, mais une heuristique particulière consiste à choisir le sommet qui a le plus de voisins : cela aura pour effet de diminuer le nombre de voisins de beaucoup d'autres registres et de permettre de trouver plus facilement par la suite un sommet avec moins de N voisins. On retire ce sommet du graphe, et on l'ajoute au sommet de la pile en indiquant qu'il sera évincé. Dans notre code on met sur la pile `Spill r`.

2. Une fois cette pile construite, on va effectivement associer des couleurs à chaque sommet marqué `NoSpill` et des emplacements sur la pile à chaque sommet marqué `Spill`.

Question 9.3. Écrivez une fonction `remove_from_rig (rig : (reg, reg Set.t) Hashtbl.t) (v: reg) : unit` qui supprime un sommet `v` du graphe d'interférence `rig` passé en paramètre.

Attention, étant donnée la représentation que l'on a choisi pour le graphe (sous forme de listes d'adjacence), supprimer un sommet nécessite de parcourir tous les éléments de la table de hachage pour retirer toute référence à ce sommet.

Question 9.4. Écrivez une fonction `pick_node_with_fewer_than_n_neighbors (rig : (reg, reg Set.t) Hashtbl.t) (n: int) : reg option` qui renvoie `Some(r)`, avec `r` un registre de `rig` possédant moins de `n` voisins quand c'est possible, et `None` sinon.

Question 9.5. Écrivez une fonction `pick_spilling_candidate (rig : (reg, reg Set.t) Hashtbl.t) : reg option` qui renvoie un sommet que l'on va évincer. Choisissez le sommet avec le maximum de voisins. Comme pour la fonction précédente, renvoyez `Some(r)` lorsque le

4. Chaitin, Gregory J. ; Auslander, Marc A. ; Chandra, Ashok K. ; Cocke, John ; Hopkins, Martin E. ; Markstein, Peter W. (1981). "Register allocation via coloring".

graphe n'est pas vide ou `None` dans le cas contraire.

On vous fournit le type `regalloc_decision = Spill of reg | NoSpill of reg`

Question 9.6. Écrivez la fonction récursive `make_stack` (`rig : (reg, reg Set.t) Hashtbl.t`) (`stack : regalloc_decision list`) (`ncolors: int`) : `regalloc_decision list` qui à partir d'un graphe d'interférence, d'une pile de décision (déjà partiellement construite et éventuellement vide), d'un nombre de registres disponibles étend la pile passée en paramètre comme décrit plus haut.

Question 9.7. Écrivez la fonction `allocate` (`allocation: (reg, loc) Hashtbl.t`) (`rig: (reg, reg Set.t) Hashtbl.t`) (`all_colors: int Set.t`) (`next_stack_slot: int`) (`decision: regalloc_decision`) : `int` qui étend une allocation (éventuellement déjà partiellement construite) à partir de :

- `alloc` : une allocation partielle;
- `rig` : le graphe d'interférence, utile pour connaître les voisins d'un sommet;
- `all_colors` : un ensemble de registres physiques disponibles (les couleurs);
- `next_stack_slot` : un entier (négatif ou nul comme expliqué plus haut) représentant le prochain slot à utiliser sur la pile (si besoin);
- `decision` : une décision (de type `regalloc_decision`) stockée sur la pile de décision

Cette fonction renvoie le prochain emplacement sur la pile qu'il faudra utiliser dans le futur. Cela signifie qu'elle peut décrémenter la valeur `next_stack_slot` qu'on lui a passé en paramètre.

Les fonctions `make_stack` et `allocate` sont appelées par la fonction

```
let regalloc_fun (f: linear_fun)
  (live_out: (int, reg Set.t) Hashtbl.t)
  (all_colors: int Set.t)
: (reg, reg Set.t) Hashtbl.t      (* le graphe d'interférences *)
* (reg, loc) Hashtbl.t           (* l'allocation *)
* int                           (* le prochain slot disponible sur la pile *)
= ...
```

qui vous est donnée déjà complétée. Cette fonction déroule l'allocation de registres pour une fonction d'un programme Linear. Elle renvoie un triplet contenant :

1. le graphe d'interférence qui peut être visualisé à partir de sa représentation en fichier `.dot`.
2. une table de hachage décrivant une association entre chaque pseudo-registre du programme en Linear et un emplacement;
3. la prochaine position libre sur la pile.

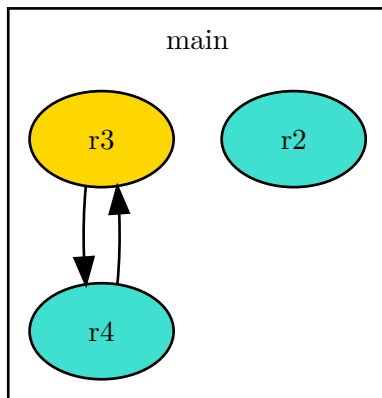
Cette fonction est elle-même appelée par la fonction `regalloc lp lives all_colors` qui fait ce même travail mais sur un programme Linear complet (éventuellement composé de plusieurs fonctions).

```
$ ./main.native -f tests/basic/gcd.e -clever-regalloc -ltl-dump -
// In function main
// LinReg 1 allocated to a1
// LinReg 4 allocated to t2
// LinReg 3 allocated to s1
// LinReg 0 allocated to a0
// LinReg 2 allocated to t2
[...]
```

Dans la commande précédente, on a passé un certain nombre d'options :

- `-clever-regalloc` pour utiliser l'allocateur de registres que l'on vient de programmer.
- `-ltl-dump -` pour afficher le programme LTL (pour comprendre à quoi correspondent les pseudo-registres)

Le graphe d'interférence est visible dans le fichier DOT `tests/basic/gcd.rig` et déjà converti en SVG pour vous dans `tests/basic/gcd.rig.svg` :



On y voit que les registres `r2` et `r4` sont alloués dans le même registre physique (`t2`, d'après le dump LTL précédent).

Les registres `r3` et `r4` sont dans des registres différents, parce qu'ils sont en interférence. Effectivement, on avait vu dans le dump Linear, quelques pages plus haut, que ces registres étaient vivants ensemble à un point de programme.

On ne voit pas les registres `r0` et `r1` dans ce graphe, qui correspondent aux arguments de la fonction. Effectivement, les arguments sont alloués dans des registres séparés (`a0` à `a7`, puis sur la pile si nécessaire, mais à des offsets positifs par rapport à `fp`, depuis la fonction appelée), leur allocation sera faite automatiquement pour vous dans `src/ltl_gen.ml`.

9.3 Tester

Pour lancer les tests, vous avez utilisé depuis le début de ces séances de TP la commande `make test`. Mais vous ne connaissez pas toute la puissance qui se cache derrière cette commande !

Par défaut, `make test` lance le script Python `tests/test.py` sur l'ensemble des fichiers `*.e` qui se trouvent dans le répertoire `tests/basic`. Aussi, par défaut, l'allocateur de registres « naïf » est utilisé.

Vous pouvez modifier ce comportement par défaut en passant des paramètres à votre commande `make`.

Pour passer des paramètres supplémentaires au compilateur (*i.e.* au `./main.native`), on passe ces paramètres dans la variable `OPTS` de la manière suivante :

```
# passer l'option -clever-regalloc au compilateur
$ make test OPTS=-clever-regalloc
# ...
./test.py -f basic/*.e -clever-regalloc
37 threads running (['basic/1_or_3a.e', 'basic/1_or_4.e', 'basic/2xpy.e', 'basic/35gt12.e',
↳ 'basic/a_19.e'])
# ...
```

Pour lancer les tests sur un ensemble de fichiers source, on passe ces paramètres dans la variable `DIR` de la manière suivante :

```
$ make test DIR="basic/prime.e basic/gcd.e"
# ...
./test.py -f basic/prime.e basic/gcd.e
2 threads running (['basic/prime.e', 'basic/gcd.e']).
All threads terminated!
2/2 OK.
0/2 KO : []
```

Vous pouvez évidemment combiner ces deux options :

```
$ make test OPTS=-clever-regalloc DIR="basic/prime.e basic/gcd.e"
# ...
./test.py -f basic/prime.e basic/gcd.e -clever-regalloc
# ...
```

10 Appels de fonctions

Le langage de votre compilateur est pour l'instant quelque peu limité. Notamment, on ne sait pas faire d'appels de fonctions. C'est la prochaine étape pour améliorer votre compilateur. Vous pourrez alors compiler des programmes qui ressemblent à celui-ci :

```
fib(n){
    if(n > 14){ return -1; }
    if(n <= 2){
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
main(n){
    return fib(n);
}
```

Comme vous le constatez, il s'agit d'une implémentation de la suite de Fibonacci de manière récursive et inefficace (on calcule plusieurs fois les mêmes valeurs...). Pour que les calculs ne durent pas trop longtemps, on abandonne et on renvoie -1 pour une entrée supérieure à 14.

Notons que pour le moment, nous n'avons toujours pas de type : toutes les variables sont entières. Le programme ci-dessus ainsi que 10 autres vous sont fournis dans le répertoire `funcall`.

Pour étendre votre compilateur, il va vous falloir modifier votre grammaire, ainsi que chacun des langages intermédiaires et des différentes passes de compilation. Voici un aperçu des changements à effectuer :

10.1 La grammaire

Une instruction peut maintenant être un appel de fonction (en plus des cas existants d'affectation, de boucles, ...) :

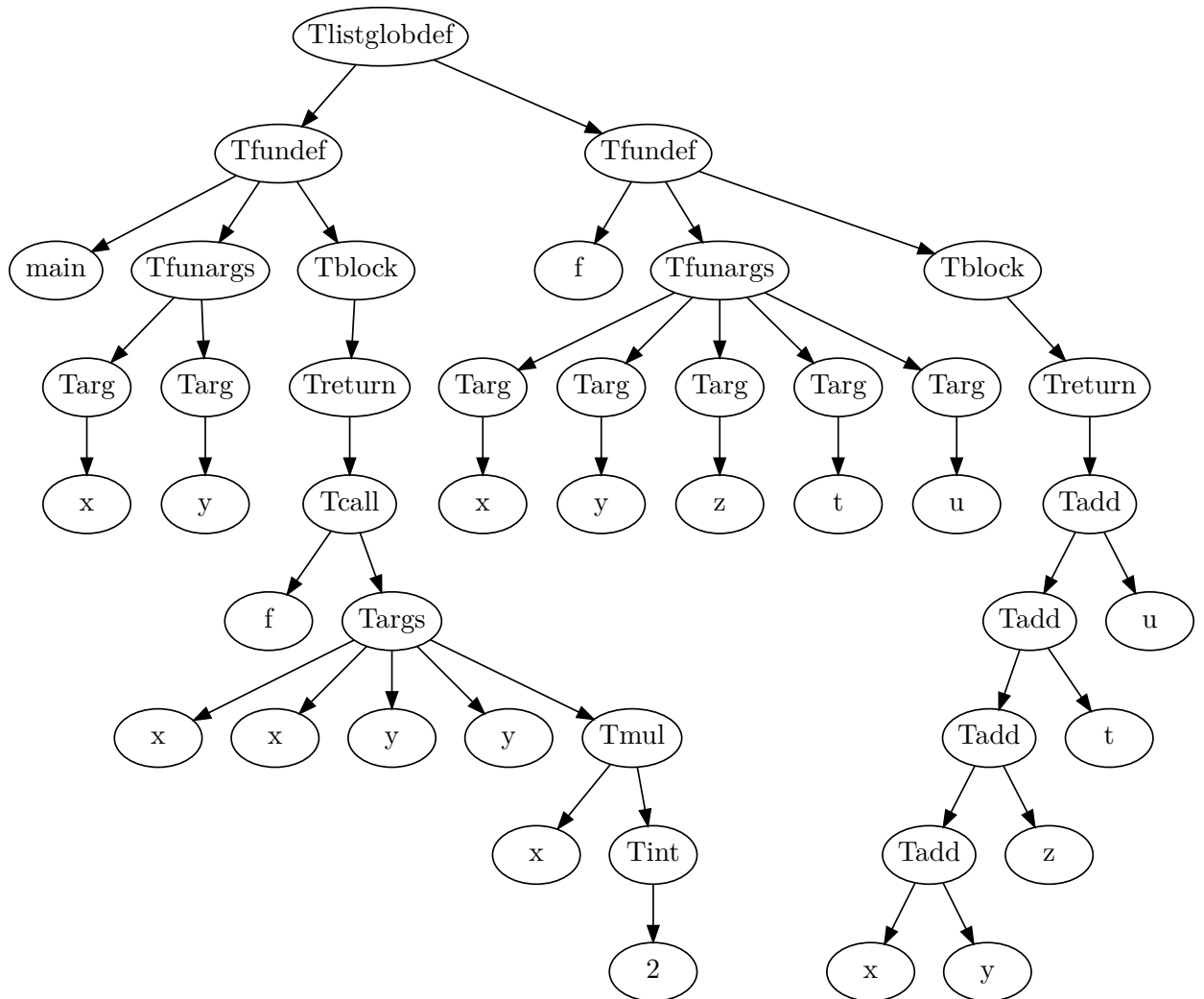
```
x = 3;
y = 8;
f(x,y); // Un appel de fonction, en tant qu'instruction.
z = x + y;
```

Une **expression** peut aussi contenir des appels de fonctions :

```
x = 3;
y = 8;
z = 5 * f(x,y) + 16; // Un appel de fonction, en tant qu'expression.
```

Étendez votre grammaire afin de reconnaître ces nouvelles constructions.

Écrivez également les actions correspondant à ces nouvelles constructions, de manière à obtenir un AST ressemblant à celui-ci :



Le programme source générant cet AST est le suivant (`tests/funcall/5args.e`) :

```
main(x, y){
    return f(x,x,y,y,x*2);
}
f(x, y, z, t, u){
    return x + y + z + t + u;
}
```

Vous pouvez bien entendu ajouter au fichier `src/ast.ml` les nouveaux tags qui vous semblent nécessaires (`Tcall`, `Targs`...).

Conseils pour modifier la grammaire Il est important de vérifier que votre grammaire ne comporte pas de conflit, à chaque fois que vous la modifiez. Pour ce faire, deux solutions :

1. Lorsque vous lancez la commande `make`, la grammaire est recompilée et un message d'avertissement sera émis si un conflit est présent.

Warning! There is a conflict in your grammar. Check the prediction table for more details.

2. Comme le message d'avertissement vous le suggère, allez voir la table de prédiction pour identifier le conflit. Par défaut, la table est générée dans le fichier `grammar.html` à la racine de votre projet. Dans le dernier tableau du fichier, une case sur rouge indique un conflit. Cela signifie que l'analyseur syntaxique ne sait pas choisir la règle à appliquer pour continuer l'analyse.

Pour résoudre ces conflits, il faut notamment s'assurer qu'on n'a pas deux règles qui peuvent commencer par le même terminal. Il faut donc factoriser ces règles.

```
A -> X A { Node (Ta, [$1; $2]) }
A -> X B { Node (Tb, [$1; $2]) }
```

devient alors

```
A -> X A_ou_B
A_ou_B -> A
A_ou_B -> B
```

Mais cela devient embêtant pour écrire les actions... On peut s'en sortir de différentes manières.

Solution 1 : définir des types Dans l'entête de la grammaire (avant le mot-clé `rules` et entre les accolades, où vous avez déjà dû écrire la fonction `resolve_associativity`), vous pouvez définir un type spécifique à votre parseur :

```
{
  type mon_super_type =
  | MonSuperTypeA of tree
  | MonSuperTypeB of tree
}
...
A -> X A_ou_B {
  match $2 with
  | MonSuperTypeA t -> Node (Ta, [$1; t])
  | MonSuperTypeB t -> Node (Tb, [$1; t])
}
A_ou_B -> A { MonSuperTypeA $1 }
A_ou_B -> B { MonSuperTypeA $1 }
```

Avantages ça fonctionne aussi si on utilise `A_ou_B` à plusieurs endroits dans la grammaire

Inconvénients c'est un peu lourd de définir de nouveaux types et de faire un `match...`

Solution 2 : renvoyer des fonctions Les actions peuvent être de n'importe quel type, y compris des fonctions!

On peut réécrire l'exemple précédent comme suit :

```
A -> X A_ou_B { $2 $1 }
A_ou_B -> A { fun x -> Node (Ta, [x; $1] }
A_ou_B -> B { fun x -> Node (Tb, [x; $1] }
```

Dans l'action de la première règle, on appelle la fonction renvoyée par le non-terminal `A_ou_B` avec, en argument, l'élément renvoyé par le non-terminal `X`.

Avantages plus élégant et plus court que la solution précédente

Inconvénients comme on a écrit toute la logique de l'action dans `A_ou_B`, on ne peut pas (à moins de passer d'autres arguments) utiliser ce non-terminal à plusieurs endroits, dans des contextes un peu différents.

10.2 Le langage E

Étendez les types des expressions et des instructions `E` de la manière suivante, dans `src/elang.ml`.

```
type expr =  
...  
| Ecall of string * expr list  
  
type instr =  
...  
| Icall of string * expr list
```

Une expression `Ecall(fname, args)` représente un appel de la fonction nommée `fname` avec les arguments représentés par la liste d'expressions `args`. La valeur de cette expression est la valeur de retour de la fonction `fname`. Si la fonction ne renvoie pas de valeur, il s'agit d'une erreur.

De manière similaire, une instruction `Icall(fname, args)` représente un appel de la fonction nommée `fname` avec les arguments représentés par la liste d'expressions `args`. La valeur de retour de la fonction est ignorée.

10.2.1 Génération de E

Dans le fichier `src/elang_gen.ml`, il vous faut maintenant traiter les nouveaux types de nœuds (`Tcall`, dans la figure précédente) introduits à la section précédente, afin de produire des expressions `Ecall` et des instructions `Icall`.

10.2.2 Interpréteur de E

Maintenant qu'une expression peut contenir des appels de fonctions, évaluer une expression peut nécessiter d'exécuter une fonction. Pour ce faire, vous allez avoir besoin d'écrire des définitions mutuellement récursives, c'est-à-dire que le corps de la fonction `eval_eexpr` pourra faire appel à la fonction `eval_efun`, qui appelle elle-même la fonction `eval_einstr`, qui appelle elle-même la fonction `eval_eexpr`...

La syntaxe OCaml pour écrire des définitions mutuellement récursive est la suivante :

```
let rec eval_eexpr st e = ...  
  
and eval_einstr ... = ...  
  
and eval_efun ... = ...
```

Il vous faudra sans doute ajouter des arguments à ces fonctions, notamment `eval_eexpr`.

À noter : pour le moment, les fonctions ne modifient pas la mémoire – les seuls effets de bords que l'on peut observer sont des affichages à l'écran. Pour autant, en prévision de l'avenir, on supposera que la mémoire peut être modifiée et on prendra donc garde à propager l'effet des appels de fonction sur l'état du programme. Plus concrètement, si vous devez évaluer une liste `[e1; e2]` d'expressions (par exemple pour évaluer les arguments d'une fonction), gardez bien en tête que l'évaluation de `e1` pourrait modifier l'état et donc influencer l'évaluation de `e2`. Le type de retour de la fonction `eval_eexpr` pourrait donc être étendu...

10.3 Le langage CFG

Comme pour le langage E, on ajoute un nouveau type d'expression `Ecall(fname, args)` et un nouveau type de nœud `Ccall(fname, args, s)` où `s` est le successeur de ce nœud, comme dans `Cassign`.

Les changements à effectuer dans `src/cfg_run.ml` sont similaires à ceux que vous aurez effectués dans `src/elang_run.ml`.

Comme vos types de données ont changé, le compilateur va se plaindre que vos `match` ne sont pas exhaustifs. Contentez le compilateur en complétant ces `match`.

10.4 Le langage RTL

On introduit une opération `Rcall of reg option * string * reg list`.

L'opération `Rcall(ord, fname, rargs)` signifie qu'on appelle la fonction `fname` avec les arguments stockés dans les pseudo-registres `rargs`. Si `ord = Some rd`, le résultat de la fonction sera stocké dans le registre `rd`, sinon, si `ord = None`, le résultat de la fonction sera ignoré. Si `ord = Some rd` mais que la fonction ne retourne pas de valeur, il s'agit d'une erreur.

10.5 Le langage LTL

La plus grande difficulté pour traiter les appels de fonction se situe dans l'émission de code assembleur (ou LTL, c'est presque la même chose).

En effet, à cet endroit, on doit être en mesure d'appeler une fonction (via l'instruction `Lcall(fname)`), déjà présente dans `src/ltl.ml`. Mais il faut aussi, et surtout émettre le code assembleur qui :

1. sauvegarde les registres dits *caller-save* (*i.e.* ceux qui doivent être sauvegardés par l'appelant) : en effet, la fonction appelée va peut-être (sûrement) utiliser ces registres pour faire un calcul, et leur valeur dans la fonction appelante serait perdue ;
2. fournit les arguments à la fonction ;
3. effectue l'appel à proprement parler (instruction `LCall`) ;
4. récupère la valeur de retour de la fonction (stockée par convention dans le registre `a0`) à l'emplacement attendu (tel que calculé par l'allocation de registres) ;
5. restaure la valeur des registres *caller-save*.

Les modifications que vous devez effectuer se situent dans le fichier `src/ltl_gen.ml` dans la fonction `ltl_instrs_of_linear_instr`, où vous devez traiter le cas où l'instruction Linear que vous devez traduire est `Rcall(rd, callee_fname, rargs)`.

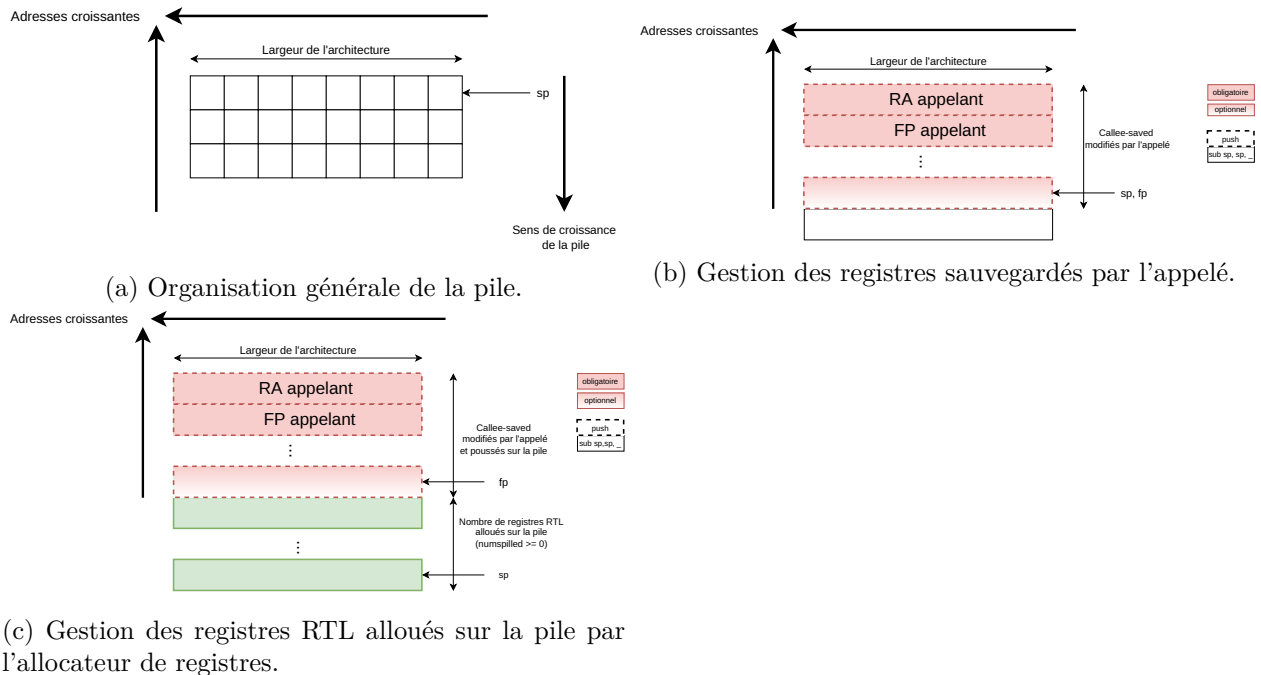


FIGURE 5 – Gestion de la pile en LTL et RiscV à l'entrée dans une fonction.

10.5.1 Sauvegarde des registres *caller-save*

Premièrement, il nous faut identifier l'ensemble des registres à sauvegarder. Une solution naïve consiste à sauvegarder tous les registres *caller-save*, à savoir les registres `a0` à `a7` et les registres `t0` à `t6`. Dans votre code, cela s'écrirait `arg_register @ reg_tmp`.

Une solution plus économe serait de ne sauvegarder que les registres *caller-save* qui sont vivants après l'appel de fonction. Cette information est disponible via l'argument `live_out` passé à la fonction `l1l_instrs_of_linear_instr`. L'ensemble des registres à sauvegarder est calculé par la fonction `caller_save`, définie juste au-dessus de la fonction `l1l_instrs_of_linear_instr`. Cette fonction prend en paramètres :

- `live_out` : le résultat de l'analyse de vivacité sur les arcs sortants ;
- `allocation` : l'allocation de registres pour cette fonction ;
- `rargs` : la liste des registres à passer en arguments.

La raison pour laquelle on passe la liste des registres arguments est la suivante. Rappelez-vous que les arguments d'une fonction sont passés dans les registres `a0` à `a7` : le premier argument dans `a0`, le second dans `a1`, etc. Supposons que l'on ait un appel de fonction Linear `f(r1,r2)`, et que `r1` soit associé au registre `a1` et `r2` au registre `a0` (c'est possible si `r1` et `r2` sont les arguments de la fonction appelante, comme illustré par le test `tests/funcall/argswap.e`).

Alors, lors du passage de paramètre (l'étape suivante), on va vouloir écrire `a1` dans `a0` et `a0` dans `a1`. Si on fait naïvement :

```
mv a0, a1
mv a1, a0
```

alors, il est évident que la valeur contenue dans `a1` ne sera pas la bonne, elle aura été écrasée par le `mv` précédent.

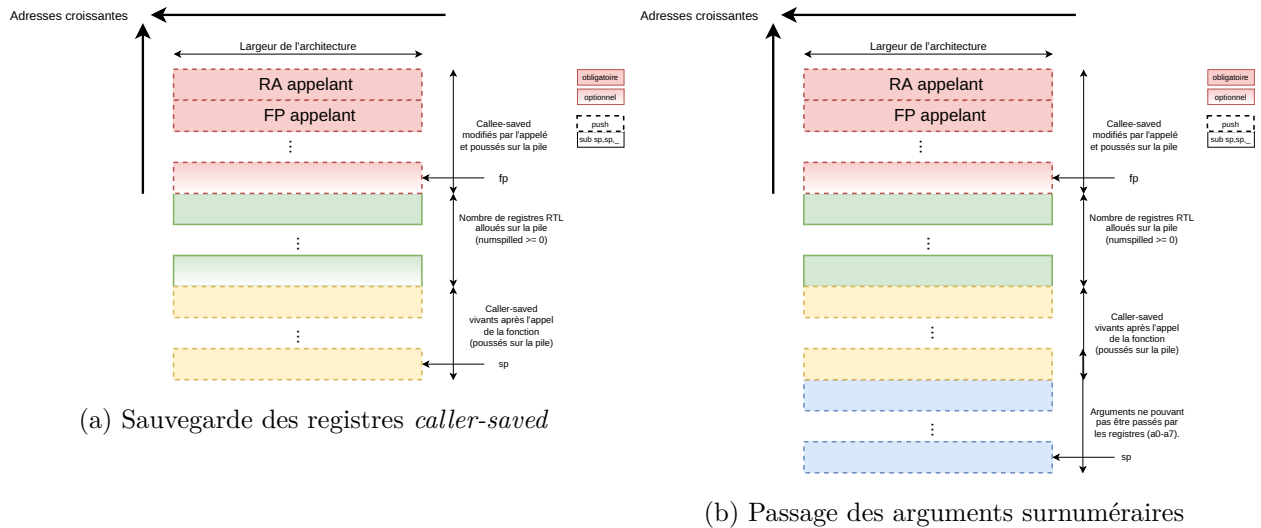


FIGURE 6 – Gestion de la pile en LTL et RiscV lors de l'appel d'une fonction.

C'est pourquoi la fonction `caller_save` anticipe ce genre de problème, calcule l'ensemble des arguments qui vont être écrasés (fonction `overwritten_args`) et les inclue dans l'ensemble des registres à sauvegarder.

Maintenant que les registres à sauvegarder sont identifiés, il faut émettre les instructions qui les sauvegardent. La fonction `save_caller_save` fait ce travail, et prend en paramètres :

- `to_save` : `l1l_reg list` : l'ensemble des variables à sauvegarder (on vient de le calculer !);
- `ofs` : `int` : le numéro du premier emplacement sur la pile disponible pour effectuer cette sauvegarde.

Si on reprend le schéma de la pile de la section précédente, et qu'on le complète.

L'emplacement prévu pour les sauvegardes de registres se situe à l'adresse pointée par `s0`, moins l'espace réservé aux registres évincés (spillés), moins l'espace réservé aux variables locales de la fonction.

Nous n'avons pas encore de variables locales à une fonction que nous stockons sur la pile, il nous suffit donc d'avoir le nombre de variables évincés, que l'on a notre disposition dans le paramètre `num_spilled` de la fonction `l1l_instrs_of_linear_instr`.

La fonction `save_caller_save` renvoie un triplet (`save_regs_instructions`, `arg_saved`, `ofs`), où :

- `save_regs_instructions` : `l1l_instr list` est la liste des instructions LTL qui se chargent de sauvegarder les registres `to_save` sur la pile;
- `arg_saved` : `(l1l_reg * int) list` est une liste d'association : une paire (`r`, `o`) dans cette liste signifie que le registre LTL `r` a été sauvegardé à l'emplacement `o` de la pile (`ofs` pour le premier, `ofs - 1` pour le deuxième, etc.)
- `ofs` est le nouveau premier emplacement disponible sur la pile, après avoir sauvegardé les registres. Nous en aurons besoin pour la suite.

Après cette liste d'instructions, une série d'écritures à des adresses `s0 - X` ont été émises. Il faut à présent positionner le pointeur de pile `sp` juste après les registres sauvegardés : c'est là qu'on utilise la valeur `ofs` retournée par la fonction `save_caller_save`.

10.5.2 Passage des paramètres

Vient ensuite le moment du passage des paramètres. La fonction `pass_parameters`, bien nommée, se charge de cette tâche. Elle prend en arguments :

- `rargs` : `reg list` : la liste des pseudo-registres RTL/Linear qui contiennent les arguments ;
- `allocation` : l'allocation de registres ;
- `arg_saved` : le résultat de `save_caller_save`, pour savoir où sont sauvegardés, si nécessaire, les arguments.

Elle peut échouer si certains registres de `rargs` ne sont pas alloués dans `allocation`. Si elle réussit, elle renvoie une paire (`parameter_passing_instructions`, `npush`), où :

- `parameter_passing_instructions` est une liste d'instructions LTL, qui effectuent le passage des paramètres ;
- `npush` est le nombre d'arguments qu'on a du passer sur la pile. (La plupart du temps, les arguments iront dans les registres `a0-7`, mais pour des fonctions avec plus de 8 arguments (comme dans `tests/funcall/lots_of_args.e` par exemple), on utilisera la pile.)

Si nécessaire, cette fonction effectue des « push » sur la pile, c'est-à-dire qu'elle décrémente le registre `sp` et écrit à l'adresse obtenue.

10.5.3 Appel de la fonction

Ici, c'est très simple, il suffit d'émettre l'instruction LTL `LCall callee_fname`, où `callee_fname` est le nom de la fonction appelée.

10.5.4 Dépilement des arguments

On restaure le pointeur de pile `sp` juste avant les arguments, *i.e.* on replace le pointeur de pile `npush` emplacements plus haut dans la pile.

10.5.5 Valeur de retour

La valeur de retour de la fonction est stockée dans `a0`, reportez-là à l'emplacement désiré (`rd`), si nécessaire.

10.5.6 Restauration des registres *caller-save*

Pour finir, il s'agit de restaurer la valeur des registres que l'on a sauvegardés. La fonction `restore_caller_save` calcule la liste des instructions qui effectuent cette tâche ; elle prend en argument une liste d'association comme celle renvoyée par `save_caller_save`.

Attention, si l'emplacement correspondant au registre dans lequel on doit stocker la valeur de retour est un registre physique, il ne faudra pas restaurer la valeur de ce registre !

10.6 Fonctions built-in

Maintenant que vous êtes capables de gérer des appels de fonctions, vous remarquerez que l'instruction `print` est en fait un cas particulier d'appel de fonction. On peut donc supprimer l'instruction `print` de notre compilateur pour la transformer en un simple appel de fonction.

Voici les endroits où vous devrez apporter des modifications :

- le lexer : le token `SYM_PRINT` n'a plus de raison d'être.

- la grammaire : un cas particulier n'est plus nécessaire.
- chacun des langages intermédiaires : plus besoin d'instruction dédiée à l'affichage !
- les interpréteurs de chacun de ces langages intermédiaires : on utilisera le mécanisme de *built-in* !

Le fichier `src/builtins.ml` contient un certain nombre de fonctions *built-in* prédéfinies pour vous. En particulier, la fonction `do_builtin oc mem fname vargs` donne la sémantique de l'appel de la fonction *built-in* dont le nom est `fname` avec les arguments donnés dans `vargs`. Le paramètre `oc` est un *output channel* comme vous en avez déjà manipulé jusqu'ici. Le paramètre `mem`, de type `Mem.t` est une mémoire, dont vous n'avez pas encore eu besoin (mais très bientôt, promis !). Dans les interpréteurs des différents langages, vous pouvez passer la mémoire associée à l'état `st` comme ceci : `st.mem`.

Cette fonction a comme type de retour `int option res` :

- `OK (Some i)` signifie que la fonction s'est évaluée sans erreur et retourne la valeur entière `i`.
- `OK None` signifie que la fonction s'est évaluée sans erreur et ne retourne pas de valeur.
- `Error msg` signifie que la fonction s'est évaluée avec l'erreur `msg`.

10.7 Débogage des programmes à bas niveau

À partir de maintenant, vous aurez peut être besoin de déboguer les programmes générés par votre compilateur afin de trouver des erreurs d'implémentation présents dans votre compilateur. Nous avons ajouté une nouvelle section (Appendix A) au sujet qui détaille comment procéder à la mise au point des programmes au niveau LTL et même RiscV.

11 Typage

Le but de cette extension de votre compilateur est d'ajouter des types à votre langage. Dans un premier temps, nous nous restreindrons aux type `int`, `char` et `void`. Dans les extensions ultérieures, nous ajouterons des pointeurs, des tableaux, des structures.

Un nouvel ensemble de fichiers de tests vous est proposé : les répertoires `tests/type_basic/`, `tests/type_funcall/` et `tests/char/`. Jetez un œil à certains de ces fichiers avant de continuer votre lecture, pour vous familiariser avec le nouveau langage que vous allez devoir compiler.

Les types sont un concept de haut-niveau : c'est-à-dire qu'ils n'impactent que les quelques premiers langages intermédiaires. Dans notre cas, vous n'aurez qu'à modifier votre grammaire `expr_grammar_action.g` et les fichiers relatifs au langage E.

Pour commencer, ajoutez au fichier `src/prog.ml` les définitions suivantes :

```
type typ =
  Tint
  | Tchar
  | Tvoid

let string_of_typ t =
  match t with
  | Tint -> "int"
  | Tchar -> "char"
  | Tvoid -> "void"
```

On définit ainsi un type `typ` qui correspond aux différents types que nous allons supporter dans un premier temps. On définit également une fonction d'affichage de ces types.

Voici un aperçu des étapes que vous devrez franchir. Celles-ci sont volontairement peu détaillées pour laisser libre cours à votre imagination !

1. Modifiez votre grammaire pour supporter :
 - les types dans les signatures de fonction : type de retour et types des arguments
 - les déclarations de variables locales, suivies ou non, d'une initialisation (*e.g.* `int x;` ou `int x = 3 * z;`)
 - les caractères littéraux : `char c = 'A';`
2. Ajoutez les définitions de tags nécessaires à `src/ast.ml`.
3. Ajoutez un constructeur `Echar of char` aux expressions de E (`src/elang.ml`).
4. Modifiez le type des fonctions `efun` dans `src/elang.ml` pour faire en sorte que :
 - `funargs` soit dorénavant une liste de `string * typ`, pour retenir le type des arguments ;
 - un nouveau champ `funvartyp : (string, typ) Hashtbl.t` enregistre le type de chaque variable locale ;
 - un nouveau champ `funrettype : typ` donne le type de retour de la fonction.
5. Dans `src/elang_gen.ml`, écrivez une fonction `type_expr (typ_var : (string, typ) Hashtbl.t) (typ_fun : (string, typ list * typ) Hashtbl.t) (e: expr) : typ res` qui calcule le type d'une expression.
 Les paramètres :
 - `typ_var` donne le type de chaque variable locale de la fonction courante.

- `typ_fun` donne la signature (type des arguments + type de retour) de chaque fonction déjà traitée dans le programme.
- `e` est l'expression à typer.

La fonction renvoie une erreur, notamment si une variable ou une fonction apparaissant dans l'expression n'a pas de type connu.

6. Toujours dans `src/elang_gen.ml`, ajoutez les paramètres `typ_var` et `typ_fun` aux fonctions `make_eexpr_of_ast` et `make_einstr_of_ast`. Vérifiez (avec un appel à `type_expr`) que les expressions construites ont un type, avant de les renvoyer dans `make_eexpr_of_ast`.

Dans `make_einstr_of_ast`, vérifiez que lors d'une affectation `v = e`, la variable `v` et l'expression `e` ont des types compatibles. (`int` et `char` sont compatibles, `int` et `void` non). (Écrivez une fonction séparée pour tester la compatibilité de 2 types.)

Toujours dans `make_einstr_of_ast`, mettez à jour `typ_var` lorsque vous rencontrez une déclaration de variable. Vérifiez au passage qu'on ne déclare pas de variable de type `void`.

Lors d'appels de fonctions (que ce soit comme une expression `Ecall` ou une instruction `Icall`), vérifiez que les arguments sont du type attendu (comparez avec la signature que vous pouvez obtenir via `typ_fun`).

7. Dans la fonction `make_fundef_of_ast`, initialisez une table `typ_var` puis mettez à jour la table `typ_fun` que vous devrez recevoir en argument.
8. Dans la fonction `make_eprog_of_ast`, initialisez la table `fun_typ` en y insérant notamment le type de trois fonctions *built-in* utiles :

```
let fun_typ = Hashtbl.create (List.length l) in
Hashtbl.replace fun_typ "print" ([Tint], Tvoid);
Hashtbl.replace fun_typ "print_int" ([Tint], Tvoid);
Hashtbl.replace fun_typ "print_char" ([Tchar], Tvoid);
...
```

9. On remarque qu'il faut avoir défini une fonction pour qu'on puisse l'appeler dans une autre. (Le typage devrait échouer si ce n'est pas le cas.)

Comment gérer les fonctions mutuellement récursives? Regardez les fichiers `tests/type_funcall/mutrec.e` et `tests/type_funcall/mutrec-hotstadter.e`. Ils contiennent des déclarations de fonctions. Faites-en sorte de traiter ces fichiers convenablement. (L'évaluation du fichier `tests/type_funcall/mutrec-hotstadter.e` prend un certain temps...)

12 Pointeurs

Nous allons maintenant ajouter des **pointeurs** à notre langage. Regardez notamment les tests dans le répertoire `tests/ptr/`. Ces programmes introduisent deux nouvelles constructions : l'opérateur `&` qui représente l'adresse d'une variable, et l'opérateur de déréférencement `*`.

Ainsi, le programme suivant, lancé avec les paramètres `x = 14` et `y = 12`, devrait renvoyer la valeur 26 :

```
void f(int* z, int x, int y){
    *z = x + y;
}
int main(int x, int y){
    int z = 3;
    f(&z, x, y);
    return z;
}
```

En effet, on passe l'**adresse** de `z`, ainsi que les **valeurs** de `x` et `y` à la fonction `f`. Cette fonction écrit à l'adresse contenue dans `z`, la somme de `x` et `y`. La fonction `main` finit par renvoyer la valeur de `z`, qui devrait contenir 26.

Ces nouveaux opérateurs engendrent les modifications suivantes dans notre compilateur :

- la grammaire doit reconnaître ces nouveaux opérateurs (d'ailleurs l'analyseur lexical doit aussi reconnaître le nouveau token `&` – pourquoi ne pas l'appeler `SYM_AMPERSAND`?)
- les types (dans `src/prog.ml`) sont enrichis avec un nouveau constructeur `Tptr ty` : un pointeur sur des éléments de type `ty`.
- on ajoute au langage E (fichier `src/elang.ml`) :
 - une expression `Eaddrof of expr` pour symboliser l'expression `&e`
 - une expression `Eload of expr` pour symboliser l'expression `*e`
 - une instruction `Istore of expr * expr` pour symboliser l'instruction `*e = v`;

De plus, une fonction (type `efun`) contient deux nouveaux champs :

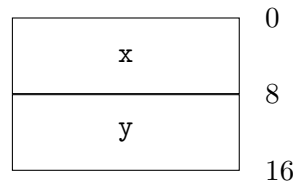
- un champ `funvarinmem : (string, int) Hashtbl.t` qui associe chaque nom de variable qui doit résider en mémoire un déplacement dans le bloc de pile associé à chaque activation de cette fonction
- un champ `funstksz : int` qui donne la taille du bloc de pile de cette fonction.

Ces derniers éléments méritent un peu d'explications. Supposons que vous ayez le code C suivant :

```
int f(){
    int x = 3;
    int y = 4;
    int z = 5;
    z = g(&y, &x);
    return z;
}
```

Dans cet exemple, on utilise l'opérateur `&` sur les variables `x` et `y` : ces variables ont donc besoin d'être allouées en mémoire, plus précisément sur la pile puisque ce sont des variables locales à la fonction `f`. En revanche, la variable `z` peut rester, comme l'étaient toutes les variables jusqu'à présent, des *temporaires* qui ne sont pas allouées en mémoire.

On aura donc pour cet exemple un bloc de pile de 16 octets (à supposer que les `int` sont codés sur 8 octets).



Ici, la table `funvarinmem` associera la variable `x` au déplacement (ou *offset*) 0 et la variable `y` au déplacement 8. Le champ `funstksz` vaudra 16.

12.1 Génération de E

Lors des séances précédentes, vous avez dû ajouter une fonction `type_expr` à `src/elang_gen.ml`. Assurez-vous de mettre à jour le typage concernant les pointeurs. On notera que `p + i`, `i + p` et `p - i` ont le même type que `p` lorsque `p` est un pointeur et `i` est un entier (ou un caractère, d'ailleurs). Étant donnés 2 pointeurs `p` et `q` de même type, on peut également comparer ces pointeurs et obtenir un entier. Toutes les autres opérations unaires et binaire sont indéfinies.

Comme on l'a vu précédemment, certaines variables seront allouées sur la pile et d'autres dans des temporaires. Le critère est le suivant : si l'adresse d'une variable est prise dans le programme (au moyen de l'opérateur `&`), alors cette variable sera allouée dans la pile. (Plus tard, on ajoutera à la pile les variables non-scalaires, *i.e.* les tableaux ou les structures.)

Écrivez une fonction `addr_taken_expr (e: expr) : string Set.t` qui récolte l'ensemble des variables dont l'adresse est prise dans une expression `e`. Faites de même pour les instructions avec une fonction `addr_taken_instr (i: instr) : string Set.t`.

Dans la fonction `make_fundef_of_ast`, initialisez une table de hachage `funvarinmem` qui associe à chaque variable allouée sur la pile un *offset* (déplacement) dans le bloc de pile associé à cette fonction. Pour ce faire, vous aurez besoin de définir une fonction `size_type (t: typ) : int res` qui renvoie la taille occupée en mémoire par une variable de type `t`. Définissez cette fonction dans `src/prog.ml`.

12.1.1 Interpréteur de E

Les différentes fonctions de l'interpréteur E prennent un argument supplémentaire : un pointeur de pile `sp : int` qui donne l'adresse de départ du bloc de pile dans la mémoire.

Ce pointeur de pile sera notamment utile pour les expressions `Eaddrof` et `Eload`, ainsi que pour les instructions `Istore`.

La mémoire est accessible depuis l'état `st` via `st.mem`. Les différentes opérations intéressantes sur la mémoire sont les suivantes :

- `Mem.write_bytes (m: Mem.t) (addr: int) (bytes: int list) : unit res.` Cette fonction écrit dans la mémoire `m` à l'adresse `addr` la liste d'octets `bytes`. Renvoie `OK ()` si tout s'est bien passé, `Error msg` sinon (accès hors des bornes de la mémoire).

Cette fonction est utilisée en conjonction avec la fonction `split_bytes n v` qui découpe un entier `v` en `n` morceaux (chacun de la taille d'un octet).

- `Mem.read_bytes_as_int (m: Mem.t) (addr: int) (sz: int) : int res`. Cette fonction lit dans la mémoire `m` aux adresses `[addr; addr + sz[` et renvoie l'entier représenté par ces octets.

Pour l'évaluation des opérateurs binaires, attention aux additions et soustractions sur les pointeurs. Soit un pointeur `p` de type `Tptr t` et un entier `i` :

- l'expression `p + i` s'évalue en fait en `p + i * sizeof(t)`
- l'expression `p - i` s'évalue en fait en `p - i * sizeof(t)`
- l'expression `i + p` s'évalue en fait en `i * sizeof(t) + p`

Attention aussi aux expressions `Evar v` et aux instructions `Iassign(v,e)` : la variable est peut-être en mémoire et il faut y accéder correctement !

Notez aussi qu'on ne peut pour l'instant appliquer l'opérateur `&` que sur des variables, et éventuellement sur des expressions `Eload e`.

Pour savoir combien d'octets lire ou écrire (expressions `Eload addr` et instructions `Istore(addr, v)`), référez-vous au type de l'expression `addr` et à la taille du type pointé.

Dans la fonction `eval_efun`, réservez la place nécessaire sur la pile en ajustant `sp` comme nécessaire.

Vous aurez besoin de typage, donc de la fonction `type_expr` définie dans `src/elang_gen.ml`. N'oubliez pas d'ajouter un `open Elang_gen` en haut de votre fichier. Vous aurez également besoin d'une table `typ_fun` que vous pouvez initialiser dans `eval_eprog` et propager dans les différentes fonctions. Pour la table `typ_var`, vous devriez trouver votre bonheur dans les champs des fonctions (type `efun`).

12.2 Génération de CFG

Dans le langage CFG, on se débarrasse des informations de typage. On ajoute, dans le fichier `src/cfg.ml` :

- une expression `Estk of int`, qui représente une adresse dans le bloc de pile de la fonction courante ;
- une expression `Eload of expr * int. Eload(e, sz)` représente une lecture en mémoire à l'adresse représentée par `e`, sur `sz` octets
- une instruction `Cstore of expr * expr * int * int. Cstore(addr, v, sz, succ)` représente une écriture en mémoire à l'adresse représentée par l'expression `addr`, la valeur que l'on écrit est celle de l'expression `v` sur `sz` octets. Le successeur de cette instruction est `succ`, comme pour l'instruction `Cassign` par exemple.
- un champ `cfgfunstksz : int` au type des fonctions CFG `cfg_fun`.

La compilation des expressions et instructions `E` en expressions et nœuds CFG imite l'interpréteur de `E` que vous venez d'écrire. Il vous faudra donc effectuer les mêmes vérifications de type, ajuster les opérations sur les pointeurs, transformer les `Evar v` et les affectations `Iassign(v,e)` en lectures et écritures sur la pile lorsque cela est nécessaire.

12.3 Interpréteur de CFG

De manière similaire au langage `E`, le langage CFG va avoir besoin d'un pointeur de pile `sp` pour évaluer nos nouvelles expressions et instructions.

12.4 RTL

Ajoutez à `src/rtl.ml` les opérations :

- `Rstk of reg * int : Rstk(rd, i)` écrit dans le registre `rd` l'adresse située à un déplacement `i` du début du bloc de pile.
- `Rload of reg * reg * int : Rload(rd, rs, sz)` stocke dans le registre `rd` la valeur sur `sz` octets lue à l'adresse contenue dans `rs`
- `Rstore of reg * reg * int : Rstore(rd, rs, sz)` écrit la valeur contenue dans `rs` sur `sz` octets à l'adresse contenue dans `rd`.

On ajoute également un champ `rtlfunstksz : int` aux fonctions `rtl_fun`.

Adaptez l'interpréteur comme vu avant.

12.5 Linear

Comme Linear partage l'ensemble des opérations RTL, il suffit d'ajouter dans `src/linear.ml` un champ `linearfunstksz` au type des fonctions.

Adaptez l'interpréteur comme vu avant.

12.6 LTL

La fonction `ltl_instrs_of_linear_instr` prend un nouvel argument `numlocals : int` : il s'agit du nombre d'emplacements sur la pile qu'il faut réserver pour les variables locales. Ce nombre sera calculé dans la fonction `ltl_fun_of_linear_fun` en fonction de la taille `linearfunstksz` de la fonction Linear. Attention, on doit passer d'un nombre d'octets (`linearfunstksz`) à un nombre d'emplacements (de taille `!Archi.wordsize`, soit 8 en 64 bits ou 4 en 32 bits).

Propagez cette valeur où cela vous semble nécessaire pour respecter les schémas de la Figure 5. Assurez-vous également de compiler comme il se doit les instructions `Rstk`, `Rload` et `Rstore`.

Dans la fonction `ltl_fun_of_linear_fun`, adaptez le *prologue* de la fonction pour réserver de l'espace pour les variables stockées sur la pile.

13 Structures

Nous nous intéressons maintenant aux **structures**. Nous souhaitons être en mesure de compiler des programmes comme le suivant :

```
1 struct mastruct {
2     int x;
3     int y;
4 };
5
6 int main(){
7     struct mastruct S;
8     S.x = 12;
9     S.y = 3;
10    return (S.x + S.y);
11 }
```

Ce test est donné dans `tests/structs/struct.e`.
Voici un aperçu des changements à effectuer.

13.1 La grammaire

Modifiez la grammaire pour autoriser des définitions de type de structure, comme dans les lignes 1 à 4 du test ci-dessus. Un fichier devient alors une liste de définitions globales, chaque définition étant soit une définition de fonction, soit une définition de type de structure.

Permettez aussi aux fonctions de prendre des arguments de type « structure », et les définitions de variables locales de type « structure ».

13.2 Le langage E

Ajoutez au langage E :

- une expression `Egetfield of expr * string` : `Egetfield(e, f)` représente un accès au champ `f` de la structure dont l'adresse est représentée par l'expression `e`
- une instruction `Isetfield of expr * string * expr` : `Isetfield(s, f, e)` représente une écriture de l'expression `e` dans le champ `f` de la structure dont l'adresse est représentée par l'expression `e`.

Aussi, le type `eprog` devient :

```
type eprog = efun prog * (string, (string * typ) list) Hashtbl.t
```

La deuxième composante de la paire est une table de hachage qui donne, pour chaque type de structure, la liste des champs de la structure avec leurs types. Cela nous sera utile, notamment pour connaître la disposition des champs au sein des structures.

13.3 Le fichier `src/prog.ml`

Dans `src/prog.ml`, ajouter un nouveau constructeur à `typ` pour les types de structures : `Tstruct of string` (le paramètre de type `string` est le nom de la structure).

Adaptez la fonction `size_type` pour calculer la taille des structures. Ajoutez-y comme argument une table de hachage comme celle décrite ci-dessus. Par exemple, la taille de la structure `mastruct` dans l'exemple précédent est 16 (2 champs de taille 8).

Ajoutez également les deux fonctions suivantes :

```
— let rec field_offset
  (structs: (string, (string * typ) list) Hashtbl.t)
  (s: string) (f: string) : int res
```

Un appel à `field_offset structs s f` renvoie l'offset du champ `f` dans une structure de type `s`. Par exemple, `field_offset structs "mastruct" "x"` renverra `OK 0` et `field_offset structs "mastruct" "y"` renverra `OK 8`. Un appel erroné `field_offset structs "mastruct" "z"` renverra `Error "No such field 'z'"`, ou un message d'erreur similaire dans la langue de votre choix.

```
— let rec field_type
  (structs: (string, (string * typ) list) Hashtbl.t)
  (s: string) (f: string) : typ res
```

Un appel à `field_type structs s f` renvoie le type du champ `f` dans une structure de type `s`. Par exemple, `field_type structs "mastruct" "x"` et `field_type structs "mastruct" "y"` renverront `OK Tint`.

13.4 Génération de E

Dans `src/elang_gen.ml`, ajoutez un argument `typ_struct : (string, (string * typ) list) Hashtbl.t` aux fonctions `type_expr`, `make_eexpr_of_ast`, `make_einstr_of_ast`.

Modifiez la fonction `make_fundef_of_ast` pour qu'elle traite aussi les définitions de types structure. Sa tâche sera alors d'augmenter la table `typ_struct` lorsqu'une définition de structure est rencontrée.

Lors de la construction de l'ensemble `funvarinmem` des variables locales qui doivent aller dans la mémoire, ajoutez, en plus des variables dont on prend l'adresse, les variables de type `struct`. En effet, celles-ci doivent être allouées en mémoire. D'une manière plus générale, on mettra en mémoire toutes les variables qui ne sont pas *scalaires*, c'est-à-dire de type *simple* (entier, pointeur, caractère).

Il faudra aussi, évidemment, modifier les fonctions `type_expr` et `make_eexpr_of_ast` pour typer et générer des expressions E `Egetfield`, et la fonction `make_einstr_of_ast` pour générer des instructions `Isetfield`. Faites bien attention à donner comme premier paramètre de `Egetfield` et `Isetfield` l'adresse de la structure : introduisez éventuellement un `Eaddrof...`

13.5 Interpréteur de E

Dans `src/elang_run.ml`, ajoutez aussi un paramètre `typ_struct` là où cela vous semble nécessaire, du même type que précédemment. Vousinstancierez ce paramètre avec la table générée dans `src/elang_gen.ml`.

Cela vous permettra d'évaluer les accès en lecture et écriture aux champs d'une structure. Vérifiez, dans l'interpréteur que lors de l'évaluation de `Egetfield(e, f)` ou `Isetfield(e, f, v)`, l'expression `e` est bien de type `Tptr (Tstruct structname)`. Les fonctions `field_offset` et `field_type` définies plus haut dans `src/prog.ml` devraient vous être utiles.

Pour l'évaluation de `Egetfield(e,f)`, si le champ est de type *scalaire* (entier, pointeur, caractère), lisez la valeur contenue en mémoire. Sinon, s'il s'agit d'une structure, renvoyez simplement l'adresse de ce champ.

13.6 Génération de CFG

Dans `src/cfg_gen.ml`, ajoutez aussi un paramètre `typ_struct`, encore du même type que précédemment, là encore, là où cela vous semble pertinent. Vousinstancierez ce paramètre avec la table générée dans `src/elang_gen.ml`.

On n'ajoute pas d'expressions ou d'instructions spécifiques aux structures dans le langage CFG. On va simplement transformer les accès aux champs des structures en simples accès à la mémoire.

Ainsi, on remplacera la lecture `S.y` en `*(&S + 8)` par exemple. L'adresse de `S` aura aussi été remplacée par un emplacement sur la pile : cela donnera donc en fait `*(stk(0) + 8)` ou quelque chose de similaire.

Une fois cette transformation effectuée, nul besoin de modifier le reste du compilateur, il doit normalement déjà savoir gérer les lectures et écritures en mémoire.

14 Tableaux

Nous ajoutons maintenant les tableaux !

Voici un exemple de programme utilisant des tableaux (`tests/array/array.e`)

```
int main(){
  int t[10];
  t[0] = 5;
  t[1] = 3 + t[0];
  return t[1];
}
```

On déclare un tableau de 10 entiers, on initialise la première case à 5, la deuxième à la valeur de la première plus 3, puis on renvoie la valeur de cette deuxième case. Le résultat attendu, sans surprise, est 8.

14.1 La grammaire

Étendez la grammaire pour être capables de déclarer des variables de type « tableau », et d’y accéder en lecture et écriture (`t[i]`).

14.2 Génération et interpréteur de E

Ajoutez un type dans `src/prog.ml` qui représente les tableaux : `Ttab of typ * int` : `Ttab(ty,n)` représente un tableau de `n` éléments, chacun de type `ty`.

On notera que les types `Ttab(ty, n)` et `Tptr ty` sont compatibles, c’est-à-dire, on peut donner un paramètre de type `Ttab(ty,n)` à une fonction qui attend un argument de type `Tptr ty`. On modifiera donc l’évaluation des expressions, notamment pour que les additions et soustractions se comportent avec les tableaux comme avec les pointeurs.

Transformez les accès à des tableaux en simples accès à la mémoire. Ainsi, l’accès `t[i]` est transformé en `*(t + i)`. Nul besoin ici de multiplier `i` par la taille des éléments pointés par `t`, cette transformation est normalement déjà faite dans l’évaluation de l’addition entre pointeurs et entiers.

De la même manière que pour les structures, lors de l’évaluation d’un `Eload(e)`, si le type pointé par `e` est un scalaire, on rendra sa valeur en mémoire, mais s’il s’agit d’un type structure ou tableau, on rendra simplement son adresse.

14.3 Génération de CFG

Comme pour l’évaluation de E, il faut traiter les tableaux comme des pointeurs lors des additions et soustractions. C’est tout ce qu’il faut modifier !

15 Variables globales

On s'attaque maintenant aux variables globales. Effectivement, jusqu'à présent, toutes les données que nous pouvions manipuler étaient des variables locales à une fonction. Voici un exemple de programme utilisant des variables globales (`tests/globals/globarray.e`) :

```
int max = 20;
int objs[20];
int main(){
    int i = 0;
    int sum = 0;
    while (i < max){
        objs[i] = sum;
        sum = sum + i;
        i = i + 1;
    }
    i = 0;
    while (i < max){
        print(objs[i]);
        i = i + 1;
    }
    return objs[max-1];
}
```

On y définit deux variables globales : un entier `max` et `objs`, un tableau de 20 entiers.

On stocke dans chaque case i du tableau la somme des entiers de 1 à i , puis on affiche chacune des 20 cases du tableau. La sortie attendue est la suivante :

```
0
0
1
3
6
10
15
21
28
36
45
55
66
78
91
105
120
136
153
171
```

15.1 La grammaire

Étendez la grammaire pour permettre la définition de variables globales, avec ou sans initialisation. Attention à ne pas générer de conflit dans votre grammaire.

15.2 Le langage E

Ajoutez une expression `Eglobvar of string` aux expressions E. L'expression `Eglobvar "max"` contient l'adresse en mémoire de la variable globale nommée `"max"`. Effectivement, toutes les variables globales sont stockées en mémoire. On verra par la suite comment les initialiser.

15.3 Génération de E

De la même manière que vous avez ajouté un paramètre `typ_struct` pour les structures, ajoutez un paramètre `typ_glob : (string, typ) Hashtbl.t` à chaque fonction qui en a besoin (`type_expr`, `make_eexpr_of_ast`, `make_einstr_of_ast`...).

Vous devrez différencier, lorsque vous rencontrez un identifiant de variable, s'il s'agit d'une variable locale (`Evar v`) ou globale (`Eglobvar v`). Utilisez les informations de typage (`typ_var` et `typ_glob`) pour ce faire.

L'initialisation d'une variable globale doit être une constante, de type entier ou caractère. En cas d'absence d'initialiseur, on se contentera de réserver de l'espace.

Pour caractériser cela, ajoutez à `src/prog.ml` le type suivant :

```
type init_data =
  | Iint of int
  | Ichar of char
  | Ispace of int
```

On changera également la définition du type `gdef` pour y ajouter un constructeur `Gvar of typ * init_data`.

Par exemple les deux définitions globales de l'exemple en début de section généreront les définitions globales suivantes : `Gvar(Tint, Iint 20)` et `Gvar(Ttab(Tint, 20), Ispace(160))` (parce que $20 * 8 = 160$).

Modifiez la fonction `make_fundef_of_ast` pour traiter les définitions de variables globales. Au passage, renommez-là en `make_globdef_of_ast` et changez son type de retour en remplaçant `(string * efun)` par `(string * efun gdef)`.

15.4 Interpréteur de E

Ajoutez un champ `glob_env : (string, 'a) Hashtbl.t` au type `state` dans `src/prog.ml`, qui recueillera les adresses des différentes variables globales.

Enfin, ajoutez, toujours dans ce fichier, une fonction

```
let init_glob
  (mem: Mem.t) (glob_env: (string, int) Hashtbl.t)
  (p: 'a prog) (startglob: int) : int res = ...
```

qui, étant donnés un programme `p` et une adresse de départ `startglob`, construit (enrichit) un environnement global `glob_env` et écrit dans la mémoire `mem`. Pour ce faire, on itérera sur les définitions du programme `p` (en ignorant les définitions de fonctions) et on écrira l'entier ou le

caractère d'initialisation de la globale en question, ou bien on se contentera de réserver de l'espace si l'initialiseur est `Ispace n`.

Ajoutez dans `src/elang_run.ml` un paramètre `typ_glob` là où cela vous semble nécessaire. L'évaluation de l'expression `Eglobvar v` devrait être similaire au cas `Evar v`, sauf que vous chercherez dans l'environnement global plutôt que dans l'environnement local et que les variables sont toujours allouées dans la mémoire.

Dans la fonction `eval_prog`, lorsque vous itérez sur les définitions du programme et que vous tombez sur une variable globale, il faut mettre à jour son type dans la table `typ_glob`. Aussi, appelez la fonction `init_state` que vous avez définie dans `src/prog.ml`. Vous pouvez utiliser, par exemple, l'adresse `0x1000` comme adresse de départ des globales. (Le choix de cette adresse n'a pas vraiment d'influence. Il faut simplement qu'on ait suffisamment de place pour y stocker l'ensemble des variables globales et que ça n'entre pas en conflit avec la pile. Lorsque vous lancez votre compilateur, la taille de la mémoire est par défaut de 10000 (soit `0x2710`). Allouer les globales à `0x1000` devrait ne pas poser de problème pour l'ensemble des tests qui vous sont proposés. La « vraie » allocation, pour les programmes RISC-V sera faite par le linker pour vous, avec davantage de distance entre les différentes zones de la mémoire.)

15.5 Génération de CFG

Répétez plus ou moins les mêmes opérations que pour l'interpréteur de E : ajoutez un champ `typ_glob`, ajoutez une expression `Eglobvar of string`, étendez `typ_glob` dans la fonction `cfg_prog_of_eprog` pour y ajouter le type des globales.

15.6 Autres langages intermédiaires

Pour RTL, ajoutez une opération `Rglobvar of reg * string`. L'opération `Rglobvar(rd,v)` met l'adresse de la globale `v` dans le registre `rd`.

Pour LTL, ajoutez une opération `LGlobvar of ltl_reg * string`, cette fois ci avec un registre machine.

Pour chacune des passes de compilation, il faut aussi propager les définitions globales. Il suffit de les laisser inchangées (pas besoin de les compiler).

On ajoutera également à l'état des programmes RTL et Linear un champ `glob_env` de la même nature que celui défini pour E et CFG dans le type `state` de `src/prog.ml`, et encore de manière similaire pour les programmes LTL dans `src/ltl_run.ml`.

Dans `src/riscv.ml`, compilez l'opération `LGlobvar(rd, s)` grâce à l'instruction RISC-V `la rd, symbol`, qui écrit l'adresse du symbole `symbol` dans le registre `rd`.

Toujours dans `src/riscv.ml`, il faut déclarer les variables globales que vous utilisez dans une section `.data`, que l'on déclare en écrivant `.section .data`. Pour chaque variable, si elle est initialisée, il faut donner son contenu sous forme d'une liste d'octets (on pourra utiliser la fonction `split_bytes`, déjà décrite dans ce document). Si elle n'est pas initialisée, on y mettra des octets nuls.

Par exemple :

```
.section .data
max:
    .byte 20,0,0,0,0,0,0,0
objs:
    .zero 160
```

16 Space Invaders !

Pour terminer ce projet de compilation en apothéose, nous vous proposons de jouer à un jeu vidéo dernier cri (1978) qui nous vient de la société japonaise Taito : スペースインベーダー .

Vous trouverez le code source de ce jeu dans le répertoire `tests/invader/invader.e`. De nouvelles constructions du langage vous empêcheront de le compiler directement. À vous de vous débrouiller pour traiter ces nouvelles constructions.

16.1 Nouvelles constructions du langage

En particulier, vous devrez traiter :

- les nombres hexadécimaux comme `0x123aBc12F`. (Vous ne devrez toucher qu'au fichier `src/e_regexp.ml` pour cela.)
- les opérateurs booléens comme `&&` et `||`.

En vrai C, l'expression `e1 && e2` sera évaluée de la façon suivante. On évalue `e1` : si sa valeur est fausse (0), alors on n'évalue pas `e2` (qui peut contenir des effets de bord) et on retourne 0. Sinon, on retourne 0 si `e2` vaut 0, et 1 sinon.

De manière similaire, lors de l'évaluation de `e1 || e2`, on n'évalue pas `e2` si `e1` vaut vrai (`!= 0`).

Cela étant dit, en première approximation, vous pourrez évaluer les deux opérandes et en calculer le ET / OU booléen, à l'aide des instructions RISC-V `and` et `or`, en notant bien que ceci ne fonctionne que lorsque les expressions `e1` et `e2` s'évaluent en 0 ou en 1. À moins d'une erreur de notre part, le code dans `invader.e` et le code de `src/riscv.ml` respectent cette convention.

- l'opérateur ET bit-à-bit `&`. Vous pourrez utiliser la fonction `land : int -> int -> int` qui effectue cette opération sur les entiers OCaml, et l'instruction RISC-V `and`.
- l'opérateur NON logique `!`. Vous utiliserez l'instruction RISC-V `seqz rd, rs` qui écrit la valeur 1 dans `rd` si `rs` vaut 0, et la valeur 0 dans `rd` sinon.

16.2 Compilation

La première étape pour compiler ce programme est d'émettre du code assembleur.

Dans le répertoire `tests/invader/` :

```
$ ../../main.native -f invader.e -m32 -nostart -clever-regalloc -no-dot -json /dev/null
```

Cette commande produit le fichier `tests/invader/invader.s` qui contient le code assembleur de cette application.

L'option `-nostart` indique que le symbole `_start`, point d'entrée du programme, ne doit pas figurer dans le code généré. En effet celui-ci apparaît dans une librairie que nous allons lier à notre programme tout de suite :

```
$ riscv32-unknown-elf-gcc -march=rv32im -mabi=ilp32 \
-I. -I./include -g -DENV_QEMU=1 -T ./extended.lds \
-nostartfiles -nostdlib -nostdinc -static \
invader.s crt.s setup.c font.c libscreen.c libfemto.a itoa.c \
-o invader
```

Vous devriez obtenir un exécutable RISC-V, ce que vous pouvez vérifier de la manière suivante :

```
$ file invader
invader: ELF 32-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, with
↳ debug_info, not stripped
```

16.3 Exécution

Vous avez un exécutable, mais vous ne savez pas exécuter des binaires RISC-V. Oh non ! Il ne vous reste plus qu'à vous procurer sur eBay une machine du type suivant :



Mais non ! On va pouvoir utiliser un émulateur de processeur RISC-V.
Nous vous avons préparé une installation de `qemu` spécifique dans un conteneur docker.
Pour installer docker sur Ubuntu ou Debian :

```
$ sudo apt install docker.io xtightvncviewer
```

Vous pouvez ensuite récupérer l'image docker préparée pour vous :

```
$ docker pull pwilke/qemu-compile:latest
```

Placez-vous maintenant dans le répertoire `tests/invader`, où vous avez généré le binaire RISC-V `invader`.

```
$ docker run --rm -v $(pwd):/data -p 5900:5900 -it qemu-compile invader
```



Ah, mais ça n'affiche rien !

Oui, en fait l'image est envoyée dans un socket VNC (Virtual Network Computing). (On aurait sans doute pu afficher directement sur votre écran, mais de manière non portable.)

Il faut donc se connecter avec un client VNC.

Si vous êtes sous Linux, on vous a fait installer un client VNC ci-dessus, il suffira de lancer ceci :

```
$ vncviewer 127.0.0.1:5900
```

Si vous êtes sous Windows ou Mac OS, télécharger un client VNC (<https://www.clubic.com/telecharger-fiche46296-vnc-viewer.html>) a l'air formidable.)

Youpi! (Enfin youpi si ça fonctionne...)

A Débogage des programmes au niveau des langages bas niveau

A.1 Débogage d'un programme au niveau du langage LTL

Il est possible de demander à l'interpréteur du langage LTL de passer dans un mode de débogage via une option particulière (`-ltdb-debug`) de la ligne de commande du compilateur. Dans ce cas, le compilateur ne rend pas la main et attend des commandes provenant d'un logiciel extérieur sur une websocket⁵ en écoute à l'adresse IP 127.0.0.1 sur le port 8080.

Nous vous fournissons le programme externe qui va agir comme un débogueur. Celui-ci est écrit en Javascript. Son code est situé dans le répertoire `ldb/`.

Ainsi pour déboguer au niveau du langage LTL le code produit par votre compilateur à partir du code source `tests/mes-tests/file.e`, il suffit d'entrer la commande suivante dans un terminal :

```
$ ./main.native -f tests/mes-tests/file.e -ltdb-debug
```

Puis dans un second terminal, ou un nouvel onglet de votre terminal :

```
$ firefox ldb/ldb.html
```

Vous verrez alors apparaître une fenêtre qui ressemble à celle illustrée par la figure 7. Cette

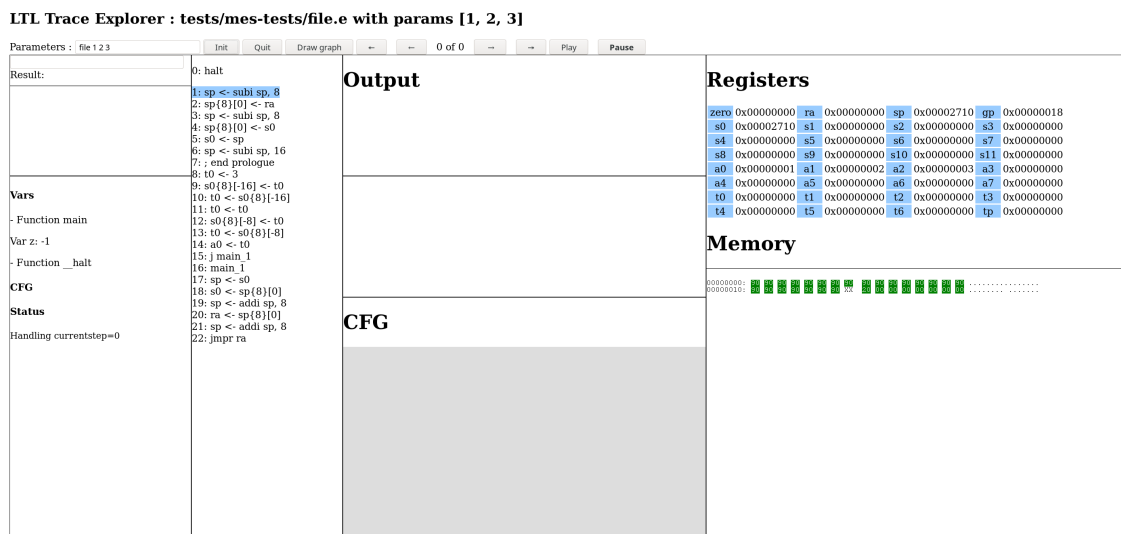


FIGURE 7 – L'interface du débogueur LDB.

fenêtre permet de piloter l'interpréteur LTL intégré au compilateur.

A.2 Débogage d'un programme au niveau du langage RiscV

5. Une websocket est une socket TCP qui peut être ouverte depuis un navigateur Web pour se connecter à un site distant avec cependant des restrictions de sécurité. Le flux TCP transporte un protocole dédié qui peut-être décodé sous la forme d'objets JSON depuis un script Javascript s'exécutant dans le navigateur.

Parameters :

Init

(a) Paramètres à passer au programme.

(b) Initialisation de l'interpréteur.



(c) Avancer d'une instruction.



(d) Revenir en arrière d'une instruction.



(e) Exécuter le programme jusqu'au prochain point d'arrêt ou la fin du programme.

(f) Revenir en arrière dans la trace jusqu'au précédent point d'arrêt ou au début du programme.

Memory

```
00000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
00000010: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000026f0: 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00002700: 10 27 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Memory

```
00000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
00000010: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000026f0: 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00002700: 10 27 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

(g) Lecture d'une zone mémoire.

(h) Écriture d'une zone mémoire.

FIGURE 8 – Quelques éléments de l'interface graphique du débogueur LDB.

```
0: halt
1: sp <- subi sp, 8
2: sp{8}[0] <- ra
3: sp <- subi sp, 8
4: sp{8}[0] <- s0
5: s0 <- sp
6: sp <- subi sp, 16
7: ; end prologue
8: t0 <- 3
9: s0{8}[-16] <- t0
10: t0 <- s0{8}[-16]
11: t0 <- t0
12: s0{8}[-8] <- t0
13: t0 <- s0{8}[-8]
14: a0 <- t0
15: j main_1
16: main_1
17: sp <- s0
18: s0 <- sp{8}[0]
19: sp <- addi sp, 8
20: ra <- sp{8}[0]
21: sp <- addi sp, 8
22: jmp ra
```

(a) Pause d'un point d'arrêt.

```
0: halt
1: sp <- subi sp, 8
2: sp{8}[0] <- ra
3: sp <- subi sp, 8
4: sp{8}[0] <- s0
5: s0 <- sp
6: sp <- subi sp, 16
7: ; end prologue
8: t0 <- 3
9: s0{8}[-16] <- t0
10: t0 <- s0{8}[-16]
11: t0 <- t0
12: s0{8}[-8] <- t0
13: t0 <- s0{8}[-8]
14: a0 <- t0
15: j main_1
16: main_1
17: sp <- s0
18: s0 <- sp{8}[0]
19: sp <- addi sp, 8
20: ra <- sp{8}[0]
21: sp <- addi sp, 8
22: jmp ra
```

(b) Point d'arrêt atteint.

FIGURE 9 – Quelques éléments de l'interface graphique du débogueur LDB.

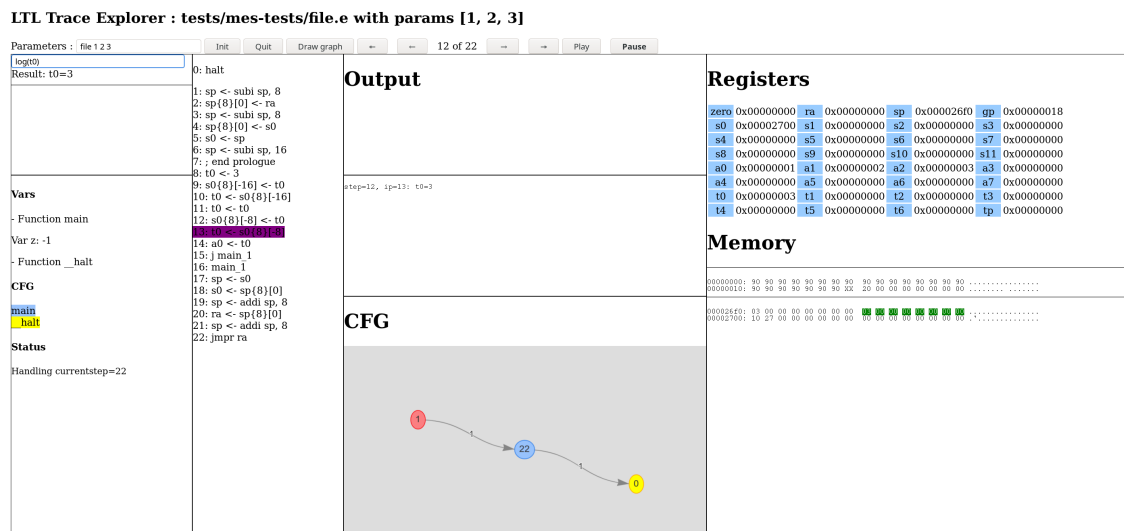


FIGURE 10 – L’interface du débogueur LDB lors de l’utilisation de l’évaluation d’expression et des points d’arrêt.