

# Conception et implémentation d'un compilateur pour le langage E

## 1 But de ces séances de travaux pratiques

Au cours de ces 5 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 le temps additionnel. 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 x86-32 et RISC-V, qui seront finalement assemblés par des assembleurs existants et qui pourront être exécutés sur vos machines.

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 (ou de vos soirées) 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. Peut-être (comprendre *sûrement*) que nous modifierons le squelette au cours de ce projet. 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 ci-dessous, et est illustré sur l'exemple de la Figure 2a.

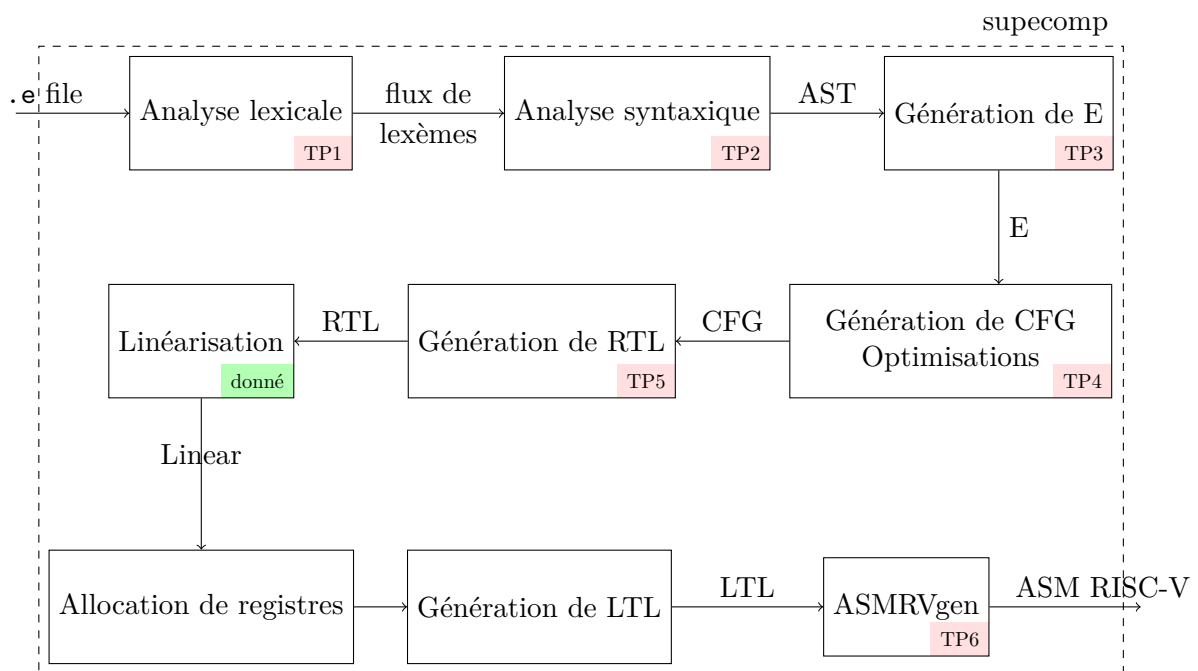
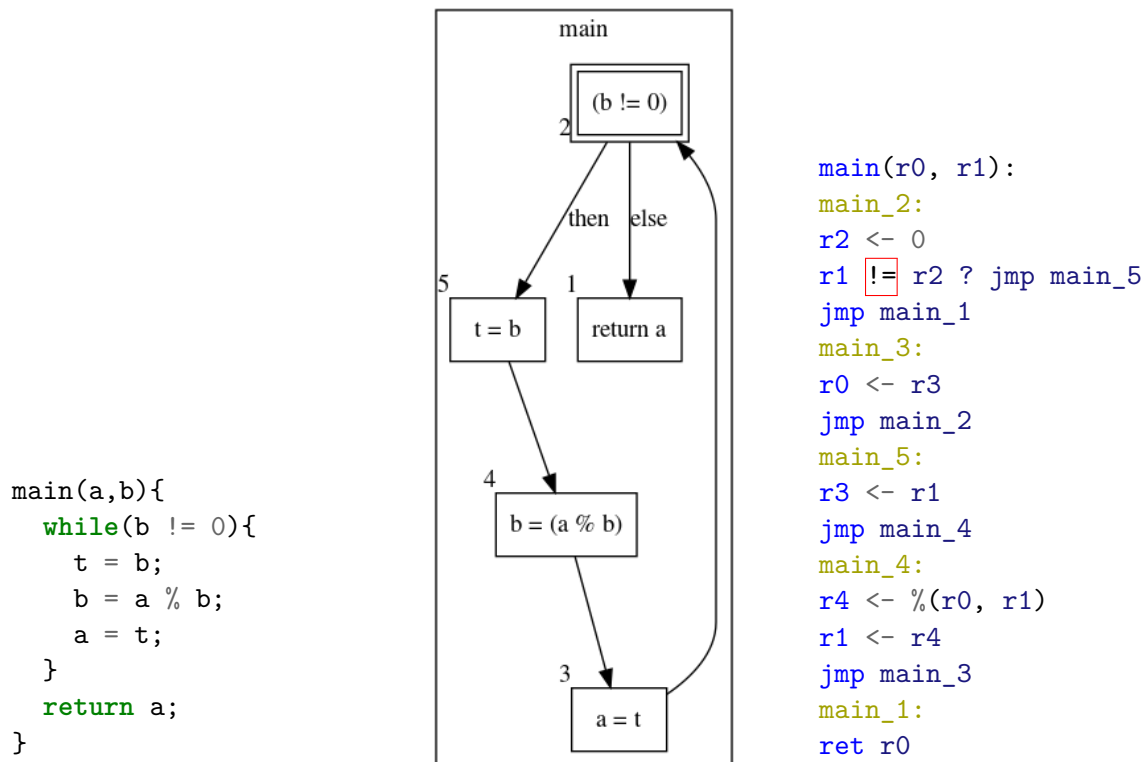


FIGURE 1 – Aperçu de la structure du compilateur

## 3 Tests

Vous trouverez dans le répertoire `tests` un ensemble d'outils vous permettant de tester votre compilateur. Les dossiers `array`, `basic`, `funcall`, `real_args`, `strings` et `struct` 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.



(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

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. Le nom des programmes sont cliquables pour avoir plus d'informations sur le déroulement de leur compilation.

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>

- `List.mem (e: 'a) (l: 'a list): bool`  
retourne vrai si `e` est dans la liste `l`
- `List.fold_left (f: 'a -> 'b -> 'a) (acc: 'a) (l: 'b list): 'a`  
applique `f` à chaque élément de `l` en stockant le résultat dans `acc`
- `List.map (f: 'a -> 'b) (l: 'a list): 'b list`  
retourne une liste dans laquelle `f` a été appliquée à chaque élément de `l`.
- `List.filter_map (f: 'a -> 'b option) (l: 'a list): 'b list`  
Comme `List.map`. De plus, les éléments pour lesquels `f` retourne `None` sont retirés de la liste retournée.
- `Set` les mêmes fonctions existent pour les ensembles `Set`
- `Set.union (s1: Set.t) (s2: Set.t): Set.t`  
retourne l'union des ensembles `s1` et `s2`
- `Set.add (e: elt) (s: Set.t): Set.t`  
ajoute l'élément `e` à l'ensemble `s`
- `Hashtbl.find_opt (tbl: ('a, 'b) Hashtbl.t) (k: 'a) -> 'b option`  
retourne l'élément associé à la clé `k`. Si cette clé n'existe pas dans `tbl` retourne `None`.
- `Hashtbl.find_replace (tbl: ('a, 'b) Hashtbl.t) (k: 'a) (v: 'b) : unit`  
associe la valeur `v` à la clé `k` dans la table `tbl`.

### 4.2 Tests

Pour compiler et tester votre code 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 :

```
$ cat tests/basic/just_a_variable_37.e
main(){
  just_a_variable = 37;
  return just_a_variable;
}
```

```
# Au début du TP dans results.html:
Lexing error:
Lexer failed to recognize string starting with
↳ 'main(){
  just_a_var'
```

```
# 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
```

Lorsque vous appelez `make test`, votre compilateur est lancé sur 30 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 Débogage

Pour vous aider à déboguer, les fonctions suivantes sont à votre disposition :

- `nfa_to_string` et `dfa_to_string` transforment des `nfa` et `dfa` en chaînes de caractères (`string`)
- `nfa_to_dot` et `dfa_to_dot` construisent une représentation visuelle d'un `dfa` dans un fichier `*.dot`. Le fichier `*.dot` pourra ensuite être transformé en image via la commande `$ dot fichier.dot -Tsvg -o fichier.svg`.

Vous trouverez des exemples d'utilisations de ces fonctions dans le fichier `src/test_lexer.ml`. Ce fichier contient ce qui s'apparente à une fonction `main` en C : une déclaration de fonction `let () = ...`. Dans cette fonction, une liste d'expressions régulières est créée, affichée, transformée en NFA. Ce NFA est affiché avec `nfa_to_string` d'une part et `nfa_to_dot` d'autre part, ce qui génère un fichier `/tmp/nfa.dot`, que vous pouvez convertir en image SVG avec la commande suivante :

```
$ dot -Tsvg /tmp/nfa.dot -o /tmp/nfa.svg
```

Ou bien en utilisant un convertisseur en ligne, ici par exemple : <https://dreampuf.github.io/GraphvizOnline>. Le NFA est ensuite déterminisé, le DFA résultant est affiché puis écrit dans

/tmp/dfa.dot.

Pour lancer ces tests, il vous suffit de lancer la commande `make test_lexer` depuis le répertoire `src`.

## 4.4 Travail à effectuer

Le développement de notre analyseur lexical se déroule en trois étapes. Premièrement, la spécification des expressions régulières permettant de reconnaître les termes du langage E. Ensuite, un NFA (*Non-deterministic Finite Automaton*) pourra être généré pour ces expressions régulières. 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_regexp.ml`.

### 4.4.1 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.

**Question 4.1.** Compléter la fonction `list_regexp` du fichier `src/e_regexp.ml` en remplaçant les regex `Eps` par une expression régulière adéquate. À noter que les variantes `regex` sont définies plus haut dans le fichier et que la liste de symboles est disponible dans le fichier `src/symbols.ml`.

### 4.4.2 Expressions régulières en NFAs

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

**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.*

**Question 4.3.** Compléter la fonction `nfa_of_regexp` qui produit un NFA à partir d'une expression régulière. Les cas `Eps` et `Charset c` sont donnés en exemple. Traiter les variantes restantes de `regexp`.

### 4.4.3 Détermination d'un NFA en DFA

Cette partie se charge de transformer un NFA en DFA 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 les fonctions `epsilon_closure` et `epsilon_closure_set` qui retournent les états accessibles par  $\varepsilon$ -transitions d'un état ou d'un ensemble d'états d'un graphe `nfa`. Plus particulièrement, il faut écrire la fonction récursive `traversal` pour `epsilon_closure`.

**Question 4.5.** 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.

**Question 4.6.** 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`.

**Question 4.7.** Pour finir la construction de notre DFA, compléter la fonction de transition `make_dfa_step` en utilisant la table de transition construite précédemment avec `build_dfa_table`.

#### 4.4.4 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 4.8.** 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.

*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 existe aussi pour vos camarades de 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	<a href="#">SYM_IDENTIFIER</a> <a href="#">SYM_INTEGER</a>
EXPR	<a href="#">SYM_IDENTIFIER</a> <a href="#">SYM_INTEGER</a>
INTEGER	<a href="#">SYM_INTEGER</a>
IDENTIFIER	<a href="#">SYM_IDENTIFIER</a>

(b) La relation First

#### Table LL

	SYM_EOF	SYM_IDENTIFIER	SYM_INTEGER	SYM_PLUS	SYM_ASTERISK
S		<a href="#">1</a>	<a href="#">1</a>		
EXPR		<a href="#">2</a> <a href="#">4</a> <a href="#">5</a> <a href="#">3</a> <a href="#">4</a> <a href="#">5</a>			
INTEGER			<a href="#">6</a>		
IDENTIFIER		<a href="#">7</a>			

(c) La table LL

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

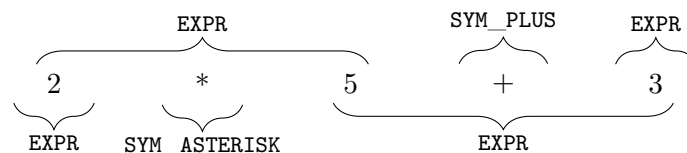
On peut donc voir (Figure 3a) la grammaire que l'on a écrite, où chaque règle est numérotée, et chaque non-terminal de la grammaire est un lien vers l'ensemble des règles associées à ce non-

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> SYM_EOF
(2) EXPR	-> <u>TERM</u> <u>TERMS</u>
(3) TERM	-> <u>FACTOR</u> <u>FACTORS</u>
(4) TERMS	-> SYM_PLUS <u>TERM</u> <u>TERMS</u>
(5)	-> $\epsilon$
(6) FACTOR	-> <u>IDENTIFIER</u>
(7)	-> <u>INTEGER</u>
(8) FACTORS	-> SYM_ASTERISK <u>FACTOR</u> <u>FACTORS</u>
(9)	-> $\epsilon$
(10) INTEGER	-> SYM_INTEGER
(11) IDENTIFIER	-> SYM_IDENTIFIER

(a) La grammaire (clicable)

## 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 `symbols` 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 les fichiers `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.

**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).

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 :

- 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. 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 à l'objet C renvoyé par le i-ième élément de la règle. Concrètement, dans la règle :

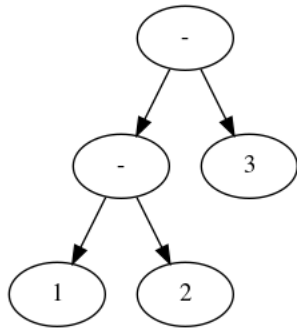
```
TERMS -> SYM_PLUS TERM TERMS { (Tadd, $2) :: $3 }
```

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
])

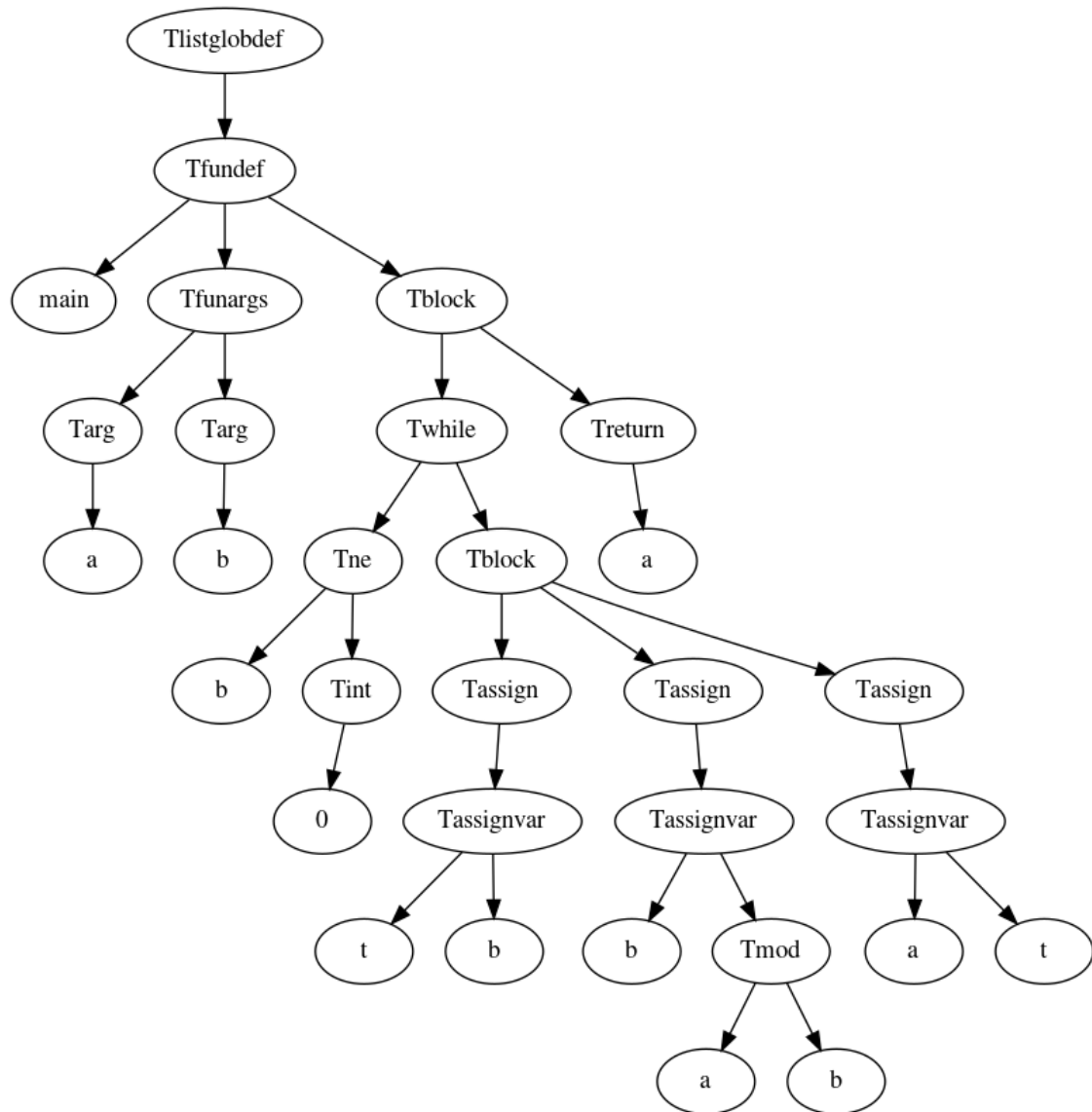
```

**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 document le type des arbres de syntaxe abstraite.

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` ;
- que tout le compilateur soit recompilé (wow!) ;
- que 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 :



## 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, présenté en Section ??, 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.<sup>1</sup>

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

1. 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`).

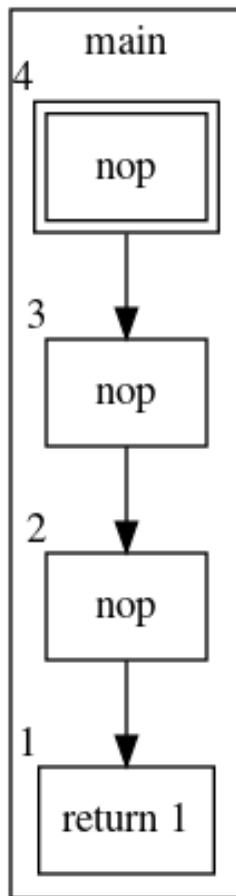
**Question 7.5.** Écrire une fonction `dead_assign_elimination_fun (f: cfg_fun) : cfg_fun` qui, étant donnée une fonction de CFG, calcule la vivacité des variables et élimine les affectations mortes.

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`.

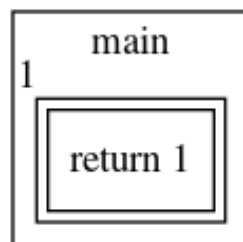
Sur l'exemple ci-contre :

```

— 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", 1)])`.

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)], 1, [("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.

## A Installation des dépendances

Dès le début, vous aurez besoin d'OCAML.

```
# Install opam
$ sudo apt install opam # ou avec votre gestionnaire de paquets favori
$ opam --version
2.0.4 # Assurez-vous d'avoir au moins opam version 2.0...
$ opam init
$ opam switch install 4.08.0 (ou plus)
# Installation des dépendances
$ opam install stdlib-shims ocamlbuild ocamlfind menhir lwt logs batteries yojson websocket
↪ websocket-lwt-unix
$ eval $(opam env)
```

Pour assembler un programme RISC-V, vous aurez besoin d'outils spécifiques :

- `riscv64-unknown-elf-gcc` : un *cross-compileur* pour RISC-V pour générer des exécutables RISC-V,
- `qemu-riscv64` : un émulateur de RISC-V pour les exécuter.

Sur des Debian (10+) ou Ubuntu (19.04+) :

```
$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu
↪ binutils-riscv64-linux-gnu
```

Sur ArchLinux :

```
$ sudo pacman -S riscv64-linux-gnu-binutils riscv64-linux-gnu-gcc riscv64-linux-gnu-gdb
↪ qemu-arch-extra
```

Si vous devez compiler vous-mêmes : (attention c'est long!)

```
# GCC:
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
# install dependencies
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk
↪ build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
# configure and build (vous pouvez changer le préfixe, c'est ici que seront installés les outils)
$ cd riscv-gnu-toolchain
$ ./configure --prefix=/usr/local
$ sudo make

# QEMU:
$ wget https://download.qemu.org/qemu-4.1.0.tar.xz
$ tar xf qemu-4.1.0.tar.xz
$ cd qemu-4.1.0
$ ./configure --disable-kvm --disable-werror --prefix=/usr/local --target-list="riscv64-softmmu"
$ make
$ sudo make install
```

## B Merlin, l'assistant de programmation OCaml

Merlin est un assistant à la programmation OCaml pouvant s'interfacer avec de nombreux éditeurs de code tels que Emacs, Vim, Atom, Visual Studio Code, Sublime Text ... Vous pouvez trouver le dépôt Github de Merlin à l'URL suivante : <https://github.com/ocaml/merlin>.

### B.1 Installation de Merlin

Avec Opam :

```
$ opam install merlin
$ opam user-setup install # configure Emacs et Vim pour Merlin
```

Manuellement : Instructions et sources sur le dépôt Github

### B.2 Merlin et Visual Studio Code

Les instructions sont tirés du blog suivant : <https://www.cosmiccode.blog/blog/vscode-for-ocaml/>

```
1. $ opam install merlin
   $ opam install ocp-indent # Outil d'indentation de code OCaml
```

2. Dans VSCode chercher et installer l'extension *OCaml and Reason IDE*

#### Remarques :

- **VSCode ne trouve pas les modules définis dans d'autres fichiers.**  
Pour que VSCode reconnaisse les modules provenant des autres fichiers, il faut au préalable avoir compilé ces modules. Pour cela, exécutez la commande `make` dans la source du projet. Les modules compilés seront stockés dans le dossier `src/_build`.
- **Si les outils OCaml ne sont pas accessibles globalement.**  
VSCode n'arrivera pas à trouver les outils de l'environnement OCaml-Merlin, il est possible de les spécifier de la manière suivante
  - `Ctrl+P > Workspace Settings`
  - Extensions → Reason configuration
  - Spécifier les chemins vers les binaires `ocamlmerlin`, `ocamlfind`, `ocp-indent` et `opam`