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. 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. Les sections 4 à 9 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/supecomp`.

2 Organisation du compilateur et langages intermédiaires

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 en un programme E, puis en un programme CFG, ensuite un programme RTL et finalement un programme Assembleur. Chacun de ces langages intermédiaire est détaillé ci-dessous, et est illustré sur l'exemple de la Figure 2a.

2.1 Quelques structures de données

Un certain nombre de structures de données sont définies pour vous (et utilisées dans le squelette fourni) dans les fichiers `datatypes.h` et `datatypes.c`.

La structure la plus importante est la liste, et quelques fonctions associées :

```
typedef struct list {
  void* elt;
  struct list* next;
} list;

list* list_append(list*, void*);
```

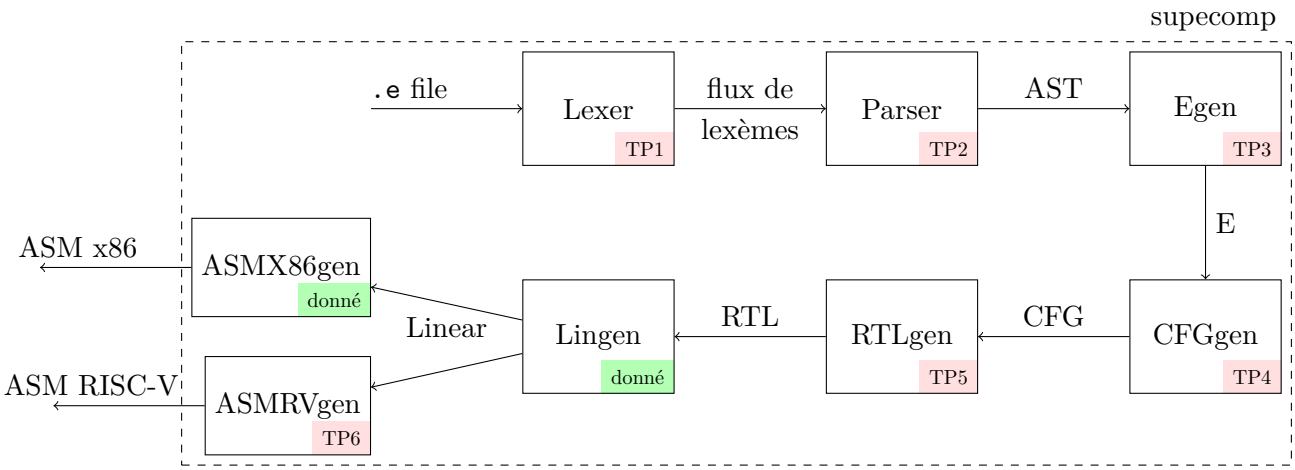


FIGURE 1 – Aperçu de la structure du compilateur

```
list* cons(void*, list*);
list* concat(list *l1, list* l2);
int list_length(list* l);
void* list_nth(list* l, int n);
```

La structure est une liste chaînée classique, similaire à ce que vous avez dû écrire dans un des BE de C en début d'année. Une liste est composée d'un élément `elt` (de type `void*` – la généricité du langage C !) et d'un pointeur vers le reste de la liste. La liste vide est simplement le pointeur `NULL`.

La fonction `list_append(l, e)` ajoute l'élément `e` en queue de la liste `l` et retourne la nouvelle liste. Il s'agit du même pointeur que la liste originale si elle n'était pas vide, d'un nouveau pointeur sinon. Les fonctions `list_append_int` et `list_append_string` sont des versions spécialisées de `list_append`. Le seul rôle de ces fonctions est d'effectuer le transtypage (*cast*) vers `void*` et rendre le code plus lisible.

La fonction `cons(e,l)` ajoute l'élément `e` en tête de la liste `l` et renvoie la nouvelle liste. Les fonctions `cons_int` et `cons_string` sont des versions spécialisées de `cons`. Le seul rôle de ces fonctions est d'effectuer le transtypage (*cast*) vers `void*` et rendre le code plus lisible.

La fonction `concat(l1,l2)` concatène les deux listes passées en paramètre. Attention, la liste `l1` n'est pas préservée ! (Son dernier pointeur `next` est modifié.)

La fonction `list_length(l)` retourne la longueur de la liste passée en paramètre, et la fonction `list_nth(l, n)` renvoie le `n`-ième élément de la liste `l`.

La fonction `free_list(l)` libère l'espace utilisé par une liste. Attention, si certains éléments de la liste doivent être libérés (des chaînes de caractères, ou des structures de données plus compliquées), cette fonction ne s'en charge pas, c'est à vous de faire attention !

Nous vous fournissons aussi les fonctions `print_int_list()` et `print_string_list()` qui font ce que vous pensez qu'elles font.

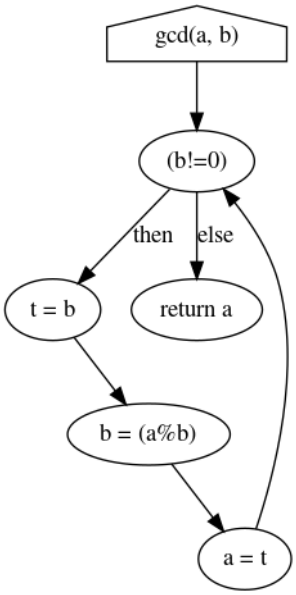
Les fonctions `list_in_int(l, x)` et `list_in_string(l, x)` renvoient vrai si et seulement si `x` appartient à la liste `l`.

Les fonctions `list_int_incl(l1, l2)` et `list_string_incl(l1, l2)` renvoient vrai si et seulement si la liste `l1` est incluse dans la liste `l2`, *i.e.* si tous les éléments de `l1` sont aussi dans `l2`.

Les fonctions `int_list_copy(l)` et `string_list_copy(l)` renvoient une copie de la liste `l`.

```
gcd(a,b){
  while(b != 0){
    t = b;
    b = a % b;
    a = t;
  }
  print a;
  return a;
}
```

(a) Programme E



(b) CFG correspondant

```
gcd(r0, r1):
goto n2
n2:
r3 <- 0
r2 <- r1 <> r3
r2 ? goto n5 : goto n1
n5:
r4 <- r1
goto n4
n4:
r5 <- r0 % r1
r1 <- r5
goto n3
n3:
r0 <- r4
goto n2
n1:
return r0
```

(c) Programme RTL

```
.global supecomp_main
supecomp_main:
push ebp
mov ebp, esp
sub esp, 20
.n2:
mov DWORD PTR [ebp+-12],0
xor ebx, ebx
mov ecx,DWORD PTR [ebp+12]
mov edx,DWORD PTR [ebp+-12]
cmp ecx, edx

setne bl
mov DWORD PTR [ebp+-8],ebx
mov eax,DWORD PTR [ebp+-8]
test eax, eax
jnz .n5
jmp .n1
.n1:
mov eax,DWORD PTR [ebp+8]
jmp .ret
.n5:
mov ebx,DWORD PTR [ebp+12]
mov DWORD PTR [ebp+-16],ebx
jmp .n4
.n4:
mov eax,DWORD PTR [ebp+8]
mov ebx,DWORD PTR [ebp+12]
mov edx, 0
div ebx
mov DWORD PTR [ebp+-4], edx
mov ebx,DWORD PTR [ebp+-4]
mov DWORD PTR [ebp+12],ebx
jmp .n3
.n3:
mov ebx,DWORD PTR [ebp+-16]
mov DWORD PTR [ebp+8],ebx
jmp .n2
.ret: leave
ret
```

(d) Assembleur x86-32

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

Les fonctions `list_remove_int(l, x)` et `list_remove_string(l, x)` suppriment l'élément `x` de la liste `l`.

La fonction `some(e)` encapsule l'élément `e` : on alloue de l'espace et on y stocke la valeur de `e`. Cela est utile pour simuler le type `option` (a.k.a. `Maybe`, a.k.a `std::optional`) que l'on trouve dans les vrais langages de programmation (OCaml, Haskell, Java, Rust, même C++). L'absence de valeur est encodée par le pointeur `NULL`.

Une autre structure définie dans ce fichier est la paire :

```
typedef struct pair {
  void* fst;
  void* snd;
} pair;
```

Les paires sont notamment utilisées pour créer des *listes d'association*. Ce sont des listes de paires `[(k1,v1);(k2,v2);...;(kn,vn)]` qui permettent d'implémenter simplement une structure qui associent des clés `ki` à des valeurs `vi`.

La fonction `assoc_set(l, k, v)` ajoute une nouvelle correspondance entre la clé `k` et la valeur `v`, et retourne la nouvelle liste d'association.

De manière symmétrique, `assoc_get(l, k)` renvoie la valeur associée à la clé `k` dans la liste `l`, encapsulée grâce à `some`. Si aucune telle valeur n'existe, un pointeur nul est renvoyé.

2.2 Arbre de syntaxe abstraite (AST)

Le type d'ASTs que vous devrez générer est donné dans le fichier `ast.h`. Un nœud d'un tel arbre est représenté par la structure `ast_node`, dont la définition est redonnée ci-dessous :

```
struct ast_node {
  enum ast_tag tag;
  union {
    int integer;
    char* string;
    struct list* children;
  };
};
```

Un nœud est composé d'un tag, qui identifie le type de ce nœud. Il peut s'agir d'une feuille de type entier (identifiée par le tag `AST_INT`), d'une feuille de type chaîne de caractères (identifiée par le tag `AST_STRING`), ou bien d'un nœud composé d'une liste d'enfants (n'importe quel autre tag). Dans le dernier cas, chaque élément de la liste est un pointeur vers un autre nœud. L'AST que vous obtiendrez pour le programme de l'exemple de la Figure 2a est donné en Figure 3.

2.3 Le langage E

Un exemple de programme E est donné en Figure 2a. La syntaxe est inspirée du C et devrait vous être familière. Ce programme calcule le PGCD de deux entiers passés en paramètre, affiche ce PGCD et retourne sa valeur.

La syntaxe des programmes E est définie dans le fichier d'en-têtes `elang.h`. Un programme E consiste en une déclaration de fonction, comme définie ci-dessous :

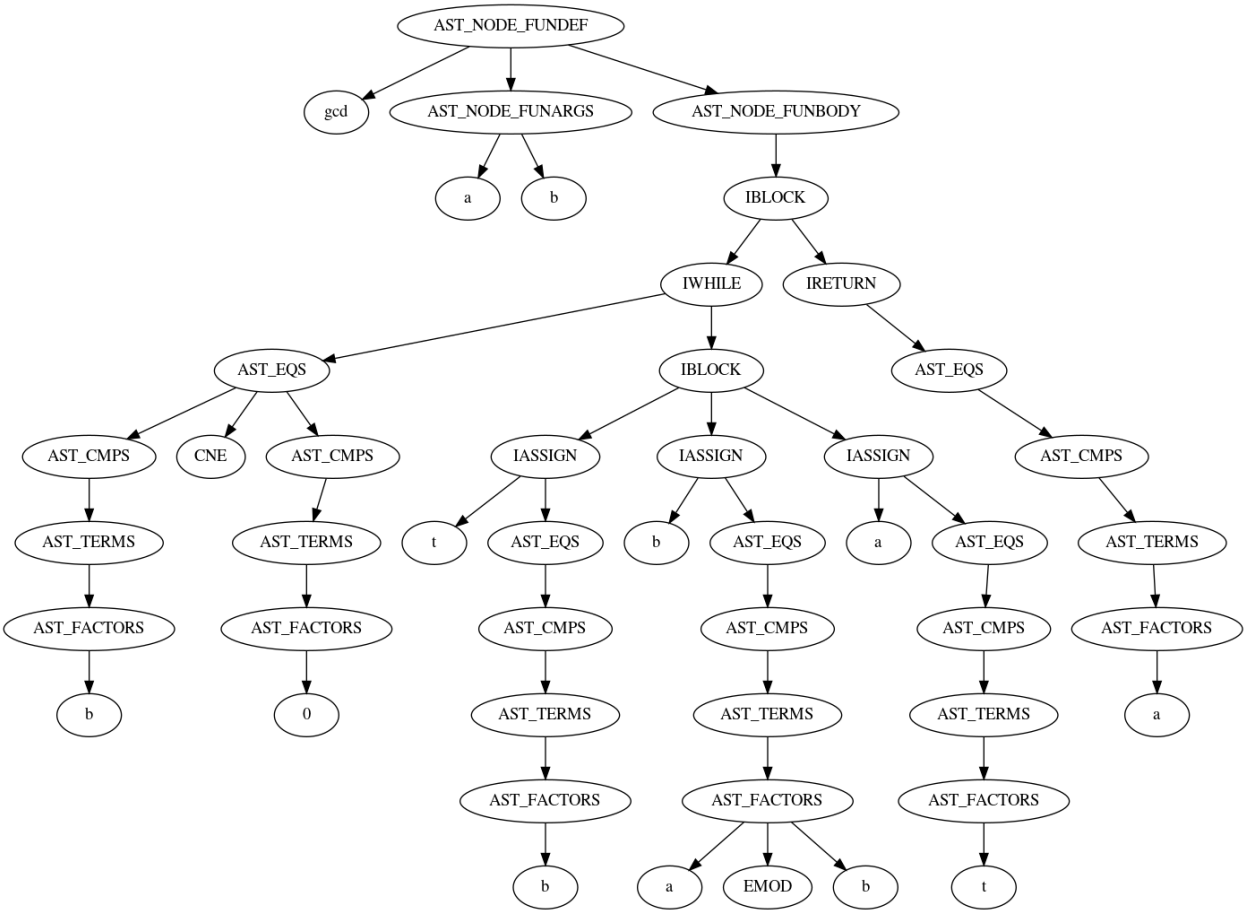


FIGURE 3 – Exemple d'arbre de syntaxe abstraite.

```
struct eprog {
  char* fname;
  list* args;
  struct instruction* body;
};
```

Une déclaration de fonction comporte le nom de la fonction (`fname`), la liste des arguments (`args`) qui est une liste de chaînes de caractères, ainsi que le corps de la fonction `body`. Dans notre exemple, la fonction s'appelle `gcd` et prend deux paramètres `a` et `b`. Il n'y a pas de déclaration de type dans le langage E : toutes les variables sont de type entier.

Les instructions, définies par la structure `instruction` et identifiées par les tags `instr_t` peuvent être de 6 types différents.

```
enum instr_t { IIFTHENELSE, IWHILE, IASSIGN,
               IRETURN, IPRINT, IBLOCK
};
struct instruction {
  enum instr_t type;
  union {
    struct instr_if iif;
    struct instr_while iwhile;
    struct instr_assign iassign;
    struct instr_return ireturn;
    struct instr_print iprint;
    struct instr_block iblock;
  };
};
```

`IASSIGN` : une affectation `v = e;` où `v` est un identifiant de variable et `e` est une expression.

`IWHILE` : une boucle `while(e) i` où `e` est une expression et `i` est une instruction que l'on doit exécuter tant que la valeur de `e` est non-nulle.

`IIFTHENELSE` : une conditionnelle `if(e) i1 [else i2]`, où `e` est une expression et `i1` et `i2` sont les instructions à exécuter si `e` est non-nul ou nul, respectivement. La branche `else` est facultative.

`IRETURN` : un retour de fonction `return e;` où `e` est l'expression dont on retourne la valeur.

`IPRINT` : un affichage `print e;` où `e` est l'expression dont on affiche la valeur.

`IBLOCK` : une liste d'instructions `{ i1 i2 ... }` où `i1`, `i2`, ... sont des instructions à exécuter les unes à la suite des autres.

Les expressions manipulées sont de 4 types différents :

```
enum expr_t { EINT, EVAR, EUNOP, EBINOP };

struct expression {
  enum expr_t etype;
  union {
    struct expr_unop unop;
    struct expr_int eint;
    struct expr_binop binop;
```

```
    struct expr_var var;
  };
};
```

- un identifiant de variable, par exemple `a`, `B32` ou `x34_b12`.
- un entier, par exemple `12`, `-15` ou `948534` ;
- une opération unaire sur une expression : `-e`. On ne considère que la négation d'un entier pour le moment.
- une opération binaire : `e1 + e2`, `e1 - e2`, `e1 * e2`, `e1 / e2`, `e1 % e2`, qui correspondent respectivement à l'addition, la soustraction, la multiplication, la division et le modulo de deux expressions.

2.4 Le langage CFG

Le langage CFG (*Control Flow Graph*, voir `cfg.h`) est une autre représentation des programmes E, sous la forme d'un graphe de flot de contrôle. Les expressions manipulées sont les mêmes que dans le langage E. Le graphe est un ensemble de nœuds associés à 0, 1 ou 2 successeurs. Le type du graphe, en C, est donné par la structure suivante :

```
typedef struct cfg {
  struct cfg* next;
  int id;
  node_t* node;
} cfg;

typedef struct node_t {
  enum nodetype type;
  union {
    struct node_assign assign;
    struct node_print print;
    struct node_return ret;
    struct node_cond cond;
    int goto_succ;
  };
} node_t;
struct node_cond {
  struct expression* cond;
  int succ1, succ2;
};
```

Un graphe est une liste de `struct cfg`, chacun associant un identifant entier `id` à un nœud `node`. Ce type de structure de listes est très commun en C ; et dans le squelette qui vous est fourni.

Les nœuds du graphe (`node_t`) peuvent être de 5 types différents (`enum nodetype`) :

`NODE_ASSIGN` : une affectation `v = e;`. Ce nœud a 1 successeur.

`NODE_RETURN` : un retour de fonction `return e;`. Ce nœud n'a pas de successeur.

`NODE_PRINT` : un affichage `print e;`. Ce nœud a 1 successeur.

`NODE_COND` : une condition `e` et 2 successeurs : un pour le cas où `e` est vrai (`succ1`), l'autre pour le cas où `e` est faux (`succ2`).

`NODE_GOTO` : un saut inconditionnel. Ce nœud a 1 successeur.

Par exemple, le programme E présenté en Figure 2a devient le programme CFG dessiné en Figure 2b.

2.5 Le langage RTL

Le langage RTL (voir `rtl.h`) est également un graphe, mais les expressions complexes et les identifiants de variables disparaissent, pour laisser la place à des registres (en nombre non borné) et des opérations élémentaires sur ces registres.

Chaque nœud du graphe contient une liste d'opérations RTL, qui peuvent être :

`RMOV` : une copie de registres *e.g.* `r0 <- r1`

`RIMM` : une valeur immédiate *e.g.* `r0 <- 18`

`RPRINT` : l'affichage d'un registre *e.g.* `print r8`

`RUNOP` : une opération unaire *e.g.* `r0 <- - r1`

`RBINOP` : une opération binaire *e.g.* `r0 <- r1 + r2`

`RRET` : un retour de fonction *e.g.* `ret r3`

`RBRANCH` : un branchement conditionnel *e.g.* `r1 ? goto n3 : goto n4` (si `r1` vaut vrai, aller au nœud `n3` sinon aller au nœud `n4`)

`RGOTO` : un branchement non conditionnel *e.g.* `goto n5`

Par exemple, le programme E présenté en Figure 2a devient le programme RTL de la Figure 2c. On y voit que les variables `a`, `b` et `t` ont été transformées en des registres `r0`, `r1` et `r4`, respectivement. Des registres intermédiaires (`r2`, `r3` et `r5`) ont été utilisés pour les étapes intermédiaires (valeurs immédiates, copies...). L'utilisation des registres n'est pas optimale : on pourrait faire avec moins de registres. Ce travail d'optimisation sera l'objet de la phase d'allocation de registres.

2.6 Le langage Linear

Le langage Linear (`linear.h`) est très similaire au langage RTL. Les opérations disponibles sont les mêmes. La liste des opérations contenues dans le graphe RTL est linéarisée via un tri topologique du graphe [1]. Le programme est donc simplement une liste d'opérations.

3 Tests

Vous trouverez dans le répertoire `tests` un ensemble de fichiers de tests, sur lesquels vous pouvez tester votre compilateur.

Pour chaque fichier `test.e`, nous vous avons fourni la sortie attendue avec les paramètres 12 et 14 dans `test.e.expect_12_14` et avec les paramètres 1 et 5 dans `test.e.expect_1_5`. Vous pouvez tester que votre compilateur est conforme à ce qui est attendu en lançant `make test` depuis la racine de votre projet.

Initialement, tous les tests devraient rendre KO. Au fur et à mesure des séances de TPs, de plus en plus de cellules devraient contenir OK, indiquant que la sortie de votre compilateur est correcte.

1. `https://fr.wikipedia.org/wiki/Tri_topologique`

Pour chaque langage intermédiaire, vous trouverez deux colonnes correspondant au lancement du programme via l'interpréteur de ce langage intermédiaire avec les deux jeux de paramètres correspondant aux fichiers `.expect_xx_xx` fournis.

4 TP1 : Analyseur lexical

Le but de cette séance de TP est de réaliser un analyseur lexical pour un langage appelé 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 Description des types et fonctions présents dans le squelette

L'essentiel du travail que vous réaliserez au cours de cette séance se déroulera dans le fichier `lexer.c`. Les déclarations présentes dans le fichier `lexer.h` vous donnent les types des symboles (*a.k.a* tokens, *a.k.a.* lexèmes) que vous allez générer. Le type `symbol_t` est une énumération des différents types de symboles que l'on va rencontrer dans des programmes E. Le type `symb` est une structure qui contient un tag de type `symbol_t` et une chaîne de caractères qui vient éventuellement compléter ce symbole. En particulier, lorsque le tag vaut `SYM_IDENTIFIER`, le champ `id` contiendra l'identifiant en question ; et lorsque le tag vaut `SYM_INTEGER`, le champ `id` contient la **chaîne de caractères** représentant l'entier en question.

Le type `lexer_state`, comme son nom l'indique, représente l'état de l'analyseur lexical. Il contient un descripteur de fichier, la position dans le fichier à laquelle on va lire le prochain caractère, le numéro de la ligne courante, ainsi que le dernier symbol lu. Cet état est initialisé par la fonction `init_lexer_state`, à la position 0, au numéro de ligne 1, et avec le symbole `SYM_EOF` qui représente la fin du fichier.

La fonction `get_character(lex)` lit le caractère à la position `lex->curpos` dans le fichier `lex->lexer_fd`. Cette fonction ne modifie pas `lex` ni le descripteur de fichier : plusieurs appels successifs à cette fonction renverront donc le même caractère. En revanche, la fonction `next_character(lex)` ne renvoie rien mais fait avancer le lexer d'une position. L'appel suivant à `get_character` renverra donc le caractère suivant. Il est possible que vous ayez besoin de revenir en arrière dans le fichier : la fonction `prev_character` remplit ce rôle.[2] L'implantation de ces fonctions est donnée dans le fichier `lexer.c` et peut éclaircir cette description.

4.2 Fonctions utiles de la librairie standard C

Vous aurez besoin d'un certain nombre de fonctions de la librairie C, que vous connaissez déjà peut-être. Une description sommaire vous est donnée ici, plus de détails dans le manuel (*e.g.* `man strcmp`).

- `#include <string.h>`
  - `int strcmp(const char *s1, const char *s2);` compare deux chaînes de caractères. Renvoie 0 si les chaînes sont égales. Pour tester l'égalité de deux chaînes, on pourra utiliser l'idiome suivant :

    ```
    if(!strcmp(s, "blabla")){
      // chaines egales
    }
    ```

  - `char *strdup(const char *s);` crée une copie (**dup**lique) une chaîne de caractères.
- `#include <ctype.h>`

2. Vous ne devriez avoir à utiliser cette fonction qu'une seule fois, au plus.

  - `isalpha`, `isdigit`, `isalnum`...
- `#include <stdlib.h>`
  - `malloc`, `free` : vous connaissez, non ?

4.3 Tests

La fonction principale du squelette qui vous est fourni se trouve dans le fichier `main_supecomp.c`. Son travail consiste principalement à récupérer les options du compilateur sur la ligne de commande. Vous pouvez lancer le compilateur sur les tests présents dans le répertoire `tests`. Pour le moment, tous devraient provoquer une erreur lexicale. Vous pouvez utiliser ces tests pour vérifier que vous n'avez pas oublié de reconnaître certains symboles. (L'exhaustivité du jeu de tests, en matière de lexèmes rencontrés, n'est absolument pas garantie. Nous nous dégageons de toute responsabilité en cas d'absence d'un lexème de ce jeu de tests.)

La fonction `main` appelle la fonction `parse_file` du fichier `parser.c`. Pour le moment, cette fonction ne fait pas d'analyse syntaxique (ce sera l'objet du TP suivant), mais se contente d'appeler le lexer et d'afficher chacun des symboles produits.

Résultats attendus sur un exemple :

```
$ cat tests/test06.e
main(){
x = 5;
y = x * 2;
x = 6 * x;
return x - y;
}
```

```
# Au début du TP :
$ expr/supecomp tests/test06.e
./squelette/supecomp: syntax error in tests/test06.e
↪  in line 1: found unknown character m
```

```
# À la fin du TP:
$ expr/supecomp tests/test06.e
Found symbol 'main'
Found symbol '('
Found symbol ')'
Found symbol '{'
Found symbol 'x'
Found symbol '='
Found symbol '5'
Found symbol ';'
Found symbol 'y'
Found symbol '='
Found symbol 'x'
Found symbol '*'
Found symbol '2'
Found symbol ';'
Found symbol 'x'
Found symbol '='
Found symbol '6'
Found symbol '*'
Found symbol 'x'
Found symbol ';'
Found symbol 'return'
Found symbol 'x'
Found symbol '-'
Found symbol 'y'
Found symbol ';'
Found symbol '}'
```

4.4 Travail à effectuer

Question 4.1. Écrire une fonction `find_next_character` qui, étant donné un `lexer_state lex`, fait avancer la position dans le fichier jusqu'au prochain caractère significatif (*i.e.* pas une

espace ou une tabulation, ou un retour à la ligne...). Ainsi, la fonction `get_character` devra retourner le caractère significatif.

Question 4.2. Écrire la fonction `next_symbol` qui lit le prochain symbole à partir d'un `lexer_state lex`. La position du curseur sera mise à jour (pour pointer après le symbole lu), et le symbol lui-même (`lex->symbol`) sera mis à jour. Quelques cas sont présentés pour vous guider.

Question 4.3 (Bonus)**.** Faites en sorte d'ignorer les commentaires. Nous suggérons de modifier la fonction `find_next_character()` pour ceci. On voudrait supporter les deux types de commentaires suivant :

— commentaires multi-lignes *à la C* :

```
/* Ceci est un commentaire
sur plusieurs
lignes */
```

— commentaire sur une ligne *à la C++* :

```
x = y + z; // Un commentaire qui dit qu'on fait une addition
```

5 TP2 : Analyseur syntaxique

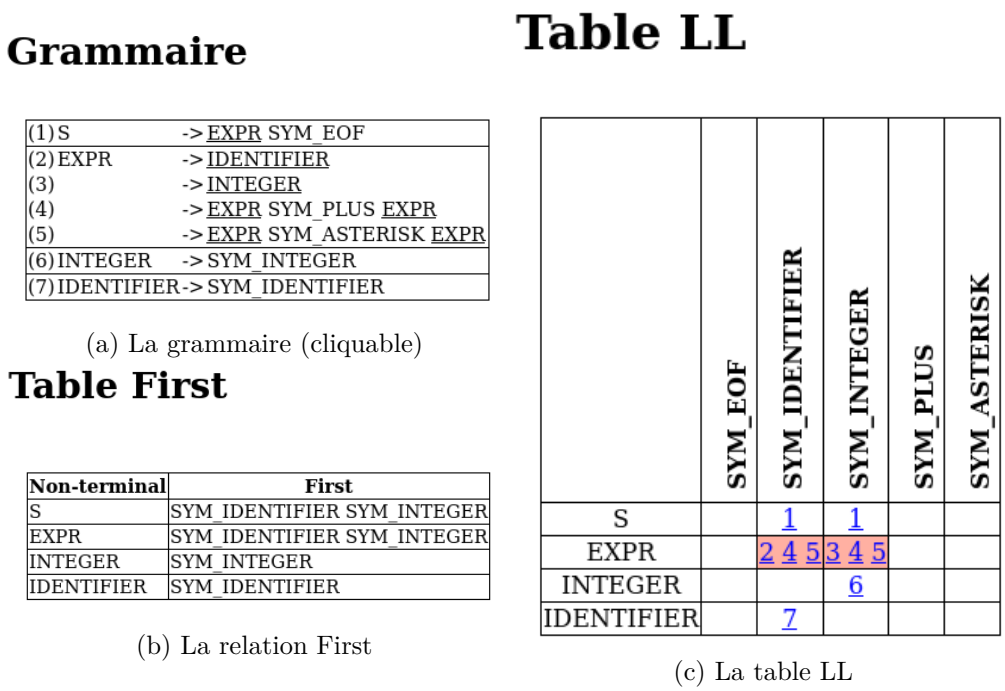
Lors du TP précédent, vous avez écrit un analyseur lexical pour le langage E, et avez donc obtenu, à partir d'un fichier source `.e` un flux de lexèmes. Le but de ce TP est de construire un analyseur syntaxique. Pour ce faire, vous allez devoir écrire la grammaire du langage E dans un format spécifique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

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 C 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 4, le fichier généré par une toute petite grammaire.



Grammaire

(1)	S	-> EXPR SYM_EOF
(2)	EXPR	-> IDENTIFIER
(3)		-> INTEGER
(4)		-> EXPR SYM_PLUS EXPR
(5)		-> EXPR SYM_ASTERISK EXPR
(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		1	1		
EXPR		2 4 5	3 4 5		
INTEGER			6		
IDENTIFIER		7			

(c) La table LL

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

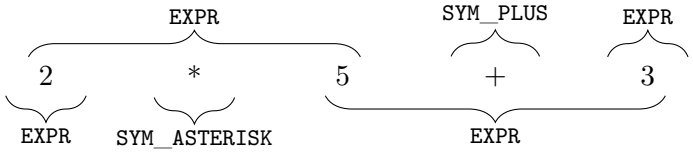
On peut donc (Figure 4a) voir la grammaire que l'on a écrite, où chaque règle est numérotée, et chaque non-terminal de la grammaire est un lien vers l'ensemble des règles associées à ce non-terminal. Cela paraît anecdotique pour cette toute petite grammaire, mais votre grammaire ne

tiendra pas sur un seul écran et cette fonctionnalité sera alors très appréciée.

La Figure 4b 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.

Finalement, la Figure 4c montre la table 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 ambigüe.

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 la multiplication est plus prioritaire que l'addition 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 5. Dans la table LL (Figure 5c), 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 rogue correspondent aux règles qui peuvent être appliquées lorsque le nonterminal 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 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 SYM_INTEGER SYM_PLUS SYM_ASTERISK
non-terminals<void*> S EXPR TERM TERMS FACTOR FACTORS INTEGER IDENTIFIER
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
FACTOR -> INTEGER
FACTORS -> SYM_ASTERISK FACTOR FACTORS
FACTORS ->
```

Grammaire

(1)	S	-> EXPR SYM_EOF
(2)	EXPR	-> TERM TERMS
(3)	TERM	-> FACTOR FACTORS
(4)	TERMS	-> SYM_PLUS TERM TERMS
(5)		-> ε
(6)	FACTOR	-> IDENTIFIER
(7)		-> INTEGER
(8)	FACTORS	-> SYM_ASTERISK FACTOR FACTORS
(9)		-> ε
(10)	INTEGER	-> SYM_INTEGER
(11)	IDENTIFIER	-> SYM_IDENTIFIER

(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		1	1		
EXPR		2	2		
TERM		3	3		
TERMS	5			4	
FACTOR		6	7		
FACTORS	9			9	8
INTEGER			10		
IDENTIFIER		11			

(c) La table LL

FIGURE 5 – Fichier généré pour une petite grammaire d'expressions

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 `symbol_t`) générés au TP précédent. On place le mot clé `tokens` suivi de la liste des tokens. On peut donner plusieurs telles lignes.

— 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`. Petite subtilité, on doit aussi déclarer le type C des objets C qui seront construits par ce non-terminal. L'axiome de la grammaire (S) correspondra à un AST (de `ast.h`) et donc sera de type `struct ast_node*`. Pour le moment on ne construit pas d'objet et on peut se contenter du type `void*`.

— Le mot-clé `rules`. Celà 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/parser_generator --help
Usage:
  -g Input grammar file (.g)
  -t Where to output tables (.html)
  -pc Where to output the parser code (.c)
  -ph Where to output the parser header (.h)
  -help  Display this list of options
  --help  Display this list of options
$ alpaga/parser_generator -g fichier-grammaire.c -pc mon-parser.c -ph mon-parser.h -t table.html
```

Cela lira le fichier de grammaire `fichier-grammaire.c` et générera le code du parser dans `mon-parser.c` et les entêtes correspondants dans `mon-parser.h`. 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.

Votre compilateur s'attend à trouver le parser généré dans les fichiers `expr_parser.[ch]`. Faites en sorte de ne pas le contrarier.

Question 5.1. Écrire 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`.

ALPAGA permet de spécifier, pour chaque règle de la grammaire, une **action**, *i.e.* une suite d'instructions qui construit *quelque chose* pour cette règle. Par défaut, lorsqu'aucune action n'est explicitement spécifiée, l'action utilisée est `return NULL;`. Pour spécifier une action, il suffit d'ajouter à la fin de la ligne correspondant à une règle, du code C entre accolades.

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

```
tokens SYM_EOF SYM_IDENTIFIER SYM_INTEGER SYM_PLUS SYM_ASTERISK
non-terminals<struct ast_node*> S EXPR TERM FACTOR INTEGER IDENTIFIER
non-terminals<struct list*> TERMS FACTORS
rules
S -> EXPR SYM_EOF { return $1; }
IDENTIFIER -> SYM_IDENTIFIER { return make_string_leaf(strdup($1)); }
INTEGER -> SYM_INTEGER { return make_int_leaf(atoi($1)); }
EXPR -> TERM TERMS { return make_node(AST_TERMS, cons($1, $2)); }
TERM -> FACTOR FACTORS { return make_node(AST_FACTORS, cons($1, $2)); }
TERMS -> SYM_PLUS TERM TERMS { return cons(make_node(AST_EADD, NULL),
↪  cons($2,$3)); }
TERMS -> { return NULL; }
FACTOR -> IDENTIFIER { return $1; }
FACTOR -> INTEGER { return $1; }
FACTORS -> SYM_ASTERISK FACTOR FACTORS  { return cons(make_node(AST_EMUL, NULL),
↪  cons($2,$3)); }
FACTORS -> { return NULL; }
```

Quelques remarques sur cette grammaire :

— On a dû spécifier le type de chaque non-terminal plus précisément. Dans notre cas, tous les non-terminaux renvoient un `struct ast_node*`, sauf `TERMS` et `FACTORS` qui renvoient une `struct list*`.

— On peut utiliser dans les actions des variables spéciales `$1`, `$2`, ... Cette variable correspond à l'objet C renvoyé par le i-ième élément de la règle. Concrètement, dans la règle :

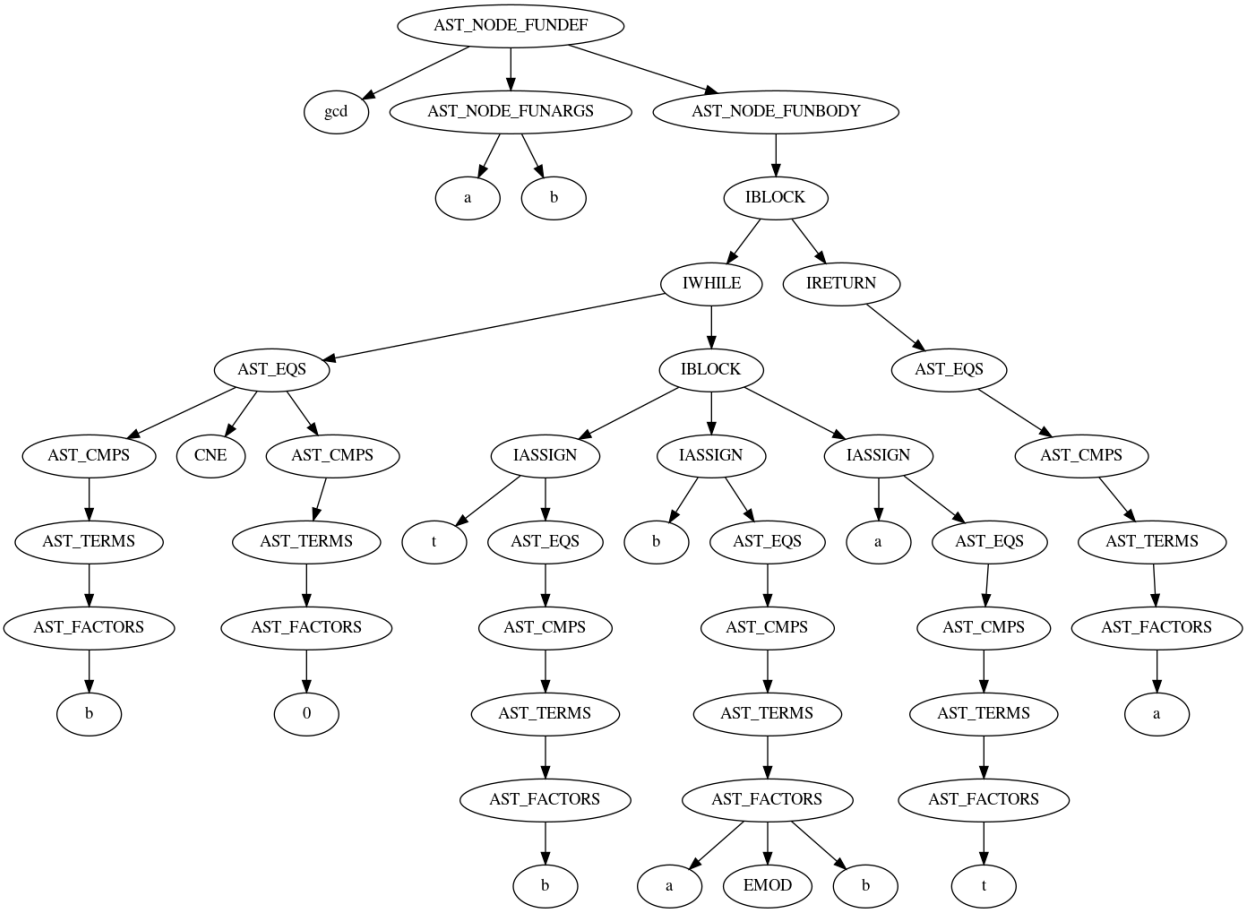
```
EXPR -> TERM TERMS { return make_node(AST_TERMS, cons($1, $2)); }
```

la variable `$1` correspond au résultat renvoyé par `TERM` et la variable `$2` 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`.

Lorsqu'ALPAGA a généré les fichiers C au bon endroit, compilez votre compilateur (wow !). Lancez votre compilateur en lui passant l'option `-ast <file>`, où `file` est un fichier dans lequel l'AST va être écrit, au format DOT (un format pour représenter des graphes). Utilisez l'utilitaire `dot` pour générer une image :

```
# Génération du parseur avec ALPAGA
$ ./alpaga/parser_generator -g ma-grammaire.g -pc expr/expr_parser.c -ph expr/expr_parser.h
...
# Compilation du compilateur (mindblowing!)
$ make -C expr
...
# Compilation et génération du graphe
$ expr/supecomp tests/test00.e -ast mon-fichier.dot
# dot -> png
$ dot -Tpng mon-fichier.dot -o mon-image.png
# Ouvrez l'image avec votre visionneur préféré :
```



Question 5.2 (Action !)**.** Ajouter des actions à votre grammaire afin de construire un AST similaire à celui montré ci-dessus. Référez-vous à la description des fonctions relatives aux types de données (section 2.1) et des ASTs (section 2.2).

6 TP3 : Génération et interprétation de programmes E

À partir de l'AST, vous allez maintenant devoir générer un programme E, tel que défini dans `elang.h`, et écrire un interpréteur pour ces programmes.

6.1 Code existant

Pour vous aider à déboguer votre code, nous vous fournissons quelques fonctions d'affichage de programmes et d'expressions E dans le fichier `elang_print.h` :

```
char* string_of_unop(enum unop_t u);
char* string_of_binop(enum binop_t b);
void print_expression(FILE*, struct expression*);
void print_eprog(FILE*, struct eprog*);
```

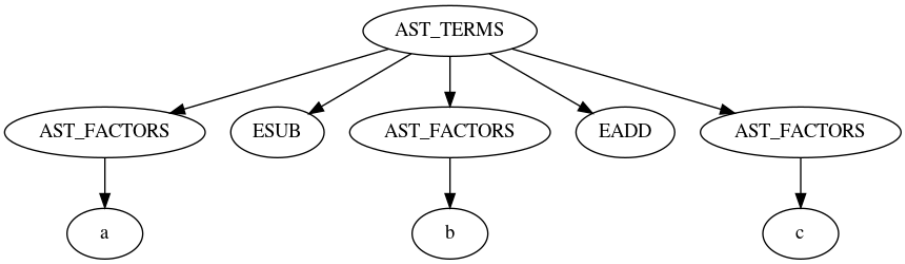
Les fonctions `string_of_unop` et `string_of_binop` donnent une représentation textuelles des opérateurs unaires et binaires, respectivement.

La fonction `print_expression(fd, e)` écrit l'expression dans le fichier représenté par le descripteur de fichier `fd`. Si vous voulez l'afficher sur la sortie standard, passez `stdout` comme descripteur de fichier. La fonction `print_prog(fd, p)` affiche un programme dans un fichier.

6.2 Génération de E

L'objectif est de générer un programme E à partir d'un AST. On aura besoin de fonctions qui génèrent des instructions à partir d'un AST et des expressions à partir d'un AST. La principale difficulté concerne les expressions. Effectivement, lors de l'analyse syntaxique, on a construit une liste de termes et d'opérateurs qu'il faut maintenant transformer en expressions binaires (*i.e.* à deux opérandes).

En particulier, l'expression $a - b + c$ donnera un AST similaire à :



Faites bien attention à générer une expression E qui correspond à $(a - b) + c$ et pas $a - (b + c)$!

Le reste des transformations est relativement non-piégeux.

Question 6.1. Écrivez la fonction `expression* make_terms(ast_node* ast)` qui transforme un sous-arbre comme celui donné dans la figure ci-dessus en une expression E.

Question 6.2. Complétez la fonction `instruction* make_instr(ast_node* ast)` afin de gérer le cas correspondant aux blocs d'instructions (tag `AST_IBLOCK`).

6.3 Interprétation de E

En plus du *printer*, un moyen plus fiable de vérifier votre code 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.

Pour cela, vous allez avoir 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 `state.h` contient des définitions adéquates.

Le langage C étant ce qu'il est, on a besoin de structures et de fonctions différentes selon que l'on associe des noms de variables (chaînes de caractères) ou des identifiants entiers (plus tard, en RTL par exemple) à des valeurs. Pour l'instant, intéressons-nous aux structures et fonctions contenant `string_int` (pour préciser que les identifiants sont des `string` et les valeurs des `int`).

Les fonctions les plus importantes sont `string_int_set_val(s,k,v)`, qui associe la valeur `v` à l'identifiant `k` ; et la fonction `string_int_get_val(s,k)` qui récupère la valeur associée à l'identifiant `k` dans l'état `s`. Contrairement à la fonction `assoc_get` présentée dans la section 2, cette dernière fonction quitte abruptement si la clé demandée n'est pas présente.

Nous vous fournissons également la fonction `print_string_int_state(s)` qui affiche un état – utile pour le débogage !

Question 6.3 (Passage de paramètres)**.** Dans la fonction `run_eprog`, 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 la liste `p->args`).

Lancez votre compilateur avec l'option `-elang-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 6.4. Écrivez le corps des fonctions `run_unop(u,i)` et `run_binop(b,i1,i2)`, qui prennent un opérateur (unaire ou binaire) et leurs paramètres entiers, et renvoient le résultat.

Question 6.5. Écrivez la fonction `run_expression(s, e)` qui évalue l'expression `e`, étant donné un état `s`.

Question 6.6. Écrivez la fonction `run_instruction(s, i)` qui exécute l'instruction `i` avec l'état `s`. Comme l'état peut être modifié par les instructions, notez qu'on passe un pointeur sur l'état : `string_int_state_t**`.

Cette fonction doit rendre la valeur retournée par une instruction `RETURN` du programme E, encapsulée dans un `some()`. Toutes les autres instructions doivent rendre `NULL`.

Dès qu'une instruction `RETURN` est rencontrée, l'exécution se termine.

Question 6.7. Appelez correctement `run_instruction` dans la fonction `run_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 deux optimisations pour notre compilateur : élimination des affectations mortes et propagation de constantes. Chacune de ces optimisations 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 2.4, utilise les mêmes expressions que le langage E, et un sous-ensemble des mêmes instructions. Les différences majeures sont que :

— 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.

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

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

```
$ expr/supecomp mon-fichier.e -cfg mon-fichier.dot
$ dot mon-fichier.dot -Tpng -o mon-fichier.png
```

Vous pouvez utiliser l'interpréteur comme ceci :

```
$ expr/supecomp mon-fichier.e -cfg-run 4 12
# Éventuellement quelques sorties de 'print'
Result: 36
```

7.1 Élimination des affectations mortes

La première 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. [3]

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

7.1.1 Analyse de vivacité

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 à partir de cela les variables vivantes après chaque nœud.

On pourra utiliser une liste d'association comme structure de données, ou bien créer une structure *ad hoc* pour l'occasion.

L'analyse se déroulera en parcourant plusieurs fois le programme, 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).

3. 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.

Chaque itération de l'analyse mettra à jour l'état de l'analyse, *i.e.* le mapping entre identifiant de nœud – liste des variables vivantes. 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 `list* expr_used_vars(struct expression* e)` qui renvoie la liste des variables utilisées par l'expression `e`.

Nous vous fournissons la fonction `list* succs_node(node_t* n)` dans le fichier `cfg_gen.c` qui renvoie la liste des successeurs d'un nœud CFG.

Question 7.2. Écrivez une fonction `list* live_after(node_t* n, list* map)` qui renvoie la liste des variables vivantes après le nœud `n`, étant donné le mapping `map` de variables vivantes avant chacun des nœuds.

Question 7.3. Écrivez une fonction `list* live_before(list* live_aft, node_t* n)` qui renvoie la liste des variables vivantes avant le nœud `n`, où `liv_aft` est la liste des variables vivantes après ce nœud.

Question 7.4. Utilisez les fonctions précédentes pour écrire une fonction `liveness_graph(list* map, cfg* c)` qui parcourt (une fois) le programme `c` et met à jour le mapping `map` des variables vivantes.

Pensez à mettre à jour une variable globale booléenne (*e.g.* `new_changes`) qui indique si l'itération en cours de l'analyse a fait du progrès (*i.e.* a identifié de nouvelles variables vivantes à un point de programme). Cela vous sera utile pour la suite.

Indication : vous pouvez utiliser la fonction `list_string_incl` de `datatypes.h` pour tester si une liste est incluse (au sens de l'inclusion ensembliste) dans une autre.

Question 7.5. Écrivez une fonction `list* liveness_prog(cfg_prog* p)`, qui renvoie la correspondance entre identifiants de nœuds et ensemble de variables vivantes. Vous devrez continuer à appeler `liveness_graph` tant que la correspondance n'est pas stabilisée.

7.1.2 Élimination de code mort

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 (type `NODE_ASSIGN`), 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 `NODE_GOTO`).

Question 7.6. Écrire une fonction `void dead_assign_elimination_graph(list* live, cfg* c)` qui, étant donnés le résultat de l'analyse de vivacité `live` et un graphe de flot de contrôle `c`, itère la transformation décrite ci-dessus sur chaque nœud du graphe.

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 7.7. Écrivez une fonction `cfg_prog* dead_assign_elimination_prog(cfg_prog* p)` qui itère la fonction précédente autant de fois qu'utile et nécessaire.

7.2 Bonus : Propagation de constantes

Dans cette partie (facultative, mais passionnante, si, si !), on propose d'optimiser encore plus les programmes en propageant les valeurs des variables dont on connaît statiquement la valeur, et ainsi faire le maximum de calculs à la compilation plutôt qu'à l'exécution. De la même manière que pour l'élimination d'affectations mortes, on va procéder en deux étapes : une première étape d'analyse qui va nous fournir un résultat intermédiaire, puis une seconde étape de transformation de programmes qui va utiliser le résultat de l'analyse précédente. Et encore une fois, c'est l'étape d'analyse qui va demander le plus de travail.

7.2.1 Analyse de valeurs

Nous souhaitons calculer, pour chaque point de programme, une valeur abstraite pour chaque variable. Une valeur abstraite appartient à un treillis dont les valeurs possibles sont $\top$ (la variable en question peut contenir n'importe quelle valeur), $Cst(c)$ (la variable contient la valeur c et rien d'autre), et $\bot$ (on n'a pas d'information sur le contenu de la variable). L'ordre partiel associé à ce treillis est donné par les règles suivantes :

$$\forall\ c, Cst(c) \sqsubseteq \top$$
$$\forall\ c, \bot \sqsubseteq Cst(c)$$
$$\bot \sqsubseteq \top$$

Pour chaque point de programme, on va stocker un état abstrait, représenté par la structure ci-dessous :

```
typedef struct astate {
  struct astate* next;
  char* var;
  aval* av;
} astate;
```

Vous reconnaissez une structure de type « liste » à la présence du champ `next`. Chaque élément est un mapping entre un nom de variable (`char* var`) et une valeur abstraite (`aval* av`).

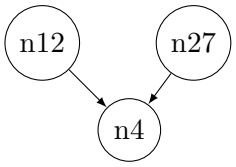
La structure correspondant au programme entier est une liste d'association (comme définies dans la section 2.1 et dans le fichier `datatypes.h`) qui à chaque identifiant de nœud du programme

(un entier) associe un état abstrait `astate`. On stockera dans cette structure les états abstraits après chaque nœud, et on calculera au besoin l'état avant un nœud en parcourant ses prédécesseurs.

Nous vous fournissons les fonctions `aunop(unop_t u, aval* a)` et `abinop(binop_t b, aval* a1, aval* a2)` qui évaluent sur notre domaine abstrait les opérations unaires et binaires, respectivement. Nous vous fournissons également la fonction `astate_lookup(astate*s, char* var)` qui renvoie la valeur abstraite associée à la variable `var` dans l'état abstrait `s`. Cette fonction renvoie $\bot$ si la variable n'est pas trouvée.

Question 7.8. Écrivez la fonction `aval* aexpr(astate* s, expression* e)` qui renvoie la valeur abstraite associée à l'expression `e`, dans l'état abstrait `s`.

Considérons un morceau de CFG comme le suivant :



En supposant qu'on ait les états abstraits à la sortie des nœuds `n12` et `n27`, on souhaite calculer l'état abstrait à l'entrée du nœud `n4`. Cela correspond à une opération qu'on appelle *meet* et que l'on note $\sqcap$. On commence par définir le *meet* sur les valeurs abstraites de la façon suivante :

$$\begin{array}{ll} \bot \sqcap a & = a \\ \top \sqcap a & = \top \\ Cst(c) \sqcap Cst(c) & = Cst(c) \\ Cst(c) \sqcap Cst(d) & = \top, \text{ si } c \neq d \\ a \sqcap b & = b \sqcap a \end{array}$$

Intuitivement, pour une variable donnée, si un prédécesseur donne la valeur abstraite $\bot$ (pas d'information), nous allons garder la valeur abstraite donnée par l'autre prédecesseur. Si un prédécesseur annonce $\top$ (n'importe quelle valeur), alors peu importe la valeur abstraite annoncée par l'autre prédécesseur, le résultat sera $\top$. Si les deux prédécesseur annoncent une constante, soit c'est la même et on garde donc cette constante, soit ce sont des constantes différentes et on renvoie $\top$.

Question 7.9. Écrire la fonction `aval* meet_aval(aval* a1, aval* a2)` qui calcule le *meet* de deux valeurs abstraites.

On souhaite maintenant calculer le *meet* de deux états abstraits. Par exemple, on aura

$$\{x : \top; y : Cst(42)\} \sqcap \{y : Cst(23); z : Cst(8)\} = \{x : \top; y : \top; z : Cst(8)\}$$

Question 7.10. Écrire la fonction `astate* meet(astate* a1, astate* a2)` qui calcule le *meet* de deux états abstraits.

On va maintenant vouloir faire le *meet* de tous les états abstraits correspondants aux prédecesseurs d'un nœud donné. On vous fournit une fonction `list* preds(cfg* c, int node_id)` qui renvoie la liste des nœuds qui sont les prédecesseurs du nœud `node_id` dans le graphe `c`.

Question 7.11. Écrivez la fonction `astate* astate_before(cfg_prog* p, int node_id, list* cfg_astate)` qui calcule l'état abstrait à l'entrée du nœud `node_id` dans le programme `p`, étant donné l'état abstrait après chaque nœud dans `cfg_astate*`.

Attention, si le nœud `node_id` correspond au nœud d'entrée du programme, on n'a aucun prédecesseur, mais l'état abstrait avant ce nœud doit associer la valeur $\top$ aux arguments du programme.

Outre ce cas particulier, l'état abstrait doit être calculé en prenant le *meet* des états abstraits de chacun des prédécesseurs du nœud `node_id`.

On arrive bientôt à la fin de l'analyse, promis ! Il ne nous reste plus qu'à parcourir le CFG, et pour chaque nœud, calculer l'état abstrait avant ce nœud et mettre à jour l'état après ce nœud en fonction du type de nœud. Seuls les nœuds d'affectation changent cet état abstrait. À vous de jouer.

Question 7.12. Écrire une fonction `list* acfg(list* cfg_astate, cfg_prog* p)` qui prend un état abstrait de programme `cfg_astate`, un programme `p` et calcule un nouvel état abstrait de programme.

De la même manière que pour l'analyse de vivacité, on itèrera cette fonction tant que des changements sont appliqués aux états abstraits, on prendra donc soin de notifier si un état abstrait a changé au moyen d'une variable globale par exemple.

Note : un état abstrait change si au moins une variable change de valeur abstraite. Vous pouvez utiliser la fonction `int more_precise(astate* a1, astate* a2)` qui détermine si `a1` est plus précis que `a2`.

Question 7.13. Écrivez l'ultime fonction de cette analyse `list* value_analysis_cfg_prog(cfg_prog* p)` qui renvoie l'état abstrait du programme `p`. Itérez autant que nécessaire.

Vous pouvez afficher les valeurs abstraites et états abstraits grâce aux fonctions `print_aval`, `print_astate` et `print_value_analysis_result` présentes dans le fichier `cfg_value_analysis.c`, et que nous vous laissons découvrir.

7.2.2 Propagation de constantes

Il est temps d'utiliser le résultat de cette analyse, obtenue à la sueur de vos fronts.

Pour commencer, essayons de simplifier des expressions étant donné un état abstrait. Cela est plus subtil que simplement appeler la fonction `aexpr` définie dans la partie précédente, puisque celle-ci va renvoyer $\top$ si une variable non constante apparaît dans l'expression. Par exemple, pour l'expression $(1 + 3 * 8 - 23) * x$, la fonction `aexpr` rendra $\top$, alors que nous souhaiterions obtenir $2 * x$, *i.e.* simplifier autant que possible les sous-expressions.

Question 7.14. Écrivez la fonction `expression* asimpl(astate* as, expression* e)` qui effectue la simplification d'expressions.

Question 7.15. Écrivez la fonction `void constant_propagation_graph(list* value_analysis, cfg_prog* p)` qui parcourt le graphe et simplifie toutes les expressions qui apparaissent.

Question 7.16. On peut encore simplifier les nœuds conditionnels. Voyez-vous comment ? À vous de jouer !

Question 7.17 (Bonus)**.** La fonction `abinop` peut être plus précise (*i.e.* renvoyer plus souvent une constante plutôt que $\top$). Voyez-vous comment ? Améliorez-la !

8 TP5 : Génération de programmes RTL

L'objectif de cette séance de TP est de générer des programmes RTL. Comme expliqué dans la section 2, RTL est un langage où 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 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 intermédiare IR de Clang y ressemble aussi, et CompCert utilise également un langage 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. Nous allons utiliser une variable globale `curreg` qui contiendra le nom du prochain registre à être alloué. La fonction `new_reg()` incrémente ce compteur et vous renvoie un identifiant de registre neuf.
2. Ensuite, il nous faut une structure de données qui maintient une correspondance entre les noms de variables du programme CFG, et le registre RTL qui lui est associé. On va utiliser une **table des symboles**, représentée par la structure suivante :

   ```
   typedef struct symtable {
     struct symtable* next;
     char* var;
     int reg;
   } symtable;
   ```

   Vous devriez à présent être capables de comprendre de telles structures de données, après en avoir vu de similaires dans les séances précédentes. On associe à une variable `var` un registre `reg`.
3. Enfin, il nous faudra transformer les expressions et instructions CFG en opérations RTL.

Pour simplifier votre code, on propose de stocker la table des symboles dans une variable globale (déjà déclarée et initialisée à `NULL` pour vous – on vous chouchoute !).

Question 8.1. Écrivez une fonction `int get_reg_for_var(char* v)` qui renvoie le registre associé à la variable `v`. Si aucun registre n'a encore été associé à cette variable, créez un nouvel identifiant, associez-le à cette variable, et renvoyez-le.

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. Nous allons stocker le résultat intermédiaire de cette compilation dans une structure `expr_compiled` dont voici le type :

```
typedef struct expr_compiled {
  int r;
  list* ops;
} expr_compiled;
```

Ainsi, la compilation d'une expression donnera une structure `ec`, où `ec->r` contiendra le nom du registre dans lequel l'expression a été calculée, et `ec->ops` la liste des opérations qui effectuent le calcul de cette expression.

Par exemple, pour compiler l'expression `a + 2 * b` :

— on commence par compiler la sous-expression `a`. Supposons que la variable `a` soit associée au registre `r3`, le résultat de la compilation de cette première sous-expression donnera un `ec1` tel que `ec1->r` vaut `r3` et `ec1->ops` vaut `NULL`. (Aucune opération n'est requise.)

— on compile ensuite la sous-expression `2 * b`. Supposons que la variable `b` soit associée au registre `r4`. Nous allons utiliser un nouveau registre, disons `r5`, pour stocker la valeur `2`, et encore un nouveau registre, disons `r6`, pour stocker le résultat de la multiplication. Le résultat de la compilation de cette seconde sous-expression donnera un `ec2` tel que `ec2->r` vaut `r6` et `ec2->ops` vaut `r5 <- 2; r6 <- r5 * r4`.

— On combine ces deux sous-résultats : il nous faut un nouveau registre pour stocker le résultat de l'expression en entier, disons `r8`. On renvoie finalement un `ec` avec `ec->r` ayant pour valeur `r8` et `ec->ops` étant la concaténation de `ec1->ops` et `ec2->ops`, suivi de l'addition (à vous de trouver comment écrire cette addition...).

Question 8.2. Écrivez la fonction `expr_compiled* rtl_ops_of_expression(expression* e)` qui transforme une expression CFG (*i.e.* une expression E) en un résultat de type `expr_compiled`.

Rappelez-vous des fonctions utiles sur les listes : `concat`, `list_append`, `cons`...

Les expressions sont compilées, maintenant 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 8.3. Écrivez la fonction `list* rtl_ops_of_cfg_node(node_t* c)` qui renvoie, étant donné un nœud CFG, la liste des opérations RTL correspondantes.

Ça y est, la plus grande partie du travail est faite ! Il n'y a plus qu'à utiliser ces fonctions correctement pour produire un programme RTL complet.

Question 8.4. Écrivez une fonction `rtl* rtl_of_cfg_graph(cfg* c)` qui renvoie un graphe RTL, comme défini ci-dessous (et dans `rtl.h`).

```
typedef struct rtl {
  struct rtl* next;
  int id;
  list* ops;
} rtl;
```

Question 8.5. Écrivez la fonction `rtl_prog* rtl_of_cfg_prog(cfg_prog* cfg)`. Un programme RTL est défini par la structure suivante :

```
typedef struct rtl_prog {
```

```
  char* fname;
  struct list* args;
  rtl* graph;
  int entry;
} rtl_prog;
```

Attention, les arguments du programme RTL sont des identifiants de registre, contrairement aux arguments du programme CFG, qui sont des chaînes de caractère.

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

```
$ expr/supecomp tests/gcd.e -rtl -
gcd( r0, r1):
goto n2
n2:
r2 <- 0
r3 <- r1 != r2
r3 ? goto n5 : goto n1
n5:
r4 <- r1
goto n4
n4:
r5 <- r0 % r1
r1 <- r5
goto n3
n3:
r0 <- r4
goto n2
n1:
return r0
$ expr/supecomp tests/gcd.e -rtl-run 21 14
Result = 7
```

9 TP6 : Allocation de registres et génération d'assembleur RISC-V

Le but de cette séance de travaux pratiques est de produire du code assembleur qui pourra être assemblé et exécuté par vos machines. Votre compilateur est déjà doté d'un générateur d'asssembleur x86 ; vous allez construire un générateur d'assembleur RISC-V.

Il reste une étape fondamentale pour passer du code RTL au code assembleur : il nous faut faire l'allocation de registres, c'est-à-dire spécifier, pour chaque pseudo-registre RTL, à quel emplacement matériel on va le stocker. Concrètement, on va associer 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 x86 exécutables !

```
$ expr/supecomp tests/gcd.e -target x86-64 -libdir expr/runtime_x86-64 -naive-regalloc -o gcd
$ ./gcd 54 24
Result = 6

$ expr/supecomp tests/prime.e -target x86-64 -libdir expr/runtime_x86-64 -naive-regalloc -o prime
$ ./prime 182
2
7
13
Result = 0
```

Quelques mots sur les options du compilateur que l'on vient d'utiliser :

- `-target arch` : choisit l'architecture cible, *i.e.* pour quelle architecture le compilateur doit-il émettre de l'assembleur ? Les choix possibles sont `x86-64` (processeur x86 sur 64 bits), `x86-32` (processeur x86 sur 32 bits) ou bien `riscv` (processeur RISC-V sur 64 bits). Pour le moment, seules les cibles `x86-xx` sont opérationnelles.
- `-libdir <dir>` : indique le répertoire où trouver la fonction `main` qui va se charger de lancer notre programme compilé. Vous pouvez aller jeter un œil à ce fichier `main.c` : son rôle est de pousser les arguments, convertis en entier, sur la pile pour que notre programme puisse y accéder. On y recourt à de l'assembleur *inliné* pour forcer les arguments à être mis sur la pile.

  Note : on ne peut pas simplement appeler la fonction `supecomp_main` normalement, puisqu'on ne connaît pas le nombre d'arguments à passer ; et on attend les arguments sur la pile, pas dans des registres.
- `-naive-regalloc` : cette option (activée par défaut) choisit l'allocateur de registre qui alloue tout sur la pile, le seul opérationnel au début de cette séance. Lorsque vous écrirez votre allocateur de registres utilisant n registres, vous pourrez utiliser l'option `-clever-regalloc n`.

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

Au programme de ce TP, deux objectifs principaux :

- écrire un générateur de code assembleur RISC-V ; et
- écrire un allocateur de registres plus intelligent.

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 vous aurez besoin pour écrire votre compilateur.

9.1.1 Registres RISC-V

RISC-V existe en 32 bits ou 64 bits (et même 128 bits), mais nous allons écrire du code pour RISC-V 64. Dans cette architecture, RISC-V donne accès à 32 registres de 64 bits chacun, 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 *callee-save*
x3	gp	*global pointer*	x19	s3	
x4	tp	*thread pointer*	x20	s4	
x5	t0	temporaires	x21	s5	
x6	t1		x22	s6	
x7	t2		x23	s7	
x8	s0/fp	*frame pointer*	x24	s8	
x9	s1	registre *callee-save*	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. [4]

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`.

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

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 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 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`), vous aurez 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, 1 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

Pour simplifier un peu votre travail, nous introduisons un nouveau langage que l'on appelle Linear. Ce langage est très similaire à RTL, sauf qu'un programme est une liste d'opérations RTL (plutôt qu'un graphe dont les nœuds contiennent des listes d'opérations RTL, comme c'était le cas en RTL). Vous trouverez la définition du langage dans `linear.h`, la passe de compilation depuis RTL vers Linear dans `linear_gen.[ch]`, un afficheur pour Linear dans `linear_print.[ch]` et un interprète pour Linear dans `linear_run.[ch]`.

Le fichier `riscv_gen.c` contient déjà quelques définitions. Le type `riscvreg` représente l'ensemble des registres RISC-V. Le tableau `riscv_regs` contient le nom de chacun de ces registres. Ainsi, `riscv_regs[A0]` contient la chaîne `"a0"`.

Comme nous l'avons brièvement expliqué, certains registres ont des utilisations spécifiques et ne peuvent pas être utilisés par l'allocateur de registres. Le tableau `riscv_allocable` contient la liste des registres utilisables.

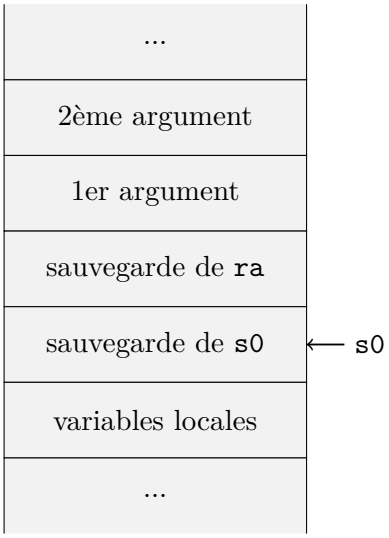
Pour garder l'allocateur indépendant de l'architecture cible, on demande simplement à l'allocateur de donner soit un emplacement sur la pile, soit un numéro de registre dans un intervalle $[0; n[$. La fonction `riscv_fix_allocation` remplace ces entiers par les registres RISC-V correspondants dans le tableau `riscv_allocable`.

Plus précisément, l'allocateur de registres renvoie une `allocation`, qui associe à chaque pseudo-registre un emplacement (*location*) :

```
typedef enum loc_t {STACK, PARAM, REG} loc_t;

typedef struct loc {
  enum loc_t type;
  union {
    int stkoffset;
    int paramnum;
    int reg;
  };
} loc;

typedef struct allocation {
  struct allocation* next;
  int pseudoreg;
  struct loc loc;
} allocation;
```



Une `loc` peut être de trois types :

— soit un emplacement sur la pile (type `STACK`), sera compilé vers `x(s0)`, où `x` vaut `-8*l.stkoffset` ;

— soit un paramètre (type `PARAM`), sera compilé vers `x(s0)`), où `x` vaut `8*(l.paramnum+2)` ;

— soit un registre (type `REG`), sera compilé vers `riscv_allocation[l.reg]`

La fonction `riscv_loc(FILE* f, loc l)` affiche l'emplacement `l` comme décrit ci-dessus.

La fonction `get_loc(allocation* alloc, int pseudoreg)` renvoie l'emplacement alloué pour le pseudo-registre RTL `pseudoreg`. Cette fonction s'occupe d'afficher une erreur et de quitter si le pseudo-registre n'est pas alloué, ce qui ne devrait pas arriver (sauf si dans la suite du TP vous écrivez un allocateur de registres buggé !). Vous pouvez donc utiliser cette fonction sans vous soucier de gérer les erreurs.

La fonction `make_ld(FILE* f, riscvreg reg, loc* l)` émet une instruction `ld` où `reg` est le registre destination et `l` est l'emplacement d'où on veut lire. Symétriquement, la fonction `make_sd(FILE* f, riscvreg reg, loc* l)` émet une instruction `sd` où `reg` est le registre source et `l` est l'emplacement où l'on veut écrire.

La fonction `make_mv(FILE* f, riscvreg rd, riscvreg rs)` émet une instruction `mv` du registre `rs` vers le registre `rd`.

Question 9.1. Écrivez la fonction `void riscv_make_move(allocation* alloc, FILE* f, int pseudoregsrc, int pseudoregdst)` qui copie la valeur du pseudo-registre `pseudoregsrc` dans le pseudo-registre `pseudoregdst`.

Selon le type l'emplacement alloué pour chacun des pseudo-registres, il faudra peut-être plusieurs instructions.

Si nécessaire, vous pouvez utiliser les registres `T0`, `T1` et `T2` comme bon vous semble. (Ces registres ne sont pas utilisés par l'allocateur.)

Question 9.2. Écrivez la fonction `void riscv_make_move_to_reg(allocation* alloc, FILE* f, int pseudoregsrc, riscvreg dst)`, presque identique à la fonction précédente, à l'exception que la destination est déjà un registre RISC-V.

On utilisera en particulier cette fonction pour copier une valeur dans un registre temporaire.

La fonction `riscvreg optional_read_reg(allocation* alloc, FILE* f, int pseudoreg, riscvreg temp)` renvoie un registre dans lequel on trouvera la valeur contenue dans le pseudo-registre RTL `pseudoreg`. Si l'emplacement associé à ce pseudo-registre est un registre, il suffit de renvoyer ce registre. Sinon, on copie depuis la pile vers le registre `temp` passé en paramètre et on renvoie `temp`.

La fonction `riscvreg optional_write_reg(allocation* alloc, FILE* f, int pseudoreg, riscvreg temp)` renvoie un registre dans lequel on devra écrire pour compiler une écriture dans le pseudo-registre RTL `pseudoreg`. Si l'emplacement associé à ce pseudo-registre est un registre, il suffit de renvoyer ce registre. Sinon, on écrira le registre `temp` dans la pile et on renvoie `temp`. À noter qu'on devra appeler cette fonction deux fois : une fois pour savoir dans quel registre il faut écrire notre valeur, et une deuxième fois pour émettre l'instruction `sd` correspondante, le cas échéant.

Pour mieux comprendre, analysons la fonction `make_unop` qui vous est fournie.

```
1 void make_unop(FILE* f, allocation* alloc, char* op, int pseudord, int pseudors){
2   riscvreg rs = optional_read_reg(alloc, f, pseudors, T0);
3   riscvreg rd = optional_write_reg(alloc, NULL, pseudord, T0);
4   fprintf(f, "%s %s, %s\n", op, riscv_regs[rd], riscv_regs[rs]);
5   optional_write_reg(alloc, f, pseudord, T0);
6 }
```

L'objectif de cette fonction est d'émettre le code assembleur RISC-V correspondant à l'opération RTL `pseudord <- op pseudors`. La première chose à faire est de comprendre à quel emplacement matériel la source de cette opération se trouve. C'est ce que fait la ligne 2.

— Si `pseudors` est alloué dans le registre matériel `s1`, alors `rs` vaut `S1` et rien n'a été écrit dans `f`.

— Si `pseudors` est alloué dans la pile à l'offset -16, alors on écrit `ld t0 -16(s0)` dans le fichier `c` et `rs` vaut `T0`.

Ensuite, on doit savoir dans quel registre l'instruction doit écrire son résultat. On appelle une première fois `optional_write_reg` avec `NULL` comme fichier, pour ne pas émettre d'instruction. Le sens de ce premier appel (ligne 3) est « si je veux écrire dans l'équivalent du pseudo-registre `pseudord`, dans quel registre matériel dois-je écrire ? et si besoin, sache que le registre `temp` est disponible... ».

Après ce premier appel, on aura :

— Si `pseudord` est alloué dans le registre matériel `s3`, alors `rs` vaut `S3` et rien n'a été écrit dans `f`.

— Si `pseudord` est alloué dans la pile à l'offset -24, alors on écrit `sd t0 -24(s0)` dans le fichier `c` et `rs` vaut `T0`.

On sait maintenant quels registres RISC-V notre opération doit utiliser, il est donc aisé d'écrire cette instruction (ligne 4).

Finalement, on rappelle la fonction `optional_write_reg`, cette fois-ci avec le vrai descripteur de fichier pour émettre, si besoin, la copie depuis le registre temporaire vers la pile.

La fonction `make_binop` est similaire, et vous la comprendrez très bien tous seuls.

La fonction `riscv_of_rtl_op` est pour l'instant vide, ce sera à vous de la compléter.

La fonction `riscv_of_rtl_ops` répète la fonction précédente sur une liste d'opérations, *i.e.* sur un programme.

Finalement, la fonction `riscv_of_lin_prog(linear_prog* lin)` produit un fichier assembleur RISC-V complet. On commence par définir la fonction `supecomp_main` qui correspond à notre programme E. Puis, on sauvegarde les registres `ra` et `s0` sur la pile et on réserve de la place sur la pile pour les variables locales. Ensuite viennent la compilation des opérations RTL, puis un épilogue. On définit une étiquette `".ret"` à laquelle il faudra sauter lors du retour de fonction après avoir stocké la valeur de retour de la fonction dans le registre `A0`, puis on nettoie la pile : on restaure les registres `s0` et `ra` dans l'état dans lequel ils étaient au moment de l'appel de fonction, et on appelle l'instruction `ret` – qui est du sucre syntaxique pour `jr ra`, qui saute à l'adresse contenue dans le registre `ra`.

La compilation des instructions `PRINT` s'effectue en faisant un appel à la fonction `print_int`, définie dans `<libdir>/main.c`. Vous pouvez l'appeler en mettant le paramètre dans le registre `A0` et en appelant la fonction avec `jal ra, print_int`.

Question 9.3. Écrivez la fonction `void riscv_of_rtl_op(allocation* alloc, FILE* f, rtl_op* n)` qui génère le code assembleur correspondant à une opération RTL.

Vous pouvez dorénavant compiler pour RISC-V avec la commande suivante :

```
$ expr/supecomp tests/gcd.e -target riscv -libdir expr/runtime_riscv -o gcd
$  qemu-riscv64 gcd 54 24
Result = 6
$ expr/supecomp tests/prime.e -target riscv -libdir expr/runtime_riscv -o prime
$ qemu-riscv64 prime 182
2
7
13
Result = 0
```

Vous devriez avoir le même résultat que lors de la compilation pour la cible x86. Dans l'exemple ci-dessus, on lance les binaires RISC-V avec l'émulateur `qemu-riscv64`. Si vous utilisez Linux équipé de `binfmt_misc`[5], vous pouvez même lancer les binaires directement, comme si c'étaient des binaires x86.

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 registre 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. Nous allons commencer par écrire une analyse de vivacité des pseudo-registres RTL. 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 Vivacité des pseudo-registres

On commence par écrire une analyse de vivacité des pseudo-registres. Le programme Linear est une liste d'opérations RTL. Notre objectif est d'associer à chaque point de programme n la liste des pseudo-registres vivants avant la n-ième instruction.

Cette analyse sera définie dans le fichier `linear_liveness.c`. On commence par définir la fonction `list* setup_labels(list* ops)` qui renvoie une liste d'association dont les clés sont des labels et les valeurs sont les points de programme correspondant (la position du label dans le programme).

Ensuite vient la fonction `int resolve_label(int lab, list* labels)` qui renvoie le point de programme correspondant au label `lab`.

L'intérêt des fonctions précédentes est de définir la fonction `list* succs(rtl_op* op, int pc, list* labels)` qui renvoie la liste des successeurs d'une opération RTL `op` située au point de programme `pc`. Pour la plupart des opérations, la liste des successeurs ne contient que l'instruction immédiatement après dans le programme, soit `pc+1`. Pour l'instruction `RBRANCH`, les successeurs sont les points de programmes correspondant aux deux labels de l'instruction ; et pour l'instruction `RGOTO`, le seul successeur est le point de programme associé au label de l'instruction. L'opération `RRET` n'a pas de successeur.

La fonction `list* live_before_op(list* live_aft, rtl_op* op)` sera à compléter par vos soins. Elle renverra la liste des pseudo-registres vivants avant une opération `op`, étant donnée la liste des variables vivantes après cette opération `live_aft`.

La fonction `list* live_after_op(int pc, list* ops, list* labels, list* map)` renvoie la liste des pseudo-registres vivants après l'opération située au point de programme `pc`, dans le programme `ops`, étant donnée l'association de labels – points de programme `labels` et le résultat partiel de l'analyse de vivacité `map`. Ce résultat est une liste d'association dont les clés sont les points de programme et les valeurs sont les listes de pseudo-registres vivants avant le point de programme en question. Les pseudo-registres vivants après un point de programme sont l'union des

5. Et c'est le cas sur les ordinateurs de la salle 509 !

pseudo-registres vivants avant chacun des successeur de ce point de programme. (Cela ne devrait pas être surprenant, puisque c'est très similaire à l'analyse de vivacité que l'on a effectué au TP4 sur le langage CFG.)

La fonction `list* liveness_linear_ops(list* map, list* ops)` parcourt entièrement le programme et met à jour le résultat `map`. Comme pour l'analyse de vivacité sur CFG au TP4, on va répéter ce parcours tant qu'on apporte des changements au résultat de l'analyse. Cette répétition est faite dans la fonction principale `list* liveness_linear_prog(linear_prog* p)`.

Question 9.4. Votre seul travail pour cette analyse de vivacité consiste à compléter la fonction `list* live_before_op(list* live_aft, rtl_op* op)`.

Vous aurez sans doute besoin des fonctions sur les listes `cons_int` et `list_remove_int`.

Souvenez-vous : un registre est vivant avant une opération si :

- il est nécessaire à l'évaluation de cette opération, ou
  - il est vivant après cette opération, et
  - il n'est pas écrasé par cette opération.

Vous pouvez afficher le programme Linear avec le résultat de l'analyse de vivacité avec la commande suivante :

```
expr/supecomp tests/prime.e -lin -
main(r0):
.n10: # Live before : [0]
r1 <- 2 # Live before : [0]
r2 <- r0 % r1 # Live before : [1, 0]
r3 <- 0 # Live before : [2, 0]
r4 <- r2 == r3 # Live before : [2, 3, 0]
r4 ? goto n12 : goto n9 # Live before : [4, 0]
.n9: # Live before : [0]
r9 <- 3 # Live before : [0]
r8 <- r9 # Live before : [9, 0]
goto n4 # Live before : [8, 0]
.n4: # Live before : [8, 0]
r10 <- r8 * r8 # Live before : [8, 0]
r11 <- r10 <= r0 # Live before : [10, 8, 0]
r11 ? goto n8 : goto n3 # Live before : [11, 8, 0]
.n3: # Live before : [0]
r18 <- 1 # Live before : [0]
r19 <- r0 != r18 # Live before : [18, 0]
r19 ? goto n2 : goto n1 # Live before : [19, 0]
.n1: # Live before : []
r20 <- 0 # Live before : []
return r20 # Live before : [20]
.n2: # Live before : [0]
print r0 # Live before : [0]
goto n1 # Live before : []
...
```

9.2.2 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 `regalloc.c`.

Construction du graphe d'interférence. La construction du graphe d'interférence sera effectuée par la fonction `list* build_interference_graph(linear_prog* p)` qui renvoie le graphe d'interférence. Ce graphe est une liste d'association où les clés sont les sommets du graphe, c'est-à-dire les pseudo-registres utilisés dans le programme, et les valeurs sont les listes d'adjacence, c'est-à-dire les sommets voisins du sommet clé.

La première partie de la construction du graphe est l'initialisation des sommets. La fonction `list* regs_of_rtl_ops(list* n)`, fournie, calcule la liste des pseudo-registres utilisés dans une liste d'opérations RTL `n`. On initialise le graphe avec une liste de voisins vide pour chaque sommet.

Ensuite, on calcule l'analyse de vivacité :

```
list* live = liveness_linear_prog(p);
```

Puis 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 9.5. Écrivez une fonction `list* add_interf(int x, int y, list* rig)` 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 9.6. Utilisez la fonction `add_interf` pour compléter la fonction `build_interference_graph`.

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 1981 [6].

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), ou non (dans ce cas, il devra être alloué sur la pile – on dit qu'il est *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 existe, appelons-le S, 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 mettre sur la pile la paire $(s, \texttt{NOSPILL})$.

   — Sinon, on va devoir *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

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

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 *spillé*. Dans notre code on mettre sur la pile la paire $(s, \texttt{SPILL})$.

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`.

   On maintient le numéro du prochain emplacement disponible sur la pile dans une variable `next_stack_slot`, qu'on initialise à 1. On initialise notre allocation `alloc` à `NULL`.

   Pour chaque paire $(s, \textit{decision})$ sur la pile :

   — Si $\textit{decision} = \texttt{SPILL}$, on alloue s sur la pile au prochain emplacement disponible.

   — Sinon, on cherche une couleur (un entier dans l'intervalle $[0; N[$) disponible, c'est-à-dire qui n'est pas déjà allouée à un voisin de s. Une telle couleur existe forcément, puisque s a moins de N voisins.

Question 9.7. Écrivez une fonction `int* pick_node_with_fewer_than_n_neighbors(list* rig, int n)` qui renvoie `some(s)`, avec `s` un sommet de `rig` avec moins de `n` voisins si possible, et `NULL` sinon.

On utilisera les fonctions `list_length` et `clear_dup` (qui retire les doublons d'une liste d'entiers).

Question 9.8. Écrivez une fonction `int* pick_spilling_candidate(list* rig)` qui renvoie un sommet que l'on va *spiller*. Choisissez le sommet avec le maximum de voisins. Comme pour la fonction précédente, renvoyez `some(s)` ou `NULL`.

On vous fournit la fonction `list* remove_from_rig(list*rig, int x)` qui retire un sommet d'un graphe, et le type `regalloc_decision_t` avec les valeurs `SPILL` et `NOSPILL` expliquées ci-dessus.

Question 9.9. Écrivez la fonction `list* make_stack(list* rig, int ncolors)` qui construit la pile décrite dans la première phase de l'algorithme de coloration de graphe décrit plus haut.

Question 9.10. Écrivez la fonction `allocation* make_allocation(allocation* alloc, list* rig, list* stack, int* next_stack_slot, int ncolors)` qui construit une allocation à partir de :

— `alloc` : une allocation partielle (notamment les arguments sont déjà alloués) ;

— `rig` : le graphe d'interférence, utile pour connaître les voisins d'un sommet ;

— `stack` : la pile créée par la fonction `make_stack` ;

— `next_stack_slot` : un pointeur vers le numéro du prochain emplacement disponible sur la pile. Il s'agit d'un pointeur parce que la fonction appelante aura besoin de savoir

combien d'espace il est nécessaire de prévoir pour les pseudo-registres *spillés* ;
— `ncolors` : le nombre de couleurs disponibles

Les fonctions `make_stack` et `make_allocation` sont appelées par la fonction `regalloc_prog` qui vous est donnée déjà complétée. Cette fonction affiche le résultat de l'allocation si le flag `print_allocation_flag` est à 1, ce que vous pouvez activer en passant l'option `-show-regalloc` au compilateur.

```
$ expr/supecomp tests/gcd.e -clever-regalloc 3 -lin - -show-regalloc -s /dev/null
gcd(r0, r1):
.n2: # Live before : [1, 0]
r2 <- 0 # Live before : [1, 0]
r3 <- r1 != r2 # Live before : [2, 1, 0]
r3 ? goto n5 : goto n1 # Live before : [3, 1, 0]
.n1: # Live before : [0]
return r0 # Live before : [0]
.n5: # Live before : [0, 1]
r4 <- r1 # Live before : [0, 1]
goto n4 # Live before : [0, 1, 4]
.n4: # Live before : [0, 1, 4]
r5 <- r0 % r1 # Live before : [0, 1, 4]
r1 <- r5 # Live before : [5, 4]
goto n3 # Live before : [4, 1]
.n3: # Live before : [4, 1]
r0 <- r4 # Live before : [4, 1]
goto n2 # Live before : [1, 0]
r0 -> param(0)
r1 -> param(1)
r5 -> reg(2)
r4 -> reg(1)
r3 -> reg(2)
r2 -> reg(2)
```

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

— `-clever-regalloc 3` pour utiliser l'allocateur de registres que l'on vient de programmer avec 3 couleurs.
— `-show-regalloc` pour afficher l'allocation de registres obtenue.
— `-lin -` pour afficher le programme Linear (pour comprendre à quoi correspondent les pseudo-registres)
— `-s /dev/null` pour déclencher la génération de code assembleur (et donc l'allocation de registres), mais sans conserver le résultat. On aurait pu donner l'option `-s monfichier.s` pour obtenir l'assembleur généré sous forme textuelle dans `monfichier.s` ou bien `-o monfichierexecutable` pour générer un fichier exécutable.

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 switch install 4.07.0
$ opam install menhir
$ eval $(opam env)
```

Pour le TP6, vous aurez besoin d'outils spécifiques :

— `riscv64-unknown-elf-gcc` : un *cross-compilateur* pour RISC-V pour générer des exécutables RISC-V,

— `qemu-riscv64` : un émulateur de RISC-V pour les exécuter.

Sur des Debian récentes :

```
$ 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...

```
# 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
```