TP 4 & 5: Génération de code Risc-V

Lors de ces deux séances de TP, vous allez utiliser vos résultats du TP3 sur la construction d'un arbre de syntaxe abstraite (*Abstract Syntax Tree*, ou AST) et compiler celui-ci vers de l'assembleur RISC-V. Nous vous fournissons, comme pour les TPs précédents, une partie du code du compilateur. Le travail que vous aurez à réaliser est le suivant : finir la construction et l'évaluation de l'AST (TP3) et générer du code RISC-V à partir de cet AST. Nous vous proposons à la fin de ce TP des pistes d'amélioration pour votre compilateur. Ce TP fera l'objet d'une évaluation et constituera votre note pour la mineure de compilation.

1 Description du code fourni

1.1 Suite de tests

Comme pour le TP3, vous trouverez une suite de tests dans le répertoire `expr/tests` sous la forme de fichiers `testXX.e`. Nous vous encourageons à écrire vos propres fichiers de test.

1.2 Options du compilateur

L'exécutable `cstar`, généré lorsque vous lancez la commande **make**, accepte un certain nombre d'options.

— L'option `-c <file>` analyse le fichier `<file>` (un fichier `.e`) et génère un AST, en utilisant la fonction `parse_S()` définie depuis les actions vous avez écrites dans la grammaire pour le non-terminal `S`.

— L'option `-show-ast <dotfile>` doit être placée après l'option `-c <file>`, et prend en paramètre le nom d'un fichier `<dotfile>` dans lequel sera écrit un fichier au format DOT qui représente votre AST. Pour visualiser ce fichier sous la forme graphique, vous pouvez :

1. lancer la commande `dot -Tpng <dotfile> -o <pngfile>`, puis visualiser le fichier `<pngfile>` avec votre visionneur d'images préféré ; ou
2. copier-coller le contenu de ce fichier dans `http://www.webgraphviz.com/` et cliquer sur le bouton **Generate Graph !**

Cet outil devrait vous aider à terminer la création de votre AST et déboguer les inévitables erreurs que vous aurez introduites.

— L'option `-dump-ast` doit être placée après l'option `-c <file>`. Cela affiche votre AST en utilisant l'afficheur d'AST que vous avez écrit dans le TP3.

— L'option `-eval-ast`, sans paramètre, doit être placée après l'option `-c <file>`. Cela affiche le résultat de l'évaluation de votre AST, en utilisant la fonction `eval_ast()` que vous aurez écrite dans le fichier `eval_ast.c`.

— L'option `-gen-riscv`, sans paramètre, doit être placée après l'option `-c <file>`. Cela appelle la fonction `gen_riscv_prog` que vous devrez écrire dans le fichier `genriscv.c`.

— L'option `-dump-riscv`, sans paramètre, doit être placée après l'option `-c <file>`. Cela affiche une représentation textuelle du programme RISC-V généré au préalable.

— L'option `-eval-riscv`, sans paramètre, doit être placée après l'option `-c <file>`. Cela évalue votre programme RISC-V et affiche la valeur stockée dans le registre `REG_RET` à la fin de l'exécution.

1.3 Affichage de l'AST

L'option –show–ast du compilateur vous permet d'afficher de manière graphique une structure de données faite de listes et de tuples. À des fins de débogage, vous pouvez appeler la fonction dump_tree (déclarée dans dump_ast.h) pour obtenir le même effet avec un morceau d'AST ailleurs dans votre code. Vous pouvez aussi appeler la fonction show_structure qui affiche de manière textuelle les mêmes informations.

Cependant, pour que ces fonctions marchent correctement, l'AST doit être *typé* pour que l'on sache comment interpréter les feuilles de l'arbre (entier ou chaîne de caractères). Ce typage est effectué à l'aide des fonctions make_int() et make_string() qui encapsulent les entiers et les chaînes dans des structures de données interprétables par les fonctions d'affichage. Par conséquent, les actions de la grammaire doivent être modifiées pour intégrer ce typage. Par exemple, la règle pour l'affectation, que vous avez sans doute écrite comme suit lors du TP3 :

```
ASSIGN -> SYM_IDENTIFIER SYM_ASSIGN EXPR SYM_SEMICOLON { return pair($1,$3); }
```

doit à présent être écrite comme suit :

```
ASSIGN -> SYM_IDENTIFIER SYM_ASSIGN EXPR SYM_SEMICOLON { return pair(make_string($1)
    ↪ ,$3); }
```

De la même manière, la règle

```
FACTOR -> SYM_INTEGER { return pair((uint64_t*)EINT, (uint64_t*)atoi($1)); }
```

devient :

```
FACTOR -> SYM_INTEGER { return pair(make_int(EINT), make_int(atoi($1))); }
```

On obtiendra l'entier ou la chaîne encapsulée de cette manière grâce aux accesseurs int_get() et string_get(). Ainsi, int_get(make_int(i)) == i et string_get(make_string(s)) == s.

Notez que cette discipline ne doit être respectée que pour les structures que vous aurez à afficher avec les fonctions dump_tree ou show_structure. En d'autres termes, respectez cela pour votre AST (qui sera affiché), mais pas forcément pour d'autres structures dont vous aurez besoin ici et là dans le code du compilateur (pour l'état lors de l'évaluation de l'AST par exemple, ne vous souciez pas de cela).

1.4 Nouvelle interface des listes

Concernant les listes, la liste vide notée nil dans le TP3 doit aussi être encapsulée et transformée en nil(). Pour tester si une liste l est vide, on ne doit plus tester si l == NULL mais on utilisera la fonction is_empty(l). Ainsi, le parcours de liste auparavant effectué avec :

```
while(l){
  uint64_t* elt = get_elt(l);
  print(elt);
  l = next_elt(l);
}
```

devient :

```
while(!is_empty(l)){
  uint64_t* elt = get_elt(l);
  print(elt);
  l = next_elt(l);
}
```

Mettez à jour votre TP3 pour respecter cette nouvelle interface.

2 Assembleur Risc-V

Les processeurs Risc-V sont des processeurs RISC dont la conception a commencé récemment. Leur conception est publiquement documentée `https://github.com/riscv/riscv-isa-manual` Chaque choix de conception fait l'objet d'un commentaire argumenté et étayé par de nombreuses références bibliographiques du domaine. On peut dire qu'il s'agit de choix reposant sur l'expérience des soixante dernières années en matière de conception de processeur. Un de partie pris des auteurs de la conception est de maintenir une grande simplicité.

L'architecture se divise en plusieurs familles de processeurs adaptée chacune à un usage dédié :

— La famille RV32I dont la taille des registres est de 32 bits et dont le jeu d'instruction se limite aux manipulations entières.

— La famille RV32E qui est une version réduite de la famille précédente (avec moins de registres) et qui se veut bien adaptée pour des processeurs de systèmes embarqué.

— La famille RV64I dont la taille des registres est de 64 bits et dont le jeu d'instruction se limite aux manipulations entières. Il s'agit d'une extension de la famille RV32I. Toutes les instructions présentes dans le jeu d'instruction RV32I font aussi partie du jeu d'instruction de cette famille. Ces instructions manipulent les 64 bits des registres (au lieu de 32). Il existe de nouvelles instructions supplémentaires qui permettent de ne manipuler que les 32 bits de poids faibles.

— Il existe encore de nombreuses extensions du jeu d'instruction de base (RV32I). Mais elles ne sont pas toutes aussi bien finalisées que celles citées plus haut.

Nous allons nous concentrer sur le jeu d'instruction RV64I (car le langage C⋆ manipule des entiers 64 bits). Cependant, comme ce jeu d'instruction coïncide exactement avec le jeu d'instruction RV32I tant que l'on manipule des entiers sur la longueur complète de l'architecture sous-jacente (ici 64 bits donc), nous pouvons dire que nous documentons aussi le jeu d'instruction RV32I en même temps. Néanmoins, nous n'avons pas suffisamment de temps pour réaliser un compilateur complet, donc nous nous restreindrons à un jeu d'instructions simplifié.

Dans notre processeur simplifié les registres mesurent 64 bits de longueur et sont en nombre infini (ce qui est notablement plus simple) ! Ils portent des noms symboliques pour les trois premiers :

— r_0 s'appelle zero car son contenu est toujours égal à zéro même si on tente de le modifier (ce qui ne déclenche aucune erreur) ;

— r_1 s'appelle a_0 et comme dans l'ABI Risc-V il contiendra la valeur de retour d'une fonction.

— r_2 s'appelle sp et sert de pointeur de pile. La pile augmente vers les adresses basses comme sur les processeurs Intel.

— Les autres registres commencent à r_3 et servent à stocker des valeurs temporaires.

— Un registre dédié appelé pc contient le pointeur d'instruction.

Ceci constitue une abstraction du processeur Risc-V qui lui comporte 32 registres de 64 bits.

2.1 Les instructions

2.1.1 Chargement de constantes

Afin de stocker une constante dans un registre nous supposerons avoir à notre disposition une instruction du type const rd, imm. Cette instruction permet de charger directement une valeur immédiate encodée sur 64 bits dans un registre.

Notez que sur le processeur Risc-V réel ceci n'est pas possible. Pour charger une valeur encodée sur 64 bits, il faut procéder en plusieurs étapes. En effet les instructions sont encodées sur des mots de 32 bits. Et par conséquent, elles ne peuvent donc pas contenir de constante aussi grande qu'une valeur sur 64 bits. D'ailleurs la plus grande constante que l'on peut passer à une instruction Risc-V réelle est de taille 20 bits via l'instruction appelée lui (comme *Load Upper Immediate*). Cette instruction couplée avec une addition qui fixera les 12 bits de poids faible, permet de charger les 32 bits de poids faible d'un registre. Si l'on veut charger 64 bits il faudra procéder à un décalage des 32 bits de poids faible vers les bits de poids fort et repositionner les 32 bits de poids faible de la même manière (lui suivi de add).

2.1.2 Instructions arithmétiques

Il est possible de procéder à l'addition et la soustraction de deux registres via les instructions add rd, rs1, rs2 et sub rd, rs1, rs2. Leur sémantique est assez évidente et n'est pas détaillée davantage. Notez que comme sur le processeur Risc-V réel, il n'est pas prévu d'indiquer les dépassements via un registre de contrôle.

On peut aussi procéder à la multiplication de deux registres via l'instruction mul rd, rs1, rs2. Cette instruction remplit les bits de poids faibles du résultat de la multiplication dans le registre de destination. On supposera ici que l'on se contente de faire de petites multiplications qui ne dépassent pas la capacité du registre de destination. Dans le processeur Risc-V réel, une seconde instruction appelée mulh est disponible qui permet de calculer les bits de poids forts.

2.1.3 Instructions de transfert

Il est possible de lire et d'écrire vers la mémoire centrale via deux instructions dédiées notées respectivement ld rd, offset, rs (comme *load* pour lire) et st, rs1, offset, rs2 (*store* pour écrire). La sémantique de ces deux instructions est la suivante :

— ld rd, offset, rs charge dans le registre rd le contenu situé à l'adresse rs+offset.

— st, rs2, offset, rs1 charge dans la mémoire centrale à l'adresse rs1+offset le contenu du registre rs2.

Notez que ces deux instructions font partie du jeu d'instruction du processeur Risc-V réel. Dans ce cas, le décalage en mémoire (offset) est encodé sur 12 bits seulement. Ceci permet donc d'obtenir des valeurs situées sur la pile par exemple pour des décalages raisonnables. Pour des décalages plus importants il faut recourir à des chargements explicites de constantes et des additions explicites pour obtenir l'adresse voulue, puis à utiliser les instructions de lecture et d'écriture en mémoire avec des décalages nuls et l'adresse calculée précédemment comme base. Pour notre processeur virtuel, nous ne restreindrons pas la taille du décalage.

2.1.4 Instructions de branchement inconditionnel

Nous allons introduire des instructions permettant explicitement de définir des labels dans le code. Sur le processeur Risc-V réel, de telles instructions n'existent pas bien sûr, mais elle sont supportées par tous les assembleurs Risc-V. Nous définissons donc l'instruction label nom qui permet de positionner un nom symbolique (ou label) pour l'instruction qui la suit immédiatement. Dans notre processeur Risc-V virtuel il est alors possible de sauter inconditionnellement vers un label qui aura été défini via l'instruction jmp label.

Notez que dans le processeur Risc-V réel cette instruction s'appelle en fait jal (comme *Jump And Link*) et s'utilise comme suit : jal rd, offset. Sa sémantique est la suivante :

1. Elle charge la valeur pc + 4 dans rd. Ceci correspond à la sauvegarde de l'adresse de retour de la fonction (pour un appel de fonction) dans le registre rd puisque chaque instruction est encodée sur 32 bits (4 octets).
2. Puis elle poursuit l'exécution à l'adresse pc + offset × 2. Le décalage est signé et encodé sur 20 bits. Il permet donc d'atteindre en une instruction, des instructions situées à 1Mo de part et d'autres du pointeur d'instruction. Ceci est conçu pour favoriser les codes indépendants de leur position (PIE *Position Independant Executable*, cf le cours de connaissance de la menace sur les attaques par débordement de tampon).

On peut donc dire qu'un saut inconditionnel qui ne concernerait donc pas un appel de fonction (l'instruction jmp pour INTEL) est équivalente à jal zero, offset.

2.1.5 Instructions de branchement conditionnel

Notre processeur RISC-V virtuel fournit les instructions de branchement conditionnel suivantes :

— beq rs1, rs2, label qui permet de sauter à l'instruction portant le label label lorsque rs1 = rs2.
— bne rs1, rs2, label qui permet de sauter à l'instruction portant le label label lorsque rs1 ≠ rs2.
— blt rs1, rs2, label qui permet de sauter à l'instruction portant le label label lorsque rs1 < rs2.
— ble rs1, rs2, label qui permet de sauter à l'instruction portant le label label lorsque rs1 ≤ rs2.
— bgt rs1, rs2, label qui permet de sauter à l'instruction portant le label label lorsque rs1 > rs2.
— bgt rs1, rs2, label qui permet de sauter à l'instruction portant le label label lorsque rs1 ≥ rs2.

Sur le processeur RISC-V réel ces instructions sont aussi disponibles mais le label doit être indiqué comme un décalage relatif au pointeur d'instruction (registre pc). Il est alors encodé sur 12 bits signé et multiplié par deux avant ajout au registre pc. Ceci permet d'atteindre des instructions à 4ko de part et d'autre du pointeur d'instruction.

2.1.6 Pseudo-instructions

Comme le jeu d'instruction du processeur RISC-V est assez réduit, la plupart des assembleurs RISC-V proposent des pseudo-instructions qui se comportent comme des macro-instructions qui seront dépliées à la compilation en leurs équivalents en instruction de base. Par exemple, l'instruction nop n'existe pas dans le jeu d'instruction RISC-V mais peut être définie comme add zero, zero, zero. De même l'instruction mov rd, rs qui permet de transférer le contenu du rs vers le registre rd, peut se définir comme add rd, rs, zero.

Pour le processeur virtuel RISC-V que nous vous fournissons nous définissons l'instruction textsfmov rd, rs.

2.2 Le langage Risc-V

Vous allez écrire des programmes RISC-V, sous la forme d'une liste d'instructions. Les instructions seront encodées en C, grâce à des fonctions qui vous sont fournies, déclarées dans le fichier riscv.h.

2.2.1 Registres Risc-V

Les programmes que vous générerez pourront utiliser un nombre illimité de registres temporaires, contrairement aux *vrais* programmes RISC-V qui ne peuvent en manipuler qu'un nombre fini. Les registres auxquels vous avez accès sont REG_RET qui servira à stocker la valeur de retour

de votre programme et le registre REG_SP qui contiendra le pointeur de pile. Les entiers plus grands que 3 représentent d'autres registres, que vous pourrez utiliser.

Les registres peuvent être affichés en utilisant la fonction print_reg(r).

2.2.2 Instructions Risc-V

Vous pourrez générer des instructions grâce aux fonctions dont le prototype est donné ci-dessous.

```
// Arithmetic instructions
uint64_t* make_const(uint64_t rd, uint64_t i);
uint64_t* make_add(uint64_t rd, uint64_t rs1, uint64_t rs2);
uint64_t* make_mul(uint64_t rd, uint64_t rs1, uint64_t rs2);
uint64_t* make_sub(uint64_t rd, uint64_t rs1, uint64_t rs2);

// Move instruction
uint64_t* make_move(uint64_t rd, uint64_t rs);

// Load and store instructions
uint64_t* make_load(uint64_t rd, uint64_t rs, uint64_t i);
uint64_t* make_store(uint64_t rd, uint64_t i, uint64_t rs);

// Label instruction
uint64_t* make_label(uint64_t lab);

// Branch instructions
uint64_t* make_beq(uint64_t rs1, uint64_t rs2, uint64_t lab);
uint64_t* make_blt(uint64_t rs1, uint64_t rs2, uint64_t lab);
uint64_t* make_ble(uint64_t rs1, uint64_t rs2, uint64_t lab);
uint64_t* make_bgt(uint64_t rs1, uint64_t rs2, uint64_t lab);
uint64_t* make_bge(uint64_t rs1, uint64_t rs2, uint64_t lab);
uint64_t* make_bne(uint64_t rs1, uint64_t rs2, uint64_t lab);
uint64_t* make_jmp(uint64_t lab);
```

La convention utilisée dans le nommage des paramètres permet de comprendre la sémantique des instructions dans la plupart des cas.

- un paramètre rd représente le numéro d'un **r**egistre, utilisé comme une **d**estination pour l'opération ;
- un paramètre rs représente le numéro d'un **r**egistre utilise comme la **s**ource d'une opération ;
- un paramètre lab représente le numéro d'un label.

La fonction make_load(rd, rs, i) génère une instruction rd $\leftarrow *(\text{rs} + \text{i})$, *i.e.* on lit la mémoire à l'adresse obtenue en additionant la valeur contenue dans le registre rs à l'entier i, et on stocke le résultat dans le registre rd.

La fonction make_store(rd, i, rs) génère une instruction $*(\text{rd} + \text{i}) \leftarrow \text{s}$, *i.e.* on écrit la valeur contenue dans le registre rs dans la mémoire à l'adresse obtenue en additionnant la valeur contenue dans le registre rd à l'entier i.

Les instructions peuvent être affichées en utilisant la fonction show_riscv_instr(i). Les programmes (listes d'instructions) peuvent être affichées en utilisant la fonction show_riscv_prog(p).

3 Fin du TP 3

Importez le code que vous avez écrit lors du TP3 dans ce nouveau squelette de TP. Vous devriez importer :

— votre grammaire qui remplacera le fichier expr_grammar.g ;
— le code des fonctions make_terms et make_factors dans expr/ast.c ;
— le code de votre fonction dump_ast dans expr/expr_dump.c ;
— tout votre fichier expr/expr_eval.c

Une fois que cela est fait (et votre TP3 terminé), vous devriez obtenir la sortie suivante :

```
$ cat tests/test6.e
x = 5;
y = x * 2;
x = 6 * x;
return x - y
# Votre afficheur d'AST
$ ./cstar -c tests/test6.e -dump-ast
x = 5;
y = (x) * (2);
x = (6) * (x);
return (x) - (y)
# L'afficheur d'AST générique (fourni)
$ ./cstar -c tests/test6.e -show-ast tests/test6.dot
# Pour générer un .png à partir du .dot (ou alors http://www.webgraphviz.com/)
$ dot -Tpng tests/test6.dot -o tests/test6.png
# Votre évaluation d'AST
$ ./cstar -c tests/test6.e -eval-ast
Result = 20.
```

Le fichier expr/tests/test6.png devrait ressembler à la Figure 1.

4 Génération de code

On rentre maintenant dans le travail que vous devez réaliser. Il s'agit de générer un programme RISC-V à partir d'un AST. On dispose, dans notre langage RISC-V, d'un nombre illimité de registres pour stocker des valeurs intermédiaires. Ces registres sont représentés par des entiers supérieurs ou égaux à 3. (Les entiers 0, 1 et 2 sont réservés respectivement pour le registre zero, qui vaut toujours 0, le registre de retour, qui s'appelle a1 en RISC-V et qu'on appelle REG_RET dans le cadre de ce TP, et le registre de pile, qui s'appelle sp et qu'on appelle REG_SP dans ce TP.)

Pour le programme ci-dessous :

```
x = 3 + 7;
y = 5 * x;
return x + y
```

le code généré pourrait ressembler à cela :

```
r3 ← 3
r4 ← 7
r5 ← add(r3,r4)
*(sp - 8) ← r5
r6 ← 5
r7 ← *(sp - 8)
r8 ← mul(r6,r7)
*(sp - 16) ← r8
r9 ← *(sp - 8)
r10 ← *(sp - 16)
```

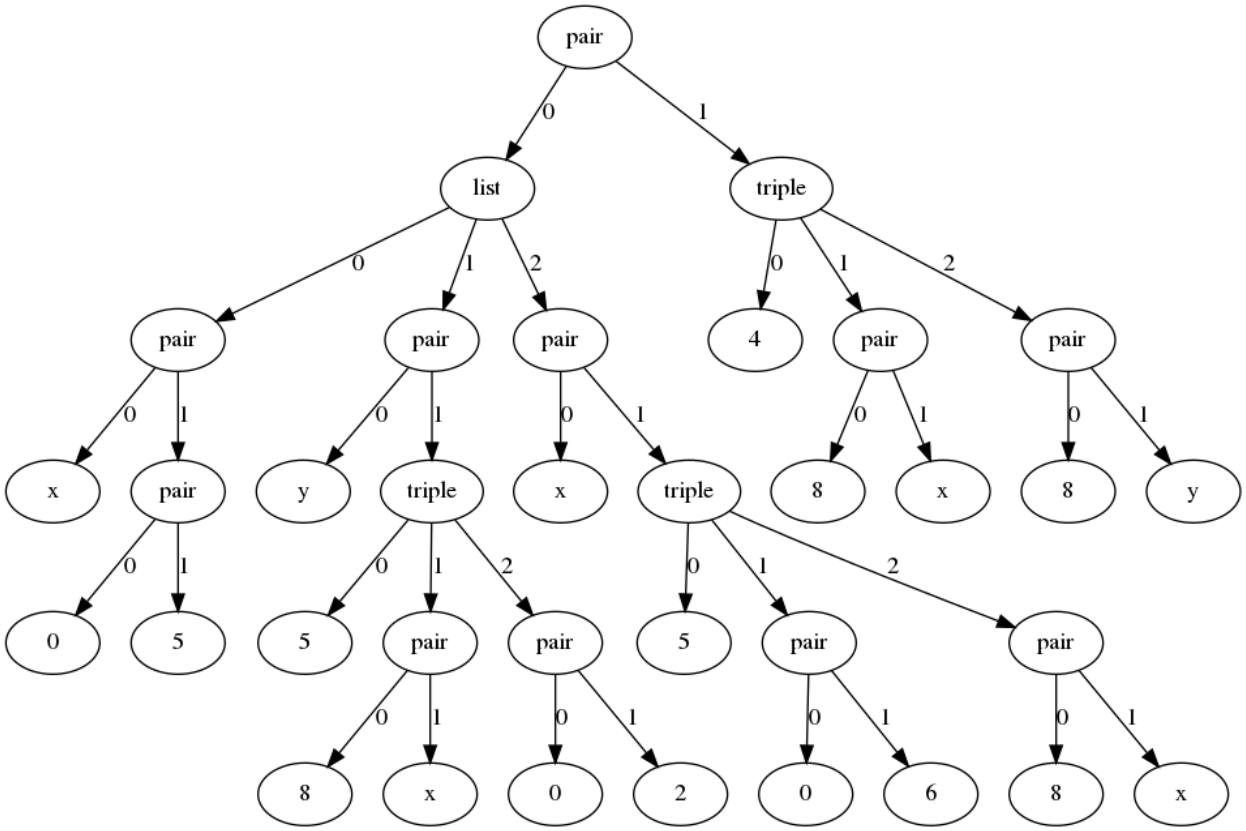

FIGURE 1 – L'AST du fichier `tests/test6.e` sous forme graphique

```
r11 ← add(r9,r10)
ret ← r11
```

On peut aussi constater que dans ce cas on n'a strictement besoin que de trois registres temporaires pour effectuer toutes les opérations. Le code obtenu serait est alors le suivant :

```
r3 ← 3
r4 ← 7
r5 ← add(r3,r4)
*(sp − 8) ← r5
r3 ← 5
r4 ← *(sp − 8)
r5 ← mul(r3,r4)
*(sp − 16) ← r5
r3 ← *(sp − 8)
r4 ← *(sp − 16)
r5 ← add(r3,r4)
ret ← r5
```

On ne demande pas que vous parveniez à un tel résultat.

4.1 Variables et table des symboles

Les variables de notre programme (représentées par des identifants dans notre langage) seront stockées sur la pile. Chaque variable aura un emplacement déterminé sur la pile. Cet emplacement sera maintenu dans une table des symboles. La table de symboles sera représentée par une liste

d'associations symtab qui associe chaque variable à un emplacement de la pile. Dans l'exemple ci-dessus, symtab = [("x",−8),("y",−16)] puisque la variable x est stockée sur la pile à l'offset -8 par rapport au pointeur de pile, et la variable y est stockée sur la pile à l'offset -16 par rapport au pointeur de pile.

Activité 1. Écrivez une fonction find_var(uint64_t* var) qui cherche dans la table des symboles l'offset associé à l'identifiant de variable var. Si la variable est présente dans la table des symboles, retournez l'offset associé. Sinon, ajoutez la variable à la table des symboles avec un nouvel offset, et retournez-le.

4.2 Compilation des expressions

Vous aurez besoin de générer des nouveaux noms de registres pour placer les résultats intermédiaires de vos expressions. Pour ce faire, vous pourrez utiliser la fonction new_temporary(), définie dans le fichier genriscv.c.

Activité 2. Écrivez une fonction compile_expr qui prend une expression (sous forme d'un AST) et qui émet des instructions pour calculer cette expression. La fonction devra retourner une paire composée de

1. une liste d'instructions ;
2. le nom du registre dans lequel le résultat est stocké.

4.3 Compilation des affectations

Activité 3. Écrivez une fonction compile_assign qui prend un identifiant de variable var et une expression e, et qui émet des instructions pour l'affectation var = e;. Cette fonction rend une liste d'instructions.

4.4 Compilation du retour

La valeur de retour est, par convention, stockée dans le registre RISC-V a0. Dans notre TP, ce registre est REG_RET.

Activité 4. Écrivez une fonction compile_return qui prend une expression et émet du code pour calculer cette expression et stocke le résultat dans le registre REG_RET. Cette fonction retourne une liste d'instructions.

4.5 Compilation d'un programme

Activité 5. Écrivez une fonction compile_program qui prend un programme entier (l'AST issu de l'analyseur syntaxique) et compile toutes ses affectations et son instruction **return**. Cette fonction retourne une liste d'instructions (*i.e.*, un programme RISC-V).

Vous pouvez à présent utiliser les options du compilateur cstar pour tester votre génération de code.

```
$ ./cstar -c tests/test6.e -gen-riscv -dump-riscv
r3 <- 5
*(sp + -8) <- r3
r4 <- *(sp + -8)
r5 <- 2
```

```
r6 <- mul(r4,r5)
*(sp + -16) <- r6
r7 <- 6
r8 <- *(sp + -8)
r9 <- mul(r7,r8)
*(sp + -8) <- r9
r10 <- *(sp + -8)
r11 <- *(sp + -16)
r12 <- sub(r10,r11)
ret <- r12
$ ./cstar -c tests/test6.e -gen-riscv -eval-riscv
Result = 20.
```

L'option -debug-riscv vous permet d'avoir un aperçu de l'état des registres entre chaque instruction.

```
$ ./cstar -c tests/test6.e -gen-riscv -debug-riscv -eval-riscv
sp = 800
Executing r3 <- 5
sp = 800        r3 = 5
Executing *(sp + -8) <- r3
sp = 800        r3 = 5
Executing r4 <- *(sp + -8)
sp = 800        r3 = 5  r4 = 5
Executing r5 <- 2
sp = 800        r3 = 5  r4 = 5  r5 = 2
Executing r6 <- mul(r4,r5)
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10
Executing *(sp + -16) <- r6
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10
Executing r7 <- 6
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10 r7 = 6
Executing r8 <- *(sp + -8)
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10 r7 = 6  r8 = 5
Executing r9 <- mul(r7,r8)
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10 r7 = 6  r8 = 5  r9 = 30
Executing *(sp + -8) <- r9
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10 r7 = 6  r8 = 5  r9 = 30
Executing r10 <- *(sp + -8)
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10 r7 = 6  r8 = 5  r9 = 30 r10 = 30
Executing r11 <- *(sp + -16)
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10 r7 = 6  r8 = 5  r9 = 30 r10 = 30       r11 = 10
Executing r12 <- sub(r10,r11)
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10 r7 = 6  r8 = 5  r9 = 30 r10 = 30       r11 = 10
    ↪ r12 = 20
Executing ret <- r12
sp = 800        r3 = 5  r4 = 5  r5 = 2  r6 = 10 r7 = 6  r8 = 5  r9 = 30 r10 = 30       r11 = 10
    ↪ r12 = 20      ret = 20
Result = 20.
```

5 Pistes d'améliorations

Vous trouverez ci-dessous des pistes d'amélioration pour votre compilateur, qui visent à enrichir le langage source du compilateur. Les indications sont volontairement imprécises pour vous laisser la liberté d'implémenter ces améliorations.

5.1 Conditionnelles et boucles ★

Notre langage d'expressions est plutôt restrictif. Ajoutez-y des structures de contrôle (conditionnelles, boucles). Vous aurez besoin pour cela de modifier la grammaire du langage, la génération de l'AST, l'afficheur de programmes, l'évaluation des AST et finalement le générateur de code. Pour la génération de code, vous aurez besoin d'introduire des labels et des instructions de branchement. Celles-ci vous sont présentées dans la section 2.2.2 de ce sujet.

Testez petit à petit vos modifications (d'abord simplement le parseur, puis seulement l'afficheur, etc.).

5.2 Paramètres ★★

Une autre piste d'amélioration serait d'ajouter des paramètres à notre programme. En effet, pour le moment, la valeur de retour des programmes est connue à la compilation et les programmes pourraient donc être réécrits en un simple retour d'un entier.

Commencez par considérer un seul paramètre de votre programme, par exemple `x`. L'évaluation d'AST prendra donc un paramètre en plus, à savoir la valeur initiale de `x`. Pour la génération de code, on supposera que la valeur de `x` est passée sur la pile, à un offset +8 par rapport à `sp`.

Il vous faudra modifier l'évaluation des AST, l'évaluation des programmes RISC-V et la gestion des options dans `main-cstar.c` pour pouvoir passer un paramètre à l'exécution, via une nouvelle option `-param` par exemple.

5.3 Allocation de registres ★★★

Nous l'avons précisé, notre langage autorise un nombre illimité de registres. Or, le vrai langage RISC-V ne propose que 7 registres temporaires, nommés `t0` à `t6`. En pratique, si ces 7 registres ne suffisent pas, les valeurs qui ne rentrent pas dans les registres sont stockées sur la pile. Transformez la génération de code pour les expressions pour optimiser l'utilisation des registres, et utiliser éventuellement des emplacements de pile si nécessaire.