IFT209 –Programmation système

Devoir 5

Enseignants: Mikaël Fortin et François Janson
Date de remise: indiquée sur Turnin
Modalités de remise: Turnin
À réaliser: personne seule ou en équipe de deux

Le but de ce devoir est de se pratiquer avec la récursivité et les chaînes de bits.

Problème. Un compilateur est un programme qui traduit les énoncés d'un langage de haut niveau en une suite d'instructions machine. Après avoir vérifié la validité des énoncés et leur sens (analyses lexicales et syntaxiques), le compilateur produit habituellement une structure nommée ***arbre syntaxique***.

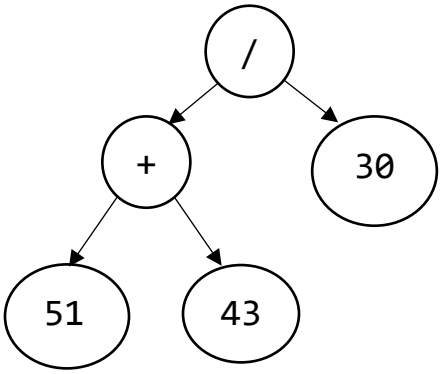
L'arbre syntaxique est ensuite parcouru durant la génération de code pour produire les instructions machine qui constitueront le programme compilé.

Langage à compiler. Les expressions arithmétiques simples comprenant les opérateurs `+`, `-`, `*` (multiplication) et `/` (quotient) peuvent être transformées en arbre syntaxique qui organise les opérations selon leur priorité. Les noeuds de l'arbre représentent des opérations, alors que les feuilles représentent des opérandes (nombres entiers).

Lorsque le code est généré pour une architecture à 0 opérandes (machine à pile), la suite d'instructions résultante peut être obtenue lors d'un parcours en profondeur de l'arbre syntaxique. On parcourt d'abord tous les noeuds enfants d'un noeud. Lorsqu'on rencontre une feuille, on génère une instruction d'empilement (PUSH). Lorsqu'on a terminé tous les noeuds enfants d'un noeud, on génère l'instruction arithmétique correspondant (ADD, SUB, MUL, ou DIV). Les opérations sont donc organisées en ordre inverse de priorité en partant de la racine (celles plus proches des feuilles doivent être faites en premier).

Exemple de parcours. Soit l'expression arithmétique `(51 + 43) / 30`

L'arbre syntaxique résultant serait:



Un parcours en profondeur de l'arbre (les enfants d'abord, et l'enfant de gauche en premier), donnerait la suite d'actions suivantes et les instructions correspondantes:

Ordre	1	2	3	4	5	6	7
Action	/ (début)	+ (début)	51	43	+ (fin)	30	/ (fin)
Instruction	Aucune	Aucune	PUSH 51	PUSH 43	ADD	PUSH 30	DIV

On fait donc une opération uniquement lorsque toutes les sous-opérations (celles qui sont dans les sous-arbres de gauche ou de droite) sont résolues. À l'entrée dans un noeud d'opération, il n'y a aucune instruction à générer. On génère celle-ci à la sortie d'un noeud, quand on a terminé de parcourir les sous-arbres.

Si l'on prend le programme résultant et qu'on l'exécute sur une machine à pile, on obtient:

Instruction	**État de la pile**	**Modifications à la pile**
État initial	Dessus de la pile	Pile initialement vide.
PUSH 51	Dessus de la pile 51	Empile le nombre 51
PUSH 43	Dessus de la pile 43 51	Empile le nombre 43
ADD	Dessus de la pile 94	Dépile 43 et 51, calcule 51+43, puis empile 94
PUSH 30	Dessus de la pile 30 94	Empile le nombre 30
DIV	Dessus de la pile 3	Dépile 30 et 94, calcule 94 / 30, puis empile 3

Remarques.

— Les opérations non-commutatives (SUB et DIV) assument que l'opérande de gauche est empilé en premier.

— L'instruction DIV est un quotient, le résultat est donc toujours entier.

Tâche. Vous devez écrire un sous-programme en langage d'assemblage ARMv8 nommé `Compile` qui parcourt un arbre syntaxique reçu en paramètre et génère les instructions machine correspondantes sous la forme d'une suite d'octets.

Vous recevrez en paramètre:

- `x0`: l'adresse du noeud racine de l'arbre à compiler;
- `x1`: l'adresse d'un tableau d'octets dans lequel vous écrirez les instructions compilées

Vous retournerez comme résultat:

- `x0`: la taille du programme généré en octets;
- (effet de bord) le tableau d'octets aura été modifié.

Votre sous-programme sera appelé par un programme principal écrit en C++. Celui-ci réalise la lecture de l'expression arithmétique au clavier, puis la production de l'arbre syntaxique que vous allez compiler. Il appelle ensuite votre sous-programme, et selon la taille du programme que vous produisez, il affiche les octets du tableau en hexadécimal.

Les octets générés doivent suivre la spécification d'une machine à pile virtuelle (document **specification.pdf**). Vos programmes résultants seront testés sur cette machine à pile.

Structure Noeud. L'arbre syntaxique est constitué de noeuds interreliés. Ce sont des structures simples en mémoire contenant l'information suivante:

- Le type de noeud (opérateur ou nombre);
- La valeur du noeud (numéro d'opérateur ou valeur du nombre);
- Adresse du noeud enfant de gauche (si opérateur)
- Adresse du noeud enfant de droite (si opérateur).

Les noeuds qui contiennent des nombres sont des feuilles et n'ont donc pas d'enfant de gauche ou de droite, les valeurs de leurs adresses sont donc NULL (0).

Les valeurs associées aux opérateurs sont les suivantes:
0 : addition
1 : soustraction
2 : multiplication
3 : division (quotient)

Les valeurs admissibles pour les nombres sont de 0 à 32767 inclusivement. Seuls des entiers positifs respectant cette contrainte peuvent se retrouver dans les feuilles.

En C++, la structure est définie comme ceci dans le fichier `analyse.h`:

```
struct Noeud
{
 int type;
 int valeur;
 Noeud* gauche;
 Noeud* droite;
};
```

Le type `int` occupe 4 octets, alors que le type `Noeud*` occupe 8 octets. La structure totale occupe donc 24 octets. Ceci implique qu'à partir de l'adresse d'un noeud, telle que celle que vous recevez dans `x0`:

- Les octets 0 à 3 sont le type de noeud;
- Les octets 4 à 7 sont la valeur du noeud;
- Les octets 8 à 15 sont l'adresse de l'enfant de gauche;
- Les octets 16 à 23 sont l'adresse de l'enfant de droite.

Algorithme. Le parcours d'un arbre est habituellement réalisé récursivement. Le cas général survient lorqu'un noeud a des enfants, impliquant un appel récursif par enfant. Le cas d'arrêt survient lorsqu'on rencontre une feuille.

Le pseudo-code suivant résume l'algorithme de parcours:

```
int CompileRec (Noeud, tableau)
{
      si Noeud contient un nombre (vérifier type de noeud, octets 0 à 3)
      alors
            -produire l'instruction PUSH nombre. Le nombre est la valeur du noeud
            (octets 4 à 7).

            -Écrire les octets générés dans le tableau de résultats.

            -Retourner la taille de l'instruction PUSH (3 octets).

      sinon
            -Appeler CompileRec avec le noeud de gauche (octets 8 à 15).
            Le début du tableau où écrire le résultat reste celui reçu en paramètre.

            -Appeler CompileRec avec le noeud de droite (octets 16 à 23).
            Le début du tableau où écrire le résultat est déplacé de la taille du
            programme produit par la compilation du sous-arbre de gauche.

            -Générer l'instruction correspondant à l'opération (octets 0 à 3), puis
            l'écrire dans le tableau après les résultats de compilation des sous-
            arbres de gauche et de droite.

            -Retourner la taille du code produit qui est la taille du code produit
            par le sous-arbre de gauche, plus celle du code produit par le sous-arbre
            de droite, plus 1 (la taille de ADD, SUB, MUL ou DIV).
}
```

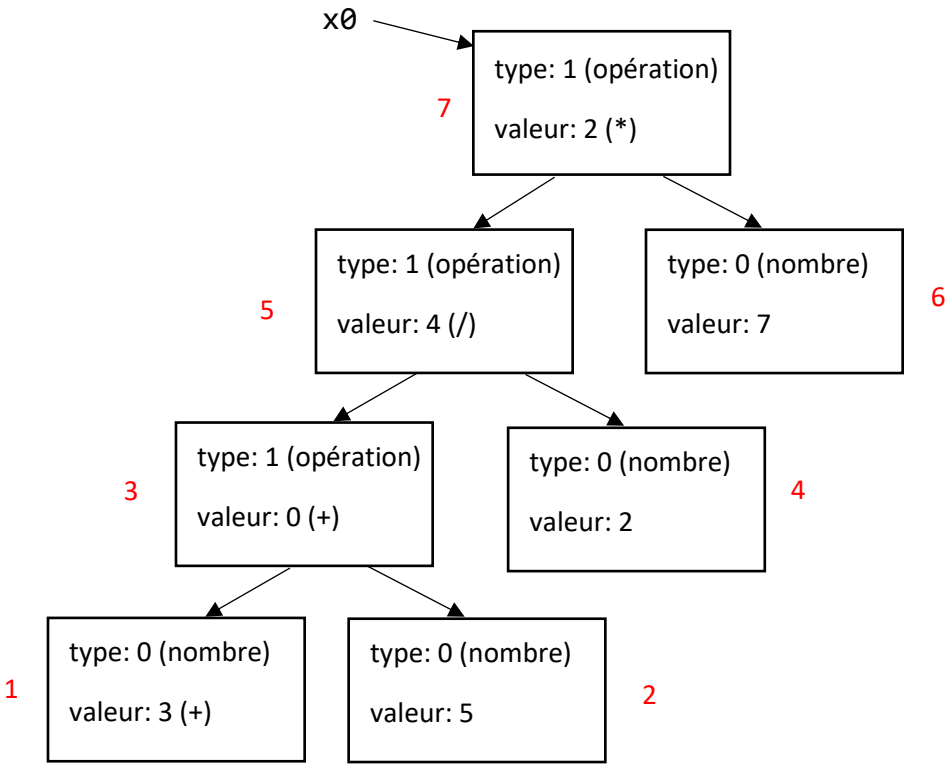
Votre programme doit également produire deux instructions qui le finalisent: `WRITE %d` et `HALT`.
La première sert à afficher le résultat final, la seconde, signaler que le programme est terminé. Ces instructions doivent être écrites dans le tableau d'octets après le programme compilé. Pour ajouter ces instructions, il est recommandé de faire une coquille, c'est-à-dire, une fonction qui a pour rôle d'appeler la fonction récursive. Une fois que celle-ci a écrit le programme dans le tableau et a retourné sa taille, vous pouvez écrire les deux instructions restantes dans le tableau après le programme généré. L'étiquette `Compile` sera donc celle de la coquille, pas celle de la fonction récursive, que vous pouvez appeler `CompileRec`.

Exemple complet. Soit la ligne de commande suivante:

```
tp5 < tests/input2
```

Le fichier `input2` contient l'expression arithmétique ((3+5)/2)*7

L'arbre syntaxique résultant avec l'ordre dans lequel les instructions seront générées lors de votre parcours indiqué en rouge:



Un parcours de l'arbre devrait donner la suite d'instructions suivantes, chacune compilée et placée dans le tableau dans l'ordre de leur génération:

Ordre	1	2	3	4	5	6	7	Coquille
Instruction	`PUSH 3`	`PUSH 5`	`ADD`	`PUSH 2`	`DIV`	`PUSH 7`	`MUL`	`WRITE %d, HALT`
Compilée (hexadécimal)	`40 00 03`	`40 00 05`	`48`	`40 00 02`	`54`	`40 00 07`	`50`	`21 00`
Taille en octets retournée	3	3	3+3+1 = 7	3	7+3+1 = 11	3	11+3+1 = 15	15+2 = 17
Position dans le tableau	0	3	6	7	10	11	14	15

Le tableau d'octets résultant serait donc:

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Valeur	40	00	03	40	00	05	48	40	00	02	54	40	00	07	50	21	00

Rappel. Déterminez le code binaire (hexadécimal) des instructions compilées à l'aide du document de spécification de la machine à pile (**specification.pdf**).

Tests. Plusieurs tests sont disponibles dans le dossier `/tests`. Ils viennent en 2 parties: les fichiers nommés `input` sont les entrées à fournir à votre programme. Les fichiers nommés `res` sont les résultats attendus. Vous pouvez de plus tester le déroulement d'un programme que vous avez généré avec l'exécutable `machine`. Par exemple:

```
tp5 < tests/input2 > monresultat
machine < monresultat
```

La première ligne utilise le fichier `/tests/input2` comme entrée de votre programme place le résultat dans un fichier (`monresultat`). Ce résultat devrait être comparable au fichier `/tests/res2`. La seconde ligne utilise ce résultat comme entrée pour le programme `machine`.

Assurez-vous que l'exécutable `machine` a les permissions d'exécution avant de tester. Si ce n'est pas le cas, entrez la commande suivante:

```
chmod +x machine
```

Suggestion. Vous pouvez subdiviser le travail en plusieurs étapes.

1) Commencez par écrire un algorithme qui parcourt récursivement l'arbre syntaxique. À chaque noeud visité, utilisez `printf` pour afficher des informations sur le noeud. Pour déterminer si votre ordre de parcours générerait le bon programme, vous pouvez afficher, selon les valeurs et les types des noeuds, les opérations correspondantes (PUSH, ADD, SUB, DIV, MUL). Retournez une taille générée de 0 pour éviter que le programme principal n'affiche le contenu du tableau.

Effet observable: Vous pouvez voir la suite d'instructions, mais celle-ci n'est pas compilée.

2) Générez les instructions compilées selon la spécification de la machine à pile, puis affichez-les en hexadécimal, octet par octet. Retournez une taille générée de 0 pour éviter que le programme principal n'affiche le contenu du tableau.

Effet observable: Votre programme affichera la suite d'instructions presque complète, mais pas la taille ni les instructions WRITE et HALT.

3) Écrivez les octets générés dans le tableau résultant et retournez la taille de programme compilé correctement à chaque étape. Retirez vos affichages avec `printf`.

Effet observable: Votre programme devrait afficher le programme compilé correctement avec la bonne taille sauf pour WRITE et HALT.

4) Implantez la coquille qui ajoute WRITE et HALT à la fin du tableau résultant.

Effet observable: Votre programme affiche maintenant la bonne taille ainsi que toutes les instructions du programme compilé.

Contraintes.

- Modifiez uniquement `tp5.as`;
- Compilez le programme original avec la commande `make`;
- Ne modifiez pas les fichiers `analyse.h, analyse.o et tp5.cc`;
- Utilisez l'exécutable `machine` pour tester vos programmes résultants.
- Comparez vos résultats à ceux prévus avec les fichiers du répertoire `/tests`.

Directives.

- Votre programme doit être obtenu en complétant le code fourni sur Turnin;
- Votre programme doit être remis dans un seul fichier nommé `tp5.as`;

Pointage. Vous pouvez obtenir jusqu'à 20 points répartis ainsi:

- 10 points si votre programme réussit correctement les tests `input1` à `input5`;
- 3 points si votre code effectue un parcours récursif de l'arbre syntaxique;
- 3 points si votre programme génère et écrit les instructions dans le tableau résultant;
- 2 points pour l'implémentation d'une coquille;
- 2 points pour la lisibilité du code (indentation, commentaires, conventions);