Compilation de C−−: phase de synthèse

Projet « Langages formels »: partie 1

Dans la deuxième partie du projet « Programmation 1 », vous avez programmé quatre phases d'analyse sémantique d'un compilateur pour un sous-ensemble du langage C, appelé C−−. La première partie du projet « Langages formels » vous demande de programmer la phase de synthèse de ce compilateur, afin de générer du code LC3 [1] à partir d'un programme C–.

Le rendu de cette partie du projet sera fait sur la forme d'une archive compressée (`.zip` ou `.tar.gz`) d'un répertoire contenant les sous-repertoires suivants :

- `src` : vous y déposerez le code OCaml de votre projet ainsi que les fichiers utilisées pour le compiler ; cette partie doit inclure le code de la phase d'analyse,
- `tst` : les programmes écrits en C−− (extension de fichier `.c`) que vous avez utilisés pour tester votre compilateur. Chaque programme doit être accompagné d'un commentaire ou d'un fichier ayant le même nom mais l'extension `.log` contenant les résultats de la phase d'analyse et d'un fichier `.lc3` contenant le code généré pour l'exemple,
- `doc` : la documentation (en format textuel Markdown ou LaTeX) qui décrit l'architecture de votre code, les choix de codage, les résultats obtenus, les tests utilisés.

✎ : Le rendu de l'archive doit être fait **avant le 14 mars 2025** sur ecampus, dans l'espace du cours « Programmation 1 ».

LC3 : langage d'une machine à registres

La machine pour laquelle est conçu le langage LC3 est une machine à registres dont les mots ont 16 bits (donc les instructions et les registres ont 16 bits aussi). La machine n'inclut pas des instructions pour gérer une pile, ce complique la compilation des fonctions récursives.

Registres. La machine dispose de 8 registres appelés R0, R1, ..., R7. Seul le registre R7 a une utilisation spéciale par les instructions d'appel (JSR) et de retour (RET) de fonction.

Instructions. Les instructions sont soit des declarations (avec réservation) de blocks en mémoire, soit des opérations de l'unité de calcul. L'adresse à laquelle se trouve chaque instruction peut être nommée par une étiquetée qui precède l'instruction.
Pour les déclarations de blocs mémoire, les instructions sont :

.ORIG début du code en mémoire,

.END fin du code à interpréter, ce qui suit est ignoré,

.FILL v réservation d'un mot (16 bits) rempli avec la valeur v,

.BLKW s v réservation de s mots initialisés avec la valeur v ; si v est absent, la valeur est 0,

1. `http://lc3tutor.org/`

.STRINGZ t réservation d'un bloc de mots de longueur égale à celle du texte t plus 1 ; le bloc contient les lettres du texte t et un mot final égal à 0.

Les valeurs entières, écrites en base 10, sont préfixées par '#' ; les valeurs texte sont écrites entre guillemets.

Pour décrire les instructions de l'unité de calcul, on utilise les notations suivantes :
- DR : registre destination, rempli avec une valeur par l'instruction,
- SR : registre lu par l'instruction,
- PC : adresse de l'instruction suivant l'instruction courante,
- mem : mémoire du programme.

Les instructions de *lecture depuis ou d'écriture dans la mémoire* sont :

LD DR, v affectation de DR à la valeur mem[PC+v],

LDI DR, v affectation de DR à la valeur mem[mem[PC+v]],

LDR DR, SR, v affectation de DR à la valeur mem[SR+v],

LEA DR, v affectation de DR à v (qui peut être une étiquette) sur 9 bits,

ST SR, v écrit la valeur de SR à mem[PC+v] (sur 9 bits),

STI SR, v écrit la valeur de SR à mem[mem[PC+v]],

STR SR, DR, v écrit la valeur de SR à mem[DR+v].

Attention, si v est une étiquette d'instruction, elle est traduite dans la différence (offset) entre l'instruction courante et l'adresse nommée par l'étiquette.

Les instructions de *calcul arithmétique ou logique* sont :

ADD DR, SR1, SR2 écrit en DR la valeur SR1+SR2 ; SR2 peut être une constante,

AND DR, SR1, SR2 écrit en DR la valeur SR1 et logique SR2 ; SR2 peut être une constante,

NOT DR, SR écrit en DR la valeur négation bit-à-bit de la valeur de SR.

Les instructions de changement de l'instruction suivante ou de *saut* sont :

BR v change PC à v sans condition,

BRn v change PC à v si la dernière valeur calculée était négative,

BRz v change PC à v si la dernière valeur calculée était zéro,

BRp v change PC à v si la dernière valeur calculée était positive ; les conditions de saut peuvent être combinées, par exemple BRnz si valeur négative ou nulle, etc.

JMP SR change PC à SR, donc la prochaine instruction exécutée sera celle à l'adresse SR,

JSR v correspond à un appel de sous-routine (fonction) qui change PC à v et l'ancienne valeur de PC est mise dans le registre R7 (pour utilisation part RET),

JSRR SR correspond à un appel de sous-routine (fonction) qui change PC à SR et l'ancienne valeur de PC est mise dans le registre R7 (pour utilisation part RET),

RET change PC à la valeur de R7.

L'instruction **HALT** arrête l'exécution du programme.

Les instructions suivantes sont des appels de sous-routines d'entrées-sorties depuis le clavier ou l'écran :

GETC lit un caractère depuis le clavier dans le registre R0,

IN lit un caractère depuis le clavier dans le registre R0 et l'affiche,

OUT écrit la valeur de R0 à l'écran,

PUTS écrit le texte à l'adresse R0 à l'écran.

Synthèse de code LC3

La génération de code LC3 peut partir soit du CFG calculé par la phase d'analyse sémantique, soit directement de l'AST. La solution partant du CFG demande moins de code car elle exploite la mise en forme CFG du programme et la simplification des expressions dans les sommets (blocs) du CFG. La solution partant de l'AST, aussi appelée *traduction dirigée par la syntaxe* est plus lourde à programmer car elle demande de refaire la traduction des expressions en expressions simplifiées. Nous vous recommandons fortement la première solution.

Dans les deux solutions, pour faciliter la génération de code, vous aurez besoin de calculer une *table de symboles* qui associe, à chaque utilisation d'un identificateur (de variable ou fonction) des informations nécessaires à la traduction. Ainsi, pour une fonction, les informations utiles sont :

— l'étiquette (adresse dans l'espace code) à la quelle se trouve le code de la fonction ; elle sera utilisée pour effectuer l'appel à la fonction (en utilisant `JSR`),
— le nombre de paramètres de la fonction,
— le nombre de variables locales à la fonction.

Les deux derniers nombres sont utilisés pour leur réserver de l'espace en mémoire à l'appel de la fonction.

Pour une variable, les informations utiles sont :

— le nom unique (calculé à la création du CFG ou par une analyse de l'AST),
— le nom de déclaration (qui peut être le même que d'autres variables),
— le type de sa déclaration (adresse ou pointeur),
— la type de déclaration : globale, paramètre de fonction ou locale à une fonction),
— l'étiquette de sa déclaration pour une variable globale, la position du paramètre ou de la variable locale des fonctions dans le bloc mémoire réservé à l'appel de la fonction.

Traduction du programme principal

Dans une première étape, vous devez vous concentrer sur la traduction du programme principal (fonction `main`) sans paramètres, en traitant uniquement les blocs d'instructions sans appels de fonction. La difficulté de cette étape est le nombre restreint de registres disponibles.

Pour traiter cette difficulté vous pouvez utiliser une mémoire supplémentaire, réservée pour le programme principal, dans laquelle sont sauvegardées (une fois) les valeurs de chaque registre. Le code suivant donne l'exemple de la déclaration d'un bloc de 8 mots permettant de sauvegarder les valeurs de registres et son utilisation pour sauvegarder R1 :

```
;------------------------
; MAIN
;------------------------
.blkw #8 ; dans l'ordre R7 R6 ... R1 R0
MAIN
; instructions de main
...
; sauvegarde de R1
LEA R0, MAIN
STR R1, R0, #-2 ; sauve R1 à MAIN-2
...
; restaure R1
LEA R0, MAIN
LDR R1, R0, #-2 ; restaure R1 depuis MAIN-2
```

L'utilisation du CFG facilite cette traduction grâce à la simplification des expressions complexes en expressions à 2 opérandes et l'utilisation de variables temporaires (registres). On suppose que chaque fonction a besoin d'au plus 8 registres [2].

✎ : Codez la phase de synthèse de code à partir de la fonction principale dans le fichier `compile2lc3.ml`. Mettez à jour le code du fichier `main.ml` pour appeler la phase de synthèse après la phase d'analyse.

Traduction d'un appel de fonction

La traduction d'un appel de fonction demande de fixer les éléments suivants, vus aussi en cours pour une machine à pile :

- *Le calcul des arguments* est fait par l'appellant, comme pour la machine à pile.
- *La transmission des arguments* à la fonction appelée. Pour une machine sans pile, la transmission des arguments utilise la solution vue ci-dessus pour la sauvegarde des registres du programme principal. Ainsi, on declare un bloc de mots avant l'étiquette de la fonction dont la taille peut être calculée grâce à la table de symboles et qui comprend :
  - une partie réservée aux valeurs des paramètres d'appel,
  - une partie réservée aux variables locales de la fonction,
  - une partie réservée à la sauvegarde des registres.

  Cette solution permet de traiter uniquement les fonctions non-récursives.
- *La mémoire pour les variables locales* est donc prévue dans le bloc ci-dessus.
- *La transmission du résultat* est faite dans le registre R0.
- *Le retour à l'appelant* est fait grâce à l'instruction RET qui utilise la valeur du registre R7. La valeur de R7 est fixée par l'instruction JSR à l'adresse de code après l'appel de fonction. Il faut donc que cette valeur ne soit pas changée dans la fonction ou au moins qu'elle soit récupérée avant l'instruction RET.

  Attention : les instructions IN, OUT, GETC et PUTS sont des appels de fonctions, donc elles changent la valeur de R7.

✎ : Codez la phase de synthèse de code de chaque fonction dans le fichier `compile2lc3.ml`. Ecrivez des tests pour illustrer la traduction des fonctions avec ou sans paramètres, avec un ou plusieurs appels, avec des chaînes d'appel incluant plusieurs fonctions, etc.

2. Le nombre maximum de registres nécessaires peut être obtenu en utilisant un algorithme de coloriage de graphes sur le graphe représentant les dépendances entre les calculs de la fonction.