TP compilation : STK

Le langage STK s'affranchit de la gestion des registres en stockant les valeurs intermédiaires et résultats de chaque calcul dans une pile :

— chaque valeur calculée est placée au sommet de la pile,
— chaque opération prend pour paramètres zéro, un ou deux éléments au sommet de la pile (et les en retire) avant de replacer son résultat éventuel au sommet.

1 Spécification

Un programme STK est un fichier texte comportant deux parties :

1. la première commence par la mention `.text` et contient des instructions,
2. la deuxième commence par la mention `.data` et contient les données du programme.

Des commentaires peuvent être présents d'une chacune de ces deux parties. Les données du programme, comme dans ASM, sont présentées sous la forme d'une alternance d'étiquettes et de valeurs entières. Les instructions sont séparées par des espaces et/ou des retours à la ligne et prennent l'une des formes suivantes :

— un identifiant suivi d'un deux-points, pour la définition d'une étiquette,
— un nombre entier n, pour placer n au sommet de la pile,
— une étiquette l, pour placer au sommet de la pile l'adresse représentée par l,
— un nom d'instruction existant dans ASM, pour réaliser l'instruction correspondante en prenant les éventuels paramètres sur la pile (ils en sont retirés au passage) et en plaçant l'éventuel résultat sur la pile, les instructions accessible de la sorte étant :
  — les instructions spéciales `NOP`, `EXIT` qui n'affectent pas la pile, et `PRINT` qui retire l'élément au sommet pour l'afficher,
  — les instructions mémoire indirectes `READ`, qui prend sur la pile une adresse a et y remet la valeur lue à cette adresse, et `WRITE`, qui prend sur la pile une valeur v et une adresse a et écrit la valeur v en mémoire à l'adresse a,
  — les instructions de saut `JUMP`, `JUMPWHEN` qui prennent sur la pile une adresse cible, et éventuellement une valeur de test,
  — les opérations arithmétiques et logiques unaires `MINUS` et `NOT`, qui prennent un paramètre sur la pile et remettent à la place un résultat,
  — les opérations arithmétiques et logiques binaires `ADD`, `SUB`, `MULT`, `DIV`, `REM`, `EQ`, `NEQ`, `LT`, `LE`, `GT`, `GE`, `AND`, `OR`, qui prennent deux paramètres sur la pile et y replacent un résultat.

  Sont donc exclues de cette liste `CONST`, `ADDRESS`, `DIRECTREAD` et `DIRECTWRITE` qui sont déjà représentables naturellement, `MOVE` qui n'a plus lieu d'être en l'absence de registres, et `INCR` et `DECR` que l'on abandonne.

2 Compilation vers ASM

La traduction d'un programme STK en un programme ASM implique plusieurs tâches.

Réaliser une pile On utilise pour la pile une zone de la mémoire commençant à la dernière adresse ($2^{16}-1$) et progressant vers le bas. On ajoute aux données du programme une donnée étiquetée `''stack_pointer''` contenant l'adresse du sommet de la pile, c'est-à-dire l'adresse du dernier élément ajouté. Lorsque la pile est vide, `''stack_pointer''` pointera à l'adresse immédiatement au-delà de l'espace de la pile, c'est-à-dire 2^{16}.

Fournir des opérations `push` **et** `pop` L'opération `push` place un nouvel élément sur la pile. Il faut donc

— décrémenter la valeur de `''stack_pointer''` pour faire grandir la pile,

— placer la valeur voulue au nouveau sommet de la pile.

L'opération pop retire l'élément au sommet de la pile. Il faut donc

— lire la valeur présente au sommet actuel de la pile,

— incrémenter la valeur de `''stack_pointer''` pour faire rapetisser la pile.

Passer de la pile aux registres Pour traduire une instruction STK qui travaille exclusivement sur la pile en une instruction ASM qui travaille exclusivement sur les registres, il faut :

— transférer les paramètres depuis la pile vers des registres de son choix,

— effectuer l'opération ASM,

— transférer le résultat depuis son registre vers la pile.

3 Objectifs du TP

3.1 Questions préalables

1. Que fait le programme ASM suivant ?

```
  CONST $r0 2
  ADDRESS $r1 cinq
  READ $r1 $r1
  ADDRESS $r2 a
  READ $r2 $r2
  MULT $r3 $r0 $r1
  ADD $r4 $r3 $r2
  PRINT $r4
  EXIT
a:
  97
cinq:
  5
```

Écrire un programme STK réalisant les mêmes calculs.

Correction. Le tableau suivant montre l'état de chaque registre après l'exécution de chacune des instructions. On note respectivement `&a` et `&cinq` les adresses désignées par les étiquettes `a` et `cinq`.

Instruction	&r0	&r1	&r2	&r3	&r4	Commentaire
`CONST $r0 2`	2					
`ADDRESS $r1 cinq`	2	`&cinq`				
`READ $r1 $r1`	2	5				
`ADDRESS $r2 a`	2	5	`&a`			
`READ $r2 $r2`	2	5	97			
`MULT $r3 $r0 $r1`	2	5	97	10		
`ADD $r4 $r3 $r2`	2	5	97	10	107	
`PRINT $r4`	2	5	97	10	107	affiche `k`
`EXIT`	2	5	97	10	107	fin

Un programme STK effectuant les mêmes calculs :

```
.text
  2 cinq READ MULT a READ ADD PRINT
  EXIT
.data
a: 97
cinq: 5
```

Détail de l'état de la pile après chaque instruction :

Instr.	2	cinq	READ	MULT	a	READ	ADD	PRINT
Pile	2	2	2	10	10	10	107	
		&cinq	5		&a	97		

2. Écrire un programme STK affichant toutes les lettres de l'alphabet, et le traduire en un programme ASM.
Correction. Le programme utilise une donnée globale nommée a et contenant le code ASCII de la lettre courante (initialisée à 97 pour a). On code ensuite une boucle à l'aide de deux étiquettes : une étiquette `loop` pour le corps de la boucle et une étiquette `test` pour le code évaluant la condition de la boucle et décidant d'effectuer ou non un tour.

```
.text
  test JUMP              # Saut au test de la boucle
loop:
  a READ PRINT           # Affichage lettre courante
  10 PRINT               # Affichage retour à la ligne
  a a READ 1 ADD WRITE   # Incrément de la donnée a
test:
  loop a READ 123 LT JUMPWHEN  # Nouveau tour si a<123
  EXIT
.data
a: 97
```

Détail de l'état de la pile pour l'exécution de la sixième ligne (lors du premier tour, au début duquel a contient encore 97).

Instr.	a	a	READ	1	ADD	WRITE
Pile	&a	&a	&a	&a	&a	
		&a	97	97	98	
				1		

3.2 Contrat de base : réalisation d'un compilateur

Objectif Réaliser, dans un unique fichier lex `STKCompiler.mll`, un compilateur de STK vers ASM suivant la stratégie décrite en section 2.

3.3 Extensions

3.3.1 Accumulateur

Vous pourrez remarquer que le code ASM généré en suivant la stratégie décrite ici fait de nombreuses opérations inutiles sur la pile. En particulier, il est courant qu'une valeur soit placée au sommet de la pile pour en être retirée immédiatement. En effet, de nombreuses

instructions STK sont traduites par une séquence terminant par une opération `push`, et sont suivies d'une autre instruction STK pour laquelle la séquence obtenur commence par une opération `pop`.

Pour éviter de placer sur la pile une valeur qui en serait retirée aussitôt, on peut décider que le résultat d'une instruction produisant une valeur ne sera pas stocké au sommet de la pile mais plutôt dans un registre dédié (qu'on appelle *accumulateur*). On peut par exemple choisir `$r0` pour cela (et utiliser par exemple `$r1` et `$r2` comme autres registres de travail lorsque c'est nécessaire). Ce résultat ne sera ensuite transféré sur la pile que lorsque c'est nécessaire.

Objectif Produire un nouveau compilateur `STKCompilerAcc.mll` qui réalise cela.

3.3.2 Allocation et registres

Étant donnée une suite d'instructions STK s'exécutant à partir d'une pile vide, il est tout à fait possible de prédire dès l'origine à quelle hauteur de la pile sera stockée chaque valeur intermédiaire [1]. Pour limiter le nombre d'opérations `push` et `pop`, on peut donc décider que toutes les valeurs qui devraient être stockées dans les 12 premières cases de la pile seront à la place stockées dans les registres `$r0` à `$r11`, et que seules les valeurs suivantes seront réellement ajoutées sur la pile. Les registres `$r12` à `$r15` sont encore conservés pour servir d'accumulateur et de registres de travail.

Objectif Produire un nouveau compilateur `STKCompilerAlloc.mll` qui réalise cela.

Amélioration. On peut aussi s'inspirer du fonctionnement d'un cache mémoire et faire en sorte que, lorsque tous les registres ont été utilisés, se sont les « vieilles » valeurs qui sont transférées sur la pile pour laisser de la place aux plus récentes (indication : liste circulaire).

1. Le fait qu'on parte de la pile vide, mais aussi que la pile soit à nouveau vide à chaque fois que l'exécution arrive au niveau d'une étiquette, n'est pas anodin. Mais pourquoi ?