Troisième partie, cahier des charges

29 août 2001

Dans cette dernière partie, vous allez terminer la chaîne de compilation en implantant la production de code. Votre compilateur ne produira pas directement du code pour la machine concrète sur laquelle vous travaillez (généralement un processeur Intel de la famille 386) mais pour une machine virtuelle que nous introduisons dans la section suivante. Le jeu d'instructions de cette machine est semblable (mais bien moins grand) que celui du 386 mais une machine virtuelle nous permet (1) de vous fournir un interprète muni d'outils de déverminage (trace, exécution pas à pas) (2) de vous fournir une version plus ludique de cet interprète où les primitives permettent d'animer des insectes dans un labyrinthe.

1 Présentation de la machine virtuelle

1.1 Mémoire

La mémoire est composée de mots. Chaque mot est un entier 31 bits signé.

1.2 Registres

La machine virtuelle contient sept registres :
- quatre registres généraux : `EAX`, `EBX`, `ECX` et `EDX` ;
- deux registres de pile :
  - `FP` pointeur de cadre de pile (*frame pointer*) ;
  - `SP` pointeur de pile (*stack pointer*). Il doit contenir une adresse qui désigne le premier octet du dernier double mot mis sur la pile. La pile est stockée de "haut en bas" dans la mémoire. Ainsi quand on empile quelque chose, `SP` est diminué et quand on dépile il est augmenté.
- un compteur de programme `PC`.

Tous les registres contiennent des entiers 31 bits signés.

1.3 Jeu d'instructions

Les instructions de la machine virtuelle sont données figure 1. *reg* représente le nom d'un registre quelconque. La syntaxe concrète des registres est la suivante :

$$reg \ ::= \ \texttt{EAX} \mid \texttt{EBX} \mid \texttt{ECX} \mid \texttt{EDX} \mid \texttt{FP} \mid \texttt{SP} \mid \texttt{PC}$$

opr représente une opérande. Les opérandes obéissent à la syntaxe suivante :

$$opr \ ::= \ \texttt{\#}\mathit{entier} \mid \mathit{ident} \mid reg \mid \texttt{(}reg\texttt{)} \mid \mathit{entier}\texttt{(}reg\texttt{)}$$

Si $mem(a)$ désigne le mot situé en mémoire à l'adresse a, la valeur $val(opr)$ d'une opérande opr est définie par

$$\begin{array}{lcl} val(\texttt{\#}n) & = & n \\ val(ident) & = & \text{l'adresse correspondant à } ident \\ val(r) & = & \text{le contenu du registre } r \\ val(\texttt{(}r\texttt{)}) & = & mem(val(r)) \\ val(n\texttt{(}r\texttt{)}) & = & mem(val(r)+n) \end{array}$$

Dans la notation $n(r)$, n peut être négatif. Dans le tableau figure 1, on note directement *reg* pour désigner la valeur contenue dans le registre *reg*.

Instruction	Action	Commentaire
`Load` *reg*`,`*opr*	$reg \leftarrow val(opr)$	
`Store` *reg*`,`*opr*	$mem(reg) \leftarrow val(opr)$	
`Add` *reg*`,`*opr*	$reg \leftarrow reg + val(opr)$	
`Mul` *reg*`,`*opr*	$reg \leftarrow reg \times val(opr)$	
`Div` *reg*`,`*opr*	$reg \leftarrow reg \div val(opr)$	Déclenche l'erreur `Division by zero` si la valeur de l'opérande est nulle
`Neg` *reg*	$reg \leftarrow -reg$	
`And` *reg*`,`*opr*	$reg \leftarrow reg\ and\ val(opr)$	ET logique bit à bit
`Or` *reg*`,`*opr*	$reg \leftarrow reg\ or\ val(opr)$	OU logique bit à bit
`Xor` *reg*`,`*opr*	$reg \leftarrow reg\ xor\ val(opr)$	OU exclusif bit à bit
`Jz` *reg*`,`*opr*	si $reg = 0$ alors $\texttt{PC} \leftarrow val(opr)$	Sinon `PC` incrémenté normalement
`Jnz` *reg*`,`*opr*	si $reg \neq 0$ alors $\texttt{PC} \leftarrow val(opr)$	idem
`Jge` *reg*`,`*opr*	si $reg \geq 0$ alors $\texttt{PC} \leftarrow val(opr)$	idem
`Jgt` *reg*`,`*opr*	si $reg > 0$ alors $\texttt{PC} \leftarrow val(opr)$	idem
`Jmp` *opr*	$\texttt{PC} \leftarrow val(opr)$	Saut inconditionnel
`Push` *opr*	$\texttt{SP} \leftarrow \texttt{SP} - 1$; $mem(\texttt{SP}) \leftarrow val(opr)$	
`Pop` *reg*	$reg \leftarrow mem(\texttt{SP})$; $\texttt{SP} \leftarrow \texttt{SP} + 1$	
`Call` *opr*	$\texttt{SP} \leftarrow \texttt{SP} - 1$; $mem(\texttt{SP}) \leftarrow \texttt{PC}$; $\texttt{PC} \leftarrow val(opr)$	
`Ret`	$\texttt{PC} \leftarrow mem(\texttt{SP})$; $\texttt{SP} \leftarrow \texttt{SP} + 1$	

FIG. 1 – Jeu d'instructions

1.4 Directives

En plus des instructions présentées dans la section précédente, l'assembleur est muni de deux directives. L'une pour introduire une étiquette à un endroit quelconque du programme, avec la syntaxe

ident `:`

L'autre pour allouer statiquement une zone de données sur le tas, avec la syntaxe

`Data` *ident n*

où n est un entier positif ou nul représentant le nombre de mots à allouer. *ident* est l'étiquette associée à la zone de données allouée. L'adresse lui correspondant est l'adresse basse de la zone. Noter que la zone de données n'est pas allouée au mileu du code, mais sur le tas : on ne peut donc faire aucune hypothèse sur l'adresse associée à une zone de données par rapport à la position de la directive `Data` dans le code. Les directives `Data` peuvent être placées n'importe où dans le code (y compris après l'utilisation de leur étiquette).

1.5 Conventions

1.5.1 Mémoire

L'organisation de la mémoire est illustrée figure 2. Le code est situé dans les adresses les plus basses et chargé de bas en haut (`PC` est incrémenté lorsqu'une instruction autre qu'un saut est exécutée). La zone de données est partagée entre le tas et la pile. Les zones mémoire allouées avec `Data` se situent juste après le code. La pile est représentée *à l'envers* : sa base se situe à l'adresse la plus haute ($M-1$ si M est le nombre de mots mémoire) et elle croît vers les adresses basses.

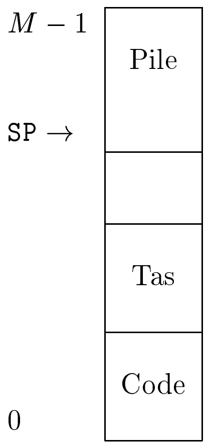

FIG. 2 – Organisation de la mémoire

1.5.2 Erreurs de la machine virtuelle

Les différentes erreurs de la machine virtuelle sont les suivantes :

Illegal Instruction : lorsqu'au moment de l'exécution `PC` pointe sur un mot ne représentant pas une instruction ;

Segmentation Fault : si l'on tente d'accéder en lecture à une adresse en dehors de la mémoire, ou en écriture à une adresse dans la zone de code ;

Bus Error : si l'on tente d'accéder en écriture en dehors de la mémoire ;

Stack Overflow : si la pile vient à rencontrer la zone de données ;

Division by zero : si la valeur de l'opérande est nulle à l'exécution de l'instruction `Div`.

1.5.3 Appels de fonctions

La machine virtuelle adopte pour les fonctions prédéfinies les conventions suivantes :

- Les arguments d'une fonction sont empilés de droite à gauche (quand on appelle `f(3,4)` on empile d'abord 4 puis 3) avant le `Call` ;
- Toute fonction restaure la pile dans l'état où elle était au moment du `Call`. C'est donc l'appelant qui se charge de dépiler les arguments qu'il utilise pour appeler une procédure ;
- Le résultat de la fonction, s'il y a en un, est stocké dans le registre `EAX` ;
- Toute fonction qui modifie les registres `FP` et `SP` doit les restaurer avant de retourner.

Vous n'êtes pas obligés d'adopter ces conventions pour appeler vos fonctions. Mais vous devez bien sûr les respecter en ce qui concerne l'appel des fonctions prédéfinies.

1.6 Exemple

Voici un exemple de programme assembleur implantant la fonction factorielle et affichant la factorielle de 10.

```
// programme assembleur implantant la factorielle et affichant 10!

        Jmp main                // saut au programme principal

// fonction fact
fact:
        Push FP                 // on sauve la valeur de FP sur la pile
        Load FP, SP             // FP pointe maintenant sur le sommet de pile
        Load EAX, 2(FP)         // on stocke l'argument dans EAX
                                // il est situé en FP+2
        Jnz  EAX, fact_non_zero // si EAX est non nul on saute a fact_non_zero
        Load EAX, #1            // sinon on charge EAX avec 1
        Jmp  fact_end           // et on termine

fact_non_zero:
        Add  EAX, #-1           // on décrémente EAX
        Push EAX                // on l'empile
        Call fact               // on appelle fact récursivement
        Mul  EAX, 2(FP)         // on multiplie le résultat (dans EAX)
                                // par l'argument (en FP+2)

fact_end:
        Load SP, FP             // on rétablit SP
        Pop  FP                 // on rétablir FP
        Ret                     // on retourne

// programme principal
main:
        Push FP                 // on sauve la valeur de FP sur la pile
        Load FP, SP             // FP pointe maintenant sur le sommet de pile
        Push #10                // on empile l'argument de fact
        Call fact               // on appelle fact
```

```
Push eax               // on empiler l'argument de print
                       // qui est le resultat de fact
Call print             // on appelle print
Load SP, FP            // on rétablit SP
Pop FP                 // on rétablit FP
Ret                    // on retourne
```

2 L'assembleur `scalpas`

Nous vous fournissons un assembleur, appelé `scalpas`. Il s'utilise de la façon suivante :

`scalpas` [`-o` *fichier*] *fichier*

Le fichier assembleur passé en argument doit porter le suffixe `.asm`. Le fichier produit peut être spécifié avec l'option `-o`. S'il n'est pas spécifié, le fichier produit s'appelle `a.out`.

Le fichier produit est directement un exécutable UNIX[1], que vous pouvons donc lancer directement. Mais vous pouvez également le passer en argument à l'interprète `scalpa`, en particulier pour l'exécuter pas à pas.

L'assembleur `scalpas` est insensible à la casse. Les commentaires débutent par `//` et se terminent à la fin de la ligne.

3 L'interprête `scalpa`

Nous vous fournissons un interprète pour la machine virtuelle, appelé `scalpa`. Il vous permet d'exécuter un fichier produit par l'assembleur `scalpas` dans divers modes. Son utilisation est la suivante :

`scalpa` [`-m` entier] [`-t`] [`-i`] [*fichier*]

L'option `-m` permet de fixer la taille de la mémoire. La valeur par défaut est de 5000 mots. Initialement, le sommet de pile (*i.e.* la case mémoire pointée par le registre de pile `SP`) contient l'adresse du code de retour du chargeur (*loader*) de la machine virtuelle. Votre programme devra donc se terminer par `Ret`, le pointeur de pile ayant retrouvé sa valeur initiale.

L'option `-t` active le mode de `trace` où chaque instruction qui va être exécutée est affichée, accompagnée de la valeur des registres. L'option `-i` active le mode interactif. Un fichier de code peut être passé en argument. S'il est omis, cela suppose que l'option `-i` a été indiquée.

En mode interactif, les commandes sont les suivantes :

`load "`*fichier*`"` : Charge le code contenu dans le fichier en mémoire et positionne le compteur de programme sur 0.

`trace` / `untrace` : Active et désactive respectivement le mode de `trace`.

`step` : Exécute un pas de calcul ; affiche les valeurs des registres à l'issue de l'exécution ainsi que l'instruction sur laquelle pointe maintenant le compteur de programme.

`run` : Exécute le programme jusqu'à ce qu'il termine, soit normalement, soit en ayant provoqué une erreur dans la machine virtuelle.

`dump` *adresse* : Décode l'instruction se trouvant à l'adresse donnée.

`dump memory` : Affiche le contenu complet de la mémoire (code, tas et pile) sous forme d'entiers.

[1] Il s'agit en fait d'un script `#!` utilisant l'interpréteur `scalpa`.

`dump registers` : Affiche les valeurs des registres.

`dump code` : Affiche le code.

`dump stack` : Affiche le contenu de la pile.

`quit` : Quitte l'interprète.

Chaque commande peut être abrégée en sa première lettre, ainsi que chaque argument de `dump`. On pourrai ainsi par exemple taper `d s` au lieu de `dump stack`. En mode interactif nous vous conseillons d'utiliser `ledit scalpa` plutôt que `scalpa`.

4 Travail demandé

Implantez la fin de la chaîne de compilation : `scalpac toto.pas` doit maintenant faire l'analyse lexicale, syntaxique et sémantique d'un fichier Scalpa et produire le fichier assembleur `toto.asm` qu'il suffira de compiler avec le programme `scalpas` fourni pour obtenir l'exécutable.

NB : On se contentera d'un schéma de traduction vers l'assembleur naïf. C'est pourquoi on ne prendra pas la peine de passer par un code intermédiaire.

Vous rendrez avec votre programme comme pour les parties précédentes un très court rapport indiquant quels sont les choix que vous avez faits, ainsi qu'un jeu de tests (version électronique exigée — vous pouvez en ajouter une version papier si vous le désirez, mais il faut que l'on puisse exécuter vos tests de façon immédiate).

Annexe : rappel des fonctions prédéfinies

`print(`*int v*`)` : *unit*	`random(`*int v*`)` : *int*
`move()` : *bool*	`x_pos()` : *int*
`y_pos()` : *int*	`power()` : *int*
`look()` : *int*	`where()` : *int*
`turn_left()` : *unit*	`turn_right()` : *unit*
`x_door()` : *int*	`y_door()` : *int*