Réalisation d'un compilateur pour le langage Z en Python

Exercices à faire ensemble lors de la 1re séance

1. Parcourir le chapitre 2 *Principes de conception d'un analyseur syntaxique* des notes de cours.

2. Analysez la grammaire du langage Z donnée dans la section 3.2.1 des notes de cours (page 33) et utilisez-la pour représenter l'arbre syntaxique pour chacune des expressions suivantes :

   - 18 + 45
   - 18 + 45 * 92
   - 18 * 45 + 92 * 77
   - 18 + 2 * abc

3. Testez pour les 4 expressions données ci-dessus le compilateur en Python figurant dans la section 3.2.2 (page 33 et 34) construit à partir de la grammaire de la section 3.2.1.

4. Analysez la grammaire du langage Z donnée dans la section 3.3.1 des notes de cours (page 36). Comparez cette grammaire avec celle figurant dans la section 3.2.1. Analysez et testez le compilateur figurant dans la section 3.3.2 (pages 36 à 38) construit à partir de cette grammaire.

Étapes de réalisation du compilateur

1. Partez de la grammaire du langage Z donnée dans la section 3.3.1 des notes de cours (page 36). Modifiez dans cette grammaire les symboles de début et de fin de code source pour qu'ils deviennent **start>** et **<stop**. Partez du compilateur donné dans la section 3.3.2 (pages 36 à 38) et mettez-le à jour.

2. Ajoutez dans la grammaire du langage Z l'opérateur modulo (**%**). Cet opérateur a la même priorité qu'en langage C. Ensuite, mettez à jour le compilateur.

   Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`    512 * 17 % 3 + 7`
`<stop`
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 512`
	`  push dword ptr 17`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  push dword ptr 3`
	`  pop ebx`
	`  pop eax`
	`  cdq`
	`  idiv ebx`
	`  push edx`
	`  push dword ptr 7`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop eax`
	`  }`
	`}`

   En testant le code en assembleur dans Visual Studio, on doit avoir 8 comme valeur finale dans EAX.

3. Ajoutez dans la grammaire du langage Z les nombres en binaire (préfixe **0b**) et les nombres en hexadécimal (préfixe **0x**). Ensuite, mettez à jour le compilateur.

   Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`    0b101 + 0xf0 * 3`
`<stop`
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 5`
	`  push dword ptr 240`
	`  push dword ptr 3`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop eax`
	`  }`
	`}`

   En testant le code en assembleur dans Visual Studio, on doit avoir 725 comme valeur finale dans EAX.

4. Ajoutez dans la grammaire du langage Z les opérateurs logiques & et |. Ces opérateurs ont la même priorité qu'en langage C. Ensuite, mettez à jour le compilateur.

   Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`  512 * (144 + 0b11) | 0x12 - 541 & 0xff`
`<stop`
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 512`
	`  push dword ptr 144`
	`  push dword ptr 3`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  push dword ptr 18`
	`  push dword ptr 541`
	`  pop ebx`
	`  pop eax`
	`  sub eax, ebx`
	`  push eax`
	`  push dword ptr 255`
	`  pop ebx`
	`  pop eax`
	`  and eax, ebx`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  or eax, ebx`
	`  push eax`
	`  pop eax`
	`  }`
	`}`

En testant dans Visual Studio, on doit avoir 75509 comme valeur finale dans EAX.

5. Ajoutez dans la grammaire du langage Z les opérateurs unaires ~ et -. Ces opérateurs ont la même priorité qu'en langage C. Ensuite, mettez à jour le compilateur.

   Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`    -(-(144 + 0b11) + ~0xff)`
`<stop`
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 144`
	`  push dword ptr 3`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop eax`
	`  neg eax`
	`  push eax`
	`  push dword ptr 255`
	`  pop eax`
	`  not eax`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop eax`
	`  neg eax`
	`  push eax`
	`  pop eax`
	`  }`
	`}`

   En testant dans Visual Studio, on doit avoir 403 comme valeur finale dans EAX.

6. Ajoutez dans la grammaire du langage Z l'opérateur d'affectation (=) pour pouvoir stocker le résultat de l'expression dans une variable entière signée de 4 octets. La forme de l'instruction d'affectation est `variable = expression`. Ensuite, mettez à jour le compilateur.

   Le nom d'une variable est **`i`**`___`, où **`i`** est un préfixe indiquant qu'il s'agit du type int et `___` est le reste du nom de la variable constitué uniquement de lettres entre a et z ou entre A et Z.

   Dans le code généré par votre compilateur, on doit y trouver au début la déclaration de la variable.

   Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`    iVar = -(-(144 + 0b11) + ~0xff)`
`<stop`
	`int iVar;`
	
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 144`
	`  push dword ptr 3`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop eax`
	`  neg eax`
	`  push eax`
	`  push dword ptr 255`
	`  pop eax`
	`  not eax`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop eax`
	`  neg eax`
	`  push eax`
	`  pop iVar`
	`  }`
	`}`

   En testant dans Visual Studio, on doit avoir 403 dans iVar.

7. Ajoutez dans la grammaire du langage Z la possibilité de stocker le résultat d'une expression également dans une variable du type short appelée **s___** ou du type char appelée **b___**. Le préfixe **s** ou **b** indique qu'il s'agit du type short ou du type char et ___ est le reste du nom de la variable constitué uniquement de lettres entre a et z ou entre A et Z. Ensuite, mettez à jour le compilateur.

   Comme une variable dont le nom commence par b ou par s n'a pas une taille de 4 octets, il faut penser à réduire la taille de la valeur avant de la copier dans cette variable.

   Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`  sVal = 0b110010 | 9008 / (2 * 0xf)`
`<stop`
	`short sVal;`
	
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 50`
	`  push dword ptr 9008`
	`  push dword ptr 2`
	`  push dword ptr 15`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  cdq`
	`  idiv ebx`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  or eax, ebx`
	`  push eax`
	`  pop eax`
	`  mov sVal, ax`
	`  }`
	`}`

   En testant dans Visual Studio, on doit avoir 318 dans sVal.

8. Ajoutez dans la grammaire la possibilité d'avoir plusieurs instructions d'affectation. Le symbole terminal **;** sert de séparateur entre 2 instructions d'affectation. Ensuite, mettez à jour le compilateur.

   Une même variable peut être rencontrée à plusieurs reprises dans le programme en langage Z. Cependant, le compilateur ne doit générer qu'une seule fois la déclaration de cette variable dans le programme en assembleur.

   Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`  sVar = (2 + 33) * 5;`
`  bVal = 8 / 2 * 0xf0;`
`  sVar = 0b110 + bVal`
`<stop`
	`char bVal;`
	`short sVar;`
	
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 2`
	`  push dword ptr 33`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  push dword ptr 5`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  pop eax`
	`  mov sVar, ax`
	`  push dword ptr 8`
	`  push dword ptr 2`
	`  pop ebx`
	`  pop eax`
	`  cdq`
	`  idiv ebx`
	`  push eax`
	`  push dword ptr 240`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  pop eax`
	`  mov bVal, al`
	`  push dword ptr 6`
	`  movsx eax, bVal`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop eax`
	`  mov sVar, ax`
	`  }`
	`}`

   En testant dans Visual Studio, on doit avoir -58 dans sVar et -64 dans bVal.

9. Ajoutez dans la grammaire du langage Z la possibilité d'utiliser des variables dans les expressions à droite de l'opérateur d'affectation. Ensuite, mettez à jour le compilateur.

   Il faut tester si toute variable utilisée à droite de l'opérateur d'affectation existe déjà, car une variable qu'on utilise dans une opération doit avoir préalablement reçu une valeur, sinon on stoppe la compilation avec un message d'erreur.

   Comme une variable dont le nom commence par s ou par b n'a pas une taille de 4 octets, le compilateur doit générer l'instruction MOVSX pour étendre sur 4 octets la valeur de cette variable, car, comme avec le langage C, les traitements sont réalisés sur des entiers de 32 bits.

   Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`  bVar = 3;`
`  sVar = 2 + bVar;`
`  iNb = 0b110 * sVar;`
`  iNb = iNb + 1`
`<stop`
	`int iNb;`
	`short sVar;`
	`char bVar;`
	
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 3`
	`  pop eax`
	`  mov bVar, al`
	`  push dword ptr 2`
	`  movsx eax, bVar`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop eax`
	`  mov sVar, ax`
	`  push dword ptr 6`
	`  movsx eax, sVar`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  pop iNb`
	`  push iNb`
	`  push dword ptr 1`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  pop iNb`
	`  }`
	`}`

   En testant dans Visual Studio, on doit avoir 3 dans bVar, 5 dans sVar et 31 dans iVar.

10. En plus de l'instruction d'affectation, ajoutez dans la grammaire du langage Z l'instruction **print** permettant d'afficher le résultat d'une expression. La forme de l'instruction print est `print expression`. Le code généré en assembleur pour afficher invoque la fonction printf(). Ensuite, mettez à jour le compilateur.

    Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`  sNombre = 45;`
`  print sNombre + 2;`
`  print -sNombre * 0b10`
`<stop`
	`#include <stdio.h>`
	`const char msgAffichage[] = "Valeur = %d\n";`
	`short sNombre;`
	
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push dword ptr 45`
	`  pop eax`
	`  mov sNombre, ax`
	`  movsx eax, sNombre`
	`  push eax`
	`  push dword ptr 2`
	`  pop ebx`
	`  pop eax`
	`  add eax, ebx`
	`  push eax`
	`  push offset msgAffichage`
	`  call dword ptr printf`
	`  add esp, 8`
	`  movsx eax, sNombre`
	`  push eax`
	`  pop eax`
	`  neg eax`
	`  push eax`
	`  push dword ptr 2`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  push offset msgAffichage`
	`  call dword ptr printf`
	`  add esp, 8`
	`  }`
	`}`

    Tout comme avec l'instruction d'affectation, l'instruction print peut bien entendu être utilisée à plusieurs reprises dans le programme en langage Z, mais le compilateur ne doit générer les lignes suivantes au début du programme en assembleur qu'une seule fois :

```
#include <stdio.h>
const char msgAffichage[] = "Valeur = %d\n";
```

    Voici ce que donne l'exécution du programme en assembleur dans Visual Studio :

```
Valeur = 47
Valeur = -90
```

11. Ajoutez dans la grammaire du langage Z l'instruction **input** permettant de saisir au clavier une valeur entière et de la stocker dans une variable. La forme de l'instruction input est `input variable`. Le code généré en assembleur pour la saisie au clavier utilise la fonction scanf(). Ensuite, mettez à jour le compilateur.

Exemple pour tester votre compilateur :

Programme source en Z	*Programme en assembleur après compilation*

`start>`
`  input sNombre;`
`  input bNb;`
`  print sNombre * -bNb`
`<stop`
	`#include <stdio.h>`
	`const char msgAffichage[] = "Valeur = %d\n";`
	`const char msgSaisie[] = "%d";`
	`const char msgEntrez[] = "Entrez une valeur : ";`
	`int varSaisie;`
	`char bNb;`
	`short sNombre;`
	
	`void main()`
	`{`
	`  _asm`
	`  {`
	`  push offset msgEntrez`
	`  call dword ptr printf`
	`  add esp, 4`
	`  push offset varSaisie`
	`  push offset msgSaisie`
	`  call dword ptr scanf`
	`  add esp, 8`
	`  push varSaisie`
	`  pop eax`
	`  mov sNombre, ax`
	`  push offset msgEntrez`
	`  call dword ptr printf`
	`  add esp, 4`
	`  push offset varSaisie`
	`  push offset msgSaisie`
	`  call dword ptr scanf`
	`  add esp, 8`
	`  push varSaisie`
	`  pop eax`
	`  mov bNb, al`
	`  movsx eax, sNombre`
	`  push eax`
	`  movsx eax, bNb`
	`  push eax`
	`  pop eax`
	`  neg eax`
	`  push eax`
	`  pop ebx`
	`  pop eax`
	`  imul eax, ebx`
	`  push eax`
	`  push offset msgAffichage`
	`  call dword ptr printf`
	`  add esp, 8`
	`  }`
	`}`

Voici ce que donne l'exécution du programme en assembleur dans Visual Studio :

```
Entrez une valeur : 20
Entrez une valeur : 10
Valeur = -200
```

L'instruction input peut être utilisée à plusieurs reprises dans le programme en langage Z, mais le compilateur ne doit générer les lignes suivantes au début du programme en assembleur qu'une seule fois :

```
#include <stdio.h>
const char msgSaisie[] = "%d";
const char msgEntrez[] = "Entrez une valeur : ";
int varSaisie;
```

La variable *varSaisie* est du type int et son usage est généré par le compilateur automatiquement quand l'instruction input est utilisée, car c'est dans cette variable de 4 octets qu'est stocké le nombre entier saisi au clavier avec scanf(). Après la saisie, le contenu de cette variable est alors copié dans la variable utilisée dans l'instruction input qui peut être une variable d'une taille de 1, 2 ou 4 octets. Par exemple, pour l'instruction *input sNombre*, la valeur saisie est enregistrée par scanf() dans la variable *varSaisie*, puis, les 16 bits inférieurs de cette variable sont copiés dans la variable *sNombre* qui est du type short.

À présent, le compilateur est capable de reconnaître maintenant 3 types d'instructions différentes : l'instruction d'affectation (`variable = expression`), l'instruction print (`print expression`) et l'instruction input (`input variable`). D'autres instructions pourraient, bien entendu, être ajoutées comme : `if expression then instructions`, `for expression do instructions`, etc.