TP1 - Retour sur le C++

Introduction au projet AbstractVM

Giron David thor@epitech.net
Montinet Maxime montin_m@epitech.net

Abstract: AbstractVM est une machine a pile. Nous aborderons dans ce TP les pistes a explorer pour realiser ce projet.

Table des matières

.1 Une machine a pile

AbstractVM etant une machine a pile, la premiere chose intelligente a faire est de (re)decouvrir ce qu'est une machine a pile...

La commande `dc` est une calculatrice a precision arbitraire en notation polonaise inverse, basee sur un systeme de pile. Curieusement, ca ressemble beaucoup a AbstractVM...

FIGURE 1 – `Ceci n'est PAS une machine a pile`

- Lisez le man de la commande `dc` jusqu'a la section "Stack control" incluse
- Lancez `dc` et faites quelques calculs simples avec pour vous familiariser avec la pile

http://fr.wikipedia.org/wiki/Notation_polonaise_inverse

Pour aller plus loin sur les notations des operateurs, on en distinguera trois :

- Prefixe : Par exemple "+ 3 4". On parle egalement de notation fonctionnelle.
- Infixe : Par exemple "3 + 4". C'est la notation usuelle des operateurs binaires.
- Suffixe : Par exemple "3 4 +". Egalement appellee notation polonaise inverse. Les langages assembleurs utilisent ce fonctionnement.

.2 Machine a pile basique a la main

.2.1 Des classes pour calculer

- Declarez et implementez la classe `Operand`. Elle devra respecter les criteres suivants :
  - Representation d'une valeur entiere passee a la construction de l'instance. La gestion d'une construction sans parametre est laissee a votre discretion.
  - Fonctions membres `add`, `sub`, `mul`, `div` et `mod`, qui prennent toutes les cinq une reference constante sur `Operand` et renvoient une nouvelle instance de la classe `Operand` representant le resultat de l'operation. L'instance courante represente l'operande de gauche de l'operation.

- Declarez et implementez la classe `Calculator`. Elle devra manipuler des `Operand` et posseder les fonctions membres suivantes :
  - `push(int)` : Ajoute une nouvelle operande sur la pile, dont la valeur est passee en parametre
  - `pop()` : Depile l'operande au sommet de la pile et renvoit sa valeur (un entier, donc)
  - `dump()` : Affiche la pile sur la sortie standard
  - `add()` : Depile les deux valeurs au sommet de la pile, puis empile le resultat de l'addition de ces deux valeurs
  - `sub()` : Depile les deux valeurs au sommet de la pile, puis empile le resultat de la difference de ces deux valeurs
  - `mul()` : Depile les deux valeurs au sommet de la pile, puis empile le resultat du produit de ces deux valeurs
  - `div()` : Depile les deux valeurs au sommet de la pile, puis empile le resultat du quotient de ces deux valeurs
  - `mod()` : Depile les deux valeurs au sommet de la pile, puis empile le modulo de ces deux valeurs

`La STL propose une excellente classe de pile...`

Voici un petit `main` d'exemple pour vous donner une idee de ce qu'il devrait etre possible de faire avec vos deux classes. Ce `main` est purement informatif.

```
int main(void)
{
  Calculator calc;

  calc.push(21);
  calc.push(42);
  calc.dump();
  calc.sub();
  calc.dump();
  calc.push(21);
  calc.add();
  calc.dump();
}
```

Ce code pourra avoir une sortie semblable a la suivante :

```
Stack dump :
42
21
Stack dump :
21
Stack dump :
42
```

.2.2 Une machine scriptable

Le lexer

Un lexer est un programme qui decoupe la chaine de characteres recue en entree en une liste de lexemes (ou "tokens" en anglais). La theorie accompagnant les lexers nous ramene a la theorie des langages reguliers que vous avez aborde a travers l'utilisation des expressions rationnelles si rependues en informatique. Si vous voulez en savoir plus, je vous recommande la lecture des travaux de Noam Chomsky et de Marcel-Paul Schützenberger.

Un lexer est un automate complexe qui n'est plus code a la main depuis la nuit des temps et l'ecriture du generateur de lexer `lex`. On trouve d'ailleurs la mention "The asteroid to kill this dinosaur is still in orbit" a la fin du man de `lex` sur le systeme d'exploitation `Plan9`.

En restant simples et pratiques il est en realite tout a fait abordable d'ecrire un petit lexer raisonnable pour un langage simple, comme celui d'AbstractVM par exemple. Pour cela il suffit de se concentrer sur l'aspect purement fonctionnel du lexer : Decouper un flux d'entree en une liste de lexemes et faire avec les moyens du bord... Pas besoin d'un automate !

Consultez ce document : http://perso.epitech.eu/~koala/ressources/conference_tech1_2010.ppt.

Le flux d'entree peut aller du rustique tableau de `char` aux plus modernes flux, et la liste de sortie de la modeste liste chainee forgee a la main a la liste template de la STL. Dans notre exemple, nous considererons un flux d'entree de type `std::istream` et une liste resultat de type `std::list`.

Nous allons definir un lexeme par un couple d'attributs : Sa classe et la chaine de characteres associee. Attention, ici le mot "classe" fait reference a un groupe, pas a la classe de programmation objet que l'ont peut instancier. Nous representerons les classe de lexemes possibles du langage assembleur d'AbstractVM grace a un enum ayant les valeurs suivantes (les identifiants sont indicatifs) :

- Intruction
- Separateur
- Parenthese_ouvrante
- Parenthese_fermante
- Entier
- Fin_de_flux

Les chaines de characteres associees seront de type `std::string`.

Nous pouvons a present debuter l'implantation d'une classe `Lexer` permettant les actions suivantes :

- Instancier la classe en prenant en parametre le flux a lexer (ou un comportement similaire)
- Lexer le flux et renvoyer une `std::list` des lexemes. Durant cette etape, les characteres blancs ont ete convenablement elimines
- En cas d'entree invalide, le programme ne doit pas planter et reagir de facon raisonnable.

Le parser

Maintenant que vous avez un beau lexer tout beau tout propre, vous allez pouvoir vous pencher sur le parser. Un parser est un programme qui s'assure que la liste de lexemes resultat du lexer represente une entree valide. Pour s'assurer de cette validite, le parser compare sont entree a la grammaire du langage concernee. Il existe plusieurs methode d'analyse syntaxique. Mais toutes reposent sur le concept de grammaire. Je vous encourage a aller consulter les traveaux de Noam Chomsky et de Marcel-Paul Schützenberger concernant les langages algebriques. Les langages informatiques usuels correspondent a cette classe de langages et sont obtenu a partirs de grammaires (algebriques) dont la forme doit commencer a vous etre familiere.

Pour ce TP, nous utiliserons la grammaire simplifiee suivante pour traiter le sous-ensemble du langage assembleur d'AbstractVM qui nous interesse (le # marque la fin de l'entree, pas un lexeme "#") :

```
1   S := [INSTR SEP]* #
2
3   INSTR :=
4       push(N)
5     | pop
6     | dump
7     | add
8     | sub
9     | mul
10    | div
11
12  N := [-]?[0..9]+
13
14  SEP := '\n'
```

De facon similaire aux lexers, les parsers sont generalement generes par des outils vieux comme le monde tels que `yacc`. Ces programmes contruisent des automates complexes et diffciles a ecrire a la main. On distingue deux familles d'analyseurs syntaxiques : Ascendants et descendants. Une sous famille des analyseurs descendants dits "LL(0)" ne necessite pas d'automate pour fonctionner et peut etre represente par une serie de fonctions mutuellement recursives repsentant la grammaire du langage. De tels parseurs sont particulierement simples a ecrire et suffisent plus que largement a traiter le cas de l'assembleur AbstractVM. Vous trouverez des informations interessante sur l'implantation d'un tel parser dans le document disponible a l'adresse http://perso.epitech.eu/~koala/ressources/conference_tech1_2010.ppt.

Le resultat d'une analyse syntaxique reussis est un arbre particulier appele "AST" pour "Abstract Syntax Tree" ("arbre de syntaxe abstraite"). Cet arbre est le resultat de la reorganisation des lexemes sous une forme plus simplement exploitable. Il n'existe pas de forme canonique d'AST car celle-ci est directement liee au langage represente. Toutefois la simplicite extreme du langage assembleur d'AbstractVM laisse presager que l'AST resultat aura une forme proche d'un arbre dit "peigne".

Nous pouvons a present debuter l'implantation d'une classe `Parser` permettant les actions suivantes :

- Instancier la classe en prenant en parametre la liste resultat du lexer, ou un comportement similaire.
- Parser la liste de lexemes pour s'assurer de la validite systaxique du programme d'entree.
- En cas d'entree invalide, le programme ne doit pas planter et reagir de facon raisonnable.

Chapitre I

Conclusion

Nous espérons que vous avez apprécié ce sujet autant que nous avons apprécié le rédiger pour vous.

Vos avis sont très importants pour nous et nous permettent chaque jour d'améliorer nos contenus. C'est pourquoi nous comptons beaucoup sur vous pour nous apporter vos retours.

Si vous trouvez que certains points du sujet sont obscurs, pas assez bien expliqués ou tout simplement contiennent des fautes d'orthographe, signalez-le nous. Pour cela, il vous suffit de nous envoyer un mail à l'adresse koala@epitech.eu.

Si vous souhaitez aller plus loin après le premier TP C++, nous vous recommandons le livre "Effective C++" : http://bibliotech.epitech.eu/?page=book&id=157

Si vous avez des doutes sur la syntaxe, nous vous recommandons "The C++ Programming Language" : http://bibliotech.epitech.eu/?page=book&id=49.

Beaucoup d'autres livres sur le C++ sont empruntables au koalab : http://bibliotech.epitech.eu/?page=user&id=52. N'hésitez pas à passer vous renseigner auprès des koalas. Ils seront ravis de vous conseiller. Certains livres sont disponibles en version PDF si vous êtes connectés sur le site.