IFT2035 – Travail pratique #1 – 2019.10.05

GESTION MÉMOIRE, POINTEURS ET COMPILATION

Marc Feeley

Le TP1 a pour but de vous faire pratiquer la programmation impérative et en particulier les concepts suivants : la gestion mémoire manuelle, les pointeurs et la compilation des langages impératifs. Vous avez à réaliser un programme en C ainsi que rédiger un rapport contenant une analyse du programme.

---

1 Programmation

Vous devez réaliser un petit compilateur qui est basé sur le programme `petit-comp.c` qui a été présenté en classe et en démonstration et dont le code est disponible sur la page Studium du cours. Le codage est donc à faire en langage C avec une version récente (≥ 8) du compilateur `gcc` de GNU.[1] Vous devez exploiter les forces du langage C pour exprimer au mieux votre programme, en particulier l'arithmétique sur les pointeurs et `setjmp`/`longjmp`.

Vous devez modifier le programme `petit-comp.c` pour atteindre les objectifs suivants :

- Le programme doit détecter lorsqu'une allocation mémoire dynamique a échouée (e.g. `malloc` a retourné `NULL`) et faire un traitement d'exception approprié, incluant l'affichage d'un message indiquant la situation et une terminaison du programme.
- Il faut éliminer toutes les fuites de mémoire. Donc tout espace mémoire alloué avec `malloc` soit être récupéré avec `free` avant la fin de l'exécution du programme. Ceci doit se passer même si le programme termine à cause d'une erreur syntaxique dans le programme compilé ou un manque de mémoire. Dans la version d'origine de `petit-comp.c` les noeuds de l'ASA ne sont jamais récupérés et vous devez donc remédier à ce problème. Si vous utilisez `malloc` pour d'autres allocations, ces espaces mémoire doivent aussi être récupérés.
- Vous devez étendre le compilateur pour qu'il puisse compiler des programmes du langage précisé dans la prochaine section. Cela vous demandera d'étendre l'analyseur syntaxique, le générateur de code, le jeu d'instructions de la machine virtuelle, et l'interprète de bytecode.

2 Langage

Le langage source accepté par la version d'origine de `petit-comp.c` est un langage impératif avec une syntaxe inspirée du langage C et des types de données et structures de contrôle limitées. En effet il y a seulement le type entier et 26 variables globales implicitement déclarées (chaque lettre minuscule de l'alphabet). Il y a seulement les opérateurs `+`, `-`, `<` et l'affectation. Les énoncés disponibles sont le `if`, le `while`, le `do`/`while` et le bloc `{...}`.

Lorsque le compilateur est exécuté il lit de son entrée standard le programme source à compiler. À l'aide d'une redirection on peut faire en sorte que le compilateur lise le programme source d'un fichier. Par exemple :

```
% ./petit-comp < prog.c
```

[1] Veuillez noter que sur macOS le programme `gcc` par défaut n'est pas celui de GNU. Vous pouvez installer celui de GNU avec homebrew à l'aide de la commande `brew install gcc@8`.

Le compilateur lira le contenu du fichier `prog.c`, l'analysera pour s'assurer qu'il respecte la grammaire du langage, construira un ASA qui représente le programme, générera le bytecode à partir de l'ASA et fera l'exécution du bytecode avec l'interprète de la machine virtuelle.

Le compilateur doit être étendu pour accepter le langage spécifié par la grammaire suivante. Les nouveautés sont indiquées par un commentaire à droite (qui ne fait pas partie de la grammaire!).

```
<program> ::= <stat>

<stat> ::= "if" <paren_expr> <stat>
         | "if" <paren_expr> <stat> "else" <stat>
         | "while" <paren_expr> <stat>
         | "do" <stat> "while" <paren_expr> ";"
         | ";"
         | "{" { <stat> } "}"
         | <expr> ";"
         | "print" <paren_expr> ";"            ***nouveau***
         | "break" ";"                         ***nouveau***
         | "continue" ";"                      ***nouveau***

<expr> ::= <test>
         | <id> "=" <expr>

<test> ::= <sum>
         | <sum> "<"  <sum>
         | <sum> "<=" <sum>                    ***nouveau***
         | <sum> ">"  <sum>                    ***nouveau***
         | <sum> ">=" <sum>                    ***nouveau***
         | <sum> "==" <sum>                    ***nouveau***
         | <sum> "!=" <sum>                    ***nouveau***

<sum> ::= <mult>                               ***nouveau***
        | <sum> "+" <mult>                     ***nouveau***
        | <sum> "-" <mult>                     ***nouveau***

<mult> ::= <term>                              ***nouveau***
         | <mult> "*" <term>                   ***nouveau***
         | <mult> "/" "10"                     ***nouveau***
         | <mult> "%" "10"                     ***nouveau***

<term> ::= <id>
         | <int>
         | <paren_expr>

<paren_expr> ::= "(" <expr> ")"
```

Le langage a des nouvelles formes d'expressions. La catégorie `<mult>` a été ajoutée pour les opérateur multiplicatifs `*`, `/` et `%`. Notez que la seule valeur permise à la droite de `/` et `%` est un lexème `<int>` égal à 10. D'autre part la catégorie `<test>` a été étendue pour compléter les opérateur de comparaison avec `<=`, `>`, `>=`, `==` et `!=`. Ces opérateurs font le même calcul qu'en C. Entre autres, les opérateurs de comparaison retournent la valeur 0 ou 1, représentant respectivement "faux" et "vrai". Notez que vous devrez ajouter des nouvelles instructions et bytecodes à la machine virtuelle pour pouvoir compiler ces nouvelles opérations (inspirez-vous de `IADD` et `IFLT`).

Le type entier doit être étendu pour faire des calculs sur des valeurs entières de taille quelconque, c'est-à-dire avec un nombre de chiffres illimité (voir les détails d'implantation dans la prochaine section).

Le langage a aussi des nouvelles formes d'énoncés (catégorie `<stat>`). Il y a un énoncé `print(...)` pour imprimer une valeur entière et passer à la prochaine ligne. Vous devez ajouter une instruction à la machine virtuelle pour implanter le `print(...)`.

Il y a aussi les énoncés `break` et `continue` qui ont un comportement identique au langage C. Voici un exemple de programme qui utilise ces énoncés :

```
{
  y = 1;
  while (1) {
    y = y*2;
    if (y > 99999) break;
    if (y < 20) continue;
    x = y;
    while (x > 0) {
      print(x % 10);
      x = x / 10;
    }
  }
}
```

Le compilateur doit signaler une erreur lorsqu'un `break` ou `continue` est utilisé ailleurs que dans le corps d'une boucle.

Pour implanter ces énoncés de contrôle il y a déjà tout ce qu'il faut dans le jeu d'instructions de la machine virtuelle. Donc n'ajoutez rien de nouveau pour ça. Cependant, les instructions de branchement de la machine virtuelle sont limitées à -128..+127 comme distance de branchement. Le compilateur doit donner un message d'erreur à la compilation si la distance de la cible de branchement est trop éloignée (dans la version d'origine cette situation n'est pas vérifiée).

Dans la version d'origine de `petit-comp.c` à la fin de l'exécution le contenu des variables ayant une valeur différente de 0 est affiché. Ce comportement doit être retiré car le langage a maintenant l'énoncé `print(...)` pour faire des affichages.

Il est primordial d'éviter les fuites de mémoire et les pointeurs fous. Lorsque votre programme sera testé, nous ferons varier artificiellement l'espace mémoire disponible à votre programme. Il est donc tout à fait possible que n'importe quel appel à la fonction `malloc` retournera `NULL`.

Votre programme C doit seulement inclure les fichiers d'entête "`stdio.h`", "`stdlib.h`" et "`string.h`". Il doit se compiler et exécuter sans erreur **sous linux** avec les commandes :

```
% gcc -o ./petit-comp petit-comp.c
% ./petit-comp < prog.c
```

3 Grands entiers

Les valeurs manipulées par le programme compilé par votre compilateur sont de type entier signé sans limite sur le nombre de chiffres (à part la capacité mémoire de l'ordinateur). On n'est donc pas limité à des calculs sur des entiers de 32 ou 64 bits comme ça serait le cas avec la plupart des compilateurs C. On appèlera ce type les *grands entiers*. Voici un exemple de programme qui fait des calculs avec des grands entiers, précisément le programme calcule 2 à la puissance 200 :

```
{
  i = 1;
  j = 0;
  n = 200;
  while (j < n) {
    i = i*2;
    j = j+1;
  }
  print(i);
  print(i % 10);
  print(i / 10 % 10);
  print(i / 10 / 10 % 10);
  print(i / 10 / 10 / 10 % 10);
  print(i / 10 / 10 / 10 / 10 % 10);
}
```

Et le résultat affiché par le programme :

```
1606938044258990275541962092341162602522202993782792835301376
6
7
3
1
0
```

Il y a plusieurs approches pour implanter les grands entiers. Pour vous faciliter la tâche et vous faire travailler avec les pointeurs, la représentation suivante est imposée. Elle consiste à représenter une valeur entière à l'aide d'une liste de ces chiffres décimaux, avec en plus un indicateur du signe de la valeur. Voici les définitions de type pour cette représentation :

```
struct grand_entier { int negatif; struct cellule *chiffres; };
struct cellule { char chiffre; struct cellule *suivant; };
```

La structure `grand_entier` est une représentation dite "signe et magnitude", donc le champ `negatif` est vrai (valeur différente de 0) si et seulement si la valeur entière est négative. La valeur entière zéro a un champ `chiffres` égal à `NULL` et un champ `negatif` égal à 0. Pour toute valeur entière différente de zéro, le champ `chiffres` est non-`NULL` et pointe à la première cellule d'une liste chaînée des chiffres décimaux, en commençant par le chiffre de poids faible (une représentation "little endian"). Dans le champ `chiffre` on retrouve un caractère de "0" à "9". Le dernier chiffre de la chaîne n'est jamais égal à "0".

Avec cette représentation il est relativement simple de coder les algorithmes de "petite école" (chiffre par chiffre) pour l'addition, la soustraction et la multiplication. Notez que le langage ne permet que la division (et modulo) par 10, qui s'implante aisément avec cette représentation.

De façon générale, incluant des affectations comme `i = j;`, vous ne devez pas dupliquer les valeurs entières (c'est-à-dire avoir que `i` et `j` contiennent des pointeurs vers des `grand_entier` différents avec les mêmes chiffres). Cela pourrait gaspiller de la mémoire lorsque les nombres sont grands. Vous devez modifier le type `grand_entier` et `cellule` pour ajouter un compteur de référence que votre interprète devra maintenir à jour explicitement et récupérer l'espace mémoire le plus tôt possible.

4 Rapport

Vous devez rédiger un rapport qui :

1. Explique brièvement le fonctionnement général du programme (maximum de 1 page au total).
2. Explique comment les problèmes de programmation suivants ont été résolus (en 2 à 4 pages au total) :
   (a) comment se fait le traitement des manques de mémoire?
   (b) comment se fait le traitement des erreur de syntaxe?
   (c) comment se fait la récupération de l'espace mémoire?
   (d) comment se fait l'analyse syntaxique des nouvelles formes syntaxiques?
   (e) comment se fait la gestion mémoire des grands entiers?
   (f) quelles sont les nouvelles instructions que vous avez ajoutées à la machine virtuelle?
   (g) comment sont implantés les nouveaux types d'énoncés (`print`, `break` et `continue`)?

5 Évaluation

- Ce travail compte pour 15 points dans la note finale du cours. Indiquez vos noms clairement au tout début du programme. **Vous devez faire le travail par groupes de 2 personnes. Vous devez confirmer la composition de votre équipe (noms des coéquipiers) au démonstrateur au plus tard le 16 octobre. Si vous ne trouvez pas de partenaire d'ici quelques jours, venez me voir.**
- Le programme sera évalué sur 8 points et le rapport sur 7 points. Un programme qui plante à l'exécution, même dans une situation extrême, se verra attribuer zéro sur 8 (c'est un incitatif à bien tester votre programme). Assurez-vous de prévoir toutes les situations d'erreur (en particulier un appel à la fonction `malloc` de C qui retourne `NULL`).
- Vous devez remettre votre rapport (un fichier ".pdf") et le programme (votre fichier `petit-comp.c`) au plus tard le dimanche 3 novembre à 23:55 *** sur le site Studium du cours.*** Notez que votre programme doit être contenu dans un seul fichier.
- L'élégance et la lisibilité du code, l'exactitude et la performance, la lisibilité du rapport, et l'utilisation d'un français sans fautes sont des critères d'évaluation. Vous pouvez cependant écrire votre code avec des identifiants et des commentaires en anglais.