Polytech Nice Sophia
SI4 / 2018-2019
Erick Gallesio

Compilation

Feuille 7

Toy base

Encore des extensions

Les exercices de cette feuilles sont à rendre.

Par conséquent: **cette feuille n'aura pas de corrigé**

Pour travailler, vous devez prendre **l'archive** du compilateur. Cette version du compilateur correspond au langage *Toy-base* du TD précédent où toutes les extensions demandées ont été implémentées (en gros, c'est donc le corrigé du TD n°6 :-)

Introduction

Cette feuille comporte des extensions supplémentaires à la version du compilateur *Toy-base* de la dernière feuille de TD.

1 Concaténation de chaînes

On veut permettre la concaténation de chaînes de caractères avec l'opérateur ' `+` ' (comme en Python ou en Java). Ainsi, le programme *Toy* suivant:

```
int main() {
  string s = "Hello" + " " + "world.";

  print(s, "\n");
  return 0;
}
```

permet d'afficher le message " `Hello, world.` " sur la sortie standard.

Pour implémenter cette fonctionnalité, vous aurez besoin d'écrire une fonction C permettant la concaténation de deux chaînes de caractères. Cette fonction sera écrite dans le runtime (on ne se préoccupera pas ici des **fuites de mémoire** qui pourraient résulter de l'introduction de cet opérateur dans un programme *Toy*).

Pour tester votre extension, vous pouvez utiliser les tests suivants:

- ok-concat.toy,
- fail-concat.toy

2 Un switch pour Toy

On veut ici implémenter une forme de `switch` pour le langage Toy. La forme choisie ressemble à la construction `switch` du langage Go. Un exemple d'utilisation est donné ci dessous:

```
switch {
    case i < 0:
        s = "negative";
    case i == 0:
        { s = "zero"; x = 1000; }
    case (i > 0) and (i < 1000):
        s = "positive";
    default:
        s = "big";
}
```

Les clauses de cette structure de contrôle sont évaluées en séquence. Dès que l'évaluation d'un test suivant un `case` réussit, l'énoncé associé est exécuté et on sort du `switch` . Cette structure ne nécessite pas de `break` . Pour exécuter plusieurs instructions, on utilisera un bloc (comme dans le cas `i==0` au dessus). Si aucun test ne réussit, le code associé à la clause `default` (si elle existe) est exécutée. La syntaxe du `switch` pour *Toy* est donnée ci-dessous:

```
%%
...
    stmt:           ...
            |       KSWITCH '{' cond_list defcond '}' { ..... }
                    ...
            ;
    cond_list:      cond_list KCASE expr ':' stmt
                                        { list_append($1, $3, FREE_NODE);
                                          list_append($1, $5, FREE_NODE);
                                          $$ = $1; }
            |       /* empty */         { $$ = list_create(); }
            ;
    defcond :       KDEFAULT ':' stmt   { ... }
            |       /* empty */         { ... }
            ;
```

Les actions sémantiques permettant de construire la liste des conditions/énoncés est complète. On utilise ici les fonctions génériques de `lib/list.h` . Ces fonctions permettent de gérer des liste d'objets quelconques. Ici, on ajoute donc pour chaque clause le nœud correspondant à la condition suivi du nœud de l'énoncé qui lui est associé (vous pouvez admettre que le troisième paramètre de `list_append` doit être FREE_NODE).

Une utilisation de ces fonctions est donnée ci-dessous:

```
List l = list_create();

list_append(l,  "foo", NULL);
list_append(l,  "bar", NULL);
list_prepend(l, "start", NULL);
list_append(l,  "end", NULL);
printf("taille de la liste %d\n", list_size(l));

for (List_item p = list_head(l); p; p = list_item_next(p))
  printf("%s\n", (char*) list_item_data(p));
list_destroy(l);
```

L'exécution du programme suivant affiche:

```
taille de la liste 4
start
foo
bar
end
```

Terminez l'implémentation du `switch` en *Toy*. Cela veut dire que vous devez implémenter l'action associée à la règle `KSWITCH` de `stmt` et tout ce qui en découle.

Pour tester votre extension, vous pouvez utiliser les programmes de tests:

- ok-switch1.toy,
- ok-switch2.toy,
- ok-switch3.toy,
- fail-switch.toy

3 Exceptions en Toy

On va ici ajouter un mécanisme d'exception simple au langage *Toy*. Le mécanisme retenu est une version simplifiée du mécanisme que l'on trouve dans des langages comme Java ou JavaScript.

En Toy, un bloc `try-cath-finally` a la forme:

```
int x = 0, y = 0;
try {
  // Instructions dans lesquelles une exception peut être levée
  x = 1;
  f();          // peut provoquer une exception
  y = 2;        // pas exécuté si exception

} catch {
  // Instructions exécutés si il y a une exception
  x++;
} finally {
  // Instructions exécutées dans tous les cas (exception ou pas)
  x++;
}
print(x, " ", y);  // "3 0" si exception levée et "2 2" si pas d'exception
```

Le code produit en C pour l'exemple précédent doit être:

```
TRY {
  x = 1;
  f();
  y = 2;
} CATCH {
  x++;
}
FINALLY {
  x++;
}
ENDTRY;
```

Ici, `TRY`, `CATCH`, `FINALLY` et `ENDTRY` sont des macros C définies dans `toy-runtime.h`. Ces macros utilisent le mécanisme `setjmp/longjmp` de C99. Vous n'avez pas besoin de comprendre leur fonctionnement.

Remarque:

- En *Toy*, les parties `catch` et `finally` sont optionnelles.
- Le code produit en C doit **toujours contenir** les appels aux quatre macros précédentes (et dans cet ordre), même si les clauses `catch` ou `finally` sont absentes du source *Toy*.

Pour lever une exception en *Toy*, il faut utiliser l'instruction `throw`. Cet énoncé n'admet pas de paramètre (les exceptions n'ont pas de valeur associée en *Toy*). Quant à la traduction de l'énoncé `throw` en C, elle consistera seulement en l'appel de la macro C `THROW` (définie elle aussi dans `toy-runtime.h`).

Vous pourrez tester votre implémentation pour avec les programmes de tests suivants:

- fail-try.toy
- ok-try1.toy
- ok-try2.toy
- ok-try3.toy
- ok-try4.toy
- ok-try5.toy