POLYTECH
NICE-SOPHIA

SI4 / 2016-2017

Compilation

Contrôle final

15 décembre 2016

Durée: 3 heures
Documents non autorisés

Introduction — Consignes

Pour se connecter: utiliser votre identifiant de l'université. Le mot de passe est `user@epu` pour tout le monde.

Vous devez travailler dans le répertoire `RENDU`. Ce dernier comporte deux sous-répertoires:

- `RENDU/Toy` contient une version du compilateur avec les dernières extensions vues en TD.
  Le compilateur qui vous est donné est organisé comme celui qui a été utilisé en TD. En particulier, la construction et le test se font de la même manière qu'en TD (`make` et `make tests`).

- `RENDU/Reponses` contient **vos réponses aux questions de ce sujet**. Pour chaque question, il vous sera indiqué dans quelle fichier vous devez répondre.

Important:

- L'épreuve comporte des questions *indépendantes* qui peuvent être donc exécutées dans n'importe quel ordre.
- Les questions sont données dans un ordre de difficulté (plus ou moins) croissante.
- **Vous devez répondre dans les fichiers du dossier** `RENDU/Reponses`
- **Vous serez noté sur les fichiers précédents principalement**. Ce sont eux qui constituent votre copie finale et pas seulement le programme que vous allez construire.
- Les tests entrent pour une **faible proportion** dans la note finale.
- Essayez de ne pas trop modifier la stucture des fichiers de réponse. Vous pouvez copier/coller les parties *significatives* de vos modifications (n'y copiez des fichiers entiers!!!).

- le code c'est bien, mais s'il est expliqué c'est mieux. Par exemple:

```
Avec l'introduction du type float, j'ai rajouté un test permettant de combiner
un entier et un flottant (et vice-versa) dans la fonction compatible_types:

if ((t1 == int_type   && t2 == float_type) ||
    (t1 == float_type && t2 == int_type))    return true;
```

Dans votre environnement, vous trouverez aussi:

- une copie du Cours,
- la documentation de *flex* et *bison*,
- une archive du compilateur (qui ne devrait vous servir que si vous avez *cassé* des fichiers et que vous vouliez revenir à la version distribuée au départ). Attention à ce que vous faites si vous devez revenir en arrière.

1 Primitive println

On désire étendre le langage *Toy* en lui ajoutant la primitive `println`. Celle-ci a un comportement identique à la primitive qui est déjà présente dans la version du compilateur distribuée, si ce n'est qu'elle termine l'affichage par l'impression d'un saut de ligne. Ainsi, l'affichage:

```
print("a", "b"); println("c", "d"); print("f", "g");
```

affichera:

```
abcd
fg
```

Modifiez le compilateur pour y introduire la primitive **println** dans **Toy**. Au besoin, vous pouvez si vous le voulez modifier les prototypes des fonctions existantes.

Pour tester votre extension, vous pouvez utiliser les tests qui se trouvent dans `%ok-print2.toy` et `%fail-print.toy`.

Rappel: Pour qu'un test soit *vu* par la commande `make tests`, il faut lui enlever le caractère '%' initial.

Répondre dans le fichier `RENDU/Reponses/question1.md`

1. Indiquer sommairement les modifications que vous avez apporté au compilateur pour introduire la nouvelle primitive.

2. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).

2 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. Pour cela vous pourrez vous inspirer éventuellement de l'exemple donné à la question 3.

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 qui se trouvent dans `%ok-concat.toy` et `%fail-concat.toy`.

Répondre dans le fichier `RENDU/Reponses/question2.md`

1. Quel est le code C produit pour l'initialisation de la variable `s` dans l'exemple ci-dessus.
2. Expliquez les principales choses qui doivent être modifiées dans le compilateur pour pouvoir implémenter la concaténation de chaînes de caractères.
3. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).

3 Fonctions Toy écrites en C

On veut pouvoir écrire des fonctions *Toy* dont le code est écrit en C. Pour cela, on introduit des fonctions dont le corps est compris entre les séquence de caractères `%{` et `%}`. Un exemple d'une telle fonction est donné ci dessous:

```
string concat(string s1, string s2) %{
    extern void *malloc(size_t size);
    extern void exit(int status);

    _toy_string res = malloc(strlen(s1) + strlen(s2) + 1);
    if (!res) {
        fprintf(stderr, "concat: allocate error for '%s' and '%s'\n", s1, s2);
        exit(1);
    }
    return strcat(strcpy(res, s1), s2);
%}
```

Cette fonction est présente dans le fichier `%ok-ccode1.toy`. Son prototype est écrit en *Toy* alors que sont corps est écrit en *C*.

Vous trouverez deux autres programmes de test dans les fichiers `%ok-ccode2.toy` et `%fail-ccode.toy`.

Indications:

- Pour réaliser cette extension, on va définir un nouveau type d'unité syntaxique appelée `CCODE`. Cette unité syntaxique sera donc vue comme une sorte de chaîne et ne pourra être utilisée que pour définir un corps de fonction.
- Pour simplifier, vous pouvez supposer que la taille du code constituant une chaîne `CCODE` n'excédera pas 4096 caractères.

Répondre dans le fichier `RENDU/Reponses/question3.md`

1. Expliquer sommairement le principe de fonctionnement de cette extension.
2. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).

4 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 évaluée 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 sonnée 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   { $$ = $3; }
        |       /* empty */         { $$ = NULL; }
        ;
```

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 conditionb 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 fonction 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 `%ok-switch1.toy`, `%ok-switch2.toy`, `%ok-switch3.toy` et `%fail-switch.toy`.

Répondre dans le fichier `RENDU/Reponses/question4.md`

1. Quel est le code C que doit produire le compilateur pour le `switch` précédent?
2. Pour chacun des fichiers que vous avez modifiés, indiquez la (ou les) modification(s) apportée(s).