Correction TD Compilation: TP2

Informatique 2ème année. ENSEIRB-MATMECA 2014/2015

——

▸**Exercice 1.** *Construction d'un arbre syntaxique abstrait On souhaite construire un arbre syntaxique abstrait pour les expressions. On ignorera les autres instructions ainsi que les déclarations. Le principe est de construire cet arbre dans les actions sémantiques des règles pour les expressions et affectations. On pourra prendre un arbre binaire, dont la racine définit l'opération effectuée. Par exemple :*

```
typedef struct node_s {
  char kind; // kind of operation, or leaf
  struct node_s *left, *right; // NULL if leaf
  union {
   int i;
   float f;
   char *id;
  } leaf;
} node_t;
```

Les feuilles de cet arbre seront des constantes ou des identificateurs.

1. *Déclarer ce type dans l'analyseur syntaxique et l'utiliser comme type pour les attributs de* `primary_expression`, `argument_expression_list`, `unary_expression`, `multiplicative_expression`, `additive_expression`, `comparison_expression` *et* `expression`.
2. *Faire une fonction C qui étant données 2 valeurs de type* `node`, *crée un noeud père dont le type (kind) est passé en paramètre. Faire une fonction qui crée une feuille d'un certain type (entier, flottant ou identifiant).*
3. *Faire une fonction qui affiche l'arbre d'une expression (en C par exemple).*
4. *Utiliser ces fonctions pour créer l'arbre syntaxique des expressions et l'afficher à chaque noeud* `expression`.

✂ ----------

1. Il suffit d'ajouter un champ de type `struct node` dans l'union :

   ```
   %union {
     ..
     node_t *n;
   }
   ```

   Pour toutes les variables de la grammaire citées, il faut faire :

   ```
   %type <n> primary_expression
   ```

2. Pour les fonction de creation d'un noeud :

   ```
   node_t *create(char kind, node_t *left, node_t *right) {
     node_t *n=malloc(sizeof(node_t));
     n->kind=kind;
     n->left=left;
     n->right=right;
     return n;
   }
   node_t *create_int(int i) {
     node_t *n=malloc(sizeof(node_t));
     n->kind='I'; // for leaf of type integer
     n->leaf.i=i;
     n->left=n->right=NULL;
     return n;
   }
   ```

3. Il faut choisir un codage sur un char (par exemple) pour représenter chaque opération. Pas de difficulté sinon.

   ```
   void print_node(node_t *n) {
     switch (n->kind) {
     // example for one leaf
   ```

```
  case 'I': printf("%d",n->leaf.i); break;
  // example for a binary operation
  case '+': print_node(n->left); printf("+"); print_node(n->right); break;
  /// for the assignment
  case '=': print_node(n->left); printf("="); print_node(n->right); break;
  ...
  }
}
```

4. On ajoute les actions sémantique. Par exemple :

```
primary_expression
: ICONSTANT         { $$ = create_int('I',$1); }
...
additive_expression
: multiplicative_expression   { $$ = $1 ; }
| additive_expression '+' multiplicative_expression { $$ = create_node('+', $1, $3); }
...
expression
: IDENTIFIER '=' comparison_expression  { $$ = create_node('=', create_id('V',$1), $3);
                                          print_node($$);
                                        }
```

On peut générer de la même facon un arbre syntaxique pour toute la grammaire (meme binaire). Ca peut etre pratique pour ensuite génerer le code ou parcourir l'arbre dans l'ordre que l'on souhaite, indépendamment de l'analyse syntaxique.

▸**Exercice 2.** *Threads Le but de cet exercice est la manipulation des fonctions de création et d'attente des threads. On pourra faire* `man pthread(3)` *pour avoir une liste exhaustive des fonctions utilisables pour les threads. La création de threads se fait avec la fonction* `pthread_create(3)` *qui prend notamment en argument un pointeur sur la fonction qu'exécutera le thread.*

1. *Ecrire un programme qui lance 8 nouveaux threads (en plus de celui du programme principal) sur une fonction* `void *start(void *)` *qu'on écrira. Cette fonction affichera un numéro (entre 0 et 7) passé à la fonction* `start` *par le dernier argument de* `pthread_create` *. On compilera avec l'argument* `-lpthread`*.*
2. *La fonction* `pthread_join(3)` *permet d'attendre un thread identifié par son THREAD ID (défini lors de l'appel de* `pthread_create`*). Modifiez le programme précédent pour que le thread principal attende tous les threads qu'il a crée puis affiche un message sur la sortie standard.*

Il faut faire attention au passage de paramètre de la fonction `pthread_create`. Le thread n'est pas forcément créé et executé dès que cette fonction est appelée. Ainsi, si on passe l'adresse de i, ca ne marche pas (i change de valeur avant que le thread s'éxécute).

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define POOL 8

/* Fonction exécutée par les threads. Les threads terminent quand la
fonction quitte */
void *start(void *ptr) {
  pid_t x=*(int *)ptr;
  printf("thread %d\n",x);
  return NULL; // Equivalent à pthread_exit(NULL)
}

int main() {
  pthread_t pid[POOL];
  int i;
  int val[POOL];
  /* Création des threads */
  for (i=0; i<POOL; i++) {
    val[i]=i;
    pthread_create(&pid[i], NULL, start, (int *)&i); //&val[i]);
  }
  /* Attente des autres threads */
  for (i=0; i<POOL; i++) {
    pthread_join(pid[i], NULL);
  }
  printf("processus %d termine\n",getpid());
}
```
