Projet 25 avril 2026

Langages Formels – Projet Analyse Syntaxique

Dans le projet, il s'agit d'implémenter un petit langage de programmation. Un programme peut faire des appels récursifs et générer de nouveau threads. Le programme est d'ailleurs équipé d'une liste de spécifications. En fonction de vos ambitions, vous pouvez implémenter un parmi deux projets ; la notation prendra en compte le projet que vous avez choisi.

— Projet de base : Tester si un programme est syntaxiquement correct, et en faire un "pretty print".

— Projet avancé : Simuler le programme aléatoirement pour tester si les spécifications sont satisfaites ou pas.

Vous pouvez travailler en groupes de deux et réutiliser et/ou adapter les fichiers du TP (notamment `langlex.l` et `lang.y`).

Modalités de rendu. La date de rendu est le samedi 30 mai, 18 h. Envoyez un mail à `schwoon@lmf.cnrs.fr` avec *un seul archive* (format zip ou tgz) qui contient les fichiers sources ainsi qu'un readme avec toutes les explications qui vous paraissent utiles, surtout quel projet vous avez choisi et comment le compiler. Si vous travaillez en groupes de deux, la personne qui envoye le mail mettra son parténaire en copie.

Le langage

L'analyse syntaxique doit être traitée par tous les groupes, indépendamment du niveau. La syntaxe et la sémantique du langage ne sont pas fixées formellement – il est à vous de ce faire en créant une grammaire. Votre grammaire devrait traiter au moins les trois exemples fournis (`peterson.prog`, `sort.prog`, `lock.prog`), mais vous pouvez y apporter des améliorations, à spécifier dans votre readme.

Quelques détails :

— Le programme est divisé en fonctions, dont obligatoirement une qui s'appelle *main*. L'exécution commence avec un seul thread qui exécute cette fonction.

— Au début du programme, on peut déclarer des entiers (*int*) et des tableaux d'entiers (*array*). Ces variables sont partagées entre tous les threads. Toute variable est initialisé à 0.
— Les fonctionnes peuvent prendre une liste des paramètres entiers et déclarer leurs propres variables. Ces variables appartiennent à une instance particulier de cette fonction, du coup en cas d'appels récursifs il en existe plusieurs copies. Les variables locales sont limitées aux entiers simples (sans tableaux).
— Un appel récursif est fait par un *call*, suivi par le nom d'une fonction et des arguments. Un nouveau thread est créé par *fork* avec la même syntaxe ; le nouveau thread commence donc sa vie dans la fonction spécifiée et avec les arguments fournis.
— Autre que *call* et *fork*, les instructions contiennent des affectations des variables, un *return* qui termine la fonction et des *while* et *if*.
— Dans les expressions on peut écrire *rnd(n)* qui choisit un nombre aléatoire entre 0 et $n-1$ et *?* qui est équivalent à *rnd(2)*.
— Les commentaires commencent par `//` et vont jusqu'à la fin de la ligne.

Tâches communs

Indépendamment du niveau choisi, vous devez créer une grammaire. Votre grammaire ne doit comporter aucun conflit de type "shift-reduce" ou "reduce-reduce" ; tous ces conflits sont à éliminer avant le rendu.

D'ailleurs, vous devez tester si un fichier donné est syntaxiquement correct. Ça inclut de tester s'il existe une fonction *main*, si les *call* et *fork* ont le bon nombre d'arguments, et si les arrays sont toujours utilisé avec des indices (et seulement les arrays). Il n'est pas exigé que ces vérifications soient faites par la grammaire ; il est tout à fait acceptables de créer d'abord un arbre syntaxique et ensuite tester ces conditions.

Projet de base : Analyse et affichage

Le projet de base comporte les tâches suivantes supplémentaires :
— Créer un arbre syntaxique qui représente le programme et les spécifications en mémoire.
— Reproduire le contenu du fichier dans la sortie standard. À noter que

cet affichage doit être réalisé à partir de l'arbre syntaxque, notamment, il n'est pas permis de réaliser cet affichage *pendant* l'analyse syntaxique.
— Le résultat de cet affichage doit être *sémantiquement équivalent* au fichier en entrée. Autrement dit, on pourra alimenter votre programme avec son propre résultat (et qu'il applique une transformation idempotente).
— L'affichage doit porter un minimum d'attention au formattage et à la lisibilité du résultat.

Projet étendu : Simulation

Après l'analyse syntaxique, simuler le programme un certain nombre de fois. Il s'agit des programmes non-déterministes, du coup différentes simulations peuvent donner de différents résultats. Les programmes n'ont d'ailleurs aucune garantie de terminaison, du coup il est recommandé de terminer chaque simulation après un nombre maximal d'instructions.

L'état d'un programme est défini par le contenu des variables globales et l'ensemble des threads et appels récursifs actifs, qui ont chacun un compteur d'instruction et le contenu de leurs variables locales. Une étappe de la simulation consiste à choisir aléatoirement l'un des threads actifs et d'en exécuter la prochaine instruction. Pour les *if* et *while* ceci consiste à évaluer leur condition et de choisir la prochaine instruction en fonction. L'exécution se termine si tous les threads sont terminés ou après un nombre maximal d'instructions.

Après chaque instruction on testera pour chaque spécification si elle est satisfaite (les spécifications ne prennent en compte que les variables globales). Les spécifications expriment des propriétés qui devraient être accessibles ou non accessibles. À la fin de toutes les simulations on affiche pour chaque spécifications le nombre de simulations pendant lesquelles elle était satisfaite au moins une fois. Exemple (après 100 simulations) :

```
property #1 was satisfied 36 out of 100 times
```