DLP : Développement d'un langage de programmation

Master STL, Université Pierre et Marie Curie

Année 2020–2021

TME 3 – Extension de la bibliothèque d'exécution

Christian Queinnec & Antoine Miné

Objectif : étendre la bibliothèque d'exécution d'ILP1 avec des fonctions et un nouveau type de données : le vecteur.

Buts :
- ajouter une primitive à l'interprète ;
- ajouter une primitive au compilateur ;
- ajouter une primitive à la bibliothèque d'exécution (en C) ;
- ajouter un nouveau type de données à la bibliothèque d'exécution (en C) ;
- écrire un nouveau patron C.

Tous vos fichiers seront dans le *package* `com.paracamplus.ilp1.ilp1tme3`. En particulier, quand vous êtes amenés à modifier la bibliothèque d'exécution, vous placerez vos fichiers C modifiés dans le *package* `com.paracamplus.ilp1.ilp1tme3.C`, c'est à dire dans le répertoire `Java/src/com/paracamplus/ilp1/ilp1tme3/C`, afin de préserver intacte la bibliothèque d'exécution originale d'ILP1 dans le répertoire `C`. Rappelons que la dernière étape de la compilation consiste à appeler le script `C/compileThenRun.sh` qui se charge de compiler le fichier C généré par le compilateur en y ajoutant la bibliothèque d'exécution ILP (`ilp.c`). Le chemin du script est défini dans l'attribut `scriptCommand` de `CompilerTest`. Le script sélectionne le fichier `ilp.c` contenu dans le même répertoire que le script, donc `C/ilp.c` pour le script `C/compileThenRun.sh` utilisé par `CompilerTest` d'ILP1. Afin de tester une extension du compilateur ILP qui utilise une nouvelle version de la bibliothèque d'exécution contenue dans `Java/src/com/paracamplus/ilp1/ilp1tme3/C/ilp.c`, il sera donc nécessaire de :

1. placer une copie du script `C/compileThenRun.sh` dans le répertoire `Java/src/com/paracamplus/ilp1/ilp1tme3/C` (il est inutile de modifier le script) ;
2. créer une version de `CompilerTest`, dans `com.paracamplus.ilp1.ilp1tme3.compiler.test`, dont l'attribut `scriptCommand` pointe vers le chemin complet du nouveau script (`Java/src/com/paracamplus/ilp1/ilp1tme3/C/compileThenRun.sh`).

1 Une nouvelle primitive : sinus

En ILP1, il n'est pas possible de calculer directement le sinus d'un nombre. Nous souhaitons dans cet exercice ajouter une primitive, qui sera nommée `sinus` en ILP, pour pouvoir effectuer ce calcul (attention : nous utilisons le nom `sinus` et non `sin` pour éviter les collisions avec la bibliothèque C standard qui définit également une fonction `sin`). Nous essayons d'adopter une approche générale qui permettra d'ajouter simplement par la suite d'autres fonctions si nous le souhaitons.

Travail demandé :

1. déterminez les grandes étapes des modifications à apporter ;
2. implantez ces modifications, dans l'interprète, mais aussi dans le compilateur ;
3. testez votre extension en local ;
4. ajoutez vos tests à la configuration de l'intégration continue en ajoutant une règle `TME3` au fichier `.gitlab-ci.yml` ; faites un *push* et vérifiez que votre extension fonctionne bien sur le serveur GitLab.

Remarque : vous pouvez vous inspirer de l'implantation des primitives `print` et `newline` qui sont présentes dans ILP1. Comme pour les précédents travaux, les extensions ne doivent pas modifier le code existant, mais l'étendre.

2 Un nouveau type : les vecteurs

Nous souhaitons maintenant pouvoir gérer des données stockées dans des vecteurs (ou tableaux). Il faut donc ajouter ce nouveau type dans ILP. Avec ce type, nous ajouterons trois nouvelles primitives pour manipuler les vecteurs : `makeVector`, `vectorLength` et `vectorGet` (attention à la distinction entre majuscules et minuscules) dont voici les signatures plus précises :

```
makeVector(taille, valeur)
vectorLength(vecteur)
vectorGet(vecteur, index)
```

— la primitive `makeVector` crée un vecteur ayant pour taille son premier argument, chaque cellule de ce vecteur sera initialisée avec le second argument ;
— la primitive `vectorLength` renvoie la taille du vecteur qu'elle reçoit en argument ;
— la primitive `vectorGet` renvoie la valeur à la position `index` du vecteur.

Attention, `taille`, `valeur`, `vecteur`, `index` sont des expressions qu'il faudra évaluer à l'exécution. Ce ne sont pas forcément des constantes litérales.

Travail demandé :

1. déterminez les grandes étapes des modifications à apporter ;
2. implantez ces modifications, dans l'interprète, mais aussi dans le compilateur ;
3. testez avec au moins un programme utilisant des vecteurs, d'abord en local, puis sur le serveur GitLab.

3 Rendu

Comme pour le TME précédent, vous effectuerez un rendu en vous assurant que tout le code développé a été envoyé sur le serveur GitLab (*push*), que les tests d'intégration continue ont été configurés et fonctionnent sur le serveur, et enfin en ajoutant un tag « rendu-initial-tme3 » en fin de séance, puis un tag « rendu-final-tme3 » quand le TME est finalisé.