1 Installation du compilateur Lisaac

Installez le compilateur via son archive et testez-le avec les exemples du répertoire `lisaac/example/`. (exemple : `lisaac hello_world`)

2 Test des diffèrents modes du compilateur

Décompressez le fichier `exo.zip`. Il contient l'ensemble des exemples que nous allons étudier.

1. Editez le programme `factorial.li` à l'aide d'Emacs. Vérifiez la prise en compte de la mise en forme du mode Lisaac sous Emacs (couleur, auto-indentation, ...).
2. Compilez le programme sans option et exécutez-le. Vous pouvez constater la vitesse de calcul et la facilité de prise en charge transparente des nombres astronomiques.
3. Compilez le programme avec l'option `-d` et exécutez-le. Suivez la pile d'exécution pour comprendre l'erreur. Cette option permet une vérification rigoureuse de votre code à l'exécution.

Remarque :

- L'option `-O` permet d'activer toutes les optimisations du compilateur. Le code s'exécute de 2 à 10 fois plus vite selon votre programme. Ici, le programme est trop petit pour tester cette diffèrence.

Intérêt :

- Mettre en évidence la programmation par contrat avec l'option `-d`.
- Vitesse accrue avec l'option `-O`.
- Facilité d'utilisation des nombres sans limite de taille.

3 Interface graphique en Lisaac

Testez le programme `graph.li`, il vous montre le *début* d'une interface graphique multi-plateforme, simple et dynamique.

1. Dans le fichier `my_win.li` vous pouvez constater la simplicité de la gestion graphique et la prise en compte des événements du système. Testez le programme directement sous Linux (le prototype à compilé est `graph.li`. C'est ce prototype qui utilise le prototype `my_win.li`). Ce programme vous donnera exactement le même résultat sous Windows, sous Isaac, et même sous DOS !
2. Je vous invite à transformer la ligne : `LAB_OUT.create " Vive Lisaac !"` par `LAB_BMP.create "save.bmp" + LAB_OUT.create " Vive Lisaac !"` et à constater la diffèrence.

Intérêt :

- Présentation de la *Graphic User Interface* du Lisaac.
- Le compilateur cherche automatiquement les prototypes nécessaire à la compilation du prototype principal. Les chemins (*directory*) pour sa recherche sont spécifiés dans le fichier `lisaac/path.li`.
- Facilité d'utilisation et multi-plateforme.
- Calcul de l'interface dynamique (*Ce point est en cours de développement et l'exemple n'est pas encore probant*).

4 Compilation de IsaacOS sous Linux

L'archive `IsaacOS.zip` est une partie du système d'exploitation Isaac.

1. Je vous invite à compiler le système pour architecture Linux avec la commande : `lisaac startup.li -t os_lin`

Intérêt :

- Présentation d'Isaac sous Linux.
- La compilation d'IsaacOS pour une autre architecture n'est qu'un paramètre de compilation...(`-t os_lin` pour Linux, `-t os_win` pour Windows, `-t os_x86` pour la version natif sous *Intel*, ...).

5 Redéfinition de slot

Editez le programme `redef.li`, il vous présente la transparence entre message calculé et message de donnée. Aussi, cette transparence est accrue en donnant la possibilité de transformer une méthode en donnée...

La possibilité de cette transformation (méthode en donnée) permet notamment de gérer rapidement un *cache* d'un calcul d'un slot. C'est à dire que lors d'un premier appel d'un slot, il calcule sa valeur de retour (une méthode), puis, pour les appels suivant, il renvoie directement le résultat (donnée).

1. Je vous invite à créer et tester avec deux appels le slot suivant :

```
- un_kilo:INTEGER <-
(
  ''C'est mon premier appel, alors je calcule !''.print;
  un_kilo := 2 << 10
);
```

Vous venez de créer une sorte de petit *cash* d'un calcul d'une valeur.

Intérêt :

- Redéfinition dynamique de la valeur d'un slot par du code ou par une donnée.
- Non distinction entre slot de donnée et slot de code.

6 Héritage dynamique

Editez le programme `herit.li`, il vous montre une des plus grandes capacités des prototypes : l'héritage dynamique.

1. Je vous invite à ajouter sous le commentaire QUESTION, une affectation du slot `parent_insect` avec le prototype BUTTERFLY qui est dejà présent. Puis, testez votre programme. Imaginez le pouvoir d'expressivité que vous avez en modifiant durant l'exécution les parents de vos objets !

Intérêt :

- Mettre en avant l'héritage dynamique. Cette possibilité est particulièrement pratique pour métamorphoser l'état d'un même objet durant l'exécution.

7 Utilisation des blocks

Editez le fichier `action.li`.

1. Sur le point QUESTION 1, que pouvez-vous dire par rapport à un programme C équivalent ?
2. L'utilisation des blocs à évaluation retardés permet de factoriser du code, même en cas de variation sémantique de celui-ci. (Ici, les deux collections sont sémantiquement très diffèrentes, nous sommes loin du pauvre `define` C).
3. Enlevez les commentaires des lignes de code et réalisez votre propre `action2` permettant le calcul de la somme des éléments d'une collections (utilisez la variable `somme`).

Intérêt :

- Agrandissement dynamique des collections.
- Utilisation des BLOCK pour factoriser du code.
- Présentation de la généricité pour une implémentation élégante des collections (absent en Java, obligeant des *cast*s dangeureux à l'exécution)
- Uniformisation des profils des méthodes pour les diffèrents type de collections. Une liste chaînée ou autres collections s'utilisent comme un tableau.

Merci de votre attention, et bon code...