

Revision: 1.1

Master d'informatique 2009-2010
 Spécialité STL
 « Implantation de langages »
 ILP – MI016
 épisode ILP7

C. Queinnec

- Compilation indépendante, séparée
- Édition de liens

Compilation indépendante/séparée

- Tronçonner la définition d'une application
- tout en assurant la même sémantique qu'une définition unique.

Quelle est l'unité de compilation : fichier (`load`) ou module (`use`) ?

Partage

Le problème se pose sur les variables ou fonctions globales :
 comment les partager ?

ILP n'a qu'un seul espace de noms. Un même nom global, à deux endroits différents, doit désigner la même chose.

```
// fichier f
function foo (x) {
  print("foo()");
  return 2*x
}
foo(11);

// fichier g
function bar (x,y) {
  print("bar()");
  return x*y
}
foo(22);
foo = bar;
foo(33, 44);
```

Évaluation séquentielle des expressions dans le même espace global.

- Évaluation avec
 - `% ilp f.ilp g.ilp`
- `#include "fichier"`
- `use module;`
 - Quel est le lien entre un nom de module et un nom de fichier ?
- `load("fichier")`

Aux variables globales sont associées des adresses qui permettent d'obtenir des valeurs (fonctionnelles ou autres).

En C : distinguer déclaration et initialisation, même espace de noms pour fonctions et variables globales (pareil pour `ld`).

En C : `ld` est assez primitif.

- Utiliser l'éditeur de liens `ld`
- Les variables globales d'ILP sont des variables globales de C
- Les `.o` de C sont les `.c` d'ILP

```
% ls
f.ilp g.ilp
% ilpc f.ilp g.ilp && ls
f.c f.ilp g.c g.ilp # des fichiers C
% ilpld f.c g.c && ls # travail sur fichiers C
a.out f.c f.ilp g.c g.ilp main.c
% ./a.out
```

Principes de traitement

Exemple : fichier f

Pour toute unité de compilation *m*,

- Compiler `function foo () {...}` en `foo = function () {...}`
- Collecter les variables globales (les déclarer en `extern`)
- Compiler les variables globales par leur nom en C
- Placer tout le code de *m* dans une fonction `m_init()`
- Déterminer un ordre de chargement des fichiers
- Engendrer un fichier `main.c` qui déclare toutes les variables globales, les exporte et invoque séquentiellement toutes les `m_init()`

```

/* fichier f */
extern ILP_Object foo;

static ILP_Object
ILP_foo (ILP_Object x)
{
    ILP_print(...);
    return ILP_Times(ILP_Integer2ILP(2), x);
}

void
ILP_f_init () {
    /* les initialisations */
    foo = ILP_Function2ILP(ILP_foo, 1);
    /* les instructions */
    ILP_Invoke(foo, 1, ILP_Integer2ILP(11));
}

```

Nouveau type

Exemple : fichier g

Au passage, la bibliothèque d'exécution s'enrichit d'un type fonctionnel créé par `ILP_Function2ILP` (qui prend l'arité) et d'un mécanisme d'invocation `ILP_Invoke` (qui prend le nombre d'arguments).

```

/* fichier g */
extern ILP_Object foo;
extern ILP_Object bar;

static ILP_Object
ILP_bar (ILP_Object x, ILP_Object y)
{
    ILP_print(...);
    return ILP_Times(x, y);
}

void
ILP_g_init () {
    /* les initialisations */
    bar = ILP_Function2ILP(ILP_bar, 2);
    /* les instructions */
    ILP_Invoke(foo, 1, ILP_Integer2ILP(22));
    foo = bar;
    ILP_Invoke(foo, 2, ILP_Integer2ILP(33), ILP_Integer2ILP(44));
}

```

Lancement

```

/* Engendre automatiquement par

cat *.c | \
sed -re 's/^extern (ILP_Object .+);/\1 = NULL;/'

*/
ILP_Object foo = NULL;
ILP_Object bar = NULL;

int
main () {
    ILP_f_init();
    ILP_g_init();
    return EXIT_SUCCESS;
}

```

Conclusions partielles

- Avantages :
 - protocole simple pour modules faits main.
 - Accès à variables globales C simple (mais conversion de leurs valeurs vers LLP).
 - Couplage à bibliothèques C simple.
- Inconvénients :
 - toute variable globale est potentiellement modifiable,
 - plus aucune intégration possible de fonction globale.
 - Pas de chargement dynamique de module.
 - Ordre de chargement des modules délicat (certaines variables globales pourraient ne pas être initialisées).
 - Les `.c` sont lus par `ilpld` pour collecter les variables globales qui peut aussi numéroter les classes.

Stratégie 2

Mise en œuvre

Plus dynamique, plus de contrôle sur les partages. Plus besoin des fichiers `.c`

Permet le renommage de variables globales (et ainsi prendre en compte les traductions de noms de variables).

```

% ls
f.ilp g.ilp
% ilpc f.ilp g.ilp && ls
f.o f.ilp g.o g.ilp # des .o
% ilpld f g && ls # travail sur noms
a.out f.o f.ilp g.o g.ilp
% ./a.out

```

Exemple : fichier f

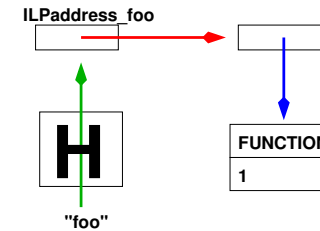
```

/* fichier f */
static ILP_Object* ILPAddress_foo;

static ILP_Object ILP_foo (ILP_Object x)
{
    ILP_print(...);
    return ILP_Times(ILP_Integer2ILP(2), x);
}

void
ILP_f_init () {
    /* l'edition de liens */
    ILPAddress_foo = ILP_register("foo");
    /* les initialisations */
    *ILPAddress_foo = ILP_Function2ILP(ILP_foo, 1);
    /* les instructions */
    ILP_Invoke(*ILPAddress_foo, 1, ILP_Integer2ILP(11));
}

```



Exemple : fichier g

```

/* fichier g */
static ILP_Object* ILPAddress_foo;
static ILP_Object* ILPAddress_bar;

static ILP_Object
ILP_bar (ILP_Object x, ILP_Object y)
{
    ILP_print(...);
    return ILP_Times(x, y);
}

void
ILP_g_init () {
    /* l'edition de liens */
    ILPAddress_foo = ILP_register("foo");
    ILPAddress_bar = ILP_register("bar");
    /* les initialisations */
    *ILPAddress_bar = ILP_Function2ILP(ILP_bar, 2);
    /* les instructions */
    ILP_Invoke(*ILPAddress_foo, 1, ILP_Integer2ILP(22));
    *ILPAddress_foo = *ILPAddress_bar;
    ILP_Invoke(*ILPAddress_foo, 2, ILP_Integer2ILP(33), ILP_Integer2ILP(44));
}

```

Lancement

Le module `main` correspond à une simple concaténation. La fonction d'initialisation est la seule chose exportée d'un module.

```

/* Engendre automatiquement */
int
main () {
    ILP_init();
    /* pour chaque module: */
    ILP_f_init();
    ILP_g_init();
    return EXIT_SUCCESS;
}

```

Espace global

La fonction `ILP_register` maintient une table associative (chaîne C vers valeur ILP). Elle repose sur des maillons de la forme :

```
struct HashItem {
    char*      name; /* Nom variable globale */
    ILP_Object value; /* valeur d'icelle */
}
```

L'adresse du second champ du maillon deviendra la valeur des pointeurs d'accès `ILPaddress_foo`.

Avantages : prise en compte des différences de nommages des variables globales entre ILP et C. Plus besoin de conserver les fichiers `.c` (sauf pour mise au point).

Inconvénients : indirection pour accès à variables globales.

Extensions

La directive (pas l'instruction) `use f` dans le module `g` revient à ajouter `ILP_f_init()` dans la fonction d'initialisation du module `g`. Comme cela fixe un ordre, on peut utiliser cet ordre pour presque se passer de l'édition de liens.

```
/* Engendre automatiquement */
int
main () {
    ILP_Symbols hash = ILP_makeSymbols();
    ILP_g_init(hash); /* invoquera ILP_f_init(hash)
    return EXIT_SUCCESS;
}
```

Prévenir la double inclusion !
Pas de cycle !

Renommages

Renommage à l'export `foo@current = foobar`

```
void
ILP_g_init (ILP_Symbols hash) {
    ILPaddress_foo = ILP_register(hash, "foobar");
    ...
}
```

Renommage à l'import `foobar = bar@current`

```
void
ILP_g_init (ILP_Symbols hash) {
    ...
    ILP_f_init(hash);
    ILPaddress_bar = ILP_register(hash, "foobar");
    ...
}
```

Manipulation de la table des symboles globaux (comme en Perl) avec de nouvelles fonctions telles que `ILP_unregister`

Conclusions générales d'ILP

- Lecture de programmes imparfaits
- Progression en Java
- Génération de code C
- Meilleure connaissance des exceptions et des objets