TP compilation : compilateurs VAR → FUN → CLL → IMP

On veut compiler les programmes en langage VAR vers des programmes IMP équivalents, en passant par deux langages intermédiaires FUN et CLL.

1 Compilation CLL → IMP

Le langage CLL va être compilé directement vers IMP. Par rapport à IMP, le langage CLL remplace l'instruction `goto` par un mécanisme d'appel de procédure. Il permet en outre de définir des procédures sans paramètres.

1.1 Spécification du langage de CLL

Expressions Les expressions et expressions gauches de CLL et IMP sont identiques.

Instructions Les instructions `nop`, `exit`, `print`, `:=`, `if` et `while` conservent la même forme que dans IMP. Les définitions d'étiquettes et les sauts `goto` disparaissent. On a en revanche deux nouvelles formes d'instructions pour l'appel de procédure et la fin de procédure :

$$\begin{array}{rcl} \mathit{instr} & ::= & \mathit{l\_expr}\ \texttt{(}\ \texttt{)}\ \texttt{;} \\ & | & \texttt{return}\ \texttt{;} \end{array}$$

La procédure appelée est représentée par une expression gauche, elle sera donc techniquement identifiée par son adresse (plus précisément par l'adresse de son code).

Définition de fonctions Le langage CLL permet de définir des fonctions avec la forme suivante :

$$\mathit{proc\_decl} \ ::= \ \texttt{id}\ \texttt{(}\ \texttt{)}\ \texttt{\{}\ \mathit{instr}^*\ \texttt{\}}$$

Le corps de la fonction est formé par une séquence d'instructions arbitraires, qui peut contenir un certain nombre d'instructions `return`.

Programmes Un programme CLL est constitué d'une séquence de définitions de procédures, ainsi que d'un ensemble de déclarations de variables globales prenant la même forme que dans les langages précédents.

On supposera que l'une des procédures s'appelle `main` et vérifie les conditions suivantes :

— elle ne contient pas d'instruction `return`,

— elle se termine par une instruction `exit`.

Exemple Voici un exemple de programme CLL.

```
main() {
  while (a < 123) {
    print(a);
    incr_a();
  }
  print(10);
  exit;
}

incr_a() {
  a := a + 1;
  return;
}
```

```
.data
a: 97
```

1.2 Conventions du compilateur

Le programme compilé devra maintenir à chaque instant une *chaîne d'appels*, sous la forme d'une liste chaînée dont chaque cellule correspond à un appel de procédure en cours et contient :

1. l'adresse à laquelle il faudra revenir à la fin de l'exécution de cette procédure,
2. un pointeur vers la cellule précédente, qui correspond à la procédure appelante.

La manipulation des fonctions utilisera deux variables globales dédiées :
- `return_address` sert à transmettre à une procédure appelée l'adresse à laquelle elle devra revenir une fois qu'elle aura terminé son exécution,
- `frame_pointer` contient l'adresse de la cellule courante de la chaîne d'appels.

Protocole d'appel Pour réaliser un appel de procédure, on exécute les opérations suivantes.

1. L'appelant, avant l'appel :
   - enregistre l'adresse de retour dans la variable `return_address` (pour connaître cette adresse, introduire une nouvelle étiquette juste après l'appel), et
   - saute au code de la procédure appelée.
2. L'appelé, au début de l'appel :
   - Crée sur la pile une nouvelle cellule de la chaîne d'appels, dans laquelle il stocke la valeur placée dans `return_address`, et
   - met à jour `frame_pointer` en conséquence.
3. (exécution du code de la procédure appelée)
4. L'appelé, à la fin de l'appel :
   - replace le sommet de la pile (`stack_pointer`) au niveau de la cellule d'appel courante (`frame_pointer`)[1],
   - restaure `return_address` et `frame_pointer` avec les valeurs sauvegardées dans la première cellule de la chaîne d'appels, et
   - retire cette cellule de la pile.
5. L'appelant, après l'appel :
   - reprend son exécution normalement.

On fera en sorte que l'étape 4 du protocole soit exécutée à la fin de l'exécution de la procédure, même en l'absence d'instruction `return`.

Structure du compilateur Pour compiler CLL vers IMP, on veut écrire une fonction de traduction prenant en entrée un arbre de syntaxe abstraite CLL et produisant un arbre de syntaxe abstraite IMP.
- La syntaxe abstraite de CLL est définie par les fichiers `CLL.ml`, `CLLInstr.ml`, `IMPExpr.ml` et `Op.ml`.
- La fonction de traduction se trouve dans le fichier `CLLtoIMP.ml`

1.3 Contrat de base : première étape

Compléter le fichier `CLLtoIMP.ml` pour obtenir une fonction de traduction entre les syntaxes abstraites CLL et IMP.

1. Cette action équivaut à retirer de la pile tout ce qui y a éventuellement été placé lors de l'exécution du code de la procédure.

2 Compilation FUN → CLL

Le langage FUN améliore CLL en remplaçant les procédures par des fonctions pouvant recevoir des paramètres et renvoyer un résultat. Le langage FUN va être compilé vers CLL.

2.1 Spécification du langage FUN

Expressions Les expressions et expressions gauches de FUN sont identiques à celles de CLL et IMP. Petite subtilité toutefois : une étiquette `id` peut maintenant désigner soit un paramètre de fonction soit une variable globale. Un paramètre de fonction portant le même nom qu'une variable globale masque cette variable globale à l'intérieur de la fonction.

Instructions Les instructions `nop`, `exit`, `print`, `:=`, `if` et `while` conservent la même forme que dans CLL et IMP.

Les instructions d'appel et de fin de procédure de CLL sont remplacées respectivement par une instruction d'appel de fonction avec paramètres et une instruction de renvoi d'un résultat.

$$
\begin{array}{rcl}
instr & ::= & \ldots \\
 & | & l\_expr \texttt{ := } l\_expr \texttt{ (} expr \texttt{ , } \ldots \texttt{ , } expr \texttt{) ;} \\
 & | & \texttt{return (} expr \texttt{) ;}
\end{array}
$$

où $expr$, … , $expr$ désigne un nombre arbitraire d'expressions (potentiellement zéro) séparées par des virgules. La forme x := $f(e_1, \ldots, e_n)$ désigne un appel à une fonction f avec les paramètres e_1 à e_n, dont le résultat est placé dans la variable x. La fonction f est une expression gauche, c'est-à-dire qu'on y accède par son adresse (en l'occurrence, l'adresse de son code).

Définition de fonctions La définition d'une fonction FUN, similaire à une définition de procédure CLL, contient en outre les noms des paramètres formels.

$$fun\_decl \quad ::= \quad \texttt{id (id, ..., id) \{ } instr^* \texttt{ \}}$$

où `id`, …, `id` désigne un nombre arbitraire d'identifiants (potentiellement zéro) séparés par des virgules. Le corps de la fonction est formé par une séquence d'instructions arbitraires, qui peut contenir un certain nombre d'instructions `return` (avec résultat).

Programmes Un programme FUN est constitué d'une séquence de définitions de fonctions, ainsi que d'un ensemble de déclarations de variables globales prenant la même forme que dans les langages précédents.

On supposera que l'une des fonctions s'appelle `main` et vérifie les conditions suivantes :

— elle attend zéro paramètres,
— elle ne contient pas d'instruction `return`,
— elle se termine par une instruction `exit`.

Exemple Voici un exemple de programme FUN.

```
main() {
  while (a < 123) {
    print(a);
    a := succ(a);
  }
  print(10);
  exit;
}
```

```
succ(n) {
  return(n+1);
}

.data
a: 97
```

2.2 Conventions du compilateur

La pile va servir à transmettre les paramètres d'un appel de fonction à la fonction appelée. Le résultat sera retransmis de la fonction appelée au contexte appelant via variable globale dédiée `function_result`.

Protocole d'appel Pour réaliser une appel de fonction avec n paramètres, on exécute les opérations suivantes.

1. L'appelant, avant l'appel :
   — place sur la pile les valeurs des n paramètres, dans l'ordre, et
   — effectue un appel de procédure CLL.
2. L'appelé, au début de l'appel, ne fait rien de particulier.
3. (exécution du code de la fonction appelée)
4. L'appelé, à la fin de l'appel :
   — place la valeur renvoyée dans la variable globale `function_result`, et
   — exécute une fin de procédure CLL.
5. L'appelant, après l'appel :
   — retire les paramètres de la pile,
   — récupère la valeur placée dans `function_result` pour la transférer dans la variable cible.

Forme de la pile et accès aux paramètres Les paramètres d'un appel de fonction sont placés sur la pile juste avant la cellule de la chaîne d'appel et peuvent donc être considérés comme faisant partie de cette cellule. On peut accéder aux paramètres de la fonction avec une adresse calculée en fonction de `frame_pointer` (dont on rappelle qu'il pointe vers la case RA).

fond de la pile	...	p_0	...	p_{n-1}	FP parent	RA	...

Ainsi, on accède au premier paramètre à l'adresse `frame_pointer + n + 1` et au dernier paramètre à l'adresse `frame_pointer + 2`. C'est à la traduction de FUN vers CLL de remplacer chaque accès à un paramètre formel par un accès mémoire à la bonne adresse.

Structure du compilateur Pour compiler FUN vers CLL, on veut écrire une fonction de traduction prenant en entrée un arbre de syntaxe abstraite FUN et produisant un arbre de syntaxe abstraite CLL.

— La syntaxe abstraite de FUN est définie par les fichiers `FUN.ml`, `FUNInstr.ml`, `IMPExpr.ml` et `Op.ml`.
— La fonction de traduction se trouve dans le fichier `FUNtoCLL.ml`.

2.3 Contrat de base : deuxième étape

Compléter le fichier `FUNtoCLL.ml` pour obtenir une fonction de traduction entre les syntaxes abstraites FUN et CLL.

3 Compilation VAR → FUN

3.1 Langage VAR

La spécification du langage VAR est celle donnée dans le document précédent (interprète pour VAR). *À noter :* une variable locale à une fonction portant le même nom qu'un paramètre de cette fonction ou qu'une variable globale les masque tous deux dans le corps de la fonction.

3.2 Conventions du compilateur

La pile va servir à stocker les valeurs des variables locales à une fonction. On utilisera plus précisément l'espace placé immédiatement après la cellule courante de la chaîne d'appel.

Protocole d'appel Une fonction utilisant des variables locales est responsable de leur allocation et de leur désallocation sur la pile. Ainsi, une fonction VAR est transformée en une fonction FUN dont le code :

— commence par le placement sur la pile des valeurs de ses variables locales (ou zéro pour des variables qui n'ont pas de valeur initiale explicite), et
— termine par l'enlèvement de ces valeurs.

Remarquez que l'enlèvement des valeurs ne nécessite aucune action à ce niveau : la traduction de CLL vers IMP est déjà supposée introduire une réinitialisation de la valeur de `stack_pointer` à la fin de l'appel, ce qui a exactement l'effet escompté.

Forme de la pile et accès aux variables locales Les variables locales d'une fonction sont placées sur la pile juste après la cellule de la chaîne d'appel et peuvent donc être considérées comme faisant partie de cette cellule. On peut accéder aux variables locales avec une adresse calculée en fonction de `frame_pointer` (dont on rappelle qu'il pointe vers la case RA).

fond de la pile	...	p_0	...	p_{n-1}	FP parent	RA	x_0	...	x_{k-1}	...

Ainsi, on accède à la première variable locale à l'adresse `frame_pointer - 1` et à la dernière à l'adresse `frame_pointer - k`. C'est à la traduction de VAR vers FUN de remplacer chaque accès à une variable locale par un accès mémoire à la bonne adresse.

Structure du compilateur Pour compiler VAR vers FUN, on veut écrire une fonction de traduction prenant en entrée un arbre de syntaxe abstraite VAR et produisant un arbre de syntaxe abstraite FUN.

— La syntaxe abstraite de VAR est définie par les fichiers `VAR.ml`, `FUNInstr.ml`, `IMPExpr.ml` et `Op.ml`.
— La fonction de traduction se trouve dans le fichier `VARtoFUN.ml`.
— Par ailleurs, les fichiers `VARLexer.mll` et `VARParser.mly` fournissent un analyseur syntaxique pour le langage VAR.

3.3 Contrat de base : troisième étape

Compléter le fichier `VARtoFUN.ml` pour obtenir une fonction de traduction entre les syntaxes abstraites VAR et FUN.

3.4 Contrat de base : produit fini

Le fichier `VARCompiler.ml` définit l'exécutable principal, qui prend en entrée un fichier source `.var`, et produit un fichier cible `.imp` ainsi que deux fichiers intermédiaires `.fun` et `.cll`. Vous n'avez rien de plus à faire ici.

4 Extensions

Les extensions suivantes, de difficulté variable, sont presque toutes accessibles dès la mise en place du langage FUN.

4.1 Paramètres pour `main`

Permettre à la fonction `main` de prendre des paramètres (à passer sur la ligne de commande).

4.2 Fonction primitives

Retirer l'instruction `print` du langage de surface, pour faire de `print` une fonction ordinaire.

4.3 Déclaration des variables locales au fil de l'eau

Déclaration au fil de l'eau Définir une variante du langage VAR, dans laquelle les déclarations des variables locales d'une fonction peuvent être mélangées avec les instructions de cette fonction. Ce langage pourra être compilé vers VAR. On pourra, mais ce n'est pas obligatoire, généraliser les variables globales de la même manière.

Initialisations enrichies Étendre le langage VAR (ou la variante définie à l'occasion du paragraphe précédent) pour que l'initiliasition d'une variable puisse se faire avec une expression arbitraire et plus seulement une constante entière immédiate.

Si cette extension est combinée avec la précédente, faire attention à ce que les valeurs calculées pour l'initialisation des variables soient les bonnes.

4.4 Appels de fonctions dans des expressions

Définir une variante du langage VAR (ou de l'une de ses extensions, ou du langage FUN), dans laquelle un appel de fonction `f(e_1, ..., e_n)` peut intervenir n'importe où dans une expression. Ce langage doit être compilé vers VAR (ou extension, ou FUN) en associant à chaque appel de fonction une nouvelle variable locale (ou globale).

4.5 Optimisation des appels terminaux

Introduire une nouvelle forme d'instruction `return f(e_1, ..., e_n)` effectuant un appel terminal à une fonction `f`, et faire que l'exécution d'un tel appel ne crée pas de nouvelle cellule dans la chaîne d'appels.

On pourra se limiter aux cas où la fonction appelée a un nombre de paramètres inférieur ou égal à la fonction appelante. Quels aménagements faire sinon ?

4.6 Passage par référence

Opérateur adresse Ajouter au langage VAR un opérateur `&` permettant de récupérer l'adresse représentée par une expression gauche.

Passage par référence Définir une variante du langage VAR (compilée vers la version étendue de VAR du paragraphe précédent) où on peut indiquer avec l'annotation `&` qu'un paramètre de fonction doit être passé par référence. Cette variante n'a besoin d'aucune autre forme d'utilisation de `&`.