Compilation

Génération de code

C. ALIAS

Avec ce TP, nous attaquons maintenant le coeur du problème: comment se fait la génération de code machine à proprement parler. Afin de démystifier tout ca, téléchargez `src2_codegen.tgz`, et décompactez avec `tar xvfz src2_codegen.tgz`.

Comme on l'a vu cours, un compilateur est constitué de plusieurs parties qui intéragissent. Chaque partie est matérialisée par un fichier:

```
          lexer.l: analyse lexicale
             PUIS:
         parser.y: analyse syntaxique + génération de code
          UTILISE:
code_generation.*: briques élémentaires de code machine
   symbol_table.*: variables + position en mémoire
       register.*: génération de registres virtuels
          label.*: génération et pile de labels pour les sauts
                   (utilisé pour coder le if et le while)
           test.d: jeu de test
```

Regardons tout çà dans le détail...

Table des symboles: *symbol_table.**

La table des symboles contient la liste des variables manipulées par la fonction en cours de traduction. La table des symboles fourni pour chaque variable sa position dans la pile (*via* get_address()) et son registre virtuel (*via* get_register()). Ceci nous permettra de résoudre l'accès aux variables.

1. Inspecter symbol_table.h et symbol_table.c
2. Dans main.c, jouer avec les fonctions de la table des symboles:
   - Créer un argument n et deux variables locales x et y.
   - Afficher le registre et l'adresse dans la pile de chaque variable.
3. Quel est le rôle de emit_local() ? Tester...

Primitives de génération de code: *code_generation.**

1. Inspecter chaque fonction de code_generation.h et code_generation.c.
2. Quel intérêt de passer par des primitives de génération de code ?
3. Dans main.c, générer le code pour additionner x et y et placer le resultat dans un nouveau registre. On créera un nouveau registre avec la fonction new_register() définie dans register.h.
4. A quoi servent les primitives de la catégorie 3/ ? Etudier le fonctionnement de cmp_g().
5. A quoi sert jmp_if_false() ?
6. A quoi servent emit_prelude() et emit_postlude() ? Tester dans main.c et analyser le code produit.

Génération de code (dirigée par la syntaxe): *parser.y*

expr: (L99-)

- Quel type est associé au non-terminal expr ? Pourquoi ?
- Quel est le rôle de caller_arg_list (L130) ?
- Compléter MINUS et MUL (utiliser les primitives de code_generation.c).
- Tester sur la fonction add() (dans le fichier `test.d`).

conditions: (L145-)

- Quel type est associé au non-terminal test ? Pourquoi ?
- Compléter EQUAL et NEQUAL
- Tester sur la fonction discriminant().

stmt: (L170-)
→ *Affectations (assignments):*

- Tester sur la fonction add(). Observer où sont stockés result, a et b.

→ *Condition (if):*

- Observer la partie dédiée au if. A quoi sert label_endif ?
- Tester sur la fonction fact(). Quels sont les rôles de label_2 et label_1 ? A quoi servent push_label() et pop_label() ? Que se passerait-il sans ?

→ *While:*

- Donner le schéma de traduction de la boucle while.
- Ajouter les actions sémantiques pour générer le code.

→ *For:*

- Donner le schéma de traduction de la boucle for.
- Compléter lexer.l et parser pour reconnaitre une boucle for.
- Ajouter les actions sémantiques pour générer le code.

caller_arg_list: (L258-)

- A quoi sert le code produit par ce non-terminal ?
- Repérer et comprendre le code produit sur la fonction fact().

function: (L258-)

- Quel type de code doit etre généré localement par les actions de function ?
- Comprendre les actions effectuées par decl_args et decl_local_vars.
- Compléter les parties manquantes.
- Tester sur la fonction fact(). La récursivité est elle gérée ?

Conclusion: Il ne reste plus qu'a remplacer les registres virtuels r_1, r_2,... par des registres physiques AX, BX,... pour obtenir un code assemblable. Cette étape est plus compliquée qu'il n'y parait, comme on le verra au TD suivant.