Réalisation d'un Compilateur C

Fabrice Bellard

But du projet:
Faire un auto-amorçage pour démontrer que le compilateur fonctionne effectivement.

1. Les choix

2. L'architecture globale

3. Le compilateur C

4. L'assembleur

5. La machine virtuelle

6. Les Résultats

Les Choix - 1

Nous avons envisagé la compilation de deux types de langage:

- CAML
- C

Nous avons choisi le C par goût personnel et pour pouvoir tester le compilateur sur une large gamme de sources.

Les grands choix:

- Pas de préprocesseur: on utilise `gcc -E`
- Pas de flottants
- Pas de reprise des erreurs

Les Choix - 2

- On utilise `flex(lex)` et `bison(yacc)` pour le *parser*. Donc le compilateur doit pouvoir accepter le code généré par ces 2 utilitaires.

- Gestion de modules séparés, donc un système d'édition de liens est nécessaire.

- Dans la mesure du possible, on reste conforme à la norme *ANSI C*.

- Utilisation d'une machine à pile. Simple, mais empêche l'évolution du compilateur.

- Pour des questions de performances, on est dépendant du *hardware*: taille des pointeurs, alignement des données, *endianité*.

- On s'impose enfin la contrainte de dépendre au minimum des fonctions des librairies standards: on recompile une partie du code pour `stdlib.h`, `stdio.h`, `stdarg.h` et `string.h`.

Architecture Générale - 1

Compilateur C (`fbcc`):
(4700 lignes)

1. *Lexer*
2. *Parser*
3. Gestion des déclarations
4. Production des données pour les variables statiques
5. Typage des expressions
6. Propagation des constantes
7. Génération de code assembleur

Assembleur (`fbas`):
(700 lignes)

1. *Lexer* - *Parser*
2. Gestion des symboles exportés, globaux, ou privés
3. Production d'un exécutable relogeable avec table de relocation

Architecture Générale - 2

Machine virtuelle (`fbvm`):
(900 lignes)

1. Chargement de l'exécutable et relocation
2. Emulation des instructions
3. Appel de quelques fonctions de la librairie C standard grâce à des *traps*

Librairies (`startup.s`,`fblib.c`):
(900 lignes)

1. Code de *startup*
2. Compilation des fonctions {`fvs`}`printf` et `stdarg.h`
3. Compilation des fonctions de `string.h`

Le Compilateur C - 1

Caractéristiques ANSI non implémentées:

- Types `long`, `float` et `double`
- Qualificatifs de types `const` et `volatile`
- Affectations de structures, passage par valeur et retour de structures
- Caractères de type `wchar_t`
- Parsing des constantes de type `unsigned int`
- Certaines formes d'initialisations statiques
- Quelques contrôles de type dans les expressions et structures de contrôle.

Le Compilateur C - 2

Exemples de caractéristiques implémentées:

- Anciens et nouveaux prototypes de fonctions, avec les ... pour indiquer une fonction à nombre de paramètres variables
- Blocs dans les fonctions
- Tableaux multidimensionnels, `struct`, `union`, `enum`, pointeurs sur les fonctions
- Gestion complète de `typedef`
- Toutes les instructions de contrôle, y compris les `goto`
- Tables des symboles séparées pour les étiquettes de `struct`, `union`, et `enum`, les champs de structures, les étiquettes de `goto`
- Initialisations statiques et dynamiques autorisées
- Typage pleinement conforme à la norme ANSI pour presque tous les opérateurs

Le Compilateur C - 3

Détails d'implémentation:

- Il n'y a pas de représentation intermédiaire pour le code autre que celle des expressions. Le compilateur pourrait être vu comme un premier pas vers un générateur de code intermédiaire

- Toutes les tables de symboles sont gérées avec des tables de hachage

- Pratiquement toutes les données sont stockées dans des listes. Cela permet de simplifier la gestion de la mémoire et évite d'abord un trop grand nombre de structures à définir de façon explicite.

- Le modèle de pile est le même que celui du C standard. On utilise un lien dynamique et on sauve aussi la taille des arguments passés en paramètre.

Le Compilateur C - 4

La syntaxe des déclarations impose de construire une représentation intermédiaire pour les types. On utilise pratiquement la même représentation pour le type définitif.

Types:

```
type :==  (base_type)
        | (TYPE_POINTER) + type
        | (TYPE_ARRAY dim) + type
        | (TYPE_STRUCT sym) | (TYPE_UNION sym) | (TYPE_ENUM sym)
        | (TYPE_FUNC func_type var_list) + type

base_type :==  TYPE_CHAR | TYPE_UCHAR
             | TYPE_SHORT | TYPE_USHORT
             | TYPE_INT | TYPE_UINT

func_type :== FUNC_ELLIPSIS | FUNC_OLD | FUNC_NEW

var_list :== var1 + ... + varN

var :== ( (nom) var_storage type var_init )

var_storage :==  STORAGE_DEFAULT | STORAGE_AUTO | STORAGE_REGISTER
               | STORAGE_STATIC | STORAGE_EXTERN

var_init :== (INIT_EXPR expr) | (INIT_LIST var_init1 ... var_initN)
```

Le Compilateur C - 5

Table des symboles:

```
var_location :== VAR_STACK | VAR_DATA

sym_var :== ( SYM_VAR var_storage type (var_location var_offset))

sym_field_struct :== (type offset)

sym_typedef :== (SYM_TYPEDEF type)

sym_struct :==  (TYPE_STRUCT -1)  /* si non défini */
              | (TYPE_STRUCT symbol_table size align)

sym_enum_const :== (SYM_ENUM_CONST val)
```

Expressions:

```
expr :== (type tag expr1 ... exprN )

expr_ident :== (type EXPR_IDENT sym)

expr_call :== (type EXPR_CALL expr_func n param1 paramN)

expr_int :== (type EXPR_INT n)

expr_str :== (type EXPR_STR str1 ... strN)

expr_cast :== (type EXPR_CAST expr)

etc...
```

L'Assembleur

- Génération de code et données dans 2 segments `.text` et `.data`
- Données: `.byte` *num*, `.short` *num*, `.int` *expr*,`.align` *num*, `.zero` *num*
- Etiquettes: `.equ` *sym*,*num* , *sym*:, `.globl` *sym*
- Expressions de la forme: *sym*, *num*, *sym* +*num*
- Début d'un nouveau module: `.module`

On utilise pour plus de simplicité l'assembleur comme éditeur de liens. Un symbole peut être externe (non défini pour le moment), global (grace à la directive `.globl`) ou privé.
La directive `.module` efface tous les symboles privés.

L'exécutable généré contient une table de relocation car on veut que les pointeurs de la machine virtuelle soient compatibles avec les pointeurs réels.

La Machine Virtuelle

C'est une machine à pile. Le fichier `fbvmspec.h` donne des informations vitales sur l'architecture (*endianité*, alignement, taille des types de base, modèle de pile). On a supposé ici pour simplifier qu'un pointeur avait la même taille qu'un `int`.

Les instructions:

```
lecture mém.    : ld_b,ld_ub,ld_w,ld_uw,ld_i
écriture mém.   : st_b,st_w,st_i
arithmétiques   : add_i,sub_i,mul_i,mul_ui,div_i,
                  div_ui,mod_i,mod_ui,neg_i
comparaisons    : cmplt_i,cmple_i,cmpge_i,cmpgt_i,
                  cmpeq_i,cmpne_i,
                  cmplt_ui,cmple_ui,cmpge_ui,cmpgt_ui
logiques        : and_i,or_i,xor_i,not_i,
                  shl_i,shr_i,shr_ui
conversions     : cvt_i_b,cvt_i_ub,cvt_i_w,cvt_i_uw
constantes      : li_i n,libp_i n
sauts           : jeq_i n,jne_i n,switch_i,jmp n
fonctions       : jsr n,rts
gestion pile    : dup,pop,addsp n
systeme         : libcall n
```

Résultats - Conclusion

- Le compilateur se compile lui-même, et la version compilée se recompile elle-même en donnant le même code. La vitesse reste raisonnable. Pour en arriver là, il a fallu non seulement réaliser le compilateur, mais aussi fabriquer tout un environnement de compilation et d'exécution.

- Grande complexité du typage en C. Finalement, un compilateur de langage plus évolué doit être paradoxalement plus simple.

- Le choix d'une machine à pile n'a pas été optimal. Nous pensions ne pas avoir le temps de générer du code intermédiaire puis un code assembleur pour une machine à registres. Finalement, avec le recul, si un tel choix avait été fait dès le début, nous aurions pu mener à bien cette tâche. Cela aurait permis de plus travailler sur l'optimisation du code généré.

- Il faut donc plutôt voir ce projet comme un générateur de code intermédiaire pour le C et non comme un compilateur complet.