Rapport de Projet de Compilation
Partie Backend

Rémi DESTIGNY - Isaline LAURENT

23 janvier 2012

Introduction

Un compilateur est généralement composé de deux parties, appelées FrontEnd et BackEnd. A partir d'un certain langage, ici proche du C, le FrontEnd retranscrit le code d'entrée en une forme intermédiaire. Ensuite, le BackEnd utilise ce code intermédiaire pour à nouveau le traduire en langage Assembleur.

Notre partie du projet concerne le BackEnd. Le code que nous devons traiter a donc plusieurs caractéristiques :

- C'est un code deux adresses. Les opérations se font donc avec des instructions comme += ou -=.
- Les seules boucles autorisées sont effectuées à partir de goto et de label. Les structures de plus haut niveau telles que while, sont prohibées.
- Les structures conditionnelles sont simplifiées. Il n'y a pas de clause else.
- Comme en C, on pourra traiter des bloc d'instructions et des fonctions.

Notre but principal est de traduire le code donné en assembleur x86 pour machine Intel en 32 bits. La vérification du type des variables est traité dans le FrontEnd et ne nous concerne pas.

1 Table de symboles

Avant de rentrer dans le vif du sujet, il est important de spécifier comment chaque identifiant est géré.
Deux choses nous ont poussés à implémenter une telle structure :

- Dans un premier temps, les variables étant stockées dans la pile, on va avoir besoin de connaître leur position pour pouvoir y accéder. A chaque symbole sera donc associé un offset.
- Ensuite, lorsque l'on a plusieurs blocs d'instructions, les variables ne sont pas forcément visibles dans chaque bloc. Il va donc être nécessaire d'établir un arbre de table de symboles, où chaque nœud est propre à un bloc, et n'a de visibilité que dans sa propre liste de symboles et dans celles de la branche qui relie à la table racine.

Structures

Pour gérer cette structure, nous utilisons trois structures en terme de programmation C.

Tout d'abord, une structure pour décrire chaque nœud de l'arbre, c'est à dire chaque table.

```
struct symbolTableTreeNode
{
  struct symbolTableTreeNodeList* sons;
  struct symbolTableTreeNode* father;
  struct symbolTableIdentifierList* identifierList;
  char* functionName;
  struct string* code;
  int currentOffset;
  int parameterSize;
};
```

Elle est composée d'un champ father, et d'un champ sons pour respecter la navigation dans l'arbre. Le champ identifierList correspond aux symboles en eux-même. Il s'agit d'une liste. functionName et parameterSize sont des champs qui décrivent la fonction, c'est à dire son nom, et la taille de ses paramètres. code permet de gérer l'affichage du code asm. currentOffset correspond au FramePointer et permet de définir le début de la fonction dans la pile, et donc sa visibilité. Le champ code est relativement important pour comprendre comment nous gérons la génération du code. Chaque table correspond à un bloc d'instructions. Parfois, nous le verrons plus tard dans ce rapport, il est nécessaire de connaître l'ensemble des instructions avant d'écrire le code. Dans un premier temps, les instructions générées sont donc stockées dans le champ code. Cela nous permet de les imprimer seulement au moment où toutes les informations nécessaires sont connues.

La structure qui suit décrit un étage de l'arbre. Il s'agit d'une liste de tables des symboles situés à la même distance de la racine.

```
 struct symbolTableTreeNodeList
{
  struct symbolTableTreeNodeList* next;
  struct symbolTableTreeNode* data;
};
```

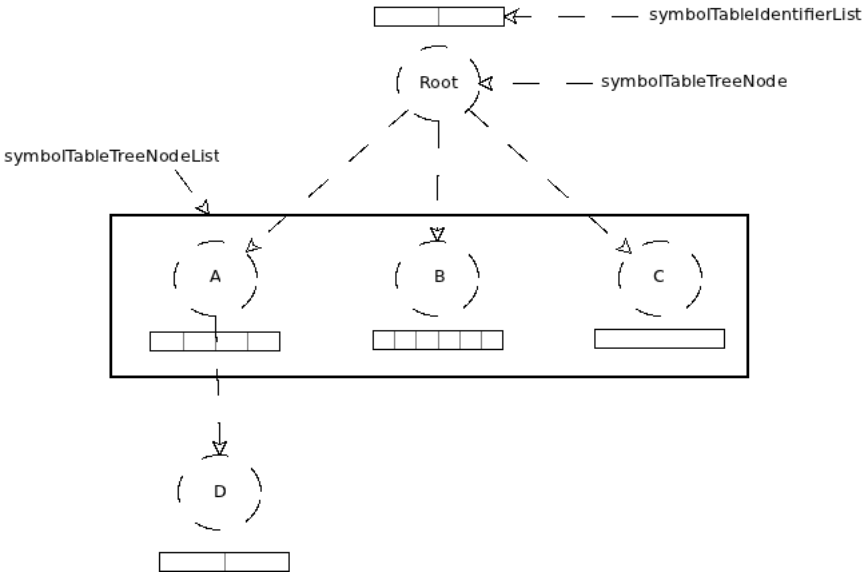
Enfin, voici la liste des symboles en elle-même.

```
struct symbolTableIdentifierList
{
  struct symbolTableIdentifierList* next;
  char* name;
  int type;
```

```
  int offset;
  int size;
  int nbArrayDimension;
  int dimensionSizes[256];
};
```

Elle contient bien évidemment un pointeur vers l'élément suivant. Elle contient de plus toutes les informations concernant un symbole, c'est à dire son nom, son type, son offset (sa position dans la pile), et sa taille. Dans le cas d'un tableau, elle possède aussi sa dimension, et la taille de chacun des vecteurs qui le composent.

Graphiquement, nous avons donc le système qui suit :



Ici, Root a pour father NULL, et pour sons A, B et C. Les tables B et C ont toutes pour père Root, et pas de fils. A a pour père father, et pour fils D. Cette dernière a donc pour père A, et pas de fils. Root a deux éléments dans sa table des symboles, A quatre, B six, C un seul et D, deux.

2 Un exemple de traitement : l'instruction IF

Le traitement d'une instruction IF est le résultat de la règle "selection_statement".

Dans un premier temps, il faut traiter l'instruction de comparaison. Si celle-ci n'est pas respectée, le programme sera directement envoyé vers un label que nous devons alors créer, et qui lui permettra de ne pas exécuter le code conditionnel. Le passage par la règle comparison_expression va écrire la comparaison à exécuter. Prenons comme exemple le code qui suit :

```
a = 4;
b = 5;
if (a < b) {
int c;
c += a;
}
```

En admettant que a soit disponible à -8(%ebp), et b à -16(%ebp), le code produit par la comparaison sera :

```
movl -8(%ebp), %ebx
cmpl %ebx, -16(%ebp)
```

Et le résultat remonté à selection_statement sera "jge". Selection_statement étant responsable de la gestion du label, il a besoin de cette information pour orienter l'exécution du code. Il est à noter qu'à première vue, on aurait pu simplement écrire :

```
cmpl -8(%ebp), -16(%ebp)
```

Cependant, la plupart des instructions ne peuvent pas gérer deux références à la fois.

Au terme de l'exécution de cette instruction, les drapeaux du processeur sont modifiés en conséquence et transmettront ainsi le résultat de la comparaison à l'instruction jump qui la suit.

De retour dans la règle selection_statement, la première instruction stockée est le résultat de comparison_statement (l'instruction jump) suivi du label créé, c'est à dire dans notre exemple (en admettant que le label est nommé "label") :

```
jge label
```

Vient ensuite le corps de l'instruction IF, qui est une suite d'instructions. Une fois cette portion de code traitée, on sort du IF, et il faut donc préciser que la suite est accessible via le label précédemment créé.

Finalement, le code produit est le suivant :

```
movl -8(%ebp), %ebx
cmpl %ebx, -16(%ebp)
jge label
...
label :
...
```

Dans le fichier yacc, le code est géré de cette façon :

```
selection_statement
 : IF '(' comparison_expression ')'
 { /* Conditional statement*/
   // Creation of a new IF label
```

```
    char* lbl = newLabel("IF");
    symbolTableCurrentNode->code =
     addString(symbolTableCurrentNode->code,"%s %s\n", $5, lbl);
    push(lbl,labelPile);
}
statement
{
  /* End of the statement */
  char* lbl = pop(labelPile);
  // Write label name after the statement
  symbolTableCurrentNode->code =
   addString(symbolTableCurrentNode->code,"%s:\n",lbl);
  }
;
```

3 Les fonctions

En assembleur, les fonctions ne sont qu'un label. Cependant, elles ont besoin d'un traitement particulier, pour gérer la pile et les paramètres par exemple. Deux parties sont à distinguer : l'appel et la définition.

L'appel

Les fonctions sont appelées via l'instruction "call". Si la fonction a besoin d'arguments, ils lui sont passés par la pile avec des instructions "pushl" Le retour de la fonction est par convention stocké dans le registre %eax. Il faut donc récupérer ce résultat à la fin de l'exécution de la fonction à l'aide d'une instruction "movl".

Par exemple, pour une fonction à deux paramètres, le code produit sera similaire à celui-ci :

```
pushl  -8(%ebp)
pushl  -4(%ebp)
call  bar
movl  %eax, -12(%ebp)
```

La définition

La définition d'une fonction se fait via la règle qui suit :

```
function_definition
: type_name  declarator compound_statement
```

La table des symboles est créée lors de la déclaration de la fonction. La définition d'une fonction passant par la règle "declarator", si celle ci n'a pas été déclarée précedemment, la table y est créée. Ainsi, on est sûr de n'avoir qu'une seule et unique table pour la fonction. Lors de la déclaration, les paramètres sont ajoutés dans la table.

Suite à la règle "declarator", on peut donc récupérer la table propre à la fonction définie. Nous avons pour cela implémenté une fonction getFunctionNode qui va chercher la table des symboles de la fonction correspondante.

La corps de la fonction est ensuite traitée par la règle "compound_statement", dont le résultat est stocké dans le code de la table des symboles courante.

Une fois le corps traité, les informations nécessaires à l'initialisation de la pile sont connues.

```
.globl bar
  .type  bar, @function
bar:
  pushl  %ebp
  movl   %esp, %ebp
  subl   $16, %esp
  movl   -8(%ebp), %ebx
  addl   %ebx, -12(%ebp)
  movl   -4(%ebp), %ebx
  addl   %ebx, -12(%ebp)
  movl   %ebx, %eax
  leave
  ret
```

Dans le code précédent, il y a deux paramètres de type int, et une variable est déclarée. Il faut donc réserver suffisamment de place dans la pile après avoir initialisé le Stack Pointer. C'est ce dont se chargent les trois premières lignes du label.

1. pushl %ebp place la valeur du Frame Base Pointer sur la pile. Cela permettra de récupérer cette information à la fin de l'exécution de la fonction.
2. movl %esp, %ebp place la valeur du Stack Pointer en tant que nouveau Frame Base Pointer pour réduire la visibilité de la fonction.
3. subl $16, %esp réserve 3 + 1 cases mémoires pour les paramètres et variables.

A la fin du label, cette fonction renvoyant un résultat, il faut le placer dans %eax. Les instructions leave et ret permettent de remettre la pile dans un état cohérent après l'exécution de la fonction.