Travaux Dirigés Compilation: TP2

Informatique 2ème année. ENSEIRB 2013/2014444

——

Le projet de cette année est disponible en ligne.

Pour la génération de code, le code assembleur LLVM est documenté en ligne également.

Génération de code

Le code assembleur LLVM est structuré en fonctions (comme en C), utilise des variables locales préfixées par % et des variables globales préfixées par @. Les opérations et les fonctions sont typées explicitement. Les types peuvent être `i32` pour un entier, `float`, `i8` pour un char, et les pointeurs utilisent `*` comme en C.

En assembleur LLVM, une variable ne peut être définie que par une seule instruction par fonction.

Un exemple de code assembleur :

```
; un commentaire
define i32 @mafonction(i32 %arg) {    ; mafonction est une fonction (globale)
  %x = mul i32 %arg, 2   ; multiplication entière, retournant un i32
  %y = add i32 %x , 32   ; addition entière
  %z = sub i32 %y , %x   ; soustraction entière
  %a = sdiv i32 %z, 4    ; division entière
  ret i32 %a
}
define i32 @main() {    ; main est une variable globale
  %retval = call i32 @mafonction(i32 42)
  ret i32 0
}
```

Pour compiler, si le fichier assembleur s'appelle `fichier.ll`, alors il faut taper :

```
llvm-as fichier.ll -f -o fichier.bc
llvm-ld -o=fichier -native fichier.bc
```

▸**Exercice 1.** *Calcul d'expressions On reprend la grammaire d'expression très proche de la feuille de TP1 (le fichier lex peut être repris du TP1) :*

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%token ID N
%%

S
: E
;
;
E
: T '+' E
| T '-' E
| T
;
T
: F '*' T
| F
;
F
: N
```

```
| '(' E ')'
;
%%
int main (int argc, char *argv[]) {
    yyparse ();
    return 0;
}
```

1. *Ecrire une fonction* `i32 calcule(i32 %x)` *en assembleur LLVM qui prend un paramètre entier* $\%x$ *et retourne* $\%x * \%x - 10$. *Pour lancer cette fonction et afficher le résultat, on utilisera le petit code à mettre dans le même fichier :*

```
@str = constant [7 x i8] c"=> %d\0A\00"
declare i32 @printf(i8*, ...)

define i32 @main() {
%x = call i32 @calcule(i32 32)
call i32 (i8*, ...)* @printf(i8* getelementptr ([7  x i8]* @str, i32 0, i32 0),
i32 %x)
ret i32 0
}
```

2. *Modifier les actions sémantiques pour génerer en assembleur LLVM une fonction* `i32 calcule(i32 %x)` *qui retourne la dernière variable entière calculée.*

▸**Exercice 2.** *Calcul d'expressions avec variables On reprend cette fois la grammaire d'expression de la feuille de TP1 (le fichier lex peut être repris du TP1) :*

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%token ID N

%%

S
: I S
| I
;
I
: ID '=' E ';'
;
E
: T '+' E
| T '-' E
| T
;
T
: F '*' T
| F
;
F
: N
| ID
| '(' E ')'
;
%%
int main (int argc, char *argv[]) {
    yyparse ();
    return 0;
}
```

En assembleur LLVM, on peut déclarer de nouvelles variables locales sur la pile, de type pointeur sur entier, avec :

```
%x = alloca i32
```

`%x` est alors une adresse sur la pile. On peut accéder à son contenu par :

```
store i32 123, i32* %x   ; stocke 123 dans x
%val = load i32*  %x     ; lit x
```

Ces variables sont automatiquement desallouées à la sortie de la fonction.

1. *Ecrire une fonction `calcule(%x)` qui calcule*

   ```
   a = 3 * x
   y = 5 + a
   z = y * a
   ```

   et retourne z, en stockant a,y et z sur la pile.
2. *Donner les actions sémantiques pour générer le code de la fonction `calcule(%x)` à partir des expressions. La fonction retournera la denière expression calculée. Tous les identificateurs devront correspondre à des variables sur la pile et seront alloués au moment où on les définit.*
3. *Afin d'éviter d'allouer plusieurs fois sur la pile la même variable, ajouter une table des symboles au code précédent. On utilisera également la table pour trouver les cas où les variables sont utilisées avant d'être définies. On générera alors une erreur.*