TP2 - Compilateur VSL+ - OCaml/LLVM

1 Cadre du TP

Au cours des prochaines séances de TP Compilation vous aurez à réaliser, à l'aide de OCaml et des librairies LLVM, un compilateur du langage VSL+. Une description informelle du langage VSL+ vous est fournie sur la feuille ci-jointe. Le code produit sera du code 3 adresses que vous avez commencé à découvrir en cours et TD.

Les analyseurs lexical et syntaxique de VSL+ vous sont fournis (fichier `parser.ml`). Ceux-ci produisent à partir du texte d'un programme VSL+ sa représentation sous forme d'un AST (défini dans le fichier `ast.ml`).

Votre travail va consister à réaliser la vérification de type et la génération de code à partir d'un AST. Concrètement, cela va prendre une forme similaire à une grammaire attribuée, mais en parcourant un AST plutôt qu'un *stream* de tokens. Vous aurez une fonction pour chaque type de construction (ex., expression, instruction, programme), définie par un filtrage sur les différentes constructions possible pour ce type de construction (ex., addition, soustraction, etc. pour les expressions). Pour chaque construction, vous devrez effectuer des vérifications de type et des appels aux librairies LLVM pour la génération de code.

Plus tard, nous vous donnerons les moyens de produire du code exécutable à partir du code 3 adresses, en utilisant LLVM comme une face arrière.

2 Travail demandé

Dans l'esprit du développement agile, nous vous proposons de développer votre compilateur de façon itérative avec des cycles courts de réflexion-codage-tests. Chaque cycle devrait faire moins de une séance, afin d'éviter une accumulation de bugs ingérable. L'idée est d'avoir une génération de code opérationnelle à la fin de chaque cycle pour des fragments du langage VSL+ de plus en plus grands.

Nous vous fournissons une version 1 (fichier `codegen.ml`) du générateur de code qui couvre uniquement les constantes et l'addition des expressions arithmétiques (expressions simples). Votre première tâche est de lire, comprendre, compiler et tester cette version. La section 4 fournit quelques explications concernant LLVM. Il vous revient de produire les fichiers de test (programmes réduits à des expressions simples).

La commande pour produire un exécutable de votre générateur de code est :

```
./build.sh
```

L'exécutable produit, `main.native`, prend un fichier VSL+ en paramètre (ou bien lit du VSL+ sur l'entrée standard) et affiche le code LLVM produit, d'abord sous forme brute, puis sous forme optimisée.

Vous allez ensuite ajouter une à une les différentes constructions de VSL+, en passant des expressions aux instructions, puis des instructions aux programmes. Nous vous suggérons de traiter dans l'ordre :

- Les expressions simples
- L'instruction d'affectation
- La gestion des blocs
- La déclaration des variables
- Les expressions avec variables
- Les instructions de contrôle `if`, `while` et la séquence
- La définition et l'appel de fonctions (avec les prototypes)
- Les fonctions de la bibliothèque : `PRINT` et `READ`
- La gestion des tableaux (déclaration, expression, affectation et lecture)

Pour chaque extension, vous devrez :

1. identifier les structures pertinentes de l'AST,
2. compléter le générateur de code pour ces structures, sans oublier la vérification de type,
3. produire les cas de tests utiles et vérifier que le code généré est correct.

Vous devrez adapter les exemples de programmes VSL+ fournis, et la fonction `main` du module `main.ml`, en fonction de votre niveau d'avancement.

3 Travail à rendre

Vous devez rendre par binôme un rapport plus les sources de votre compilateur début décembre (la date précise sera communiquée ultérieurement). Cependant, en raison de la longeur de ce TP (6 séances), il vous est également demandé de rendre à la fin de la 3ème séance une version couvrant toutes les instructions et expressions sauf les appels et retours de fonctions.

4 LLVM

LLVM (Low-Level Virtual Machine) [1] est une infrastructure facilitant le développement de compilateurs en facilitant la réutilisation de modules et d'outils. Vous allez utiliser les librairies OCaml du noyau LLVM qui permettent l'optimisation et la génération de code cible pour la plupart des architectures, et ce à partir d'une même représentation intermédiaire (LLVM IR [2]). C'est cette représentation intermédiaire que vous allez produire. Le fichier fourni `codegen.ml` donne l'exemple de la génération pour l'addition. Un tutoriel assez complet couvrant les principaux aspects des langages impératifs est disponible à l'URL `http://llvm.org/docs/tutorial/OCamlLangImpl1.html`. Ce sera une source importante d'information pour vous.

Voici en quelques phrases les principaux concepts de LLVM. Un *context* est un contenant principal pour les données globales et un *module* est un contenant principal pour le code intermédiaire. Un *builder* est utilisé pour générer du code intermédiaire. Celui-ci a une *position* courante représentant le point d'insertion de la prochaine instruction. Un *module* contient des définitions de fonctions. Une

1. `http://llvm.org`
2. Manuel de référence : `http://llvm.org/docs/LangRef.html`

fonction est constitué d'un ensemble de *blocks* (blocs de base). Chaque bloc doit être terminé par une instruction de saut ou de retour de fonction. Les *blocks* contiennent les instructions élémentaires du code intermédiaire. Le type `Llvm.llvalue` joue un rôle central et représente à la fois les constantes, les variables globales, les fonctions et même les instructions. Chaque *llvalue* a un type représenté par le type `Llvm.lltype`. On trouve les types élémentaires et les types composés (ex., fonctions, tableaux). Tous les types et toutes les fonctions de LLVM dont vous avez besoin sont disponibles dans le module OCaml `Llvm`, dont la documentation est accessible sur le share.

5 Exemple

Des exemples de programmes VSL+ sont disponibles sur le share. Le programme suivant calcule et affiche la fonction factorielle jusqu'à 10.

```
PROTO INT fact(k)
FUNC VOID main()
{
  INT i, t[11]
  i := 0

  WHILE 11 -i
  DO
   {
     t[i] := fact(i)
     i := i+1
   }
  DONE
  i := 0
  WHILE 11 -i
  DO
   {
     PRINT "f(", i, ") = ", t[i], "\n"
     i := i+1
   }
  DONE
}
FUNC INT fact(n)
{
  IF n
  THEN
    RETURN n * fact(n-1)
  ELSE
    RETURN 1
  FI
}
```

Et voici une traduction en code intermédiaire LLVM.

```
; ModuleID = 'main'

@"%s" = global [3 x i8] c"%s\00"
@"%d" = global [3 x i8] c"%d\00"
@"f(" = global [3 x i8] c"f(\00"
@") = " = global [5 x i8] c") = \00"
@"\0A" = global [2 x i8] c"\0A\00"

declare i32 @printf(i8* nocapture, ...) nounwind

declare i32 @scanf(i8* nocapture, ...) nounwind

define i32 @fact(i32 %n) {
entry:
  %return = alloca i32
  %n1 = alloca i32
  store i32 %n, i32* %n1
  %n2 = load i32* %n1
  %icmp = icmp ne i32 %n2, 0
  br i1 %icmp, label %then, label %else

then:                                             ; preds = %entry
  %n3 = load i32* %n1
  %n4 = load i32* %n1
  %minus = sub i32 %n4, 1
  %ecall = call i32 @fact(i32 %minus)
  %mul = mul i32 %n3, %ecall
  store i32 %mul, i32* %return
  br label %fi

else:                                             ; preds = %entry
  store i32 1, i32* %return
  br label %fi

fi:                                               ; preds = %else, %then
  %return5 = load i32* %return
  ret i32 %return5
}

define void @main() {
entry:
  %t = alloca i32, i32 11
  %i = alloca i32
```

```
  store i32 0, i32* %i
  br label %loop

loop:                                             ; preds = %body, %entry
  %i1 = load i32* %i
  %minus = sub i32 11, %i1
  %icmp = icmp ne i32 %minus, 0
  br i1 %icmp, label %body, label %after

body:                                             ; preds = %loop
  %i2 = load i32* %i
  %ecall = call i32 @fact(i32 %i2)
  %i3 = load i32* %i
  %"t[i3]" = getelementptr i32* %t, i32 %i3
  store i32 %ecall, i32* %"t[i3]"
  %i4 = load i32* %i
  %plus = add i32 %i4, 1
  store i32 %plus, i32* %i
  br label %loop

after:                                            ; preds = %loop
  store i32 0, i32* %i
  br label %loop5

loop5:                                            ; preds = %body6, %after
  %i8 = load i32* %i
  %minus9 = sub i32 11, %i8
  %icmp10 = icmp ne i32 %minus9, 0
  br i1 %icmp10, label %body6, label %after7

body6:                                            ; preds = %loop5
  %print_text = call i32 (i8*, ...)* @printf(
       i8* getelementptr inbounds ([3 x i8]* @"%s", i32 0, i32 0),
       i8* getelementptr inbounds ([3 x i8]* @"f(", i32 0, i32 0))
  %i11 = load i32* %i
  %print_expr = call i32 (i8*, ...)* @printf(
       i8* getelementptr inbounds ([3 x i8]* @"%d", i32 0, i32 0),
       i32 %i11)
  %print_text12 = call i32 (i8*, ...)* @printf(
       i8* getelementptr inbounds ([3 x i8]* @"%s", i32 0, i32 0),
       i8* getelementptr inbounds ([5 x i8]* @") = ", i32 0, i32 0))
  %i13 = load i32* %i
  %gep_arrayelt = getelementptr i32* %t, i32 %i13
  %arrayelt = load i32* %gep_arrayelt
  %print_expr14 = call i32 (i8*, ...)* @printf(
```

```
       i8* getelementptr inbounds ([3 x i8]* @"%d", i32 0, i32 0),
       i32 %arrayelt)
  %print_text15 = call i32 (i8*, ...)* @printf(
       i8* getelementptr inbounds ([3 x i8]* @"%s", i32 0, i32 0),
       i8* getelementptr inbounds ([2 x i8]* @"\0A", i32 0, i32 0))
  %i16 = load i32* %i
  %plus17 = add i32 %i16, 1
  store i32 %plus17, i32* %i
  br label %loop5

after7:                                           ; preds = %loop5
  ret void
}
```