TP2 - Compilateur VSL+

1 Cadre du TP

Au cours des prochaines séances de TP Compilation vous aurez à réaliser, à l'aide de OCaml ou Java et 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.

Votre travail va consister à réaliser la génération de l'AST à partir d'un fichier source VSL+, puis réaliser la vérification de type et la génération de code à partir d'un AST.

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-programmation-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 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).

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.

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 longueur de ce TP (6 séances), il vous est également demandé de rendre à la fin de la 3^{e} 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 générer une représentation intermédiaire [2] (IR). Votre compilateur agit donc en *front-end*. Les outils LLVM s'occupent ensuite de l'allocation de registres, de la génération de code machine, de l'optimisation, etc (*middle-end* et *back-end*).

Pour bien comprendre l'IR, lisez au moins ces parties de sa documentation :

— Abstract et Introduction
— Indentifiers
— Module Structure
— Global Variables
— Functions
— Type System
— Les instructions ret, br, add (et autres opérations binaires), call, getelementptr, icmp, load, store, alloca

Lisez attentivement les explications générales et les détails sur les instructions. Cependant, dans notre cas, nous conserverons une utilisation très simple de ces instructions. Basez également votre compréhension sur les exemples qui donnent les cas d'utilisation les plus simples des instructions.

Il est inutile de fournir une représentation OCaml ou Java *complète* de l'IR. Restreignez-vous au sous-ensemble que vous utilisez.

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

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 n, i, t[11]
  PRINT "Input n between 0 and 11:\n"
  READ n
  i := 0

  WHILE n-i
  DO
   {
     t[i] := fact(i)
     i := i+1
   }
  DONE
  i := 0
  WHILE n-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 (avant optimisation).

```
; ModuleID = 'main'
```

```
@"%s" = global [3 x i8] c"%s\00"
@"%d" = global [3 x i8] c"%d\00"
@"Input n between 0 and 11:\0A" = global [27 x i8] c"Input n between 0 and 11:\0A\00"
@"f(" = global [3 x i8] c"f(\00"
@") = " = global [5 x i8] c") = \00"
@"\0A" = global [2 x i8] c"\0A\00"

; Function Attrs: nounwind
declare i32 @printf(i8* nocapture, ...) #0

; Function Attrs: nounwind
declare i32 @scanf(i8* nocapture, ...) #0

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
  %n = alloca i32
  %print_text = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @"%s", i32
  %read = call i32 (i8*, ...)* @scanf(i8* getelementptr inbounds ([3 x i8]* @"%d", i32 0, i32
  store i32 0, i32* %i
  br label %loop

loop:                                             ; preds = %body, %entry
  %n1 = load i32* %n
  %i2 = load i32* %i
  %minus = sub i32 %n1, %i2
  %icmp = icmp ne i32 %minus, 0
  br i1 %icmp, label %body, label %after

body:                                             ; preds = %loop
  %i3 = load i32* %i
  %ecall = call i32 @fact(i32 %i3)
  %i4 = load i32* %i
  %"t[i4]" = getelementptr i32* %t, i32 %i4
  store i32 %ecall, i32* %"t[i4]"
  %i5 = load i32* %i
  %plus = add i32 %i5, 1
  store i32 %plus, i32* %i
  br label %loop

after:                                            ; preds = %loop
  store i32 0, i32* %i
  br label %loop6

loop6:                                            ; preds = %body7, %after
  %n9 = load i32* %n
  %i10 = load i32* %i
  %minus11 = sub i32 %n9, %i10
  %icmp12 = icmp ne i32 %minus11, 0
  br i1 %icmp12, label %body7, label %after8

body7:                                            ; preds = %loop6
  %print_text13 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @"%s", i
  %i14 = load i32* %i
  %print_expr = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @"%d", i32
  %print_text15 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @"%s", i
```

```
  %i16 = load i32* %i
  %gep_arrayelt = getelementptr i32* %t, i32 %i16
  %arrayelt = load i32* %gep_arrayelt
  %print_expr17 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @"%d", i
  %print_text18 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @"%s", i
  %i19 = load i32* %i
  %plus20 = add i32 %i19, 1
  store i32 %plus20, i32* %i
  br label %loop6

after8:                                           ; preds = %loop6
  ret void
}

attributes #0 = { nounwind }
```

6 Environnement

Les environnements de développement sont identiques à ceux présentés en TP1.

Le script `compile` permet de compiler un fichier `.vsl` en fichier exécutable. Il prend en argument ce fichier à compiler, appelle votre compilateur puis appelle le compilateur `clang`. Il génère donc deux fichiers : la représentation intermédiaire (`.ll`) et l'exécutable (du même nom que le fichier `.vsl`).