TP2 - Compilateur VSL+

1 Cadre du TP

Au cours des prochaines séances de TP PDS 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.

Vous pourrez produire du code exécutable à partir du code 3 adresses, en utilisant LLVM comme 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 6 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` et `while`
- 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'ASD
2. augmenter l'ASD
3. éventuellement, tester les modifications avec des AST écrits manuellement
4. augmenter la grammaire
5. compléter le générateur de code pour ces structures, sans oublier la vérification de type
6. 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 Conseils

Implémentez un *pretty-printer* basique pour chaque extension avant la génération de code. Ces fonctions sont plus simples à réaliser et permettent de vérifier que votre ASD a du sens et que l'analyse d'un programme source est correcte en comparant la source à la sortie du *pretty-printer*.

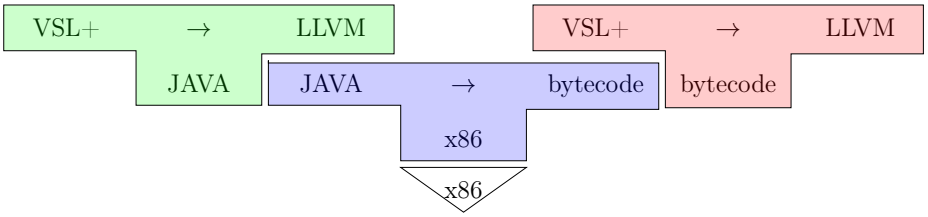
Testez votre code manuellement mais aussi automatiquement. Par exemple, en java, le fichier `src/test/java/TP2/SymbolTableTest.java` donne un exemple de tests unitaires basés sur le *framework* `JUnit`. Les tests unitaires sont exécutés à chaque compilation, ce qui permet de détecter une régression le plus tôt possible.

5 Vue d'ensemble

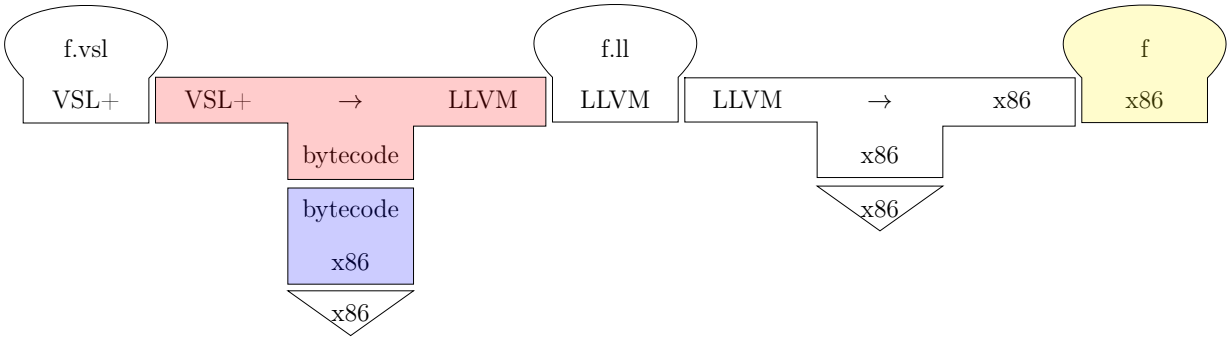
Les diagrammes en T suivants représentent différentes phases que vous aurez à exécuter durant le TP.

Le compilateur en vert est celui que vous écrivez. Les outils Java (compilateur et machine virtuelle) sont représentés en bleu. Le compilateur en blanc correspond à la face arrière de LLVM.

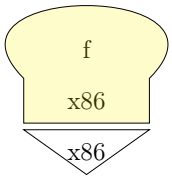
La compilation du compilateur VSL+ que vous écrivez est représentée par le diagramme en T ci dessous, pour la version Java. La compilation des grammaires (ANTLR vers Java) n'est pas représentée.



Les deux étapes de compilation d'un fichier VSL+ sont représentées par le diagramme en T suivant.



Enfin, l'exécution de l'exécutable obtenu `f` correspond à ce diagramme en T :



6 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 les parties ci-dessous de sa documentation :

— Abstract et Introduction
— Identifiers
— Module Structure
— Global Variables

1. `http://llvm.org`
2. Manuel de référence : `http://releases.llvm.org/3.9.1/docs/LangRef.html`

— 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.

7 Exemple

Des exemples de programmes VSL+ sont disponibles sur le share. Le programme suivant calcule et affiche la fonction factorielle jusqu'à un nombre choisi par l'utilisateur.

```
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).

```
; Target
target triple = "x86_64-unknown-linux-gnu"
; External declaration of printf and scanf functions
declare i32 @printf(i8* noalias nocapture, ...)
declare i32 @scanf(i8* noalias nocapture, ...)

@.fmt0 = global [27 x i8] c"Input n between 0 and 11:\0A\00"
@.fmt1 = global [3 x i8] c"%d\00"
@.fmt2 = global [12 x i8] c"f(%d) = %d\0A\00"

; Actual code begins
define i32 @fact(i32 %n){
  %n1 = alloca i32
  store i32 %n, i32* %n1
  %tmp21 = load i32, i32* %n1
  %tmp22 = icmp ne i32 %tmp21, 0
  br i1 %tmp22, label %then7, label %else8

then7:
  %tmp23 = load i32, i32* %n1
  %tmp24 = load i32, i32* %n1
  %tmp25 = sub i32 %tmp24, 1
  %tmp26 = call i32 @fact(i32 %tmp25)
  %tmp27 = mul i32 %tmp23, %tmp26
  ret i32 %tmp27
  br label %fi9

else8:
  ret i32 1
  br label %fi9

fi9:
  ret i32 0
}

define void @main(){
  %n = alloca i32
  %i = alloca i32
  %t = alloca [11 x i32]
  call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([27 x i8], [27 x i8
     ]* @.fmt0, i64 0, i64 0))
  call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]*
     @.fmt1, i64 0, i64 0), i32* %n)
```

```
  store i32 0, i32* %i
  br label %while1

while1:
  %tmp1 = load i32, i32* %n
  %tmp2 = load i32, i32* %i
  %tmp3 = sub i32 %tmp1, %tmp2
  %tmp4 = icmp ne i32 %tmp3, 0
  br i1 %tmp4, label %do2, label %done3

do2:
  %tmp5 = load i32, i32* %i
  %tmp6 = call i32 @fact(i32 %tmp5)
  %tmp7 = load i32, i32* %i
  %tmp8 = getelementptr [11 x i32], [11 x i32]* %t, i64 0, i32 %tmp7
  store i32 %tmp6, i32* %tmp8
  %tmp9 = load i32, i32* %i
  %tmp10 = add i32 %tmp9, 1
  store i32 %tmp10, i32* %i
  br label %while1

done3:
  store i32 0, i32* %i
  br label %while4

while4:
  %tmp11 = load i32, i32* %n
  %tmp12 = load i32, i32* %i
  %tmp13 = sub i32 %tmp11, %tmp12
  %tmp14 = icmp ne i32 %tmp13, 0
  br i1 %tmp14, label %do5, label %done6

do5:
  %tmp15 = load i32, i32* %i
  %tmp16 = load i32, i32* %i
  %tmp17 = getelementptr [11 x i32], [11 x i32]* %t, i64 0, i32 %tmp16
  %tmp18 = load i32, i32* %tmp17
  call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([12 x i8], [12 x i8
      ]* @.fmt2, i64 0, i64 0), i32 %tmp15, i32 %tmp18)
  %tmp19 = load i32, i32* %i
  %tmp20 = add i32 %tmp19, 1
  store i32 %tmp20, i32* %i
  br label %while4

done6:
```

```
  ret void
}
```

8 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`).

8.1 Utilisation d'Eclipse

8.1.1 Importation du projet

Les machines de l'ISTIC disposent de plusieurs versions d'Eclipse, que vous pouvez utiliser pour travailler sur ce TP. Il est recommandé de choisir Eclipse 2018.12, sur lequel le TP a été testé.

L'importation du projet se fait en utilisant la commande "Import..." du menu "File", puis en sélectionnant "Existing Gradle project" dans la section "Gradle".

Pour compiler les sources ANTLR, il est notamment nécessaire d'exécuter l'étape de *build* Gradle. Cette commande est accessible dans la vue "Gradle Tasks". Les éventuelles erreurs retournées par ANTLR sont alors accessibles dans la vue "Console".

8.1.2 Erreurs les plus communes

Erreurs de version de la JVM Des erreurs de version de la JVM se produisent lorsque le projet est utilisé avec des versions d'Eclipse différentes utilisant des JVM différentes (sur les machines de l'ISTIC ou entre les machines de l'ISTIC et une machine personnelle, par exemple). Pour résoudre ce problème, utilisez la version d'Eclipse avec laquelle vous avez initialement importé le projet (notamment si vous travaillez sur les machines de l'ISTIC).

Si vous choisissez de travailler en utilisant Git et Eclipse, il vous est recommandé d'ignorer les fichiers de projet Eclipse, afin d'éviter que ces erreurs se produisent après synchronisation [3].

Eclipse ne détecte pas VSLParser Il faut d'abord s'assurer que ANTLR est capable de générer correctement les analyseurs lexical et syntaxique, en compilant le projet en exécutant la tâche "build" de Gradle. Si, malgré la réussite de cette opération, Eclipse continue d'affcher des erreurs sur la classe VSLParser, il faut alors rafraîchir l'index d'Eclipse. Pour cela, dans le menu contextuel du projet TP2, il faut sélectionner "Gradle > Refresh Gradle project".

3. Un fichier d'exemple Gitignore pour Eclipse est disponible à l'adresse suivante : `https://github.com/github/gitignore/blob/master/Global/Eclipse.gitignore`