TP2 - Compilateur VSL+ - ANTLR/Java

1 Cadre du TP

Au cours des prochaines séances de TP Compilation vous aurez à réaliser, à l'aide de ANTLR, 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. Ceux-ci produisent à partir du texte d'un programme VSL+ sa représentation sous forme d'un AST ANTLR. Votre travail va consister à écrire un arpenteur d'arbre qui effectue la vérification de type et la génération de code 3 adresses pour chacune des constructions du langage VSL+. Des librairies vous sont fournies pour la représentation du code 3 adresses, des types et de la table des symboles, ainsi que pour la génération de variables temporaires et étiquettes de code. L'ensemble de ces éléments constituent la face avant de votre compilateur.

Dans la phase finale du TP (2 dernières séances) vous pourrez brancher votre face avant sur une face arrière fournie, à savoir un générateur de code pour la machine MIPS. La machine MIPS est émulée dans l'environnement NACHOS qui fournit un assembleur et un éditeur de liens. La production de code exécutable suit donc les enchainements classiques : production assembleur → code machine incomplet → édition de liens → code exécutable.

2 Documentation

Vous disposez des documentations et sources d'information suivantes :

— Le document ci-joint donne une description informelle de VSL+. La grammaire précise de VSL+ ainsi que la forme des AST générés est à découvrir dans la grammaire ANTLR de VSL+ : fichier `VSLParser.g`.

— Une description du code 3 adresses est donnée dans l'annexe A.

— Concernant les classes fournies pour la génération de code et la vérification de type, une documentation HTML est disponible sur le share. Cette documentation a été générée en faisant du reverse engineering sur le code à l'aide de l'outil BOUML. Vous pouvez accéder à cette documentation en ouvrant le fichier : `/share/m1info/COMP/TP2-antlr/docHTML/bouml/index-withframe.html` dans votre navigateur favori. Une documentation plus classique, générée par javadoc est disponible à partir de : `/share/m1info/COMP/TP2-antlr/docHTML/javadoc/index.html`. Des informations importantes sont également données dans ce sujet en section 6.

— Les instructions pour mettre en place la phase finale avec NACHOS vous sont données en annexe C.

— Enfin, un exemple complet de programme VSL+ et de sa traduction en code 3 adresses vous est donné dans l'annexe B.

3 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 `VSLTreeParser.g` dans `/share/m1info/COMP/TP2-antlr`) du générateur de code qui couvre uniquement les constantes et les opérateurs binaires des expressions arithmétiques (expressions simples). Votre première tâche est de lire, comprendre, compiler et tester cette version. La section 5 fournit toutes les explications nécessaire et va vous permettre de découvrir les classes fournies. 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. augmenter l'arpenteur d'arbre avec ces structures,
3. le compléter avec des actions pour la vérification de type et la génération de code,
4. produire les cas de tests utiles et vérifier que le code généré est correct.

Afin de vérifier le code 3 adresses produit, vous pouvez utiliser la fonction d'affichage : `print` de `Code3a.java`. Lors de votre réalisation, vous pouvez, par exemple, déplacer votre axiome sur le non-terminal `expression` ou `statement` dans le fichier `VslComp.java`.

4 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 sera 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.

5 Tutoriel de génération de codes 3 adresses

On décrit ici la méthode générale pour la génération de code 3 adresses qu'on illustre sur l'exemple des expressions (étiquette `PLUS`) :

```
expression : ^(PLUS expression expression)
```

La règle expression (1) produit (synthétise) un *code* permettant de l'évaluer, une *place* indiquant le nom de la variable 3 adresses dans laquelle le résultat de l'évaluation est stocké et le type de l'expression et (2) a besoin (hérite) de la table des symboles. ANTLR ne permet de synthétiser qu'un objet Java, on regroupe (encapsule) donc code, place et type sous formes d'attributs dans un objet de la classe `ExpAttribute` (aller voir le code dans le fichier `ExpAttribute.java`). La signature de la règle `expression` est donc :

```
expression [SymbolTable ts] returns [ExpAttribute e] :
        ^(PLUS expression[ts] expression[ts])
```

Pour générer le code évaluant l'expression considérée il faut *récupérer* le code et la place des opérandes ainsi que leurs types pour la vérification sémantique de type. On utilise deux variables `e1` et `e2`. On devra alors :

- vérifier que les types sont compatibles avec l'opérateur et retourner le type du résultat. Dans le cas de `+`, il faut vérifier que les opérandes sont de type `INT` et le résultat sera de type `INT`. Le type `INT` est un type constant défini dans la classe `Type` (vous pouvez voir les types prédéfinis dans cette classe, mais vous n'aurez pas besoin de la modifier ; quelques types plus complexes, comme les tableaux et types pour les fonctions, sont définis dans des sous-classes de `Type`). La classe `TypeCheck` est destinée à accueillir toutes les fonctions de vérification de types [1].

  Vous devrez écrire dans `TypeCheck` les méthodes de vérification de type.

  Par exemple, pour le cas des expressions arithmétiques sur les entiers, on écrit :

```
public static Type checkBinOp(Type t1, Type t2) {
    if (t1 == Type.INT && t2 == Type.INT)
        return Type.INT;
    else {
        // ... optionnel : émettre un message d'erreur pour l'utilisateur ...
        //
        return Type.ERROR;
    }
}
```

- Identifier comment construire le code à l'aide du code synthétisé par les sous-règles. La classe `Code3a` fournit des méthodes pour construire et assembler du code :

```
/** créé un nouvel objet code ne contenant aucune instruction **/
public Code3a()
/** créé un nouvel objet code ne contenant qu'une instruction : i **/
public Code3a(Inst3a i)
```

1. La documentation HTML indique des méthodes qui ne vous sont pas fournies ; vous pouvez considérer les noms et signatures comme des suggestions, mais vous n'avez pas besoin de les implémenter de la même façon.

```
/** ajoute le code c à la fin du code **/
public void append(Code3a c)
/** ajoute l'instruction i à la fin du code **/
public void append(Inst3a inst)
```

Les instructions sont construites à l'aide du constructeur de la classe `Inst3a` qui prend la constante de l'opération (voir la liste des opérations disponibles dans `Inst3a.java` et les 3 opérandes.
Pour l'opération `+`, il faut assembler dans cet ordre le code pour évaluer le premier opérande, le code pour évaluer le second et le code pour effectuer l'opération en lui spécifiant *où* trouver les opérandes et *où* placer le résultat. On définit une méthode dans la classe `Code3aGenerator` pour générer le code, elle pourra être réutilisée pour les autres opérateurs :

```
/**
 * génère le code pour effectuer l'opération op sur les opérandes e1 et e2
 * et placer le résultat à la place tmp
**/
public static Code3a genBinOp(Inst3a.TAC op, Operand3a tmp,
                  ExpAttribute e1, ExpAttribute e2){
   Code3a c = exp1.code;
   c.append(exp2.code);
   // ajoute l'instruction : allouer une variable 3 adresses tmp
   //voir le code de genVar
   c.append(genVar(tmp));
   // ajoute l'instruction : effectuer le calcul et placer le résultat dans tmp
   c.append(new Inst3a(op, tmp, exp1.place, exp2.place));
   return c;
}
```

La règle de l'expression pour l'étiquette `PLUS` devient donc :

```
expression [SymbolTable ts] returns [ExpAttribute exp] :
   ^(PLUS e1 = expression[ts]  e2 = expression[ts]) {
      Type t = TypeCheck.checkBinOp(e1.type, e2.type);
      VarSymbol tmp = SymbDistrib.newTemp();
      Code3a c = Code3aGenerator.genBinOp(Inst3a.TAC.ADD, tmp, e1, e2);

      exp = new ExpAttribute(t, c, tmp);
   }
```

On remarque que l'opérande décrivant une *place* est générée à l'aide de la méthode `newTemp` de la classe `SymbDistrib`. Aller découvrir les autres méthodes à votre disposition dans le fichier Java de cette classe.

6 Informations importantes

6.1 Table des symboles

Le langage VSL+ impose que chaque variable ou fonction soit déclarée. La portée d'une variable est le plus petit bloc englobant la déclaration. La structure utilisée est une pile de tables d'associations nom/variable, implémentée par la classe `SymbolTable` (aller voir le fichier `SymbolTable.java`). Quand on entre dans un nouveau bloc ou fonction on créé une nouvelle table et on l'empile. Lorsqu'on

en sort, on dépile la table au sommet de la pile car la portée des variables qu'elle contient se limite au bloc courant. Lorsqu'une variable est rencontrée, dans une expression par exemple, on retrouve les informations qui lui ont été associées à la déclaration (type, place, etc.) en recherchant dans toutes les tables, dans l'ordre de la pile, en s'arrêtant à la première qui a la clef.

```
/** création d'une table des symboles */
public SymbolTable()

/** empilement d'une nouvelle table à l'entrée d'un bloc */
public void enterScope()

/** dépilement d'une table à la sortie d'un bloc */
public void leaveScope()

/** renvoie la hauteur courante de la pile de tables */
public int getScope()

/** cherche une variable par son nom (en commençant par la table au sommet) */
public Operand3a lookup(String name)

/** définit une variable dans la table courante */
public void insert(String name, Operand3a t)
```

Allez voir le diagramme de classes "symbols" pour comprendre les différentes dérivations de la classe `Operand3a`. La classe `SymbDistrib.java` fournit des méthodes pour construire des `Operand3a` de temporaires (VarSymbol) ou d'étiquettes (LabelSymbol).

6.2 Fonctions

Il est important de noter que si les paramètres d'une fonction sont à première vue équivalents à des variables locales à l'intérieur du corps de celle-ci, pour le code assembleur généré (instruction `arg` du code 3 adresses), il faut cependant les différencier. On précisera qu'un `Operand3a` représente un paramètre et non une variable locale à l'aide de la méthode `setParam` de la classe `VarSymbol`.

```
/** Spécifie que l'opérande est un paramètre */
public void setParam()
```

Le type d'une fonction se définit à l'aide d'un objet `FunctionType` (fichier `FunctionType.java`) construit avec le type de la valeur de retour de la fonction. Pour les déclarations de fonctions/prototypes, on utilise également un attribut booléen qui indique si nous sommes en train de déclarer un prototype (sans le corps de la fonction) ou de définir une fonction (en spécifiant son code).

```
/** Type d'une fonction renvoyant un élément de type ret */
public FunctionType(Type ret)

/** Si prototype == true, ceci est une déclaration de prototype,
    sinon, ceci est une définition de fonction. */
public FunctionType(Type ret, boolean prototype)

/** Constructeur pour les appels de fonction (ni déclarations, ni définitions). */
public FunctionType(Type ret)
```

A Code 3 adresses

Le langage intermédiaire choisi pour le compilateur VSL+ est un code trois adresses, sorte de pseudo-assembleur relativement puissant. Ce code est structuré en une suite de déclarations de fonctions comme le code VSL+. Le corps de ces fonctions est une suite de déclarations et d'instructions. Il n'y a pas d'ordre imposé entre déclarations et instructions. Par contre, une variable utilisée doit avoir été déclarée avant son utilisation.

Déclarations d'étiquettes Une étiquette est une pseudo-instruction permettant de repérer une adresse dans le code. La forme générale d'une déclaration d'étiquette est :

```
label Lnnn
```

où `nnn` désigne un entier. *Les numéros d'étiquettes de 0 à 9 sont réservés*. Le distributeur de nouvelles étiquettes fournit des labels avec ce type de nom.

Les fonctions Toute déclaration de fonction commence par une étiquette qui servira à générer l'adresse de saut pour l'appel à cette fonction. On a alors la forme suivante :

```
label ident
beginfunc
code
endfunc
```

L'étiquette de la fonction doit impérativement porter le nom de la fonction. On rappelle que dans le code VSL+ doit exister une fonction `main` qui sera la première exécutée.

Les variables Toutes les variables (provenant du programme source ou créées par le compilateur) doivent être déclarées de la manière suivante (une seule variable par déclaration).

```
var nomvar
```

Les variables de tableau sont déclarées de la même manière, sans dimension. Les informations de type contenues dans les attributs associés au token sont utilisées lors de la génération de code machine et aussi pour la génération de messages d'erreur informatifs à l'utilisateur. Les seules constantes admises dans le code 3 adresses sont les entiers.

Les instructions Les instructions comprennent des opérations arithmétiques, une affectation, des appels et des retours de fonctions et des instructions de saut.

Opérations arithmétiques La forme générale d'une opération binaire est :

```
nvar := oper_1 op oper_2
```

où `oper_i` est un opérande qui peut être une variable ou une constante entière, `op` un opérateur parmi `+`, `-`, `*`, `/`. On dispose également du moins unaire qui prend la forme :

```
nomvar := - oper_1
```

Quant à l'affectation, elle est de la forme :

```
nomvar := oper_1
```

Noter les affectations spéciales pour les éléments de tableau :

```
T[i] := nomvar
nomvar := T[i]
```

Ce sont les seules instructions où les éléments de tableau apparaissent. Dans ces instructions, l'index i doit représenter le déplacement, en octets, depuis le début du tableau.

Appels de fonctions Pour appeler une fonction, il est nécessaire de décrire les paramètres effectifs selon leur ordre d'écriture (ceci prépare leur empilement). Un appel de fonction a alors la forme générale :

```
arg add_1
........
arg add_k
var = call ident
```

où `ident` est l'étiquette générée lors de la déclaration de fonction et `add_i` est l'adresse où se trouve l'argument, c'est-à-dire un identificateur de variable, un entier (adressage immédiat) ou une étiquette désignant un emplacement mémoire contenant une constante. Le code de la fonction s'exécute ensuite jusqu'à la rencontre de `endfunc` ou de l'instruction `return oper_i` qui permet d'interrompre la fonction en retournant une valeur qui est récupérée dans `var`.

Les fonctions d'impression/lecture ne rendent pas de résultat et ont donc une séquence d'appel plus simple :

```
arg add
call Lnnn
```

où `add` est soit l'étiquette repérant la zone mémoire où est stockée la chaîne à imprimer, soit une variable de type entier (pour la lecture).

Instructions de saut Il y a un saut inconditionnel `goto Lnnn` et deux sauts conditionnels :

```
ifz nvar goto Lnnn
ifnz nvar goto Lnnn
```

Le premier saut est effectif si la valeur de la variable de nom nvar vaut 0 (faux), le deuxième si elle a une valeur différente de zéro (vrai).

Les fonctions de bibliothèque Comme tous les compilateurs, celui de VSL+ utilise des fonctions de bibliothèque (aussi appelées *built-ins*) pour implémenter certaines instructions. Il s'agit ici des instructions d'impression et de lecture. Pour l'impression on a deux fonctions de bibliothèque : `printn`, située à l'adresse repérée par `L2`, qui imprime un entier (un seul argument) et `prints`, située à l'adresse repérée par `L4`, pour imprimer une chaîne de caractères ; l'argument unique de cette dernière est toujours rangé dans la zone de données à une adresse repérée par une étiquette. La lecture est limitée aux entiers. La fonction de lecture d'adresse `L8` lit un entier constitué d'une suite de chiffres terminée par n'importe quel caractère non numérique. Ce dernier caractère est évidemment ignoré. Ces fonctions peuvent être générées en utilisant les étiquettes définies dans `SymbDistrib`.

Les constantes chaînes de caractère Tout programme comporte des données constantes. En VSL+, ces données sont soit des constantes entières, soit des chaînes de caractères. Les constantes entières sont stockées comme opérandes des instructions 3a. Par contre, il est nécessaire de réserver de la mémoire pour les constantes chaînes de caractères. La représentation d'un programme en code 3 adresses comprend donc une zone d'instructions et une zone de données correspondant grosso modo aux `.text` et `.data` du format ELF (*Executable and Linkable Format*). La méthode `appendData` de `Code3a` permet de stocker ces constantes. Afin de les visualiser, la fonction d'impression du code les fait apparaître sous la forme générale :

```
Lnnn : "du texte"
```

Détection et notification d'erreurs Il est important de notifier l'utilisateur en cas d'erreurs de compilation. Des erreurs de syntaxe sont détectées par ANTLR : en présence d'un fichier syntaxiquement incorrect, ANTLR échoue l'analyse et il n'est pas nécessaire de prendre une action en particulier. Pour les erreurs dites *sémantiques*, par contre, telles que l'incompatibilité entre types ou l'utilisation d'une variable non déclarée, votre code doit les détecter et émettre un message pour en informer l'utilisateur. Les classes `Errors` et `Util` fournissent des méthodes pour l'émission de messages d'erreur.

B Exemple de compilation

Vous avez ci-dessous le code d'un programme VSL+ (à gauche) et le code 3 adresses produit à droite. Une lecture attentive de ce fichier permet d'inférer comment générer du code pour la plupart des instructions VSL+.

Remarque : le code source est disponible dans `tests/example_fact.vsl`.

```
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
}
```

```
label main
beginfunc
var n
var i
var t
arg L10
call L4
n =  call L8
i = 0
label L11
var T_0
T_0 = n - i
ifz T_0 goto L12
var T_1
arg i
T_1 =  call fact
t [ i ] = T_1
var T_2
T_2 = i + 1
i = T_2
goto L11
label L12
i = 0
label L16
var T_3
T_3 = n - i
ifz T_3 goto L17
arg L13
call L4
arg i
call L2
arg L14
call L4
var T_4
```

```
T_4 = t [ i ]
arg T_4
call L2
arg L15
call L4
var T_5
T_5 = i + 1
i = T_5
goto L16
label L17
endfunc

label fact
beginfunc
var n
ifz n goto L18
var T_7
var T_6
T_6 = n - 1
arg T_6
T_7 =  call fact
var T_8
T_8 = n * T_7
return T_8
goto L19
label L18
return 1
label L19
endfunc

L10: "Input n between 0 and 11:\n"
L13: "f("
L14: ") = "
L15: "\n"
```

C NACHOS

NACHOS (*Not Another Completely Heuristic Operating System*) est un système d'exploitation à vocation pédagogique conçu à l'origine à l'université de Berkeley [2]. Son objectif est de permettre à des étudiants d'appréhender le fonctionnement interne d'un système d'exploitation. Pour ce faire, il dispose des mécanismes utilisés dans les systèmes commerciaux et il manipule des programmes binaires compilés pour le jeu d'instructions MIPS en utilisant un compilateur croisé (cross-compilateur).

Dans le cadre du TP de compilation, NACHOS sera utilisé comme un simulateur permettant d'exécuter les programmes générés par votre compilateur. Vous aurez l'occasion de voir plus en détail le fonctionnement interne de NACHOS lors du cours de système.

C.1 Compilateur croisé

À la différence d'un compilateur standard qui produit des fichiers binaires pour la machine et le système d'exploitation sur lesquels il s'exécute, un compilateur croisé est un compilateur produisant des fichiers binaires vers une cible différente de la machine hôte. Dans notre cas, la machine hôte est un x86 sous un système LINUX et la cible est une machine MIPS sous le système NACHOS.

Pour produire des exécutables MIPS à partir des fichiers de sortie de votre compilateur VSL+, vous disposez du compilateur croisé gcc (cross-cc, cross-ld...) dans le répertoire :
/share/m1info/cross-mips/cross-tool/mips/

C.2 Compilation d'exécutable MIPS à partir de VSL+ pour NACHOS

La procédure de compilation pour produire un exécutable MIPS est la suivante :
— Production de code 3 adresses à partir du langage VSL+
— Production de code assembleur MIPS à l'aide de la classe MIPSCodeGenerator
— Utilisation du compilateur croisé pour MIPS afin de produire un exécutable

Production de code 3 adresses. Cette phase est à réaliser par vos soins, dans le cadre des TP de Compilation. Considérons pour l'heure qu'à l'issue de cette phase, vous disposez d'un objet de type *Code3a* nommé *code* représentant le code 3 adresses produit.

Production de code assembleur MIPS. Cette phase permet de transformer le code 3 adresses en un fichier assembleur MIPS. Cette transformation est réalisée en utilisant la classe MIPSCodeGenerator dans *VslComp.java* de la façon suivante :

```
MIPSCodeGenerator cg = new MIPSCodeGenerator(output);
cg.genCode(code);
```

avec l'objet *output* (de type *PrintStream*) pour préciser le flux de sortie : soit la sortie standard (`System.out`), soit un fichier (nommé *file.s* par la suite).

La classe *MIPSCodeGenerator* offre également une méthode *addStubMain(Code3a c)* qui a pour effet de produire l'entête et la sortie de la fonction *main*. Cette méthode vous permet de tester le code

2. Toutes les informations relatives à NACHOS pourront être trouvées sur la page d'accueil NACHOS à l'URL *http ://www.cs.washington.edu/homes/tom/nachos*

produit par votre compilateur VSL+ sur NACHOS avant la génération de code pour les fonctions. Une fois appelée avec le code généré pour une expression, instruction ou bloc, elle produit un nouveau code qui peut être ensuite fourni à la fonction *genCode*. La fonction *addStubMain* sera par la suite à retirer pour laisser place à votre propre génération de code.

Production de l'exécutable. Dans cette dernière étape, vous utiliserez le compilateur croisé MIPS afin de produire des exécutables.

Cette production se réalise en deux temps :

— Compilation du fichier assembleur en un fichier objet (extension .o)

— Edition de liens avec les librairies de NACHOS (pour disposer des fonctions print et read) produisant l'exécutable.

Le script *asm2bin.sh* fourni dans le répertoire du simulateur vous permet de réaliser cette phase. Vous devez placer le fichier assembleur *file.s* dans le répertoire *test* de NACHOS et exécuter la commande suivante dans le répertoire principal du simulateur :

```
./asm2bin.sh file
```

qui aura aura pour effet de générer un fichier objet *file.o* ainsi que l'exécutable *file* dans le répertoire *test* si le fichier assembleur fourni est valide.

Finalement, pour exécuter le programme sur le simulateur NACHOS vous exécuterez la commande suivante dans le répertoire principal du simulateur :

```
./exec.sh file
```

D Diagrammes de classes

Nous présentons ici trois diagrammes de classes qui donnent un aperçu de l'architecture du compilateur.

Le premier diagramme (Figure 1) synthétise la vue globale du compilateur : la classe *VslComp*, qui contient la fonction *main*, démarre l'exécution du compilateur. Après les étapes d'analyse lexicale et syntaxique, l'analyseur produit un AST, qui est alors fourni à *VSLTreeParser*. Celui-ci invoque à la fois des fonctions de génération de code (*Code3aGenerator*) et vérification de types (*TypeCheck*). *Errors* et *Util* fournissent quelques éléments pour émission de messages d'erreur et débogage. Le code assembleur généré est composé de données constantes (*Data3a*) et instructions (*Code3a*, qui est une liste de *Inst3a*). Tous les deux incluent des opérandes (*Operand3a*) qui sont aussi des symboles de compilation (détaillées dans le prochain diagramme). Chaque instruction possède un *opcode* parmi ceux énumérés dans *TAC*.

La hiérarchie des *symboles* est présentée dans la Figure 2 : des variables, constantes, étiquettes et fonctions ont chacune leur représentation, avec des attributs particuliers. Chacun de ces éléments a un identifiant qui sert comme clé dans la table de symboles (*SymbolTable*). Pour la génération de variables temporaires, ou étiquettes spéciales, un distributeur de symboles (*SymbDistrib*) est présent.

Enfin, dans la Figure 3 nous présentons la hiérarchie des types connus par le compilateur : en plus des types *INT* (variables entières), *I_CONST* (constantes entières), *LABEL* (pour des étiquettes) et *POINTER* (lié aux tableaux), nous avons aussi le type *VOID* pour les fonctions sans valeur de

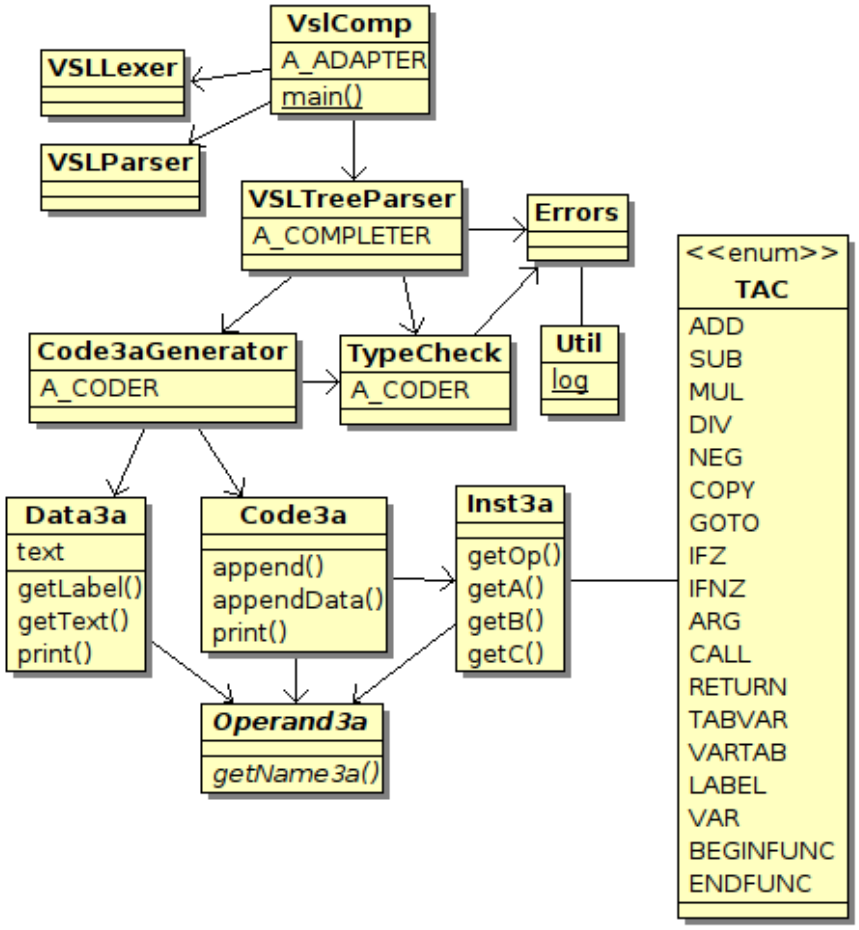

FIGURE 1 – Vue globale du compilateur, centrée sur le parcours de l'AST et génération de code.

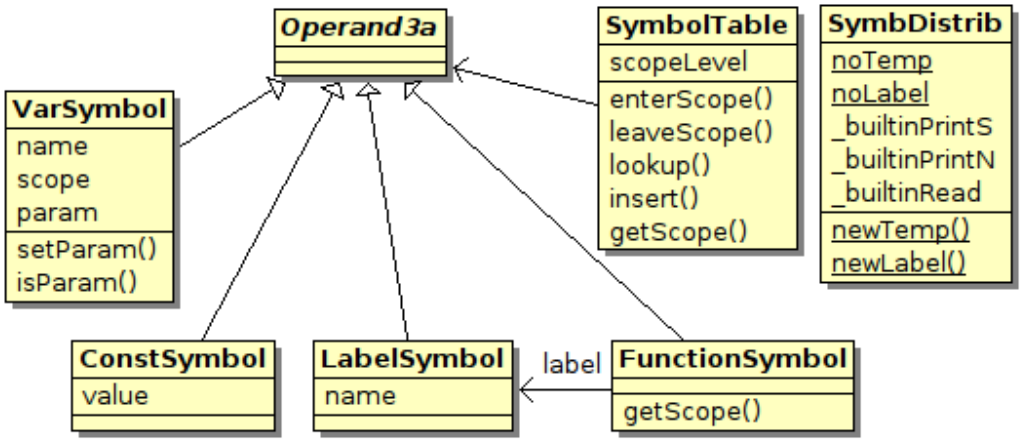

FIGURE 2 – Hiérarchie des symboles (*Operand3a*) et classes liées : table de symboles (*SymbolTable*) et distributeur de symboles (*SymbDistrib*).

retour, et le type *ERROR* qui peut être utile pour l'affichage des erreus de compilation. Enfin, les types pour les tableaux (*ArrayType*) et les fonctions (*FunctionType*), qui sont plus complexes, ont leurs propres classes.

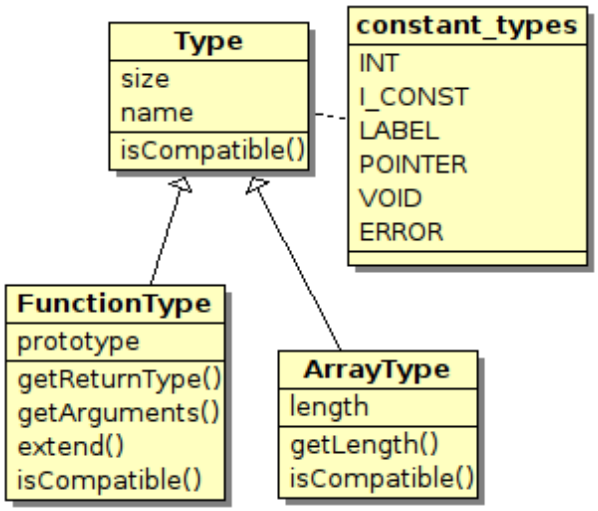

FIGURE 3 – Hiérarchie des types : la plupart des types sont des types constants (énumérés dans *constant_types*), mais nous avons aussi *FunctionType* et *ArrayType*, qui les étendent.