Compilation du langage 4D avec LLVM

Aubry Hervé – Florian Lemaitre

Mars 2015

Superviseurs :

Supélec	4D
Gianluca Quercini	Damien Gérard
Idir Aït Sadoune	Laurent Esnault

Table des matières

Introduction

4D est une entreprise fournissant aux développeurs une plateforme logicielle intégrée et puissante, accélérant et simplifiant le développement et le déploiement d'applications métiers.

Dans l'optique d'avoir un logiciel 4D opérationnel sur toutes les plateformes en constantes évolutions et pour bénéficier des dernières optimisations en date, l'idée d'utilisé la technologie de compilateur LLVM a été proposée.

En effet, LLVM est une infrastructure de compilateur conçue pour l'optimisation à la compilation, l'édition de liens ou l'exécution. La technologie LLVM est libre, en constante évolution et se développe dans la plupart des systèmes.

L'objectif de ce Projet CEI est donc de tester la faisabilité d'une telle entreprise par la réalisation d'un POC ou « Proof Of Concept » pour avoir une idée concrète de la faisabilité et les difficultés d'une telle entreprise.

1 Attentes

Ce projet à pour objectif de réaliser un POC (proof of concept). Il doit montrer si la création d'un compilateur pour le langage 4D utilisant la technologie LLVM est réalisable.

Concrètement le but prioritaire est de parvenir à faire tourner une fonction 4D simple en utilisant la technologie LLVM.

Les étapes du projet sont les suivantes :

- Réaliser un lexer qui reconnaît et transforme un code 4D en lexèmes.
- Réaliser une structure AST (Abstract Syntax Tree) qui symbolise le code 4D.
- Réaliser un parser qui crée l'AST à partir des lexèmes.
- Réaliser un builder qui transforme l'AST en code LLVM IR.

Pour assurer un suivi et une forte qualité du code produit, chaque partie à été systématiquement testée par des tests fonctionnels. Chaque nouvelle étape doit passer toutes les bancs de test précédents.

2 Le Langage 4D

Le langage 4D est un langage impératif de haut niveau. Il est multi-plateforme et permet de faire de la gestion de base de données relationnelle dans le logiciel 4D.

Voici les différentes spécificités que nous avons dû prendre en compte pour notre compilateur :

2.1 Les fonctions

Un fichier 4D correspond à une fonction informatique. Le nom de la fonction correspond au nom du fichier sans le chemin ni l'extension.

Les arguments de la méthode sont nommés $1, $2...

La valeur de retour de la fonction doit être stockée dans $0

```
C_TEXT($0)     // Type de la valeur de retour
C_LONGINT($1) // premier paramètre
$0 := "Valeur = " + String($1)
```

code 1: Paramètres en 4D

La vérification de la signature de la fonction se fait dans le code grâce aux fonctions de typage (voir 2.2). On gère aussi de cette façon les paramètres optionnels : le type doit alors correspondre à celui déclaré si l'argument est passé.

```
C_LONGINT($1) //le premier paramètre sera un int

If (Count parameters >= 2)
  ALERT($2)
End if
```

code 2: Fonctions de typage en 4D

2.2 Les variables

Une variable peut être de trois types :

- global_process : caractérisé par son absence de préfixe, elle existe dans tout le processus.
- local_var : caractérisé par son préfixe $, elle n'existe que dans la fonction.
- interprocess : caractérisé par son préfixe <>, elle existe en dehors du processus (accessible par plusieurs processus).

En dehors de ça, il n'y a pas de sous-espace de noms (une variable déclarée dans un if ne disparaît pas à la fin de celui-ci).

```
global_process:=10
If(1>0)
  $local_var:="Some text"
End if
<>persistent:=$local_var
ALERT ($local_var)
```

code 3: Variables en 4D

Les variables peuvent être typées de manière dynamique. On peut cependant les typer statiquement (ce qui permet de vérifier le type des arguments par exemple).

Les fonctions et les variables acceptent des espaces a l'intérieur de leur noms (les espaces avant et après ne sont pas pris en compte).

```
C_LONGINT(global_value)
C_TEXT($1; $2)
```

code 4: Typage statique en 4D

```
Ma Variable := 13
Ma Fonction()
```

code 5: Espaces en 4D

2.3 Opérateurs

On retrouve tous les opérateurs usuels.

Il faut noter qu'il n'y a pas de priorité d'opérateurs en 4D.

```
$a := 2+3*6    // =30
$a := 2+(3*6) // =20
```

code 6: Priorité des opérateurs 4D

Le symbole « := » n'est pas considéré comme un opérateur et, à ce titre, s'applique donc après la résolution de sa partie droite.

2.4 Tableaux

Les tableaux commencent à 1 même si l'index 0 reste utilisable. Ils sont créés par `ARRAY <TYPE>(<nom de la variable>; <nombre d'éléments>)`.

```
ARRAY TEXT(tt;3)
tt[1] := 42
```

code 7: Tableaux en 4D

2.5 Tables et champs

Les tables sont des objets spécifiques à 4D. Ils symbolisent les tables d'une Base de Données SQL et ont la syntaxe suivante : `[NomDeLaTable]`.

De même chaque champs d'une table peut être sélectionné par `[NomDeLaTable]NomDuChamps`.

Le langage 4D supporte l'insertion de commandes SQL entre les deux balise `Begin SQL` et `End SQL`. Ces commandes SQL peuvent prendre en compte les variables 4D utilisées précédemment.

```
ARRAY TEXT(tt;0)
Begin SQL
  SELECT * FROM [Table1] into :tt
End SQL
ALERT(tt{1})
```

code 8: Requête SQL en 4D

3 Implémentation actuelle du compilateurs

Nous avons à ce jour, un compilateur à même de compiler et d'exécuter toutes fonctions 4D ne manipulant que des entiers.

Il peut s'exécuter sur Linux et sur Windows 7.

3.1 Calendrier du projet

Le projets à connu l'avancement suivant :

1. Novembre 2014 : Création de la plateforme de développement et entraînement sur le tutoriel LLVM.
2. Décembre 2014 : Création du LEXER et des premiers tests fonctionnels du LEXER
3. 1ère moitié de Janvier 2015 : Création de la structure de l'AST
4. 2ème moitié de Janvier 2015 : Création du PARSER (+ tests)
5. Février 2015 : BUILDER et génération de code LLVM IR (+ tests). Début de parcours de l'AST en vu d'un typage des expressions.
6. Mars 2015 : Fin du générateur de code, ajout des appels de fonctions (builtins ou provenant d'autres fichiers), Refactor général.

3.2 Fonctionnalités prise en comptes

Le compilateur est capable de gérer :

1. Les variables de type entier.
2. les arguments de fonctions et les retours de type entier.
3. Les fonction builtins `ALERT` et `ABORT` (`ABORT` ne fait, certes, pas parti du langage mais nous a été bien utile pour les tests).
4. les IF avec ou sans ELSE.
5. Les boucles WHILE et REPEAT.
6. Les boucles FOR avec ou sans la valeur d'incrément.
7. Les appels de fonctions codées dans d'autres fichiers 4D.

4 Restrictions de l'implémentation du langage

Par souci de simplification, une partie des fonctionnalités de 4D n'a pas été implémentée. Le but de ce choix est d'obtenir une version exploitable du compilateur 4D sous LLVM dans le temps imparti. Le projet étant un POC, il n'a pas à être exhaustif mais il doit permettre d'obtenir une idée fidèle des fonctionnements basiques du compilateur 4D sous LLVM.

4.1 Types

Pour simplifier la déclaration des types de variable, il a été décidé que les variables seraient déclarées en début de programme (fonctions `C_LONGINT`…). Cette option permettait d'éviter tout emplois de variables n'ayant pas un type connu.

Par la suite, les variables ont toutes été considérées comme des INT. Cette grande simplification permet d'obtenir des programmes dont ont peut vérifier l'efficacité. Les autres types de variables seront bien sûr indispensables à un compilateur industrialisable mais nous avons manqué de temps pour cette fonctionnalité.

Une fonction de parcours de l'AST typant les expressions à été amorcé mais n'est à ce jour pas encore fonctionnelle en ce qui concerne le typage.

4.2 Pointeurs

Les pointeurs rajoutant de nombreux comportement et opérateurs supplémentaires, ils ont été écartés dans ce POC.

4.3 Requêtes Bases de données

Toute la partie gestion des bases de données a été mise de côté. Le compilateur ne gère donc pas :

- les objets de type table (`[Table]`).
- les champs (`[Table]champ`).
- les scripts SQL (`Begin sql […] End sql`).

Les raisons de cette omission sont les suivantes :

- la reconnaissance du langage SQL 4D modifié aurait été trop consommatrice de temps et nous avons préféré se focaliser sur la grammaire 4D.
- La gestion de BDD rajoute de nombreux comportements à rajouter.

4.4 Switch-Case

Le `switch-case` est une fonctionnalité d'ergonomie permettant de remplacer une succession de IF inélégants et répétitifs. Cependant il n’apporte rien d'un point de vue logique informatique. Il a donc été écarté pour le POC, il pourra être rajouté pour une version industrialisable.

4.5 Objets

Les Objets nécessitant l'utilisation de pointeurs, de tableaux et rajoutant de nombreux comportements au compilateur. Ils ont été écartés.

Une partie présentera plus en détail comment peut-on implémenter des objets avec LLVM.

5 Architecture et implémentation

Dans cette section, il sera abordé le sujet de l'architecture utilisée pour le compilateur ainsi que des détails d'implémentations de ce dernier.

5.1 Architecture générale

Le compilateur est centré sur l'objet Builder qui centralise le processus de compilation. Le Builder fait d'abords appel au Lexer sur le fichier à compiler, afin d'en extraire les lexèmes. Ensuite, le Parser lit les lexèmes afin de créer l'AST du code. Le Builder engage alors le tag de l'AST afin d'en extraire le typage de chaque nœud ainsi que pour récupérer les variables étant utilisées. Une fois cette étape finie, le Builder peut alors lancer la phase de génération de LLVM IR. Finalement, tout ceci est passé au moteur JIT qui va transformer le LLVM IR en binaire directement exécutable sur la machine. Il suffit alors de lancer cet exécutable depuis le Builder.

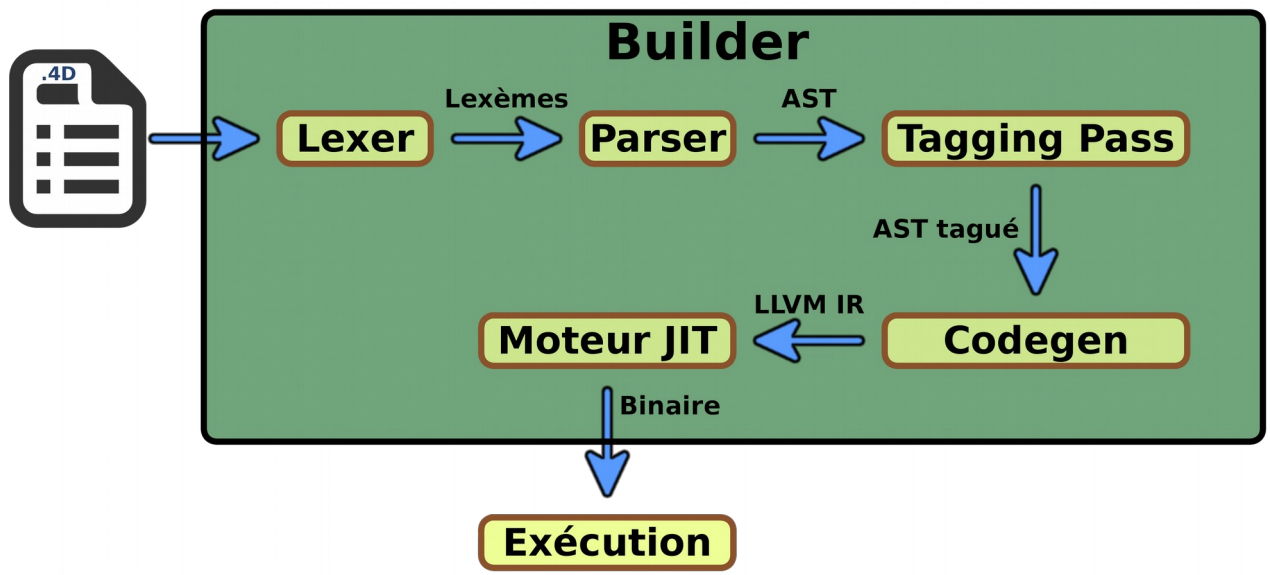
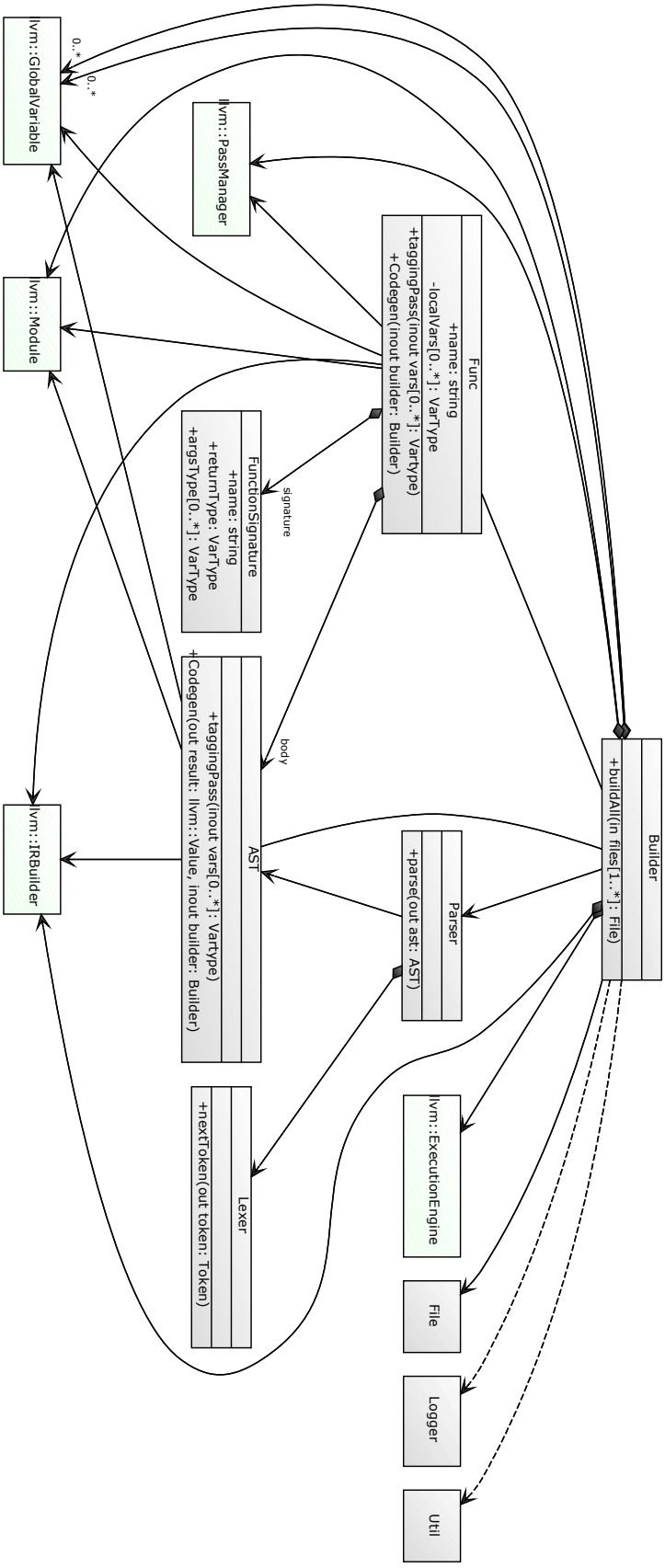


Schéma 1: Architecture générale du compilateur



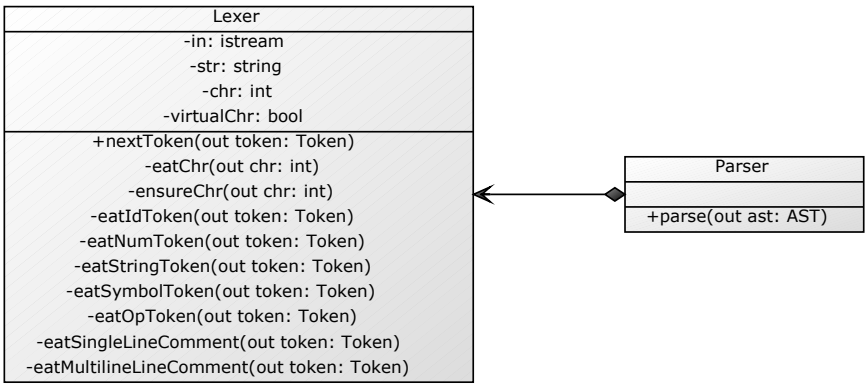
UML 1: Diagramme de classe global

5.2 Lexer

Le Lexer permet de transformer le texte (une suite de caractères) en une suite de lexèmes. Un lexème correspond à un morceau de texte insécable et portant un sens. On distingue par exemple plusieurs types de lexèmes :

- Lexème identifiant : tout ce qui est identifiable par un nom.
- Lexème nombre : correspond à un nombre.
- Lexème IF : correspond au mot clé IF.
- Lexème SEMICOLON : correspond à ';'.
- Lexème Opérateur : correspond aux opérateurs divers du langage, ex : '+', '-', '*'…
- …

Les lexèmes sont ensuite utilisés par le Parser.



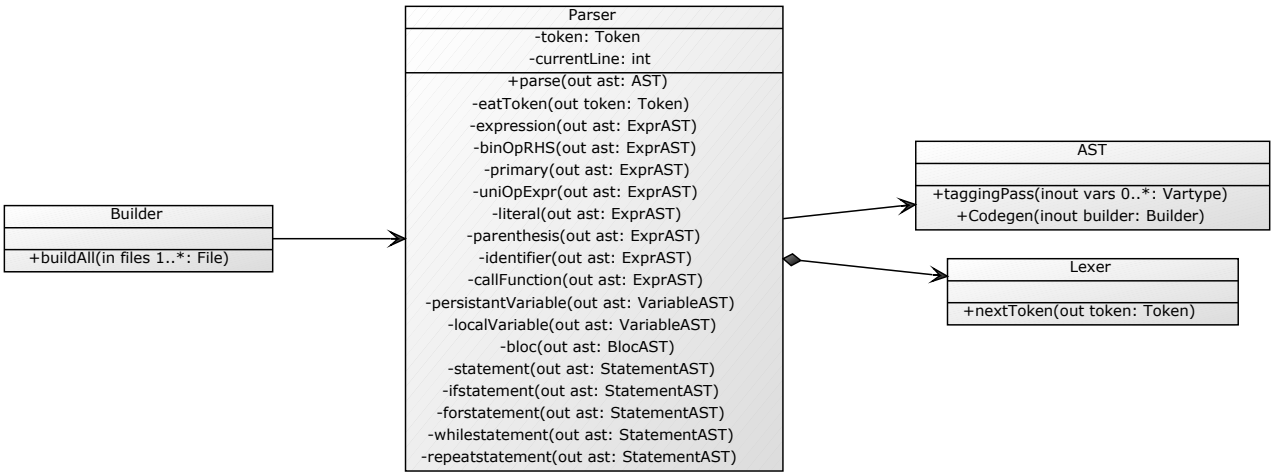
UML 2: Diagramme de classe du Lexer

5.3 Parser

Le Parser, quant à lui, transforme la suite de lexèmes en un AST selon la grammaire du langage. Ici, la grammaire n'a pas été définie formellement. On peut toutefois remarquer que notre implémentation du Parser repose sur le fait que la grammaire est LL(1). En effet, notre Parser commence par parser un bloc d'instructions, et descend, comme ceci, dans l'AST, tout cela en ne s'autorisant de regarder un unique lexème à l'avance : il est possible de regarder le lexème suivant, mais pas celui d'encore après.

Ceci permet notamment d'écrire le Parser très simplement. On observe ainsi le découpage dans les fonctions du Parser. Par exemple, la fonction `bloc()` permet de parser un bloc, tandis que la fonction `callFunction()` permet de parser un appel de fonction.

Le Parser est une étape très importante du compilateur car c'est cette partie qui donne du sens au code. C'est à ce moment précis que l'on commence à donner la sémantique du langage.



UML 3: Diagramme de classe du Parser

5.4 AST

L'AST (Abstract Syntax Tree) permet de représenter la structure du programme sous forme d'arbre. La structure que nous avons adoptée comporte 3 grandes familles de nœuds :

- Les Blocs
- Les Expressions
- Les Statements ou assertions

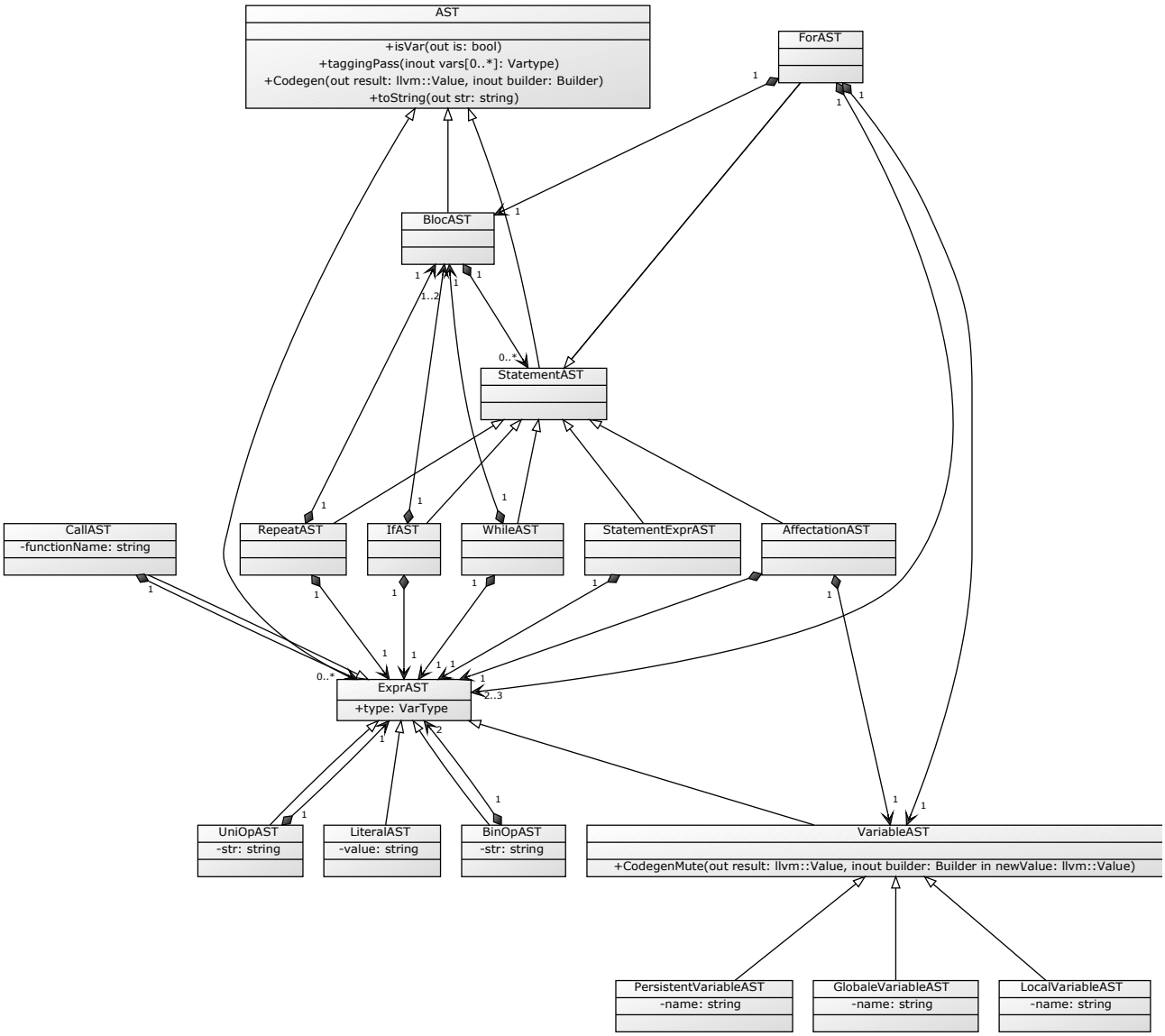
Les Blocs sont en fait des conteneurs pour une liste de statements. Générer un bloc revient donc à générer tous les statements qu'il contient.

Les Expressions correspondent à tout ce qui peut avoir une valeur comme par exemple une constante, une variable, un opérateur ou encore un appel de fonction.

Les Statements, quant à eux, correspondent à du code sans valeur. Il peut s'agir d'un IF, d'un WHILE, d'une affectation… Il existe aussi un nœud particulier qui permet d'avoir une expression comme statement. Ceci n'a d'utilité que pour garder une cohérence au niveau de la construction de l'arbre. Ainsi, un Bloc n'est qu'une suite de Statements, même si les Statements sont des expressions, comme par exemple des appels de fonctions.

Les nœuds de l'AST ont plusieurs fonctions utiles :

- `toString()` : permet d'afficher l'AST.
- `taggingPass()` : permet de taguer l'AST, en l'occurence d'inférer les types et lister les variables utilisées.
- `Codegen()` : permet de construire le LLVM IR de l'AST.



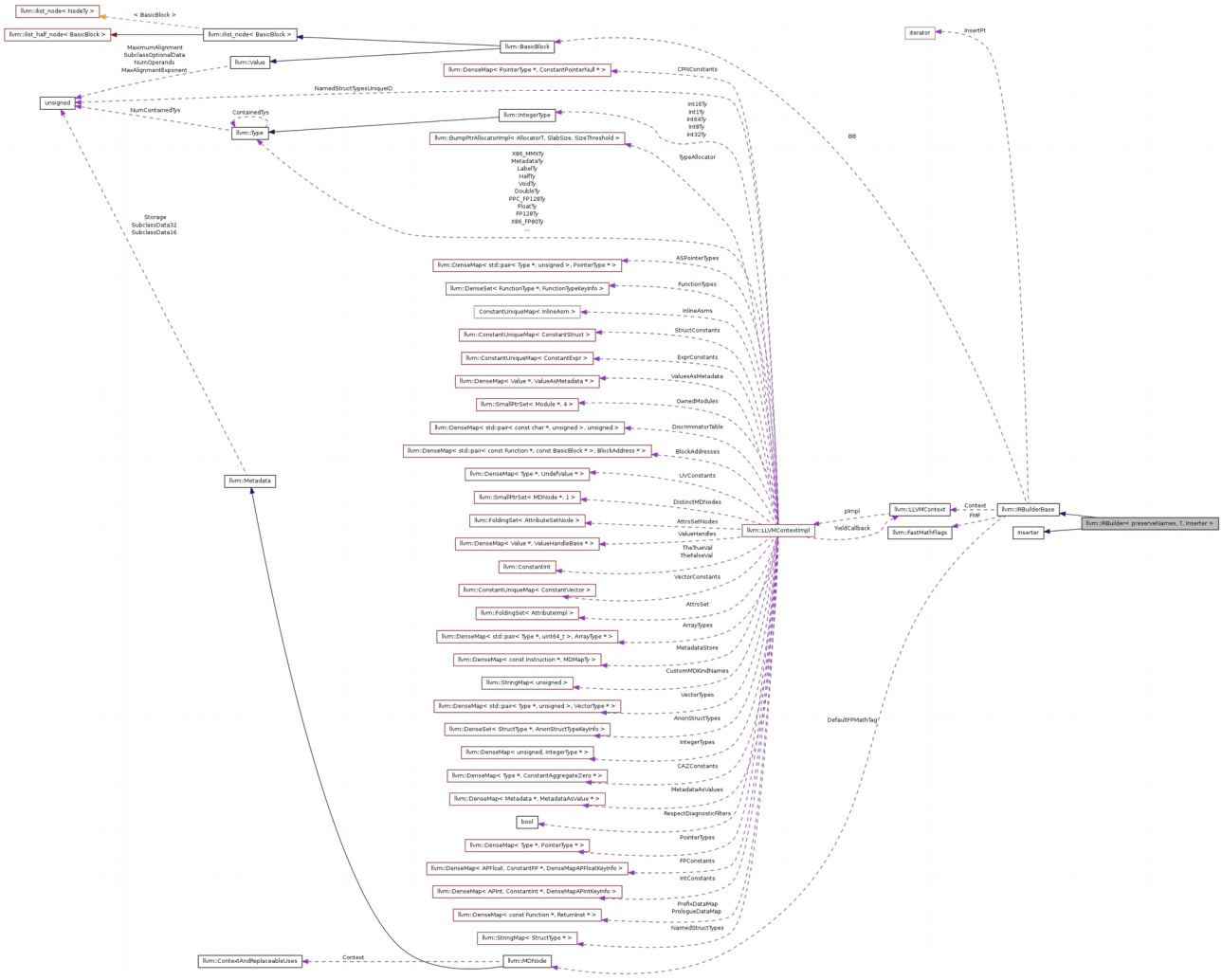
UML 4: Diagramme de classe de l'AST

5.5 LLVM IR

LLVM utilise une représentation interne pour la génération de code. Cette représentation s'appelle LLVM IR. La principale difficulté du projet résidait ici : Transformer notre AST en LLVM IR afin que LLVM puisse compiler notre code.

La création de LLVM IR utilise les concepts suivants :

- `Module` : Objet contenant le code généré ainsi que le contexte de génération.
- `IRBuilder` : Objet permettant de construire l'IR.
- `BasicBlock` : Bloc de code. C'est dans ces objets qu'il faut insérer le code généré.
- `Value` : Code pour l'IR. Ces objets génériques contiennent généralement une suite d'instructions permettant d'arriver à un résultat. Ces objets permettent de référencer ces résultats. Il est intéressant de noter que les BasicBlock sont des Value.
- `Function` : Représente les fonctions. Ces fonctions contiennent une liste de BasicBlock.

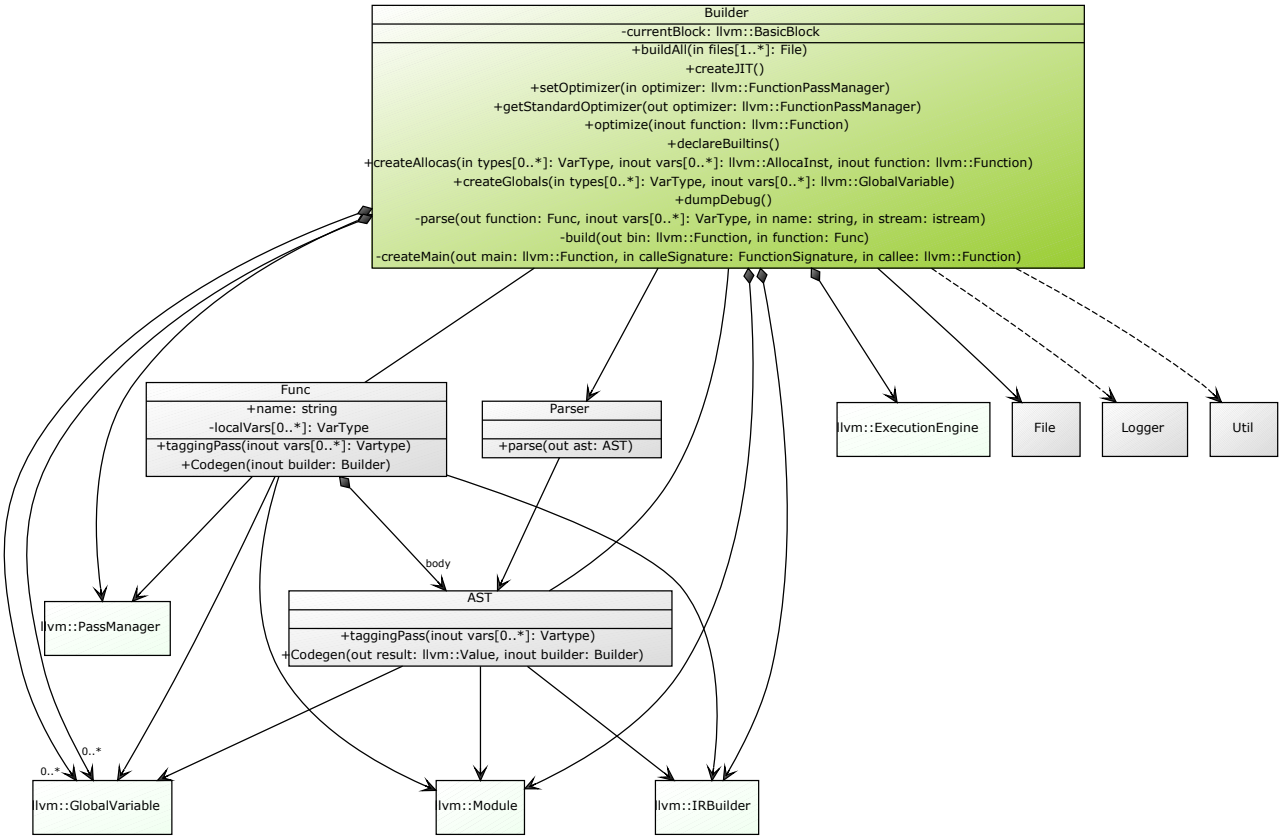


UML 5: Diagramme de classe simplifié de LLVM

5.6 Builder

Le Builder représente en quelque sorte le chef d'orchestre du compilateur. C'est, en effet, lui qui appel les autres classes et les interfaces entre elles. C'est aussi lui qui s'occupe de garder en mémoire les différents objets tels que les variables globales utilisées, les fonctions compilées, le Module qui sert à la compilation, l'IRBuilder qui construit l'IR, le contexte de compilation, le bloc courant, l'optimiseur, ainsi que le moteur JIT.

Toutefois, le Builder n'est pas un objet complexe, car il ne fait, principalement, qu'appeler des méthodes sur d'autres objets qui, elles, feront le véritable traitement. À ce titre, le Builder représente le cœur du compilateur.



UML 6: Diagramme de classe du Builder

5.7 Autres

Les points précédents ne couvrent pas la totalité du code. En effet, des parties du code sont là pour faciliter l'écriture du reste, ou n'avaient pas vraiment leur place dans le reste de la section.

5.7.1 Builtins

Les Builtins représentent une notion importante d'un langage, à savoir les fonctions disponibles dans le langages qui serait, soit impossible, soit difficile à écrire dans le langage lui-même.

Le choix a été fait, ici, de les écrire en C++ et de les interfacer avec le reste. En fait, chaque Builtin possède un identifiant, une signature, et une fonction C++ associée. Si dans l'AST, on essaye de générer l'appel à une fonction portant le même nom que l'identifiant d'un Builtin, alors l'appel sera remplacé par l'appel au Builtin.

L'ajout d'un Builtin se fait facilement en ajoutant une entrée dans la map `Builtin::_list` (fichier *`src/builtins.cpp`*).

On pourrait aussi imaginer la possibilité de pouvoir créer des Builtins inline, c'est à dire sans appel de fonction. Il faudrait alors appeler une fonction qui générerait le code LLVM à partir des arguments passés au Builtin. Ceci pourrait être très pratique pour implémenter le Builtin `NOT()` par exemple.

5.7.2 Types

Les types sont aussi très important dans un langage. Ils sont énumérés dans VarType. Voici les différents types qui sont référencés :

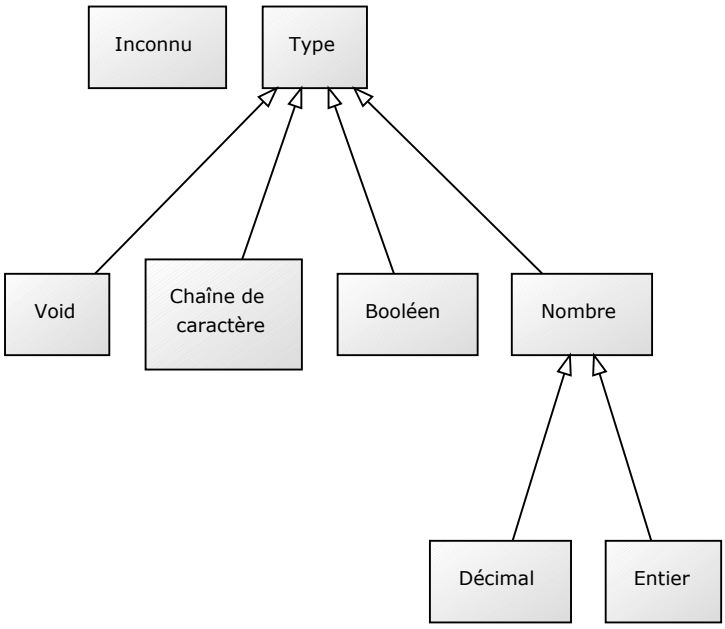


Schéma 2: Types

Ces types sont conceptuellement sous forme d'arbre pour exprimer les familles, mais ceci n'apparaît pas dans l'implémentation. Il sera ainsi possible de considérer un nombre sans savoir si c'est un entier ou un flottant. À noter également que tous les types qui existent en 4D n'apparaissent pas. Cela s'explique par le fait qu'ils ont été écartés pour se focaliser sur le principal et ainsi gagner du temps.

Il était prévu de faire correspondre à chaque type 4D un type LLVM, ainsi que des méthodes de conversion d'un type 4D en un autre type 4D. Mais ceci n'a pas été fait, faute de temps.

On peut aussi voir que ceci n'est pas utile si le compilateur gère le typage variant (cf : Typage Variant).

5.7.3 Logger

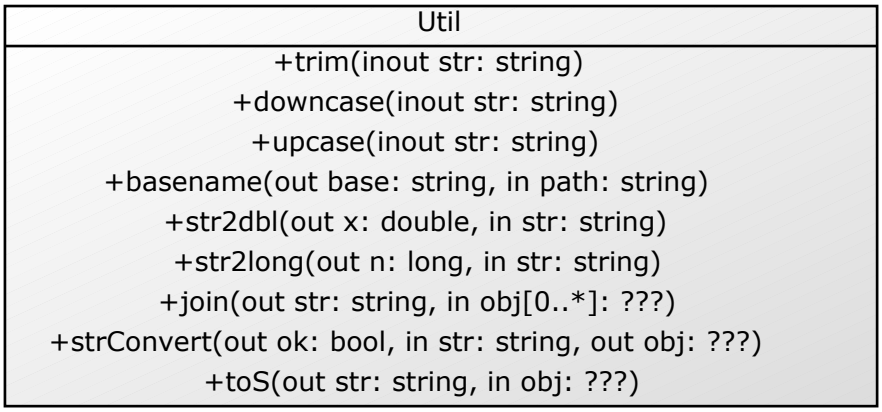
Le Logger n'est en fait qu'une collection de flux de sortie nommés. Nous disposons ainsi des flux suivants :

- Logger::debug
- Logger::info
- Logger::warning
- Logger::error
- Logger::critical

Ces flux permettent d'écrire les sorties selon le sens de ce que l'on veut écrire et non sur leur implémentation logiciel. Cela permet de choisir quel flux sera dirigé sur quelle sortie, voire de ne pas dirigé le flux du tout, pour ignoré le Debug par exemple.

5.7.4 Util

Util est un module présentant des fonctions facilitant certaines opérations, telles que la mise en minuscule d'une chaîne de caractères.



UML 7: Classe Util

6 Procédures de tests

Afin de s'assurer le bon fonctionnement de notre compilateur, nous avons utilisé et approvisionné une base de tests fonctionnels. Ces tests se présentent comme des programmes 4D dont l'exécution doit réussir ou échouer suivant la nature du test.

6.1 Ensemble de tests

Ces tests ont été principalement regroupés en 3 parties :

- Les tests orientés Lexer (11 tests)
- Les tests orientés Parser (18 tests)
- Les tests orientés Compilateur (14 tests)

Ces tests couvrent principalement les fonctionnalités que nous avons intégrées, mais très peu les fonctionnalités non implémentées du langage.

Les tests associés au Lexer sont principalement des tests pour vérifier que le compilateur ne bloque pas sur des lexèmes particuliers. Le seul test devant échouer ici est le test d'une chaîne de caractères non fermée.

Les tests associés au Parser vérifient que l'on arrive bien à créer l'AST que l'on souhaite à partir des lexèmes. Par exemple, Un branchement If sans condition ne doit pas compiler car non valide.

Les tests Compilateur quant à eux permettent de vérifier que le programme compile bien, même sur des exemples pouvant être critiques comme des boucles imbriquées ou des appels de fonctions récursives.

A l'heure actuelle, tous les tests associés au Lexer passent ; tous les tests associés au Parser passent également. Toutefois, sur les 14 tests associés au compilateur, seuls 11 passent correctement. 2 des tests ne passant pas concernent le typage qui n'a pas été fini. Le dernier test concerne la division par 0 qui ne renvoie aucune erreur.

6.2 Programme Testeur

Le programme que l'on a utilisait pour effectuer les tests est un script `bash` que nous avons réalisé pour l'occasion. Le script teste une succession de commandes et regarde la valeur de retour de cette commande : si la commande renvoie 0, il n'y a eu aucune erreur, sinon une erreur est survenue. Il est possible de préciser si une commande doit réussir ou échouer. Ainsi, si une commande renvoie 1 alors qu'elle devait échouer, alors le test a été passé avec succès. En revanche, si une commande renvoie 0 alors qu'elle devait échouer, le test sera considéré comme échoué. Il est alors possible de tester, à la fois, les cas devant fonctionner, ainsi que les cas devant échouer.

Les cas de tests sont décrits dans des fichiers prévus à cet effet et suivant la syntaxe suivante :

- Les lignes commençant par '**+**' désignent les cas de tests devant réussir.
- Les lignes commençant par '**-**' désignent les cas de tests devant échouer.
- Les lignes commençant par '**i**' désignent d'autres fichiers de tests à tester.
- Mis à part le symbole '**+**' ou '**-**' du début de ligne, chaque ligne décrit les paramètres qui seront donnés au programme à tester (ici *4dc*).
- Les lignes vides sont ignorées.
- Les lignes commençant par '**#**' sont ignorées
- Les lignes incorrectes sont ignorées avec un avertissement.

```
#Tests valides
+ lexerTests/testIf.4d
+ 4dcTests/testFunctionCall.4d 4dcTests/testFunctionCallMain.4d
#Tests d'erreur
- lexerTests/errorString.4d
#Inclusion d'un autre fichier
i parserTests.txt
```

code 9: Exemple de fichier de tests

Le programme s'exécute avec la commande suivante (avec '`programme`' le programme à tester et '`fichierDeTests`' le fichier contenant les tests à exécuter) :

- *`./execute_tests`* `[programme=../4dc/Bin/Debug/4dc] fichierDeTests`

Pour chaque test, le programme de test affiche le flux d'erreur de la commande en gris. Toutes les lignes contenant le mot « warning » sont surlignées en orange. Toutes les lignes contenant le mot « error » sont surlignées en rouge. Le flux de sortie standard n'est pas affiché.

```
bash-3.1$ ./execute_tests lexerTests.txt
Command: '../4dc/bin/Debug/4dc lexerTests/test1.4d' shall pass
SUCCESS Test successfully passed
Command: '../4dc/bin/Debug/4dc lexerTests/test2.4d' shall pass
SUCCESS Test successfully passed
Command: '../4dc/bin/Debug/4dc lexerTests/test3.4d' shall pass
        !Lex Warning: Comments not stored
SUCCESS Test successfully passed
Command: '../4dc/bin/Debug/4dc lexerTests/test4.4d' shall pass
        !Lex Warning: Comments not stored
SUCCESS Test successfully passed
Command: '../4dc/bin/Debug/4dc lexerTests/stringError.4d' shall fail
        !Lex Error: Expected '"' but found EOF
SUCCESS Test successfully failed
bash-3.1$
```

Illustration 1: Exemple d'exécution du programme de tests

7 Problèmes Rencontrés

Durant le projet, nous nous sommes vus confrontés à plusieurs problèmes principalement centrés sur LLVM. Ces problèmes nous ont considérablement ralentit, mais ce qui en ressort est souvent important.

7.1 Compilation d'un projet avec LLVM

Le premier problème que nous avons rencontré était la compilation d'un projet avec LLVM. Il nous a été, en effet, assez difficile de compiler le code que nous avions fait avec la bibliothèque LLVM.

La principale raison se trouvait être un problème d'édition de liens. En effet, dans la commande de compilation, il faut mettre les options spécifiques à LLVM à la fin de la commande, ce que nous n'avons pas réussi à faire dans Code::Blocks, peut-être dû à connaissance trop partielle du dit logiciel.

Exemple de commandes :

- `gcc -std=c++11 -c myfile.cpp -o myfile.o $(llvm-config --cxxflags core jit native)`
- `gcc -o myprogram myfile.o $(llvm-config --ldflags --libs core jit native)`

C'est pourquoi nous nous sommes tournés vers l'utilisation d'un Makefile qui nous a permis d'arranger l'ordre des options de compilation comme bon nous semblait. Ce Makefile a été écrit pour être exécuté par Gnu Make ; en ce sens, il est possible que ce fichier ne soit pas compatible avec d'autres versions de *make*. Il reste toutefois exécutable aussi bien sous Linux que sous Windows dans la mesure où il est possible d'avoir GCC et Gnu Make sur ce dernier OS grâce à Cygwin ou MinGW.

Il a également été choisi de compiler LLVM depuis ses sources afin d'avoir un projet « embarquable ». En effet, il est alors possible de compiler le projet sur une machine n'ayant pas LLVM d'installé, ou ayant une version différente de celle utilisée.

La compilation de LLVM sous Linux ne pose pas particulièrement problème. Il faut, toutefois, noter que la compilation prend du temps et de la place : à savoir entre 1 et 2 heures et nécessitant environ 4 Gio d'espace disque disponible pour la compilation. Il faut aussi que la compilation soit effectuée avec les bons paramètres, notamment en ce qui concerne le dossier d'installation de la bibliothèque.

Il est intéressant de noter également que les sources de LLVM pèsent environ 90 Mio, et la version compilée environ 300 Mio.

7.2 Compilation de LLVM sous Windows

Par contre, la compilation sous Windows a été une toute autre affaire. Tout d'abord, il semble nécessaire d'utiliser le combo GCC et Gnu Make pour compiler LLVM, donc d'avoir Cygwin ou MinGW d'installé sur la machine utilisée pour la compilation. Il est, peut-être, aussi possible de créer un projet pour Visual Studio grâce à *cmake*, mais n'ayant aucune connaissance dans ces deux technologies, cette option n'a même pas été considérée.

Outre le problème du lancement de la compilation sous Windows, qui est résolu grâce à l'utilisation de *bash* sous Windows, la compilation échouait après environ une heure. La raison, bien qu'obscure, a été très simple à corriger. Il a suffit d'ajouter un paramètre à la commande *make* :

```
make all CXXFLAGS=-D_GLIBCXX_HAVE_FENV_H
```

Cette option ne résolvait cependant pas tous les problèmes. En effet, bien que la compilation se terminait sans message d'erreur, il était impossible d'utiliser la bibliothèque précédemment compilée. Le problème venait ici dans la copie d'un dossier qui ne se faisait pas. La solution est alors très simple : effectuer la copie manuellement.

Toutes ces solutions ont alors été réunies au sein du fichier `compile-llvm.bat` permettant l'exécution séquentiel de toutes les commandes avec leur correction. Ce fichier est un fichier *batch* exécutable sous Windows et faisant explicitement appel aux programmes *sh* (alias pour *bash*) et *make*, ainsi que le programme Windows *xcopy*.

7.3 Génération de LLVM IR

Nous avons également rencontrer un problème lors de la génération de LLVM IR. Ce problème est apparu lors du début de la génération de l'IR, lorsque le programme ne générait que des expressions.

À ce stade, la génération ne produisait aucune erreur, mais semblait ne rien faire. En effet, le bloc dans lequel la génération se faisait restait désespérément vide alors qu'aucune optimisation n'était activée. Ce problème a miraculeusement disparu après avoir géré la génération de code pour les appels de fonctions.

L'explication est en faite assez simple : bien qu'il n'y avait aucune passe d'optimisation, LLVM effectue des optimisations à la génération de l'IR. Une des optimisations est la simplification automatique de constantes : par exemple, si on génère 3+5, LLVM générera 8. De la même manière, si on génère une valeur non-utilisée, LLVM omettra cette valeur et cette dernière n'apparaîtra donc pas dans le code généré.

7.4 Documentation LLVM

Enfin, un problème qui nous a suivi durant tout le projet concerne la documentation de LLVM. En effet, bien qu'étant assez utilisé, LLVM ne possède pas, à notre avis, une bonne documentation.

La documentation LLVM qui nous intéresse se décompose en 3 grandes parties :

- La documentation Doxygen
- Le manuel du développeur
- Le tutoriel Kaleidoscope

La documentation Doxygen est très précise et permet d'obtenir des informations sur une classe ou une fonction spécifique. Cependant, elle ne permet pas de savoir où chercher un comportement précis, par exemple comment créer une constante entière. Elle ne permet pas, non plus, de comprendre l'architecture utilisée par LLVM, et ne permet donc pas de comprendre la philosophie de LLVM. Cette documentation est donc très peu adaptée lorsque l'on cherche à faire quelque chose sans savoir comment. Par contre, elle est très efficace lorsque l'on cherche la signature d'un fonction, par exemple. Toutefois, les descriptions des fonctions et des classes sont assez lacunaires ; en ce sens, il est important de savoir ce que la classe fait.

Le manuel du développeur s'oriente plus vers les bonnes pratiques de l'utilisation de l'API LLVM. Il montre des comportements qui peuvent être intéressants avec des petits exemples de code. Mais là encore, ceci ne permet pas de rendre compte de l'architecture de l'API et ne fournit que des conseils, par exemple sur des conteneurs à utiliser. Si nous reprenons l'exemple de la constante entière, ce n'est toujours pas ici qu'il faut chercher.

Quant au tutoriel Kaleidoscope, il est, certes, très bien pour débuter et avoir du code fonctionnel. Cependant, il est très succinct et les explications qui sont données ne sont, généralement, ni très

claires, ni très complètes. Il permet donc une prise en main assez rapide de l'API, mais devient assez vite inutile lorsqu'il s'agit d'approfondir. Une fois encore, la création de notre constante entière ne peut pas être facilement déduite du tutoriel.

Le reste des documents disponibles couvrent plusieurs domaines qui ne nous intéressent pas directement :

- écrire un LLVM IR en texte, comme de l'assembleur
- utiliser LLVM comme back-end pour un compilateur existant (généralement *`clang`*), notamment comment écrire des passes d'optimisations
- utiliser LLVM pour compiler du code C/C++

Ce dernier point devient gênant lorsque l'on recherche des informations sur un moteur de recherche car c'est généralement dans ces 3 cas que l'on tombe. Il y a en effet beaucoup plus de résultats sur comment écrire une passe d'optimisation avec LLVM que comment générer notre constante entière.

7.5 Limitations imposées par LLVM

Pour des raisons de performances, LLVM impose de n'utiliser ni les exceptions, ni le cast dynamique. Ces limitations peuvent être contraignantes, mais devraient être respectées.

En ce qui concerne le cast dynamique, LLVM suggère d'utiliser des fonctions virtuelles à la place du cast dynamique. Toutefois, LLVM propose aussi une alternative au `dynamic_cast<>` et propose ainsi 3 primitives :

- `isa<>`
- `cast<>`
- `dyn_cast<>`

Toutefois, nous n'avons pas suivi ce conseil, notamment à cause d'une méconnaissance de ces alternatives proposées par LLVM, et nous avons donc réactivé RTTI auprès du compilateur.

L'autre point concerne les exceptions. LLVM conseille en effet de ne pas utiliser d'exceptions du fait que les exceptions ont un impact important sur les performances et interdit, par défaut, leur utilisation au niveau du compilateur C++. Cette fois-ci, nous avons respecter ce choix, bien que celui-ci complexifie considérablement le traitement des erreurs avec notamment une propagation de pointeurs nuls. Pour certaines erreurs, nous avons décidé d'arrêter l'exécution du compilateur en appelant la fonction `exit()`.

8 Réflexions diverses

Cette section a pour but de donner des voies de réflexion supplémentaires quant au projet au travers, notamment, de fonctionnalités qui n'ont pas été abordées au sein du projet faute de temps. Ces fonctionnalités restent néanmoins très intéressantes pour un projet de compilateur industriel.

8.1 JIT

La compilation Just-In-Time permet de compiler du code juste au moment où il y en a besoin. En voici les principaux avantages et inconvénients :

- Il n'y a pas de phase de compilation avant exécution, cela revient en quelques sortes à exécuter directement le code source.
- La compilation des fonctions se fait lors de leur premier appel, ce qui permet de gagner du temps lorsque de grosses fonctions ne pas appelées.
- Les performances à l'exécution peuvent être moindre car certaines optimisations ne peuvent être effectuées car trop gourmandes. Ce à quoi il faut aussi rajouter le temps de compilation des fonctions qui n'ont pas été compilées au moment de leur première exécution.
- Le débogage est simplifié car il devient possible, sous certaines conditions, de modifier le code à chaud. L'introspection du code à l’exécution est aussi facilitée car le moteur d'exécution connaît le code source ainsi que sa signification.

Un moteur d'exécution JIT a été mis en place sur le projet permettant d'exécuter le code fraîchement compilé. Cependant, son utilisation ne reprend qu'un seul des points précédents, à savoir l'absence de phase de compilation avant exécution. En fait, cette dernière existe, mais est masquée à l'utilisateur car l'exécution enchaîne la compilation. Cette phase ne serait perceptible que sur des gros programmes.

Toutefois, il semble être assez simple de passer à une véritable utilisation du moteur JIT. En effet, LLVM propose d'appeler une fonction lorsqu'une fonction inconnue est appelée (`llvm::ExecutionEngine::InstallLazyFunctionCreator(void *(*P)(const std::string &))`).
Il suffit alors d'appeler le Builder pour qu'il compile cette fonction. De plus, ce schéma est grandement amélioré par le fait que l'on ait exactement une fonction par fichier : il est donc tout à fait possible de retarder le parse du fichier au moment de compiler sa fonction.

Ceci ne devrait pas poser de problème car, dans ce cas, LLVM ne semble pas avoir besoin de connaître la signature de la fonction à appeler. Il va juste vérifier le nombre et les types des arguments une fois que la fonction à été compilée.

De plus, la structure actuelle du Builder est facilement adaptable pour réaliser cette compilation à la volée. Il devrait suffir d'appeler la fonction `llvm::Function* Builder::build(Func* Fdef)`.

8.2 Debug

LLVM prévoit des mécanismes pour effectuer du débogage des programmes compilés. De ce qu'il nous a été possible de voir, il s'agit principalement d'incorporer des symboles de débogages qui seront ensuite visible dans le débogueur *lldb*, le débogueur de LLVM. À noter également que LLVM utilise l'API de *gdb* qui permet notamment de déboguer du code compiler en JIT.

L'insertion de symboles de débogages n'a pas l'air excessivement difficile dans l'absolu et repose sur les principes suivants :

- Remplacement de l'objet IRBuilder par Dbuilder
- Ajout d'informations de localisations du code générer dans le code source
- Ajout de méta-informations supplémentaires à la génération

Le principal problème pour implémenter ceci réside dans la perte d'informations qu'il y a lors du traitement par le lexer, puis le parser. Toutefois, ceci ne devrait pas rester un problème puisque la version industrielle d'un tel compilateur devra embarquer de tels informations dans l'AST afin de pouvoir l'enregistrer.

En ce qui concerne les possibilités de débogage évoquées avec le moteur JIT, il ne semble pas y avoir beaucoup d'informations à ce sujet. Il pourra peut-être être difficile d'avoir un débogueur comme en JavaScript avec notamment l'exécution de code via une console afin d'exécuter du code dans le contexte d'exécution courant. Il sera, aussi, sûrement difficile d'avoir des informations complètes sur les objets manipulés.

De toutes façons, ce point nécessitera beaucoup d'efforts et pourrait bien être, à lui seul, au moins aussi complexe que le compilateur lui-même.

8.3 Compilation Multi-plateforme

A l'heure actuelle, le compilateur est compilable aussi bien sous Windows que sous Linux. Il y a d'ailleurs fort à parier que la compilation sous MacOS ne pose pas de soucis.

Il est intéressant de noter qu'il n'y a absolument pas besoin d'avoir LLVM sur la machine qui exécutera le compilateur en vue de compiler du code 4D.

Il existe cependant une subtilité avec la version Windows qui nécessite que MinGW soit installé pour pouvoir être exécuter. Ceci est dû au simple fait d'avoir compilé notre projet avec MinGW. Afin de palier ce problème, il suffit d'inclure statiquement les bibliothèques de MinGW pour que le projet fonctionne aussi sans MinGW. Ceci se fait très simplement en passant 3 paramètres en plus à *gcc* lors de l'édition de liens :

```
gcc -static -static-libgcc -static-libstdc++ ...other_options
```

8.4 Compilation Croisée

La compilation croisée permet de compiler un code qui sera exécuté sur une machine avec une architecture différente de la machine qui a compilé le programme. Dans notre cas, la compilation croisée peut apparaître à deux niveaux.

Il peut, en effet, être intéressant d'avoir recours à la compilation croisée pour compiler notre compilateur. A ce moment, la plus grosse difficulté risque d'être la compilation de la bibliothèque LLVM. Ceci ne devrait, toutefois, pas poser problème pour les architectures Windows et Unix-Like en x86 et amd64. Il existe tout de même une page donnant les ressources nécessaires pour effectuer la compilation croisée pour la compilation de LLVM :

http://llvm.org/docs/HowToCrossCompileLLVM.html.

En ce qui concerne la compilation croisée par notre compilateur, LLVM propose de pouvoir définir l'architecture cible. Une telle fonctionnalité ne devrait donc pas trop alourdir notre compilateur du fait que LLVM gère tout.

Il faut, en revanche, penser à toutes les fonctions écrites en C/C++ dont notre code pourrait avoir besoin. C'est par exemple le cas des BUILTINs qui sont écrites en C++ et appelés directement par le moteur JIT. Il est très simple de les compiler à part pour pouvoir les lier à notre programme, mais cela peut être gênant si l'architecture cible n'est pas la même que l'architecture de la machine. Il a été évoqué la possibilité de compiler au préalable ces fonctions vers de l'assembleur LLVM afin de pouvoir les compiler de la même manière que le programme 4D.

8.5 Compilation parallèle

La compilation parallèle pourrait être intéressante pour accélérer les temps de compilations, mais cela impose quelques limitations.

Tout d'abord, cette problématique rentre en conflit avec la compilation JIT. En effet, pour de la compilation JIT, une seule fonction est compilée à la fois, et la compilation d'une unique fonction ne semble pas se paralléliser très bien, et ce pour plusieurs raisons :

- Le parse se fait exclusivement séquentiellement en lisant les caractères un à un
- La compilation s'effectue en ajoutant séquentiellement les instructions à des blocs d'instructions. Ainsi, si on parallélisait la compilation au sein d'un bloc, les instructions pourraient être désordonnées. Il ne serait alors possible de ne compiler parallèlement des instructions exécutables parallèlement.
- LLVM ne semble pas savoir gérer la génération de plusieurs blocs au sein d'un même contexte en parallèle.

Cependant, LLVM peut compiler des fonctions différentes dans des contextes différents. Cela est faisable si l'on doit compiler toutes les fonctions en même temps, donc sans JIT. On peut alors appliquer le schéma suivant :

Tous les fichiers associés aux fonctions sont lus, et parsés en parallèle, par des objets Lexer et Parser différents. A ce moment, il n'y a aucun problème de concurrence. Il même possible d'effectuer la passe de Tag de l'AST en parallèle. Une fois ceci fait pour tous les fichiers, il faut alors créer plusieurs IRBuilder associés à plusieurs Modules. On déclare alors les variables globales trouvées dans la passe de Tag dans chacun des modules. A partir de ce moment, il est possible de compiler les fonctions en parallèle dans les modules séparés. Il suffit à la fin d'effectuer l'édition de liens afin de grouper tout le code au sein d'un unique programme.

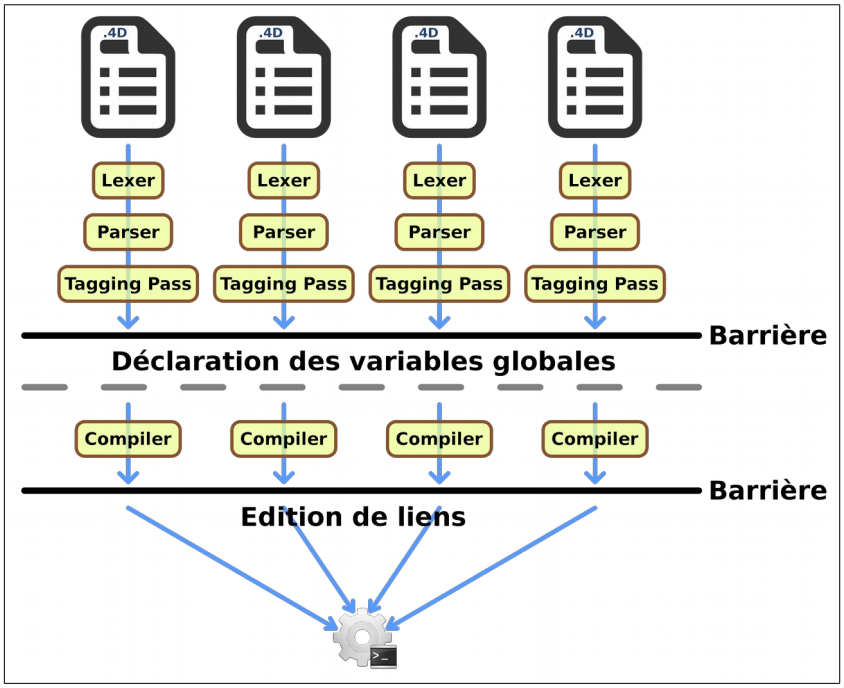


Schéma 3: Compilation parallèle

Cette approche peut vraiment être performante car il n'y a que 2 barrières dans l'algorithme, toutes les étapes intermédiaires pouvant être exécutées en parfaite concurrence.

8.6 Compilation incrémentale

La compilation incrémentale est une technique permettant de ne compiler que les parties du code qui ont été modifiées depuis la dernière compilation. Encore une fois, cette technique ne peut s'appliquer à la compilation JIT car cela n'aurait aucun sens dans ce cas précis.

En fait, la compilation incrémentale nécessite une approche très similaire à la compilation parallèle. En effet, comme dans ce dernier cas, il faut être capable de compiler de manière complètement indépendante les fichiers. La différence ici réside sur le fait que certains fichiers ont déjà été compilés et n'ont pas besoin d'être recompilés.

Toutefois, il faut faire attention aux variables globales qui ne doivent pas être initialisées à deux endroits. Pour ce faire, une solution consisterait à compiler les variables globales dans un fichier à part en récupérant les variables globales déclarées dans les fichiers compilés.

8.7 Objets

Les objets représentent une partie importante de beaucoup de langages de programmation. Il n'est cependant pas si simple de les implémenter. Il faut alors distinguer deux parties d'un objet : ses attributs, et ses méthodes.

Pour comprendre comment gérer les méthodes, voici un petit exemple de code objet écrit en C++ et traduit en C :

```
class Foo {
  double bar;
  int baz;
  void print() {
    printf("%lf %d", this->bar, this->baz);
  }
};

Foo* foo = new Foo();
foo->print();
```

code 10: Objet en C++

```
struct Foo {
  double bar;
  int baz;
};

void _Foo_print(Foo* this) {
  printf("%lf %d", this->bar, this->baz);
}

Foo* foo = malloc(sizeof(Foo));
_Foo_print(foo);
```

code 11: Objet en C

Dans cet exemple, nous voyons que l'appel à la méthode `Foo::print()` a été remplacé par l'appel à la procédure `_Foo_print(Foo*)` en passant notre objet comme premier argument de la fonction. Bien sûr, ces mécanismes sont cachés à l'utilisateur du langage pour ne pas qu'il ai à s'en soucier. C'est, d'ailleurs, aussi le cas en C++ : dans notre cas, la méthode `Foo::print()` a été remplacée en interne par la fonction `__ZN3Foo5printEv(Foo*)`. Ce procédé s'appelle « name mangling » et permet aussi d'avoir une surcharge de fonctions en donnant aux fonctions surchargés des noms différents selon la convention de nommage choisie.

En ce qui concerne les attributs, le problème est plus simple car LLVM gère les structures de données. Les attributs agissent alors comme un offset sur l'adresse de l'objet. Il n'est donc pas plus difficile d'accéder à un attribut d'un objet qu'à la valeur d'un pointeur.

```
%class.Foo = type { double, i32 }


; Function Attrs: uwtable
define linkonce_odr void @_ZN3Foo5printEv(%class.Foo* %this) #2 align 2 {
  %foo.ptr.ptr = alloca %class.Foo*, align 8
  store %class.Foo* %this, %class.Foo** %foo.ptr.ptr, align 8
  %foo.ptr     = load %class.Foo** %foo.ptr.ptr
  %foo.bar.ptr = getelementptr inbounds %class.Foo* %foo.ptr, i32 0, i32 0
  %foo.bar     = load double* %foo.bar.ptr, align 8
  %foo.baz.ptr = getelementptr inbounds %class.Foo* %foo.ptr, i32 0, i32 1
  %foo.baz     = load i32* %foo.baz.ptr, align 4
  %1 = call i32 (i8*, ...)* @printf(i8* "%lf %d", double %foo.bar, i32 %foo.baz)
  ret void
}
```

code 12: Objet en LLVM

8.8 Typage variant

Ce typage permet d'avoir un typage dynamique, disponible exclusivement lors de l'exécution du programme. Il est utilisé dans beaucoup de langages de script notamment JavaScript. Même si un code est plus facile à écrire avec un typage variant, le compilateur est beaucoup plus complexe. Il faut, en effet, que le compilateur génère du code pour tous les types de variables, et que le programme sache le code à utiliser pour effectuer l'opération.

Une des solutions consiste à encapsuler notre objet dans une structure comme ci-dessous :

```
struct var {
  void *value;
  char  type;
}
```

code 13: Structure d'un variant

Ainsi, au moment d'effectuer une addition, il faudrait « juste » vérifier le type des opérandes et d'agir en conséquences. On peut imaginer que si le type stocké est sur 1 bits, on génère un `switch-case` de cette forme :

```
switch ((left.type << 1) | right.type) {
case 0b00:
  // int + int
case 0b01:
  // int + float
case 0b10:
  // float + int
case 0b11:
  // float + float
}
```

code 14: switch-case variant

Une autre solution pourrait consister en un tableau de fonctions à appeler, dont l'indice est donné par : `((left.type << 1) | right.type)`. On aurait ainsi un très faible coût en place pour l'opération. Par contre, cette solution possède le coup de l'appel de fonction en plus.

```
void** addFunctions[4] = {
  &addIntInt,
  &addIntFloat,
  &addFloatInt,
  &addFloatFloat
};
```

code 15: Tableau de fonctions

```
addFunctions[(left.type << 1) | right.type](left, right);
```

code 16: Addition de variants

Il faut toutefois faire attention à une éventuelle surcouche objet. En effet, il ne serait plus possible de considérer les attributs comme de simples décalages d'adresses et les méthodes comme connues au moment de la compilation. Il faudrait alors garder le nom complet de l'attribut ou de la méthode et de les retrouver par une quelconque manière, une table de hachage par exemple. Ceci induit encore une chute de performance qui pourrait être préjudiciable dans certains cas.

Conclusion

Ce CEI est parvenu à prouver que la création d'un compilateur 4D restreint utilisant la technologie LLVM est possible.

Ce compilateur fonctionne sur les systèmes Windows et Linux et peut compiler des programmes complexes sous les restrictions imposées. Il peut par exemple compiler une programme de calcul de factorielle récursif.

Pour pouvoir réaliser un compilateur industrialisable à partir d'ici, trois pistes nous sont apparues :

- Rendre opérationnel et utiliser la passe de TAG de l'AST pour pouvoir permettre l'utilisation des variables non-entières.
- Rajouter les fonctionnalités 4D écartées lors de ce POC.
- Optimiser le processus de compilation et permettre le Just-in-Time ou le Debug.