TELECOM SudParis

Projet Informatique 1ère Année
CSC 3502

GROMATH

MAËL HOSTETTLER
AUGUSTIN PERRIN

Enseignant responsable : Pascal HENNEQUIN

Table des matières

Chapitre 1

Introduction

SageMath, aussi System for Algebra and Geometry Experimentation est un programme pour le calcul mathématique numérique et symbolique utilisant principalement le langage Python. Il trouve son utilité en CTF (pour Capture the Flag) qui sont des compétitions de cybersécurité ou il est demandé aux participants de résoudre des épreuves notamment dans le domaine de la cryptographie.
Cependant, l'exécution de programmes dans ces conditions nécessite d'être pleinement optimisée pour gagner du temps (précieux dans ce genre de compétition). D'où la naissance du projet GroMath (les mathématiques grolandaises).

L'objectif du projet est de créer notre propre langage qui serait un langage impératif auquel nous avons pour objectif d'ajouter des structures propices à la pratique de l'algèbre : nous pensons notamment à des structures de groupes, d'anneaux, de corps à l'aide de foncteurs.

Pour cela, nous verrons dans un premier temps le cahier des charges du projet : la grammaire choisie, les choix techniques. Puis dans un second temps, le développement nous permettra de détailler les choix de conceptions et l'utilisation de LLVM et Bison au travers de différentes ressources, ainsi que la mise en place de tests pour le projet.
Enfin, nous comparerons nos temps d'exécution avec SageMath et nos fonctionnalités afin d'interpréter nos résultats.

Chapitre 2

Cahier des charges

Le projet GroMath se positionne dans le domaine du calcul scientifique et mathématique en développant un nouveau langage de programmation impératif enrichi de structures mathématiques avancées. Ce langage, inspiré par la syntaxe familière à celle du langage C, est conçu pour combler le fossé entre les besoins computationnels complexes et la facilité d'utilisation. L'intégration de concepts tels que les groupes, anneaux, et corps directement dans le langage vise à offrir une puissance et une flexibilité sans précédent pour le calcul scientifique.

La pierre angulaire de ce projet réside dans son compilateur, qui exploite les technologies modernes telles que LLVM et GNU Bison. L'usage de LLVM est stratégique, offrant non seulement des avantages en termes de performances de compilation grâce à des fonctionnalités comme la JIT compilation et les Phi Nodes, mais aussi une portabilité et une optimisation avancée du code généré. GNU Bison, quant à lui, est employé pour parser la grammaire complexe du langage, permettant une traduction efficace du code source en instructions machine.

La grammaire du langage a été soigneusement conçue pour supporter une variété de constructions, allant des déclarations d'objets et de fonctions à des expressions complexes et des structures de contrôle. Cette grammaire est le résultat d'une réflexion approfondie sur les besoins spécifiques du calcul mathématique et scientifique, offrant ainsi une base solide pour le développement du compilateur.

Le développement du compilateur se déroule en plusieurs étapes clés, débutant par la création d'un analyseur syntaxique à l'aide de GNU Bison. Cette étape est cruciale pour transformer le code source en une structure intermédiaire que le compilateur peut manipuler. Ensuite, une série d'analyses sémantiques garantit que le code est logique et cohérent selon les règles du langage. La génération de code transforme cette représentation intermédiaire en code machine, avec LLVM jouant un rôle central dans l'optimisation et la portabilité du code généré.

Ce projet n'est pas seulement une prouesse technique, mais aussi une aventure collaborative. La conception du langage, le développement du compilateur, et les phases d'optimisation nécessitent une approche itérative et collaborative. Chaque étape, de la définition initiale de la grammaire à l'optimisation finale du compilateur, est l'occasion d'itérer, de tester, et d'affiner, assurant ainsi que le langage et le compilateur répondent aux exigences de performance et d'utilisabilité.

En somme, GroMath est un projet ambitieux qui vise à établir de nouvelles normes dans le domaine du calcul scientifique et mathématique. Par l'intégration de structures mathématiques avancées directement dans le langage et l'utilisation de technologies de compilation de pointe, GroMath promet d'offrir une solution puissante et flexible pour les chercheurs et les développeurs. Avec un engagement envers l'innovation et la qualité, ce projet est bien parti pour devenir un outil incontournable dans le domaine du calcul scientifique.

La grammaire est la suivante :

```
program : stmts
```

```
        ;

stmts : stmt
      | stmts stmt
      ;

stmt : var_decl TSCOLON
     | func_decl
     | expr TSCOLON
     | if_stmt
     | for_stmt
     | while_stmt
     | ret_stmt TSCOLON
     | break_stmt TSCOLON
     | continue_stmt TSCOLON
     ;

if_stmt : TIF expr TCOLON block
        | TIF expr TCOLON stmt
        | TIF expr TCOLON block TELSE TCOLON block
        | TIF expr TCOLON stmt TELSE TCOLON block
        | TIF expr TCOLON block TELSE TCOLON stmt
        | TIF expr TCOLON stmt TELSE TCOLON stmt
        ;

iterator : TRANGE TLPAREN expr TRPAREN
         | TRANGE TLPAREN expr TCOMMA expr TRPAREN
         | TRANGE TLPAREN expr TCOMMA expr TCOMMA expr TRPAREN
         | TLLBRACK expr TCOMMA expr TRRBRACK
         ;

for_stmt : TFOR simple_var_decl TIN iterator TCOLON block
         | TFOR simple_var_decl TIN iterator TCOLON stmt
         ;

while_stmt : TWHILE expr TCOLON block
           | TWHILE expr TCOLON stmt
           ;

break_stmt : TBREAK
           ;

continue_stmt : TCONTINUE
           ;

ret_stmt : TRET
         | TRET expr
         ;

block : TLBRACE stmts TRBRACE
      | TLBRACE TRBRACE
      ;

type : ident
     | type TRARROW ident
     ;
```

```
simple_var_decl : ident TCOLON type
                ;

var_decl : ident TCOLON type TEQUAL expr
         | simple_var_decl
         ;

func_decl : TFN ident TLPAREN func_decl_args TRPAREN TCOLON type block
          ;

func_decl_args : /*blank*/
          | var_decl
          | func_decl_args TCOMMA var_decl
          ;

ident : TIDENTIFIER
      ;

cst_string : TCSTSTRING
           ;

cst_int : TINTEGER
  //    | TDOUBLE
        ;

expr : ident TEQUAL expr
     | ident TLPAREN call_args TRPAREN
     | ident
     | cst_int
     | cst_string
     | binop
     | TLPAREN expr TRPAREN
     | TTRUE
     | TFALSE
     ;

call_args :
          | expr
          | call_args TCOMMA expr
          ;

binop : expr TCEQ   expr
      | expr TCNE   expr
      | expr TCLT   expr
      | expr TCLE   expr
      | expr TCGT   expr
      | expr TCGE   expr
      | expr TPLUS  expr
      | expr TMINUS expr
      | expr TMUL   expr
      | expr TDIV   expr
      | expr TMOD   expr
      | expr TPOW   expr
      ;
```

Chapitre 3

Liste des fonctionnalités de l'application

Le développement de GroMath, un langage de programmation innovant dédié aux applications mathématiques et scientifiques, nécessite une approche méthodique et structurée. La création de ce langage et de son compilateur passe par plusieurs étapes cruciales, allant de l'analyse du problème à la spécification fonctionnelle, puis à la conception préliminaire, et enfin au codage de l'application. Chacune de ces étapes joue un rôle essentiel dans la transformation de l'idée initiale en un produit fonctionnel et efficace.

3.1 Analyse du problème et spécification fonctionnelle

Le projet GroMath naît d'un constat simple : les outils actuels de programmation mathématique et scientifique ne répondent pas toujours aux besoins spécifiques des amateurs de cryptographie en CTF (compétition de cybersécurité), notamment en termes de performance, de flexibilité, et d'intégration de structures mathématiques avancées. L'analyse du problème a donc pour but de déterminer comment GroMath peut combler ces lacunes, en offrant un langage à la fois puissant et intuitif, capable de modéliser des structures mathématiques complexes tout en garantissant des performances de compilation optimales.

La spécification fonctionnelle découle directement de cette analyse. Elle détaille les caractéristiques uniques de GroMath, telles que la prise en charge de types de données avancés (groupes, anneaux, corps), la possibilité d'utiliser des foncteurs pour la manipulation de ces structures, et l'intégration d'optimisations de compilation pour faire gagner du temps à l'utilisateur, précieux en compétition. Cette spécification sert de feuille de route pour le développement, en définissant clairement les objectifs à atteindre et les fonctionnalités à implémenter.

3.2 Conception préliminaire

La phase de conception préliminaire transforme les exigences fonctionnelles en une architecture logicielle cohérente. Pour GroMath, cela implique de décider de la structure du langage, y compris sa syntaxe et sa sémantique, ainsi que de l'architecture du compilateur. Cette étape comprend la définition de la grammaire du langage, le choix des technologies de compilation (LLVM et GNU Bison), et la planification de l'analyse syntaxique, de l'analyse sémantique, et de la génération de code.

La conception préliminaire envisage également l'intégration des différentes composantes du compilateur pour assurer une traduction efficace du code source en code machine optimisé. Elle pose les bases de l'optimisation des performances, en identifiant les opportunités pour l'utilisation de techniques avancées permises par LLVM. Cette phase repose aussi sur le développement de compétences en

théorie des catégories pour atteindre un niveau d'abstraction qui permettrait d'apréhender sereinement l'implémentation de nos structures algébriques.

3.3 Codage

Le codage de l'application est l'étape où les concepts et plans établis lors des phases précédentes prennent vie. Pour GroMath, cela signifie implémenter la grammaire dans GNU Bison, développer l'analyseur syntaxique, et coder les routines d'analyse sémantique. Cette phase voit également la mise en œuvre de la génération de code intermédiaire LLVM, ainsi que l'intégration des optimisations de compilation.

Le codage doit être réalisé avec soin, en veillant à ce que chaque partie du compilateur fonctionne harmonieusement avec les autres et respecte les spécifications fonctionnelles. Des tests unitaires et d'intégration sont effectués tout au long de cette phase pour garantir la fiabilité et la performance du compilateur. L'objectif final est de produire un compilateur robuste, capable de traduire efficacement les programmes GroMath en code machine optimisé, ouvrant ainsi la voie à des applications mathématiques et scientifiques de nouvelle génération.

Chapitre 4

Conception détaillée

Pour ce qui est de la conception détaillée du projet, la création du langage s'est étalée sur plusieurs champs très diverses sur lesquels nous avons dû nous autoformer tout au long de l'année. L'idée initiale était un projet compilé LLVM, pour cela nous devions donc coder en C++ (qui est le langage majoritairement utilsé pour LLVM, une crate est en cours de développement en rust mais cela reste très anecdotique). N'en ayant jamais fait, nous nous sommes donc tout d'abord mis à expérimenter la programmation en C++.

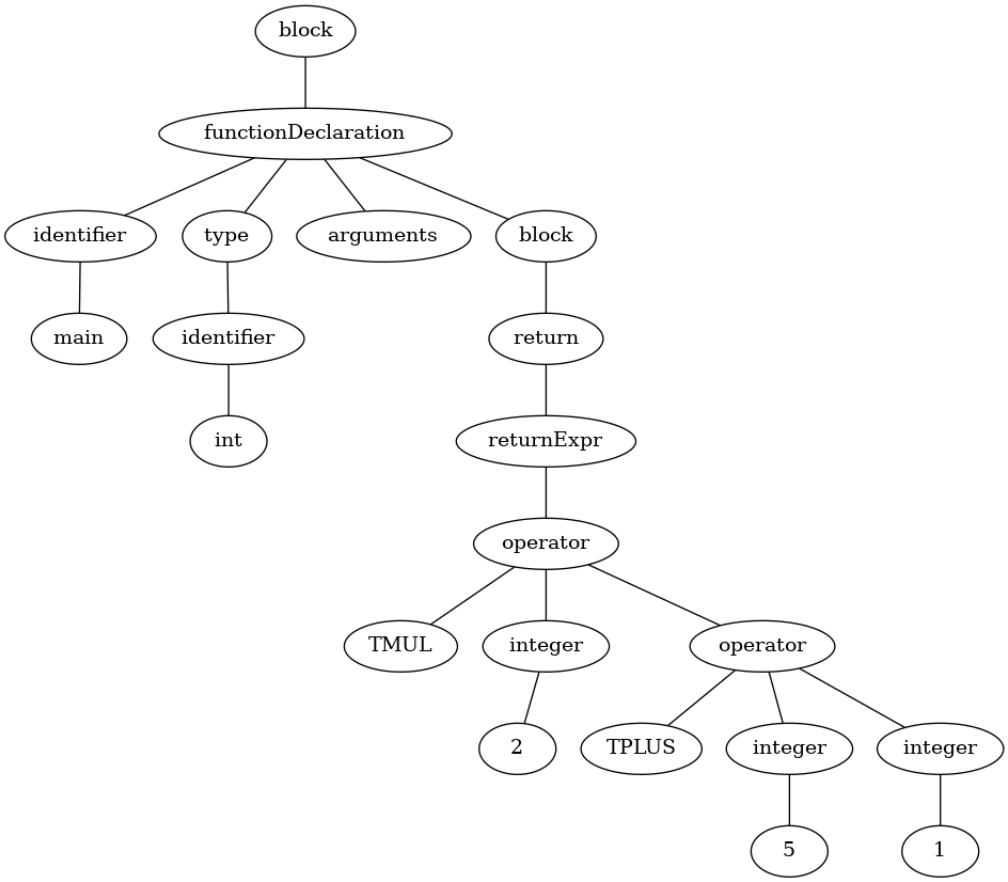
Après cela, nous avons enfin pu commencer réellement le projet. Initialement, le parsing était implémenté à la main (parsing LR) en C++, et pour ce qui est du reste, nous avions suivi le tutoriel fourni sur le site de LLVM, nommé Kaleidoscope et implémentant un langage fonctionnel relativement simple. Ce tutoriel était cependant très limité car il s'arrêtais assez rapidement et était écrit en LLVM 17, puis adapté en LLVM 19 au milieu du tutoriel. Tout cela ne nous a pas arrangé car on devait relativement tout intuiter : les explications n'étaient pas claires, le passage sous LLVM 19 non plus, et nous avons terminé l'implémentation sans réelle conviction de continuer sur cette lancée.

Le majeur problème lié à la documentation liée aux outils du projet était que, du fait qu'ils soient récents, nous n'avions pas de réelle base sur laquelle travailler ou bien les ressources en ligne étaient payantes. Après plusieurs mois de remise en question, nous avons finalement décidé sur la période décembre d'investir dans l'une d'entre-elles : celle de Dmitry Soshnikov, intitulée **programming language with LLVM**. Cette formation nous a permis de grandement avancer du fait de sa grande richesse : elle abordait tous les points clés pour l'implémentation d'un compilateur pour un langage rudimentaire. Le seul point de doute restait le parsing.

Pour ce qui est du parsing, après de lourdes batailles contre Bison nous avons finalement réussi à mettre en place une grammaire qui nous semblait pertinente en vue de nos expériences passées en programmation (en Haskell, en OCaml et en C majoritairement). La tâche la plus lourde était la désambiguification de cette grammaire mais il se trouve que du fait de l'implémentation pratique de Bison, il était possible d'écrire des règles pour qu'en pratique Bison effectue un parsing correct selon nos spécifications.

Après tout cela, il s'agissait désormais de debugger notre code. Pour cela, nous avons mis en place plusieurs stratégies : tout d'abord des tests unitaires ont été mis en place sur les parties du projets qui étaient pertinentes à tester de cette manière mais passé cette étape il était très difficile de tester chaque partie du code, nous avons donc décidé de mettre en place des moyen d'afficher les objets intermédiaires que l'on générait notamment l'AST (Abstract Syntax Tree) lié au parsing d'une fonction.

Pour l'affichage des AST, nous avons utilisé Python avec la librairie GraphViz, ce qui nous a permis d'afficher des graphes tels que celui-ci :



Finalement, il nous était possible de naviguer via les output LLVM dans la représentation intermédiaire LLVM qui s'apparente à de l'assembleur :

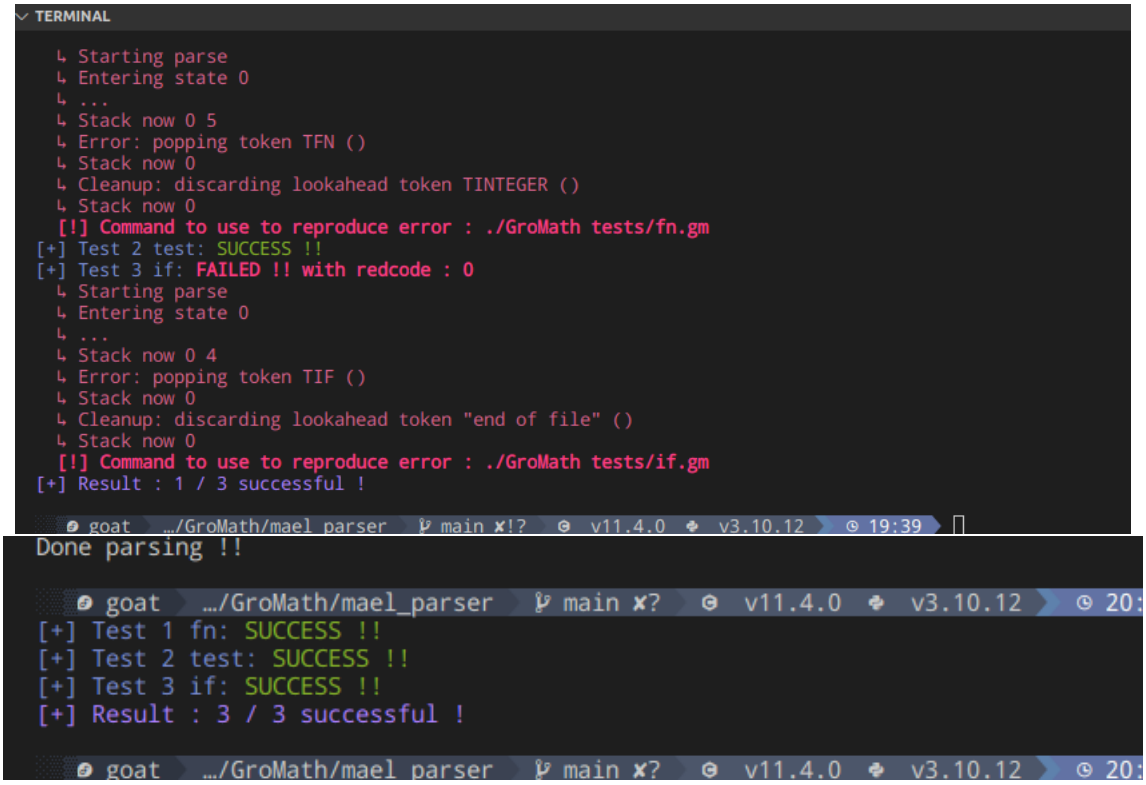
```
; ModuleID = 'GroMathLLVM'
source_filename = "GroMathLLVM"

@VERSION = global i32 1, align 4
@0 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

declare i32 @printf(ptr, ...)

define i32 @main() {
entry:
  %i = alloca i32, align 4
  store i32 0, ptr %i, align 4
  br label %cond

cond:                                             ; preds = %body, %entry
  %0 = load i32, ptr %i, align 4
  %1 = icmp slt i32 %0, 5
  br i1 %1, label %body, label %loopend

body:                                             ; preds = %cond
  %i1 = load i32, ptr %i, align 4
  %2 = call i32 (ptr, ...) @printf(ptr @0, i32 %i1)
  %3 = load i32, ptr %i, align 4
  %4 = add i32 %3, 1
  store i32 %4, ptr %i, align 4
  br label %cond

loopend:                                          ; preds = %cond
  ret i32 0
}
```

Chapitre 5

Résultat des tests unitaires, d'intégration et de validation

Pour ce qui est des tests unitaires, cela a déjà été évoqué au dessus, mais le code permettant de tester notre projet a été lui-même testé (cela est visible dans le dossier tests du projet).

Il générait ainsi les AST et nous permettait de voir l'étape du parsing. Pour ce qui est ensuite de la compilation, la génération de tests unitaire s'avérait très compliqué (malgré la lecture de multiples papier sur la génération de code à partir d'une grammaire), nous avons donc décidé de générer plusieurs snippets de code écrits dans notre langage (fichiers en .gm) puis nous avons testé ces fichiers au fur à mesure de l'avancement du projet en essayant chaque feature une à une après son implémentation.



Chapitre 6

Histoire du projet

6.1 Gestion du projet

Le projet est très dense, conplexe et est constitué de multiples composantes, on distingue : l'analyse lexiale, l'analyse syntaxique, la vérification la génération (vers LLVM) et l'optimisation. Mais il y a aussi une composante très importante que nous avons initialement négligé : la simplicité de débogage. En effet nous avons dû passer un temps significatif à implémenter la conversion vers json et puis la représentation graphique afin d'avoir une première manière concrète de débugger les données et donc de rédiger des tests unitaire. Le projet a connu plusieurs révisions (trouvable dans _deprecated sur github), la partie mathématique du projet à donc techniquement été abordée mais n'est pas encore intégré avec LLVM.

Il est important de noter que nous avons aussi contribué financièrement au projet à hauteur de 50€ pour avoir de la documentation à jour sur LLVM. Les différentes révisions du projet nous ont permis d'avancer sur des bases solides.

6.2 Technologies du projet

Le projet se base principalement sur LLVM et est écrit en C++, bison, flex et en python pour la partie débugging. LLVM est une grosse librairie pour faciliter la création de compilateur quasi universel (autrement dit qui ont la capacité de compiler vers plusieurs plateforme), LLVM correspond à l'étape finale de la compilation et gère aussi les passes d'optimisation.

Comme énoncé précédement le C++ est plus ou moins imposé et nous avons longtemps hésité à utiliser ou non bison/flex, nous avons même écrit un parser LL + lexer entièrement à la main mais pour des raisons de flexibilité nous avons finalement opté pour flex/bison (car nous avons trouvé une très bonne ressource en ligne sans laquelle l'intégration de bison aurait été très complexe).

La branche de développement interne est en train d'intégrer la librairie NTL (number theoretic library) pour permettre le calcul avec les grands nombres. Cet aspect est de loin le plus difficle à mettre en place pour des raisons techniques que nous détaillerons ci-dessous.

6.3 Les problèmes

6.3.1 Problèmes rencontrés

Le principal problème que nous n'avons pas encore résolu provient comme cité plus tôt de la difficulté à inclure l'utilisation de grands nombres qui sont absolument nécessaire à l'utilisation cryptographique. Le problème vient du fait que les grands nombres par définition peuvent s'étendre en taille et doivent donc être des objets alloués sur la heap (le tas) et cela sous entend aussi soit un mécanisame de suivi constructeur/destructeur couplé à de multiples optimisation difficiles à mettre en place (comme les compilateurs C++) soit la mise en place d'un garbage collector (glaneur de cellule).

Notre plus gros problème dans le projet était la difficulté à débugger, le manque de documentation sur les technologies utilisées et la difficulté de faire articuler ces différentes technologies entre elles.

6.3.2 Solutions

Pour ce qui est de la difficulté à debug, la possibilité d'afficher l'arbre en JSON puis graphiquement nous as bien aidé, nous avons aussi dû appréhender la lecture d'output de debug de Bison (les logs de l'automate ne sont pas très intuitifs à première vue mais se sont avérés très très utiles...). La difficulté de documentation à été résolue par ... la formation payante et beaucoup de recherche et de discussion avec des contributeurs LLVM (notamment sur discord).

Je tiens à remercier tout particulièrement Antoine MORRIER (ancien président club code) sans qui le projet n'aurais jamais vu le jour, des heures entières (5 heure de train...) il nous a aidé à apprendre des notions de C++ et à régler pleins de problèmes.

Chapitre 7

Conclusion

Le language est fonctionnel (enfin impératif mais il marche :P), il manque encore d'intégration de fonctionnalités plus poussées mais nous pensons que cela pourrait faire l'objet d'un projet cassiopé. Le plus gros de ce qu'il reste à faire n'est pas vraiment intéressant à proprement dit dans le cadre de la création d'un language et serait plutôt de petites améliorations de qualité de vie pour le programmeur, notamment du syntax highlighting (que nous avons commencé à faire pour VS-Code mais cela requiert de réécrire la grammaire dans un nouveau format...) et du type checking mais cela pourrait faire l'objet d'un projet de thèse, nous avons donc évité de le faire.

Chapitre 8

Manuel utilisateur

Pour ce qui est de l'installation des différents outils nécessaires à l'utilisation du projet, un fichier requirements.txt :

```
# for archive signature : wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key | sudo ap

clang-19
lldb-19
lld-19

libllvm-19-ocaml-dev
libllvm19
llvm-19
llvm-19-dev
llvm-19-doc
llvm-19-examples
llvm-19-runtime

clang-tools-19
clang-19-doc
libclang-common-19-dev
libclang-19-dev
libclang1-19
clang-format-19
python3-clang-19
clangd-19
clang-tidy-19

libllvmlibc-19-dev

flex
bison

jq
```

Il est aussi possible de passer par LLVM-17 pour les utilisateurs de l'AUR. De plus, pour ce qui est de l'affichage des ASTs, la librairie python GraphViz est nécessaire.

Passé cette étape, des fichiers build.sh ont été mis en place pour, suite à l'écriture d'un snippet de code dans test_llvm.gm, il soit possible de compiler le code. Après cela, il suffit d'exécuter le code python pour voir l'affichage de l'AST généré.

Chapitre 9

Ressources

— Formation LLVM
— Tutoriel LLVM officiel
— Tutoriel pour un language flex/bison basé sur LLVM
— Documentation LLVM
— Le github de Qnope (Antoine MORRIER)
— La documentation NTL
— Le manuel GMP
— Une documentation supplémentaire pour LLVM
— GraphViz