Вінницький національний технічний університет
Факультет комп’ютерних систем та автоматики
Кафедра автоматизації та інтелектуальних інформаційних технологій

Пояснювальна записка

до бакалаврської дипломної роботи

бакалавр
(освітньо-кваліфікаційний рівень)

на тему: Розробка модулів JIT компілятора шейдероподібної мови програмування

Виконав: студент 4 курсу, групи 1СІ-16б
Спеціальність 151 – Автоматизація та комп’ютерно-інтегровані технології
Кулик М. С.

Керівник: к.т.н., доцент каф. МПА
Севастьянов В. М.

Рецензент: _____

Вінниця ВНТУ – 2020 року

Вінницький національний технічний університет

Факультет комп'ютерних систем і автоматики

Кафедра автоматизації та інтелектуальних інформаційних технологій

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 151 – «Автоматизація та комп'ютерно-інтегровані технології»

ЗАТВЕРДЖУЮ
Завідувач кафедрою АІІТ
д.т.н., проф. Квєтний Р. Н.
«___»_____ 202_ р.

ЗАВДАННЯ

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Кулику Максиму Сергійовичу

1. Тема роботи: Розробка модулів JIT компілятора шейдероподібної мови програмування
   Керівник роботи: к.т.н., доцент каф. МПА Севастьянов В. М.
   Затверджені наказом ВНТУ від «__»_____ 202_ року №___
2. Строк подання студентом роботи: «___» _____ 202_ р.
3. Вихідні дані до роботи: код програми, мова програмування *C++*.
4. Зміст розрахунково-пояснювальної записки: огляд предметної області; розробка структури бібліотеки; розробка програмного забезпечення.
5. Перелік графічного матеріалу: UML діаграма ПО; алгоритми; лістинг програмного забезпечення.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання отримав
1-3	Бевз О. М., к.т.н., доцент каф. АІІТ		

7. Дата видачі завдання: «___»_____ 202_ р.

КАЛЕНДАРНИЙ ПЛАН

№ з	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд предметної області		
2	Розробка структури бібліотеки		
3	Розробка програмного забезпечення		
4	Оформлення пояснювальної записки і графічного матеріалу		
5	Попередній захист роботи		
6	Остаточний захист роботи		

Студент _____ Кулик М. С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Севастьянов В. М.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

У даній роботі розглядається задача JIT компіляції шейдероподібної високорівневої мови програмування. Для її розв’язання використано LLVM. Розроблено специфікацію мови, алгоритмічне та програмне забезпечення, яке дозволяє виконати поставлену задачу.

ABSTRACT

In this work the task of JIT compiling shader like high level language has been considered. To solve it, LLVM compiler framework is used. The language specification, an algorithms and software that allows to solve the problem has been developed.

ЗМІСТ

ВСТУП

Актуальність. З розвитком цифрових технологій збільшується швидкість та кількість обчислювальних ядер у архітектурах мікропроцесорів, а також збільшується кількість наборів інструкцій. Зокрема збільшується кількість наборів так званих векторних інструкцій при яких у векторний регістр пакується набір величин та операції проводяться над усіма величинами одразу що дозволяє значно збільшити пропускну обчислювальну здатність [1, 2]. Останнім нововведенням є набір AVX-512 що дозволяє оперувати регістрами вмістом 512 біт, тобто, для прикладу, виконувати операції одночасно на шістнадцяти тридцяти двох бітних величинах [3]. Проте у використанні знаходиться широкий спектр мікропроцесорних архітектур з різною підтримкою наборів інструкцій, тому у разі використання AOT компільованих мов програмування виникає проблема мінімально підтримуваних наборів інструкцій. На даний момент мінімальною вважається наявність інструкцій SSE2 введених Intel у 2000 році, тому при компіляції ПЗ для широкого спектру користувачів інструкції сімейства AVX не використовуються.

JIT компіляція фундаментально уникає цієї проблеми адже процес компіляції відбувається на комп’ютері кінцевого користувача ПЗ [4].

При застосуванні C++ для прискорення обчислень використовується паралелізація коду та GPGPU моделі обробки що використовують графічні АПІ та шейдерні програми які також JIT компілюються графічним драйвером та виконуються на GPU [5]. Проте використання цих АПІ пов’язано зі складнощами, а застосування GPU не завжди є доцільним або можливим, що підводить до JIT компілятора шейдероподібної мови програмування з повною оптимізацією при компіляції у машинний код що виконується на CPU з можливістю вбудовування у будь який проект на мові C++.

Метою роботи є розробка програмного забезпечення у вигляді бібліотеки для JIT компіляції шейдероподібної мови програмування.

Для досягнення мети необхідно розв’язати такі *задачі*:

– розробити мову програмування;

– розробити компілятор розробленої мови програмування;

– розробити систему класів що дозволить використовувати компілятор у вигляді бібліотеки.

Об’єктом дослідження є процес компіляції.

Предметом дослідження є принципи та методи компіляції.

Методи дослідження. При виконанні роботи використовувалися такі методи досліджень: методи аналізу та синтезу для розробки шейдероподібної мови програмування та модулів компілятора, зокрема структури AST дерева. Основним засобом для розробки програмного забезпечення є мова програмування C++

Науково-технічний результат роботи полягає у тому, що розроблено шейдероподібну мову програмування, алгоритми її токенізації та парсингу.

Практична цінність роботи полягає у тому, що розроблено програмне забезпечення для JIT компіляції розробленої мови.

Апробація результатів та публікації. За результатами даної роботи опубліковано тези доповіді [6] на XLIX науково-технічній конференції Факультету комп'ютерних систем і автоматики (м. Вінниця, 2020).

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

Мови програмування - позначення для опису обчислень для людей та машин. Світ, як ми знаємо, залежить від мов програмування, тому що все програмне забезпечення, яке працює на всіх комп’ютерах, було написане мовою програмування. Але, перш ніж програма може бути запущена, її спочатку потрібно перевести у форму, в якій її може виконати комп'ютер. Програмні системи, які роблять цей переклад, називаються компіляторами.

Простіше сказати, компілятор - це програма, яка може читати програму однією мовою - мовою джерела - і перевести її в еквівалентну програму іншою мовою - цільовою мовою Важлива роль компілятора - повідомляти про будь-які помилки у вихідній програмі, які він виявляє в процесі перекладу [7].

1.1 Структура компілятора

Фази компіляції можна розбити на дві основні частини – аналіз та синтез. Під час аналітичної фази програма на мові джерела розбивається на складові частини та формує їх у граматичну структуру. Надалі ця структура використовується для створення проміжного представлення, на далі IR (Intermediate Representation), вхідної програми. Також під час цієї фази відбувається заходження синтаксичних та синтаксичних помилок та їх представлення користувачеві у вигляді повідомлень для коригування. Під час цієї фази також відбувається заповнення таблиці символів (symbol table) яка разом з IR передається у фазу синтезу.

Під час синтезу відбувається конструювання цільової програми з IR та таблиці символів. Часто фазу аналізу називають фронтендом (front end), а фазу синтезу – бекендом (back end).

Якщо розглянути процес компіляції детальніше то від буде представлений у вигляді набору фаз кожна з яких перетворює одне представлення вхідної програми у інше. Типова декомпозиція процесу компіляції наведена на рисунку 1.1.

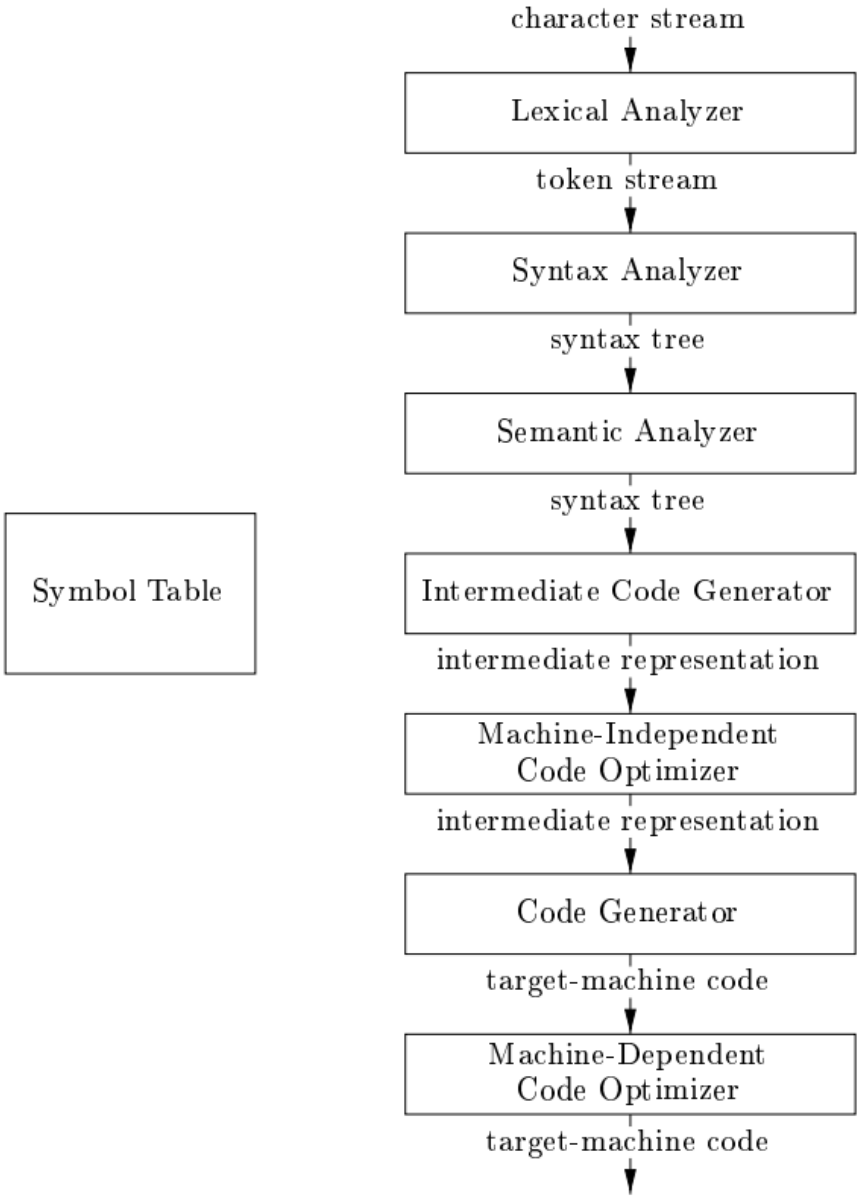

Рисунок 1.1 – Фази компіляції [7]

На практиці певні фази можуть бути згруповані а проміжне представлення не обов’язково має бути явно сформоване. Таблиця символів містить інформацію про

усю вхідну програму та використовується та використовується у всіх фазах компіляції [8].

Деякі компілятори мають фазу машинно-незалежної оптимізації яка полягає у перетворенні IR для того щоб згенерувати кращий результуючий машинний код ніж з неоптимізованої IR. Робота кожної з фаз проілюстрована на прикладі компіляції виразу присвоєння (рисунок 1.2).

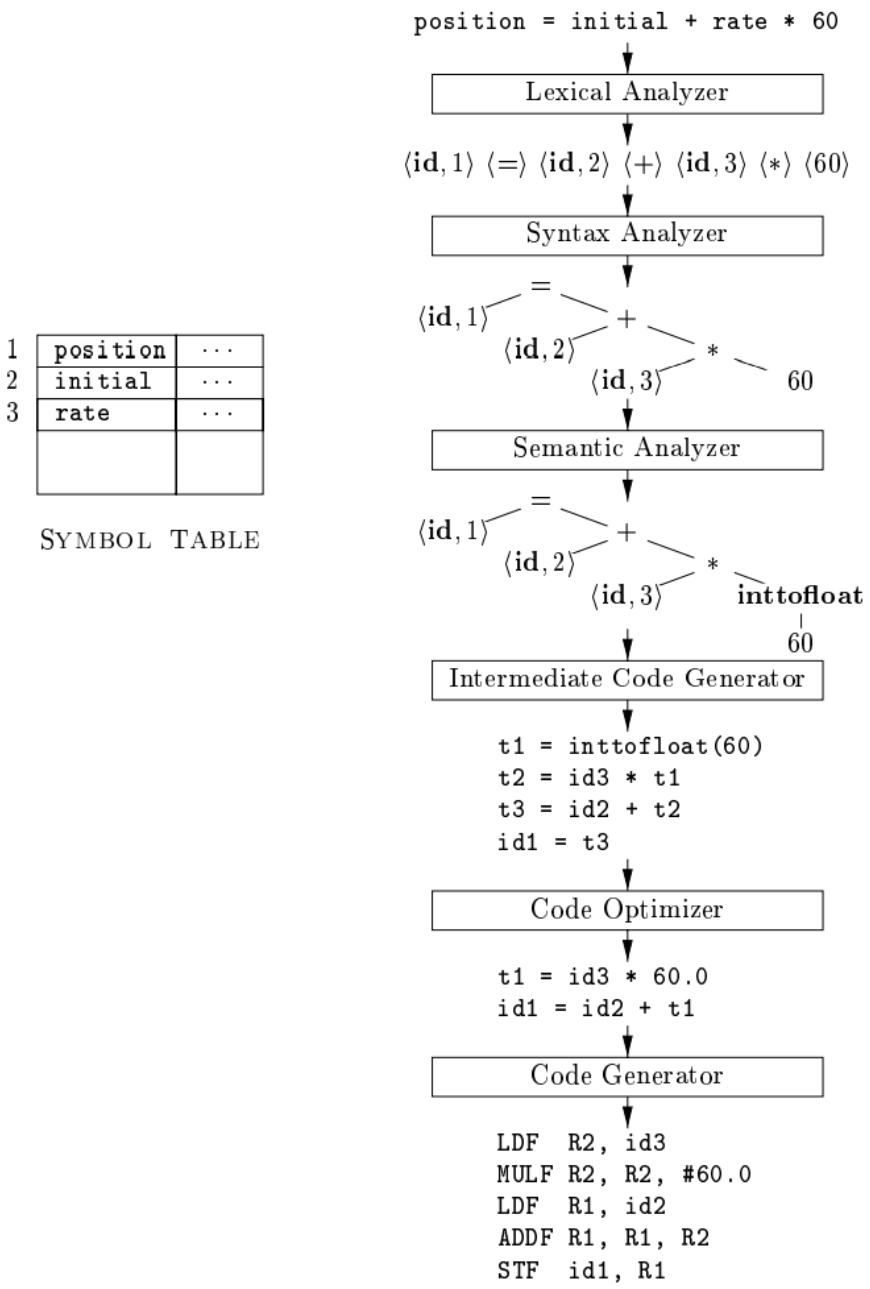

Рисунок 1.2 – Компіляція виразу присвоєння

1.1.1 Лексичний аналізатор (Lexical Analyzer)

Першою фазою компілятора є лексичний аналіз або сканування. Лексичний аналізатор читає потік символів вхідної програми та групує їх у змістовні послідовності які називаються лексемами. Таблиця символів заповнюється лексемами. Лексеми передаються у синтаксичний аналізатор у вигляді токенів які містять ім’я токену що відповідає типу лексеми та посилання на лексему у таблиці символів [8, 9, 10, 11]. Приклад розкладу на токени зображений на рисунку 1.2.

1.1.2 Синтаксичний аналізатор (Syntax Analyzer)

Друга фаза компіляції це синтаксичний аналіз або парсинг (parsing). Парсер використовує токени отримані від лексичного аналізатора для створення структури даних по типу бінарного дерева що називається абстрактним синтаксичним деревом чи AST (abstract syntax tree) та є представленням граматичної структури потоку токенів. У звичайному синтаксичному дереві кожен вузол представляє собою операцію а похідні – операнди [9, 12, 13]. Приклад розкладу на синтаксичне дерево зображений на рисунку 1.2.

1.1.3 Семантичний аналізатор (Semantic Analyzer)

Семантичний аналізатор використовує синтаксичне дерево та таблицю символів для перевірки програми на семантичну (змістову) відповідність до визначення вхідної мови програмування. Семантичний аналізатор також збирає інформацію про типи та зберігає її синтаксичному дереві чи таблиці символів для

використання на фазі генерації проміжного коду. Важливою частиною семантичного аналізатора є перевірка типів на відповідність. Специфікація мови може дозволяти перетворення типів які називають примусовими (coercion), тобто непрямі перетворення типів [9]. Приклад примусового перетворення типів зазначений на рисунку 1.2.

1.1.4 Генератор проміжного коду (Intermediate Code Generator)

У процесі перетворення вхідної програми у цільовий код компілятор може сконструювати одну або декілька IR які можуть мати різні форми. Синтаксичне дерево є прикладом IR. Після семантичного аналізу зазвичай компілятори генерують явну низькорівневу машиноподібну IR яка являє собою програму для абстрактної обчислювальної машини. Це представлення має бути легко генерованим та легко претворюваним у код для цільової обчислювальної машини [7]. Приклад генерації такої IR наведено на рисунку 1.2.

1.1.5 Оптимізатор коду (Code Optimizer)

Фаза оптимізації машино-незалежного коду намагається покращити IR для генерації кращого машинного коду. У цьому контексті кращого зазвичай означає швидшого, проте можуть переслідуватись інші цілі такі як код меншого об’єму чи код який потребує менших затрат енергії для виконання. Оптимізатор може вивести що деякі частини коду можна виконати під час компіляції один раз, або прибрати непотрібні операції присвоєння, замінити одні інструкції на інші, а у випадку розгалуження прибрати недоступний код [7]. Приклад оптимізації IR наведено на рисунку 1.2.

Існує велика кількість можливих оптимізацій, зокрема векторизація, та розгортання циклу. Векторизація - це заміна послідовності простих однакових інструкції на одну інструкцію з використанням векторних регістрів.

Розгортання циклу – це заміна циклу на набір інструкції що будуть виконані без циклу, це дозволить уникнути операцій передачі управління та перевірки індексу ітерації. Зазвичай використовується коли довжина циклу відома на етапі компіляції, проте у деяких випадках така оптимізація можлива шляхом заміни тіла циклу на набір інструкцій що будуть виконувати N ітерацій початкового циклу за один цикл оптимізованого [14].

1.1.6 Генератор коду (Code Generator)

На цій фазі використовується оптимізоване IR вхідної програми яке ставиться у відповідність з цільовою мовою. Якщо цільова мова – машинна то обираються регістри чи регіони пам’яті для кожної із змінних у програмі, потім інструкції IR перетворюються у послідовності машинних інструкцій що виконують ту саму задачу. Важливий аспект генерації коду це доцільне виділення регістрів під змінні [9]. Приклад генерації машинного коду з IR наведено на рисунку 1.2.

1.2 JIT компіляція

Принципіальна різниця між AOT та JIT компіляцією полягає у останній фазі компіляції. AOT компілятор записує результуючий машинний код у об’єктні файли чи файли бібліотек для послідуючого статичного чи динамічного лінкування (linking) чи у виконуваний файл для послідуючого виконання на комп’ютері користувача. JIT компілятор записує результуючий код у активну пам'ять з дозволом виконання, та, залежно від типу JIT компілятора, передає управління

точці входу чи надає сторонній програмі доступ до відкомпільованих символів (функції) для послідуючого їх виклику. Тобто фактично процес JIT компіляції вхідної програми включає у себе AOT компіляцію, програмно ці фази можуть бути як об’єднані так і розділені.

Інші відмінності залежать від призначення JIT компілятора. Наприклад, для зменшення використовуваної пам’яті використовують ліниву компіляцію де символ не компілюється доки не буде використаний, а для відкомпільованих символів реалізується механіка витіснення з пам’яті на основі частоти використання. Для отримання швидкого реагування реалізують декілька шляхів компіляції з різними ступенями оптимізації, що дозволяє значно зменшити час першої компіляції символу, та підвищити рівень оптимізації залежно від частоти використання. Також може використовуватися інтерпретатор для виконування коду до завершення компіляції. Існують різні стратегії JIT компіляції, що часто можуть поєднуватись на практиці [15].

1.3 Огляд аналогів

JIT компіляція шейдероподібних мов для CPU використовується досить вузько, із відомих з відкритим кодом є Skia – бібліотека для графічного інтерфейсу та 2D графіки у цілому, проте місцевий JIT компілятор є частиною бібліотеки та окремо не використовується. Є MPSL який виступає в якості натхнення для цієї роботи, проте сам проект не завершений тому оцінити його важко. Порівняння з іншими JIT компіляторами мов таких як Lua, C#, JavaScript не є доцільним адже поставлена задача значно відрізняється, вони можуть виконувати поставлену задачу за певних модифікацій чи параметризації, проте не є аналогами. Порівняння з JIT компіляторами для GPU також не є доцільним.

1.4 Висновки до розділу

У даному розділі проведено огляд ключових компонентів та фаз процесу компіляції. Компіляція поділяється на дві основні послідовні частини – аналіз та синтез, які в свою чергу поділяються на набір послідовних фаз, кожна з яких перетворює вихідні дані попередньої фази. Оскільки для вирішення поставленої задачі у пріоритеті знаходиться швидкість обчислень відкомпільованої програми, то для побудови JIT компілятора необхідно лише додати фазу запису цільового коду у активну пам'ять.

Таким чином поставлену задачу розбиваємо на написання AOT компілятора та написання JIT модуля.

2 РОЗРОБКА СТРУКТУРИ БІБЛІОТЕКИ

Перелічуваний тип даних Token містить список можливих токенів: tok_eof, tok_type, tok_extern, tok_identifier, tok_number, tok_if, tok_else, tok_for.

Клас Tokenizer відповідає за токенізацію вхідної програми (рисунок 2.1).

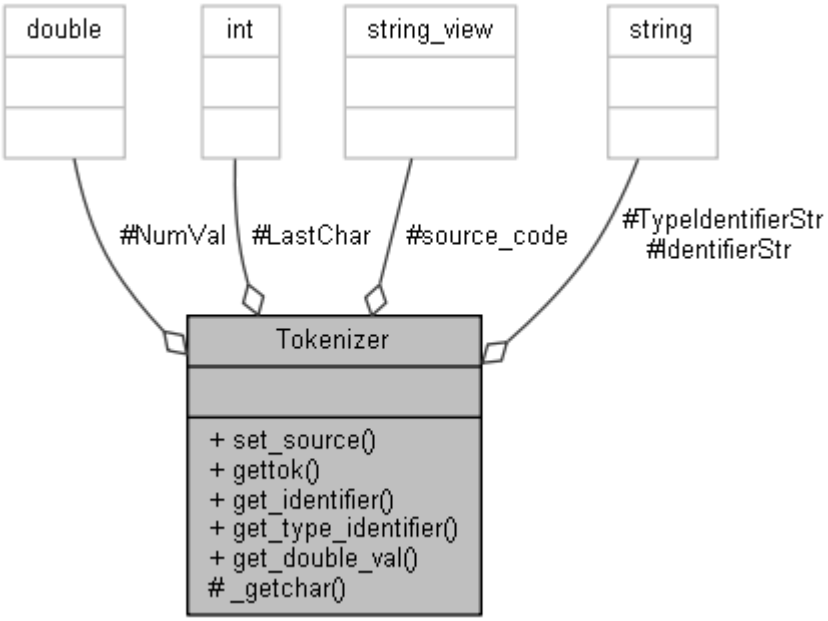

Рисунок 2.1 – Діаграма класу Tokenizer

Клас Parser використовує Tokenizer для отримання токенів, виділяє атрибути та генерує AST на основі отриманих токенів та атрибутів, заповнює таблицю символів. Parser має внутрішній буфер токенів у вигляді змінної CurTok, та зберігає останній ідентифікатор та останній ідентифікатор типу для обробки. Обробка програми є послідовною та посимвольною що дозволяє ефективно повідомляти про помилики (рисунок 2.2).

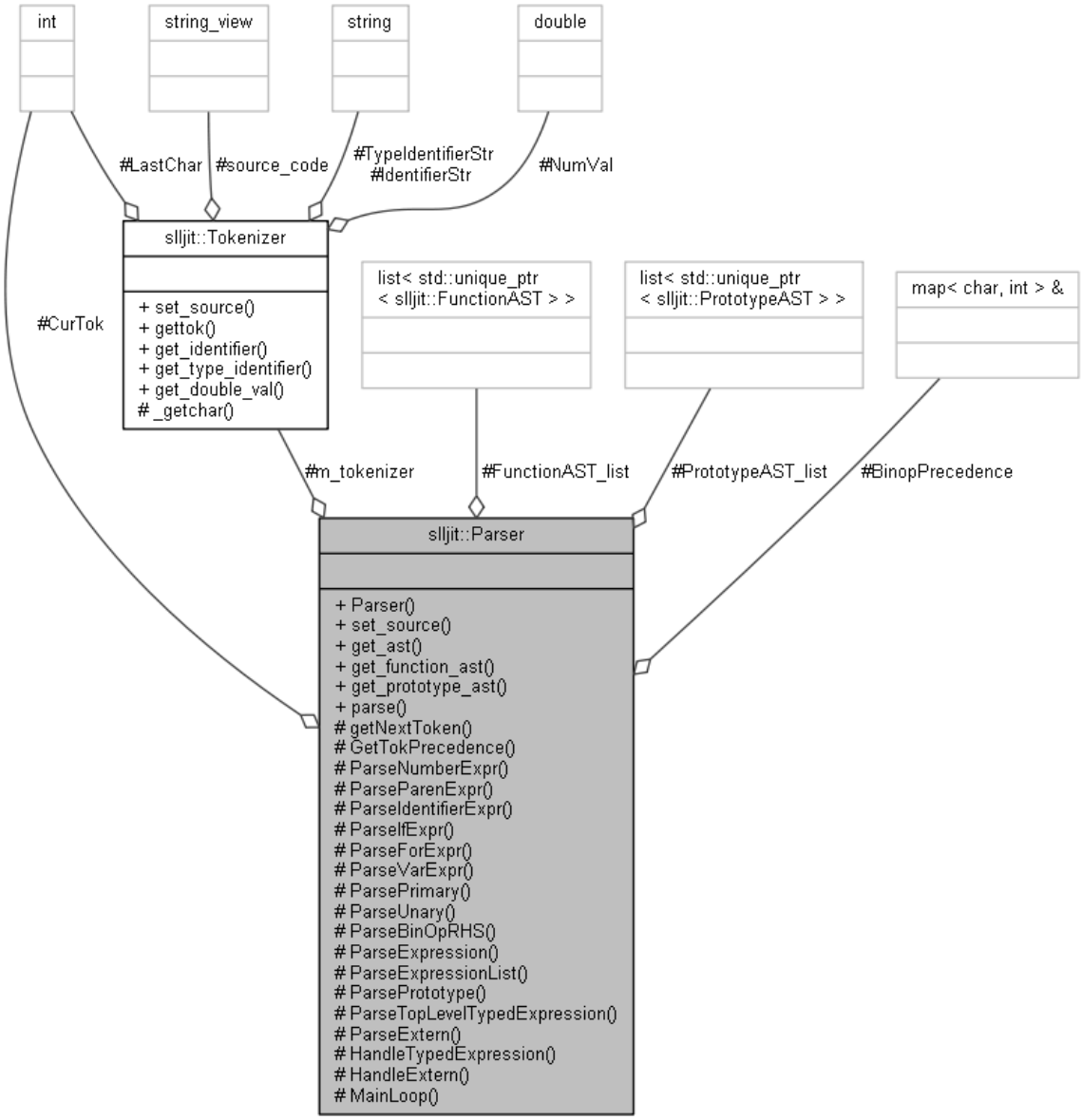

Рисунок 2.2 – Діаграма класу Parser

AST представлено набором класів. ExprAST – абстрактний базовий клас який наслідують усі інші AST класи. NoOpAST – клас що описує пусту операцію(nop). NumberExprAST – клас що описує числову константу. UnaryExprAST – описує унарний вираз. BinaryExprAST – бінарний оператор. VarExprAST – вираз оголошення змінної. VariableExprAST – посилання на змінну. PrototypeAST – прототип функції. FunctionAST – функція. CallExprAST – виклик функції.

IfExprAST – розгалуження з умовою. ForExprAST – цикл. Перелічені класи представлені у вигляді діаграми на рисунку 2.3.

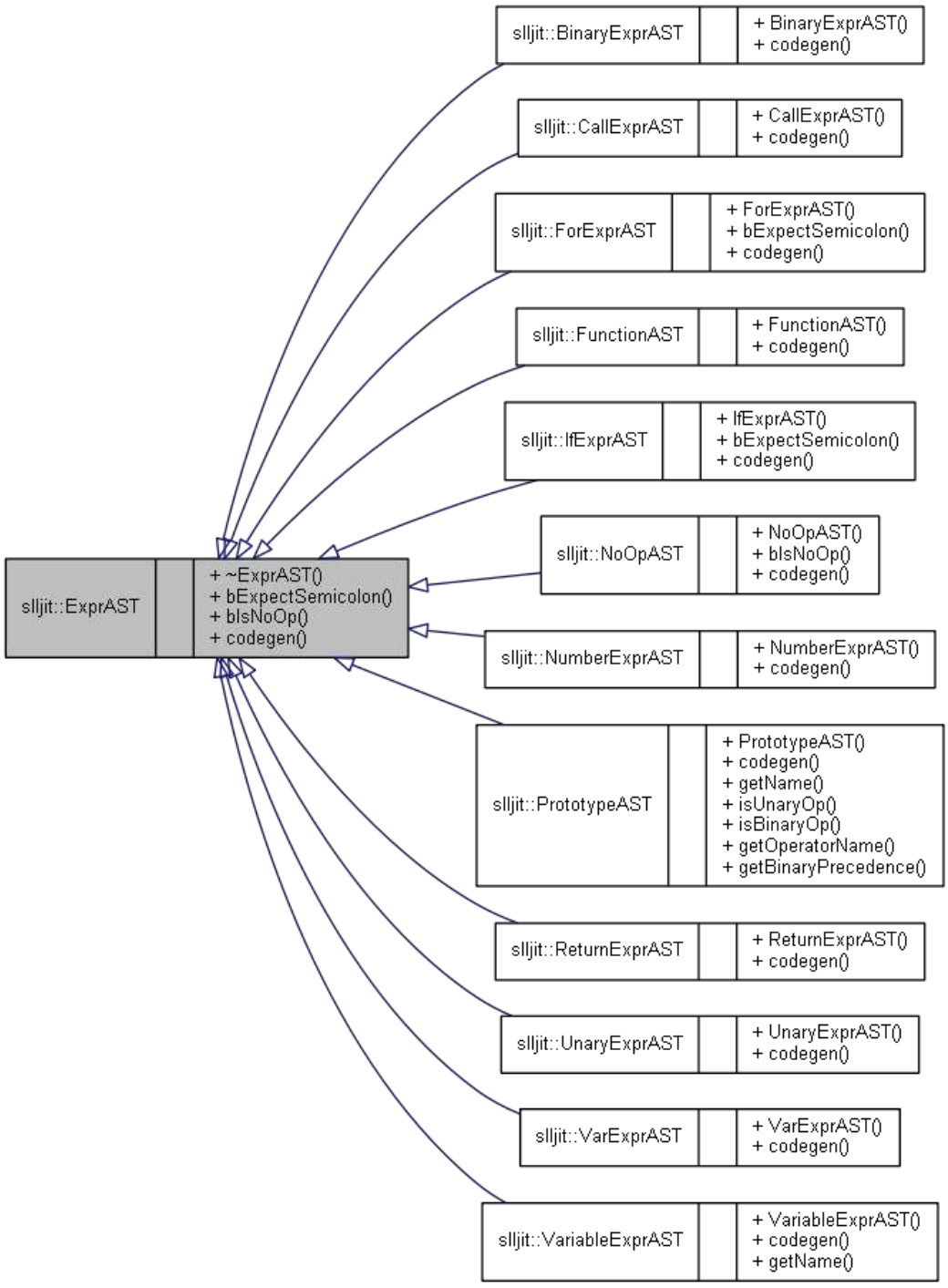

Рисунок 2.3 – Діаграма AST класів

Клас CodeGen (рисунок 2.4) виконує генерацію LLVM IR [16] на основі AST, виконує оптимізацію та генерацію машинного коду з LLVM IR. До оптимізацій входить переміщення змінних зі стеку у регістри, спрощення виразів, злиття інструкцій, спрощення структури розгалуження, векторизація, злиття операцій завантаження та вивантаження пам’яті.

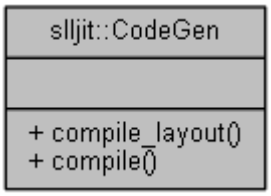

Рисунок 2.4 – Діаграма класу CodeGen

Клас ShaderJIT (рисунок 2.5) виконує оптимізацію та генерацію машинного коду з LLVM IR. Цей клас також надає доступ до символів що вже були відкомпільовані, та надає управління модулями що дозволяє звільняти пам’ять при відсутності потреби у певному модулі.

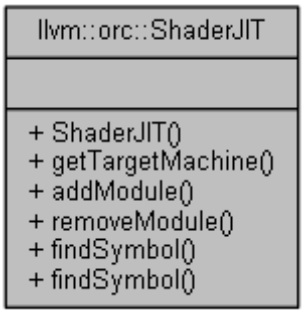

Рисунок 2.5 – Діаграма класу ShaderJIT

Перелічені класи є внутрішніми класами бібліотеки проте за необхідності можуть використовуватися.

Клас Context містить інформацію про контекст виконання та JIT компіляції коду (рисунок 2.6).

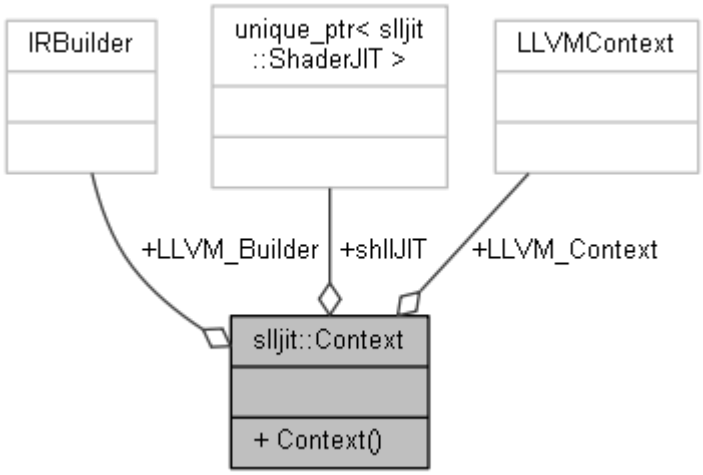

Рисунок 2.6 – Діаграма класу Context

Клас LocalContext (рисунок 2.7) описує контекст конкретної відкомпільованої програми. Об’єкт цього класу зберігає інформацію про модуль, прототипи функцій, змінні та оператори.

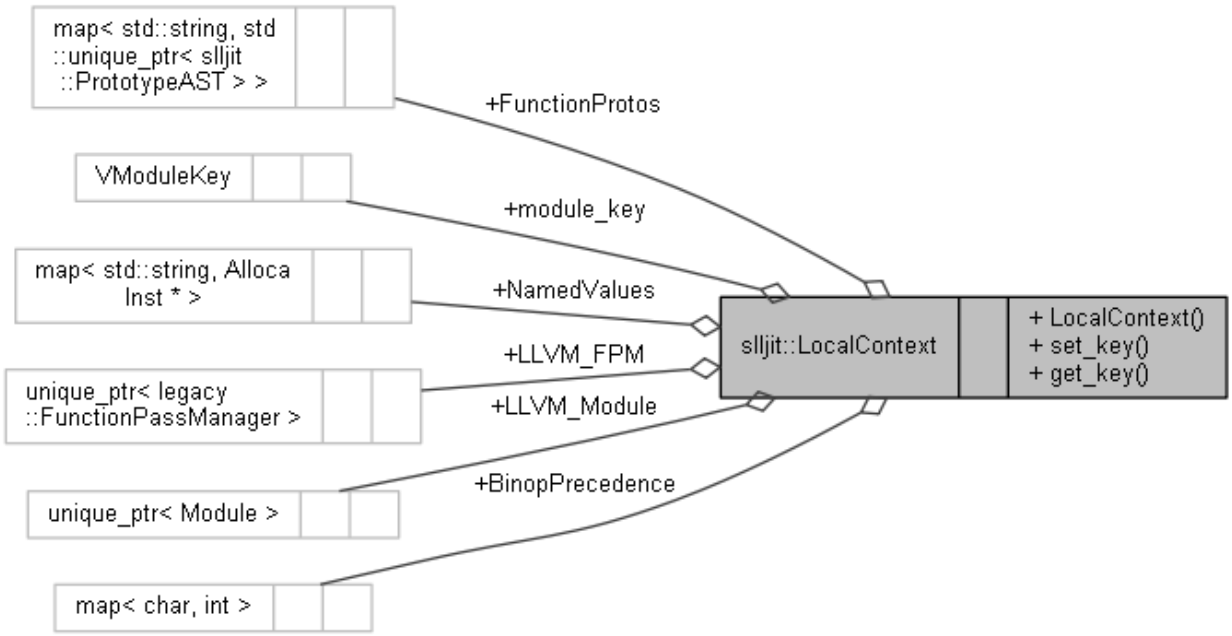

Рисунок 2.7 – Діаграма класу LocalContext

Клас Layout (рисунок 2.8) використовується для створення макету даних для розміщення глобальних змінних модуля. Дозволяє описувати константи які генеруються під час компіляції, та замінні які заповнюються при запуску відкомпільованої функції.

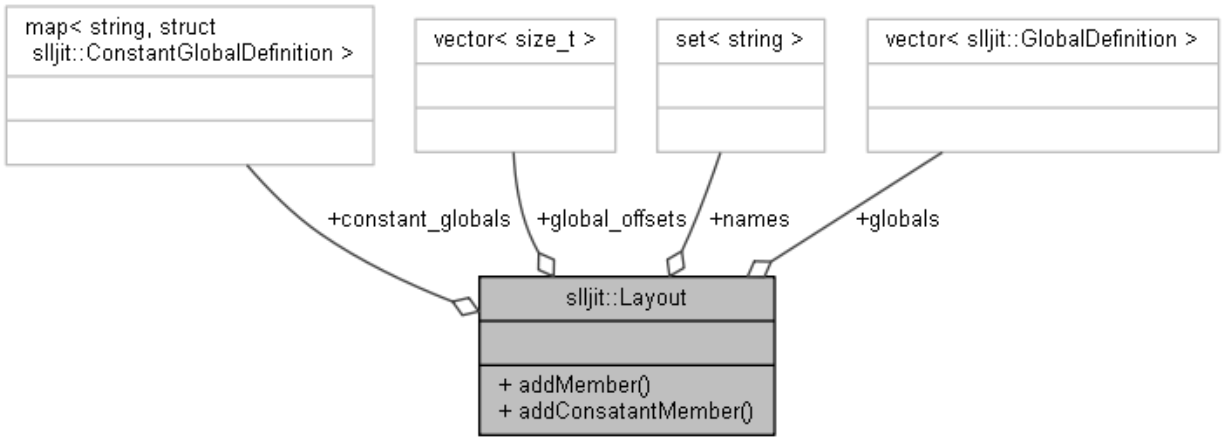

Рисунок 2.8 – Діаграма класу Layout

Клас Program (рисунок 2.9) створює модуль у вказаному контексті з вказаним макетом даних, після чого компілює програмно наданий код у цей модуль. Після компіляції клас Program використовується для запуску відкомпільованої програми.

Клас Program є шаблонним класом шо інстанціюється з типом T який може бути будь якою структурою даних що задана користувачем. Ця структура має бути описана користувачем за допомогою класу Layout. Після компіляції при використанні функції run у неї передається вказівник на об’єкт/структуру типу T. Програма може модифікувати елементи структури під час роботи, таким чином структура типу T слугує буфером даних у якому можуть бути початкові дані та у який у результаті роботи можна записати вихідні дані.

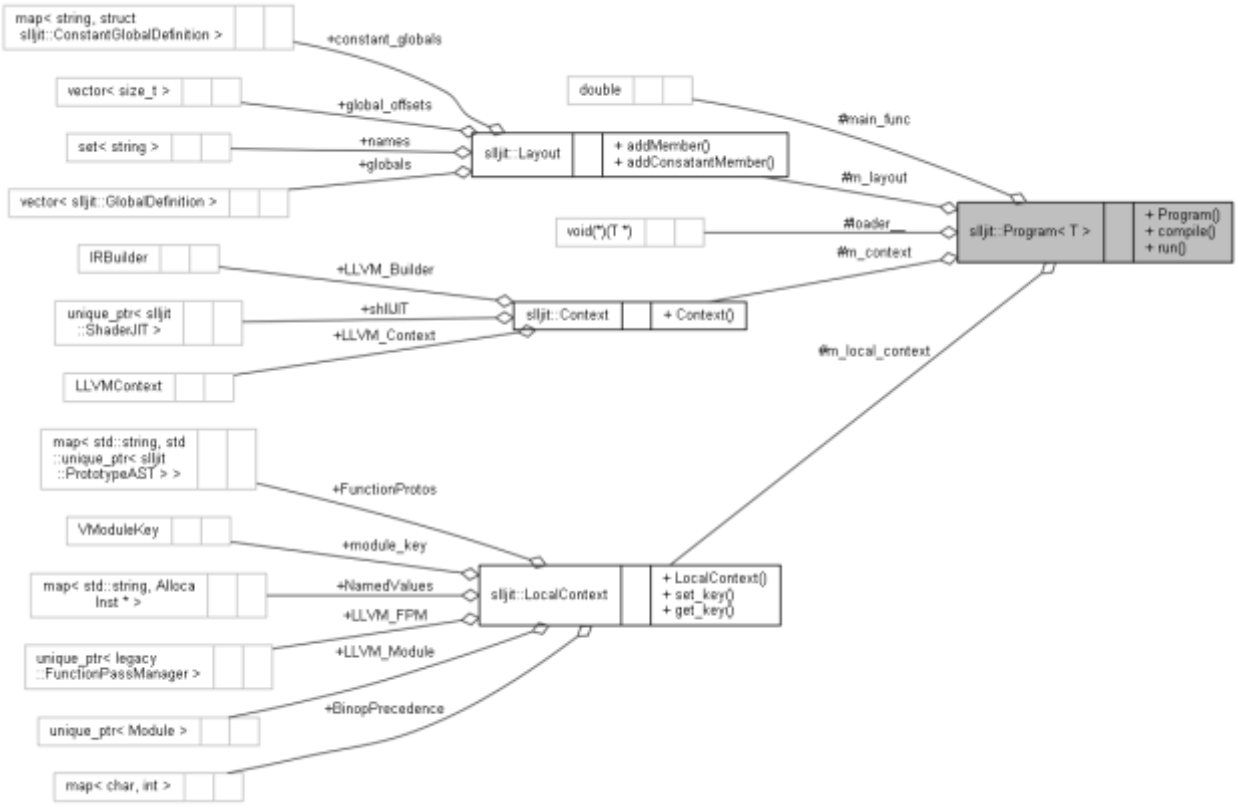

Рисунок 2.9 – Діаграма класу Program

Така організація дозволить використовувати бібліотеку оперуючи лише трьома класами Context, Layout та Program. Повна UML діаграма зображена на рисунку А.1.

2.1 Висновки до розділу

У даному розділі проведено синтез структури бібліотеки за принципом ООР. Фази компіляції розбито на задачі кожну із яких реалізує окремий клас. Представлено інтерфейс бібліотеки (набір класів) для кінцевого користувача що дозволяє просте вбудовування у будь який проект.

Зауважимо що винесення контексту компіляції у інтерфейс користувача дозволяє користувачеві писати мультипоточні програми.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Обґрунтування інструментів розробки

У якості мови програмування обрано C++. Це високорівнева мова програмування що традиційно використовується для розробки компіляторів бібліотек, операційних систем - у цілому для програмних продуктів де важлива швидкодія адже це типізована мова програмування. C++ використовується у таких продуктах як Microsoft Windows, MacOS X, Tensorflow, ML бібліотеках, графічних АПІ [17].

У якості бекенду для компілятора обрано LLVM 10. LLVM це інфраструктура для створення компіляторів що немає аналогів. На його основі написані такі компілятори як Clang, Rust Compiler, Swift Compiler та інші [18].

3.2 Розробка специфікації шейдероподібної мови

За основу взята мова GLSL [19] що використовується для шейдерів OpenGL.

Вхідна точка програми - функція main що не приймає аргументів, проте може повертати значення.

Базові типи:

– double за специфікацією IEEE 754 [20].

Списки простих виразів розділяються символом ; (крапка з комою).

Складені вирази:

– тип ідентифікатор(аргументи) {тіло} - функція;
– тип ідентифікатор; - змінна;
– if(умова){тіло}else{тіло} - розгалуження;
– for(вираз;умова;вираз;){тіло} - цикл;

– return вираз - повернення результату за функції

– extern тип ідентифікатор(аргументи); - визначення сторонньої (external) функції.

Програми (шейдери) компілюються у самодостатні модулі які розміщуються у пам’яті. Функції одного модуля за необхідності можуть викликати функції іншого уже відкомпільованого модуля.

Глобальні змінні задаються програмно у вигляді макету даних (layout) та зберігаються на рівні програмного модуля. Глобальна змінна представляє собою константний вказівник на область пам’яті що виділяється у модулі під час компіляції. Програмно можуть вказуватися як константні так і не константні (mutable) глобальні змінні.

Функція __layout_loader_ є зарезервованою.

3.3 Алгоритм токенізації

Алгоритм токенізації зображено на рисунку Б.1. При токенізації пропускаються пусті символи (пробіли, табуляція, символ нової стрічки та переводу каретки у початок стрічки) доки не буде знайдено перший відображуваний символ. Далі, якщо цей символ входить до набору символів початку ідентифікатора (у вигляді регулярного виразу – *[a-zA-Z_]*) то починаємо заповнювати стрічку ідентифікатора доки не зустрінемо перший символ що не входить до набору *[a-zA-Z0-9_]*, після чого перевіряємо стрічку ідентифікатора на відповідність зарезервованим ідентифікаторам таким як extern, if, else, for та double. При співпадінні повертаємо необхідний токен, за не співпадіння повертаємо токен загального ідентифікатора.

Якщо ж символ не входить до набору *[a-zA-Z_]* та починається з символів набору *[0-9]* то маємо число. Заповнюємо стрічку числа доки не зустрінемо символ що не входить до набору *[0-9.]* та повертаємо токен числа.

Якщо символ не входить до набору *[0-9]* то перевіряємо на кінець вхідних даних. Якщо кінець то повертаємо токен кінця, якщо ні – повертаємо сам символ [21].

3.4 Алгоритм парсування

Алгоритм парсування зображено на рисунку Б.2. При парсуванні токени зачитуються підряд та на основі їх послідовності визначається тип виразу для парсування.

На найвищому рівні компільованої програми можуть зустрічатися токени кінця вхідних даних, extern функції та токен типу. Основний цикл парсування полягає у перевірці на один із цих токенів та виклик необхідного субалгоритму.

З типу можуть починатися вирази декларації функції та змінної. Таким чином наступним токеном має бути токен ідентифікатору. Якщо після нього наступний токен належить до *[(]* то це функція, якщо до *[=;]* то це змінна.

При парсуванні функції спочатку виділяється список аргументів та їх типів що записані через кому доки не зустрінемо токен що належить до *[)]* пясля чого запускається субалгоритм парсування списку виразів.

Очікується що список виразів починається *[{]* та закінчується *[}]*, а вирази розділені символом *[;]*. До можливих виразів входить виклик функції, декларація змінної, розгалуження, цикл, термінатор та бінарний вираз.

При парсуванні декларації змінної очікується ідентифікатор, потім токен що належить до *[=;]*. При наявності ініціалізуючого виразу значення змінної визначене користувачем, за його відсутності змінна занулюється.

При парсуванні бінарного виразу визначається оператор та його прецедентність, після чого порядок операцій визначається рекурисвним підвищенням прецедентності та обміном операндів між операндами.

Парсування виклику функції практично ідентичне парсуванню її прототипу.

Парсування розгалуження починається з токену tok_if після чого очікується вираз умови у круглих дужках, та список виразів. Наявність tok_else є опціональною, алгоритм ідентичний за винятком умови.

Парсування циклу починається з токену tok_for після чого очікується 3 вирази у круглих дужках розділені між собою токеном *[;]*. Перший вираз – декларація змінної циклу, дургий – умова циклу, третій – вираз що виконується після кожної ітерації. Після чого очікується список виразів що будуть тілом циклу. Зауважимо що наявність NoOp виразу дозволяє не вказувати жоден з виразів. У такому випадку умова визначається як завжди *true* – отримуємо безкінечний цикл, при чому такий цикл повинен містити хоча б один досяжний термінатор.

Парсування термінатору функції починається з отримання токену tok_identifier. Якщо ідентифікатор відповідає командному слову *return* то очікуємо вираз та токен *[;]*.

3.5 Приклад використання бібліотеки

Для прикладу використання візьмемо правило Фібоначчі. Наступний код на C++ використовує бібліотеку для генерування та використання функції яка реалізує ітеративний пошук наступного числа за правилом Фібоначчі та заданими початковими числами:

```
01: #include "Tokenizer.h"
02: #include "Context.h"
03: #include "Layout.h"
04: #include "Program.hpp"
05:
06: struct Data
07: {
08:     double iter;
```

```
09:     double start_0;
10:     double start_1;
11: };
12:
13: int main()
14: {
15:     Context m_context;
16:     Program<Data> m_program(m_context);
17:     Layout m_layout;
18:     m_layout.addMember("iter", ::Kdouble, offsetof(Data, iter));
19:     m_layout.addMember("start_0", ::Kdouble, offsetof(Data, start_0));
20:     m_layout.addMember("start_1", ::Kdouble, offsetof(Data, start_1));
21:
22:     std::string source_code = R"(
23:     double main()
24:     {
25:             double left = start_0;
26:             double right = start_1;
27:             for(double idx = 0;idx<iter;idx = idx+1 )
28:             {
29:                     double tmp = right;
30:                     right = right+left;
31:                     left = tmp;
32:             }
33:             return right;
34:     }
35: )";
36:     m_program.compile(source_code, m_layout);
```

```
37:     Data data{4.0,0.0, 1.0};
38:     auto retval = m_program.run(&data);
39:     return 0;
40: }
```

У наведеному коді стрічки 1-4 включають необхідні файли заголовків. Стрічки 6-10 декларують структуру типу Data з трьома полями типу double. У стрічці 13 починається основна функція main, далі у стрічках 15-17 створюються необхідні для роботи екземпляри об’єктів контексту, програми та макету даних відповідно. У стрічках 18-20 макет даних заповнюється інформацією про елементи структури Data для того щоб її можна було використовувати при компіляції коду. У стрічках 22-35 задається тіло програми що буде відкомпільована. Як бачимо у стрічках 25-26 створюються локальні змінні що ініціалізуються глобальними змінними що були задані у макеті даних. Після чого у стрічках 27-32 простий цикл що реалізує правило Фібоначчі. У стрічці 33 кінцеве число повертається з функції. Далі у стрічці 36 викликається функції що компілює задану програму за заданим макетом даних та повідомляє про помилки. У стрічці 37 створюється та ініціалізується екземпляр структури Data. У стрічці 38 викликається відкомпільована програма з вказівником на екземпляр структури Data, цей вказівник буде використано зарезервованою функцією __layout_loader_ для заповнення глобальних змінних з макету даних для подальшого використання.

Слід зауважити що при зміні значень глобальних змінних у тілі компільованої програми поля екземпляру структури Data вказівник на який було передано функції також будуть змінені.

3.6 Генерація IR, цільового коду та JIT модуль

Безпосередньо генерація IR реалізована за допомогою LLVM. Завдяки тому що усі елементи AST дерева наслідують абстрактний клас, то задача генерації IR зводиться до ітерації по AST дереву та виклику віртуальної функції кожного з елементів. Кожен клас перевизначає цю віртуальну функцію для генерації відповідного проміжного коду, що записується у модуль [22, 32 ,24].

Для прикладу візьмемо програму наведену у розділі 3.5. Після первинної генерації IR отримаємо наступну LLVM IR:

```
02: target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
03:
04: %layout__ = type { double, double, double }
05:
06: @iter = global double* null
07: @start_0 = global double* null
08: @start_1 = global double* null
09: @v = constant double 5.000000e+00
10:
11: define void @__layout_loader_(%layout__* %data_ptr) {
12: entry:
13:   %0 = getelementptr %layout__, %layout__* %data_ptr, i32 0, i32 0
14:   store double* %0, double** @iter, align 8
15:   %1 = getelementptr %layout__, %layout__* %data_ptr, i32 0, i32 1
16:   store double* %1, double** @start_0, align 8
17:   %2 = getelementptr %layout__, %layout__* %data_ptr, i32 0, i32 2
18:   store double* %2, double** @start_1, align 8
```

```
19:   ret void
20: }
21:
22: define double @main() {
23: entry:
24:   %tmp = alloca double, align 8
25:   %idx = alloca double, align 8
26:   %right = alloca double, align 8
27:   %left = alloca double, align 8
28:   %start_0 = load double*, double** @start_0, align 8
29:   %start_01 = load double, double* %start_0, align 8
30:   store double %start_01, double* %left, align 8
31:   %start_1 = load double*, double** @start_1, align 8
32:   %start_12 = load double, double* %start_1, align 8
33:   store double %start_12, double* %right, align 8
34:   store double 0.000000e+00, double* %idx, align 8
35:   br label %loop_header
36:
37: loop_header:                                ; preds = %loop_body, %entry
38:   %idx3 = load double, double* %idx, align 8
39:   %iter = load double*, double** @iter, align 8
40:   %iter4 = load double, double* %iter, align 8
41:   %cmptmp = fcmp ult double %idx3, %iter4
42:   %booltmp = uitofp i1 %cmptmp to double
43:   %loopcond = fcmp one double %booltmp, 0.000000e+00
44:   br i1 %loopcond, label %loop_body, label %afterloop
45:
46: loop_body:                                  ; preds = %loop_header
```

```
47:   %right5 = load double, double* %right, align 8
48:   store double %right5, double* %tmp, align 8
49:   %right6 = load double, double* %right, align 8
50:   %left7 = load double, double* %left, align 8
51:   %addtmp = fadd double %right6, %left7
52:   store double %addtmp, double* %right, align 8
53:   %tmp8 = load double, double* %tmp, align 8
54:   store double %tmp8, double* %left, align 8
55:   %idx9 = load double, double* %idx, align 8
56:   %addtmp10 = fadd double %idx9, 1.000000e+00
57:   store double %addtmp10, double* %idx, align 8
58:   br label %loop_header
59:
60: afterloop:                                    ; preds = %loop_header
61:   %right11 = load double, double* %right, align 8
62:   ret double %right11
63: }
```

У наведеному коді стрічки 2-10 це шапка модуля у якій декларуються глобальні змінні та макет збереження даних самого модуля. У стрічках 11-20 декларується функція __layout_loader_ що виконує загрузку глобальних змінних за вказівником на екземпляр структури. Слід зазначити що функція _layout_loader_ генерується в залежності від макету даних заповненого користувачем. У стрічках 22 – 63 декларується основна функція main. Як видно із стрічок 24-34 усі локальні змінні виділену у стеку. У стрічках 38-40, 47-57 зайві операції завантаження, у стрічках 41-44 операції порівнянь можуть бути об’єднані.

Після первинної генерації IR у модулі проходить через конвеєр оптимізації LLVM який задається програмістом у вигляді проходів. Проходи можуть включати

у себе спрощення протоку виконання, злиття інструкцій, векторизацію, прибирання недосяжного коду, прибирання коду що не має наслідків, розподілення змінних по регістрам, вставка коду функції замість її виклику, та інші. Проходи можуть мати робочий список, тобто повертатися до вузлів IR у разі їх зміни у наступних проходах для переоцінки та переоптимізації. У даному випадку використано перелічені проходи. При попущенні наведеної LLVM IR через конвеєр оптимізації отримаємо:

```
02: target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
04: %layout__ = type { double, double, double }
06: @iter = global double* null
07: @start_0 = global double* null
08: @start_1 = global double* null
09: @v = constant double 5.000000e+00
11: define void @__layout_loader_(%layout__* %data_ptr) {
12: entry:
13:   %0 = getelementptr %layout__, %layout__* %data_ptr, i64 0, i32 0
14:   store double* %0, double** @iter, align 8
15:   %1 = getelementptr %layout__, %layout__* %data_ptr, i64 0, i32 1
16:   store double* %1, double** @start_0, align 8
17:   %2 = getelementptr %layout__, %layout__* %data_ptr, i64 0, i32 2
18:   store double* %2, double** @start_1, align 8
19:   ret void
20: }
22: define double @main() {
23: entry:
24:   %start_0 = load double*, double** @start_0, align 8
25:   %start_01 = load double, double* %start_0, align 8
```

```
26:   %start_1 = load double*, double** @start_1, align 8
27:   %start_12 = load double, double* %start_1, align 8
28:   %iter.pre = load double*, double** @iter, align 8
29:   %iter4.pre = load double, double* %iter.pre, align 8
30:   br label %loop_header
32: loop_header:                                  ; preds = %loop_body, %entry
33:   %right.0 = phi double [ %start_12, %entry ], [ %addtmp, %loop_body ]
34:   %idx.0 = phi double [ 0.000000e+00, %entry ], [ %addtmp10, %loop_body ]
35:   %left.0 = phi double [ %start_01, %entry ], [ %right.0, %loop_body ]
36:   %cmptmp = fcmp ult double %idx.0, %iter4.pre
37:   br i1 %cmptmp, label %loop_body, label %afterloop
39: loop_body:                                    ; preds = %loop_header
40:   %addtmp = fadd double %right.0, %left.0
41:   %addtmp10 = fadd double %idx.0, 1.000000e+00
42:   br label %loop_header
44: afterloop:                                    ; preds = %loop_header
45:   ret double %right.0
46: }
```

Як бачимо локальні змінні тепер знаходяться у регістрах, зайві завантаження значень без необхідності прибрані. Ініціалізація змінної циклу та змінних що містять числа послідовності за правилом Фібоначчі тепер відбувається на основі вузла переходу, тобто значення змінної залежить від базового блоку з якого відбувається перехід що дозволяє додаткову оптимізацію.

Після машино незалежної оптимізації код передається у JIT модуль де визначається конфігурація ЕВМ користувача, та IR модуля проходить через машино залежну оптимізацію що визначається у LLVM для кожної доступної архітектури. Така оптимізація включає у себе використання розширень наборів

інструкцій таких як SSE чи AVX, інструкцій так званих машинних циклів (коли цикл реалізується не на основі розгалуження, а на основі спеціальних інструкцій), та інші. При проходженні наведеної раніше LLVM IR через конвеєр машинно залежної оптимізації отримуємо наступний псевдоасемблерний (LLVM CGFT ASM IR) код:

```
01: ; Assembly:
02:         .text
03:         .def     @feat.00;
04:         .scl    3;
05:         .type   0;
06:         .endef
07:         .globl  @feat.00
08: .set @feat.00, 0
09:         .file   "Program"
10:         .def     __layout_loader_;
11:         .scl    2;
12:         .type   32;
13:         .endef
14:         .globl  __layout_loader_
15:         .p2align        4, 0x90
16: __layout_loader_:
17:         movabsq $iter, %rax
18:         movq    %rcx, (%rax)
19:         leaq    8(%rcx), %rax
20:         movabsq $start_0, %rdx
21:         movq    %rax, (%rdx)
22:         addq    $16, %rcx
23:         movabsq $start_1, %rax
```

```
24:         movq    %rcx, (%rax)
25:         retq
27:         .def     main;
28:         .scl    2;
29:         .type   32;
30:         .endef
31:         .globl  __real@3ff0000000000000
32:         .section        .rdata,"dr",discard,__real@3ff0000000000000
33:         .p2align       3
34: __real@3ff0000000000000:
35:         .quad   0x3ff0000000000000
36:         .text
37:         .globl  main
38:         .p2align       4, 0x90
39: main:
40:         movabsq $start_0, %rax
41:         movq    (%rax), %rax
42:         movsd   (%rax), %xmm1
43:         movabsq $start_1, %rax
44:         movq    (%rax), %rax
45:         movsd   (%rax), %xmm5
46:         movabsq $iter, %rax
47:         movq    (%rax), %rax
48:         movsd   (%rax), %xmm2
49:         xorpd   %xmm3, %xmm3
50:         movabsq $__real@3ff0000000000000, %rax
51:         movsd   (%rax), %xmm4
52:         .p2align       4, 0x90
```

```
53: .LBB1_1:
54:         movapd  %xmm5, %xmm0
55:         ucomisd %xmm2, %xmm3
56:         jae     .LBB1_3
57:         addsd   %xmm0, %xmm1
58:         addsd   %xmm4, %xmm3
59:         movapd  %xmm1, %xmm5
60:         movapd  %xmm0, %xmm1
61:         jmp     .LBB1_1
62: .LBB1_3:
63:         retq
65:         .bss
66:         .globl  iter
67:         .p2align        3
68: iter:
69:         .quad   0
71:         .globl  start_0
72:         .p2align        3
73: start_0:
74:         .quad   0
76:         .globl  start_1
77:         .p2align        3
78: start_1:
79:         .quad   0
81:         .section        .rdata,"dr"
82:         .globl  v
83:         .p2align        3
84: v:
```

```
85:       .quad   0x4014000000000000
87:       .globl  _fltused
```

Як видно з наведеного коду, була проведена оптимізація з урахуванням архітектури ЕВМ, операції зі змінними перетворені у операції над регістрами, та замість операцій додати чи відняти та інших маємо конкретні асемблерні інструкції.

Після машино залежної оптимізації модуль передається у шар компілятора LLVM який виконує генерацію цільового коду та записує отриманий модуль у пам'ять. Надалі отримується адрес точки входу у модуль для послідуючого використання класом Program як було продемонстровано у розділі 3.5. При компіляції наведеного раніше коду отримуємо послідовність машинних інструкції, дизасембювавши які за допомогою інструменту ghidra отримаємо наступний асемблерний код:

```
002:                    // .bss
003:                    // PhysAddr:0x2000 Size:0x18 Flags:0xc0400080
004:                    // ram: 00002000-00002017
005:                    //
006:     assume DF = 0x0  (Default)
007:                iter                            XREF[4]:   __layout_loader_:00002200(*),
008:                                                           __layout_loader_:0000220a(W),
009:                                                           main:00002252(*),
010:                                                           main:0000225c(R)
011: 00002000              undefined8 ??
012:                start_0                         XREF[4]:   __layout_loader_:00002211(*),
013:                                                           __layout_loader_:0000221b(W),
014:                                                           main:00002230(*),
015:                                                           main:0000223a(R)
016: 00002008              undefined8 ??
```

```
017:              start_1                         XREF[5]:   __layout_loader_:00002222(*),
018:                                                         __layout_loader_:00002222(*),
019:                                                         __layout_loader_:0000222c(W),
020:                                                         main:00002241(*),
021:                                                         main:0000224b(R)
022: 00002010              undefined8 ??
024:                  // .rdata
025:                  // PhysAddr:0x2100 Size:0x8 Flags:0x40401040
026:                  // ram: 00002100-00002107
027:                  //
028:                  .data                           XREF[3]:   main:00002267(*),
029:                  __real@3ff0000000000000                    main:00002267(*),
030:                                                             main:00002271(R)
031: 00002100 00 00 00       undefined8 3FF0000000000000h
032:        00 00 00
033:        f0 3f
035:                  // .rdata
036:                  // PhysAddr:0x2108 Size:0x8 Flags:0x40400040
037:                  // ram: 00002108-0000210f
038:                  //
039:                  v
040: 00002108 00           ??         00h
041: 00002109 00           ??         00h
042: 0000210a 00           ??         00h
043: 0000210b 00           ??         00h
044: 0000210c 00           ??         00h
045: 0000210d 00           ??         00h
046: 0000210e 14           ??         14h
```

```
047: 0000210f 40          ??         40h    @
049:                      // .text
050:                      // PhysAddr:0x2200 Size:0x9d Flags:0x60500020
051:                      // ram: 00002200-0000229c
052:                      //
053:
*************************************************************
054:                      *                FUNCTION                    *
*************************************************************
056:                      undefined __layout_loader_()
057:     undefined        AL:1           <RETURN>
058:                      __layout_loader_                   XREF[1]:     Entry Point(*)
059: 00002200 48 b8 00     MOV        RAX,iter                          = ??
060:          20 00 00
061:          00 00 00 00
062: 0000220a 48 89 08     MOV        qword ptr [RAX]=>iter,RCX                 = ??
063: 0000220d 48 8d 41 08   LEA        RAX,[RCX + 0x8]
064: 00002211 48 ba 08     MOV        RDX,start_0                       = ??
065:          20 00 00
066:          00 00 00 00
067: 0000221b 48 89 02     MOV        qword ptr [RDX]=>start_0,RAX              = ??
068: 0000221e 48 83 c1 10   ADD        RCX,0x10
069: 00002222 48 b8 10     MOV        RAX=>start_1,start_1                  = ??
070:          20 00 00
071:          00 00 00 00
072: 0000222c 48 89 08     MOV        qword ptr [RAX]=>start_1,RCX               =
073: 0000222f c3           RET
*************************************************************
```

```
075:                *                    FUNCTION                    *
*************************************************************
077:                double __stdcall main(void)
078:     double          XMM0_Qa:8     <RETURN>
079:                main                                XREF[1]:     Entry Point(*)
080: 00002230 48 b8 08       MOV       RAX,start_0                          = ??
081:          20 00 00
082:          00 00 00 00
083: 0000223a 48 8b 00       MOV       RAX=>start_0,qword ptr [RAX]               = ??
084: 0000223d f2 0f 10 08    MOVSD     XMM1,qword ptr [RAX]
085: 00002241 48 b8 10       MOV       RAX,start_1                          = ??
086:          20 00 00
087:          00 00 00 00
088: 0000224b 48 8b 00       MOV       RAX=>start_1,qword ptr [RAX]               = ??
089: 0000224e f2 0f 10 28    MOVSD     XMM5,qword ptr [RAX]
090: 00002252 48 b8 00       MOV       RAX,iter                             = ??
091:          20 00 00
092:          00 00 00 00
093: 0000225c 48 8b 00       MOV       RAX=>iter,qword ptr [RAX]                  = ??
094: 0000225f f2 0f 10 10    MOVSD     XMM2,qword ptr [RAX]
095: 00002263 66 0f 57 db    XORPD     XMM3,XMM3
096:      00002267      48      b8      00                                   MOV
RAX=>__real@3ff0000000000000,__real@3ff0000000   = 3FF0000000000000h
097:          21 00 00
098:          00 00 00 00
099:   00002271   f2   0f   10   20            MOVSD              XMM4,qword   ptr
[RAX]=>__real@3ff0000000000000    = 3FF0000000000000h
100: 00002275 90             NOP
```

```
101: 00002276 90          NOP
102: 00002277 90          NOP
103: 00002278 90          NOP
104: 00002279 90          NOP
105: 0000227a 90          NOP
106: 0000227b 90          NOP
107: 0000227c 90          NOP
108: 0000227d 90          NOP
109: 0000227e 90          NOP
110: 0000227f 90          NOP
111:                LAB_00002280                    XREF[1]:    0000229a(j)
112: 00002280 66 0f 28 c5    MOVAPD    XMM0,XMM5
113: 00002284 66 0f 2e da    UCOMISD   XMM3,XMM2
114: 00002288 73 12          JNC       LAB_0000229c
115: 0000228a f2 0f 58 c8    ADDSD     XMM1,XMM0
116: 0000228e f2 0f 58 dc    ADDSD     XMM3,XMM4
117: 00002292 66 0f 28 e9    MOVAPD    XMM5,XMM1
118: 00002296 66 0f 28 c8    MOVAPD    XMM1,XMM0
119: 0000229a eb e4          JMP       LAB_00002280
120:                LAB_0000229c                    XREF[1]:    00002288(j)
121: 0000229c c3          RET
123:                // EXTERNAL
124:                // ram: 000032a0-000032a7
125:                //
126:                _fltused
```

3.7 Висновки до розділу

У даному розділі описана алгоритмічна база для парсування вхідної програми на розробленій шейдероподібній мові високого рівня, та представлено процес перетворення AST у цільовий код з використанням інфраструктури LLVM.

Приклад наведено на основі функції що реалізує правило Фібоначчі для двох будь яких заданих чисел на задану кількість ітерацій.

ВИСНОВКИ

У даній роботі розглядаються питання, пов’язані з компіляцією мови програмування для застосування у JIT компіляторі.

У першому розділі розглянуто структуру компілятора та JIT компіляцію. Визначено, що компіляція ділиться на 2 основні етапи – аналіз та синтез які поділяються на набір фаз: лексичний аналіз, синтаксичний аналіз, семантичний аналіз, генерація проміжного коду, оптимізація та генерація цільового коду. У другому розділі розроблено мову програмування та програмне забезпечення для JIT компіляції. Основними результатами роботи є:

а) шейдероподібна мова програмування;

б) програмне забезпечення для її JIT компіляції.

Одним із напрямків застосування результатів розробки може бути розробка 2D векторного графічного двигуна із CPU шейдерами. Отримані результати дозволять збільшити пропускну здатність обчислень проектів на мові C++.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Performance differences when using AVX instructions [Web resource]. – Access mode: https://albertherd.com/2018/01/11/performance-differences-when-using-avx-instructions/.

2. How Intel® AVX Improves Performance on Server Applications [Web resource]. – Access mode: https://software.intel.com/content/www/us/en/develop/blogs/how-intel-avx-improves-performance-on-server-application.html.

3. Gathering Intel on Intel AVX-512 Transitions [Web resource]. – Access mode: https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html.

4. JIT compiler vs offline compilers [Web resource]. – Access mode: http://net-informations.com/faq/qk/jit.htm.

5. GPGPU Processing in CUDA Architecture - arXiv [Web resource]. – Access mode: https://arxiv.org/ftp/arxiv/papers/1202/1202.4347.pdf.

6. Кулик М.С. Огляд лексичного та синтаксичного аналізу мов програмування для застосування у компіляторах / М.С. Кулик, В.М. Севастьянов, О.М. Бевз [Електронний ресурс]: XLIX науково-технічна конференція факультету комп'ютерних систем і автоматики (2020). – Режим доступу: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2020/paper/view/8984/7876.

7. Alfred A. Compilers - Principles, Techniques, and Tools / A. Alfred, L. Monica, S. Ravi, U. Jeffrey. Pearson Education, 2007. - 1035 p.

8. Compiler Theory: Introduction [Web resource]. – Access mode: https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Introduction.html.

9. Compiler Architecture [Web resource]. – Access mode: https://cs.lmu.edu/~ray/notes/compilerarchitecture/.

10. Farhanaaz An Exploration on Lexical Analysis / Farhanaaz, V Sanju // In Proceedings of the International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT). – 2016. – P. 253-258. – DOI: 10.1109. - ICEEOT.2016.7755127

11. Chuhan R. Implementation of Lexical Analysis / R. Chuhan, V. Singh, K. Makhan, M. Kumar // International Journal for Research in Applied Science & Engineering Technology. - 2017. – P. 614-617. – ISSN: 2321-9653

12. Six Steps in a Syntactic Analysis [Web resource]. – Access mode: http://www.ello.uos.de/field.php/Syntax/TG6Steps.

13. Syntax Analysis: Top Down, Bottom Up Parsing Types [Web resource]. – Access mode: https://www.guru99.com/syntax-analysis-parsing-types.html.

14. Vectorization-Aware Loop Unrolling with Seed Forwarding [Web resource]. – Access mode: https://dl.acm.org/doi/pdf/10.1145/3377555.3377890.

15. JIT through the ages [Web resource]. – Access mode: http://web2.cs.columbia.edu/~aho/cs6998/reports/12-12-17_Ramanan_JIT.pdf.

16. LLVM Documentation [Web resource]. – Access mode: https://llvm.org/10.0.0/docs/index.html.

17. C++ Language: Features, Uses, Applications & Advantages [Web resource]. – Access mode: https://hackr.io/blog/features-uses-applications-of-c-plus-plus-language.

18. The LLVM Compiler Infrastructure [Web resource]. – Access mode: https://llvm.org/.

19. The OpenGL Shading Language [Web resource]. – Access mode: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.40.pdf.

20. Double precision floating-point format [Web resource]. – Access mode: https://borninsummer.com/files/2018/06/Double-precision-floating-point-format-Wikipedia.pdf.

21. Дональд Кнут Искусство программирования. Том 1. Основные алгоритмы / К. Дональд; [пер. с англ. С. Н. Тригуба]. - Вильямс, 2010. - 720 с.

22. Б'ярн Страуструп Характеристики Программирование: принципы и практика с использованием C++ / С. Б'ярн; [пер. с англ. Д. А. Клюшиной]. - Диалектика, 2011. - 1248 с.

23. Б'ярн Страуструп Дизайн и эволюция языка C++ / С. Б'ярн; [пер. с англ. А. В. Слинкина]. - ДМК Пресс, 2011. - 446 с.

24. Б'ярн Страуструп Язык программирования C++ / С. Б'ярн; [пер. с англ. С. Н. Тригуба]. - Бином-Пресс, 2012. - 1136 с.

ДОДАТКИ

Додаток А

(обов’язковий)

UML діаграма ПО

Додаток А

(обов’язковий)

UML діаграма ПО

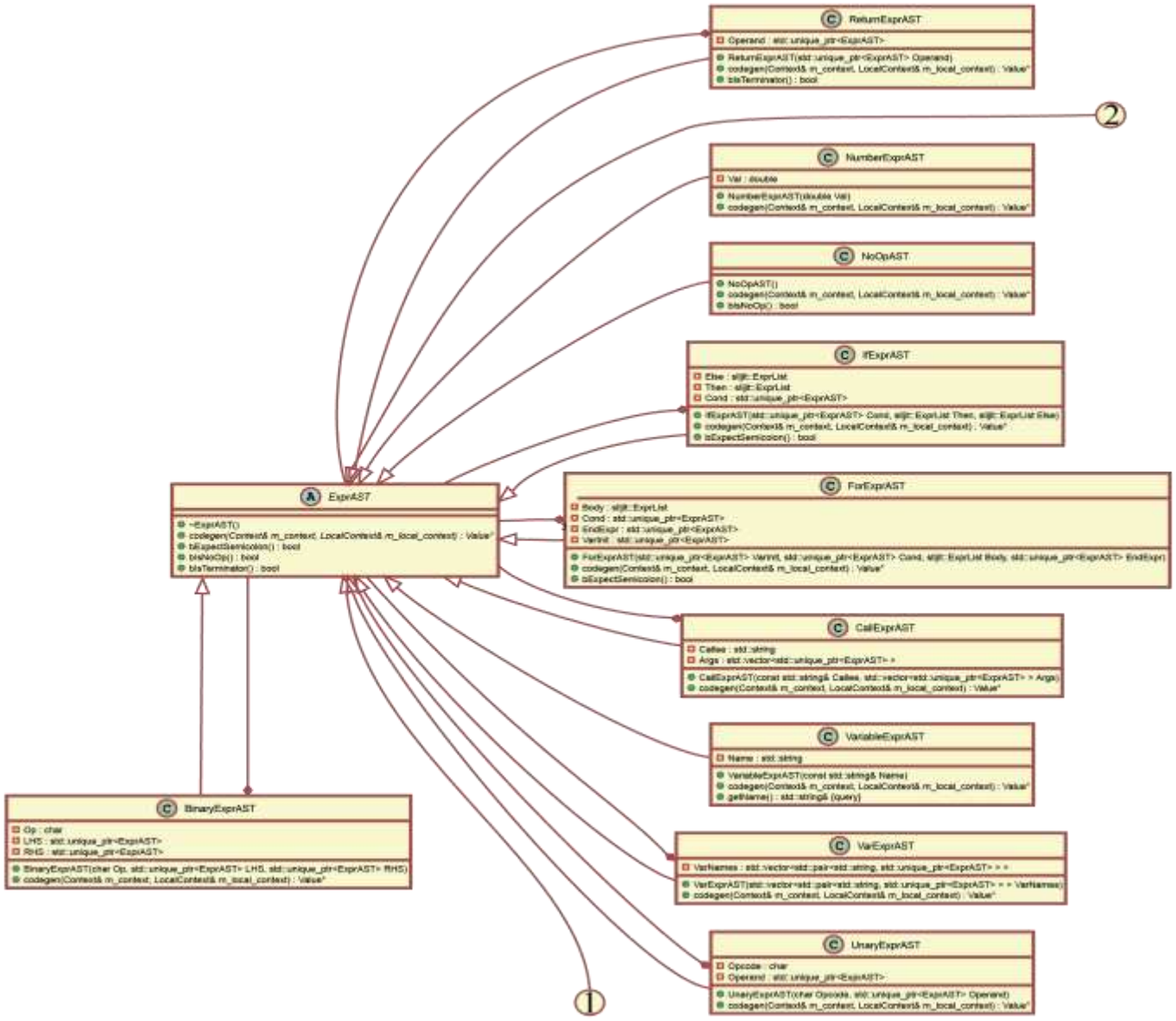

Рисунок А.1, аркуш 1 – UML діаграма ПО

Додаток А

(обов’язковий)

UML діаграма ПО

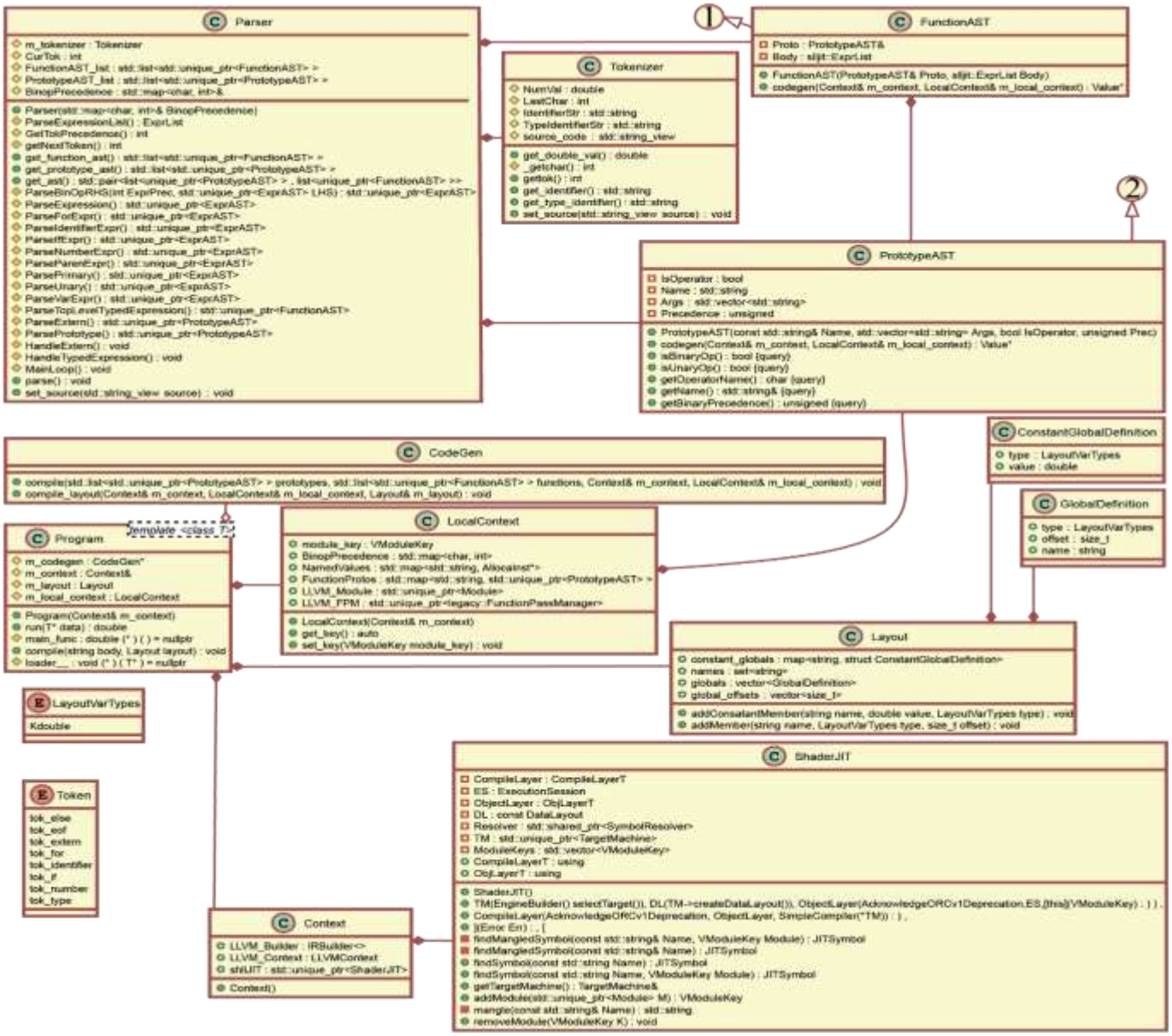

Рисунок А.1, аркуш 2 – UML діаграма ПО

Додаток Б

(обов’язковий)

Алгоритми

Додаток Б

(обов’язковий)

Алгоритми

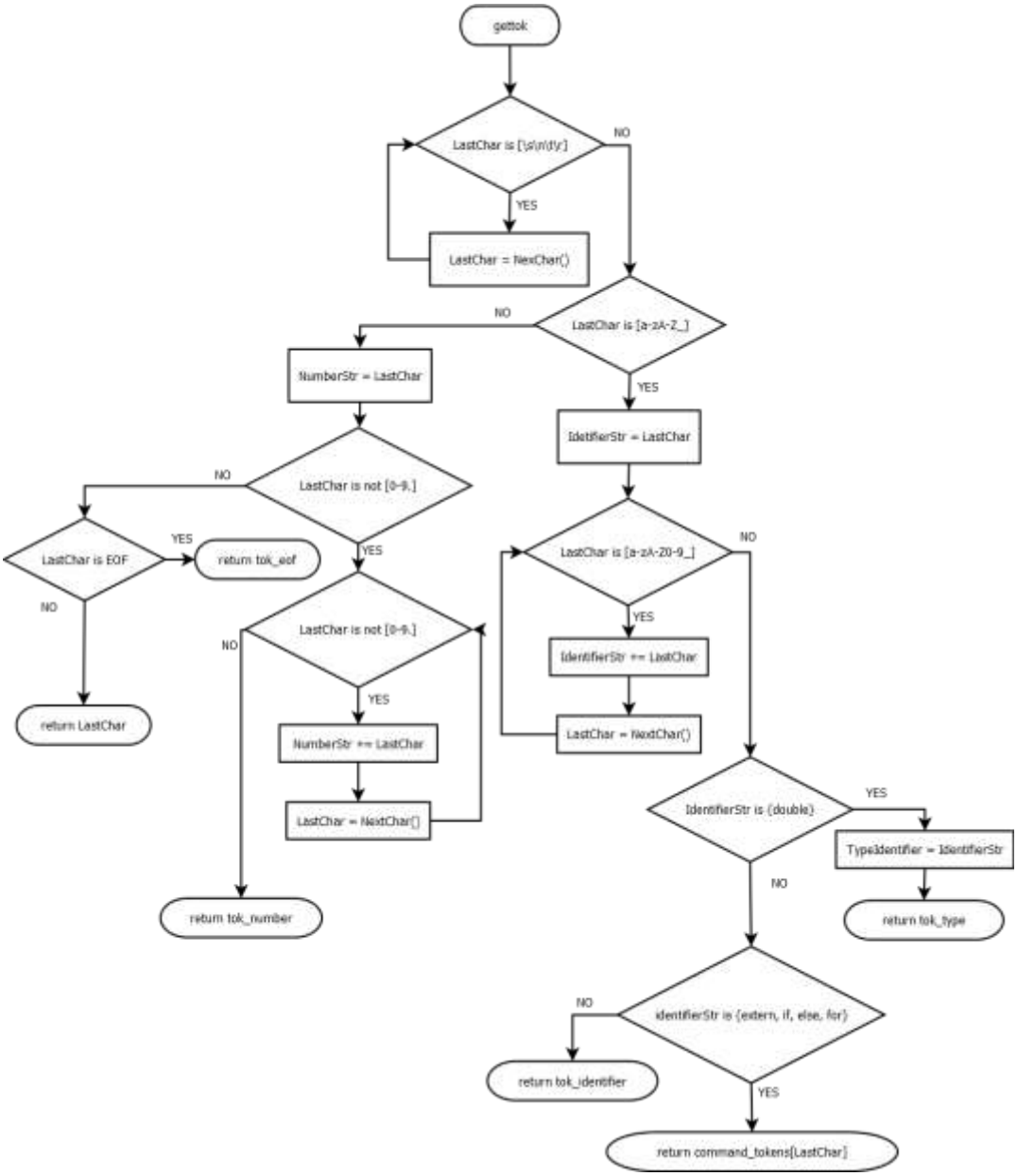

Рисунок Б.1 – Алгоритм Токенізації

Додаток Б

(обов’язковий)

Алгоритми

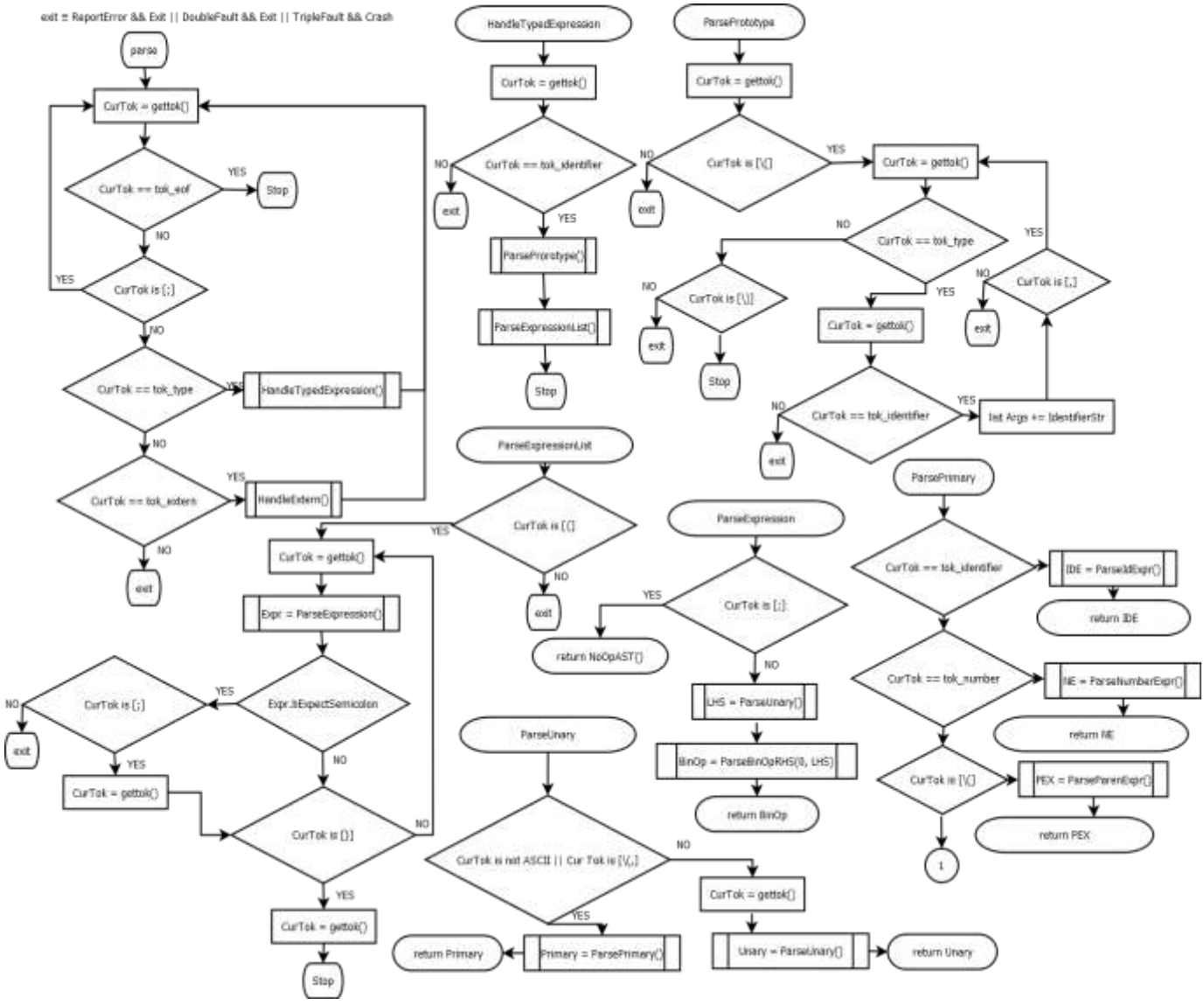

Рисунок Б.2, аркуш 1 – Алгоритм Парсування

Додаток Б

(обов’язковий)

Алгоритми

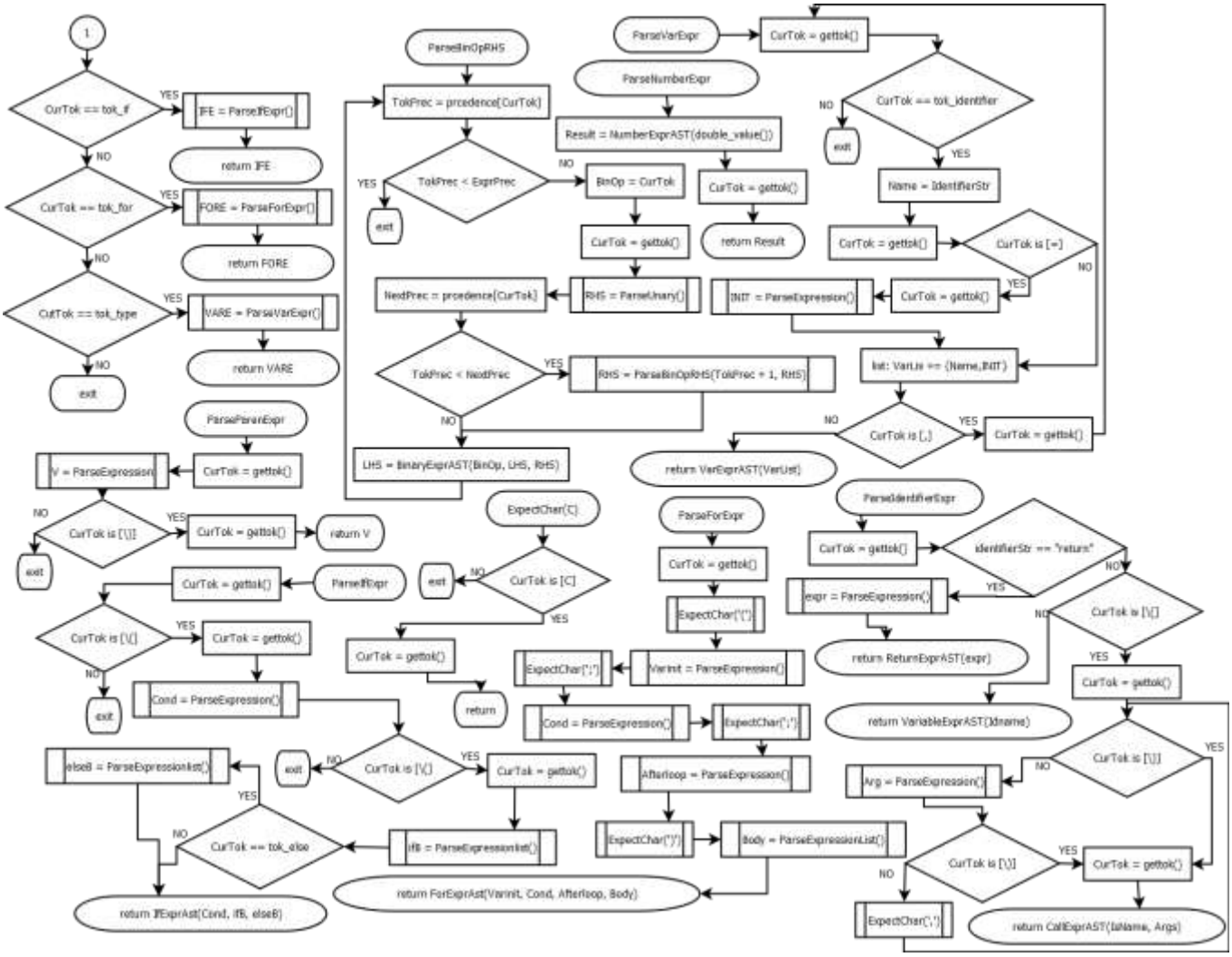

Рисунок Б.2, аркуш 2 – Алгоритм Парсування

Додаток В

(обов’язковий)

Лістинг програмного забезпечення

```
// pch.h
#ifndef PCH_H
#define PCH_H
#include "llvm/ADT/STLExtras.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/ExecutionEngine/JITSymbol.h"
#include "llvm/ExecutionEngine/Orc/CompileUtils.h"
#include "llvm/ExecutionEngine/Orc/IRCompileLayer.h"
#include "llvm/ExecutionEngine/Orc/IRTransformLayer.h"
#include "llvm/ExecutionEngine/Orc/IndirectionUtils.h"
#include "llvm/ExecutionEngine/Orc/LambdaResolver.h"
#include "llvm/ExecutionEngine/Orc/RTDyldObjectLinkingLayer.h"
#include "llvm/ExecutionEngine/RTDyldMemoryManager.h"
#include "llvm/ExecutionEngine/SectionMemoryManager.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/IR/Mangler.h"
#include "llvm/Support/DynamicLibrary.h"
#include "llvm/Support/Error.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetIntrinsicInfo.h"
#include "llvm/Transforms/InstCombine/InstCombine.h"
#include "llvm/Transforms/IPO.h"
#include "llvm/Transforms/AggressiveInstCombine/AggressiveInstCombine.h"
#include "llvm/Transforms/Vectorize.h"
#include "llvm/Transforms/Scalar.h"
#include "llvm/Transforms/Scalar/GVN.h"
#include "llvm/Analysis/BasicAliasAnalysis.h"
#include "llvm/Analysis/Lint.h"
#include "llvm/IR/PassManager.h"
#include "llvm/ADT/APFloat.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/Verifier.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Transforms/Utils.h"
#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <map>
#include <memory>
#include <string>
#include <vector>
#include <charconv>
#include <list>
#endif // PCH_H
//pch.cpp
#include "pch.h"
// Tokeizer.h
#pragma once
#include <string>
#include <set>
#include <map>

namespace slljit
{
```

```
        using namespace std;

        //===----------------------------------------------------------------------===//
        // Lexer
        //===----------------------------------------------------------------------===//

        /**
         * @enum slljit::Token
         * @brief Token representation. All values < 0.
        */
        enum Token : int
        {
                tok_eof = -1,

                // commands
                tok_type   = -2,
                tok_extern = -3,

                // primary
                tok_identifier = -4,
                tok_number     = -5,

                // control
                tok_if   = -6,
                tok_else = -8,
                tok_for  = -9,
        };

        class Tokenizer
        {
        protected:
                std::string IdentifierStr;     // Filled in if tok_identifier
                std::string TypeIdentifierStr; // Filled in if tok_type
                double NumVal;                 // Filled in if tok_number
                std::string_view source_code;
                static const set<char> identifier_start_charset;
                static const set<char> identifier_charset;
                static const map<string, Token> reserved_identifier_set;
                inline int _getchar();
                static inline pair<bool, Token> is_reserved_command_id(string c);
                static inline bool is_id_start_char(int c);
                static inline bool is_id_char(int c);
                int LastChar = ' ';
        public:
                void set_source(std::string_view source);
                /**
                 * @brief get next token from source code.
                 * @return an integer representation of a token. Either a Token or a character.
                 * @ref slljit::Token
                */
                int gettok();
                std::string get_identifier();
                std::string get_type_identifier();
                double get_double_val();
        };
}; // namespace slljit
// Tokenizer.cpp
#include "pch.h"
#include "Tokenizer.h"
using namespace llvm;
using namespace llvm::orc;
using namespace slljit;
using namespace std;
namespace slljit
{
        const set<char> Tokenizer::identifier_start_charset = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C',
            'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '_'};
        const set<char> Tokenizer::identifier_charset = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D',
```

```
            'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '_'};
        const map<string, Token> Tokenizer::reserved_identifier_set = {{"extern", tok_extern}, {"if", tok_if},
{"else", tok_else}, {"for", tok_for}};
        inline int Tokenizer::_getchar()
        {
                static size_t source_idx = 0;
                if (source_idx < source_code.size())
                {
                        return source_code[source_idx++];
                }
                else
                {
                        return EOF;
                }
        }
        inline pair<bool, Token> Tokenizer::is_reserved_command_id(string c)
        {
                auto it = reserved_identifier_set.find(c);
                if (it != reserved_identifier_set.end())
                {
                        return {true, it->second};
                }
                return {false, tok_eof};
        }
        inline bool Tokenizer::is_id_start_char(int c)
        {
                return identifier_start_charset.find(c) != identifier_start_charset.end();
        }
        inline bool Tokenizer::is_id_char(int c)
        {
                return identifier_charset.find(c) != identifier_charset.end();
        }
        void Tokenizer::set_source(std::string_view source)
        {
                this->source_code = source;
        }
        int Tokenizer::gettok()
        {
                // Skip any whitespace.
                while (isspace(LastChar))
                        LastChar = _getchar();
                if (is_id_start_char(LastChar))
                { // identifier: [a-zA-Z_][a-zA-Z0-9_]*
                        IdentifierStr = LastChar;
                        while (is_id_char((LastChar = _getchar())))
                                IdentifierStr += LastChar;
                        // BASIC TYPES
                        if (IdentifierStr == "double")
                        {
                                TypeIdentifierStr = IdentifierStr;
                                return tok_type;
                        }
                        // COMMANDS
                        {
                                auto [is_command, cmd_tok] = is_reserved_command_id(IdentifierStr);
                                if (is_command)
                                        return cmd_tok;
                        }
                        return tok_identifier;
                }

                if (isdigit(LastChar) || LastChar == '.')
                { // Number: [0-9.]+
                        std::string NumStr;
                        do
                        {
                                NumStr += LastChar;
                                LastChar = _getchar();
                        } while (isdigit(LastChar) || LastChar == '.');
                        std::from_chars(NumStr.c_str(), NumStr.c_str() + NumStr.size(), NumVal);
                        return tok_number;
```

```
                }
                // Check for end of file.  Don't eat the EOF.
                if (LastChar == EOF)
                        return tok_eof;
                // Otherwise, just return the character as its ascii value.
                int ThisChar = LastChar;
                LastChar     = _getchar();
                return ThisChar;
        }
        std::string Tokenizer::get_identifier()
        {
                return IdentifierStr;
        }
        std::string Tokenizer::get_type_identifier()
        {
                return TypeIdentifierStr;
        }
        double Tokenizer::get_double_val()
        {
                return NumVal;
        }
}; // namespace slljit
// Parser.h
#pragma once
#include <list>
#include <string>
#include <vector>
#include <memory>

#include "Tokenizer.h"
#include "AST.h"

namespace slljit
{
        using namespace std;
        class Parser
        {
        protected:
                std::map<char, int>& BinopPrecedence;
                std::list<std::unique_ptr<FunctionAST>> FunctionAST_list;
                std::list<std::unique_ptr<PrototypeAST>> PrototypeAST_list;
                Tokenizer m_tokenizer;
                int CurTok;
                int getNextToken();
        public:
                Parser(std::map<char, int>& BinopPrecedence);
                void set_source(std::string_view source);
                std::pair<list<unique_ptr<PrototypeAST>>, list<unique_ptr<FunctionAST>>> get_ast();
                std::list<std::unique_ptr<FunctionAST>> get_function_ast();
                std::list<std::unique_ptr<PrototypeAST>> get_prototype_ast();
                void parse();
        protected:
                int GetTokPrecedence();
                std::unique_ptr<ExprAST> ParseNumberExpr();
                std::unique_ptr<ExprAST> ParseParenExpr();
                std::unique_ptr<ExprAST> ParseIdentifierExpr();
                std::unique_ptr<ExprAST> ParseIfExpr();
                std::unique_ptr<ExprAST> ParseForExpr();
                std::unique_ptr<ExprAST> ParseVarExpr();
                std::unique_ptr<ExprAST> ParsePrimary();
                std::unique_ptr<ExprAST> ParseUnary();
                std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec, std::unique_ptr<ExprAST> LHS);
                std::unique_ptr<ExprAST> ParseExpression();
                ExprList ParseExpressionList();
                std::unique_ptr<PrototypeAST> ParsePrototype();
                std::unique_ptr<FunctionAST> ParseTopLevelTypedExpression();
                std::unique_ptr<PrototypeAST> ParseExtern();
                void HandleTypedExpression();
                void HandleExtern();
                void MainLoop();
        };
}; // namespace slljit
```

```
// Parser.cpp
#include "pch.h"
#include "Parser.h"
#include "AST.h"
using namespace llvm;
using namespace llvm::orc;
using namespace slljit;
using namespace std;
namespace slljit
{
	int Parser::getNextToken()
	{
		return CurTok = m_tokenizer.gettok();
	}
	Parser::Parser(std::map<char, int>& BinopPrecedence)
	    : BinopPrecedence(BinopPrecedence)
	{
	}
	void Parser::set_source(std::string_view source)
	{
		m_tokenizer.set_source(source);
		getNextToken();
	}
	std::pair<list<unique_ptr<PrototypeAST>>, list<unique_ptr<FunctionAST>>> Parser::get_ast()
	{
		return std::pair{std::move(PrototypeAST_list), std::move(FunctionAST_list)};
	}

	std::list<std::unique_ptr<FunctionAST>> Parser::get_function_ast()
	{
		return std::move(FunctionAST_list);
	}
	std::list<std::unique_ptr<PrototypeAST>> Parser::get_prototype_ast()
	{
		return std::move(PrototypeAST_list);
	}
	void Parser::parse()
	{
		MainLoop();
	}
	int Parser::GetTokPrecedence()
	{
		if (!isascii(CurTok))
			return -1;
		// Make sure it's a declared binop.
		int TokPrec = BinopPrecedence[CurTok];
		if (TokPrec <= 0)
			return -1;
		return TokPrec;
	}
	std::unique_ptr<ExprAST> Parser::ParseNumberExpr()
	{
		auto Result = std::make_unique<NumberExprAST>(m_tokenizer.get_double_val());
		getNextToken(); // consume the number
		return std::move(Result);
	}
	std::unique_ptr<ExprAST> Parser::ParseParenExpr()
	{
		getNextToken(); // eat (.
		auto V = ParseExpression();
		if (!V)
			return nullptr;
		if (CurTok != ')')
			return LogError("expected ')'");
		getNextToken(); // eat ).
		return V;
	}
	std::unique_ptr<ExprAST> Parser::ParseIdentifierExpr()
	{
		auto IdName = m_tokenizer.get_identifier();
		getNextToken(); // eat identifier.
		// return expression
```

```
		if (IdName == "return")
		{
			auto expr = ParseExpression();
			return std::make_unique<ReturnExprAST>(std::move(expr));
		}
		if (CurTok != '(') // Simple variable ref.
			return std::make_unique<VariableExprAST>(IdName);
		// Call.
		getNextToken(); // eat (
		std::vector<std::unique_ptr<ExprAST>> Args;
		if (CurTok != ')')
		{
			while (true)
			{
				if (auto Arg = ParseExpression())
					Args.push_back(std::move(Arg));
				else
					return nullptr;
				if (CurTok == ')')
					break;
				if (CurTok != ',')
					return LogError("Expected ')' or ',' in argument list");
				getNextToken();
			}
		}
		// Eat the ')'.
		getNextToken();
		return std::make_unique<CallExprAST>(IdName, std::move(Args));
	}
	std::unique_ptr<ExprAST> Parser::ParseIfExpr()
	{
		getNextToken(); // eat the if.
		if (CurTok != '(')
			return LogError("expected (");
		getNextToken(); // eat (
		// condition.
		auto Cond = ParseExpression();
		if (!Cond)
			return nullptr;
		if (CurTok != ')')
			return LogError("expected )");
		getNextToken(); // eat )
		auto ifBody = ParseExpressionList();
		if (ifBody.empty())
			return nullptr;
		if (CurTok == tok_else)
		{
			getNextToken(); // eat else
			auto ElseBody = ParseExpressionList();
			if (ElseBody.empty())
				return nullptr;
			return std::make_unique<IfExprAST>(std::move(Cond), std::move(ifBody),
std::move(ElseBody));
		}
		else
		{
			return std::make_unique<IfExprAST>(std::move(Cond), std::move(ifBody), ExprList());
		}
	}
	std::unique_ptr<ExprAST> Parser::ParseForExpr()
	{
		getNextToken(); // eat the for.
		if (CurTok != '(')
			return LogError("expected ( after for");
		getNextToken(); // eat (
		// g_var init expr.
		auto Varinit = ParseExpression();
		if (CurTok != ';')
			return LogError("Expected ;");
		getNextToken(); // Eat ;
		// Cond.
		auto Cond = ParseExpression();
```

```
                if (CurTok != ';')
                        return LogError("Expected ;");
                getNextToken(); // Eat ;
                // AfterloopExpr.
                auto Afterloop = ParseExpression();
                if (CurTok != ')')
                        return LogError("expected )");
                getNextToken(); // eat (
                // Body
                auto Body = ParseExpressionList();
                if (Body.empty())
                        return LogError("Empty loop body.");

                return std::make_unique<ForExprAST>(std::move(Varinit), std::move(Cond), std::move(Body),
std::move(Afterloop));
        }
        std::unique_ptr<ExprAST> Parser::ParseVarExpr()
        {
                getNextToken(); // eat type.
                std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;
                // At least one variable name is required.
                if (CurTok != tok_identifier)
                        return LogError("expected identifier after var");
                while (true)
                {
                        auto Name = m_tokenizer.get_identifier();
                        getNextToken(); // eat identifier.
                        // Read the optional initializer.
                        std::unique_ptr<ExprAST> Init = nullptr;
                        if (CurTok == '=')
                        {
                                getNextToken(); // eat the '='.
                                Init = ParseExpression();
                                if (!Init)
                                        return nullptr;
                        }
                        VarNames.push_back(std::make_pair(Name, std::move(Init)));
                        // End of g_var list, exit loop.
                        if (CurTok != ',')
                                break;
                        getNextToken(); // eat the ','.
                        if (CurTok != tok_identifier)
                                return LogError("expected identifier list after var");
                }
                return std::make_unique<VarExprAST>(std::move(VarNames));
        }
        std::unique_ptr<ExprAST> Parser::ParsePrimary()
        {
                switch (CurTok)
                {
                default: return LogError("unknown token when expecting an expression");
                case tok_identifier: return ParseIdentifierExpr();
                case tok_number: return ParseNumberExpr();
                case '(': return ParseParenExpr();
                case tok_if: return ParseIfExpr();
                case tok_for: return ParseForExpr();
                case tok_type: return ParseVarExpr();
                }
        }
        std::unique_ptr<ExprAST> Parser::ParseUnary()
        {
                // If the current token is not an operator, it must be a primary expr.
                if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
                        return ParsePrimary();
                // If this is a unary operator, read it.
                int Opc = CurTok;
                getNextToken();
                if (auto Operand = ParseUnary())
                        return std::make_unique<UnaryExprAST>(Opc, std::move(Operand));
                return nullptr;
        }
        std::unique_ptr<ExprAST> Parser::ParseBinOpRHS(int ExprPrec, std::unique_ptr<ExprAST> LHS)
```

```
{
	// If this is a binop, find its precedence.
	while (true)
	{
		int TokPrec = GetTokPrecedence();
		// If this is a binop that binds at least as tightly as the current binop,
		// consume it, otherwise we are done.
		if (TokPrec < ExprPrec)
			return LHS;
		// Okay, we know this is a binop.
		int BinOp = CurTok;
		getNextToken(); // eat binop
		// Parse the unary expression after the binary operator.
		auto RHS = ParseUnary();
		if (!RHS)
			return nullptr;
		// If BinOp binds less tightly with RHS than the operator after RHS, let
		// the pending operator take RHS as its LHS.
		int NextPrec = GetTokPrecedence();
		if (TokPrec < NextPrec)
		{
			RHS = ParseBinOpRHS(TokPrec + 1, std::move(RHS));
			if (!RHS)
				return nullptr;
		}
		// Merge LHS/RHS.
		LHS = std::make_unique<BinaryExprAST>(BinOp, std::move(LHS), std::move(RHS));
	}
}
std::unique_ptr<ExprAST> Parser::ParseExpression()
{
	if (CurTok == ';')
		return std::make_unique<NoOpAST>();

	auto LHS = ParseUnary();
	if (!LHS)
		return nullptr;
	return ParseBinOpRHS(0, std::move(LHS));
}
ExprList Parser::ParseExpressionList()
{
	if (CurTok != '{')
		return LogErrorEX("Expected {");
	getNextToken(); // Eat {
	ExprList e_list;
	while (true)
	{
		auto retval = ParseExpression();
		if (retval->bExpectSemicolon())
		{
			if (CurTok != ';')
				return LogErrorEX("Expected ;");
			getNextToken(); // Eat ;
		}
		e_list.push_back(std::move(retval));
		if (CurTok == '}')
			break;
	}
	getNextToken(); // Eat }
	return e_list;
}
std::unique_ptr<PrototypeAST> Parser::ParsePrototype()
{
	std::string FnName;
	unsigned Kind             = 0; // 0 = identifier, 1 = unary, 2 = binary.
	unsigned BinaryPrecedence = 30;
	switch (CurTok)
	{
	default: return LogErrorP("Expected function name in prototype");
	case tok_identifier:
		FnName = m_tokenizer.get_identifier();
		Kind   = 0;
```

```
                getNextToken();
                break;
        }
        if (CurTok != '(')
                return LogErrorP("Expected '(' in prototype");
        std::vector<std::string> ArgNames;
        while (getNextToken() == tok_type)
        {
                if (getNextToken() == tok_identifier)
                        ArgNames.push_back(m_tokenizer.get_identifier());
                else
                        return LogErrorP("Expected identifier after type in prototype");
                if (getNextToken() != ',')
                        break;
        }
        if (CurTok != ')')
                return LogErrorP("Expected ')' in prototype");
        // success.
        getNextToken(); // eat ')'.

        if (!Kind && FnName == "main" && ArgNames.size() != 0)
                return LogErrorP("Invalid number of operands for main function");
        // Verify right number of names for operator.
        if (Kind && ArgNames.size() != Kind)
                return LogErrorP("Invalid number of operands for operator");
        return std::make_unique<PrototypeAST>(FnName, ArgNames, Kind != 0, BinaryPrecedence);
}
std::unique_ptr<FunctionAST> Parser::ParseTopLevelTypedExpression()
{
        getNextToken(); // eat type.
        if (CurTok == tok_identifier)
        {
                auto Proto = ParsePrototype();
                if (!Proto)
                        return nullptr;
                auto E = ParseExpressionList();
                if (!E.empty())
                {
                        auto& P = *Proto;
                        PrototypeAST_list.emplace_back(std::move(Proto));
                        return std::make_unique<FunctionAST>(P, std::move(E));
                }
                return LogErrorF("Empty function");
        }
        else
        {
                return LogErrorF("Expected identifier after type");
        }
}
std::unique_ptr<PrototypeAST> Parser::ParseExtern()
{
        getNextToken(); // eat extern.
        return ParsePrototype();
}
void Parser::HandleTypedExpression()
{
        if (auto FnAST = ParseTopLevelTypedExpression())
        {
                FunctionAST_list.emplace_back(std::move(FnAST));
        }
        else
        {
                // Skip token for error recovery.
                getNextToken();
        }
}
void Parser::HandleExtern()
{
        if (auto ProtoAST = ParseExtern())
        {
                PrototypeAST_list.emplace_back(std::move(ProtoAST));
        else
```

```
                {
                        // Skip token for error recovery.
                        getNextToken();
                }
        }
        void Parser::MainLoop()
        {
                while (true)
                {
                        switch (CurTok)
                        {
                        case tok_eof: return;
                        case ';': // ignore top-level semicolons.
                                getNextToken();
                                break;
                        case tok_type: HandleTypedExpression(); break;
                        case tok_extern: HandleExtern(); break;
                        default: return; break;
                        }
                }
        }
}; // namespace slljit
// AST.h
#pragma once
#include <list>
#include <string>
#include <vector>
#include <memory>
#include "llvm/IR/Value.h"
#include "llvm/IR/Instructions.h"

namespace slljit
{
        using namespace llvm;
        using namespace std;
        class Context;
        class LocalContext;
        /// ExprAST - Base class for all expression nodes.
        class ExprAST
        {
        public:
                virtual ~ExprAST() = default;
                virtual bool bExpectSemicolon()
                {
                        return true;
                }
                virtual bool bIsNoOp()
                {
                        return false;
                }
                virtual bool bIsTerminator()
                {
                        return false;
                }
                virtual Value* codegen(Context& m_context, LocalContext& m_local_context) = 0;
        };
        typedef std::list<std::unique_ptr<ExprAST>> ExprList;
        class NoOpAST : public ExprAST
        {
        public:
                NoOpAST()
                {
                }
                virtual bool bIsNoOp() override
                {
                        return true;
                }
                Value* codegen(Context& m_context, LocalContext& m_local_context) override;
        };
        /// NumberExprAST - Expression class for numeric literals like "1.0".
        class NumberExprAST : public ExprAST
        {
```

```
        double Val;
public:
        NumberExprAST(double Val)
            : Val(Val)
        {
        }
        Value* codegen(Context& m_context, LocalContext& m_local_context) override;
};
/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST
{
        std::string Name;
public:
        VariableExprAST(const std::string& Name)
            : Name(Name)
        {
        }
        Value* codegen(Context& m_context, LocalContext& m_local_context) override;
        const std::string& getName() const
        {
                return Name;
        }
};
/// UnaryExprAST - Expression class for a unary operator.
class UnaryExprAST : public ExprAST
{
        char Opcode;
        std::unique_ptr<ExprAST> Operand;
public:
        UnaryExprAST(char Opcode, std::unique_ptr<ExprAST> Operand)
            : Opcode(Opcode)
            , Operand(std::move(Operand))
        {
        }
        Value* codegen(Context& m_context, LocalContext& m_local_context) override;
};
/// UnaryExprAST - Expression class for a unary operator.
class ReturnExprAST : public ExprAST
{
        std::unique_ptr<ExprAST> Operand;
public:
        ReturnExprAST(std::unique_ptr<ExprAST> Operand)
            : Operand(std::move(Operand))
        {
        }
        virtual bool bIsTerminator() override
        {
                return true;
        }
        Value* codegen(Context& m_context, LocalContext& m_local_context) override;
};
/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST
{
        char Op;
        std::unique_ptr<ExprAST> LHS, RHS;
public:
        BinaryExprAST(char Op, std::unique_ptr<ExprAST> LHS, std::unique_ptr<ExprAST> RHS)
            : Op(Op)
            , LHS(std::move(LHS))
            , RHS(std::move(RHS))
        {
        }
        Value* codegen(Context& m_context, LocalContext& m_local_context) override;
};
/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST
{
        std::string Callee;
        std::vector<std::unique_ptr<ExprAST>> Args;
public:
        CallExprAST(const std::string& Callee, std::vector<std::unique_ptr<ExprAST>> Args)
```

```
                    : Callee(Callee)
                    , Args(std::move(Args))
                {
                }
                Value* codegen(Context& m_context, LocalContext& m_local_context) override;
        };
        /// IfExprAST - Expression class for if/else.
        class IfExprAST : public ExprAST
        {
                std::unique_ptr<ExprAST> Cond;
                ExprList Then;
                ExprList Else;
        public:
                IfExprAST(std::unique_ptr<ExprAST> Cond, ExprList Then, ExprList Else)
                    : Cond(std::move(Cond))
                    , Then(std::move(Then))
                    , Else(std::move(Else))
                {
                }
                bool bExpectSemicolon() override
                {
                        return false;
                }
                Value* codegen(Context& m_context, LocalContext& m_local_context) override;
        };
        /// ForExprAST - Expression class for for/in.
        class ForExprAST : public ExprAST
        {
                std::unique_ptr<ExprAST> VarInit, Cond;
                ExprList Body;
                std::unique_ptr<ExprAST> EndExpr;
        public:
                ForExprAST(std::unique_ptr<ExprAST> VarInit, std::unique_ptr<ExprAST> Cond, ExprList Body,
std::unique_ptr<ExprAST> EndExpr)
                    : VarInit(std::move(VarInit))
                    , Cond(std::move(Cond))
                    , Body(std::move(Body))
                    , EndExpr(std::move(EndExpr))
                {
                }
                bool bExpectSemicolon() override
                {
                        return false;
                }
                Value* codegen(Context& m_context, LocalContext& m_local_context) override;
        };
        /// VarExprAST - Expression class for g_var/in
        class VarExprAST : public ExprAST
        {
                std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;
        public:
                VarExprAST(std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames)
                    : VarNames(std::move(VarNames))
                {
                }
                Value* codegen(Context& m_context, LocalContext& m_local_context) override;
        };
        /// PrototypeAST - This class represents the "prototype" for a function
        class PrototypeAST : public ExprAST
        {
                std::string Name;
                std::vector<std::string> Args;
                bool IsOperator;
                unsigned Precedence; // Precedence if a binary op.
        public:
                PrototypeAST(const std::string& Name, std::vector<std::string> Args, bool IsOperator = false,
unsigned Prec = 0)
                    : Name(Name)
                    , Args(std::move(Args))
                    , IsOperator(IsOperator)
                    , Precedence(Prec)
                {
```

```
                }
                // Cast to Function*
                Value* codegen(Context& m_context, LocalContext& m_local_context);
                const std::string& getName() const
                {
                        return Name;
                }
                bool isUnaryOp() const
                {
                        return IsOperator && Args.size() == 1;
                }
                bool isBinaryOp() const
                {
                        return IsOperator && Args.size() == 2;
                }
                char getOperatorName() const
                {
                        assert(isUnaryOp() || isBinaryOp());
                        return Name[Name.size() - 1];
                }
                unsigned getBinaryPrecedence() const
                {
                        return Precedence;
                }
        };
        /// FunctionAST - This class represents a function definition itself.
        class FunctionAST : public ExprAST
        {
                PrototypeAST& Proto;
                ExprList Body;
        public:
                FunctionAST(PrototypeAST& Proto, ExprList Body)
                    : Proto(Proto)
                    , Body(std::move(Body))
                {
                }
                // Cast to Function*
                // Cast to Function*
                Value* codegen(Context& m_context, LocalContext& m_local_context);
        };
        std::unique_ptr<ExprAST> LogError(const char* Str);
        std::unique_ptr<PrototypeAST> LogErrorP(const char* Str);
        std::unique_ptr<FunctionAST> LogErrorF(const char* Str);
        ExprList LogErrorEX(const char* Str);
        Value* LogErrorV(const char* Str);
        Function* getFunction(std::string Name, Context& m_context, LocalContext& m_local_context);
        static AllocaInst* CreateEntryBlockAlloca(Function* TheFunction, const StringRef VarName, Context&
m_context, LocalContext& m_local_context);
}; // namespace slljit
// AST.cpp
#include "pch.h"
#include "AST.h"
#include "Context.h"
using namespace llvm;
using namespace llvm::orc;
using namespace slljit;
using namespace std;
namespace slljit
{
        Value* BinaryExprAST::codegen(Context& m_context, LocalContext& m_local_context)
        {
                // Special case '=' because we don't want to emit the LHS as an expression.
                if (Op == '=')
                {
                        VariableExprAST* LHSE = static_cast<VariableExprAST*>(LHS.get());
                        if (!LHSE)
                                return LogErrorV("destination of '=' must be a variable");
                        // Codegen the RHS.
                        Value* Val = RHS->codegen(m_context, m_local_context);
                        if (!Val)
                                return nullptr;
                        // Look up global
```

```
                        GlobalVariable* gVar = m_local_context.LLVM_Module->getNamedGlobal(LHSE->getName());
                        if (gVar)
                        {
                                if (gVar->isConstant())
                                        return LogErrorV("Trying to store in constant global");
                                auto ptr = m_context.LLVM_Builder.CreateLoad(gVar, LHSE->getName());
                                m_context.LLVM_Builder.CreateStore(Val, ptr);
                                return Val;
                        }
                        // Look up variable.
                        Value* Variable = m_local_context.NamedValues[LHSE->getName()];
                        if (!Variable)
                                return LogErrorV("Unknown variable name");
                        m_context.LLVM_Builder.CreateStore(Val, Variable);
                        return Val;
                }
                Value* L = LHS->codegen(m_context, m_local_context);
                Value* R = RHS->codegen(m_context, m_local_context);
                if (!L || !R)
                        return nullptr;
                switch (Op)
                {
                case '+': return m_context.LLVM_Builder.CreateFAdd(L, R, "addtmp");
                case '-': return m_context.LLVM_Builder.CreateFSub(L, R, "subtmp");
                case '*': return m_context.LLVM_Builder.CreateFMul(L, R, "multmp");
                case '/': return m_context.LLVM_Builder.CreateFDiv(L, R, "divtmp");
                case '<':
                        L = m_context.LLVM_Builder.CreateFCmpULT(L, R, "cmptmp");
                        // Convert bool 0/1 to double 0.0 or 1.0
                        return m_context.LLVM_Builder.CreateUIToFP(L, Type::getDoubleTy(m_context.LLVM_Context),
"booltmp");
                case '>':
                        L = m_context.LLVM_Builder.CreateFCmpUGT(L, R, "cmptmp");
                        // Convert bool 0/1 to double 0.0 or 1.0
                        return m_context.LLVM_Builder.CreateUIToFP(L, Type::getDoubleTy(m_context.LLVM_Context),
"booltmp");
                default: break;
                }
                // If it wasn't a builtin binary operator, it must be a user defined one. Emit
                // a call to it.
                Function* F = getFunction(std::string("binary") + Op, m_context, m_local_context);
                assert(F && "binary operator not found!");
                Value* Ops[] = {L, R};
                return m_context.LLVM_Builder.CreateCall(F, Ops, "binop");
        }
        std::unique_ptr<ExprAST> LogError(const char* Str)
        {
                fprintf(stderr, "Error: %s\n", Str);
                return nullptr;
        }
        std::unique_ptr<PrototypeAST> LogErrorP(const char* Str)
        {
                LogError(Str);
                return nullptr;
        }
        std::unique_ptr<FunctionAST> LogErrorF(const char* Str)
        {
                LogError(Str);
                return nullptr;
        }
        ExprList LogErrorEX(const char* Str)
        {
                LogError(Str);
                return {};
        }
        Value* LogErrorV(const char* Str)
        {
                LogError(Str);
                return nullptr;
        }
        Function* getFunction(std::string Name, Context& m_context, LocalContext& m_local_context)
        {
```

```
		if (auto* F = m_local_context.LLVM_Module->getFunction(Name))
			return F;
		auto FI = m_local_context.FunctionProtos.find(Name);
		if (FI != m_local_context.FunctionProtos.end())
			return static_cast<Function*>(FI->second->codegen(m_context, m_local_context));
		return nullptr;
	}
	AllocaInst* CreateEntryBlockAlloca(Function* TheFunction, const StringRef VarName, Context& m_context,
LocalContext& m_local_context)
	{
		IRBuilder<> TmpB(&TheFunction->getEntryBlock(), TheFunction->getEntryBlock().begin());
		return TmpB.CreateAlloca(Type::getDoubleTy(m_context.LLVM_Context), nullptr, VarName);
	}
	Value* NoOpAST::codegen(Context& m_context, LocalContext& m_local_context)
	{
		return ConstantFP::get(m_context.LLVM_Context, APFloat(0.0));
	}
	Value* NumberExprAST::codegen(Context& m_context, LocalContext& m_local_context)
	{
		return ConstantFP::get(m_context.LLVM_Context, APFloat(Val));
	}
	Value* VariableExprAST::codegen(Context& m_context, LocalContext& m_local_context)
	{
		// Look this variable up in the function.
		GlobalVariable* gVar = m_local_context.LLVM_Module->getNamedGlobal(Name);
		if (gVar)
		{
			if (gVar->isConstant())
				return m_context.LLVM_Builder.CreateLoad(gVar, Name);
			else
			{
				auto ptr = m_context.LLVM_Builder.CreateLoad(gVar, Name);
				return m_context.LLVM_Builder.CreateLoad(ptr, Name);
			}
		}
		Value* V = m_local_context.NamedValues[Name];
		if (!V)
			return LogErrorV("Unknown variable name");
		return m_context.LLVM_Builder.CreateLoad(V, Name.c_str());
	}
	Value* UnaryExprAST::codegen(Context& m_context, LocalContext& m_local_context)
	{
		Value* OperandV = Operand->codegen(m_context, m_local_context);
		if (!OperandV)
			return nullptr;
		Function* F = getFunction(std::string("unary") + Opcode, m_context, m_local_context);
		if (!F)
			return LogErrorV("Unknown unary operator");
		return m_context.LLVM_Builder.CreateCall(F, OperandV, "unop");
	}
	Value* ReturnExprAST::codegen(Context& m_context, LocalContext& m_local_context)
	{
		if (Value* RetVal = Operand->codegen(m_context, m_local_context))
		{
			// Finish off the function.
			m_context.LLVM_Builder.CreateRet(RetVal);
			return RetVal;
		}
		return nullptr;
	}
	Value* CallExprAST::codegen(Context& m_context, LocalContext& m_local_context)
	{
		// Look up the name in the global module table.
		Function* CalleeF = getFunction(Callee, m_context, m_local_context);
		if (!CalleeF)
			return LogErrorV("Unknown function referenced");
		// If argument mismatch error.
		if (CalleeF->arg_size() != Args.size())
			return LogErrorV("Incorrect # arguments passed");
		std::vector<Value*> ArgsV;
		for (unsigned i = 0, e = Args.size(); i != e; ++i)
		{
```

```
                        ArgsV.push_back(Args[i]->codegen(m_context, m_local_context));
                        if (!ArgsV.back())
                                return nullptr;
                }
                return m_context.LLVM_Builder.CreateCall(CalleeF, ArgsV, "calltmp");
        }
        Value* IfExprAST::codegen(Context& m_context, LocalContext& m_local_context)
        {
                Function* TheFunction = m_context.LLVM_Builder.GetInsertBlock()->getParent();
                BasicBlock* IfBB    = BasicBlock::Create(m_context.LLVM_Context, "if", TheFunction);
                BasicBlock* ElseBB  = BasicBlock::Create(m_context.LLVM_Context, "else", TheFunction);
                BasicBlock* MergeBB = BasicBlock::Create(m_context.LLVM_Context, "ifcont", TheFunction);
                Value* CondV = Cond->codegen(m_context, m_local_context);
                if (!CondV)
                        return nullptr;
                // Convert condition to a bool by comparing non-equal to 0.0.
                CondV = m_context.LLVM_Builder.CreateFCmpONE(CondV, ConstantFP::get(m_context.LLVM_Context,
APFloat(0.0)), "ifcond");
                m_context.LLVM_Builder.CreateCondBr(CondV, IfBB, ElseBB);
                // Emit then value.
                m_context.LLVM_Builder.SetInsertPoint(IfBB);
                {
                        bool isTerminated = false;
                        for (auto& then_expr : Then)
                        {
                                then_expr->codegen(m_context, m_local_context);
                                if (then_expr->bIsTerminator())
                                {
                                        isTerminated = true;
                                        break;
                                }
                        }
                        if (!isTerminated)
                                m_context.LLVM_Builder.CreateBr(MergeBB);
                }
                // Emit else block.
                //      TheFunction->getBasicBlockList().push_back(ElseBB);
                m_context.LLVM_Builder.SetInsertPoint(ElseBB);
                {
                        bool isTerminated = false;
                        for (auto& else_expr : Else)
                        {
                                else_expr->codegen(m_context, m_local_context);
                                if (else_expr->bIsTerminator())
                                {
                                        isTerminated = true;
                                        break;
                                }
                        }
                        if (!isTerminated)
                                m_context.LLVM_Builder.CreateBr(MergeBB);
                }
                m_context.LLVM_Builder.SetInsertPoint(MergeBB);
                return CondV;
        }
        Value* ForExprAST::codegen(Context& m_context, LocalContext& m_local_context)
        {
                Function* TheFunction = m_context.LLVM_Builder.GetInsertBlock()->getParent();
                // generate init variable
                if (VarInit)
                        VarInit->codegen(m_context, m_local_context);
                BasicBlock* LoopBB = BasicBlock::Create(m_context.LLVM_Context, "loop", TheFunction);
                BasicBlock* LoopBobyBB = BasicBlock::Create(m_context.LLVM_Context, "loop_body", TheFunction);
                BasicBlock* AfterBB = BasicBlock::Create(m_context.LLVM_Context, "afterloop", TheFunction);
                m_context.LLVM_Builder.CreateBr(LoopBB);
                m_context.LLVM_Builder.SetInsertPoint(LoopBB);
                if (!Cond->bIsNoOp())
                {
                        Value* condVal = Cond->codegen(m_context, m_local_context);
                        condVal = m_context.LLVM_Builder.CreateFCmpONE(condVal,
ConstantFP::get(m_context.LLVM_Context, APFloat(0.0)), "loopcond");
                        m_context.LLVM_Builder.CreateCondBr(condVal, LoopBobyBB, AfterBB);
```

```
                }
                m_context.LLVM_Builder.SetInsertPoint(LoopBobyBB);
                for (auto& expt : Body)
                {
                        expt->codegen(m_context, m_local_context);
                }
                EndExpr->codegen(m_context, m_local_context);
                m_context.LLVM_Builder.CreateBr(LoopBB);
                // Any new code will be inserted in AfterBB.
                m_context.LLVM_Builder.SetInsertPoint(AfterBB);
                //      LLVM_Builder.CreateFCmpONE(ConstantFP::get(LLVM_Context, APFloat(0.0)),
ConstantFP::get(LLVM_Context, APFloat(0.0)), "dmm");
                // for expr always returns 0.0.
                return ConstantFP::get(m_context.LLVM_Context, APFloat(0.0));
        }
        Value* VarExprAST::codegen(Context& m_context, LocalContext& m_local_context)
        {
                std::vector<AllocaInst*> OldBindings;
                Function* TheFunction = m_context.LLVM_Builder.GetInsertBlock()->getParent();
                Value* retval = nullptr;
                for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
                {
                        const std::string& VarName = VarNames[i].first;
                        ExprAST* Init              = VarNames[i].second.get();
                        Value* InitVal;
                        if (Init)
                        {
                                InitVal = Init->codegen(m_context, m_local_context);
                                if (!InitVal)
                                        return nullptr;
                        }
                        else
                        { // If not specified, use 0.0.
                                InitVal = ConstantFP::get(m_context.LLVM_Context, APFloat(0.0));
                        }
                        AllocaInst* Alloca = CreateEntryBlockAlloca(TheFunction, VarName, m_context,
m_local_context);
                        retval             = m_context.LLVM_Builder.CreateStore(InitVal, Alloca);
                        OldBindings.push_back(m_local_context.NamedValues[VarName]);
                        m_local_context.NamedValues[VarName] = Alloca;
                }
                return retval;
        }
        // Cast to Function*
        Value* PrototypeAST::codegen(Context& m_context, LocalContext& m_local_context)
        {
                // Make the function type:  double(double,double) etc.
                std::vector<Type*> Doubles(Args.size(), Type::getDoubleTy(m_context.LLVM_Context));
                FunctionType* FT = FunctionType::get(Type::getDoubleTy(m_context.LLVM_Context), Doubles, false);

                Function* F = Function::Create(FT, Function::ExternalLinkage, Name,
m_local_context.LLVM_Module.get());
                unsigned Idx = 0;
                for (auto& Arg : F->args())
                        Arg.setName(Args[Idx++]);
                return F;
        }
        // Cast to Function*
        Value* FunctionAST::codegen(Context& m_context, LocalContext& m_local_context)
        {
                Function* TheFunction = getFunction(Proto.getName(), m_context, m_local_context);
                if (!TheFunction)
                        return nullptr;
                // If this is an operator, install it.
                if (Proto.isBinaryOp())
                        m_local_context.BinopPrecedence[Proto.getOperatorName()] = Proto.getBinaryPrecedence();
                BasicBlock* BB = BasicBlock::Create(m_context.LLVM_Context, "entry", TheFunction);
                m_context.LLVM_Builder.SetInsertPoint(BB);
                // Record the function arguments in the NamedValues map.
                m_local_context.NamedValues.clear();
                for (auto& Arg : TheFunction->args())
                {
```

```
                        AllocaInst* Alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName(), m_context,
m_local_context);
                        m_context.LLVM_Builder.CreateStore(&Arg, Alloca);
                        m_local_context.NamedValues[std::string(Arg.getName())] = Alloca;
                }
                for (auto& expression : Body)
                {
                        auto test = expression->codegen(m_context, m_local_context);
                        if (!test)
                        {
                                // Error reading body, remove function.
                                TheFunction->eraseFromParent();
                                if (Proto.isBinaryOp())
                                        m_local_context.BinopPrecedence.erase(Proto.getOperatorName());
                                return nullptr;
                        }
                }
                verifyFunction(*TheFunction);
                m_local_context.LLVM_FPM->doInitialization();
                m_local_context.LLVM_FPM->run(*TheFunction);
                m_local_context.LLVM_FPM->doFinalization();
                return TheFunction;
        }
}; // namespace slljit
// Context.h
#pragma once
#include "pch.h"
#include "JIT.h"
#include "AST.h"
namespace slljit
{
        using namespace llvm;
        using namespace std;
        static void init__();
        class Context
        {
        public:
                LLVMContext LLVM_Context;
                IRBuilder<> LLVM_Builder;
                std::unique_ptr<ShaderJIT> shllJIT;
        public:
                Context();
        };
        class LocalContext
        {
        public:
                std::unique_ptr<Module> LLVM_Module;
                std::map<std::string, AllocaInst*> NamedValues;
                std::unique_ptr<legacy::FunctionPassManager> LLVM_FPM;
                std::unique_ptr<legacy::PassManager> LLVM_PM;
                std::map<std::string, std::unique_ptr<PrototypeAST>> FunctionProtos;
                std::map<char, int> BinopPrecedence;
                VModuleKey module_key;
        public:
                LocalContext(Context& m_context);
                void set_key(VModuleKey module_key);
                auto get_key();
        };
}; // namespace slljit
// Context.cpp
#include "pch.h"
#include "Context.h"
using namespace llvm;
using namespace llvm::orc;
using namespace slljit;
using namespace std;
namespace slljit
{
        void init__()
        {
                static bool b_once = true;
                if (b_once)
```

```
                {
                        InitializeNativeTarget();
                        InitializeNativeTargetAsmPrinter();
                        InitializeNativeTargetAsmParser();
                        b_once = false;
                }
        }
        Context::Context()
            : LLVM_Builder(LLVM_Context)
        {
                init__();
                shllJIT = std::make_unique<ShaderJIT>();
        }
        LocalContext::LocalContext(Context& m_context)
        {
                BinopPrecedence = {{'=', 2}, {'<', 10}, {'>', 10}, {'+', 20}, {'-', 20}, {'*', 40}, {'/', 40}};
                LLVM_Module = std::make_unique<Module>("my cool jit", m_context.LLVM_Context);
                LLVM_Module->setDataLayout(m_context.shllJIT->getTargetMachine().createDataLayout());
                LLVM_FPM = std::make_unique<legacy::FunctionPassManager>(LLVM_Module.get());
                LLVM_PM  = std::make_unique<legacy::PassManager>();

                LLVM_FPM->add(createCFGSimplificationPass());       //   Dead code elimination
                LLVM_FPM->add(createPromoteMemoryToRegisterPass()); //   SSA conversion
                LLVM_FPM->add(createSROAPass());
                LLVM_FPM->add(createLoopSimplifyCFGPass());
                LLVM_FPM->add(createLoadStoreVectorizerPass());
                LLVM_FPM->add(createLoopVectorizePass());
                LLVM_FPM->add(createLoopUnrollPass());
                LLVM_FPM->add(createGVNPass());         //       Eliminate Common SubExpressions.
                LLVM_FPM->add(createNewGVNPass());      //       Global value numbering
                LLVM_FPM->add(createReassociatePass()); //       Reassociate expressions.
                LLVM_FPM->add(createConstantPropagationPass());
                LLVM_FPM->add(createPartiallyInlineLibCallsPass()); //   standard calls
                LLVM_FPM->add(createDeadCodeEliminationPass());
                LLVM_FPM->add(createAggressiveDCEPass());
                LLVM_FPM->add(createCFGSimplificationPass());    //      Cleanup
                LLVM_FPM->add(createInstructionCombiningPass()); //      Do simple "peephole" optimizations and
bit-twiddling optzns.
                LLVM_FPM->add(createSLPVectorizerPass());
                LLVM_FPM->add(createFlattenCFGPass()); //        Flatten the control flow graph.
                LLVM_PM->add(createFunctionInliningPass());
                LLVM_PM->add(createDeadInstEliminationPass());
        }
        void LocalContext::set_key(VModuleKey module_key)
        {
                this->module_key = module_key;
        }
        auto LocalContext::get_key()
        {
                return this->module_key;
        }
}; // namespace slljit
// Layout.h
#pragma once
#include <list>
#include <string>
#include <vector>
#include <memory>
#include <map>
#include <set>
namespace slljit
{
        using namespace std;
        enum LayoutVarTypes
        {
                Kdouble = 0
        };
        struct GlobalDefinition
        {
                string name;
                LayoutVarTypes type;
                size_t offset;
```

```
        };
        struct ConstantGlobalDefinition
        {
                double value;
                LayoutVarTypes type;
        };
        class Layout
        {
        public:
                set<string> names;
                vector<GlobalDefinition> globals;
                vector<size_t> global_offsets;
                map<string, ConstantGlobalDefinition> constant_globals;
                void addMember(string name, LayoutVarTypes type, size_t offset);
                void addConsatant(string name, double value, LayoutVarTypes type);
        };
}; // namespace slljit
// Layout.cpp
#include "pch.h"
#include "Layout.h"
using namespace slljit;
using namespace llvm;
using namespace llvm::orc;
using namespace std;
namespace slljit
{
        void Layout::addMember(string name, LayoutVarTypes type, size_t offset)
        {
                globals.emplace_back(GlobalDefinition{name, type, offset});
                names.insert(name);
                global_offsets.emplace_back(offset);
        }
        void Layout::addConsatant(string name, double value, LayoutVarTypes type)
        {
                constant_globals.emplace(pair{name, ConstantGlobalDefinition{value, type}});
                names.insert(name);
        }
}; // namespace slljit
// CodeGen.h
#pragma once
#include <map>
#include <memory>
#include <string>
#include <vector>
#include <charconv>
#include <list>
#include "Context.h"
#include "AST.h"
#include "Layout.h"
namespace slljit
{
        using namespace llvm;
        using namespace llvm::orc;
        using namespace std;
        class Layout;
        class CodeGen
        {
        public:
                void compile_layout(Context& m_context, LocalContext& m_local_context, Layout& m_layout);
                void compile(std::list<std::unique_ptr<PrototypeAST>> prototypes,
std::list<std::unique_ptr<FunctionAST>> functions, Context& m_context, LocalContext& m_local_context);
        };
}; // namespace slljit
// CodeGen.cpp
#include "pch.h"
#include "CodeGen.h"
#include "Layout.h"
using namespace slljit;
using namespace llvm;
using namespace llvm::orc;
using namespace std;
namespace slljit
```

```
{
	void CodeGen::compile_layout(Context& m_context, LocalContext& m_local_context, Layout& m_layout)
	{
		std::vector<Type*> StructMembers;
		StructMembers.reserve(m_layout.globals.size());
		for (auto& global : m_layout.globals)
		{
			m_local_context.LLVM_Module->getOrInsertGlobal(global.name,
Type::getDoublePtrTy(m_context.LLVM_Context));
			GlobalVariable* gVar = m_local_context.LLVM_Module->getNamedGlobal(global.name);
			gVar->setLinkage(GlobalValue::ExternalLinkage);
			gVar-
>setInitializer(ConstantPointerNull::get(Type::getDoublePtrTy(m_context.LLVM_Context)));
			switch (global.type)
			{
			case slljit::Kdouble:
StructMembers.emplace_back(Type::getDoubleTy(m_context.LLVM_Context));
			default: break;
			}
		}
		auto loader_struct_type = StructType::create(m_context.LLVM_Context, StructMembers, "layout__",
false);
		std::vector<Type*> Args(1, loader_struct_type->getPointerTo());
		FunctionType* FT = FunctionType::get(Type::getVoidTy(m_context.LLVM_Context), Args, false);
		Function* TheFunction = Function::Create(FT, Function::ExternalLinkage, "__layout_loader_",
m_local_context.LLVM_Module.get());
		auto arg            = TheFunction->arg_begin();
		Value* data_pointer = arg;
		arg->setName("data_ptr");
		BasicBlock* BB = BasicBlock::Create(m_context.LLVM_Context, "entry", TheFunction);
		m_context.LLVM_Builder.SetInsertPoint(BB);
		auto& g_list = m_local_context.LLVM_Module->getGlobalList();
		std::vector<Value*> indices{m_context.LLVM_Builder.getInt32(0),
m_context.LLVM_Builder.getInt32(0)};
		uint32_t idx = 0;
		for (auto& g_var : g_list)
		{
			Value* gep = m_context.LLVM_Builder.CreateGEP(data_pointer, indices);
			m_context.LLVM_Builder.CreateStore(gep, &g_var);
			idx++;
			indices[1] = m_context.LLVM_Builder.getInt32(idx);
		}
		m_context.LLVM_Builder.CreateRet(nullptr);
		for (auto& global : m_layout.constant_globals)
		{
			m_local_context.LLVM_Module->getOrInsertGlobal(global.first,
Type::getDoubleTy(m_context.LLVM_Context));
			GlobalVariable* gVar = m_local_context.LLVM_Module->getNamedGlobal(global.first);
			gVar->setLinkage(GlobalValue::ExternalLinkage);
			gVar->setInitializer(ConstantFP::get(m_context.LLVM_Context,
APFloat(global.second.value)));
			gVar->setConstant(true);
		}
	}
	void CodeGen::compile(std::list<std::unique_ptr<PrototypeAST>> prototypes,
std::list<std::unique_ptr<FunctionAST>> functions, Context& m_context, LocalContext& m_local_context)
	{
		for (auto it = prototypes.begin(); it != prototypes.end(); ++it)
		{
			if (auto* FnIR = static_cast<Function*>((*it)->codegen(m_context, m_local_context)))
			{
				m_local_context.FunctionProtos[(*it)->getName()] = std::move((*it));
			}
		}
		for (auto it = functions.begin(); it != functions.end(); ++it)
		{
			if (auto* FnIR = static_cast<Function*>((*it)->codegen(m_context, m_local_context)))
			{
				fprintf(stderr, "\n");
			}
		}
		m_local_context.LLVM_PM->run(*m_local_context.LLVM_Module.get());
```

```
                m_local_context.LLVM_Module->dump();
                auto key = m_context.shllJIT->addModule(std::move(m_local_context.LLVM_Module));
                m_local_context.set_key(key);
        }
}; // namespace slljit
// Program.hpp
#pragma once
#include <list>
#include <string>
#include <vector>
#include <memory>
#include <map>
#include <set>
#include "Tokenizer.h"
#include "Parser.h"
#include "CodeGen.h"
#include "Context.h"
#include "Layout.h"
namespace slljit
{
        using namespace std;
        template <class T>
        class Program
        {
        protected:
                Context& m_context;
                CodeGen* m_codegen_ptr = nullptr;
                LocalContext m_local_context;
                Layout m_layout;
                double (*main_func)() = nullptr;
                void (*loader__)(T*)  = nullptr;
        public:
                Program(Context& m_context)
                    : m_context(m_context)
                    , m_local_context(m_context)
                {
                }
                void compile(string body, Layout layout)
                {
                        m_layout = layout;
                        Parser m_parser(m_local_context.BinopPrecedence);
                        CodeGen m_codegen;
                        m_codegen_ptr = &m_codegen;
                        m_parser.set_source(body);
                        m_parser.parse();
                        auto [prototypes, functions] = m_parser.get_ast();
                        m_codegen.compile_layout(m_context, m_local_context, m_layout);
                        m_codegen.compile(std::move(prototypes), std::move(functions), m_context,
m_local_context);
                        auto loader_symbol = m_context.shllJIT->findSymbol("__layout_loader_",
m_local_context.module_key);
                        auto symbol        = m_context.shllJIT->findSymbol("main", m_local_context.module_key);
                        main_func          = (double (*)())(intptr_t)cantFail(symbol.getAddress());
                        loader__           = (void (*)(T*))(intptr_t)cantFail(loader_symbol.getAddress());
                }
                double run(T* data)
                {
                        loader__(data);
                        auto retval = main_func();
                        return retval;
                }
        };
}; // namespace slljit
// JIT.h
#pragma once
#ifndef LLVM_EXECUTIONENGINE_ORC_SHADERJIT_H
#define LLVM_EXECUTIONENGINE_ORC_SHADERJIT_H
#include "llvm/ADT/STLExtras.h"
#include "llvm/ADT/iterator_range.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/ExecutionEngine/JITSymbol.h"
#include "llvm/ExecutionEngine/Orc/CompileUtils.h"
```

```
#include "llvm/ExecutionEngine/Orc/IRCompileLayer.h"
#include "llvm/ExecutionEngine/Orc/LambdaResolver.h"
#include "llvm/ExecutionEngine/Orc/RTDyldObjectLinkingLayer.h"
#include "llvm/ExecutionEngine/RTDyldMemoryManager.h"
#include "llvm/ExecutionEngine/SectionMemoryManager.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/Mangler.h"
#include "llvm/Support/DynamicLibrary.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetMachine.h"
#include <algorithm>
#include <map>
#include <memory>
#include <string>
#include <vector>
namespace sllJit
{
	using namespace llvm;
	using namespace orc;
	class ShaderJIT
	{
	public:
		using ObjLayerT     = LegacyRTDyldObjectLinkingLayer;
		using CompileLayerT = LegacyIRCompileLayer<ObjLayerT, SimpleCompiler>;
		ShaderJIT()
		    : Resolver(createLegacyLookupResolver(
		          ES, [this](StringRef Name) { return findMangledSymbol(std::string(Name)); }, [](Error
Err) { cantFail(std::move(Err), "lookupFlags failed"); }))
		    , TM(EngineBuilder().selectTarget())
		    , DL(TM->createDataLayout())
		    , ObjectLayer(AcknowledgeORCv1Deprecation, ES,
		          [this](VModuleKey) {
		                  return ObjLayerT::Resources{std::make_shared<SectionMemoryManager>(),
Resolver};
		          })
		    , CompileLayer(AcknowledgeORCv1Deprecation, ObjectLayer, SimpleCompiler(*TM))
		{
		        sys::DynamicLibrary::LoadLibraryPermanently(nullptr);
		}
		TargetMachine& getTargetMachine()
		{
		        return *TM;
		}

		VModuleKey addModule(std::unique_ptr<Module> M)
		{
		        auto K = ES.allocateVModule();
		        cantFail(CompileLayer.addModule(K, std::move(M)));
		        ModuleKeys.push_back(K);
		        return K;
		}
		void removeModule(VModuleKey K)
		{
		        ModuleKeys.erase(find(ModuleKeys, K));
		        cantFail(CompileLayer.removeModule(K));
		}
		JITSymbol findSymbol(const std::string Name)
		{
		        return findMangledSymbol(mangle(Name));
		}
		JITSymbol findSymbol(const std::string Name, VModuleKey Module)
		{
		        return findMangledSymbol(mangle(Name), Module);
		}

	private:
		std::string mangle(const std::string& Name)
		{
		        std::string MangledName;
		        {
		                raw_string_ostream MangledNameStream(MangledName);
		                Mangler::getNameWithPrefix(MangledNameStream, Name, DL);
```

```
                        }
                        return MangledName;
                }
                JITSymbol findMangledSymbol(const std::string& Name, VModuleKey Module)
                {
#ifdef _WIN32
                        const bool ExportedSymbolsOnly = false;
#else
                        const bool ExportedSymbolsOnly = true;
#endif
                        if (auto Sym = CompileLayer.findSymbolIn(Module, Name, ExportedSymbolsOnly))
                                return Sym;
                        if (auto SymAddr = RTDyldMemoryManager::getSymbolAddressInProcess(Name))
                                return JITSymbol(SymAddr, JITSymbolFlags::Exported);

#ifdef _WIN32
                        if (Name.length() > 2 && Name[0] == '_')
                                if (auto SymAddr =
RTDyldMemoryManager::getSymbolAddressInProcess(Name.substr(1)))
                                        return JITSymbol(SymAddr, JITSymbolFlags::Exported);
#endif
                        return nullptr;
                }
                JITSymbol findMangledSymbol(const std::string& Name)
                {
#ifdef _WIN32
                        const bool ExportedSymbolsOnly = false;
#else
                        const bool ExportedSymbolsOnly = true;
#endif
                        for (auto H : make_range(ModuleKeys.rbegin(), ModuleKeys.rend()))
                                if (auto Sym = CompileLayer.findSymbolIn(H, Name, ExportedSymbolsOnly))
                                        return Sym;
                        if (auto SymAddr = RTDyldMemoryManager::getSymbolAddressInProcess(Name))
                                return JITSymbol(SymAddr, JITSymbolFlags::Exported);
#ifdef _WIN32
                        if (Name.length() > 2 && Name[0] == '_')
                                if (auto SymAddr =
RTDyldMemoryManager::getSymbolAddressInProcess(Name.substr(1)))
                                        return JITSymbol(SymAddr, JITSymbolFlags::Exported);
#endif
                        return nullptr;
                }
                ExecutionSession ES;
                std::shared_ptr<SymbolResolver> Resolver;
                std::unique_ptr<TargetMachine> TM;
                const DataLayout DL;
                ObjLayerT ObjectLayer;
                CompileLayerT CompileLayer;
                std::vector<VModuleKey> ModuleKeys;
        };
}; // namespace slljit

#endif // LLVM_EXECUTIONENGINE_ORC_SHADERJIT_H
```