МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА КОМПІЛЯТОРА С-ПОДІБНОЇ МОВИ ПРОГРАМУВАННЯ»**

Виконав: студент 4 курсу, групи 6.1219-1пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
А.С. Пірунов
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.ф.-м.н. Мильцев О.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ 07 ” 02 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Пірунову Артему Сергійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка компілятора С-подібної мови програмування

керівник роботи Мильцев Олександр Михайлович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 26 » січня 2023 року № 102-с

2. Строк подання студентом роботи 07.06.2023 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Перелік вимог до програмного забезпечення.
2. Моделювання програмного забезпечення.
3. Реалізація та тестування програмного забезпечення.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою докладу

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.02.2023 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	08.02.2023	
2.	Збір вихідних даних.	24.02.2023	
3.	Обробка методичних та теоретичних	17.03.2023	
	джерел.		
4.	Розробка першого та другого розділів.	14.04.2023	
5.	Розробка третього та четвертого розділів.	15.05.2023	
6.	Оформлення та нормоконтроль	01.06.2023	
	кваліфікаційної роботи.		
7.	Захист кваліфікаційної роботи.	21.06.2023	

Студент _____ А.С. Пірунов
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.М. Мильцев
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка компілятора С-подібної мови програмування»: 100 с., 21 рис., 3 табл., 7 джерел, 2 додатки.

АБСТРАКТНЕ СИНТАКСИЧНЕ ДЕРЕВО, ГЕНЕРАЦІЯ КОДУ, КОМПІЛЯТОР, МОВА ПРОГРАМУВАННЯ, СЕМАНТИЧНІ ПРАВИЛА, СИНТАКСИЧНІ ПРАВИЛА, СТАНДАРТНА БІБЛІОТЕКА.

Об’єкт дослідження – використання мов програмування.

Мета роботи: розібратись у роботі мов програмування розробивши компілятор С-подібної мови програмування.

Метод дослідження – емпіричний, аналітичний, узагальнення.

Програмування поширюється у різноманітних професіях. Використання мов програмування більше не є справою виключно програмістів. Саме тому в сучасному світі дуже важливо розуміти як влаштовані мови програмування. Щоб розширити знання про мови програмування у кваліфікаційній роботі було розроблено нову мову програмування, компілятор цієї мови та стандартну бібліотеку. Розглянуто те як визначення мови програмування впливає на структуру компіляторів. Розроблений компілятор протестовано у різних умовах, включаючи деякі виключні ситуації. За допомогою нової мови програмування було вирішено декілька класичних задач у світі програмування. Отриманий досвіт виявився корисним у розумінні того як влаштовані мови програмування.

SUMMARY

Bachelor’s Qualifying Paper «Development of a C-style Programming Language Compiler»: 100 pages, 21 figures, 3 tables, 7 references, 2 supplements.

ABSTRACT SYNTAX TREE, CODE GENERATION, COMPILER, PROGRAMMING LANGUAGE, SEMANTIC RULES, SYNTACTIC RULES, STANDARD LIBRARY.

The object of the study is the use of programing languages.

The aim of the study is to explore how programming languages works by making a compiler of a C-style programming language.

The methods of research are empirical, analytical, generalization.

Programming extends to various professions. The issues of using programming languages are not only for programmers anymore. That is why in nowadays it is important to understand how programming languages work. To expand knowledge about programming languages, a new programming language, a compiler of this language, and a standard library were developed in the qualification work. Explored how exactly a programing language definition effects on a compiler structure. The developed compiler was tested on various cases, including exceptional ones. I solved some classic problems from programming world using the new programming language. The gained experience turned out to be useful in understanding of how programming languages work.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

CLI	Command-line interface
EBNF	Extended Backus-Naur form
GCC	Колекція компіляторів

ВСТУП

У різноманітних промислових галузях розповсюджується використання інформаційних технологій. Для вирішення поточних задач час від часу виявляється більш доречним написання програми або скрипту. Тому глибоке розуміння основних принципів роботи мов програмування стає все більш актуальним.

В даній роботі розглянуто як проектуються та реалізуються мови програмування на прикладі розробки компілятора С-подібної мови програмування.

Метою роботи є отримання глибокого розуміння того як влаштовані мови програмування.

1 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Вимоги до компілятора

Компілятор має транслювати правильно написану програму у код для архітектури x86 у мові GCC (GNU Compiler Collection) асемблеру. GCC це набір компіляторів, що комбінує в собі різні front-end та back-end частини компіляторів. GCC має свій асемблер що можна використовувати окремо від всіх інших компонентів.

Процес трансляції тексту програми з однієї мови програмування у іншу мову програмування (як правило мову асемблеру) називається компіляцією.

Треба щоб у користувача була можливість задавати назву файлу який містить текст програми та також задавати назву файлу куди буде збережений результат компіляції.

Компілятор повинен видавати повідомлення про помилку у тексті програми якщо деяка помилка знайдена. Повідомлення про помилку повинно давати інформацію про те в якому файлі, в якому рядку, та в якій колонці виявлено помилку, а також описувати сутність помилки користувачеві [1].

Якщо під час компіляції не було виявлено жодних помилок то компілятор повинен закінчити роботу «тихо», тобто не виводячи жодних повідомлень.

Використання компілятора повинно відбуватись через CLI (інтерфейс командного рядка) введенням команди у форматі зазначеному у таблиці 1.1. Усі аргументи є обов’язковими. Наприклад отримана команда може мати наступний вигляд (рис. 1.1).

Таблиця 1.1 – Формат команди

Індекс аргументу	Аргумент
0	Назва команди або шлях до виконуваного файлу компілятора

Продовження табл. 1.1

Індекс аргументу	Аргумент
1	Повний або відносний шлях до файлу який містить текст програми
2	Повний або відносний шлях до файлу куди зберегти згенерований код у мові асемблеру

```
compiler input.txt output.s
```

Рисунок 1.1 – Приклад команди виклику компілятора

Компілятор має виконувати тільки етап компіляції, під час всіх інших етапів створення виконуваного файлу використовуються стороннє програмне забезпечення. Процес створення виконуваного файлу з допомогою компілятора описаний у діаграмі діяльності (див. рис. 1.2).

Якщо компілятор закінчує роботу без помилок це має означати що код у мові асемблеру, що є результатом компіляції, є коректним кодом. Такий код не повинен призводити до помилок під час створення OBJ файлу за допомогою GCC асемблеру (див. рис. 1.2).

1.2 Вимоги до мови програмування

Компілятор буде реалізовувати мову програмування, яка копіює синтаксис та семантики мови C.

Мова програмування повинна бути загального призначення, дозволяти писати програми у імперативній та процедурній парадигмах. Підтримувати простір імен для змінних та рекурсію (здатність мати кілька послідовних викликів однієї і тієї самої процедури).

Мати мінімальну кількість обов’язкового коду що додається до кожного виконуваного файлу для забезпечення функціонування під час виконання.

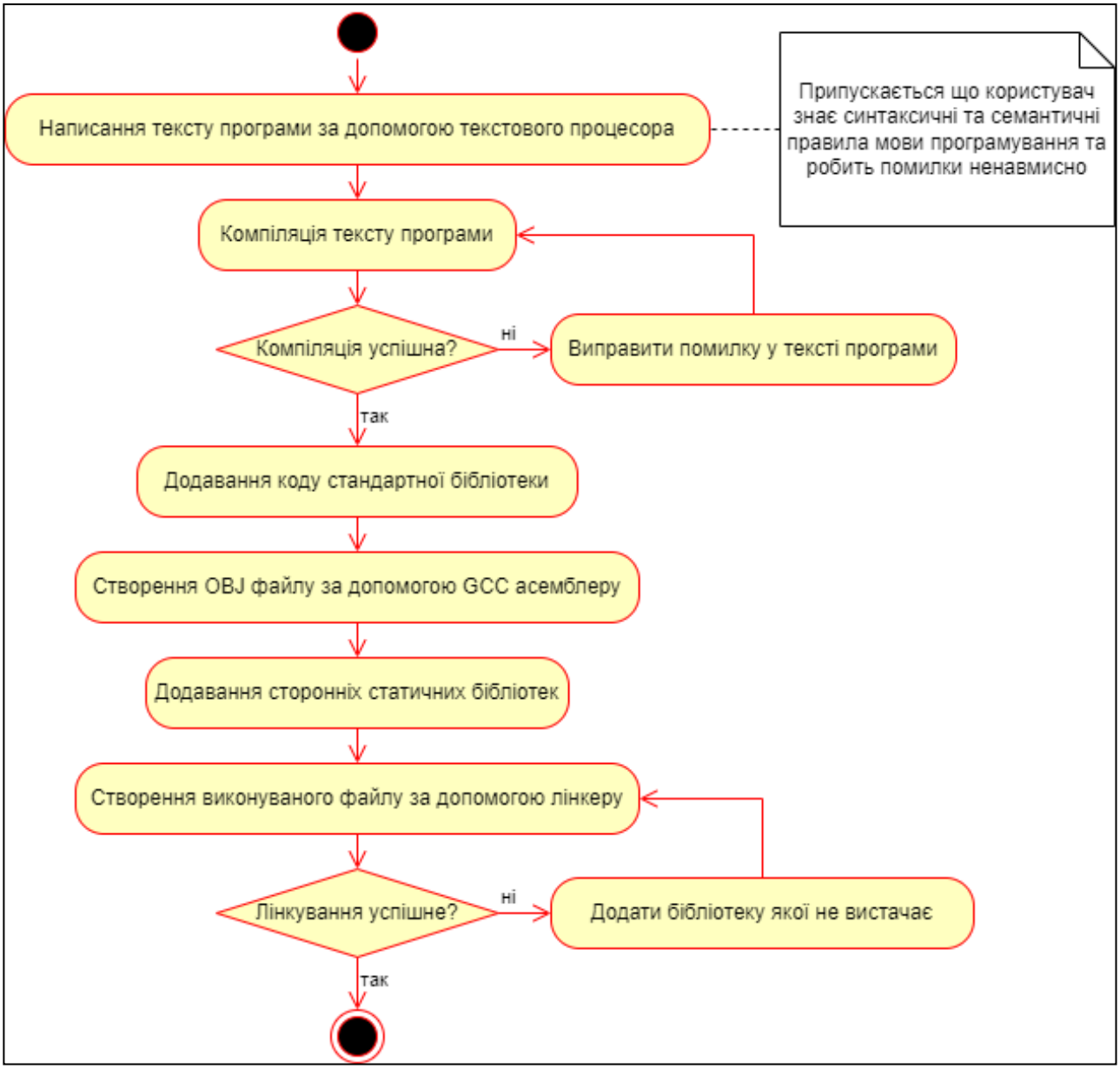

Рисунок 1.2 – Діаграма діяльності створення виконуваного файлу

1.2.1 Структура програми

Програма написана у мові програмування повинна складатись з набору функцій. Під функцією мається на увазі підпрограма, так як у термінології мови програмування C, бідь яку підпрограму (англ. subroutine) називають функцією (англ. function). Тому далі в тексті слово функція буде мати саме таке значення.

Мова програмування повинна надавати можливість визначати функції. Визначення функції повинно давати достатньо інформації для використання

функції у інших функціях та визначати поведінку функції.

У мові програмування також має бути механізм декларації функцій, тобто надання достатньої інформації для використання функції, але не маючи визначення цієї функції. Це надає можливість створювати бібліотеки та модулі, тобто код який визначений окремо від програми, але може бути використаний у самій програмі.

У мові програмування має існувати можливість ділити текст програми на кілька файлів.

При декларації або визначенні функції має бути можливість вказати які значення їй потрібні для роботи. Так щоб код який використовує функцію був зобов’язаний надати необхідні значення для функції.

У мові програмування повинен бути механізм передачі деякого значення від функції тому коду який функцію використав, при завершенні роботи функції.

Мова програмування має надавати можливість створення змінних для зберігання значень під час розрахунків. З можливістю використовувати збережені значення у наступних розрахунків, в рамках однієї роботи функції.

У мові програмування повинна бути можливість починати виконання програми з деякої функції.

1.2.2 Доступні операції

Визначення функції має описувати які операції функція виконує та визначати порядок цих операцій.

Із арифметичних операцій повинні бути доступні операції додавання та віднімання цілих чисел. Цих операцій достатньо для отримання більш складних операцій, таких як множення та ділення цілих чисел. Бо множення та ділення можна представити як послідовність операцій додавання та віднімання.

Має бути можливість для вводу та виводу даних з програми. Щоб програма в цілому могла виконувати якусь роботу, опрацьовувати дані на вході та давати

результати на виході. Це надає сенс існуванню програми, а тому і сенс розробляти програму.

Потрібно щоб у мові програмування був механізм виконання деяких операцій тільки якщо деяка умова виконується. А також потрібна можливість виконувати одні і ті самі операції кілька разів в рамках однієї роботи функції.

Виконання деяких операцій у циклі, тобто продовжувати виконувати ряд операцій поки деяка умова не перестане виконуватись.

Найголовніша операція, яка обов’язково повинна бути у мові програмування це можливість використовувати одні функції в інших функціях. Це дозволяє ділити задачу на під-задачі та користуватись вже готовими рішеннями у будь якій функції.

2 МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Опис синтаксису мови у EBNF

Для визначення синтаксису мови програмування було використано нотацію EBNF (Extended Backus–Naur Form) [2]. EBNF це нотація для опису формальних мов яку можна використовувати для опису синтаксису мов програмування. Опис у EBNF деякої формальної мови це список правил які загалом описують усі можливі у такій мові тексти. Маючи повний опис формальної мови у EBNF можна перевірити будь який текст на те чи відповідає він усім правилам, якщо так тоді такий текст є текстом який написаний цією формальною мовою.

Описаний синтаксис мови програмування (див. додаток А) обумовлений вимогами до мови програмування (див. 1.1). Визначення синтаксису поділене на три рівні які помічені коментарями від 1 до 3. Ділення на рівні зумовлено тим що текст програми описується синтаксичними елементами, які в свою чергу описуються лексичними елементами, котрі збираються із окремих символів.

Перший рівень визначає структуру абстрактного синтаксичного дерева де кореневим елементом є файл. Тобто описує синтаксично правильний текст файлу за допомогою синтаксичних елементів. Перевірка того чи можливо побудувати абстрактне синтаксичне дерево з деякого тексту називають синтаксичним аналізом.

Другий рівень визначає те з яких лексичних елементів (токенів) складаються синтаксичні елементи. Тобто токени це листя абстрактного синтаксичного дерева. Таке листя проводить лінію між абстрактними синтаксичними елементами та конкретними лексичними елементами у тексті програми.

Третій рівень визначає з яких символів які токени складаються. Таким чином текст як послідовність символів ділиться на деякі елементи кожен з яких

несе у собі якесь значення. Ділення тексту на токени називають лексичним аналізом.

2.2 Визначення семантики мови

У цьому розділі назви які написані *курсивом* посилаються на відповідні елементи визначенні у EBNF.

Успішно побудувавши абстрактне синтаксичне дерево ми отримуємо мовленнєві конструкції які написані за всіма правилами. Семантика мови програмування в свою чергу придає мовленнєвим конструкціям значення, дозволяє зрозуміти про що написано в тексті.

Будь яка інформаційна система робота якої дозволяє визначити сенс коду написаного визначеною мовою програмування називають реалізацією цієї мови програмування. Процес визначення сенсу називають трансляцією.

2.2.1 Семантика file

Реалізація повинна запам’ятовувати кожен *file identifier* який зустрінить під час трансляції.

Будь який *include directive* треба розуміти так ніби замість нього підставлений код із файлу який визначений за допомогою *file name* у спосіб який залежить від реалізації. Але якщо визначений файл має такий *file identifier* який вже не перший раз зустрічається реалізацією тоді вміст цього файлу повністю ігнорується. Під час трансляції усі файли зазначені в усіх *include directive* повинні існувати.

Після того як значення усіх *include directive* в усіх файлах було визначено реалізація транслює усі *function definition*. Кожен *function definition* описує окрему функцію у якої повинен бути такий *function signature* який не

зустрічається у інших *function definition*. Те які дії виконує функція коли її викликають визначається у *function body* який є або *return statement*, або *statement list*.

Два або більше *function declaration* не можуть мати одне і те саме *function name* і при цьому мати різну кількість *parameter name*. Реалізація повинна дозволяти викликати функцію навіть якщо така функція має тільки *function declaration* без *function definition*.

Якщо у тексті програми є функція у якої *function name* це «gw_name», а *parameter list* немає жодного *parameter name*, то ця функція повинна бути викликана реалізацією після процесу трансляції.

2.2.2 Семантика statement

Треба інтерпретувати *statement* відповідно до його типу.

Будь який *return statement* закінчує роботу функції у якій знаходиться, де результат роботи функції визначається у відповідному *expression*. Усі *function definition* повинні гарантувати що під час їх виконання один з *return statement* буде виконано.

Усі *statement* в середині *statement list* повинні бути виконані один за одним починаючи з того який іде раніше в тексті. Якщо під час цього буде виконано *return statement* безпосередньо або в середині інших *statement* то усі *statement* які ідуть після цього не будуть виконані.

Виконання *expression statement* означає виконання відповідного *expression*. Результат цього *expression* не зберігається.

Виконання *while statement* означає виконання відповідного *sub expression* і якщо результат цього *expression* не дорівнює нулю тоді виконується відповідний *statement*. Після того як *statement* буде виконано, і якщо під час виконання не було жодного *return statement* то виконання *while statement* починається з початку. Послідовне виконання *while statement* закінчується тоді коли виконання

expression в результаті дає нуль. Якщо нуль було отримано при першому виконанні тоді відповідний *statement* взагалі буде проігноровано.

2.2.3 Семантика expression

Виконання деякого *expression* відбувається відповідно того *assignation expression* це чи *sum expression*.

Виконання *assignation expression* це виконання відповідного *expression* результат якого буде збережено під ім’ям *parameter name* або *variable name*. Причому відповідний *name* повинен хоча б раз зустрічатись у *variable list* або *parameter list* з функції у якій *assignation expression* знаходиться. Тобто *name* повинен існувати у просторі імен функції. Результат *assignation expression* це результат відповідного *expression*.

Виконання *sum expression* це виконання першого *postfix expression* та збереження результату у тимчасову змінну. Всі наступні пари *binary operator* з *postfix expression* виконуються послідовно, так що спочатку виконується *postfix expression*, а результат додається або віднімається від тимчасової змінної, якщо *binary operator* був *addition operator* або *subtraction operator* відповідно. Результат *sum expression* є значенням тимчасової змінної після всіх операцій додавання та віднімання.

Для будь якого *function call* реалізація повинна гарантувати що зазначений *function name* зустрічається у якомусь із *function signature* у тексті раніше цього *function call*. Виконання *function call* призводить до послідовного виконання усіх *expression* у *argument list*. Тільки потім відбувається виклик функції за ім’ям *function name*. Коли виконання цієї функції починається значення збережені під іменами усіх *parameter name* з *parameter list* повинні бути результатами відповідних *expression* з *argument list*. Результат *function call* це результат роботи викликаної функції.

Реалізація повинна гарантувати що кожен виклик функції матиме свої

змінні для відповідних імен *name* із *parameter list* та *variable list*. Тобто асоціювати ці змінні не з самою функцією, а з кожною подією виклику цієї функції.

Виконання *mutable* у *prefix expression* це отримання останнього значення що було записано під відповідним ім’ям. Реалізація повинна гарантувати що якесь значення буде збережено у змінній до першої спроби отримати значення цієї змінної.

Результатом *number* є ціле позитивне число що безпосередньо записане у токені *unsigned integer* у десятковій системі числення.

Результатом *string* є деяке число унікальне для кожного *string* у абстрактному синтаксичному дереві. Реалізація повинна надати можливість використання такого числа для отримання значень окремих символів відповідного *string* (не враховуючи лапки на початку та в кінці). Так щоб при спробі отримати значення символу який іде після останнього буде отримано нуль.

2.2.4 Пріоритети операцій

Загалом у визначеній мові програмування можна виділити 4 оператори які можна поділити на 3 групи пріоритету та визначити їх асоціативність (див. табл. 2.1). Чим менше номер групи у якій знаходиться оператор тим раніше така операція буде виконана в рамках одного *expression*. Якщо у виразі є декілька операторів з однієї групи пріоритету то такі операції будуть виконані по черзі зліва направо або справа наліво відповідно до асоціативності.

Таблиця 2.1 – Пріоритет операторів

Пріоритет	Оператор	Опис	Асоціативність
1	`a(...)`	Виклик функції з передачею відповідних аргументів	Зліва направо

Продовження табл. 2.1

Пріоритет	Оператор	Опис	Асоціативність
2	`a+b` `a-b`	Додавання та віднімання значеннь	Зліва направо
3	`a=b`	Зміна значення змінної чи параметра функції	Справа наліво

2.3 Опис структури компілятора

Знаючи синтаксичні та семантичні правила деякої мови програмування можна визначити основні етапи реалізації цієї мови. Для переліку основних етапів реалізації мови програмування визначеної у розділах 2.1 та 2.2 засновуючись на [3] була створена діаграма діяльності (рис. 2.1).

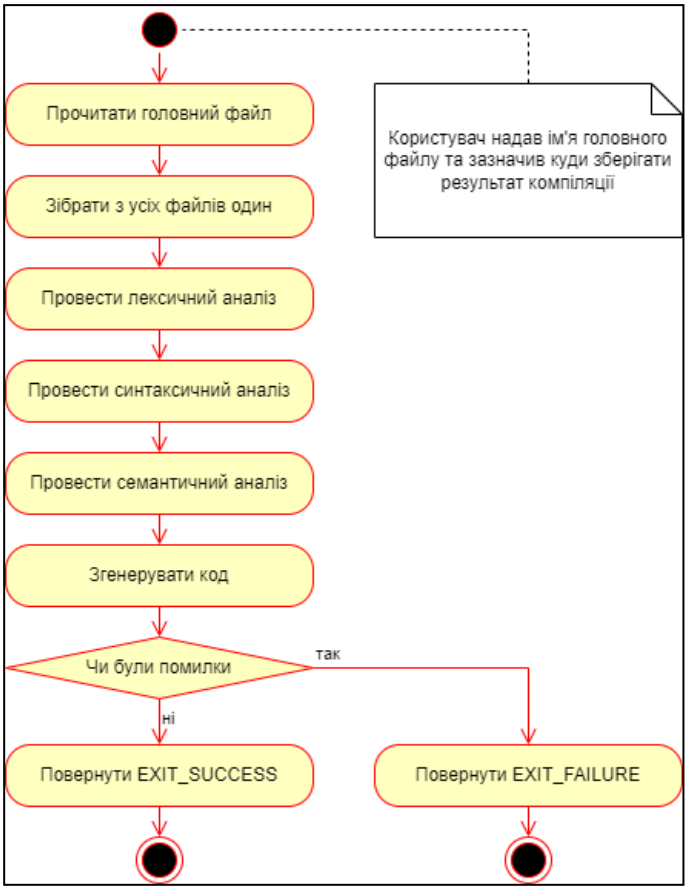

Рисунок 2.1 – Діаграма діяльності процесу компіляції

В даному випадку реалізація є компілятором, тобто один з етапів транслює код програми у мову асемблеру. При цьому отримані інструкції відповідають поведінці програми яка визначена семантикою мови програмування, тобто інтерпретуються комп’ютером так ніби інтерпретується безпосередньо текст програми.

Перелічені етапи роботи компілятора (рис. 2.1) майже повністю відповідають тим етапам які перелічені у [4, с. 5]. Але відсутні такі етапи як генерація допоміжного коду та загальна оптимізація коду. Через це код у мові асемблеру згенерований компілятором буде не ефективним.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Стандартна бібліотека

Усі вимоги до мови програмування що не забезпечені семантичними правилами (розділ 2.2) реалізовані стандартною бібліотекою. Стандартна бібліотека це код (рис. 3.1) який буде додаватись до кожної програми щоб отримати виконуваний файл.

```
# include <stdio.h>
# include <stdint.h>

typedef intptr_t NUM;

NUM gw_main(void);

NUM gw_ldchar(NUM x)
{ return *(char *)x; }
NUM gw_stchar(NUM x, NUM y)
{ return *(char *)x = (char)y; }
NUM gw_putchar(NUM x)
{ return putchar(x); }
NUM gw_getchar(void)
{ return getchar(); }
int main(void)
{ return gw_main(); }
```

Рисунок 3.1 – Код стандартної бібліотеки

Стандартна бібліотека містить функції для вводу та виводу даних що відповідає вимогам до мови програмування (див. 1.1). Також у бібліотеці є функція що надає можливість за допомогою числа асоційованого зі string отримати значення кожного з символів у цій string, включно з нульовим символом який іде після останнього символу. Усі функції перелічені в таблиці 3.1.

Таблиця 3.1 – Функції стандартної бібліотеки

Ім’я функції	Аргументи	Опис
`gw_putchar`	`ch`	Відправити символ у стандартний вивід, де `ch` це значення символу.
`gw_getchar`		Читає один символ зі стандартного вводу та повертає значення символу.
`gw_ldchar`	`ptr`	Повертає значення символу зі string, де `ptr` це число асоційоване зі string до якого додали індекс символу, значення якого треба дізнатись.
`gw_stchar`	`ptr, ch`	Змінює значення символу у string, де `ch` це нове значення, а `ptr` це число асоційоване зі string до якого додали індекс символу, значення якого треба змінити.

Щоб скористуватись функціями стандартної бібліотеки при написанні програми треба повідомити компілятору назви та списки параметрів усіх цих функцій це можна зробити наприклад вставивши за допомогою include directive файл з відповідним змістом (рис. 3.2).

```
# file gwlib

: Write a character ch to standard output
gw_putchar(ch);

: Read the next character from standard input
gw_getchar();

: Get the value of the character in memory
: pointed to by ptr
gw_ldchar(ptr);

: Set the value of the character in memory
: pointed to by ptr
gw_stchar(ptr, ch);
```

Рисунок 3.2 – Код заголовку стандартної бібліотеки

3.2 Лексичний аналізатор

Лексичний аналізатор це модуль який ділить текст файлу на токени, якщо поділити текст вдалося то такий текст вважається лексично правильним [5].

У даній реалізації лексичний аналізатор є потоком токенів, де щоб отримати наступний токен треба викликати функцію chop_token. Це дозволяє опрацьовувати файл поступово, на випадок якщо буде виявлено що цей файл насправді треба проігнорувати.

Лексичний аналізатор ділить текст по правилам третього рівня які визначенні у EBNF (додаток А). Але передає далі тільки ті токени які будуть використані у наступних модулях. Наприклад жоден delimiter не потрапить у наступні модулі тому що не несе ніякої семантичної інформації і не буде використовуватись у наступних модулях.

Якщо на етапі синтаксичного аналізу було виявлено уривок тексту що не підпадає під правила створення токенів то відповідну помилку буде виведено в терміналі, місце помилки в тексті визначається початком проблемної частини цього тексту. При цьому функція chop_token поверне токен-індикатор що повідомить наступному модулю що той отримав некоректні дані.

3.3 Препроцесор

Препроцесор у даній реалізації це модуль який об’єднує усі файли в один, тобто замінює усі include directive змістом відповідних файлів, там де це потрібно. Для цього препроцесор проводить лексичний аналіз, частково синтаксичний та частково семантичний аналіз, для того щоб знайти та опрацювати усі include directive.

Якщо під час обробки однієї з include directive виявиться що відповідний файл не існує чи не може бути прочитаний то у термінал буде виведено відповідне повідомлення про помилку з зазначеним місцезнаходженням проблемної include directive.

Результатом роботи препроцесору є список токенів зібраний з токенів з усіх файлів, крім тих які були проігноровані. Але якщо під час опрацювання виникла помилка з include directive або у лексичному аналізаторі тоді результатом роботи препроцесору буде некоректний список токенів. І щоб повідомити наступному модулю статус списку токенів у цей список було додано індикатор коректності.

3.4 Синтаксичний аналізатор

Синтаксичний аналізатор це модуль який бере список токенів та намагається побудувати абстрактне синтаксичне дерево відповідно до синтаксису мови програмування. Якщо дерево побудоване успішно то текст програми називають синтаксично правильним.

Якщо під час синтаксичного аналізу виявлено токен або токени які не відповідають синтаксису мови програмування то у термінал буде виведено відповідне повідомлення про помилку, включаючи місцезнаходження першого з проблемних токенів.

Результатом роботи синтаксичного аналізатору є абстрактне синтаксичне дерево яке відповідає правилам другого та першого рівня які визначені у EBNF (додаток А). При отримані некоректного списку токенів чи виникнення помилки під час створення абстрактного синтаксичного дерева результатом роботи синтаксичного аналізатору буде некоректне абстрактне синтаксичне дерево, тобто з відповідним індикатором.

3.5 Семантичний аналізатор

Семантичний аналізатор це модуль що перевіряє абстрактне синтаксичне дерево на наявність семантичних помилок [6]. У даній реалізації ця перевірка

ділиться на чотири етапи. Ці етапи були перелічені у діаграмі діяльності (рис. 3.3).

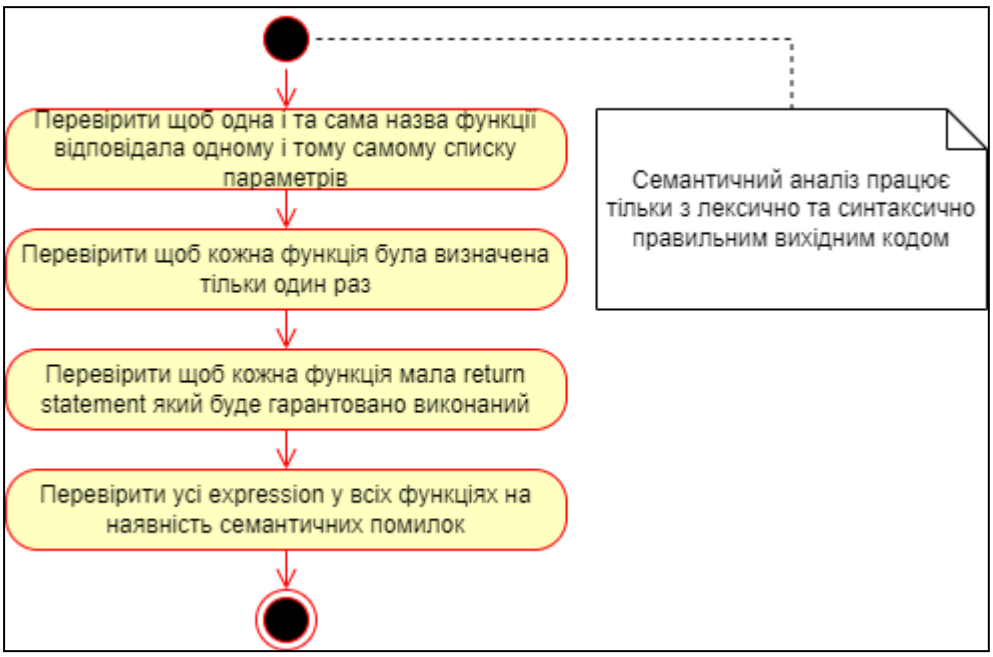

Рисунок 3.3 – Діаграма діяльності процесу семантичного аналізу

Перевірка абстрактного синтаксичного дерева є обходом усіх віток цього дерева від кореневого елементу до листя, з пошуком елементів які не відповідають семантичним правилам.

У випадку якщо якась семантична помилка була знайдена вона буде виведена у терміналі, зазначаючи токен відповідно до проблемної мовленнєвої конструкції. При цьому абстрактне синтаксичне дерево буде відмічене як некоректне.

3.6 Генератор коду

Генератор коду це модуль що бере абстракте синтаксичне дерево та генерує код у мові асемблеру [7]. Згенерований код передає суть написаної програми у вигляді інструкцій та даних, таким чином щоб асемблер зміг транслювати цей код у відповідні машинні інструкції, а потім лінкер створив на

основі цих інструкцій виконуваний файл.

У даній реалізації генерація коду ділиться на два етапи. Спочатку іде пошук усіх string та розміщення їх у секції даних. А потім вже генеруються код для кожного визначення функції у секції тексту.

Пошук string у абстрактному синтаксичному дереві це обхід усіх віток цього дерева від кореневого елементу до усіх prefix expression. Під час обходу дерева іде підрахунок кількості string, де кожному такому елементу видається унікальний номер за яким той і буде збережений у секції даних.

Генерація коду у мові асемблеру є обходом абстрактного синтаксичного дерева, від кореневого елементу до листя.

Під час генерації коду у мові асемблеру помилки можуть виникнути тільки у тому випадку якщо якийсь з попередніх модулів працює некоректно. В такому випадку неможливо вивести інформативне повідомлення про помилку бо суть самої помилки не відома.

4 ПРИКЛАДИ РОБОТИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Hello, World!

Програма «Hello, World!» є класичною перевіркою того що компілятор працює правильно. Зазвичай це дуже коротка програма яка виводить текстове повідомлення за допомогою функції для виведення тексту. Але у розробленій стандартній бібліотеці немає функції для виводу тексту тому довелося написати таку (рис. 4.1). На скріншоті командного рядка (рис. 4.2) можна побачити використання компілятору для отримання коду у мові асемблеру, створення виконуваного файлу, та роботу отриманої програми.

```
# file helloworld
# include "gwlib.gw"

print_line(str) {
  while (gw_ldchar(str)) {
    gw_putchar(gw_ldchar(str));
    str = str + 1;
  }
  gw_putchar(10);
  return 0;
}

gw_main() {
  print_line("Hello, World!!!");
  return 0;
}
```

Рисунок 4.1 – Код програми

```
./gw ./examples/helloworld.gw ./out.s
gcc -m32 -o out out.s gwlib.s
./out
Hello, World!!!
```

Рисунок 4.2 – Виконання програми у терміналі

4.2 FizzBuzz

Програма «FizzBuzz» є класичною задачею у програмуванні. Треба вивести числа від одного до якогось числа N. Але якщо число ділиться на 3 то замість самого числа вивести текст «Fizz», якщо число ділиться на 5 – вивести «Buzz». Якщо число ділиться і на 3 і на 5 – тоді «FizzBuzz». Рішення можна побачити на рисунках (4.3) – (4.5). На скріншоті командного рядка (див. рис. 4.6) можна побачити використання компілятору для отримання коду у мові асемблеру, створення виконуваного файлу, та роботу отриманої програми.

```
# file fizzbuzz

# include "gwlib.gw"

print_line(str) {
  while (gw_ldchar(str)) {
    gw_putchar(gw_ldchar(str));
    str = str + 1;
  }
  return 0;
}

equal(a, b) {
  while (a - b) {
    return 0;
  }
  return 1;
}

not_equal(a, b) {
  while (a - b) {
    return 1;
  }
  return 0;
}

divi(a, b) i, n, m {
  i = 0;
n = 0;
  m = 0;
while (not_equal(i, a)) {
    i = i + 1;
    n = n + 1;
    while (equal(n, b)) {
m = m + 1;
      n = 0;
```

Рисунок 4.3 – Код програми

```
    }
  }
  return m;
}

modi(a, b) i, n, m, j {
  i = 0;
  n = 0;
  m = 0;
  j = 0;
  while (not_equal(i, a)) {
    i = i + 1;
    n = n + 1;
    while (equal(n, b)) {
      m = m + 1;
      n = 0;
      j = i;
    }
  }
  return i - j;
}

print_number(num) p, t {
  p = 1000;
  while (p - 1) {
    t = modi(num, p);
    p = divi(p, 10);
    gw_putchar(divi(t, p) + 48);
  }
  gw_putchar(10);
  return 0;
}

fizz(num) {
  while (modi(num, 3)) {
    return 0;
  }
  print_line("Fizz");
  return 1;
}

buzz(num) {
  while (modi(num, 5)) {
    return 0;
  }
  print_line("Buzz");
  return 1;
}

fizzbuzz(num) a {
  a = fizz(num);
  a = a + buzz(num);
  while (a) {
    gw_putchar(10);
```

Рисунок 4.4 – Код програми

```
    return 0;
  }
  print_number(num);
  return 0;
}

gw_main() i, n {
  i = 1;
  n = 25;
  while (not_equal(i, n)) {
    fizzbuzz(i);
    i = i + 1;
  }
  return 0;
}
```

Рисунок 4.5 – Код програми

```
./gw ./examples/fizzbuzz.gw ./out.s
gcc -m32 -o out out.s gwlib.s
./out
001
002
Fizz
004
Buzz
Fizz
007
008
Fizz
Buzz
011
Fizz
013
014
FizzBuzz
016
017
Fizz
019
Buzz
Fizz
022
023
Fizz
```

Рисунок 4.6 – Виконання програми у терміналі

4.3 Послідовність Фібоначчі

Програма що виводить послідовність Фібоначчі є класичною задачею у програмуванні. Треба вивести перші N чисел з послідовності Фібоначчі. Де перші два числа це одиниці, а кожне наступне це сума двох які ідуть до цього. Рішення можна побачити на рисунках 4.7 та 4.8. На скріншоті командного рядка (див. рис. 4.9) можна побачити використання компілятору для отримання коду у мові асемблеру, створення виконуваного файлу, та роботу отриманої програми.

```
# file fibonacci

# include "gwlib.gw"

equal(a, b) {
  while (a - b) {
    return 0;
  }
  return 1;
}

divi(a, b) i, n, m {
  i = n = m = 0;
  while (i - a) {
    i = i + 1;
    n = n + 1;
    while (equal(n, b)) {
      m = m + 1;
      n = 0;
    }
  }
  return m;
}

modi(a, b) i, n, m, j {
  i = n = m = j = 0;
  while (i - a) {
    i = i + 1;
    n = n + 1;
    while (equal(n, b)) {
      m = m + 1;
      n = 0;
      j = i;
    }
  }
  return i - j;
}
```

Рисунок 4.7 – Код програми

```
print_number(num) p, t {
  p = 1000000;
  while (p - 1) {
    t = modi(num, p);
    p = divi(p, 10);
    gw_putchar(divi(t, p) + 48);
  }
  gw_putchar(10);
  return 0;
}

gw_main() a, b, c, n {
  a = 0;
  b = 1;
  c = a + b;
  n = 25;
  while (n) {
    print_number(c);
    a = b;
    b = c;
    c = a + b;
    n = n - 1;
  }
  return 0;
}
```

Рисунок 4.8 – Код програми

```
./gw ./examples/fibonacci.gw ./out.s
gcc -m32 -o out out.s gwlib.s
./out
000001
000002
000003
000005
000008
000013
000021
000034
000055
000089
000144
000233
000377
000610
000987
001597
002584
004181
006765
010946
017711
028657
046368
075025
121393
```

Рисунок 4.9 – Виконання програми у терміналі

4.4 Вивід помилок

Також були протестовані різні виключні ситуації для того щоб з’ясувати як компілятор реагує на помилки у тексті програми, та те наскільки правильно повідомлення про помилку вказують на проблеми.

4.4.1 Відсутній return

Як зазначено семантикою мови програмування (див. 2.2.2) кожна функція повинна повертати якесь значення. Тому текст програми (рис. 4.10) містить семантичну помилку так як у функції foo відсутній return statement. При спробі скомпілювати цей код компілятор видав повідомлення про помилку (рис. 4.11) де зазначив що у функції foo відсутній return statement. Також повідомлення про помилку містить назву файлу та місце де знаходиться проблемна функція.

```
# file error

foo() {

}

gw_main() {
  foo();
  return 0;
}
```

Рисунок 4.10 – Код програми

```
./gw ./main.gw ./out.s
./main.gw:3:1: error: return statement not found in the function
   3 | foo() {
     | ^~~
compilation terminated
```

Рисунок 4.11 – Помилка що виведена у терміналі

4.4.2 Неіснуюча функція

Як зазначено семантикою мови програмування (див. 2.2.3) виклик функції може відбуватись тільки для функцій які були визначені або задекларовані раніше виклику функції. Тому текст програми (рис. 4.12) містить семантичну помилку так як у функції gw_main іде виклик функції buz, але при цьому у тексті програми buz не визначений як функція. При спробі скомпілювати цей код компілятор видав повідомлення про помилку (рис. 4.13) де зазначив що buz не є функцією. Також повідомлення про помилку містить назву файлу та місце де знаходиться проблемний виклик функції.

```
# file error

foo() return 1;

gw_main() tmp {
  tmp = foo() + buz();
  return 0;
}
```

Рисунок 4.12 – Код програми

```
./gw ./main.gw ./out.s
./main.gw:6:17: error: expression is not a function
   6 |   tmp = foo() + buz();
     |                 ^~~
compilation terminated
```

Рисунок 4.13 – Помилка що виведена у терміналі

4.4.3 Невизначена поведінка

Текст програми (див. рис. 4.14) не містить жодних лексичних, синтаксичних, та семантичних помилок. Але ця програма містить поведінку яка невизначена семантикою мови програмування. А саме намагання отримати значення символів зі string які ідуть після нульового символу.

Реалізація мови програмування не зобов’язана шукати та повідомляти про

невизначену поведінку у тексті програми. Тому дана програма успішно компілюється та працює (рис. 4.15). Реалізація не гарантує що програма з невизначеною поведінкою буде давати однакові результати при різних умовах, чи взагалі працювати.

```
# file error

# include "gwlib.gw"

print_10(str) i {
  i = 0;
  while (i - 10) {
    gw_putchar(gw_ldchar(str + i));
    i = i + 1;
  }
  return 0;
}

gw_main() tmp {
  print_10("123");
  tmp = "some text";
  return 0;
}
```

Рисунок 4.14 – Код програми

```
./gw ./main.gw ./out.s
gcc -m32 -o out out.s gwlib.s
./out
123 some t
```

Рисунок 4.15 – Виконання програми у терміналі

ВИСНОВКИ

У даній кваліфікаційній роботі бакалавра було спроектовано та розроблено компілятор С-подібної мови програмування загального призначення.

У першому розділі перелічені вимоги до компілятора та мови програмування. Описано інтерфейс компілятора. Схематично зображено як відбувається взаємодія користувача та компілятора. Перелічені можливості мови програмування.

У другому розділі визначено мову програмування та спроектовано компілятор цієї мови. Перелічені лексичні, синтаксичні, та семантичні правила мови програмування. Описано загальну структуру компілятора. Спроектовано архітектуру компілятора, де схематично зображені всі модулі з яких складається компілятор та взаємодії між ними.

У третьому розділі програмно реалізовано спроектований компілятор. Розроблено кожний модуль компілятора. Описані технічні деталі кожного модуля та взаємодію з іншими модулями. Реалізовано стандартну бібліотеку мови програмування.

У четвертому розділі успішно протестована робота компілятора. Розглянуто роботу компілятора в умовах коли текст програми не містить помилок. Розглянуті різні помилки які можуть бути в тексті програми та як компілятор реагує на ці помилки.

ПЕРЕЛІК ПОСИЛАНЬ

1. Formatting Error Messages. URL: https://www.gnu.org/prep/standards/html_node/Errors.html (дата звернення: 19.03.2023).
2. Extended Backus–Naur form. URL: https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form (дата звернення: 21.03.2023).
3. Загальна схема роботи компіляторів. URL: https://studfile.net/preview/5465784/ (дата звернення: 22.03.2023).
4. Compilers, Principles, Technics, & Tools / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, 2nd edition. Addison Wesley, 2006. 1040 p.
5. Compiler design – lexical analysis. URL: https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm (дата звернення: 23.03.2023).
6. Semantic analysis in Compiler Design. URL: https://iq.opengenus.org/semantic-analysis-in-compiler-design/ (дата звернення: 24.03.2023).
7. Compiler Design – Code Generation. URL: https://www.tutorialspoint.com/compiler_design/compiler_design_code_generation.htm (дата звернення: 26.03.2023).

ДОДАТОК А

EBNF

```
(************************************************************)
(* 1 *)
(************************************************************)
file = file directive
     , { file entry } ;
file directive = file directive prefix
               , file identifier ;
file entry = include directive
           | function declaration
           | function definition ;
include directive = include directive prefix
                  , file name ;
function declaration = function signature
                     , function declaration end ;
function definition = function signature
                    , [ variable list ]
                    , function body ;
function signature = function name
                   , parameter list ;
parameter list = open parameter list
               , [ some parameters ]
               , close parameter list ;
some parameters = parameter name
                , { parameter separator , parameter name } ;
variable list = variable name
              , { variable separator , variable name } ;
function body = statement list
              | return statement ;
statement list = open statement list
               , { statement }
               , close statement list ;
```

```
return statement = return statement prefix
                 , expression
                 , statement separator ;
statement = statement list
          | return statement
          | expression statement
          | while statement ;
expression statement = expression
                     , statement separator ;
while statement = while statement prefix
                , sub expression
                , statement ;
expression = assignation expression
           | sum expression ;
assignation expression = mutable
                       , assignation operator
                       , expression ;
mutable = variable name
        | parameter name ;
sum expression = postfix expression
               , { binary operator , postfix expression } ;
binary operator = subtraction operator
                | addition operator ;
postfix expression = function call
                   | prefix expression ;
function call = function name
              , argument list ;
argument list = open argument list
              , [ some arguments ]
              , close argument list ;
some arguments = expression
               , { argument separator , expression } ;
```

```
prefix expression = mutable
                  | number
                  | string
                  | sub expression ;
sub expression = open sub expression
               , expression
               , close sub expression ;
(*********************************************************)
(* 2 *)
(*********************************************************)
delimiter = comment
          | space ;
file directive prefix = [ delimiter ]
                      , sharp
                      , [ delimiter ]
                      , file word
                      , [ delimiter ] ;
file identifier = identifier
                , [ delimiter ] ;
include directive prefix = sharp
                         , [ delimiter ]
                         , include word
                         , [ delimiter ] ;
function declaration end = semicolon
                         , [ delimiter ] ;
file name = string literal
          , [ delimiter ] ;
function name = identifier
              , [ delimiter ] ;
open parameter list = open parenthesis
                    , [ delimiter ] ;
close parameter list = close parenthesis
                     , [ delimiter ] ;
parameter name = identifier
               , [ delimiter ] ;
```

```
parameter separator = comma
                    , [ delimiter ] ;
variable name = identifier
              , [ delimiter ] ;
variable separator = comma
                   , [ delimiter ] ;
statement separator = semicolon
                    , [ delimiter ] ;
open statement list = open brace
                    , [ delimiter ] ;
close statement list = close brace
                     , [ delimiter ] ;
return statement prefix = return keyword
                        , [ delimiter ] ;
while statement prefix = while keyword
                       , [ delimiter ] ;
assignation operator = equal
                     , [ delimiter ] ;
subtraction operator = minus
                     , [ delimiter ] ;
addition operator = plus
                  , [ delimiter ] ;
open argument list = open parenthesis
                   , [ delimiter ] ;
close argument list = close parenthesis
                    , [ delimiter ] ;
argument separator = comma
                   , [ delimiter ] ;
number = unsigned integer
       , [ delimiter ] ;
string = string literal
       , [ delimiter ] ;
open sub expression = open parenthesis
                    , [ delimiter ] ;
close sub expression = close parenthesis
                     , [ delimiter ] ;
```

```
(************************************************************)
(* 3 *)
(************************************************************)
sharp = '#' ;
file word = "file" ;
identifier = ( letter | '_' )
           , { letter | digit | '_' } ;
include word = "include" ;
semicolon = ';' ;
string literal = '"'
               , { any character }
               , '"' ;
open parenthesis = '(' ;
close parenthesis = ')' ;
open brace = '{' ;
close brace = '}' ;
comma = ',' ;
return keyword = "return" ;
while keyword = "while" ;
comment = ':'
        , any character
        , new line ;
new line = ? new line character ? ;
equal = '=' ;
minus = '-' ;
plus = '+' ;
unsigned integer = digit
                 , { digit } ;
digit = ? decimal digits ? ;
space = ? white space characters ? ;
any character = ? all non-control characters ? ;
letter = ? lowercase or uppercase letters ? ;
```

ДОДАТОК Б

Вихідний код

```
//#define PRINT_PARSE_FUNC printf(" %s\n", __func__);
#define PRINT_PARSE_FUNC

//#define PRINT_ANALYZE_FUNC printf(" %s\n", __func__);
#define PRINT_ANALYZE_FUNC

// headers

#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "stdbool.h"
#include "ctype.h"

// macro

#define err(...) fprintf(stderr, "error: " __VA_ARGS__)

#define note(...) fprintf(stderr, "note: " __VA_ARGS__)

#define REG_SIZE (4)

// types

typedef struct {
  size_t size;
  const char *at;
} View;

typedef struct {
  bool invalid;
```

```
  char *buffer;
  char *name;
  size_t size;
  View text;
  View id;
} File;

typedef enum {
  T_INVALID,
  T_EOF,
  T_SHARP,
  T_IDENT,
  T_NUMBER,
  T_STRING,
  T_PAREN_BEG,
  T_PAREN_END,
  T_BRACE_BEG,
  T_BRACE_END,
  T_WHILE,
  T_RETURN,
  T_SEMICOLON,
  T_EQUAL,
  T_PLUS,
  T_MINUS,
  T_COMMA,
} TokenType;

typedef struct {
  TokenType type;
  View text;
  size_t file_index;
} Token;

typedef struct {
  Token token;
  View text;
```

```
} TokenAndText;

typedef struct {
  bool invalid;
  size_t files_num;
  size_t files_cap;
  File *files;
  size_t tokens_cap;
  size_t tokens_num;
  Token *tokens;
} Unit;

typedef struct {
  int lineno;
  int column;
  const File *src;
  View line;
  Token token;
} TokenInfo;

typedef struct {
  Unit unit;
  View text;
} TextAndUnit;

typedef struct {
  bool invalid;
  size_t cap;
  size_t num;
  Token *at;
} TokenList;

typedef struct {
  bool invalid;
  Token name;
  TokenList params;
```

```
} Decl;

typedef enum {
  STMT_EXPR,
  STMT_RETURN,
  STMT_LIST,
  STMT_WHILE,
} StmtType;

typedef enum {
  EXPR_STRING, // string
  EXPR_NUMBER, // number
  EXPR_IDENT,  // ident
  EXPR_PAREN,  // (expr)
  EXPR_CALL,   // expr (expr, expr, ... expr)
  EXPR_SUB,    // expr - expr
  EXPR_ADD,    // expr + expr
  EXPR_ASSIGN, // expr = expr
} ExprType;

typedef struct {
  bool invalid;
  ExprType type;
  void *ptr;
} Expr;

typedef struct {
  bool invalid;
  Expr func;
  size_t args_cap;
  size_t args_num;
  Expr *args;
} ExprCall;

typedef struct {
  bool invalid;
```

```
  Expr left;
  Expr right;
} ExprBinary;

typedef struct {
  bool invalid;
  const Token *token;
} ExprLeaf;

typedef struct {
  bool invalid;
  Expr expr;
} ExprParen;

typedef struct {
  bool invalid;
  StmtType type;
  void *ptr;
} Stmt;

typedef struct StmtExpr {
  bool invalid;
  Expr expr;
} StmtExpr;

typedef struct StmtWhile {
  bool invalid;
  Expr cond;
  Stmt body;
} StmtWhile;

typedef struct StmtList {
  bool invalid;
  size_t num;
  size_t cap;
  Stmt *at;
```

```
} StmtList;

typedef struct {
  bool invalid;
  size_t decl_index;
  TokenList vars;
  Stmt body;
} Fdef;

typedef struct {
  bool invalid;
  Unit unit;
  const Token *token;
  size_t decls_cap;
  size_t decls_num;
  Decl *decls;
  size_t fdefs_cap;
  size_t fdefs_num;
  Fdef *fdefs;
} AST;

typedef struct {
  AST ast;
  Fdef fdef;
  Decl decl;
} ExprCtx;

typedef struct {
  AST ast;
  void *ptr;
} ASTAndPtr;

typedef struct {
  bool invalid;
  AST ast;
} NiceAST;
```

```
typedef struct {
  FILE *stream;
  int string_index;
} DataCtx;

typedef struct {
  FILE *stream;
  int string_index;
  int label_index;
  Fdef fdef;
  Decl decl;
  AST ast;
} TextCtx;

// functions
//   main sequence functions
File read_file(View filename);
Unit preprocess(File file);
AST parse(Unit unit);
NiceAST analyze(AST ast);
bool generate(NiceAST nast, FILE *stream);
TokenInfo token_info(Unit unit, Token token);
void print_token_location(TokenInfo info, FILE *stream);
void print_token_context(TokenInfo info, FILE *stream);
//   view handling
View vbuf(size_t size, const char *buf);
View vstr(const char *str);
View vcut(View v, size_t n);
View vrcut(View v, size_t n);
size_t vcspn(View v, const char *t);
size_t vspn(View v, const char *t);
bool vequ(View v1, View v2);
//   lexer and preprocessor
Unit empty_unit();
Unit unit_include(Unit unit, File file);
```

```
Unit unit_include_tokens(Unit unit, size_t file_index);
TokenAndText chop_token(View text, size_t file_index);
TextAndUnit run_directive(Unit unit,
                          View text, size_t file_index);
Unit unit_add_token(Unit unit, Token token);
View chop_usless(View text);
TextAndUnit run_directive_include(TextAndUnit tau,
                                  size_t file_index);
TextAndUnit run_directive_file(TextAndUnit tau,
                               size_t file_index);
//   parser
AST empty_ast(Unit unit);
AST parse_func(AST ast);
struct ParseDecl { AST ast; Decl decl; }
parse_decl(AST ast);
struct ParseFdef { AST ast; Fdef fdef; }
parse_fdef(AST ast, size_t decl_index);
struct ParseToken { AST ast; Token token; }
parse_token(AST ast, TokenType type, const char *name);
struct ParseIdentList { AST ast; TokenList list; }
parse_ident_list(AST ast, const char *name);
TokenList tl_empty();
TokenList tl_add(TokenList list, Token token);
struct ParseStmt { AST ast; Stmt stmt; }
parse_stmt(AST ast);
StmtType token_to_stmt_type(Token token);
ASTAndPtr parse_stmt_expr(AST ast);
ASTAndPtr parse_stmt_list(AST ast);
ASTAndPtr parse_stmt_return(AST ast);
ASTAndPtr parse_stmt_while(AST ast);
struct ParseExpr { AST ast; Expr expr; }
parse_expr(AST ast);
struct ParseExpr parse_postfix(AST ast);
struct ParseExpr parse_base(AST ast);
ASTAndPtr parse_expr_paren(AST ast);
ASTAndPtr parse_expr_binary(AST ast);
```

```
ASTAndPtr parse_expr_call(AST ast);
//   semantic
NiceAST analyze_decls(NiceAST n);
NiceAST analyze_fdefs(NiceAST n);
NiceAST analyze_return(NiceAST n);
NiceAST analyze_expresions(NiceAST n);
bool analyze_return_stmt(Stmt s);
bool analyze_stmt(ExprCtx ctx, Stmt stmt);
bool analyze_expr(ExprCtx ctx, Expr expr);
bool analyze_expr_binary(ExprCtx ctx,
                         ExprBinary *eb, bool lvalue);
bool analyze_expr_left(ExprCtx ctx, Expr expr);
int find_token(TokenList list, Token token);
int find_decl(AST ast, Token token);
bool analyze_expr_ident(ExprCtx ctx, ExprLeaf *leaf);
bool analyze_expr_call(ExprCtx ctx, ExprCall *call);
//   generation
bool data_ast(DataCtx *ctx, AST ast);
bool data_stmt(DataCtx *ctx, Stmt stmt);
bool data_stmt_list(DataCtx *ctx, StmtList *sl);
bool data_expr(DataCtx *ctx, Expr expr);
bool data_string(DataCtx *ctx, ExprLeaf *leaf);
bool data_binary(DataCtx *ctx, ExprBinary *binary);
bool data_call(DataCtx *ctx, ExprCall *call);
bool text_ast(TextCtx *ctx, AST ast);
bool text_func(TextCtx *ctx);
bool text_stmt(TextCtx *ctx, Stmt stmt);
bool text_stmt_expr(TextCtx *ctx, StmtExpr *s);
bool text_stmt_return(TextCtx *ctx, StmtExpr *s);
bool text_stmt_list(TextCtx *ctx, StmtList *s);
bool text_stmt_while(TextCtx *ctx, StmtWhile *s);
bool text_expr(TextCtx *ctx, Expr expr);
bool text_expr_left(TextCtx *ctx, Expr expr);
bool text_expr_left_ident(TextCtx *ctx, ExprLeaf *e);
bool text_expr_add(TextCtx *ctx, ExprBinary *e);
bool text_expr_assign(TextCtx *ctx, ExprBinary *e);
```

```
bool text_expr_call(TextCtx *ctx, ExprCall *e);
bool text_expr_ident(TextCtx *ctx, ExprLeaf *e);
bool text_expr_number(TextCtx *ctx, ExprLeaf *e);
bool text_expr_paren(TextCtx *ctx, ExprParen *e);
bool text_expr_string(TextCtx *ctx, ExprLeaf *e);
bool text_expr_sub(TextCtx *ctx, ExprBinary *e);

// inplementation

int main(int argc, char **argv) {
  if (argc < 2) {
    printf("Usage: gw input-file [output-file]\n\n");
    return EXIT_SUCCESS;
  }
  FILE *out = stdout;
  if (argc > 2) {
    out = fopen(argv[2], "wb");
    if (!out) {
      err("fail to open the file \'%s\'\n", argv[2]);
      return EXIT_FAILURE;
    }
  }
  File     file = read_file(vstr(argv[1]));
  Unit     unit = preprocess(file);
  AST      ast  = parse(unit);
  NiceAST  nast = analyze(ast);
  bool     ok   = generate(nast, out);
  if (argc > 2) {
    fclose(out);
  }
  if (!ok)
    printf("compilation terminated\n");
  return ok ? EXIT_SUCCESS : EXIT_FAILURE;
}

bool generate(NiceAST n, FILE *stream) {
```

```
  if (n.invalid)
    return false;
  bool ok = false;
  fprintf(stream, "\n.data\n");
  DataCtx dctx = {0};
  dctx.stream = stream;
  dctx.string_index = 0;
  ok = data_ast(&dctx, n.ast);
  if (!ok)
    return false;
  fprintf(stream, "\n.text\n");
  TextCtx tctx = {0};
  tctx.stream = stream;
  tctx.string_index = 0;
  tctx.label_index = 0;
  tctx.ast = n.ast;
  ok = text_ast(&tctx, n.ast);
  if (!ok)
    return false;
  fprintf(stream, "\n# THE END\n");
  return true;
}

bool text_ast(TextCtx *ctx, AST ast) {
  for (size_t i = 0; i < ast.fdefs_num; i++) {
    ctx->fdef = ast.fdefs[i];
    ctx->decl = ast.decls[ctx->fdef.decl_index];
    bool ok = text_func(ctx);
    if (!ok)
      return false;
  }
  return true;
}

bool text_func(TextCtx *ctx) {
  // prolog
```

```
  fprintf(ctx->stream, "\n.globl ");
  View name = ctx->decl.name.text;
  fwrite(name.at, 1, name.size, ctx->stream);
  fputc('\n', ctx->stream);
  fwrite(name.at, 1, name.size, ctx->stream);
  fprintf(ctx->stream, ":\n");
  fprintf(ctx->stream, "  pushl  %%ebp\n");
  fprintf(ctx->stream, "  movl   %%esp, %%ebp\n");
  // local vars
  if (ctx->fdef.vars.num != 0) {
    int offset = REG_SIZE * ctx->fdef.vars.num;
    fprintf(ctx->stream, "  subl   $%i, %%esp\n", offset);
  }
  // body
  bool ok = text_stmt(ctx, ctx->fdef.body);
  if (!ok)
    return false;
  // endlog
  return true;
}

bool text_stmt(TextCtx *ctx, Stmt stmt) {
  switch (stmt.type) {
    case STMT_EXPR:
      return text_stmt_expr(ctx, stmt.ptr);
    case STMT_RETURN:
      return text_stmt_return(ctx, stmt.ptr);
    case STMT_LIST:
      return text_stmt_list(ctx, stmt.ptr);
    case STMT_WHILE:
      return text_stmt_while(ctx, stmt.ptr);
    default:
      return false;
  }
}
```

```
bool text_stmt_expr(TextCtx *ctx, StmtExpr *s) {
  bool ok = text_expr(ctx, s->expr);
  fprintf(ctx->stream, "  popl   %%eax\n");
  return ok;
}

bool text_stmt_return(TextCtx *ctx, StmtExpr *s) {
  bool ok = text_expr(ctx, s->expr);
  fprintf(ctx->stream, "  popl   %%eax\n");
  fprintf(ctx->stream, "  movl   %%ebp, %%esp\n");
  fprintf(ctx->stream, "  popl   %%ebp\n");
  fprintf(ctx->stream, "  ret\n");
  return ok;
}

bool text_stmt_list(TextCtx *ctx, StmtList *s) {
  for (size_t i = 0; i < s->num; i++) {
    bool ok = text_stmt(ctx, s->at[i]);
    if (!ok)
      return false;
  }
  return true;
}

bool text_stmt_while(TextCtx *ctx, StmtWhile *s) {
  bool ok = false;
  int lbbody = ctx->label_index;
  ctx->label_index++;
  int lbcond = ctx->label_index;
  ctx->label_index++;
  // prolog
  fprintf(ctx->stream, "  jmp    L%i\n", lbcond);
  fprintf(ctx->stream, "L%i:\n", lbbody);
  // body
  ok = text_stmt(ctx, s->body);
  if (!ok)
```

```
    return false;
  // cond
  fprintf(ctx->stream, "L%i:\n", lbcond);
  ok = text_expr(ctx, s->cond);
  if (!ok)
    return false;
  fprintf(ctx->stream, "  popl   %%eax\n");
  fprintf(ctx->stream, "  testl  %%eax, %%eax\n");
  fprintf(ctx->stream, "  jne    L%i\n", lbbody);
  // endlog
  return true;
}

bool text_expr(TextCtx *ctx, Expr expr) {
  switch (expr.type) {
    case EXPR_ADD:
      return text_expr_add(ctx, expr.ptr);
    case EXPR_ASSIGN:
      return text_expr_assign(ctx, expr.ptr);
    case EXPR_CALL:
      return text_expr_call(ctx, expr.ptr);
    case EXPR_IDENT:
      return text_expr_ident(ctx, expr.ptr);
    case EXPR_NUMBER:
      return text_expr_number(ctx, expr.ptr);
    case EXPR_PAREN:
      return text_expr_paren(ctx, expr.ptr);
    case EXPR_STRING:
      return text_expr_string(ctx, expr.ptr);
    case EXPR_SUB:
      return text_expr_sub(ctx, expr.ptr);
    default:
      return false;
  }
}
```

```
bool text_expr_left(TextCtx *ctx, Expr expr) {
  if (expr.type != EXPR_IDENT)
    return false;
  return text_expr_left_ident(ctx, expr.ptr);
}

bool text_expr_add(TextCtx *ctx, ExprBinary *e) {
  bool ok = false;
  ok = text_expr(ctx, e->left);
  if (!ok)
    return false;
  ok = text_expr(ctx, e->right);
  if (!ok)
    return false;
  fprintf(ctx->stream, "  popl   %%eax\n"); // right
  fprintf(ctx->stream, "  popl   %%edx\n"); // left
  fprintf(ctx->stream, "  addl   %%edx, %%eax\n");
  fprintf(ctx->stream, "  pushl  %%eax\n");
  return true;
}

bool text_expr_assign(TextCtx *ctx, ExprBinary *e) {
  bool ok = false;
  ok = text_expr_left(ctx, e->left);
  if (!ok)
    return false;
  ok = text_expr(ctx, e->right);
  if (!ok)
    return false;
  fprintf(ctx->stream, "  popl   %%edx\n"); // right
  fprintf(ctx->stream, "  popl   %%eax\n"); // left
  fprintf(ctx->stream, "  movl   %%edx, (%%eax)\n");
  fprintf(ctx->stream, "  pushl  %%edx\n"); // (a = 5) == 5
  return true;
}
```

```
bool text_expr_call(TextCtx *ctx, ExprCall *e) {
  size_t i = e->args_num - 1;
  for (size_t n = 0; n < e->args_num; n++) {
    bool ok = text_expr(ctx, e->args[i]);
    if (!ok)
      return false;
    i--;
  }
  bool ok = text_expr(ctx, e->func);
  if (!ok)
    return false;
  fprintf(ctx->stream, "  popl   %%eax\n");
  fprintf(ctx->stream, "  call   *%%eax\n");
  if (e->args_num != 0) {
    int offset = REG_SIZE * e->args_num;
    fprintf(ctx->stream, "  addl   $%i, %%esp\n", offset);
  }
  fprintf(ctx->stream, "  pushl  %%eax\n");
  return true;
}

bool text_expr_ident(TextCtx *ctx, ExprLeaf *e) {
  Token token = e->token[0];
  int vi = find_token(ctx->fdef.vars, token);
  if (vi >= 0) {
    int offset = REG_SIZE * (vi + 1); // starts with 4
    fprintf(ctx->stream, "  pushl  -%i(%%ebp)\n", offset);
    return true;
  }
  int pi = find_token(ctx->decl.params, token);
  if (pi >= 0) {
    int offset = REG_SIZE * (pi + 2); // starts with 8
    fprintf(ctx->stream, "  pushl  %i(%%ebp)\n", offset);
    return true;
  }
  int di = find_decl(ctx->ast, token);
```

```
  if (di >= 0) {
    View name = ctx->ast.decls[di].name.text;
    fprintf(ctx->stream, "  pushl  $");
    fwrite(name.at, 1, name.size, ctx->stream);
    fputc('\n', ctx->stream);
    return true;
  }
  return false;
}

bool text_expr_left_ident(TextCtx *ctx, ExprLeaf *e) {
  Token token = e->token[0];
  int vi = find_token(ctx->fdef.vars, token);
  if (vi >= 0) {
    int offset = REG_SIZE * (vi + 1); // starts with 4
    fprintf(ctx->stream, "  leal   -%i(%%ebp), %%eax\n",
            offset);
    fprintf(ctx->stream, "  pushl  %%eax\n");
    return true;
  }
  int pi = find_token(ctx->decl.params, token);
  if (pi >= 0) {
    int offset = REG_SIZE * (pi + 2); // starts with 8
    fprintf(ctx->stream, "  leal   %i(%%ebp), %%eax\n",
            offset);
    fprintf(ctx->stream, "  pushl  %%eax\n");
    return true;
  }
  int di = find_decl(ctx->ast, token);
  if (di >= 0) {
    View name = ctx->ast.decls[di].name.text;
    fprintf(ctx->stream, "  pushl  $");
    fwrite(name.at, 1, name.size, ctx->stream);
    fputc('\n', ctx->stream);
    return true;
  }
```

```
  return true;
}

bool text_expr_number(TextCtx *ctx, ExprLeaf *e) {
  fprintf(ctx->stream, "  pushl  $");
  View num = e->token[0].text;
  fwrite(num.at, 1, num.size, ctx->stream);
  fputc('\n', ctx->stream);
  return true;
}

bool text_expr_paren(TextCtx *ctx, ExprParen *e) {
  return text_expr(ctx, e->expr);
}

bool text_expr_string(TextCtx *ctx, ExprLeaf *e) {
  int si = ctx->string_index;
  ctx->string_index++;
  fprintf(ctx->stream, "  pushl  $S%i\n", si);
  return true;
}

bool text_expr_sub(TextCtx *ctx, ExprBinary *e) {
  bool ok = false;
  ok = text_expr(ctx, e->left);
  if (!ok)
    return false;
  ok = text_expr(ctx, e->right);
  if (!ok)
    return false;
  fprintf(ctx->stream, "  popl   %%edx\n"); // right
  fprintf(ctx->stream, "  popl   %%eax\n"); // left
  fprintf(ctx->stream, "  subl   %%edx, %%eax\n");
  fprintf(ctx->stream, "  pushl  %%eax\n");
  return true;
}
```

```
bool data_ast(DataCtx *ctx, AST ast) {
  for (size_t i = 0; i < ast.fdefs_num; i++) {
    Fdef fdef = ast.fdefs[i];
    bool ok = data_stmt(ctx, fdef.body);
    if (!ok)
      return false;
  }
  return true;
}

bool data_stmt(DataCtx *ctx, Stmt stmt) {
  switch (stmt.type) {
    case STMT_EXPR:
      return data_expr(ctx, ((StmtExpr *)stmt.ptr)->expr);
    case STMT_RETURN:
      return data_expr(ctx, ((StmtExpr *)stmt.ptr)->expr);
    case STMT_WHILE:
      return data_stmt(ctx, ((StmtWhile *)stmt.ptr)->body);
    case STMT_LIST:
      return data_stmt_list(ctx, stmt.ptr);
    default:
      return false; // TODO: error message
  }
}

bool data_expr(DataCtx *ctx, Expr expr) {
  switch (expr.type) {
    case EXPR_NUMBER:
      return true;
    case EXPR_IDENT:
      return true;
    case EXPR_CALL:
      return data_call(ctx, expr.ptr);
    case EXPR_ADD:
      return data_binary(ctx, expr.ptr);
```

```
    case EXPR_SUB:
      return data_binary(ctx, expr.ptr);
    case EXPR_ASSIGN:
      return data_binary(ctx, expr.ptr);
    case EXPR_STRING:
      return data_string(ctx, expr.ptr);
    case EXPR_PAREN:
      return data_expr(ctx, ((ExprParen *)expr.ptr)->expr);
    default:
      return false; // TODO: error message
  }
}

bool data_string(DataCtx *ctx, ExprLeaf *leaf) {
  int si = ctx->string_index;
  ctx->string_index++;
  fprintf(ctx->stream, "\nS%i: .string ", si);
  View str = leaf->token[0].text;
  fwrite(str.at, 1, str.size, ctx->stream);
  fputc('\n', ctx->stream);
  return true;
}

bool data_binary(DataCtx *ctx, ExprBinary *binary) {
  bool ok = data_expr(ctx, binary->left);
  if (!ok)
    return false;
  return data_expr(ctx, binary->right);
}

bool data_call(DataCtx *ctx, ExprCall *call) {
  // ignore call->func
  for (size_t i = 0; i < call->args_num; i++) {
    bool ok = data_expr(ctx, call->args[i]);
    if (!ok)
      return false;
```

```
  }
  return true;
}

bool data_stmt_list(DataCtx *ctx, StmtList *sl) {
  for (size_t i = 0; i < sl->num; i++) {
    bool ok = data_stmt(ctx, sl->at[i]);
    if (!ok)
      return false;
  }
  return true;
}

File read_file(View filename) {
  File file = {0};
  file.id = vstr("");
  file.invalid = true;
  // filename to c string
  file.name = malloc(filename.size + 1);
  if (!file.name) {
    err("fail to allocate memory for file name\n");
    return file;
  }
  memcpy(file.name, filename.at, filename.size);
  file.name[filename.size] = '\0';
  // open file
  FILE *f = fopen(file.name, "rb");
  if (!f) {
    err("file not found \'%s\'\n", file.name);
    return file;
  }
  // allocate buffer
  fseek(f, 0, SEEK_END);
  file.size = ftell(f);
  fseek(f, 0, SEEK_SET);
  file.buffer = malloc(file.size);
```

```
  if (!file.buffer) {
    fclose(f);
    err("fail to allocate memory for file buffer\n");
    return file;
  }
  // read file
  size_t total = fread(file.buffer, 1, file.size, f);
  fclose(f);
  if (total != file.size) {
    err("fail to read file \'%s\'\n", file.name);
    return file;
  }
  file.text = vbuf(file.size, file.buffer);
  file.invalid = false;
  return file;
}

View vbuf(size_t size, const char *buf) {
  return (View) { .size = size, .at = buf };
}

View vstr(const char *str) {
  return (View) { .size = strlen(str), .at = str };
}

View vcut(View v, size_t n) {
  if (v.size < n)
    n = v.size;
  return vbuf(v.size - n, v.at + n);
}

View vrcut(View v, size_t n) {
  if (v.size < n)
    n = v.size;
  return vbuf(v.size - n, v.at);
}
```

```
size_t vcspn(View v, const char *t) {
  for (size_t i = 0; i < v.size; i++) {
    if (strchr(t, v.at[i]) != NULL) {
      return i;
    }
  }
  return v.size;
}

size_t vspn(View v, const char *t) {
  for (size_t i = 0; i < v.size; i++) {
    if (strchr(t, v.at[i]) == NULL) {
      return i;
    }
  }
  return v.size;
}

bool vequ(View v1, View v2) {
  if (v1.size != v2.size)
    return false;
  return memcmp(v1.at, v2.at, v1.size) == 0;
}

View chop_usless(View text) {
  for (;;) {
    if (text.size == 0)
      break;
    if (text.at[0] == ':')
      text = vcut(text, vcspn(text, "\n\r"));
    size_t spaces = vspn(text, " \t\n\r\v");
    if (spaces == 0)
      break;
    text = vcut(text, spaces);
  }
```

```
  return text;
}

TokenAndText chop_token(View text, size_t file_index) {
  TokenAndText tat = {0};
  tat.token.type = T_INVALID;
  tat.token.file_index = file_index;
  tat.text = chop_usless(text);
  if (tat.text.size == 0) {
    tat.token.type = T_EOF;
    tat.token.text = vstr("");
    return tat;
  }

#define ONE_CHAR_TOKEN(TYPE) \
  tat.token.text = vbuf(1, tat.text.at); \
  tat.token.type = TYPE;

  switch (tat.text.at[0]) {
    case '#': ONE_CHAR_TOKEN(T_SHARP); break;
    case '-': ONE_CHAR_TOKEN(T_MINUS); break;
    case '+': ONE_CHAR_TOKEN(T_PLUS); break;
    case '=': ONE_CHAR_TOKEN(T_EQUAL); break;
    case ';': ONE_CHAR_TOKEN(T_SEMICOLON); break;
    case ',': ONE_CHAR_TOKEN(T_COMMA); break;
    case '{': ONE_CHAR_TOKEN(T_BRACE_BEG); break;
    case '}': ONE_CHAR_TOKEN(T_BRACE_END); break;
    case '(': ONE_CHAR_TOKEN(T_PAREN_BEG); break;
    case ')': ONE_CHAR_TOKEN(T_PAREN_END); break;
    case '\"':
      tat.token.text = vbuf(
                         vcspn(vcut(tat.text, 1), "\"") + 2,
                         tat.text.at
                       );
      if (tat.token.text.at[tat.token.text.size - 1]
          != '\"') {
```

```
      err("missing \" character\n");
      return tat;
    }
    tat.token.type = T_STRING;
    break;
  default:
    if (isalpha(tat.text.at[0])
        || tat.text.at[0] == '_') {
      tat.token.type = T_IDENT;
      // vspn isalnum + '_'
      tat.token.text = vbuf(vspn(tat.text,
                                 "qwertyuiopasdfghjklzxcvbnm"
                                 "QWERTYUIOPASDFGHJKLZXCVBNM"
                                 "_0123456789"),
                            tat.text.at);
    }
    else if (isdigit(tat.text.at[0])) {
      tat.token.type = T_NUMBER;
      tat.token.text = vbuf(
                           vspn(tat.text, "0123456789"),
                           tat.text.at);
    }
    else {
      err("invalid token starts with \'%c\'\n",
          tat.text.at[0]);
      return tat;
    }
}
if (tat.token.type == T_IDENT) {
  if (vequ(tat.token.text, vstr("while"))) {
    tat.token.type = T_WHILE;
  }
  else if (vequ(tat.token.text, vstr("return"))) {
    tat.token.type = T_RETURN;
  }
}
```

```
  // chop literaly
  tat.text = vcut(tat.text, tat.token.text.size);
  return tat;
}

Unit unit_include(Unit unit, File file) {
  if (unit.invalid || file.invalid) {
    unit.invalid = true;
    return unit;
  }
  // add file to files
  if (unit.files_cap == unit.files_num) {
    unit.files_cap *= 2;
    unit.files = realloc(unit.files,
                         unit.files_cap * sizeof(File));
    if (!unit.files) {
      unit.invalid = true;
      err("fail to reallocate memory for files");
      return unit;
    }
  }
  unit.files[unit.files_num] = file;
  unit.files_num++;
  // add tokens
  return unit_include_tokens(unit, unit.files_num - 1);
}

Unit empty_unit() {
  Unit unit = {0};
  unit.invalid = true;
  //
  unit.files_cap = 2;
  unit.files_num = 0;
  unit.files = malloc(unit.files_cap * sizeof(File));
  if (!unit.files) {
    err("fail to allocate memory for files");
```

```
    return unit;
  }
  //
  unit.tokens_cap = 2;
  unit.tokens_num = 0;
  unit.tokens = malloc(unit.tokens_cap * sizeof(Token));
  if (!unit.tokens) {
    err("fail to allocate memory for tokens");
    return unit;
  }
  //
  unit.invalid = false;
  return unit;
}

Unit preprocess(File file) {
  return unit_add_token(unit_include(empty_unit(), file),
                        (Token) { .type = T_EOF });
}

Unit unit_add_token(Unit unit, Token token) {
  if (unit.invalid || token.type == T_INVALID) {
    unit.invalid = true;
    return unit;
  }
  if (unit.tokens_cap == unit.tokens_num) {
    unit.tokens_cap *= 2;
    unit.tokens = realloc(unit.tokens,
                          unit.tokens_cap * sizeof(Token));
    if (!unit.tokens) {
      unit.invalid = true;
      err("fail to reallocate memory for tokens");
      return unit;
    }
  }
  unit.tokens[unit.tokens_num] = token;
```

```
  unit.tokens_num++;
  return unit;
}

Unit unit_include_tokens(Unit unit, size_t file_index) {
  if (unit.invalid)
    return unit;
  File file = unit.files[file_index];
  View text = file.text;
  for (;;) {
    TokenAndText tat = chop_token(text, file_index);
    if (tat.token.type == T_EOF) break;
    if (tat.token.type == T_SHARP) {
      TextAndUnit tau = run_directive(unit,
                                      tat.text,
                                      file_index);
      unit = tau.unit;
      text = tau.text;
    }
    else {
      unit = unit_add_token(unit, tat.token);
      text = tat.text;
    }
    if (unit.invalid) break;
  }
  return unit;
}

TextAndUnit run_directive(Unit unit,
                          View text,
                          size_t file_index) {
  TextAndUnit tau = {0};
  tau.unit = unit;
  tau.text = text;
  if (unit.invalid)
    return tau;
```

```
  TokenAndText tat = chop_token(tau.text, file_index);
  tau.text = tat.text;
  if (tat.token.type == T_EOF) {
    err("missing directive name\n");
    tau.unit.invalid = true;
    return tau;
  }
  if (tat.token.type == T_INVALID) {
    tau.unit.invalid = true;
    return tau;
  }
  if (tat.token.type != T_IDENT) {
    TokenInfo info = token_info(tau.unit, tat.token);
    print_token_location(info, stderr);
    err("directive name is not an indentifier\n");
    print_token_context(info, stderr);
    tau.unit.invalid = true;
    return tau;
  }
  if (vequ(tat.token.text, vstr("include"))) {
    return run_directive_include(tau, file_index);
  }
  else if (vequ(tat.token.text, vstr("file"))) {
    return run_directive_file(tau, file_index);
  }
  TokenInfo info = token_info(tau.unit, tat.token);
  print_token_location(info, stderr);
  err("unknown directive\n");
  print_token_context(info, stderr);
  tau.unit.invalid = true;
  return tau;
}

TextAndUnit run_directive_include(TextAndUnit tau,
                                  size_t file_index) {
  TokenAndText tat = chop_token(tau.text, file_index);
```

```
  tau.text = tat.text;
  if (tat.token.type == T_EOF) {
    err("missing file name\n");
    tau.unit.invalid = true;
    return tau;
  }
  if (tat.token.type == T_INVALID) {
    tau.unit.invalid = true;
    return tau;
  }
  if (tat.token.type != T_STRING) {
    TokenInfo info = token_info(tau.unit, tat.token);
    print_token_location(info, stderr);
    err("file name is not a string\n");
    print_token_context(info, stderr);
    tau.unit.invalid = true;
    return tau;
  }
  View path = vcut(vrcut(tat.token.text, 1), 1);
  tau.unit = unit_include(tau.unit, read_file(path));
  return tau;
}

TextAndUnit run_directive_file(TextAndUnit tau,
                               size_t file_index) {
  TokenAndText tat = chop_token(tau.text, file_index);
  tau.text = tat.text;
  if (tat.token.type == T_EOF) {
    err("missing file id\n");
    tau.unit.invalid = true;
    return tau;
  }
  if (tat.token.type == T_INVALID) {
    tau.unit.invalid = true;
    return tau;
  }
```

```
  if (tat.token.type != T_IDENT) {
    TokenInfo info = token_info(tau.unit, tat.token);
    print_token_location(info, stderr);
    err("file id is not an indentifier\n");
    print_token_context(info, stderr);
    tau.unit.invalid = true;
    return tau;
  }
  for (size_t i = 0; i < tau.unit.files_num; i++) {
    if (vequ(tau.unit.files[i].id, tat.token.text)) {
      tau.text.size = 0; // already included
    }
  }
  tau.unit.files[file_index].id = tat.token.text;
  return tau;
}

TokenInfo token_info(Unit unit, Token token) {
  TokenInfo info = {0};
  info.token = token;
  info.src = &unit.files[token.file_index];
  info.lineno = 1;
  info.column = 1;
  View text = info.src->text;
  const char *line_beg = text.at;
  for (size_t i = 0; i < text.size; i++) {
    if (text.at + i == token.text.at) {
      View string = vcut(text, line_beg - text.at);
      info.line = vbuf(vcspn(string, "\n"), string.at);
      return info;
    }
    if (text.at[i] == '\n') {
      info.column = 1;
      info.lineno++;
      line_beg = text.at + i + 1;
    }
```

```
    else if (text.at[i] != '\r') {
      info.column++;
    }
  }
  return info;
}

void print_token_location(TokenInfo info, FILE *stream) {
  fprintf(stream, "%s:%i:%i: ",
          info.src->name, info.lineno, info.column);
}

void print_token_context(TokenInfo info, FILE *stream) {
  fprintf(stream, "%4i | ", info.lineno);
  fwrite(info.line.at, 1, info.line.size, stream);
  fprintf(stream, "\n     | ");
  for (size_t i = 1; i < info.column; i++) {
    fputc(' ', stream);
  }
  fputc('^', stream);
  size_t lim = info.line.size
             + info.line.at
             - info.token.text.at;
  lim--;
  lim = (lim < info.token.text.size)
        ? lim
        : info.token.text.size;
  for (size_t i = 1; i < lim; i++) {
    fputc('~', stream);
  }
  fputc('\n', stream);
}

AST empty_ast(Unit unit) {
  AST ast = {0};
  ast.invalid = true;
```

```
  ast.unit = unit;
  ast.token = unit.tokens;
  if (unit.invalid)
    return ast;
  ast.decls_num = 0;
  ast.decls_cap = 2;
  ast.decls = malloc(ast.decls_cap * sizeof(Decl));
  if (!ast.decls) {
    err("fail to allocate memory "
        "for function declarations\n");
    return ast;
  }
  ast.fdefs_num = 0;
  ast.fdefs_cap = 2;
  ast.fdefs = malloc(ast.fdefs_cap * sizeof(Fdef));
  if (!ast.fdefs) {
    err("fail to allocate memory "
        "for function definitions\n");
    return ast;
  }
  ast.invalid = false;
  return ast;
}

struct ParseToken parse_token(AST ast,
                              TokenType type,
                              const char *name) {
  PRINT_PARSE_FUNC;
  if (ast.invalid)
    return (struct ParseToken) {
      .ast = ast,
      .token = (Token) { .type = T_INVALID }
    };
  Token token = ast.token[0];
  if (token.type == T_EOF) {
    err("missing %s\n", name);
```

```
    token.type = T_INVALID;
    return (struct ParseToken) {
      .ast = ast,
      .token = token
    };
  }
  if (token.type != type) {
    TokenInfo info = token_info(ast.unit, token);
    print_token_location(info, stderr);
    err("unexpected token, where %s is expected\n", name);
    print_token_context(info, stderr);
    token.type = T_INVALID;
    return (struct ParseToken) {
      .ast = ast,
      .token = token
    };
  }
  ast.token = &ast.token[1];
  return (struct ParseToken) { .ast = ast, .token = token };
}

struct ParseDecl parse_decl(AST ast) {
  PRINT_PARSE_FUNC;
  struct ParseDecl pdecl = {0};
  pdecl.ast = ast;
  pdecl.decl.invalid = true;
  struct ParseToken pt;
  if (ast.invalid)
    return pdecl;
  //
  pt = parse_token(pdecl.ast, T_IDENT, "function name");
  if (pt.token.type == T_INVALID)
    return pdecl;
  pdecl.ast = pt.ast;
  pdecl.decl.name = pt.token;
  //
```

```
  pt = parse_token(pdecl.ast, T_PAREN_BEG, "opening paren");
  if (pt.token.type == T_INVALID)
    return pdecl;
  pdecl.ast = pt.ast;
  //
  struct ParseIdentList pil;
  pil  = parse_ident_list(pdecl.ast, "parameter");
  if (pil.list.invalid)
    return pdecl;
  pdecl.ast = pil.ast;
  pdecl.decl.params = pil.list;
  //
  pt = parse_token(pdecl.ast, T_PAREN_END, "closing paren");
  if (pt.token.type == T_INVALID)
    return pdecl;
  pdecl.ast = pt.ast;
  pdecl.decl.invalid = false;
  return pdecl;
}

AST parse_func(AST ast) {
  PRINT_PARSE_FUNC;
  struct ParseDecl pd = parse_decl(ast);
  ast = pd.ast;
  if (pd.decl.invalid) {
    ast.invalid = true;
    return ast;
  }
  // add decl
  if (ast.decls_cap == ast.decls_num) {
    ast.decls_cap *= 2;
    ast.decls = realloc(ast.decls,
                        ast.decls_cap * sizeof(Decl));
    if (!ast.decls) {
      err("fail to reallocate memory "
          "for function declarations\n");
```

```
      ast.invalid = true;
      return ast;
    }
  }
  ast.decls[ast.decls_num] = pd.decl;
  ast.decls_num++;
  if (ast.token[0].type == T_SEMICOLON) {
    ast.token = &ast.token[1];
    return ast;
  }
  //
  struct ParseFdef pf = parse_fdef(ast, ast.decls_num - 1);
  ast = pf.ast;
  if (pf.fdef.invalid) {
    ast.invalid = true;
    return ast;
  }
  // add fdef
  if (ast.fdefs_cap == ast.fdefs_num) {
    ast.fdefs_cap *= 2;
    ast.fdefs = realloc(ast.fdefs,
                        ast.fdefs_cap * sizeof(Decl));
    if (!ast.fdefs) {
      err("fail to reallocate memory "
          "for function declarations\n");
      ast.invalid = true;
      return ast;
    }
  }
  ast.fdefs[ast.fdefs_num] = pf.fdef;
  ast.fdefs_num++;
  return ast;
}

AST parse(Unit unit) {
  if (unit.invalid)
```

```
    return (AST) { .invalid = true };
  AST ast = empty_ast(unit);
  while (!ast.invalid) {
    if (ast.token->type == T_EOF)
      break;
    ast = parse_func(ast);
  }
  return ast;
}

struct ParseFdef parse_fdef(AST ast, size_t decl_index) {
  PRINT_PARSE_FUNC;
  struct ParseFdef pfdef = {0};
  pfdef.ast = ast;
  pfdef.fdef.invalid = true;
  pfdef.fdef.decl_index = decl_index;
  if (ast.invalid)
    return pfdef;
  //
  struct ParseIdentList pil;
  pil = parse_ident_list(pfdef.ast, "variable");
  if (pil.list.invalid)
    return pfdef;
  pfdef.ast = pil.ast;
  pfdef.fdef.vars = pil.list;
  //
  struct ParseStmt ps = parse_stmt(pfdef.ast);
  pfdef.ast = ps.ast;
  if (ps.stmt.invalid)
    return pfdef;
  pfdef.fdef.body = ps.stmt;
  //
  pfdef.fdef.invalid = false;
  return pfdef;
}
```

```
struct ParseIdentList parse_ident_list(AST ast,
                                       const char *name) {
  PRINT_PARSE_FUNC;
  struct ParseIdentList pil = {0};
  pil.ast = ast;
  pil.list.invalid = true;
  if (ast.invalid)
    return pil;
  pil.list = tl_empty();
  if (pil.list.invalid) {
    err("fail to create the %s list\n", name);
    return pil;
  }
  for (;;) {
    if (pil.ast.token[0].type != T_IDENT)
      break;
    pil.list = tl_add(pil.list, pil.ast.token[0]);
    if (pil.list.invalid) {
      err("fail to add element to the %s list\n", name);
      return pil;
    }
    pil.ast.token = &pil.ast.token[1];
    if (pil.ast.token[0].type != T_COMMA)
      break;
    pil.ast.token = &pil.ast.token[1];
  }
  pil.list.invalid = false;
  return pil;
}

StmtType token_to_stmt_type(Token token) {
  switch (token.type) {
    case T_RETURN: return STMT_RETURN;
    case T_WHILE: return STMT_WHILE;
    case T_BRACE_BEG: return STMT_LIST;
    default: return STMT_EXPR;
```

```
  }
}

struct ParseStmt parse_stmt(AST ast) {
  PRINT_PARSE_FUNC;
  struct ParseStmt ps = {0};
  ps.ast = ast;
  ps.stmt.invalid = true;
  if (ast.invalid)
    return ps;
  ps.stmt.type = token_to_stmt_type(ps.ast.token[0]);
  ASTAndPtr aap = {0};
  switch (ps.stmt.type) {
    case STMT_EXPR:
      aap = parse_stmt_expr(ps.ast); break;
    case STMT_LIST:
      aap = parse_stmt_list(ps.ast); break;
    case STMT_RETURN:
      aap = parse_stmt_return(ps.ast); break;
    case STMT_WHILE:
      aap = parse_stmt_while(ps.ast); break;
    default:
      break;
  }
  ps.ast = aap.ast;
  if (!aap.ptr)
    return ps;
  ps.stmt.ptr = aap.ptr;
  ps.stmt.invalid = false;
  return ps;
}

ASTAndPtr parse_stmt_expr(AST ast) {
  PRINT_PARSE_FUNC;
  ASTAndPtr aap = {0};
  aap.ast = ast;
```

```
  aap.ptr = NULL;
  if (ast.invalid)
    return aap;
  struct ParseExpr pe = parse_expr(ast);
  aap.ast = pe.ast;
  if (pe.expr.invalid)
    return aap;
  StmtExpr *s = malloc(sizeof(StmtExpr));
  if (!s) {
    err("fail to allocate memory for expression-statement");
    return aap;
  }
  s->expr = pe.expr;
  struct ParseToken pt = parse_token(aap.ast,
                                     T_SEMICOLON,
                                     "semicolon");
  if (pt.token.type == T_INVALID)
    return aap;
  aap.ast = pt.ast;
  aap.ptr = s;
  return aap;
}

ASTAndPtr parse_stmt_list(AST ast) {
  PRINT_PARSE_FUNC;
  ASTAndPtr aap = {0};
  aap.ast = ast;
  aap.ptr = NULL;
  if (ast.invalid)
    return aap;
  struct ParseToken pt = parse_token(aap.ast,
                                     T_BRACE_BEG,
                                     "opening brace");
  if (pt.token.type == T_INVALID)
    return aap;
  aap.ast = pt.ast;
```

```
StmtList *s = malloc(sizeof(StmtList));
if (!s) {
  err("fail to allocate memory for statement-list");
  return aap;
}
s->num = 0;
s->cap = 2;
s->at = malloc(s->cap * sizeof(Stmt));
if (!s->at) {
  err("fail to allocate memory for statement-list");
  return aap;
}
for (;;) {
  if (aap.ast.token[0].type == T_BRACE_END)
    break;
  struct ParseStmt ps = parse_stmt(aap.ast);
  aap.ast = ps.ast;
  if (ps.stmt.invalid)
    return aap;
  if (s->cap == s->num) {
    s->cap *= 2;
    s->at = realloc(s->at, s->cap * sizeof(Stmt));
    if (!s->at) {
      err("fail to reallocate memory for statement-list");
      return aap;
    }
  }
  s->at[s->num] = ps.stmt;
  s->num++;
}
pt = parse_token(aap.ast, T_BRACE_END, "closing brace");
if (pt.token.type == T_INVALID)
  return aap;
aap.ast = pt.ast;
aap.ptr = s;
return aap;
```

```
}

ASTAndPtr parse_stmt_return(AST ast) {
  PRINT_PARSE_FUNC;
  ASTAndPtr aap = {0};
  aap.ast = ast;
  aap.ptr = NULL;
  if (ast.invalid)
    return aap;
  struct ParseToken pt = parse_token(aap.ast,
                                     T_RETURN,
                                     "return keyword");
  if (pt.token.type == T_INVALID)
    return aap;
  aap.ast = pt.ast;
  return parse_stmt_expr(aap.ast);
}

ASTAndPtr parse_stmt_while(AST ast) {
  PRINT_PARSE_FUNC;
  ASTAndPtr aap = {0};
  aap.ast = ast;
  aap.ptr = NULL;
  if (ast.invalid)
    return aap;
  struct ParseToken pt = parse_token(aap.ast,
                                     T_WHILE,
                                     "while keyword");
  if (pt.token.type == T_INVALID)
    return aap;
  aap.ast = pt.ast;
  StmtWhile *s = malloc(sizeof(StmtWhile));
  if (!s) {
    err("fail to allocate memory for while-statement");
    return aap;
  }
```

```
  s->cond.type = EXPR_PAREN;
  ASTAndPtr aap2 = parse_expr_paren(aap.ast);
  aap.ast = aap2.ast;
  if (!aap2.ptr)
    return aap;
  s->cond.ptr = aap2.ptr;
  s->cond.invalid = false;
  struct ParseStmt ps = parse_stmt(aap.ast);
  aap.ast = ps.ast;
  if (ps.stmt.invalid)
    return aap;
  s->body = ps.stmt;
  aap.ptr = s;
  return aap;
}

struct ParseExpr parse_base(AST ast) {
  PRINT_PARSE_FUNC;
  struct ParseExpr pe = {0};
  pe.ast = ast;
  pe.expr.invalid = true;
  if (pe.ast.token[0].type == T_PAREN_BEG) {
    ASTAndPtr aap = parse_expr_paren(pe.ast);
    pe.ast = aap.ast;
    pe.expr.type = EXPR_PAREN;
    pe.expr.ptr = aap.ptr;
    if (!pe.expr.ptr)
      return pe;
    pe.expr.invalid = false;
    return pe;
  }
  ExprLeaf *e = malloc(sizeof(ExprLeaf));
  if (!e) {
    err("fail to allocate memory for expression-base\n");
    return pe;
  }
```

```
  e->token = pe.ast.token;
  if (e->token->type == T_EOF) {
    err("missing expression-base\n");
    return pe;
  }
  TokenInfo info;
  switch (e->token->type) {
    case T_STRING: pe.expr.type = EXPR_STRING; break;
    case T_NUMBER: pe.expr.type = EXPR_NUMBER; break;
    case T_IDENT: pe.expr.type = EXPR_IDENT; break;
    default:
      info = token_info(pe.ast.unit, e->token[0]);
      print_token_location(info, stderr);
      err("unexpected token, where "
          "expression-base is expected\n");
      print_token_context(info, stderr);
      return pe;
  }
  pe.ast.token = &pe.ast.token[1];
  pe.expr.invalid = false;
  pe.expr.ptr = e;
  return pe;
}

struct ParseExpr parse_postfix(AST ast) {
  PRINT_PARSE_FUNC;
  struct ParseExpr pe = parse_base(ast);
  if (pe.expr.invalid)
    return pe;
  if (pe.ast.token[0].type != T_PAREN_BEG)
    return pe;
  pe.ast.token = &pe.ast.token[1];
  //
  ExprCall *e = malloc(sizeof(ExprCall));
  if (!e) {
    err("fail to allocate memory for call-expression\n");
```

```
  pe.expr.invalid = true;
  return pe;
}
e->func = pe.expr;
e->args_cap = 2;
e->args_num = 0;
e->args = malloc(e->args_cap * sizeof(Expr));
pe.expr.invalid = true;
pe.expr.type = EXPR_CALL;
pe.expr.ptr = e;
if (!e) {
  err("fail to allocate memory for argument list\n");
  return pe;
}
for (;;) {
  if (pe.ast.token[0].type == T_PAREN_END)
    break;
  if (pe.ast.token[0].type == T_EOF)
    break;
  struct ParseExpr pe_arg = parse_expr(pe.ast);
  Expr arg = pe_arg.expr;
  pe.ast = pe_arg.ast;
  if (arg.invalid)
    return pe;
  if (e->args_cap == e->args_num) {
    e->args_cap *= 2;
    e->args = realloc(e->args,
                      e->args_cap * sizeof(Expr));
    if (!e->args){
      err("fail to reallocate memory "
          "for argument list\n");
      return pe;
    }
  }
  e->args[e->args_num] = arg;
  e->args_num++;
```

```
    if (pe.ast.token[0].type == T_PAREN_END)
      break;
    if (pe.ast.token[0].type != T_COMMA) {
      err("missing closing paren\n");
      return pe;
    }
    pe.ast.token = &pe.ast.token[1];
  }
  if (pe.ast.token[0].type != T_PAREN_END) {
    err("missing closing paren\n");
    return pe;
  }
  pe.ast.token = &pe.ast.token[1];
  pe.expr.invalid = false;
  return pe;
}

ASTAndPtr parse_expr_paren(AST ast) {
  PRINT_PARSE_FUNC;
  ASTAndPtr aap = {0};
  aap.ast = ast;
  aap.ptr = NULL;
  struct ParseToken pt = parse_token(aap.ast,
                                     T_PAREN_BEG,
                                     "opening paren");
  aap.ast = pt.ast;
  if (pt.token.type == T_INVALID)
    return aap;
  ExprParen *e = malloc(sizeof(ExprParen));
  if (!e) {
    err("fail to allocate memory "
        "for paren expression\n");
    return aap;
  }
  struct ParseExpr pe = parse_expr(aap.ast);
  aap.ast = pe.ast;
```

```
  if (pe.expr.invalid)
    return aap;
  e->expr = pe.expr;
  pt = parse_token(aap.ast, T_PAREN_END, "closing paren");
  aap.ast = pt.ast;
  if (pt.token.type == T_INVALID)
    return aap;
  aap.ptr = e;
  return aap;
}

struct ParseExpr parse_expr(AST ast) {
  PRINT_PARSE_FUNC;
  struct ParseExpr pe = parse_postfix(ast);
  if (pe.expr.invalid)
    return pe;
  Token token = pe.ast.token[0];
  if (token.type == T_PAREN_END
      || token.type == T_SEMICOLON
      || token.type == T_COMMA)
    return pe;
  //
  ExprType type = EXPR_ADD;
  TokenInfo info;
  switch (token.type) {
    case T_PLUS: type = EXPR_ADD; break;
    case T_MINUS: type = EXPR_SUB; break;
    case T_EQUAL: type = EXPR_ASSIGN; break;
    default:
      info = token_info(pe.ast.unit, token);
      print_token_location(info, stderr);
      err("unexpected token, where "
          "binary operator is expected\n");
      print_token_context(info, stderr);
      pe.expr.invalid = false;
      return pe;
```

```
  }
  pe.ast.token = &pe.ast.token[1];
  ExprBinary *e = malloc(sizeof(ExprBinary));
  if (!e) {
    err("fail to allocate memory for binary expression\n");
    pe.expr.invalid = false;
    return pe;
  }
  e->left = pe.expr;
  pe.expr.type = type;
  pe.expr.ptr = e;
  struct ParseExpr pright = parse_expr(pe.ast);
  pe.ast = pright.ast;
  e->right = pright.expr;
  // FIXME: y = a - b + c works like
  //        y = (a - (b + c)) but should be like
  //        y = ((a - b) + c)
  if (e->right.invalid) {
    pe.expr.invalid = false;
    return pe;
  }
  return pe;
}

NiceAST analyze_decls(NiceAST n) {
  PRINT_ANALYZE_FUNC;
  if (n.invalid)
    return n;
  for (size_t i = 0; i < n.ast.decls_num; i++) {
    Decl di = n.ast.decls[i];
    for (size_t j = i + 1; j < n.ast.decls_num; j++) {
      Decl dj = n.ast.decls[j];
      if (vequ(di.name.text, dj.name.text)) {
        if (di.params.num != dj.params.num) {
          TokenInfo dj_info = token_info(n.ast.unit,
                                         dj.name);
```

```
          TokenInfo di_info = token_info(n.ast.unit,
                                         di.name);
          // error
          print_token_location(dj_info, stderr);
          err("function redeclaration\n");
          print_token_context(dj_info, stderr);
          // note
          print_token_location(di_info, stderr);
          note("original function declared here:\n");
          print_token_context(di_info, stderr);
          n.invalid = true;
          return n;
        }
      }
    }
  }
  return n;
}

NiceAST analyze_fdefs(NiceAST n) {
  PRINT_ANALYZE_FUNC;
  if (n.invalid)
    return n;
  for (size_t i = 0; i < n.ast.fdefs_num; i++) {
    Fdef fi = n.ast.fdefs[i];
    Decl di = n.ast.decls[fi.decl_index];
    for (size_t j = i + 1; j < n.ast.fdefs_num; j++) {
      Fdef fj = n.ast.fdefs[j];
      Decl dj = n.ast.decls[fj.decl_index];
      if (vequ(di.name.text, dj.name.text)) {
        TokenInfo dj_info = token_info(n.ast.unit, dj.name);
        TokenInfo di_info = token_info(n.ast.unit, di.name);
        // error
        print_token_location(dj_info, stderr);
        err("function redefinition\n");
        print_token_context(dj_info, stderr);
```

```
        // note
        print_token_location(di_info, stderr);
        note("original function defined here:\n");
        print_token_context(di_info, stderr);
        n.invalid = true;
        return n;
      }
    }
  }
  return n;
}

NiceAST analyze_return(NiceAST n) {
  PRINT_ANALYZE_FUNC;
  if (n.invalid)
    return n;
  for (size_t i = 0; i < n.ast.fdefs_num; i++) {
    Fdef fdef = n.ast.fdefs[i];
    Decl decl = n.ast.decls[fdef.decl_index];
    bool ok = analyze_return_stmt(fdef.body);
    if (!ok) {
      TokenInfo info = token_info(n.ast.unit, decl.name);
      print_token_location(info, stderr);
      err("return statement not found in the function\n");
      print_token_context(info, stderr);
      n.invalid = true;
      return n;
    }
  }
  return n;
}

bool analyze_return_stmt(Stmt s) {
  PRINT_ANALYZE_FUNC;
  if (s.type == STMT_EXPR)
    return false;
```

```
  if (s.type == STMT_RETURN)
    return true;
  if (s.type == STMT_WHILE)
    return false;
  if (s.type == STMT_LIST) {
    StmtList *sl = s.ptr;
    for (size_t i = 0; i < sl->num; i++) {
      if (sl->at[i].type == STMT_RETURN)
        return true;
    }
  }
  return false;
}

int find_decl(AST ast, Token token) {
  PRINT_ANALYZE_FUNC;
  for (int i = 0; i < ast.decls_num; i++) {
    if (vequ(ast.decls[i].name.text, token.text))
      return i;
  }
  return -1;
}

int find_token(TokenList list, Token token) {
  PRINT_ANALYZE_FUNC;
  for (int i = 0; i < list.num; i++) {
    if (vequ(list.at[i].text, token.text))
      return i;
  }
  return -1;
}

bool analyze_expr_binary(ExprCtx ctx,
                         ExprBinary *eb, bool lvalue) {
  PRINT_ANALYZE_FUNC;
  bool ok = analyze_expr(ctx, eb->right);
```

```
  if (!ok)
    return false;
  if (lvalue) {
    return analyze_expr_left(ctx, eb->left);
  }
  else {
    return analyze_expr(ctx, eb->left);
  }
}

bool analyze_expr_left(ExprCtx ctx, Expr expr) {
  PRINT_ANALYZE_FUNC;
  if (expr.type == EXPR_IDENT) {
    ExprLeaf *leaf = expr.ptr;
    int i = find_token(ctx.fdef.vars, leaf->token[0]);
    if (i >= 0)
      return true;
    int j = find_token(ctx.decl.params, leaf->token[0]);
    if (j >= 0)
      return true;
    TokenInfo info = token_info(ctx.ast.unit,
                                leaf->token[0]);
    print_token_location(info, stderr);
    err("expression is immutable\n");
    print_token_context(info, stderr);
    return false;
  }
  err("expression is immutable\n");
  // TODO: print context
  return false;
}

bool analyze_stmt(ExprCtx ctx, Stmt stmt) {
  PRINT_ANALYZE_FUNC;
  if (stmt.type == STMT_EXPR) {
    StmtExpr *s = stmt.ptr;
```

```
    return analyze_expr(ctx, s->expr);
  }
  if (stmt.type == STMT_RETURN) {
    StmtExpr *s = stmt.ptr;
    return analyze_expr(ctx, s->expr);
  }
  if (stmt.type == STMT_LIST) {
    StmtList *s = stmt.ptr;
    for (size_t i = 0; i < s->num; i++) {
      bool ok = analyze_stmt(ctx, s->at[i]);
      if (!ok)
        return false;
    }
    return true;
  }
  if (stmt.type == STMT_WHILE) {
    StmtWhile *s = stmt.ptr;
    bool ok = analyze_expr(ctx, s->cond);
    if (!ok)
      return false;
    return analyze_stmt(ctx, s->body);
  }
}

NiceAST analyze_expresions(NiceAST n) {
  PRINT_ANALYZE_FUNC;
  if (n.invalid)
    return n;
  for (size_t i = 0; i < n.ast.fdefs_num; i++) {
    ExprCtx ctx = {0};
    ctx.ast = n.ast;
    ctx.fdef = n.ast.fdefs[i];
    ctx.decl = n.ast.decls[ctx.fdef.decl_index];
    bool ok = analyze_stmt(ctx, ctx.fdef.body);
    if (!ok) {
      n.invalid = true;
```

```
      return n;
    }
  }
}

NiceAST analyze(AST ast) {
  PRINT_ANALYZE_FUNC;
  if (ast.invalid)
    return (NiceAST) { .invalid = true };
  NiceAST n = {0};
  n.ast = ast;
  n = analyze_decls(n);
  n = analyze_fdefs(n);
  n = analyze_return(n);
  n = analyze_expresions(n);
  return n;
}

bool analyze_expr(ExprCtx ctx, Expr expr) {
  PRINT_ANALYZE_FUNC;
  switch (expr.type) {
    case EXPR_ADD:
      return analyze_expr_binary(ctx, expr.ptr, false);
    case EXPR_ASSIGN:
      return analyze_expr_binary(ctx, expr.ptr, true);
    case EXPR_CALL:
      return analyze_expr_call(ctx, expr.ptr);
    case EXPR_IDENT:
      return analyze_expr_ident(ctx, expr.ptr);
    case EXPR_NUMBER:
      return true;
    case EXPR_PAREN:
      return analyze_expr(ctx,
                          ((ExprParen *)expr.ptr)->expr);
    case EXPR_STRING:
      return true;
```

```
    case EXPR_SUB:
      return analyze_expr_binary(ctx, expr.ptr, false);
    default:
      return false; // TODO: error message
  }
}

bool analyze_expr_func(ExprCtx ctx, Expr expr) {
  PRINT_ANALYZE_FUNC;
  if (expr.type == EXPR_IDENT) {
    ExprLeaf *leaf = expr.ptr;
    int i = find_token(ctx.fdef.vars, leaf->token[0]);
    int j = find_token(ctx.decl.params, leaf->token[0]);
    int k = find_decl(ctx.ast, leaf->token[0]);
    if (i < 0 && j < 0 && k >= 0)
      return true;
    TokenInfo info = token_info(ctx.ast.unit,
                                leaf->token[0]);
    print_token_location(info, stderr);
    err("expression is not a function\n");
    print_token_context(info, stderr);
    return false;
  }
  err("expression is not a function\n");
  // TODO: print context
  return false;
}

bool analyze_expr_ident(ExprCtx ctx, ExprLeaf *leaf) {
  PRINT_ANALYZE_FUNC;
  int i = find_token(ctx.fdef.vars, leaf->token[0]);
  if (i >= 0)
    return true;
  int j = find_token(ctx.decl.params, leaf->token[0]);
  if (j >= 0)
    return true;
```

```
  TokenInfo info = token_info(ctx.ast.unit, leaf->token[0]);
  print_token_location(info, stderr);
  err("expression is not a variable\n");
  print_token_context(info, stderr);
  return false;
}

bool analyze_expr_call(ExprCtx ctx, ExprCall *call) {
  PRINT_ANALYZE_FUNC;
  bool ok = analyze_expr_func(ctx, call->func);
  if (!ok)
    return false;
  for (size_t i = 0; i < call->args_num; i++) {
    ok = analyze_expr(ctx, call->args[i]);
    if (!ok)
      return false;
  }
  return true;
}

TokenList tl_empty() {
  TokenList tl = {0};
  tl.invalid = false;
  tl.cap = 2;
  tl.num = 0;
  tl.at = malloc(tl.cap * sizeof(Token));
  if (!tl.at) {
    tl.invalid = true;
    return tl;
  }
  return tl;
}

TokenList tl_add(TokenList list, Token token) {
  if (list.invalid)
    return list;
```

```
  if (list.cap == list.num) {
    list.cap *= 2;
    list.at = realloc(list.at, list.cap * sizeof(Token));
    if (!list.at) {
      list.invalid = true;
      return list;
    }
  }
  list.at[list.num] = token;
  list.num++;
  return list;
}
```