МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА КОМПІЛЯТОРА МОВИ ANSI C З ОРІЄНТАЦІЄЮ НА ВИКОРИСТАННЯ НА ПРОЦЕСОРАХ АРХІТЕКТУРИ X86-64»**

Виконав: студент 2 курсу, групи 8.1213-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
А.С. Пірунов
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Мухін В.В.
(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 – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

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

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 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 мови ANSI C з орієнтацією на використання на процесорах архітектури x86-64

керівник роботи Мухін Віталій Вікторович, к.т.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » березня 2024 року № 385-с

2. Строк подання студентом роботи 25.11.2024 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 програмного забезпечення.
4. Тестування програмного забезпечення.

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання 07.03.2024 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.	15.03.2024	
2.	Збір вихідних даних.	03.05.2024	
3.	Обробка методичних та теоретичних	21.06.2024	
	джерел.		
4.	Розробка першого та другого розділу.	30.08.2024	
5.	Розробка третього розділу.	08.11.2024	
6.	Оформлення та нормоконтроль	20.11.2024	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	13.12.2024	

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

Керівник роботи _____ В.В. Мухін
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка компілятора мови ANSI C з орієнтацією на використання на процесорах архітектури x86-64»: 80 с., 39 рис., 10 табл., 17 джерел, 2 додатки.

АРХІТЕКТУРА X86-64, КОМПІЛЯТОР, ЛЕКСИЧНИЙ АНАЛІЗ, МАШИННИЙ КОД, РЕЛОКАЦІЯ СИМВОЛІВ, СЕМАНТИЧНИЙ АНАЛІЗ, СИНТАКСИЧНИЙ АНАЛІЗ, ELF ФАЙЛ.

Об’єкт дослідження – використання мови програмування ANSI C.

Мета роботи: розробити програмне забезпечення що буде автоматично перевіряти написані мовою ANSI C програми на наявність помилок та допомагати програмісту аналізувату роботу цих програм.

Метод дослідження – емпіричний, аналітичний, експеремент.

За час існування мови програмування C цією мовою було написано мільйони рядків коду що потребує подальшої підтримки. Для підримки цього коду потрібне програмне забезпечення що автоматизує перевірки коду на помилки, включаючи помилки під час роботи скомпільованої програми.

У цій кваліфікаційній роботі було розроблено новий компілятор ANSI C для архітектури процесорів x86-64 що працює на операційній системі Linux. Розроблений компілятор перевіряє програми що написані мовою ANSI C на різноманітні помилки. У розроблений компілятор було додано декілька розширень мови C, частина з яких націлена на спрощення підтримки вже написаного коду. Розглянуто те як стандарт мови C впливає на структуру компіляторів. Розроблений компілятор протестовано у різних умовах, включаючи деякі виключні ситуації. Отриманий досвіт виявився корисним у розумінні того як влаштована мова програмування ANSI C.

SUMMARY

Master's qualifying paper "Development of ANSI C Language Compiler With Orientation for Use on x86-64 Architecture Processors": 80 pages, 39 figures, 10 tables, 17 references, 2 supplements.

COMPILER, ELF FILE, LEXICAL ANALYSIS, MACHINE CODE, SEMANTIC ANALYSIS, SYMBOL RELOCATION, SYNTAX ANALYSIS, X86-64 ARCHITECTURE.

The object of the study is the usage of C programming language.

The aim of the study is to develop software that will automatically check programs written in ANSI C for errors and help the programmer analyze the runtime of these programs.

The methods of research are empirical, analytical, experiment.

During the existence of C, millions of lines of code have been written in this programming language, the lines of code that need support. A good understanding of ANSI C is required to support the written code. Most of the code is written in ANSI C, the oldest standard of C.

In this qualification work was developed a new ANSI C compiler for the x86-64 processor architecture on the Linux operating system. The developed compiler checks programs written in ANSI C for various errors. Several language extensions are added to the compiler, where some of them simplify the process of supporting the code. It was shown how the C language standard affects the structure of C compilers. The developed compiler was tested in different cases, including exceptional ones. The gained experience turned out to be useful in understanding of how ANSI C works.

ЗМІСТ

Завдання на кваліфікаційну роботу ……..……..……2
Реферат 4
Summary 5
Скорочення та умовні познаки 8
Вступ 9
1 Вимоги до компілятора 10
1.1 Основні вимоги до компілятора 10
1.2 Вимоги до виведення помилок 10
1.3 Вимоги до інтерфейсу компілятора 11
2 План розробки компілятора 13
2.1 Генерація пустого ELF файлу 13
2.2 Лексичний аналіз 15
2.3 Генерація коду та даних 16
2.4 Синтаксичний аналіз 16
2.5 Інтерпретація виразів 17
2.6 Подальша розробка 17
3 Розробка компілятора 18
3.1 Інтерфейс для створення ELF файлів 18
3.2 Лексичний аналізатор 20
3.3 Генератор коду та даних 22
3.4 Синтаксичний аналізатор 24
3.5 Інтерпретатор виразів 26
3.6 Розширення функціоналу 27
4 Приклади роботи компілятора 28
4.1 Hello, World! 28
4.2 Послідовність Фібоначчі 31
4.3 Гра Snake 34

4.4 Вивід помилок 36
4.5 Компілятор 37
4.6 Прапор zero-vars 39
4.7 Прапор log-calls 40
4.8 Пакування символів 43
Висновки 45
Перелік посилань 46
Додаток А Джерельний код компілятора 48
Додаток Б Джерельний код гри Snake 76

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

CLI	Command-Line Interface
ELF	Executable and Linkable Format
gcc	GNU компілятор C
GNU	Набір вільного програмного забезпечення
ld	GNU лінкер
SDL	Simple Direct-media Layer
uacc	Unique ANSI C Compiler, розробляємий у цій роботі компілятор

ВСТУП

За час існування мови програмування С цією мовою було написано мільйони рядків коду, коду який прямо зараз використовується іншими мовами програмування та сучасними програмами. Цей код довів свою надійність і корисність, тому доцільно витрачати час на його підримку. Для підримки існуючого коду потрібне програмне забезпечення що буде перевіряти написаний код на помилки автоматично та давати змогу програмісту аналізувати роботу коду.

Таким чином, метою кваліфікаційної роботи є розробка компілятора мови ANSI C з орієнтацією на використання на процесорах архітектури x86-64 що буде автоматично перевіряти програми та генерувати додатковий машиний код для спрощення аналізу роботи програм. Для досягнення даної мети поставлені наступні задачі:

- визначення вимог до компілятора;
- планування та проєктування компілятора;
- розробка компілятора та розширень мови ANSI C;
- аналіз роботи компілятора у різних ситуаціях.

1 ВИМОГИ ДО КОМПІЛЯТОРА

1.1 Основні вимоги до компілятора

Компілятор має транслювати правильно написану мовою ANSI C [1] програму у відповідні машинні інструкції архітектури процесорів x86-64 та зберігати згенеровані інструкції з усією додатковою інформацією у файл формату Executable and Linkable File (ELF) [2] для подальшого лінкування за допомогою стороннього лінкера.

Згенерований в результаті компіляції ELF файл має бути сумісним хоча б з одним стороннім лінкером (на момент закінчення розробки компілятора). Це також включає генерування викликів функцій та прологів функцій відповідно до конвенції викликів x86-64 cdecl [3] для успішного лінкування з кодом що скомпільовано іншими компіляторами.

Компілятор має бути розроблений з використанням мови програмування ANSI C або її підмножини так щоб була можливість скомпілювати розроблений компілятор як через самого себе, так і через будь який сторонній компілятор мови ANSI C.

1.2 Вимоги до виведення помилок

Компілятор повинен видавати повідомлення про помилку у тексті програми якщо деяка помилка знайдена. Формат виводу помилок, відповідно до [4], зображено на рисунку 1.1.

```
sourcefile:lineno:column: message
```

Рисунок 1.1 – Формат виводу помилок

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

1.3 Вимоги до інтерфейсу компілятора

Треба щоб у користувача була можливість задавати назву файлу який містить текст програми. Також треба щоб була можливість задавати назву файлу куди компілятор збереже результат роботи.

Використання компілятора повинно відбуватись через CLI. Шлях до файлу програми може бути вказано у будь якому за номером аргументі командного рядка. Шлях де зберігати результат компіляції має бути вказаний одразу після аргументу командного рядка зі словом **`output`**.

Шлях до файлу програми є обов'язковим для роботи компілятора, в той час як шлях для збереження результату є опціональним і має значення за замовченням **`a.o`**.

На рисунку 1.2 зображено два приклади використання компілятора у CLI для компіляції файлу програми **`program.c`**. У першому випадку результат компіляції буде збережено у ELF файл **`program.o`**, у другому випадку результат буде збережено у файл за замовченням.

```
$ uacc program.c output program.o
$ uacc program.c
```

Рисунок 1.2 – Два приклади виклику компілятора у CLI

На рисунку 1.3 зображено діаграму діяльності що описує як має виглядати процес розробки програмного забезпечення за допомогою компілятора.

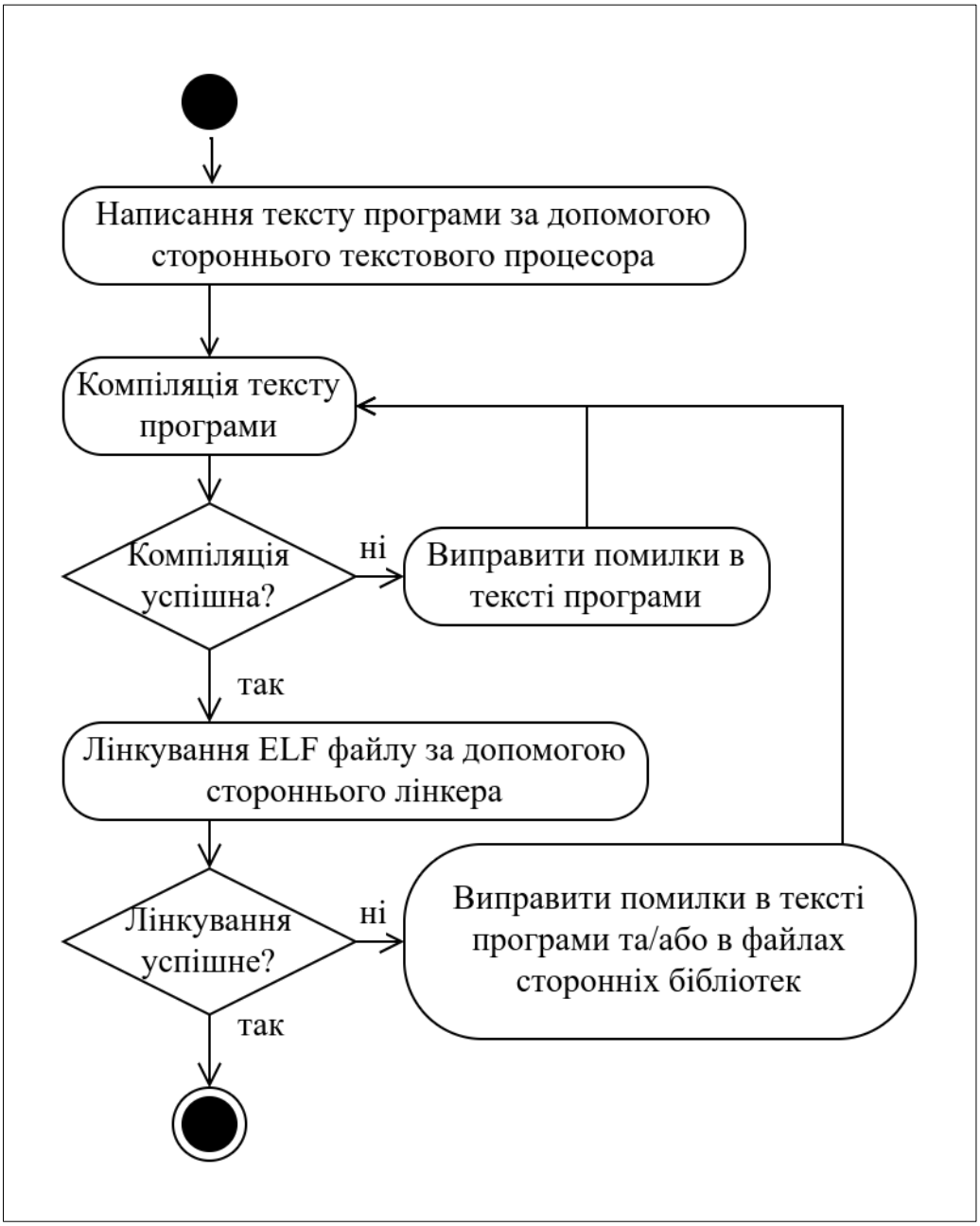


Рисунок 1.3 – Діаграма діяльності розробки ПЗ

2 ПЛАН РОЗРОБКИ КОМПІЛЯТОРА

2.1 Генерація пустого ELF файлу

Я вирішив що має сенс почати розробку компілятора з тієї частини що генерує фінальний ELF файл відповідно до вимог (див. розділ 1). Таким чином можна буде зрозуміти яка саме інформація потрібна для створення такого файлу та врахувати збір цієї інформації у подальшій розробці.

Таблиця 2.1 описує загальну структуру ELF файлу відповідно до [5]. Можна зробити висновок що сегменти не треба створювати під час компіляції. В той час як типи секцій та їх кількість треба знати заздалегідь щоб правильно заповнити заголовок файлу.

Таблиця 2.1 – Загальна структура ELF файлу

Назва	**Значення**
Заголовок файлу	Містить інформацію про весь файл загалом. Кількість сегментів. Кількість секцій.
Заголовки сегментів	Визначає сегменти файлу якщо вони є. Сегменти генеруються на етапі лінкування та потрібні для завантаження ELF файлу в пам’ять для подальшого виконання машинного коду.
Сегменти та секції	Текстові дані, бінарні дані, та інструкції які підпадають під зони впливу сегментів та секцій.
Заголовки секцій	Визначає секції файлу якщо вони є. Секції потрібні в першу чергу під час лінкування.

У таблиції 2.2 перелічені всі поля заголовку ELF файлу. Ця таблиця була сформована відповідно до [6]. Числові значення деяких полів вже визначенні

щоб відповідати архітектурі процесорів x86-64. Значення полів що не беруть участь під час лінкування і тому не потребують визначення під час компіляції написані *курсивом*.

Таблиця 2.2 – Заголовок ELF файлу для архітектури x86-64

Позиція	**Розмір**	**Значення**
0	4	Магічні числа 0x7F, 0x45, 0x4C, 0x46.
4	1	Число 2 для 64 бітного режиму.
5	1	Число 1 для little endian.
6	1	Число 1 тобто перша версія заголовку ELF файлу.
7	1	ABI операційної системи (зазвичай 0 для System V).
8	8	Не використовується.
16	2	Тип файлу. Число 1 у файлів для лінкування.
18	2	Архітектура процесорів. 0x3E для x86-64.
20	4	Число 1 тобто перша версія формату ELF файлу.
24	8	*Адрес початку програми.*
32	8	*Позиція заголовків сегментів.*
40	8	Позиція заголовків секцій.
48	4	Прапори. Число 0 тобто жодних прапорів.
52	2	Розмір заголовку файлу (64 байти).
54	2	*Розмір одного заголовку сегменту.*
56	2	*Кількість сегментів.*
58	2	Розмір одного заголовку секції (64 байти).
60	2	Кількість секцій.
62	2	Індекс секції що містить назви секцій.

Як видно більшість полів або не використовуються лінкером або визначенні заздалегідь. Єдине поле що неможливо визначети заздалегідь це позиція заголовків секцій, так як ця позиція залежить від розміру даних що ідуть до заголовків секцій.

2.2 Лексичний аналіз

Після того як інтерфейс для створення ELF файлів буде розроблено, можна буде почати інтерпретувати текст програм що подаються на вхід компілятору. Інтерпретація тексту починається з ділення цього тексту на лексичні одиниці – токени.

Стандарт мови ANSI C визначає декілкьа типів токенів ділячи їх на препроцесові та опрацьовані [7, с. 19]. Препроцесові токени це токени які зберігаються в джерельних файлах. Опрацьовані токени генеруються під час роботи препроцесору. У таблиції 2.3 наведені типи токенів та їх значення.

Таблиця 2.3 – Типи токенів

Тип	Значення
keyword	Ключове слово (напр. **while**)
identifier	Назва функції, змінної, або типу (напр. **main**)
constant	Числова константа (напр. **4278**)
string-literal	Текстова константа (напр. **"kt2s5"**)
operator	Оператор (напр. **+=**)
punctuator	Розділовий знак (напр. **;**)
header-name	Назва файлу в **#include** (напр. **<stdio.h>**)
pp-number	Препроцесова числова константа (напр. **.278**)
character-constant	Один символ (напр. **'A'**)
other	Будь що інше що не є пробілом (напр. **$@**)

2.3 Генерація коду та даних

Інтерпретація тексту програми у компіляторах завжди супроводжується генерацією машинних інструкцій. Саме тому матиме сенс на даному етапі зусередитись на створенні інтерфейсу що дозволяє генерувати машинні інструкції та дані.

Інтерфейс має дозволяти писати будь-які машинні інструкції та дані, а також давати можливість редагувати вже згенеровані інструкції та дані. Редагування має включати в себе можливість читання та запису цілих чисел різної кількості біт в обхід автоматичного alignment-у [8].

2.4 Синтаксичний аналіз

Коли стане можливим читати токени і генерувати машинний код, буде доцільним перейти до синтаксичного аналізу, так як це дозволить компілятору почати генерувати перші результати компіляції з урахуванням джерельного коду що поданий на вхід компілятору.

Такі елементи синтаксичного дерева як, декларація функції, декларація глобальної змінної, визначення функції, та конструкції **`if`**, **`if-else`**, **`while`**, **`do-while`**, **`for`**, можливо інтерпретувати та генерувати відповідний машиний код одночасно з тим як відбувається їх синтаксичний аналіз. Це можливо бо всі ці елементи мають заздалегідь відому структуру та всі вони починаються з деякого ключового слова за яким можна їх можна відрізнити.

З іншого боку, вирази що складаються з викликів операторів мови ANSI C мають непередбачувану структуру. Операції в таких виразах можуть відбуватись у порядку протележному до порядку їх появи у тексті, що унеможливлює їх інтерпретацію під час синтаксичного аналізу. Тому треба буде зберігати ці вирази у бінарному дереві що відображає правильний порядок операцій для того щоб інтерпретувати ці операції пізніше.

2.5 Інтерпретація виразів

На цьому етапі розробки компілятор вже буде здатен аналізувати різний текст програм та генерувати машинний код для різних конструкцій мови ANSI C. Однак залишається найголовніше – інтерпретація виразів та генерація відповідного машинного коду.

Потрібно буде розробити набір функції для інтерпретації бінарного дерева що відображає порядок операцій в одному виразі. Таким чином буде можливо використовувати ці функції для інтерпретації кожного виразу який компілятор зустріне під час синтаксичного аналізу.

2.6 Подальша розробка

Як тільки компілятор зможе успішно компілювати перші програми, можна буде реалізувати частини мови ANSI C які були опущені в плані до цього моменту. А саме реалізувати препроцесор мови ANSI C.

Додати та покращувати семантичний аналіз для перевірки джерельного коду на різноманітні помилки які виходять за межі синтаксичний помилок.

Розробити формат виклику компілятора у CLI. Також у цей момент можна буде додати у компілятор декілька розширень мови ANSI C.

3 РОЗРОБКА КОМПІЛЯТОРА

3.1 Інтерфейс для створення ELF файлів

Згідно з планом (див. підрозділ 2.1) було розроблено набір функцій що надають інтерфейс для створення та налаштування ELF файлів. Розроблений інтерфейс включає в себе функції для декларації та визначення секцій ELF файлу, додавання тексту у секцію **`.strtab`**, декларацію та визначення символів у секції **`.symtab`**, а також функції для створення релокацій. У таблиці 3.1 перелічені розроблені функції з описом їх призначення.

Таблиця 3.1 – Функції для створення ELF файлу

Назва функції	Призначення
`nstring`	Додати текст заданого розміру у секцію **`.strtab`**.
`string`	Додати нуль-терменований рядок у секцію **`.strtab`**.
`findsymbol`	Знайти символ починаючи з кінця секції **`.symtab`**.
`declare3`	Додавання будь-якого символу у секцію **`.symtab`**.
`declare`	Додавання глобального символу у секцію **`.symtab`**.
`define`	Визначення значення і розміру символу.
`relocate`	Створення будь-якої релокації у секції **`.rela.text`**.
`relocate_plt32`	Створення PLT32 релокації у секції **`.rela.text`**.
`relocate_pc32`	Створення PC32 релокації у секції **`.rela.text`**.
`declsect`	Декларація секцій ELF файлу.
`linksect`	Визначення полів link та info у секцій ELF файлу.
`defnsect`	Визначення позиції та розміру секції.
`writesect`	Запис даних секції у ELF файл.

Використовуючи розроблений інтерфейс було створено «пустий» ELF файл. У створеному файлі визначено наступні секції (таблиця 3.2). Цей набір секцій є однаковим при кожному запуску розробляємого компілятора.

Таблиця 3.2 – Секції ELF файлу

Назва секції	**Призначення**
Відсутня	Перша секція в ELF файлі повинна бути пустою відповідно до [9].
.shstrtab	Містить назви секцій.
.strtab	Містить назви символів та інший текст.
.note.GNU-stack	Потрібна для сумісності з лінкером ld.
.symtab	Таблиця символів.
.rela.text	Містить релокації для секції **.text**.
.text	Зберігає машинні інструкції.
.data	Зберігає ініціалізовані дані.
.bss	Зберігає не ініціалізовані дані.

У створеному файлі було також визначено декілька локальних символів що допомагають стороннім утилітам таким як objdump виводити корисну інформацію про файл. Визначенні символи перелічені у таблиці 3.3.

Таблиця 3.3 – Секції ELF файлу

Назва символу	**Призначення**
Відсутня	Перший символ має бути пустим відповідно до [10].
file.c	Назва символу є назвою джерельного файлу.
.text	Символи що вказують на початок секцій відповідної назви.
.data	
.bss	

3.2 Лексичний аналізатор

Відповідно до плану (див. підрозділ 2.2) було розроблено функцію **`read_token`**, що читає один препроцесовий токен з тексту програми. Ця функція дозволяє читати токени один за одиним до поки функція не поверне пустий токен що позначає кінець тексту.

Функція **`read_token`** також повертає спеціальний токен в кінці кожного логічного рядку тексту. Це у майбутньому дозволить препроцесору інтерпретувати логічні рядки тексту.

Дані про токен зберігається функцією **`new_token`** у 32-бітне число. Маючи ці дані можна отримати текст токена для подальшої інтерпретації. Усі дані що зберігаються перелічені у таблиці 3.4.

Таблиця 3.4 – Дані про токен

Назва	Кількість біт	Значення
`off`	17	Позиція токену в тексті. Розмір тексту що містить токен має бути не більше 128 кілобайт.
`len`	9	Довжина токена. Максимум 511 символів.
`src`	5	Індекс джерельного файлу з якого був прочитаний токен. Максимум 32 різних файлів в один момент.
Немає	1	Завжди 0. Зарезервовано на майбутнє.

Розроблена функція **`error`** що приймає токен та текст помилки. Функція виводить назву джерельного файлу, номер рядока, та текст токена у якості контексту помилки. Потім виводить текст помилки та пририває роботу компілятора.

Було створено функцію **`pull_token`** що повністю абстрагує роботу препроцесору, дозволяючи читати опрацьовані токени один за одним. А саме, якщо буфер логічного рядка тексту пустий, ця функція читає препроцесові

токени за допомогою **read_token** та зберігає їх у буфер логічного рядка тексту. Як тільки буфер був заповнений токенами, препроцесор інтерпретує вміст логічного рядка. В результаті роботи препроцесора буфер містить опрацьовані токени які функція **pull_token** може послідовно повертати. Як тільки опрацьовані токени в буфері закінчуються вся процедура повторюється.

На даному етапі розробки жодна директива препроцесора не реалізована, тому препроцесор просто конвертує усі препроцесові токени в опрацьовані токени.

Для перевірки написаного інтерфейсу у функцію **main** розробляємого компілятора був доданий тимчасовий код (рис. 3.1). Доданий код читає всі токени з джерельного файлу та виводить кожен токен на окремому рядку.

```
pull_token();
while (tokequ(_token, "") == 0) {
  printf(" %.*s\x0A", toklen(_token), tokstr(_token));
  pull_token();
}
```

Рисунок 3.1 – Код що читає всі токени

Для спрощення аналізу тексту токенів було розроблені функції **tokequ**, **toklen**, та **tokstr**. Де **tokequ** порівнює текст токена з будь яким іншим текстом, **toklen** повертає довжину токена, а **tokstr** повертає вказівник на текст токену.

На вхід компілятору було подано джерельний файл (рис. 3.2). Цей файл містить декілька токенів. І як видно на скриншоті CLI (рис. 3.3), компілятор успішно прочитав усі токени.

```
/* toks.c - Some tokens */
int main() {
  printf("Hello, World!\x0A");
  return 0;
}
```

Рисунок 3.2 – Код що читає всі токени

```
$ ./uacc toks.c
 int
 main
 (
 )
 {
 printf
 (
 "Hello, World!\x0A"
 )
 ;
 return
 0
 ;
 }
```

Рисунок 3.3 – Читання токенів

3.3 Генератор коду та даних

Згідно з планом (див. підрозділ 2.3) було розроблено функції для додавання машинних інструкцій та даних у секції **`.text`** та **`.data`** відповідно. Також були розроблені функції що дозволяють читати та записувати 16, 32, та 64 біт числа у little-endian [11] формі.

Була розроблена функція **`emit`** яка додає в кінець секції **`.text`** зазначену в аргументах функції кількість байт та ініціалізує додані байти значеннями з масиву що також передано як аргумент функції.

Розроблена функція **`emit_data`** яка працює аналогічно функції **`emit`**, за тим виключенням що функція **`emit_data`** додає байти саме в секцію **`.data`**. Послідовні виклики обох функцій дозволяють визначити вміст секцій **`.text`** та **`.data`**.

Були визначені функції **`get8`**, **`get16`**, **`get32`**, та **`get64`** що читають 8, 16, 32, та 64 біт числа за переданим у функцію вказівником відповідно. Ці функції дозволяють уникнути автоматичного alignment-у.

Були визначені функції **`set8`**, **`set16`**, **`set32`**, та **`set64`** що записують 8, 16, 32, та 64 біт числа за переданим у функцію вказівником відповідно. Ці

функції теж дозволяють уникнути автоматичного alignment-у. У функції **set64** також є варіанти **set64i** та **set64u** що записують 32 бітне число як 64 бітне додаючи біти враховуючи та не враховуючи знак відповідно.

Щоб перевірити розроблений інтерфейс у функцію **main** розробляємого компілятора був доданий наступний код (рис. 3.4). Цей код записує інструкції у секцію **.text**, записує дані у секцію **.data**, визначає сиволи та створює відповідні релокації. Усі константи в наведеному коді були заздалегідь розраховані щоб в результаті отримати програму «Hello, World!».

```
mainsym = declare(string("main"), 2);
putssym = declare(string("puts"), 2);
define(mainsym, SHN_TEXT, 0, 0);
emit(11, "\x48\x83\xEC\x08\x48\x8D\x3D\x00\x00\x00\x00");
emit(12, "\xE8\x00\x00\x00\x00\x31\xC0\x48\x83\xC4\x08\xC3");
emit_data(14, "Hello, World!");
relocate_pc32(7, SYM_DATA, -4);
relocate_plt32(12, putssym, -4);
```

Рисунок 3.4 – Генерація коду та даних

Згенерований таким чином ELF файл був залінкований використовуючи сторонній лінкер ld. Отримана в результаті лінкування програма надрукувала повідомленя (рис. 3.5) як заплановано.

```
$ gcc a.o -o a
$ ./a
Hello, World!
```

Рисунок 3.5 – Запуск предвизначеної програми

У функцію **relocate_plt32** було додано код що зменшує загальну кількість релокацій у ELF файлі. Доданий код фільтрує запити на створення релокацій за їх символами. Якщо на момент виклику функції отриманий символ вже визначений у секції **.symtab** то замість того щоб створити нову релокацію, запит на релокацію буде задовільнений одразу. Повний код функції можна побачити на скриншоті (рис. 3.6).

```
int relocate_plt32(int offset, int symi, int addend) {
  int diff;
  char *sym;
  sym = symi * 24 + _symtab;
  if (get16(sym + 6) != 0) {
    diff = get32(sym + 8) - offset;
    set32(_text + offset, diff + addend);
    return 0;
  }
  return relocate(offset, symi, 4, addend);
}
```

Рисунок 3.6 – Запуск предвизначеної програми

3.4 Синтаксичний аналізатор

Було розроблено функції **program**, **declaration**, **global**, **function**, та **statement**, що виконують синтаксичний аналіз та одразу інтерпретують конструкції мови ANSI C додаючи відповідний код у секцію **.text**. У таблиці 3.5 наведені назви функцій та їх призначення.

Таблиця 3.5 – Функції синтаксичного аналізатору

Назва	Призначення
program	Інтерпретує послідовність декларацій за допомогою функції **declaration**, поки не буде знайдено кінець джерельного файлу.
declaration	Інтерпретує одну декларацію. Визначивши тип даних, клас зберігання, та назву об’єкту, передає ці дані у функцію **global** або **function** в залежності від типу декларації.
global	Інтерпретує декларацію глобальної змінної або **typedef**.
function	Інтерпретує декларацію та визначення функції.
statement	Інтерпретує конструкції **if**, **if-else**, **do-while**, **while**, та **for**.

Інтерпретація декларацій функціями **global** та **function** супроводжується додаванням відповідних символів у таблицю символів, тобто у секцію **.symtab**. У випадках коли позиція та/або розмір символу відомі під час декларації ці значення також буде записано у таблицю символів, міняючи тип символу з невизначеного на визначений.

Функція **statement** викликає функцію **expression** у всіх випадках де очікується вираз (напр. умова **if** це вираз). На даний момент функція **expression** тільки проводить синтаксичний аналіз виразу та будує бінарне дерево що відображає правильний порядок операцій, але не генерує відповідний машинний код.

Синтаксичний аналіз виразів та побудова відповідного бінарного дерева відбувається шляхом рекурсивоного виклику спеціальних функцій (таб. 3.6). Такий метод забезпечує правильний порядок операцій.

Таблиця 3.6 – Функції для синтаксичного аналізу виразів

Назва	Призначення
x_assign	Синтаксичний аналіз оператора **=**.
x_opassign	Синтаксичний аналіз операторів **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=**, та **>>=**.
x_math	Синтаксичний аналіз бінарних операторів **+**, **-**, *****, **/**, **%**, **&**, **|**, **^**, **==**, **!=**, **<<**, **>>**, **>=**, **<=**, **>**, **<**, **&&**, **||**, а також тернарного оператора **?:**.
x_prefix	Синтаксичний аналіз префіксних операторів **+**, **-**, **++**, **--**, *****, **&**, **~**, **sizeof**, та зміна типу даних через **(type)**.
x_suffix	Синтаксичний аналіз операторів індексації масиву **[...]** та виклику функції **(...)**.
x_args	Синтаксичний аналіз списку аргументів функції.
x_primary	Синтаксичний аналіз змінних, констант, та підвиразів.

3.5 Інтерпретатор виразів

Для інтерпретації виразів та генерації відповідного машинного коду було розроблено функції що аналогічні за назвами функціям синтаксичного аналізу виразів та відповідають за інтерпретацію відповідних елементів згенерованого бінарного дерева. Усі функції наведені у таблиці 3.7.

Таблиця 3.7 – Функції для компіляції виразів

Назва	Призначення
`c_assign`	Компіляція будь-яких операцій збереження значення у пам’ять.
`c_lvident`	Генерація коду що знаходить адресу символу.
`c_math`	Компіляція бінарних операторів, а також тернарного оператора.
`c_prefix`	Компіляція префіксних операторів.
`c_suffix`	Компіляція індексації масивів (або вказівників) та викликів функцій. Виклики функцій відбуваються відповідно до конвенції викликів x86-64.
`c_primary`	Знаходження значень констант, знаходження адрес текстових констант. Компіляція підвиразів.
`c_ident`	Генерація коду що знаходить значення символу.

Згенерований цими функціями машинний код використовує регістр **`%rax`** для акумуляції результату виразу, та використовує регістр **`%rcx`** для збереження операнда поточної операції. В більшості випадків цих двох регістрів достатньо для збереження значень під час розрахування виразу. У всіх випадках де двох регістрів не достатньо – використовується стек.

Код функції **`expression`** був доповнений щоб не тільки синтаксично аналізувати вирази, але і генерувати відповідний машинний код.

3.6 Розширення функціоналу

Користуючись реалізованим у функції **`pull_token`** буфером токенів було додано підтримку директиви препроцесора **`#include`**, що змінює джерельний файл з якого компілятор читає токени. Як тільки файл що було додано через **`#include`** закінчується то функція **`pull_token`** повертається до читання минулого файлу.

Було додано перевірки типів даних під час інтерпретації виразів. Також було додано перевірки назв змінних та функцій на те чи були вони задекларовані перед використанням.

Відповідно до вимог (див. розділ 1) було реалізована можливість задавати назву джерельного файлу на вхід компілятору та задавати шлях по якому буде збережений результат компіляції.

Розроблено три розширення мови ANSI C: пакування символів, автоматична ініціалізація локальних змінних, та друк історії викликів функцій під час роботи скомпільованого коду. Детальніше ці розришення будуть описані та продемонстровані у розділі 4.

Повний код розробленого компілятора доступний у додатку А.

4 ПРИКЛАДИ РОБОТИ КОМПІЛЯТОРА

4.1 Hello, World!

Програма «Hello, World!» (рис. 4.1) є класичною перевіркою того що компілятор працює правильно. Зазвичай така програма складається з виклику функції **printf** для виведення текстового повідомлення одразу після якого програма закінчує роботу.

```
/* hello.c - Hello, World! program in ANSI C */
int printf(const char *format, ...);

int main() {
  printf("Hello, World!\x0A");
  return 0;
}
```

Рисунок 4.1 – Текст програми Hello, World!

Також є варіант цієї програми (рис. 4.2) де функція **printf** задекларована в окремому файлі який за допомогою директиви препроцесора **#include** додається у фінальний текст програми автоматично.

```
/* hello2.c - Hello, World! program in ANSI C */
#include "printf.h"

int main() {
  printf("Hello, World!\x0A");
  return 0;
}
```

Рисунок 4.2 – Текст програми Hello, World! (варіант з `#include`)

На скриншоті (рис. 4.3) зображено команди для компіляції, лінкування, та запуску обох варіантів програми. Для лінкування був використаний сторонній лінкер, а саме ld [12] викликаний за допомогою gcc [13] що автоматично кофігурує лінкер для мови C на архітектурі x86-64.

```
$ ./uacc hello.c output hello.o
$ gcc --output hello hello.o && ./hello
Hello, World!

$ ./uacc hello2.c output hello2.o
$ gcc --output hello2 hello2.o && ./hello2
Hello, World!
```

Рисунок 4.3 – Скриншот компіляції, лінкування, та запуску програм в CLI

Результат компіляції обох варіантів має ідентичні машинні інструкції та видає однакове текстове повідомлення, однак ELF файли відрізнаються назвою джерельного файлу що збережено у таблиці символів (рис. 4.4). Для витягу таблиці символів було застосовано утиліту objdump [14].

```
$ objdump -t hello.o
SYMBOL TABLE:
0000000000000000 l  df *ABS* 0000000000000000 hello.c
0000000000000000 l  d  .text 0000000000000000 .text
0000000000000000 l  d  .data 0000000000000000 .data
0000000000000000 l  d  .bss  0000000000000000 .bss
0000000000000000     F *UND* 0000000000000000 printf
0000000000000000 g   F .text 0000000000000030 main

$ objdump -t hello2.o
SYMBOL TABLE:
0000000000000000 l  df *ABS* 0000000000000000 hello2.c
0000000000000000 l  d  .text 0000000000000000 .text
0000000000000000 l  d  .data 0000000000000000 .data
0000000000000000 l  d  .bss  0000000000000000 .bss
0000000000000000     F *UND* 0000000000000000 printf
0000000000000000 g   F .text 0000000000000030 main
```

Рисунок 4.4 – Скриншот таблиці символів обох ELF файлів

За допомогою утиліти objdump було задизасемблено ELF файл результату компіляції через компілятор uacc (рис. 4.5) та порівняно з відповідним результатом компіляції через компілятор gcc з нульовим прапором оптимізації (рис. 4.6).

```
$ objdump -Sr hello.o
0000000000000000 <main>:
   0:     55                          push   %rbp
   1:     48 89 e5                    mov    %rsp,%rbp
   4:     48 81 ec 00 00 00 00        sub    $0x0,%rsp
   b:     50                          push   %rax
   c:     50                          push   %rax
   d:     48 8d 0d 00 00 00 00        lea    0x0(%rip),%rcx
                  10: R_X86_64_PC32   .data-0x4
  14:     48 89 c8                    mov    %rcx,%rax
  17:     50                          push   %rax
  18:     5f                          pop    %rdi
  19:     e8 00 00 00 00              call   1e <main+0x1e>
                  1a: R_X86_64_PLT32  printf-0x4
  1e:     48 89 c1                    mov    %rax,%rcx
  21:     58                          pop    %rax
  22:     58                          pop    %rax
  23:     48 89 c8                    mov    %rcx,%rax
  26:     b9 00 00 00 00              mov    $0x0,%ecx
  2b:     48 89 c8                    mov    %rcx,%rax
  2e:     c9                          leave
  2f:     c3                          ret
```

Рисунок 4.5 – Інструкції та релокації що згенеровані компілятором uacc

```
$ objdump -Sr hello-gcc.o
0000000000000000 <main>:
   0:     55                          push   %rbp
   1:     48 89 e5                    mov    %rsp,%rbp
   4:     48 8d 05 00 00 00 00        lea    0x0(%rip),%rax
                  7: R_X86_64_PC32    .rodata-0x4
   b:     48 89 c7                    mov    %rax,%rdi
   e:     e8 00 00 00 00              call   13 <main+0x13>
                  f: R_X86_64_PLT32   puts-0x4
  13:     b8 00 00 00 00              mov    $0x0,%eax
  18:     5d                          pop    %rbp
  19:     c3                          ret
```

Рисунок 4.6 – Інструкції та релокації що згенеровані компілятором gcc

Можна побачити що gcc навіть з нульовим прапором оптимізацій підбирає інструкції більш розумно ніж компілятор розроблений у цій роботі. Наприклад перед викликом функції **`printf`** компілятор uacc зберігає значення регістру **`%rax`** на вершині стеку не дивлячись на те що це значення не буде

використано в подальших інструкціях. В той час як компілятор gcc генерує тільки ті інструкції які дійсно необхідні.

Також компілятор gcc замінив виклик функції **printf** на менш ресурсоємний виклик функції **puts**, користуючись знаннями про функції стандартної бібліотеки у мові програмування ANSI C. В той час як компілятор uacc не розрізняє які функції є функціями стандартної бібліотеки, а які є сторонніми функціями з не відомою поведінкою.

4.2 Послідовність Фібоначчі

Програма що виводить послідовність Фібоначчі є класичною задачею у програмуванні. Перші два числа це одиниці, а кожне наступне це сума двох які ідуть до цього. Текст програми (рис. 4.7) містить функцію **fibonacci** що є рішенням задачі, ця функція викликається із функції **main**.

```
/* fibonacci.c - Fibonacci sequence */
int printf(const char *format, ...);

void fibonacci(int N) {
  int a;
  int b;
  int t;
  int i;
  b = 0;
  i = a = 1;
  while (i <= N) {
    printf("%02i) %i\x0A", i, a);
    t = b + a;
    b = a;
    a = t;
    ++i;
  }
}

int main() {
  fibonacci(14);
  return 0;
}
```

Рисунок 4.7 – Текст програми що рахує послідовність Фібоначчі

На скриншоті (рис. 4.8) можна побачити команди компіляції, лінкування, та запуску програми. Програма успішно виводить перші 14 чисел з послідовності Фібоначчі.

```
$ ./uacc fibonacci.c output fibonacci.o
$ gcc -o ./fibonacci ./fibonacci.o && ./fibonacci
01) 1
02) 1
03) 2
04) 3
05) 5
06) 8
07) 13
08) 21
09) 34
10) 55
11) 89
12) 144
13) 233
14) 377
```

Рисунок 4.8 – Компіляція, лінкування, та запуск програми

За допомогою утиліти objdump можна подивитись згенеровані інструкції (рис. 4.9 та 4.10). Не важливі для даної демонстрації частини результату компіляції були видалені та замінені трьокрапками.

```
0000000000000000 <fibonacci>:
   0:     55                          push   %rbp
   1:     48 89 e5                    mov    %rsp,%rbp
   4:     48 81 ec 30 00 00 00        sub    $0x30,%rsp
...
  34:     48 8b 8d d8 ff ff ff        mov    -0x28(%rbp),%rcx
  3b:     48 89 c8                    mov    %rcx,%rax
  3e:     48 8b 8d f8 ff ff ff        mov    -0x8(%rbp),%rcx
  45:     48 39 c8                    cmp    %rcx,%rax
  48:     48 c7 c0 01 00 00 00        mov    $0x1,%rax
  4f:     7e 03                       jle    54
  51:     48 31 c0                    xor    %rax,%rax
  54:     48 85 c0                    test   %rax,%rax
  57:     0f 84 8e 00 00 00           je     eb
...
```

Рисунок 4.9 – Інструкції що згенеровані компілятором uacc (1)

```
...
  90:     48 8b 8d e8 ff ff ff     mov    -0x18(%rbp),%rcx
  97:     48 89 c8                 mov    %rcx,%rax
  9a:     48 8b 8d f0 ff ff ff     mov    -0x10(%rbp),%rcx
  a1:     48 01 c8                 add    %rcx,%rax
  a4:     48 89 85 e0 ff ff ff     mov    %rax,-0x20(%rbp)
  ab:     48 8b 8d f0 ff ff ff     mov    -0x10(%rbp),%rcx
  b2:     48 89 c8                 mov    %rcx,%rax
  b5:     48 89 85 e8 ff ff ff     mov    %rax,-0x18(%rbp)
  bc:     48 8b 8d e0 ff ff ff     mov    -0x20(%rbp),%rcx
  c3:     48 89 c8                 mov    %rcx,%rax
  c6:     48 89 85 f0 ff ff ff     mov    %rax,-0x10(%rbp)
  cd:     48 8b 8d d8 ff ff ff     mov    -0x28(%rbp),%rcx
  d4:     48 89 c8                 mov    %rcx,%rax
  d7:     b9 01 00 00 00           mov    $0x1,%ecx
  dc:     48 01 c8                 add    %rcx,%rax
  df:     48 89 85 d8 ff ff ff     mov    %rax,-0x28(%rbp)
  e6:     e9 49 ff ff ff           jmp    34
...
```

Рисунок 4.10 – Інструкції що згенеровані компілятором uacc (2)

Видно що пролог функції **`fibonacci`** виділяє пам’ять під локальні змінні, а також впевнюється щоб адрес вершини стеку був кратний 16. Інструкції що порівнюють змінні **`i`** та **`N`** не оптимальні, бо спочатку зберігають результат порівняння як 0 або 1. Інструкції що згенерував компілятор gcc (рис. 4.11) доводять що цього кроку можна було уникнути.

```
0000000000000000 <fibonacci>:
...
  1f:     eb 37                    jmp    58
...
  3d:     8b 55 f8                 mov    -0x8(%rbp),%edx
  40:     8b 45 fc                 mov    -0x4(%rbp),%eax
  43:     01 d0                    add    %edx,%eax
  45:     89 45 f0                 mov    %eax,-0x10(%rbp)
  48:     8b 45 fc                 mov    -0x4(%rbp),%eax
  4b:     89 45 f8                 mov    %eax,-0x8(%rbp)
  4e:     8b 45 f0                 mov    -0x10(%rbp),%eax
  51:     89 45 fc                 mov    %eax,-0x4(%rbp)
  54:     83 45 f4 01              addl   $0x1,-0xc(%rbp)
  58:     8b 45 f4                 mov    -0xc(%rbp),%eax
  5b:     3b 45 ec                 cmp    -0x14(%rbp),%eax
  5e:     7e c1                    jle    21
...
```

Рисунок 4.11 – Інструкції що згенеровані компілятором gcc

Компілятор uacc згенерував 11 інструкцій що проводять розрахунок наступного числа в послідовності Фібоначчі. Як видно в результаті роботи компілятора gcc є можливість виконати тіж самі операції за 8 інструкцій.

4.3 Гра Snake

Щоб продемонструвати що лінкування не обмежується лінкуванням стандартної бібліотеки, для цієї роботи був розроблений клон гри Snake. Snake це класична відеогра що має тисячі реалізацій на різних операційних системах, з використанням різних мов програмування, та бібліотек.

Представлена у цій роботі реалізація (джерельний код у додатку Б) була розроблена з використанням компілятору uacc, лінкера ld, та бібліотеки SDL [15]. На скриншоті (рис. 4.12) можна побачити команди компіляції, лінкування, та запуску гри.

```
$ ./uacc output snake.o snake.c
$ gcc -o snake snake.o -lSDL2
$ ./snake
```

Рисунок 4.12 – Компіляція, лінкування, та запуск гри

На скриншоті (рис. 4.13) зображено початок гри. Верхня ліва клітинка це позиція (0, 0), нижня права це (9, 9). На початку гри голова змії завжди знаходиться на координатах (5, 5), а хвіст змії на клітинках (6, 5) та (7, 5). Їжа знаходиться на випадкових координатах, але ніколи не займає вже займані змією клітинки.

Рух змії на початку гри напрямлений вліво. Рух змії відбувається автоматично, рухаючи голову змії на 1 клітинку кожну секунду, та «підтягуючи» хвіст. Напрямок руху можна міняти клавішами UP, RIGHT, DOWN, LEFT.

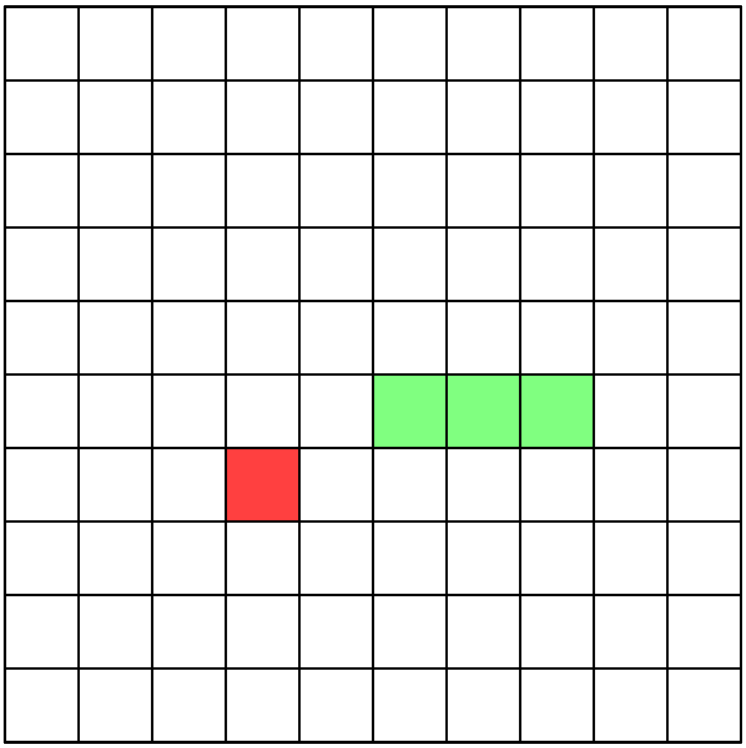


Рисунок 4.13 – Скриншот початку гри

На скриншоті (рис. 4.14) зображено момент після кількох хвилин гри. Кожен раз коли голова змії торкається їжі то довжина змії збільшується на 1 клітинку, а їжа переміщується на інші координати.

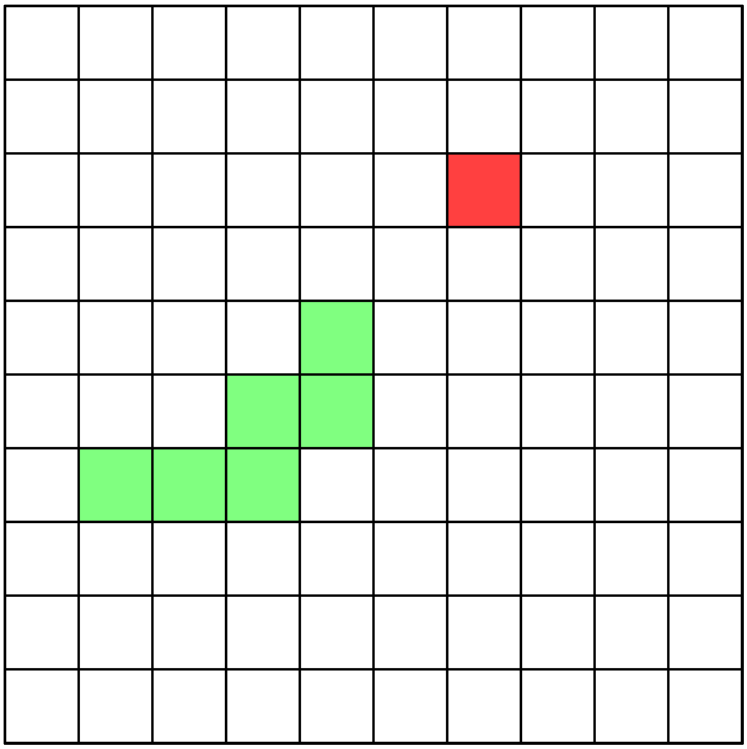


Рисунок 4.14 – Скриншот після кількох хвилин гри

На скриншоті (рис. 4.15) зображено кінець гри. Якщо змія торкається своєю головою свого хвоста то гра закінчується та рух змії припиняється.

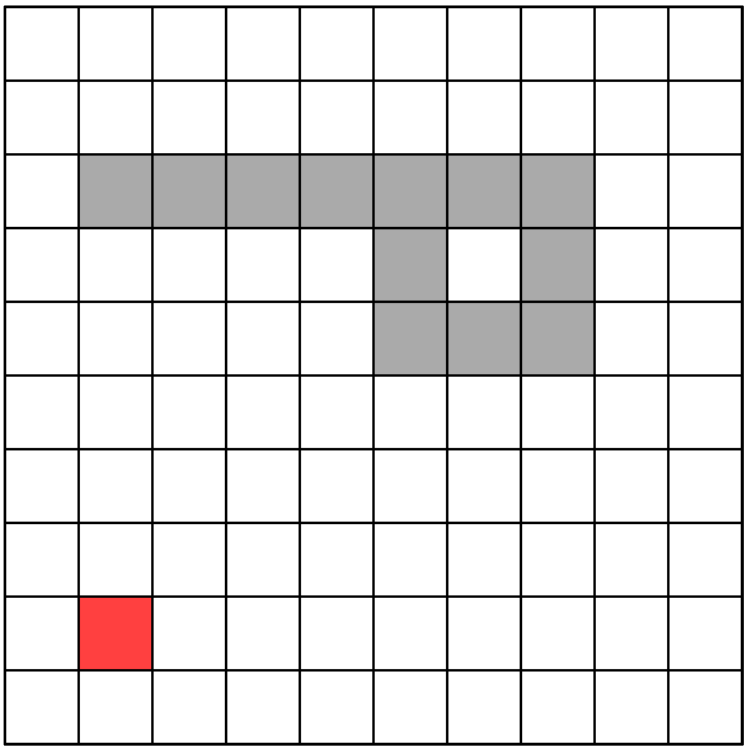


Рисунок 4.15 – Скриншот кінця гри

4.4 Вивід помилок

При використанні будь якого компілятора важливо мати можливість отримати повідомлення про помилку у випадках коли текст програми містить помилки. Розроблений у цій роботі компілятор перевіряє текст програми на різноманітні помилки. Наступна програма (рис. 4.16) використовує змінну x що не була визначена в тексті. У мові програмування ANSI C використання невизначених змінних є помилкою.

```
/* err.c - Test error messages */
int foo() {
  return x * x;
}
```

Рисунок 4.16 – Текст програми з помилкою

На скриншоті (рис. 4.17) можна побачити спробу компіляції вище зазначенної програми. Оскільки текст програми містить помилки компілятор закінчив роботу «не тихо», тобто надрукувавши повідомлення про помилку.

```
$ ./uacc err.c
err.c: 3: x: undefined symbol
```

Рисунок 4.17 – Невдала спроба компіляції програми

У повідомлені вказано назву файлу та рядок на якому виявлена помилка, також там вказано слово **x** що є причиною помилки. Найголовніше у повідомленні описано помилку, а саме «undefined symbol», тобто «невизначений символ». Результат компіляції не було збережено.

4.5 Компілятор

Розроблений у цій роботі компілятор мови програмування ANSI C був написаний мовою ANSI C відповідно до вимог до програмного забезпечення (див. перший розділ). Це дозволяє компілятору компілювати свій власний джерельний код. На скриншоті (рис. 4.18) зображено команди компіляції, лінкування, та запуску компілятора без аргументів.

```
$ ./uacc output uacc.o uacc.c
$ gcc -o uacc2 uacc.o
$ ./uacc2
no input files
```

Рисунок 4.18 – Компіляція компілятора через самого себе

Тепер існують дві версії компілятора. Перша версія uacc скомпільована за допомогою компілятору gcc. Друга версія uacc2 скомпільована за допомогою першої.

Обидві версії компілятору можна порівняти між собою порівнявши те які результати компіляції вони генерують. Для цього був написаний наступний bash скрипт (рис. 4.19) який компілює попередні приклади програм обома версіями та порівнює результати утилітою cmp. Утиліта cmp порівнює два файли та виводить повідомлення якщо вони відрізняються [16].

```
# compare.sh - Compare examples using both versions
# hello.c
./uacc hello.c output hello.o
./uacc2 hello.c output hello2.o
cmp hello.o hello2.o
# fibonacci.c
./uacc fibonacci.c output fibonacci.o
./uacc2 fibonacci.c output fibonacci2.o
cmp fibonacci.o fibonacci2.o
# snake.c
./uacc snake.c output snake.o
./uacc2 snake.c output snake2.o
cmp snake.o snake2.o
# Test error output
./uacc err.c
./uacc2 err.c
# Compile the compiler with itself again
./uacc uacc.c output uacc3.o
./uacc2 uacc.c output uacc4.o
cmp uacc3.o uacc4.o
```

Рисунок 4.19 – Bash скрипт що порівнює роботу обох версій компілятора

В результаті перевірки (рис. 4.20) видно що утиліта cmp не надрукувала жодних повідомлень, тобто результати роботи обох версій були ідентичні. Єдиний текст що було виведено під час роботи скрипту є повідомлення про помилку відвовідно до підрозділу 4.4. Причому обидві версії компілятора надрукували ідентичні повідомлення.

```
$ bash compare.sh
err.c: 3: x: undefined symbol
err.c: 3: x: undefined symbol
```

Рисунок 4.20 – Робота bash скрипту що порівнює версії компілятора

4.6 Прапор zero-vars

На прикладі наступної програми (рис. 4.21) буде продемонстровано роботу прапору компіляції **zero-vars** що записує нуль у всі локальні змінні на початку функції. Використання цього прапору компіляції робить написаний текст програми несумісним з іншими компіляторами мови програмування ANSI C.

```
/* zeros.c - Test zero-vars option */
int printf(const char *format, ...);

int main() {
  int x;
  int g;
  printf("%i\x0A", x);
  return g;
}
```

Рисунок 4.21 – Текст програми що перевіряє прапор **zero-vars**

Текст програми було спочатку скомпільовано без прапорів компіляції, а потім з прапором **zero-vars**. Як видно на скриншоті CLI (рис. 4.22) якщо не додавати прапор **zero-vars** то значення змінної **x** є непередбачуваним.

```
$ ./uacc zeros.c output zeros.o
$ gcc zeros.o -o zeros
$ ./zeros
1099934416
$ ./zeros
799976144
```

Рисунок 4.22 – Перевірка програми без прапору **zero-vars**

Але якщо зкомпілювати цю прогарму з прапором компіляції **zeros-vars** (рис. 4.23) змінна x завжди приймає значення нуль.

Використовуючи утиліту objdump можна подивитись (рис. 4.24) на те які інструкції були згенеровані при використанні прапору компіляції **zero-vars**. На початку функції компілятор згенерував виклик функції

стандартної бібліотеки **memset** щоб записати нуль у всіх байтах що належать локальним змінним.

```
$ ./uacc zero-vars zeros.c output zeros.o
$ gcc zeros.o -o zeros
$ ./zeros
0
$ ./zeros
0
```

Рисунок 4.23 – Перевірка програми з прапором **zero-vars**

```
$ objdump -Sr zeros.o
0000000000000000 <main>:
   0:     55                      push   %rbp
   1:     48 89 e5                mov    %rsp,%rbp
   4:     48 81 ec 10 00 00 00    sub    $0x10,%rsp
   b:     ba 10 00 00 00          mov    $0x10,%edx
  10:     48 89 e7                mov    %rsp,%rdi
  13:     31 f6                   xor    %esi,%esi
  15:     e8 00 00 00 00          call   1a <main+0x1a>
              16: R_X86_64_PLT32  memset-0x4
...
```

Рисунок 4.24 – Інструкції та релокації програми

4.7 Прапор log-calls

На прикладі програми що рахує слова в тексті (рис. 4.25) буде показано роботу прапору компіляції **log-calls**. При компіляції тексту програми з цим прапором, компілятор генерує додаткові інструкції на початку кожної функції що друкують назву функції. Це дозволяє швидко отримати історію викликів функцій для подальшого аналізу того як працює програма.

Скриншот CLI (рис. 4.26) містить команди компіляції, лінкування, та запуску програми що рахує кількість слів. На вхід програми було надіслано файл що містить 7 слів. Оскільки текст програми був скомпільований з прапором **log-calls**, бачимо що були надруковані назви функції в хронологічному порядку їх виклику. Функція **read_word** була викликана 8

разів, де останній виклик не прочитав жодного слова, функція **count_words** закінчила роботу і було виведено результат.

```
/* count.c - Count words in a text file */
int getchar();
int printf(const char *format, ...);
int isalnum(int ch);
/*****************************************/
int _char;
/*****************************************/
int read_word() {
  while (isalnum(_char) == 0) {
    if (_char == 0xFFFFFFFF)
      return 0;
    _char = getchar();
  }
  while (isalnum(_char) != 0)
    _char = getchar();
  return 1;
}
/*****************************************/
int count_words() {
  int n;
  n = 0;
  _char = getchar();
  while (read_word())
    ++n;
  return n;
}
/*****************************************/
int main() {
  return printf("%i\x0A", count_words());
}
```

Рисунок 4.25 – Текст програми що рахує слова

```
$ ./uacc output count.o count.c log-calls
$ gcc -o count count.o
$ ./count < 7words.txt
main
count_words
read_word
read_word
read_word
read_word
read_word
read_word
read_word
read_word
7
```

Рисунок 4.26 – Компіляція, лінкування, та робота програми

Використовуючи утиліту objdump можна побачити інструкції що друкують назву функції на початку кожної функції (рис. 4.27). Інструкції на які прапор **log-calls** не впливає були замінені трьокрапкою.

```
0000000000000000 <read_word>:
...
   b:     ba 0a 00 00 00          mov    $0xa,%edx
  10:     bf 02 00 00 00          mov    $0x2,%edi
  15:     48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi
               18: R_X86_64_PC32   .data-0x4
  1c:     e8 00 00 00 00          call   21
               1d: R_X86_64_PLT32  write-0x4
...
0000000000000102 <count_words>:
...
 10d:     ba 0c 00 00 00          mov    $0xc,%edx
 112:     bf 02 00 00 00          mov    $0x2,%edi
 117:     48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi
               11a: R_X86_64_PC32  .data+0x7
 11e:     e8 00 00 00 00          call   123
               11f: R_X86_64_PLT32 write-0x4
...
000000000000018a <main>:
...
 195:     ba 05 00 00 00          mov    $0x5,%edx
 19a:     bf 02 00 00 00          mov    $0x2,%edi
 19f:     48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi
               1a2: R_X86_64_PC32  .data+0x14
 1a6:     e8 00 00 00 00          call   1ab
               1a7: R_X86_64_PLT32 write-0x4
...
```

Рисунок 4.27 – Інструкції та релокації програми що рахує кількість слів

За допомогою утиліти objdump можна також перевірити вміст секції **.data** у створеному ELF файлі (рис. 4.28). Видно що секція містить назви усіх функцій з тексту програми. Після кожної назви компілятор додає символ переведення на наступний рядок щоб кожна назва була на окремому рядку.

```
$ objdump -s -j .data count.o
 0000 72656164 5f776f72 640a0063 6f756e74  read_word..count
 0010 5f776f72 64730a00 6d61696e 0a002569  _words..main..%i
 0020 0a00
```

Рисунок 4.28 – Вміст секції **.data** програми що рахує кількість слів

Альтернативою прапору компіляції **`log-calls`** є команда **`backtrace`** у дебагері GNU gdb [17]. Але така альтернатива потребує не тільки рекомпіляції тексту програми з додаванням інформації для дебагу, але й запуску програми через дебагер.

4.8 Пакування символів

Наступна програма (рис. 4.29) використовує розширення мови програмування ANSI C що дозволяє упаковувати (об'єднувати) до 8-ми символів в одній константі.

```
/* pack.c - Character packing */
int printf(const char *format, ...);
const char *strchr(const char *str, int ch);
/*******************************************/
int get_extension(const char *filename) {
  char *dot;
  char *cp;
  int ext;
  cp = (void *)0;
  dot = strchr(filename, '.');
  while (dot != (void *)0) {
    cp = dot;
    dot = strchr(dot + 1, '.');
  }
  if (cp == (void *)0)
    return 0;
  ext = 0;
  while (*cp) {
    ext = (ext << 8) | *cp;
    ++cp;
  }
  return ext;
}
/*******************************************/
int main() {
  if (get_extension("file1.txt") == '.txt')
    printf(".txt\x0A");
  if (get_extension("file2.txt") == '.png')
    printf(".png\x0A");
  if (get_extension("file3.bin") == '.bin')
    printf(".bin\x0A");
  return 0;
}
```

Рисунок 4.29 – Текст програми, що використовує упаковані символи

Це розширення вшите у компілятор і не може бути вимкнуто, але оскільки це розришрення ніяк не впливає на код що слідує стандарту мови програмування ANSI C, немає потреби у можливості вимкнути це розширення. У прикладі вище, упаковані символи використані щоб більш ефективно порівнювати невеликі частини тексту, такі як розширення файлів.

На скриншоті CLI (рис. 4.30) можна побачити компіляцію, лінкування, та запуск програми що знаходить та порівнює розширення файлів використовуючи упаковані символи. Порівняння значень відбувається через оператор **==** що працює швидше за виклик стандартних функцій **memcmp** та **strcmp**.

```
$ ./uacc output pack.o pack.c
$ gcc pack.o -o pack
$ ./pack
.txt
.bin
```

Рисунок 4.30 – Компіляція, лінкування, та запуск програми

ВИСНОВКИ

У даній кваліфікаційній роботі магістра було спроєктовано та розроблено новий компілятор ANSI C для архітектури процесорів x86-64, що працює на операційній системі Linux. Потім на основі цього компілятору були розроблені та протестовані унікальні розширення мови ANSI C.

У першому розділі перелічені вимоги до компілятора. Описано формат виклику компілятора у CLI. Схематично зображено як відбувається розробка з використанням компілятора.

У другому розділі було описано план розробки компілятора. Задано загальну структуру компілятора. Детально описано етапи розробки та пояснено причину такого порядку цих етапів.

У третьому розділі було програмно реалізовано спроєктований компілятор. Реалізація відбувалась відповідно до запланованих етапів розробки. Описані технічні деталі кожного модуля та взаємодію з іншими модулями.

У четвертому розділі успішно протестована та продемонстрована робота компілятора. Розглянуто роботу компілятора в умовах коли текст програми не містить помилок. Розглянуто поведінку компілятора у випадку коли тект програми містить помилку. Продемонстровані розроблені розширення мови C.

ПЕРЕЛІК ПОСИЛАНЬ

1. Kernighan B.W., Ritchie D.M. The C Programming Language, 2nd edition. Prentice Hall, Englewood Cliffs, 1988. 238 p.

2. Format of Executable and Linking Format (ELF) files. *Linux manual page*. URL: https://man7.org/linux/man-pages/man5/elf.5.html (дата звернення: 13.04.2024).

3. Common x86 Calling Conventions. URL: https://lospi.net/assembly/c/developing/software/2015/04/04/common-x86-calling-conventions.html (дата звернення: 09.05.2024).

4. Formatting Error Messages. *GNU Coding Standards*. URL: https://www.gnu.org/prep/standards/html_node/Errors.html (дата звернення: 02.06.2024).

5. ELF File Structure. *OSDev.org*. URL: https://wiki.osdev.org/ELF#File_Structure (дата звернення: 09.05.2024).

6. ELF Header. *OSDev.org*. URL: https://wiki.osdev.org/ELF#ELF_Header (дата звернення: 22.05.2024).

7. ANSI X3. 159-1989. Programming Language C. New York, 1989. 350 p.

8. Data Alignment and Timing for the 64-Bit Avalon. URL: https://www.intel.com/content/www/us/en/docs/programmable/683733/14-1/data-alignment-and-timing-for-the-64-42314.html (дата звернення: 10.07.2024).

9. Sections. *Oracle Linker and Libraries Guide*. URL: https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-94076.html (дата звернення: 10.07.2024).

10. Symbol Table Section. *Oracle Linker and Libraries Guide*. URL: https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-79797.html (дата звернення: 15.07.2024).

11. Cohen D. On Holy Wars and a Plea for Peace. *Computer*. 1981. Vol. 14, no. 10. P. 48–54. DOI: 10.1109/C-M.1981.220208.

12. Overview. *GNU ld Manual*. URL: https://ftp.gnu.org/pub/old-gnu/Manuals/ld-2.9.1/html_node/ld_1.html (дата звернення: 29.08.2024).

13. gcc(1). *Linux manual page*. URL: https://ftp.gnu.org/pub/old-gnu/Manuals/ld-2.9.1/html_node/ld_1.html (дата звернення: 29.08.2024).

14. Objdump. *GNU Binary Utilities*. URL: https://sourceware.org/binutils/docs-2.27/binutils/objdump.html (дата звернення: 17.09.2024).

15. About SDL. Simple DirectMedia Layer. URL: https://www.libsdl.org/ (дата звернення: 02.10.2024).

16. Invoking cmp. *GNU Comparing and Merging Files.* URL: https://www.gnu.org/software/diffutils/manual/html_node/Invoking-cmp.html (дата звернення: 16.06.2024).

17. GDB: The GNU Project Debugger. URL: https://www.sourceware.org/gdb/ (дата звернення: 08.10.2024).

ДОДАТОК А

Джерельний код компілятора

```
/*************************************************************/
int   isspace(int ch);
int   isupper(int ch);
int   islower(int ch);
int   isdigit(int ch);
int   isalpha(int ch);
int   isalnum(int ch);
int   write  (int fd, const char *buffer, int size);
int   _exit  (int status);
void *memset (void *a, int c, unsigned long size);
void *memcpy (void *a, const void *b, unsigned long size);
int   memcmp (const void *a, const void *b, unsigned long size);
int   strcmp (const char *a, const char *b);
void *calloc (unsigned long n, unsigned long sz);
void *fopen  (const char *name, const char *mode);
void  fclose (void *fp);
unsigned long fread(void *buf, unsigned long sz
                  , unsigned long count, void *fp);
unsigned long fwrite(const void *buf, unsigned long sz
                   , unsigned long count, void *fp);
unsigned long strlen(const char *s);
/*************************************************************/
char _srcbuf[131072]; /* 128 KB */
char _srcnamebuf[256];
int _srcnamebuf_sz;
int _cursrc;
const char *_srcname;
char *_src;
char *_cp;
char _sources[384]; /* up to 16 */
int _sources_n;
char _srcstk[8];
int _srcstk_n;
char *_token_start;
char *_outname;
int _token;
char _strtab[65536]; /* 64 KB */
int _strtab_sz;
char _symtab[24576]; /* up to 1024 symbols */
char _symprop[1024];
int _symtab_n;
char _relatext[24576]; /* up to 1024 relocations */
int _relatext_n;
char _text[131072]; /* 128 KB */
int _text_sz;
char _data[131072]; /* 128 KB */
```

```
int _data_sz;
int _bss_sz;
char _elf[131072]; /* 128 KB */
int _elf_sz;
char _sh[1024]; /* up to 16 sections */
int _sh_n;
int SHN_GNUSTACK;
int SHN_SHSTRTAB;
int SHN_STRTAB;
int SHN_SYMTAB;
int SHN_RELATEXT;
int SHN_TEXT;
int SHN_DATA;
int SHN_BSS;
int SYM_DATA;
int SYM_BSS;
int SYM_WRITE; /* optional */
int SYM_MEMSET; /* optional */
char _nt_typ[256];
char _nt_val[1024]; /* 256 * 4 */
char _nt_tok[1024]; /* 256 * 4 */
int _nt_count;
char _tempbuff[256];
int _frame_off;
int _frame_sz;
char _locals[512]; /* up to 64 */
int _locals_n;
char _strlit[512];
int _strlit_sz;
int o_log_calls;
int o_zero_vars;
char _lsline[1024]; /* up to 256 */
int _ls_n;
int _ls_i;
/************************************************************/
int align(int offset, int power_of_2) {
  int mask;
  int mod;
  mask = power_of_2 - 1;
  mod = offset & mask;
  if (mod != 0) {
    offset -= mod;
    offset += power_of_2;
  }
  return offset;
}
/************************************************************/
char *set64(char *a, unsigned long n) {
  char *t;
  char *end;
  t = a;
  end = a + 8;
  while (t != end) {
```

```
    *t = n & 0xFF;
    ++t;
    n >>= 8;
  }
  return a;
}
/*****************************************************************/
char *set32(char *a, int n) {
  char *t;
  char *end;
  t = a;
  end = a + 4;
  while (t != end) {
    *t = n & 0xFF;
    ++t;
    n >>= 8;
  }
  return a;
}
/*****************************************************************/
char *set16(char *a, int n) {
  char *t;
  char *end;
  t = a;
  end = a + 2;
  while (t != end) {
    *t = n & 0xFF;
    ++t;
    n >>= 8;
  }
  return a;
}
/*****************************************************************/
char *set8(char *a, int n) {
  *a = n;
  return a;
}
/*****************************************************************/
char *set64u(char *a, int n) {
  set32(a, n);
  memset(a + 4, 0, 4);
  return a;
}
/*****************************************************************/
char *set64i(char *a, int n) {
  int c;
  set32(a, n);
  c = 0xFF;
  if (0 <= n) c = 0;
  memset(a + 4, c, 4);
  return a;
}
/*****************************************************************/
```

```
int get8u(char *a) {
  return *a & 0xFF;
}
/*************************************************************/
int get32(char *a) {
  char *end;
  char *t;
  int n;
  t = a + 3;
  end = a - 1;
  n = 0;
  while (t != end) {
    n <<= 8;
    n += get8u(t);
    --t;
  }
  return n;
}
/*************************************************************/
long get64(char *a) {
  char *end;
  char *t;
  long n;
  t = a + 7;
  end = a - 1;
  n = 0;
  while (t != end) {
    n <<= 8;
    n += get8u(t);
    --t;
  }
  return n;
}
/*************************************************************/
int get16(char *a) {
  char *end;
  char *t;
  int n;
  t = a + 1;
  end = a - 1;
  n = 0;
  while (t != end) {
    n <<= 8;
    n += get8u(t);
    --t;
  }
  return n;
}
/*************************************************************/
int new_token() {
  int offset;
  int length;
  int source;
```

```
  offset = _token_start - _src;
  length = _cp - _token_start;
  source = _cursrc << 26;
  return (length << 17) | offset | source;
}
/*************************************************************/
char *srcname(int source) {
  return (char *)get64(source * 24 + _sources);
}
/*************************************************************/
char *srcbuf(int source) {
  return (char *)get64(source * 24 + _sources + 8);
}
/*************************************************************/
char *srccp(int source) {
  return (char *)get64(source * 24 + _sources + 16);
}
/*************************************************************/
int toksrc(int token) {
  return (token >> 26) & 0x1F;
}
/*************************************************************/
int toklen(int token) {
  return (token >> 17) & 0x1FF;
}
/*************************************************************/
int tokoff(int token) {
  return token & 0x1FFFF;
}
/*************************************************************/
char *tokstr(int token) {
  return srcbuf(toksrc(token)) + tokoff(token);
}
/*************************************************************/
int tokequ(int token, char *cstr) {
  unsigned long n;
  n = toklen(token);
  if (strlen(cstr) == n)
    return memcmp(cstr, tokstr(token), n) == 0;
  return 0;
}
/*************************************************************/
int tokrow(int token) {
  char *end;
  char *t;
  int n;
  end = tokstr(token);
  t = srcbuf(toksrc(token));
  n = 1;
  while (t != end) {
    if (*t == 10) ++n;
    ++t;
  }
```

```
  return n;
}
/*************************************************************/
int int_to_str(int num, char *buf) {
  char *t;
  int p10;
  int digits;
  p10 = 10;
  digits = 1;
  while (p10 <= num) {
    p10 *= 10;
    ++digits;
  }
  t = buf + digits;
  *t = 0;
  while (num != 0) {
    --t;
    *t = num % 10 + '0';
    num /= 10;
  }
  return digits;
}
/*************************************************************/
int error(int optional_token, char *error_message) {
  const char *name;
  int row;
  int digits;
  if (optional_token != 0) {
    row = tokrow(optional_token);
    digits = int_to_str(row, _tempbuff);
    name = srcname(toksrc(optional_token));
    write(2, name, strlen(name));
    write(2, ": ", 2);
    write(2, _tempbuff, digits);
    write(2, ": ", 2);
    write(2, tokstr(optional_token), toklen(optional_token));
    write(2, ": ", 2);
  }
  write(2, error_message, strlen(error_message));
  write(2, "\x0A", 1);
  _exit(1);
  return 0;
}
/*************************************************************/
void *do_mallocz(unsigned long size) {
  void *ptr;
  ptr = calloc(size, 1);
  if (ptr == (void *)0) error(0, "out of memory");
  return ptr;
}
/*************************************************************/
void *do_fopen(const char *name, const char *mode) {
  void *fp;
```

```
  fp = fopen(name, mode);
  if (fp == (void *)0) error(0, "no such file or directory");
  return fp;
}
/*****************************************************************/
int is_alpha(int ch) {
  return (ch == '_') + isalpha(ch);
}
/*****************************************************************/
int is_alnum(int ch) {
  return (ch == '_') + isalnum(ch);
}
/*****************************************************************/
int is_repeat(int ch) {
  if ('<' <= ch) if (ch <= '>') return 1;
  if (ch == '+') return 1;
  if (ch == '-') return 1;
  if (ch == '.') return 1;
  return 0;
}
/*****************************************************************/
int read_token() {
  int spaces;
  int q1;
  int q2;
  int end;
  _token_start = _cp;
  spaces = 1;
  while (spaces) {
    spaces = 0;
    if (*_cp == ' ') {
      while (isspace(*_cp)) ++_cp;
      spaces = 1;
    }
    _token_start = _cp;
    if (*_cp == '/') {
      ++_cp;
      if (*_cp != '*') {
        while (is_repeat(*_cp)) ++_cp;
        return new_token();
      }
      while (*_cp != '/') ++_cp;
      ++_cp;
      spaces = 1;
    }
  }
  _token_start = _cp;
  if (*_cp == 0) return _token_start - _src;
  if (is_alnum(*_cp)) {
    while (is_alnum(*_cp)) ++_cp;
    return new_token();
  }
  end = *_cp;
```

```
  q1 = end == '"';
  q2 = end == 39; /* ' */
  if (q1 + q2) {
    ++_cp;
    while (*_cp != end) ++_cp;
    ++_cp;
    return new_token();
  }
  ++_cp;
  while (is_repeat(*_cp)) ++_cp;
  return new_token();
}
/*************************************************************/
int try_load_lsline() {
  char *t;
  int tok;
  t = _lsline;
  _ls_i = _ls_n = 0;
  while (1) {
    tok = read_token();
    if (*tokstr(tok) == '\x0A') return 0;
    set32(t, tok);
    t += 4;
    ++_ls_n;
    if (toklen(tok) == 0) {
      if (_ls_n != 1) error(tok, "no newline in the end");
      return 0;
    }
  }
  return 0;
}
/*************************************************************/
int load_lsline_may_eof() {
  int tok;
  _ls_i = _ls_n = 0;
  while (_ls_i == _ls_n) try_load_lsline();
  tok = get32(_lsline);
  return toklen(tok) == 0;
}
/*************************************************************/
int load_lsline() {
  char *stk;
  char *t;
  int eof;
  int do_load;
  do_load = 1;
  while (do_load != 0) {
    eof = load_lsline_may_eof();
    do_load = 0;
    if (eof != 0) if (_srcstk_n != 0) {
      --_srcstk_n;
      stk = _srcstk + _srcstk_n;
      _cursrc = *stk;
```

```
      t = _cursrc * 24 + _sources;
      _srcname = (char *)get64(t);
      _src = (char *)get64(t + 8);
      _cp = (char *)get64(t + 16);
      do_load = 1;
    }
  }
  return 0;
}
/*************************************************************/
int new_source(const char *name) {
  char *src;
  void *fp;
  int size;
  _srcname = name;
  fp = do_fopen(_srcname, "rb");
  size = fread(_srcbuf, 1, 131072, fp);
  fclose(fp);
  if (size <= 0) error(0, "the source file is empty");
  memcpy(_src = do_mallocz(size + 1), _srcbuf, size);
  _cp = _src;
  src = _sources_n * 24 + _sources;
  set64(src, (unsigned long)_srcname);
  set64(src + 8, (unsigned long)_src);
  set64(src + 16, (unsigned long)_cp);
  ++_sources_n;
  return _sources_n - 1;
}
/*************************************************************/
int pp_include() {
  char *str;
  char *name;
  char *stk;
  int tok;
  int len;
  int oldsrc;
  /* SAVE _cp */
  oldsrc = _cursrc;
  set64(oldsrc * 24 + _sources + 16, (unsigned long)_cp);
  /* GET NEW SOURCE */
  tok = get32(_lsline + 8);
  str = tokstr(tok);
  len = toklen(tok);
  if (*str != '"') error(tok, "unknown #include format");
  if (*str == '"') {
    _srcnamebuf_sz = len - 2;
    memcpy(_srcnamebuf, str + 1, _srcnamebuf_sz);
  }
  name = do_mallocz(_srcnamebuf_sz + 1);
  memcpy(name, _srcnamebuf, _srcnamebuf_sz);
  _cursrc = new_source(name);
  /* PUSH OLD ONE */
  stk = _srcstk + _srcstk_n;
```

```
  *stk = oldsrc;
  ++_srcstk_n;
  return 1;
}
/*************************************************************/
int preprocess() {
  int direct;
  if (tokequ(get32(_lsline), "#") == 0) return 0;
  direct = get32(_lsline + 4);
  if (tokequ(direct, "include")) return pp_include();
  return 1;
}
/*************************************************************/
int pull_token() {
  if (_ls_i == _ls_n) {
    load_lsline();
    while (preprocess())
      load_lsline();
  }
  _token = get32(_ls_i * 4 + _lsline);
  ++_ls_i;
  return _token;
}
/*************************************************************/
int inspect(char *format) {
  int first;
  if (*format == ' ') {
    ++format;
    first = *tokstr(_token);
    if (*format == 'A') return is_alpha(first);
    if (*format == 'D') return isdigit(first);
    return first == *format;
  }
  return tokequ(_token, format);
}
/*************************************************************/
int accept(char *format) {
  int accepted_token;
  accepted_token = _token;
  if (inspect(format) == 0) return 0;
  pull_token();
  return accepted_token;
}
/*************************************************************/
int expect(char *format) {
  int accepted_token;
  accepted_token = accept(format);
  if (accepted_token == 0) error(_token, "unexpected token");
  return accepted_token;
}
/*************************************************************/
int set_node(int ni, int type, int token, int value) {
  char *nt;
```

```
  set8(_nt_typ + ni, type);
  nt = (ni << 2) + _nt_val;
  set32(nt, value);
  nt = (ni << 2) + _nt_tok;
  set32(nt, token);
  return ni;
}
/*************************************************************/
int new_node(int type, int token, int value) {
  ++_nt_count;
  return set_node(_nt_count - 1, type, token, value);
}
/*************************************************************/
int new_bnode(int type, int token, int left, int right) {
  return new_node(type, token, (left << 8) + right);
}
/*************************************************************/
int node_type(int ni) {
  return get8u(_nt_typ + ni);
}
/*************************************************************/
int node_token(int ni) {
  return get32((ni << 2) + _nt_tok);
}
/*************************************************************/
int node_value(int ni) {
  return get32((ni << 2) + _nt_val);
}
/*************************************************************/
int node_left(int ni) {
  return node_value(ni) >> 8;
}
/*************************************************************/
int node_right(int ni) {
  return node_value(ni) & 0xFF;
}
/*************************************************************/
int digit_value(int ch) {
  if (isdigit(ch)) return ch - '0';
  if (isupper(ch)) return ch - 'A' + 10;
  if (islower(ch)) return ch - 'a' + 10;
  return 255;
}
/*************************************************************/
int parse32(int token) {
  char *t;
  char *end;
  int num;
  int digit;
  int base;
  int len;
  int base_ch;
  t = tokstr(token);
```

```
  len = toklen(token);
  end = t + len;
  base = 10;
  if (2 <= len) if (*t == '0') {
    base = 8;
    ++t;
    base_ch = *t;
    if (base_ch == 'x') base = 16;
    if (base_ch == 'X') base = 16;
    if (base != 8) ++t;
  }
  num = 0;
  while (t != end) {
    num *= base;
    digit = digit_value(*t);
    if (base <= digit) error(token, "invalid constant");
    num += digit;
    ++t;
  }
  return num;
}
/*************************************************************/
int x_assign();
/*************************************************************/
int x_primary() {
  int tok;
  int x;
  tok = _token;
  if (accept("(")) {
    x = x_assign();
    expect(")");
    return x;
  }
  if (accept(" D")) return new_node('C', tok, parse32(tok));
  if (accept(" \x22")) return new_node('"', tok, 0);
  if (accept(" '")) return new_node(39, tok, 0);
  return new_node('I', expect(" A"), 0);
}
/*************************************************************/
int x_args() {
  int x;
  int tok;
  if (accept(")")) return 0xFF;
  x = x_assign();
  tok = _token;
  while (accept(",")) {
    x = new_bnode('`', tok, x, x_assign());
    tok = _token;
  }
  expect(")");
  return x;
}
/*************************************************************/
```

```
int x_suffix() {
  int x;
  int tok;
  x = x_primary();
  tok = accept("(");
  if (tok == 0) return x;
  return new_bnode('F', tok, x, x_args());
}
/*************************************************************/
int base_type_or_0() {
  int mods;
  mods = accept("const");
  mods |= accept("unsigned");
  if (accept("int")) return 1;
  if (accept("long")) return 1;
  if (accept("char")) return 2;
  if (accept("void")) return 2;
  if (mods != 0) return 1;
  return 0;
}
/*************************************************************/
int x_prefix() {
  int tok;
  int x;
  int rval;
  tok = _token;
  if (accept("*")) return new_bnode('X', tok, x_prefix(), 0);
  if (accept("++")) {
    x = x_prefix();
    rval = new_bnode('+', tok, x, new_node('C', tok, 1));
    return new_bnode('=', tok, x, rval);
  }
  if (accept("--")) {
    x = x_prefix();
    rval = new_bnode('-', tok, x, new_node('C', tok, 1));
    return new_bnode('=', tok, x, rval);
  }
  if (accept("(")) {
    if (base_type_or_0() == 0) {
      x = x_assign();
      expect(")");
      return x;
    }
    while (accept("*")) /* nope */;
    expect(")");
    return x_prefix();
  }
  return x_suffix();
}
/*************************************************************/
int accept_math() {
  if (accept("+"))  return '+';
  if (accept("-"))  return '-';
```

```
  if (accept("*"))  return '*';
  if (accept("/"))  return '/';
  if (accept("%"))  return '%';
  if (accept("&"))  return '&';
  if (accept("|"))  return '|';
  if (accept("^"))  return '^';
  if (accept(">>")) return 'R';
  if (accept("<<")) return 'L';
  if (accept("==")) return 'E';
  if (accept("<=")) return 'b';
  if (accept("!=")) return 'D';
  return 0;
}
/*************************************************************/
int x_math() {
  int x;
  int tok;
  int type;
  x = x_prefix();
  tok = _token;
  type = accept_math();
  while (type != 0) {
    x = new_bnode(type, tok, x, x_prefix());
    tok = _token;
    type = accept_math();
  }
  return x;
}
/*************************************************************/
int x_opassign(int xt, int tok, int lval) {
  int rval;
  rval = new_bnode(xt, tok, lval, x_assign());
  return new_bnode('=', tok, lval, rval);
}
/*************************************************************/
int x_assign() {
  int x;
  int tok;
  x = x_math();
  tok = _token;
  if (accept("+=")) return x_opassign('+', tok, x);
  if (accept("-=")) return x_opassign('-', tok, x);
  if (accept("*=")) return x_opassign('*', tok, x);
  if (accept("/=")) return x_opassign('/', tok, x);
  if (accept("%=")) return x_opassign('%', tok, x);
  if (accept("&=")) return x_opassign('&', tok, x);
  if (accept("|=")) return x_opassign('|', tok, x);
  if (accept("^=")) return x_opassign('^', tok, x);
  if (accept("<<=")) return x_opassign('L', tok, x);
  if (accept(">>=")) return x_opassign('R', tok, x);
  if (accept("=")) return new_bnode('=', tok, x, x_assign());
  return x;
}
```

```
/*************************************************************/
int x_args_count(int x) {
  int count;
  if (x == 0xFF) return 0;
  count = 1;
  while (node_type(x) == '`') {
    x = node_left(x);
    ++count;
  }
  return count;
}
/*************************************************************/
int nstring(int n, const char *s) {
  char *t;
  int i;
  int k;
  int limit;
  i = 0;
  limit = _strtab_sz - 1;
  while (i <= limit) {
    k = strlen(_strtab + i);
    if (n <= k) {
      t = _strtab + i;
      t += k;
      t -= n;
      if (memcmp(t, s, n) == 0) return t - _strtab;
    }
    i += k;
    ++i;
  }
  i = _strtab_sz;
  memcpy(_strtab + i, s, n);
  _strtab_sz = _strtab_sz + n;
  ++_strtab_sz;
  return i;
}
/*************************************************************/
int string(const char *cstr) {
  return nstring(strlen(cstr), cstr);
}
/*************************************************************/
int findsymbol(int name) {
  char *sym;
  int symi;
  symi = _symtab_n - 1;
  sym = symi * 24 + _symtab;
  while (symi != 0) {
    if (get32(sym) == name) return symi;
    --symi;
    sym -= 24;
  }
  return 0;
}
```

```
/*************************************************************/
int declare3(int name, int info, int shndx) {
  char *sym;
  int symi;
  symi = findsymbol(name);
  if (symi != 0) return symi;
  symi = _symtab_n;
  ++_symtab_n;
  sym = symi * 24 + _symtab;
  set32(sym, name);
  set8(sym + 4, info);
  set16(sym + 6, shndx);
  return symi;
}
/*************************************************************/
int declare(int name, int type) {
  return declare3(name, type + 16, 0);
}
/*************************************************************/
int define(int symi, int shndx, int value, int size) {
  char *sym;
  sym = symi * 24 + _symtab;
  set16(sym + 6, shndx);
  set64u(sym + 8, value);
  set64u(sym + 16, size);
  return symi;
}
/*************************************************************/
int relocate(int offset, int symi, int type, int addend) {
  char *r;
  int ri;
  ri = _relatext_n;
  ++_relatext_n;
  r = ri * 24 + _relatext;
  set64u(r, offset);
  set32(r + 8, type);
  set32(r + 12, symi);
  set64i(r + 16, addend);
  return ri;
}
/*************************************************************/
int relocate_plt32(int offset, int symi, int addend) {
  int diff;
  char *sym;
  sym = symi * 24 + _symtab;
  if (get16(sym + 6) != 0) {
    diff = get32(sym + 8) - offset;
    set32(_text + offset, diff + addend);
    return 0;
  }
  return relocate(offset, symi, 4, addend);
}
/*************************************************************/
```

```
int relocate_pc32(int offset, int symi, int addend) {
  return relocate(offset, symi, 2, addend);
}
/*************************************************************/
int declsect(int name, int type, int flg, int alg, int esz) {
  char *sh;
  int shndx;
  shndx = _sh_n;
  ++_sh_n;
  sh = (shndx << 6) + _sh;
  set32(sh, name);
  set8(sh + 4, type);
  set8(sh + 8, flg);
  set8(sh + 48, alg);
  set8(sh + 56, esz);
  return shndx;
}
/*************************************************************/
int linksect(int shndx, int link, int info) {
  char *sh;
  sh = (shndx << 6) + _sh;
  set8(sh + 40, link);
  set8(sh + 44, info);
  return shndx;
}
/*************************************************************/
int defnsect(int shndx, int offset, int size) {
  char *sh;
  sh = (shndx << 6) + _sh;
  set64u(sh + 24, offset);
  set64u(sh + 32, size);
  return shndx;
}
/*************************************************************/
int writesect(int shndx, int size, char *data) {
  _elf_sz = align(_elf_sz, 32);
  defnsect(shndx, _elf_sz, size);
  memcpy(_elf + _elf_sz, data, size);
  _elf_sz += size;
  return shndx;
}
/*************************************************************/
int emit(int size, char *text) {
  memcpy(_text + _text_sz, text, size);
  _text_sz += size;
  return _text_sz - size;
}
/*************************************************************/
int emit_data(int size, char *data) {
  memcpy(_data + _data_sz, data, size);
  _data_sz += size;
  return _data_sz - size;
}
```

```
/*************************************************************/
int base_type() {
  int base;
  base = base_type_or_0();
  if (base == 0) return 1;
  return base;
}
/*************************************************************/
int local(int name, int offset) {
  char *loc;
  int loci;
  loci = _locals_n;
  ++_locals_n;
  loc = (loci << 3) + _locals;
  set32(loc, name);
  set32(loc + 4, offset);
  return loci;
}
/*************************************************************/
int flocal(int name, int size) {
  int loci;
  loci = local(name, _frame_off);
  _frame_sz += size;
  _frame_off -= size;
  return loci;
}
/*************************************************************/
int findlocal(int name) {
  char *loc;
  int loci;
  loci = _locals_n - 1;
  loc = (loci << 3) + _locals;
  while (0 <= loci) {
    if (get32(loc) == name) return loci;
    --loci;
    loc -= 8;
  }
  return 0xFF;
}
/*************************************************************/
int localoff(int loci) {
  char *loc;
  loc = (loci << 3) + _locals;
  return get32(loc + 4);
}
/*************************************************************/
int collect_params() {
  int name;
  if (accept(")")) return 0;
  base_type();
  while (accept("*")) /* nope */;
  name = expect(" A");
  flocal(nstring(toklen(name), tokstr(name)), 8);
```

```
  while (accept(",")) {
    if (accept("...") == 0) {
      base_type();
      while (accept("*")) /* nope */;
      name = expect(" A");
      flocal(nstring(toklen(name), tokstr(name)), 8);
    }
  }
  expect(")");
  return 0;
}
/*************************************************************/
int c_ident(int token) {
  char *sym;
  int i;
  int name;
  int offset;
  int isarr;
  name = nstring(toklen(token), tokstr(token));
  i = findlocal(name);
  if (i != 0xFF) {
    offset = emit(7, "\x48\x8B\x8D\x00\x00\x00\x00") + 3;
    set32(_text + offset, localoff(i));
    return 0;
  }
  i = findsymbol(name);
  if (i == 0) error(token, "undefined symbol");
  sym = i * 24 + _symtab;
  if (get8u(sym + 4) == 18) error(token, "is a function");
  isarr = get8u(_symprop + i);
  if (isarr == 0)
    offset = emit(7, "\x48\x8B\x0D\x00\x00\x00\x00") + 3;
  if (isarr == 1)
    offset = emit(7, "\x48\x8D\x0D\x00\x00\x00\x00") + 3;
  relocate_pc32(offset, i, 0-4);
  return 0;
}
/*************************************************************/
int readhex(char *d0, char *d1) {
  return (digit_value(*d0) << 4) + digit_value(*d1);
}
/*************************************************************/
int grow_strlit(int token) {
  char *t;
  char *end;
  char *buf;
  int v;
  buf = _strlit + _strlit_sz;
  t = tokstr(token);
  end = t + toklen(token);
  --end;
  ++t;
  while (t != end) {
```

```
    v = *t;
    if (v == 92) { /* \ */
      *buf = readhex(t + 2, t + 3);
      t += 4;
    }
    if (v != 92) { /* != \ */
      *buf = v;
      ++t;
    }
    ++buf;
  }
  *buf = 0;
  return _strlit_sz = buf - _strlit;
}
/*************************************************************/
int c_primary(int x) {
  char *t;
  char *end;
  int offset;
  int addend;
  int xt;
  int pack;
  xt = node_type(x);
  if (xt == 'C') {
    offset = emit(5, "\xB9\x00\x00\x00\x00") + 1;
    set32(_text + offset, node_value(x));
    return 0;
  }
  if (xt == '"') {
    _strlit_sz = 0;
    grow_strlit(node_token(x));
    addend = emit_data(_strlit_sz + 1, _strlit) - 4;
    offset = emit(7, "\x48\x8D\x0D\x00\x00\x00\x00") + 3;
    relocate_pc32(offset, SYM_DATA, addend);
    return 0;
  }
  if (xt == 39) { /* ' */
    _strlit_sz = 0;
    grow_strlit(node_token(x));
    pack = 0;
    t = _strlit;
    end = t + _strlit_sz;
    while (t != end) {
      pack = (pack << 8) | *t;
      ++t;
    }
    offset = emit(5, "\xB9\x00\x00\x00\x00") + 1;
    set32(_text + offset, pack);
    return 0;
  }
  if (xt == 'I') return c_ident(node_token(x));
  error(node_token(x), "invalid primary expression");
  return 0;
```

```
}
/*************************************************************/
int c_assign(int x);
/*************************************************************/
int c_suffix(int x) {
  char *sym;
  char *reg;
  char *end;
  int func;
  int args;
  int num;
  int symi;
  int tok;
  int y;
  if (node_type(x) != 'F') return c_primary(x);
  func = node_left(x);
  args = node_right(x);
  tok = node_token(func);
  if (node_type(func) != 'I') error(tok, "is not a function");
  symi = findsymbol(nstring(toklen(tok), tokstr(tok)));
  if (symi == 0) error(tok, "undeclared function");
  sym = symi * 24 + _symtab;
  if (get8u(sym + 4) != 18) error(tok, "is not a function");
  num = x_args_count(args);
  emit(2, "\x50\x50");
  if (num == 0) {
    relocate_plt32(1+emit(5, "\xE8\x00\x00\x00\x00"), symi,0-4);
    emit(5, "\x48\x89\xC1\x58\x58");
    return 0;
  }
  y = args;
  while (node_type(y) == '`') {
    c_assign(node_right(y));
    emit(1, "\x50");
    y = node_left(y);
  }
  c_assign(y);
  emit(1, "\x50");
  reg = "\x07\x06\x02\x01\x08\x09";
  end = reg + num;
  while (reg != end) {
    if (*reg & 8) emit(1, "\x41");
    set8(_text + _text_sz, (*reg & 7) + 0x58);
    ++_text_sz;
    ++reg;
  }
  relocate_plt32(emit(5, "\xE8\x00\x00\x00\x00") + 1, symi,0-4);
  emit(5, "\x48\x89\xC1\x58\x58");
  return 0;
}
/*************************************************************/
int c_prefix(int x) {
  if (node_type(x) != 'X') return c_suffix(x);
```

```
  c_prefix(node_left(x));
  emit(7, "\x0F\xB6\x09\x48\x0F\xBE\xC9");
  return 0;
}
/*************************************************************/
int xt_is_math(int xt) {
  if (xt == '+') return 1;
  if (xt == '-') return 1;
  if (xt == '*') return 1;
  if (xt == '/') return 1;
  if (xt == '%') return 1;
  if (xt == '&') return 1;
  if (xt == '|') return 1;
  if (xt == '^') return 1;
  if (xt == 'R') return 1;
  if (xt == 'L') return 1;
  if (xt == 'E') return 1;
  if (xt == 'b') return 1;
  if (xt == 'D') return 1;
  return 0;
}
/*************************************************************/
int c_math(int x) {
  int xt;
  xt = node_type(x);
  if (xt_is_math(xt) == 0) {
    c_prefix(x);
    emit(3, "\x48\x89\xC8");
    return 0;
  }
  c_math(node_left(x)); /* RAX */
  c_prefix(node_right(x)); /* RCX */
  if (xt == '+') emit(3, "\x48\x01\xC8");
  if (xt == '-') emit(3, "\x48\x29\xC8");
  if (xt == '&') emit(3, "\x48\x21\xC8");
  if (xt == '|') emit(3, "\x48\x09\xC8");
  if (xt == '^') emit(3, "\x48\x31\xC8");
  if (xt == '*') emit(6, "\x48\x31\xD2\x48\xF7\xE1");
  if (xt == '/') emit(6, "\x48\x31\xD2\x48\xF7\xF1");
  if (xt == '%') emit(9,"\x48\x31\xD2\x48\xF7\xF1\x48\x89\xD0");
  if (xt == 'R') emit(3, "\x48\xD3\xE8");
  if (xt == 'L') emit(3, "\x48\xD3\xE0");
  if (xt == 'E') emit(15,
"\x48\x39\xC8\x48\xC7\xC0\x01\x00\x00\x00\x74\x03\x48\x31\xC0");
  if (xt == 'D') emit(15,
"\x48\x39\xC8\x48\xC7\xC0\x01\x00\x00\x00\x75\x03\x48\x31\xC0");
  if (xt == 'b') emit(15,
"\x48\x39\xC8\x48\xC7\xC0\x01\x00\x00\x00\x7E\x03\x48\x31\xC0");
  return 0;
}
/*************************************************************/
int c_lvident(int token) {
  char *sym;
```

```
  int i;
  int name;
  int offset;
  name = nstring(toklen(token), tokstr(token));
  i = findlocal(name);
  if (i != 0xFF) {
    offset = emit(7, "\x48\x89\x85\x00\x00\x00\x00") + 3;
    set32(_text + offset, localoff(i));
    return 0;
  }
  i = findsymbol(name);
  if (i == 0) error(token, "undefined symbol");
  sym = i * 24 + _symtab;
  if (get8u(sym + 4) == 18) error(token, "is a function");
  offset = emit(7, "\x48\x89\x05\x00\x00\x00\x00") + 3;
  relocate_pc32(offset, i, 0-4);
  return 0;
}
/*************************************************************/
int c_assign(int x) {
  int lv;
  int lvt;
  int lvtok;
  if (node_type(x) != '=') return c_math(x);
  c_assign(node_right(x));
  lv = node_left(x);
  lvt = node_type(lv);
  lvtok = node_token(lv);
  if (lvt == 'X') {
    c_prefix(node_left(lv));
    emit(2, "\x88\x01");
    return 0;
  }
  if (lvt == 'I') {
    c_lvident(lvtok);
    return 0;
  }
  error(lvtok, "invalid lvalue");
  return 0;
}
/*************************************************************/
int expression() {
  int x;
  _nt_count = 0;
  x = x_assign();
  c_assign(x);
  return 0;
}
/*************************************************************/
int statement() {
  int rets;
  int base;
  int name;
```

```
  int locals;
  int x1;
  int x2;
  int y1;
  int y2;
  base = base_type_or_0();
  if (base != 0) {
    accept("*");
    name = expect(" A");
    flocal(nstring(toklen(name), tokstr(name)), 8);
    expect(";");
    return 0;
  }
  if (accept("{")) {
    rets = 0;
    locals = _locals_n;
    while (accept("}") == 0) rets = rets + statement();
    _locals_n = locals;
    return rets;
  }
  if (accept("if")) {
    expect("(");
    expression();
    expect(")");
    y1 = emit(9, "\x48\x85\xC0\x0F\x84\x00\x00\x00\x00") + 5;
    statement();
    x1 = _text_sz;
    set32(_text + y1, x1 - y1 - 4);
    return 0;
  }
  if (accept("while")) {
    x2 = _text_sz;
    expect("(");
    expression();
    expect(")");
    y1 = emit(9, "\x48\x85\xC0\x0F\x84\x00\x00\x00\x00") + 5;
    statement();
    y2 = emit(5, "\xE9\x00\x00\x00\x00") + 1;
    x1 = _text_sz;
    set32(_text + y1, x1 - y1 - 4);
    set32(_text + y2, x2 - y2 - 4);
    return 0;
  }
  if (accept("return")) {
    if (inspect(";") == 0) expression();
    expect(";");
    emit(2, "\xC9\xC3");
    return 1;
  }
  if (inspect(";") == 0) expression();
  expect(";");
  return 0;
}
```

```
/*************************************************************/
int function(int name) {
  char *t;
  int symi;
  int begin;
  int frame_sz_off;
  int off2;
  int i;
  int offset;
  int addend;
  int args_sz;
  symi = declare(nstring(toklen(name), tokstr(name)), 2);
  _locals_n = 0;
  _frame_off = 0 - 8;
  _frame_sz = 0;
  collect_params();
  if (inspect("{") == 0) return expect(";");
  begin = _text_sz;
  define(symi, SHN_TEXT, begin, 0);
  emit(4, "\x55\x48\x89\xE5");
  frame_sz_off = emit(7, "\x48\x81\xEC\x00\x00\x00\x00") + 3;
  i = _locals_n - 1;
  args_sz = _locals_n * 8;
  while (0 <= i) {
    if (i == 0) emit(4, "\x48\x89\x7D\x00");
    if (i == 1) emit(4, "\x48\x89\x75\x00");
    if (i == 2) emit(4, "\x48\x89\x55\x00");
    if (i == 3) emit(4, "\x48\x89\x4D\x00");
    if (i == 4) emit(4, "\x4C\x89\x45\x00");
    if (i == 5) emit(4, "\x4C\x89\x4D\x00");
    set8(_text + _text_sz - 1, localoff(i));
    --i;
  }
  if (o_log_calls) {
    _strlit_sz = toklen(name);
    memcpy(_strlit, tokstr(name), _strlit_sz);
    t = _strlit + _strlit_sz;
    *t = 0x0A;
    ++t;
    *t = 0;
    offset = emit(5, "\xBA\x02\x00\x00\x00") + 1;
    set32(_text + offset, _strlit_sz + 1);
    emit(5, "\xBF\x02\x00\x00\x00");
    addend = emit_data(_strlit_sz + 2, _strlit) - 4;
    offset = emit(7, "\x48\x8D\x35\x00\x00\x00\x00") + 3;
    relocate_pc32(offset, SYM_DATA, addend);
    offset = emit(5, "\xE8\x00\x00\x00\x00") + 1;
    relocate_plt32(offset, SYM_WRITE, 0 - 4);
  }
  if (o_zero_vars) {
    off2 = emit(8, "\xBA\x00\x00\x00\x00\x48\x89\xE7") + 1;
    offset = emit(7, "\x31\xF6\xE8\x00\x00\x00\x00") + 3;
    relocate_plt32(offset, SYM_MEMSET, 0 - 4);
```

```
  }
  if (statement() == 0) emit(2, "\xC9\xC3");
  define(symi, SHN_TEXT, begin, _text_sz - begin);
  _frame_sz = align(_frame_sz, 16);
  set32(_text + frame_sz_off, _frame_sz);
  if (o_zero_vars) set32(_text + off2, _frame_sz - args_sz);
  return 0;
}
/*************************************************************/
int global(int base, int ptr, int name) {
  int size;
  int symi;
  symi = declare(nstring(toklen(name), tokstr(name)), 1);
  size = 8;
  if (base == 2) size = 1;
  if (ptr != 0)  size = 8;
  _bss_sz = align(_bss_sz, size);
  if (accept("[")) {
    size = parse32(expect(" D"));
    expect("]");
    set8(_symprop + symi, 1);
  }
  define(symi, SHN_BSS, _bss_sz, size);
  _bss_sz += size;
  expect(";");
  return 0;
}
/*************************************************************/
int declaration() {
  int base;
  int ptr;
  int name;
  base = base_type();
  ptr = accept("*");
  name = expect(" A");
  if (accept("(")) return function(name);
  return global(base, ptr, name);
}
/*************************************************************/
int program() {
  while (inspect("") == 0) declaration();
  return 0;
}
/*************************************************************/
int setup(int argc, void *argv) {
  char *t;
  char *end;
  char *arg;
  int p;
  t = argv;
  end = argc * 8 + t;
  t += 8;
  _outname = "a.o";
```

```
  while (t != end) {
    arg = (char *)get64(t);
    p = 0;
    if (strcmp(arg, "zero-vars") == 0) o_zero_vars = p = 1;
    if (strcmp(arg, "log-calls") == 0) o_log_calls = p = 1;
    if (strcmp(arg, "output") == 0) {
      p = 1;
      t += 8;
      if (t == end) error(0, "invalid 'output' option");
      _outname = (char *)get64(t);
    }
    if (p == 0) _srcname = arg;
    t += 8;
  }
  return 0;
}
/*************************************************************/
int main(int argc, char **argv) {
  void *fp;
  int size;
  setup(argc, argv);
  if (_srcname == (void *)0) error(0, "no input files");
  _cursrc = new_source(_srcname);
  /* DECLARE SECTIONS */
  string("");
  declsect(0, 0, 0, 0, 0);
  SHN_SHSTRTAB = declsect(string(".shstrtab"), 3, 0, 1, 0);
  SHN_STRTAB   = declsect(string(".strtab"), 3, 0, 1, 0);
  SHN_GNUSTACK = declsect(string(".note.GNU-stack"), 1,0,1,0);
  SHN_SYMTAB   = declsect(string(".symtab"), 2, 0, 8, 24);
  SHN_RELATEXT = declsect(string(".rela.text"), 4, 64, 8, 24);
  SHN_TEXT     = declsect(string(".text"), 1, 6, 1, 0);
  SHN_DATA     = declsect(string(".data"), 1, 3, 1, 0);
  SHN_BSS      = declsect(string(".bss"), 8, 3, 32, 0);
  /* SETUP STRTAB + SYMTAB */
  _symtab_n = 1;
  declare3(string(_srcname), 4, 0xFFF1);
  declare3(string(".text"), 3, SHN_TEXT);
  SYM_DATA = declare3(string(".data"), 3, SHN_DATA);
  SYM_BSS  = declare3(string(".bss"), 3, SHN_BSS);
  linksect(SHN_SYMTAB, SHN_STRTAB, _symtab_n);
  linksect(SHN_RELATEXT, SHN_SYMTAB, SHN_TEXT);
  if (o_log_calls != 0) SYM_WRITE = declare(string("write"), 2);
  if (o_zero_vars != 0) SYM_MEMSET=declare(string("memset"), 2);
  /* COMPILE */
  pull_token();
  program();
  expect("");
  /* ELF HEADER */
  memcpy(_elf,      "\x7F\x45\x4C\x46\x02\x01\x01", 7);
  memcpy(_elf + 16, "\x01\x00\x3E\x00\x01", 5);
  memcpy(_elf + 58, "\x40", 1);
  set16(_elf + 52, 64); /* e_ehsize */
```

```
  set16(_elf + 60, _sh_n);
  set16(_elf + 62, SHN_SHSTRTAB);
  _elf_sz = 64;
  /* WRITE SECTIONS */
  writesect(SHN_SHSTRTAB, _strtab_sz, _strtab);
  defnsect(SHN_STRTAB, _elf_sz - _strtab_sz, _strtab_sz);
  defnsect(SHN_GNUSTACK, 64, 0);
  writesect(SHN_SYMTAB, _symtab_n * 24, _symtab);
  writesect(SHN_RELATEXT, _relatext_n * 24, _relatext);
  writesect(SHN_TEXT, _text_sz, _text);
  writesect(SHN_DATA, _data_sz, _data);
  defnsect(SHN_BSS, 64, _bss_sz);
  /* SECTION HEADERS */
  _elf_sz = align(_elf_sz, 32);
  set64u(_elf + 40, _elf_sz); /* e_shoff */
  size = _sh_n << 6;
  memcpy(_elf + _elf_sz, _sh, size);
  _elf_sz += size;
  fp = do_fopen(_outname, "wb");
  fwrite(_elf, 1, _elf_sz, fp);
  fclose(fp);
  return 0;
}
```

ДОДАТОК Б

Джерельний код гри Snake

```
/*************************************************************/
void *SDL_CreateWindow(const char *title, int x, int y
                       , int w, int h, int flags);
void *SDL_CreateRenderer(void *window, int index, int flags);
void SDL_DestroyWindow(void *window);
void SDL_DestroyRenderer(void *renderer);
int SDL_Init(int flags);
void SDL_Quit();
unsigned long SDL_GetTicks64();
int SDL_PollEvent(void *event);
void SDL_RenderPresent(void *renderer);
int SDL_RenderClear(void *renderer);
int SDL_SetRenderDrawColor(void *R, int r, int g, int b, int a);
int SDL_RenderDrawRect(void *renderer, const void *rect);
int SDL_RenderFillRect(void *renderer, const void *rect);
int rand();
/*************************************************************/
char _cells[100]; /* 10x10 */
char _event[56];
char _rect[16];
char *_event_keysym;
void *_window;
void *_renderer;
int _running;
int _apple_x;
int _apple_y;
int _head_x;
int _head_y;
int _head_dx;
int _head_dy;
int _snake_len;
int _alive;
/*************************************************************/
int get8u(char *a) {
  return *a & 0xFF;
}
/*************************************************************/
int get32(char *a) {
  char *end;
  char *t;
  int n;
  t = a + 3;
  end = a - 1;
  n = 0;
  while (t != end) {
    n <<= 8;
```

```
    n += get8u(t);
    --t;
  }
  return n;
}
/*************************************************************/
char *set32(char *a, int n) {
  char *t;
  char *end;
  t = a;
  end = a + 4;
  while (t != end) {
    *t = n & 0xFF;
    ++t;
    n >>= 8;
  }
  return a;
}
/*************************************************************/
char *cell_at(int x0, int y0) {
  return y0 * 10 + x0 + _cells;
}
/*************************************************************/
void draw_cells() {
  const char *cell;
  int x;
  int y;
  int x0;
  int y0;
  int x60;
  int y60;
  SDL_SetRenderDrawColor(_renderer, 0, 0, 0, 255);
  y = 1;
  y0 = 0;
  y60 = 0;
  cell = _cells;
  while (y <= 10) {
    x = 1;
    x0 = 0;
    x60 = 0;
    while (x <= 10) {
      set32(_rect, x60);
      set32(_rect + 4, y60);
      if (_apple_x == x0) if (_apple_y == y0) {
        SDL_SetRenderDrawColor(_renderer, 255, 64, 64, 255);
        SDL_RenderFillRect(_renderer, _rect);
        SDL_SetRenderDrawColor(_renderer, 0, 0, 0, 255);
      }
      if (*cell) {
        if (_alive == 1)
          SDL_SetRenderDrawColor(_renderer, 128, 255, 128, 255);
        if (_alive == 0)
          SDL_SetRenderDrawColor(_renderer, 170, 170, 170, 255);
```

```
        SDL_RenderFillRect(_renderer, _rect);
        SDL_SetRenderDrawColor(_renderer, 0, 0, 0, 255);
      }
      SDL_RenderDrawRect(_renderer, _rect);
      ++x;
      ++x0;
      x60 += 60;
      ++cell;
    }
    ++y;
    ++y0;
    y60 += 60;
  }
}
/*****************************************************************/
void rand_apple_pos() {
  _apple_x = rand() % 10;
  _apple_y = rand() % 10;
}
/*****************************************************************/
void respawn_apple() {
  rand_apple_pos();
  while (*cell_at(_apple_x, _apple_y))
    rand_apple_pos();
}
/*****************************************************************/
void update() {
  char *cell;
  char *end;
  int dont_shrink;
  end = _cells + 100;
  cell = _cells;
  _head_x += _head_dx;
  _head_y += _head_dy;
  if (0 - 1 == _head_x) _head_x = 9;
  if (0 - 1 == _head_y) _head_y = 9;
  if (_head_x == 10) _head_x = 0;
  if (_head_y == 10) _head_y = 0;
  if (*cell_at(_head_x, _head_y)) {
    _alive = 0;
    return;
  }
  dont_shrink = 0;
  if (_head_x == _apple_x) if (_head_y == _apple_y) {
    ++_snake_len;
    *cell_at(_head_x, _head_y) = _snake_len;
    respawn_apple();
    dont_shrink = 1;
  }
  if (dont_shrink == 0) {
    while (cell != end) {
      if (*cell) *cell -= 1;
      ++cell;
```

```
    }
    *cell_at(_head_x, _head_y) = _snake_len;
  }
}
/*************************************************************/
void redraw() {
  SDL_SetRenderDrawColor(_renderer, 255, 255, 255, 255);
  SDL_RenderClear(_renderer);
  draw_cells();
  SDL_RenderPresent(_renderer);
}
/*************************************************************/
int main() {
  unsigned long next_sec;
  int etype;
  int scancode;
  int dir;
  set32(_rect + 8, 60);
  set32(_rect + 12, 60);
  _event_keysym = _event + 16;
  SDL_Init(0x4021);
  _window = SDL_CreateWindow("Snake", 100, 100, 600, 600, 0);
  _renderer = SDL_CreateRenderer(_window, 0 - 1, 0);
  _running = 1;
  _head_x = 5;
  _head_y = 5;
  _snake_len = 3;
  *cell_at(_head_x, _head_y) = 3;
  *cell_at(_head_x + 1, _head_y) = 2;
  *cell_at(_head_x + 2, _head_y) = 1;
  dir = 0;
  next_sec = 1000;
  _alive = 1;
  _head_dx = 0 - 1;
  respawn_apple();
  redraw();
  while (_running) {
    while (SDL_PollEvent(_event)) {
      etype = get32(_event);
      if (etype == 0x100) _running = 0; /* quit */
      if (etype == 0x300) { /* keydown */
        scancode = get32(_event_keysym);
        if (scancode == 41) _running = 0; /* esc */
        if (scancode == 82) dir = 1; /* up */
        if (scancode == 79) dir = 2; /* right */
        if (scancode == 81) dir = 4; /* down */
        if (scancode == 80) dir = 8; /* left */
      }
    }
    if (next_sec <= SDL_GetTicks64()) {
      next_sec += 1000;
      if (dir == 1) { _head_dx = 0; _head_dy = 0 - 1; }
      if (dir == 2) { _head_dx = 1; _head_dy = 0; }
```

```
      if (dir == 4) { _head_dx = 0; _head_dy = 1; }
      if (dir == 8) { _head_dx = 0 - 1; _head_dy = 0; }
      if (_alive) update();
      redraw();
      dir = 0;
    }
  }
  SDL_DestroyWindow(_window);
  SDL_DestroyRenderer(_renderer);
  SDL_Quit();
  return 0;
}
```