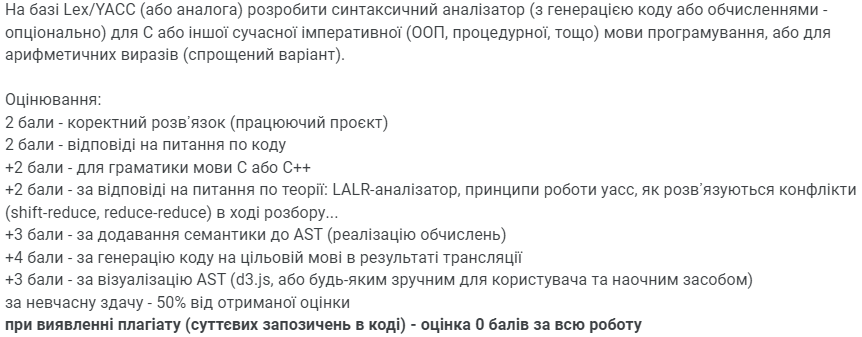
# ­Звіт до лабораторної 3

Студента групи ТТП-32

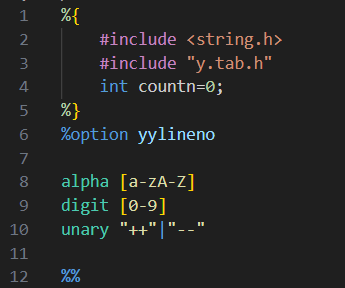
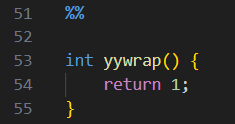
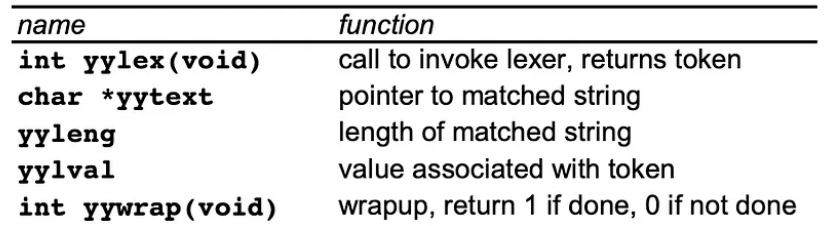
Остренка Олександра

## 1. Умова лабораторної:



## 2. Опис виконання:

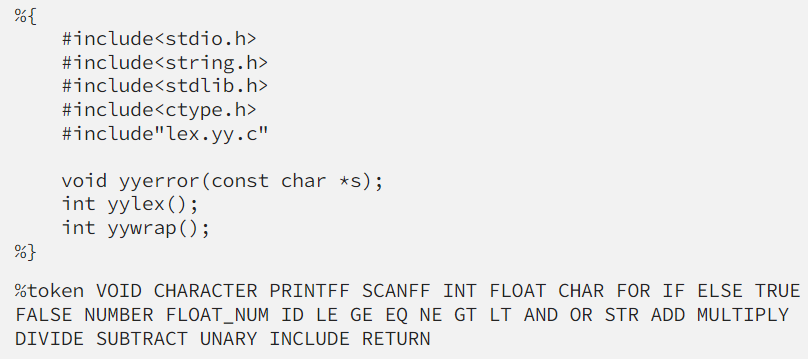
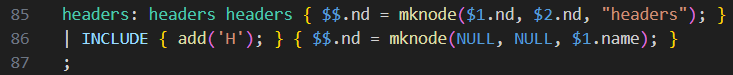
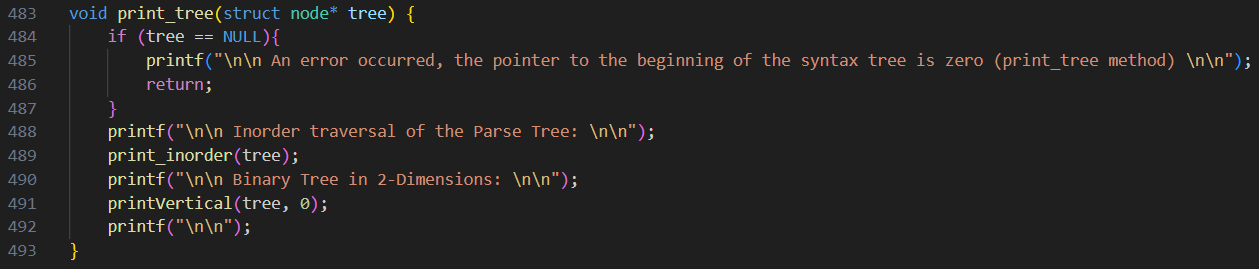
Автор пропонує природно розділити опис реалізації на 6 пунктів:

* Створення лексичного аналізатор  
  Lex — це інструмент для створення лексичного аналізатора. Відповідний файл містить набір регулярних виразів разом із діями, пов’язаними з кожним із них. Результатом є табличний сканер, який повідомляє нам, що робити, коли ми бачимо певний введений символ залежно від стану, у якому ми перебуваємо. Цей результат зберігається у файлі під назвою lex.yy.c.  
  Сам .l файл можна поділити на три частини:
  + “declarations”, там ми інклудимо усі компоненти та бібліотеки, а також означуємо необхідні змінні.  
    
  + “rules”, там ми описуємо усі правила, тобто що робити, якщо в тексті зустрічається відповідний символ\слово\конструкція. Правила задаються регулярками:  
      
    В даному випадку, якщо ми натрапимо на відповідне слово, то повернемо відповідний токін. Але крім цього бачимо, що ми щось кудись копіюємо, на даному етапі це не актуально, то знадобиться на етапі побудові синтаксичного дерева.
  + “Subroutines”, потрібен для користувацьких функцій.  
      
    В даному випадку в нас є тільки одна функція, що повертає 1, якщо лексичний аналіз відпрацював.  
    Взагалі, lex надає доступ до наступного набору функцій, які ми згодом випробуємо:  
    

### Додавання граматичних правил.

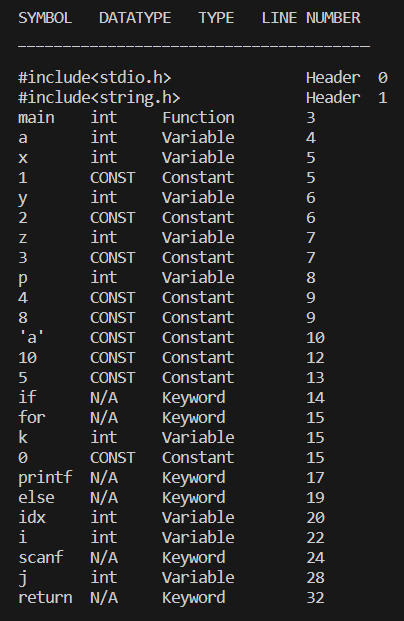
У цьому розділі наша мета полягає в тому, щоб створити файл Yacc з усіма правилами граматики, необхідними для аналізу вхідної програми C.

Yacc файл, як і lex, можна поділити на три частини:

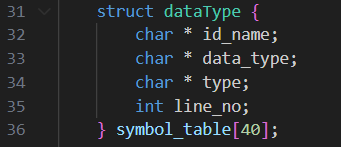
* “Declarations”:  
    
  З цікавого тут тільки %token, за яким йде купа слів. Ці слова насправді є токенами, які може прийняти файл YACC. Це зберігається в y.tab.h, який використовується програмою Lex.
* “Rules”:  
    
  Правила - це що ми можемо вивести з відповідного нетерміналу. Після правила, в дужках, можемо записати дії, що будуть виконуватись при застосуванні відповідного правилу.
* “Subroutines” - реалізація користувацьких обчислюваних функцій (головне не забувати об'являти всі функції в “Declarations”)  
    
  Приклад такої функції. Всі функції пишуться на мові С.

### Створення таблиці символів, або ж лексичний аналіз.

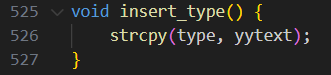
У цій частині наша мета полягає в тому, щоб створити таблицю символів і зберегти файли заголовків, змінні, ключові слова та константи разом із такими деталями, як номер рядка, тип і тип даних. Наша мета — отримати наступну таблицю символів.



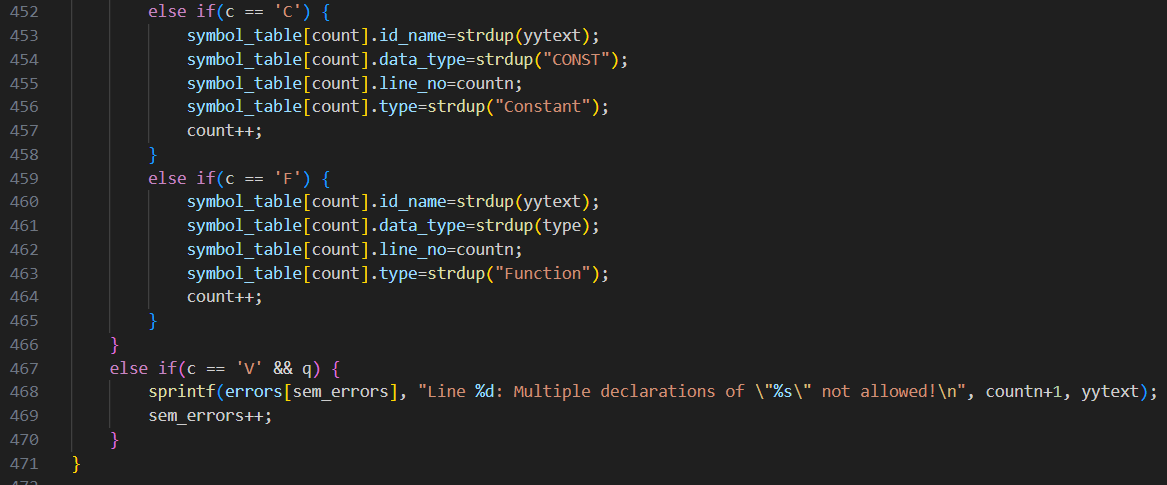
Щоб додати нашу таблицю символів, ми повинні спочатку визначити структуру та деталі. Наша таблиця символів міститиме такі деталі, як назва символу, тип даних, тип символу (ключове слово, константа, змінна тощо) і номер рядка.



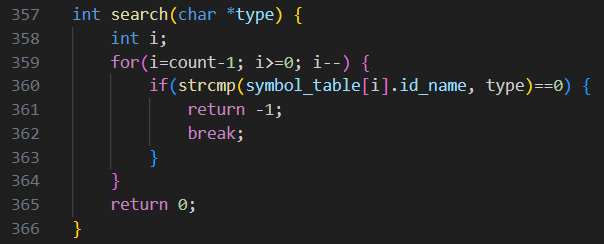
Наступна частина, яку потрібно розглянути, це функція insert\_type(). Ця функція викликається кожного разу, коли функція або змінна додається до таблиці символів. Вона копіює тип даних змінної або функції, які потрібно додати до масиву символів під назвою type.



Тепер ми підійшли до головної — додавання символів до нашої таблиці. Для досягнення цієї мети ми використовуємо функцію add.

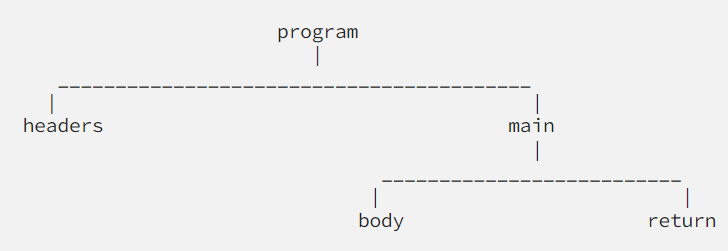
Щоб переконатися, що ми не маємо повторних входжень того самого символу в нашій таблиці, ми використовуємо функцію search.



Після того, як функції зроблені та готові, все, що нам потрібно зробити, це почати вставляти виклики функцій у відповідні місця.

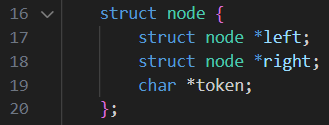
### Додавання синтаксичного дерева

Наше дерево синтаксичного аналізу буде двійковим — воно матиме двох дітей. Корінь дерева буде початком програми. Структура надзвичайно проста та інтуїтивно зрозуміла.

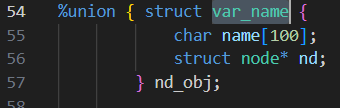
Кінцева ціль - отримати таке вертикальне дерево:  


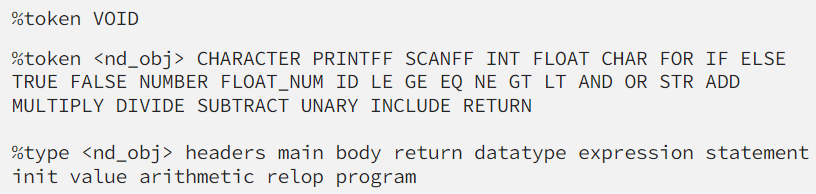
В нашому випадку воно буде виводитись не так же гарно, бо реалізація такого виду була б набагато складніше, але тим не менш зрозуміло для користувача.

Для початку об’явимо структуру node, що буде зберігати інформацію про окремий вузол.

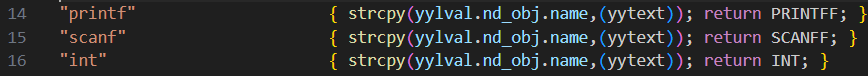


Далі потрібно змінити тип токенів, щоб вони зберігали не тільки клас токена, а й назву.





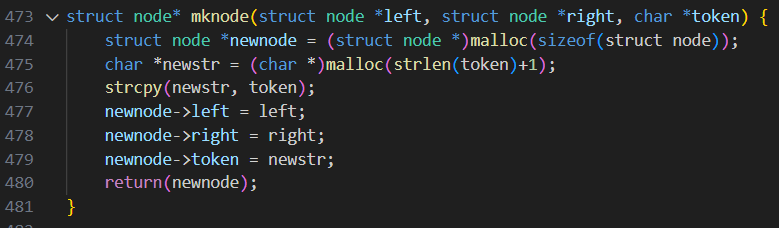
Після чого стає актуальним додати в лексері виклик функції strcpy, яка таки і буде зберігати ім’я токену.



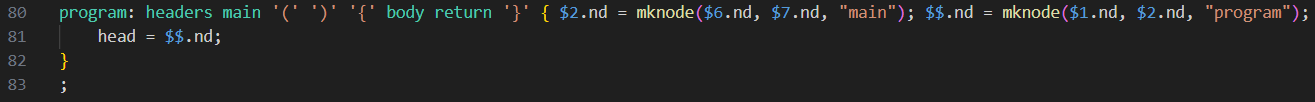
Далі залишилось правильно зберегти усі nodes і вивести їх на екран.

Корінь дерева зберігаємо у 

Функція mknode формує відповідний node.

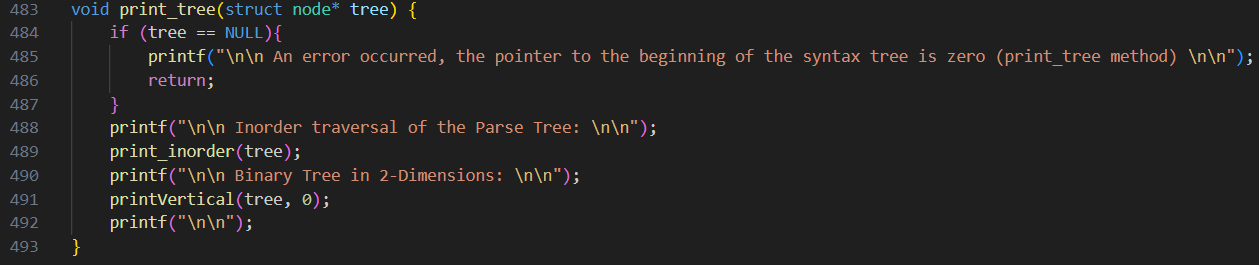
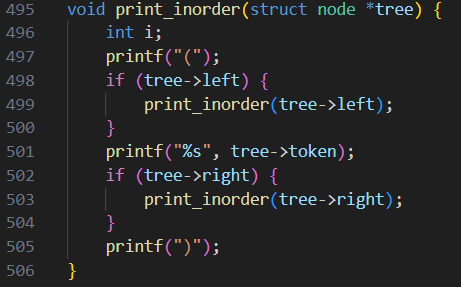


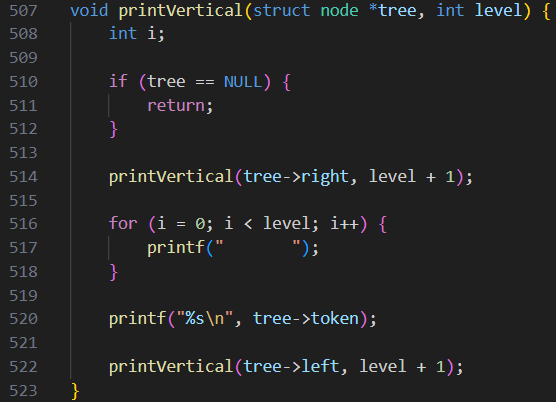
Але де цю функцію потрібно викликати?

Для доступу до main ми використовуємо $2. Для доступу до body ми використовуємо $6. Елементи індексуються від 1, а будь-які дії, які відбуваються між {}, зараховуються як один елемент. Тому body становить $6, а не $3. Ми будемо використовувати це для виклику функції mknode() і передавання продуктів як дочірніх, якщо необхідно. Щоб отримати доступ до головного терміналу, тобто program ми використовуємо $$.nd.

Ми починаємо генерувати наше дерево program. Потім ми будемо рухатися по порядку на основі правил у Parser.y.

Тут ми призначаємо main мати body та return, тоді як вузол називається «main». Програма називається «program», а її дочірніми елементами є headers і main. Оскільки це буде точка входу нашого дерева аналізу, ми призначаємо head = $$.nd.

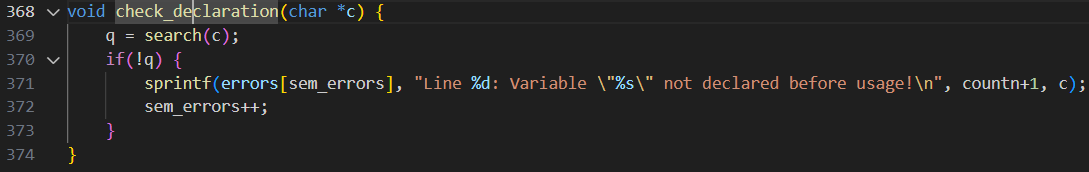
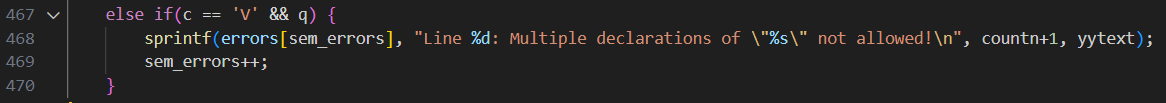
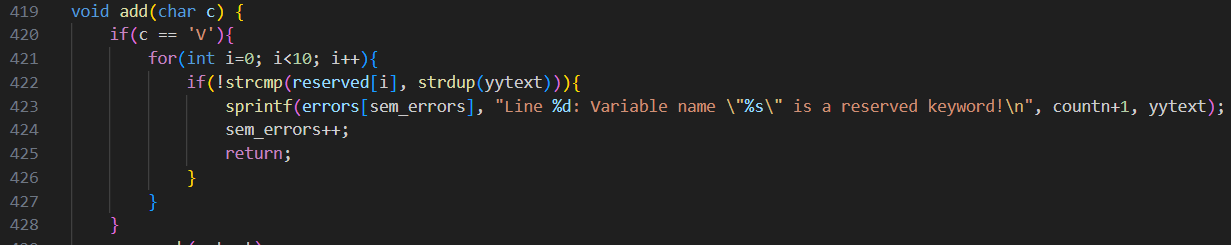
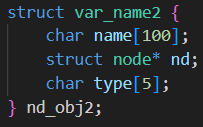
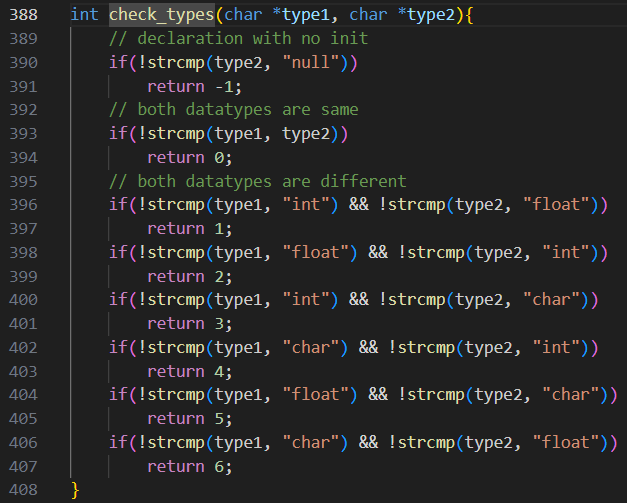
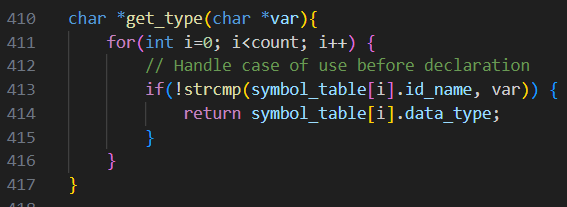
Щоб вивести усе, що ми сформували, автор використовує наступні методи:  
  




### Виконання семантичного аналізу.

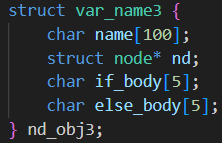
На цьому етапі буде перевірено перетворення типів, неоголошені змінні, численні оголошення змінних тощо. Ми надрукуємо помилки (якщо такі є) разом із номерами рядків.

Автор виділив чотири типи помилок:

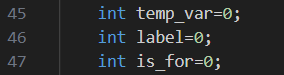
1. Змінну не оголошено.  
   
2. Кілька декларацій змінної.  
   Наш компілятор припускає, що програма C має одну область вилимості, тому змінні не можна переоголосити навіть у циклах. Для цієї перевірки функція add() змінюється, щоб перевірити, чи присутній символ у таблиці символів перед його вставленням. Якщо символ уже присутній і він належить до змінної типу, друкується повідомлення про помилку, яке інформує користувача про те, що було кілька декларацій ідентифікатора.  
   
3. Змінні не є зарезервованими словами.  
   Ми виконуємо просту перевірку, дивлячись, чи належить змінна до списку зарезервованих ключових слів. Це виконується перед тим, як ми додамо його до таблиці символів під час оголошення.  
   
4. Перевірка типу.  
   Для цього використовується функція check\_types, яка приймає типи обох змінних як вхідні дані. Якщо типи збігаються, дії не виконуються. Якщо одну змінну потрібно перетворити на інший тип, відповідний вузол перетворення типу вставляється в синтаксичне дерево (inttofloat або floattoint). Поле типу додається до структури, що представляє значення та маркери виразів, щоб відстежувати тип складених виразів. У результаті була створена нова структура під назвою nd\_obj2.  
     
     
     
     
   До того ж модифікуємо правила statement expression.

### Генерація проміжного коду.

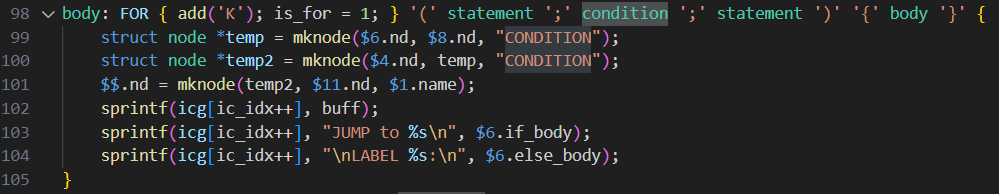
Для розбору коду вводимо ще одну структуру:

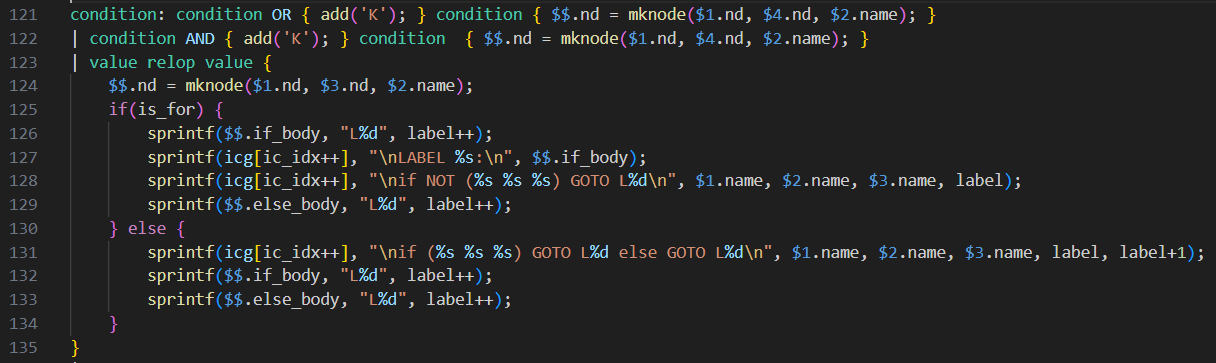

На додаток до цього, ми маємо відстежувати тимчасові змінні, які використовуються, лічильник міток і прапорець для перевірки блоку коду для або if-else.



Потім ми переходимо до друку проміжних кодів для нашої програми. Давайте подивимося на цикл for. Після цього ми можемо поширити ту саму логіку на всю програму.

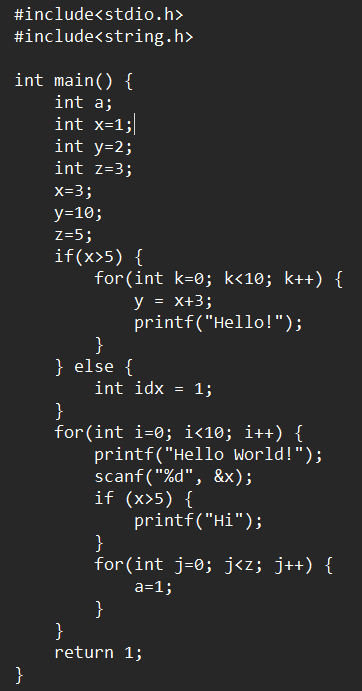
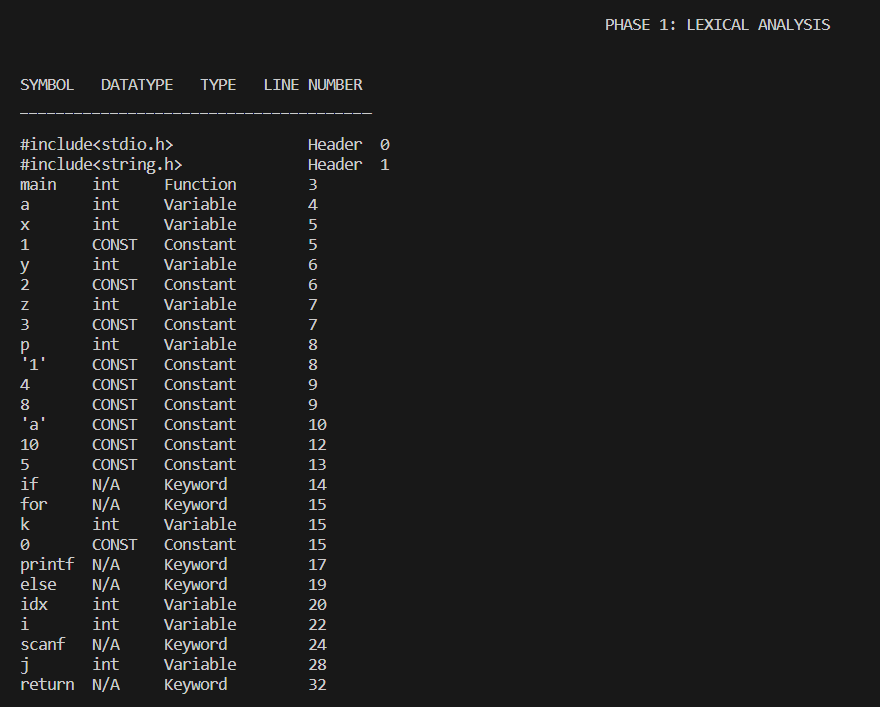


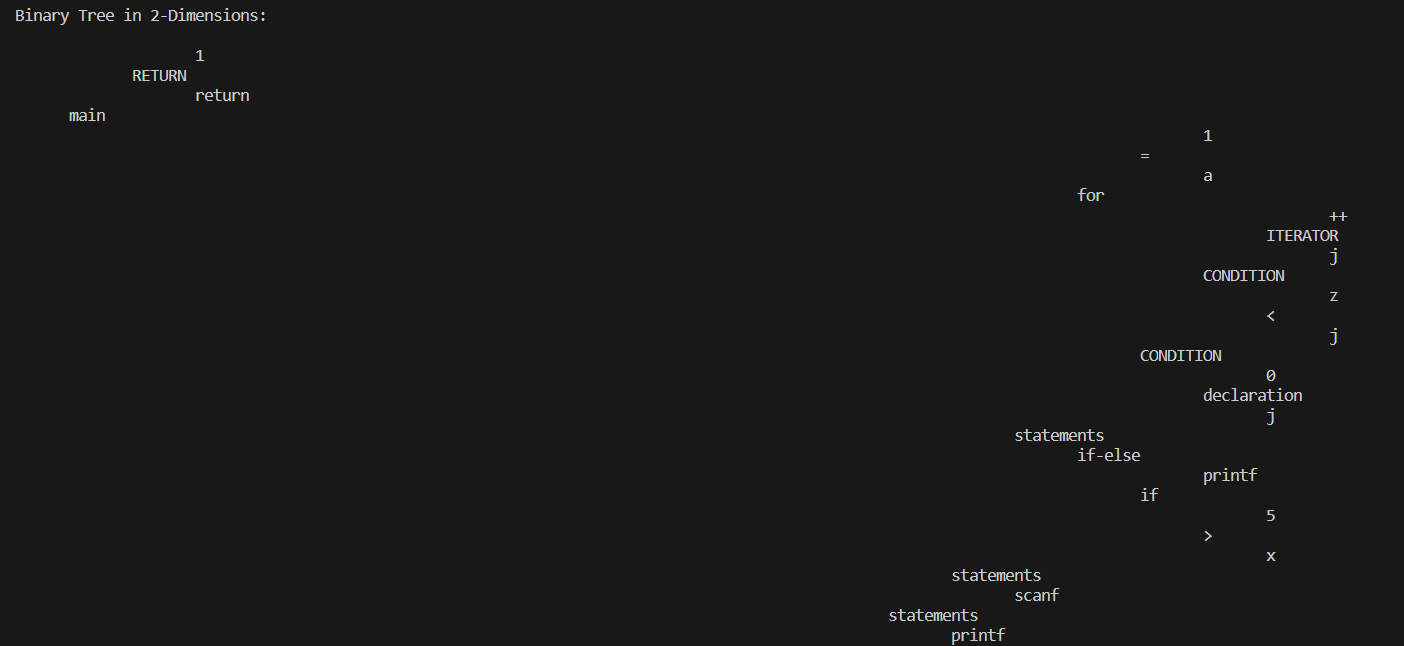
Тепер, що таке if\_body і else\_body і де вони встановлені для нашої умови? Давайте подивимось. If\_body та else\_body зберігають мітки, куди слід перенаправляти елемент керування на основі результату нашої умови.

  
Ми змінюємо виробництво умов, щоб зберігати деталі проміжного коду. If\_body та else\_body зберігають мітки, для яких необхідно перенаправити елемент керування, тоді як ми зберігаємо фактичний друк у символьному буфері під назвою icg. Тут ми використали is\_for, щоб розрізнити for та if-else, оскільки проміжний код, який потрібно згенерувати, дещо відрізняється.

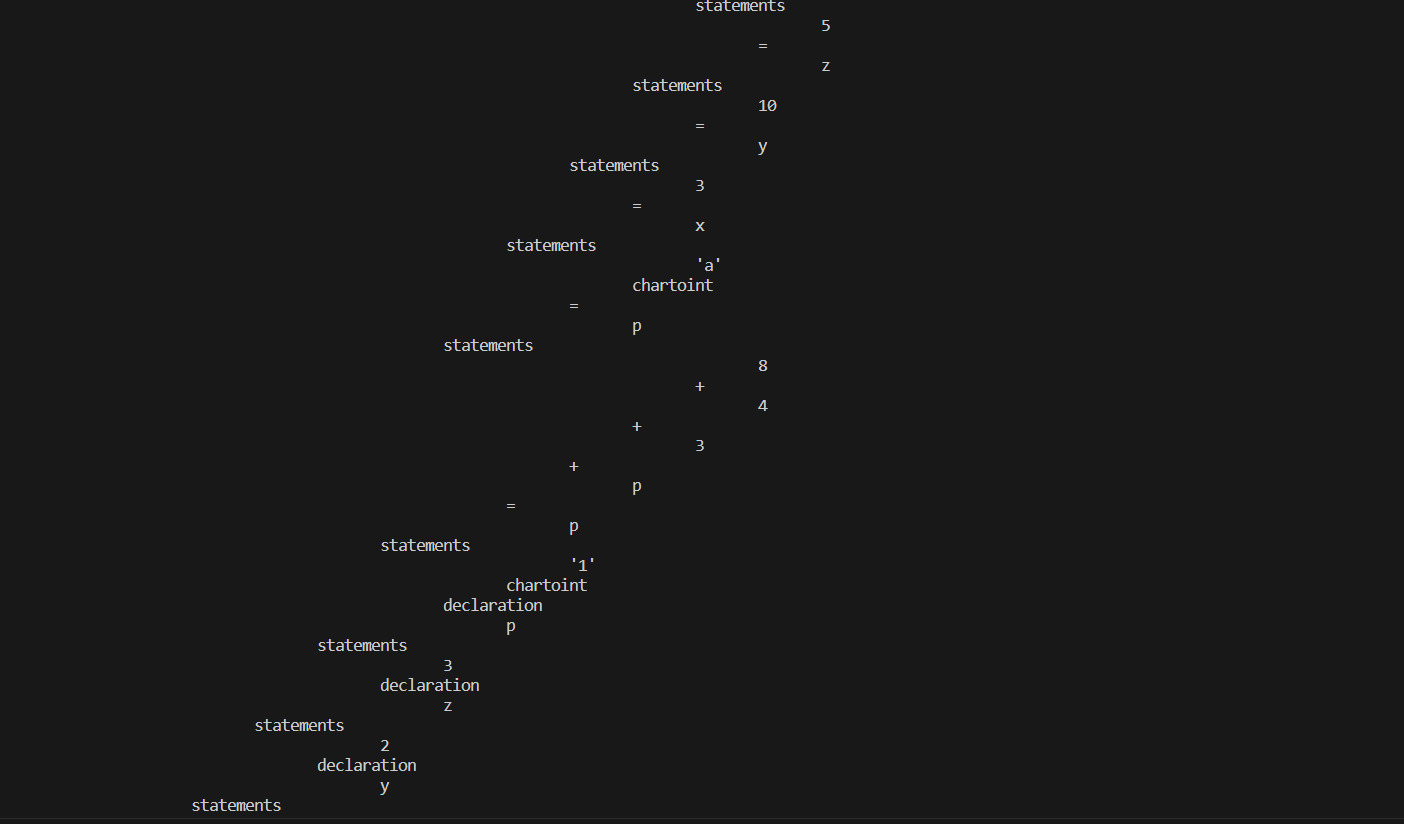
## 3. Тестування.

На вход автор подає наступний С-файл.

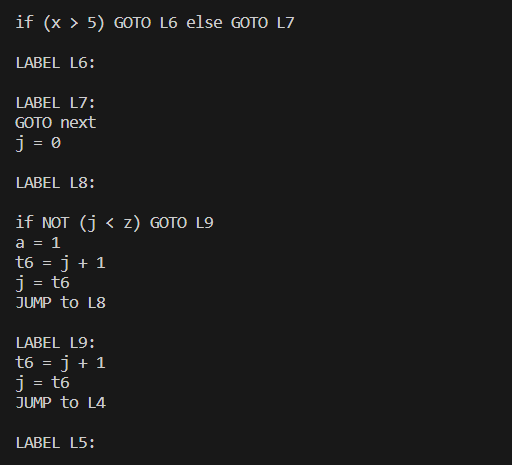
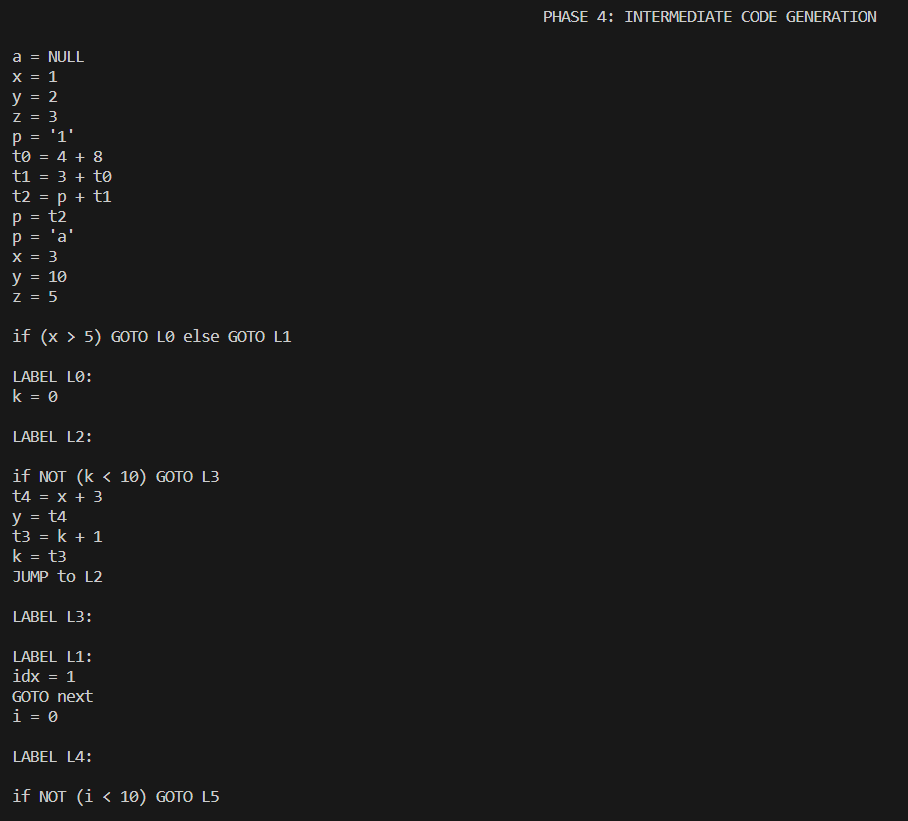












## 4. Посилання на GitHub.

[SashAmlet/C-language-compiler: A C-language compiler made using the lex-yacc parser. (github.com)](https://github.com/SashAmlet/C-language-compiler)