Міністерство освіти і науки України  
Національний університет "Львівська Політехніка"  
Кафедра СКС



Пояснювальна записка  
до курсового проєкту "СИСТЕМНЕ ПРОГРАМУВАННЯ"

на тему: “РОЗРОБКА СИСТЕМНИХ ПРОГРАМНИХ МОДУЛІВ ТА КОМПОНЕНТ

СИСТЕМ ПРОГРАМУВАННЯ”

Індивідуальне завдання 17

“РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ”

Виконав студент групи КІ-307:

Ластов’як Я.В.

Перевірив:

Козак Н.Б.

Львів-2025

ЗАВДАННЯ НА КУРСОВИЙ ПРОЄКТ

1. Цільова мова транслятора - мова програмування С або асемблер для 32/64 розрядного процесора.
2. Для отримання виконуваного файлу на виході розробленого транслятора скористатися середовищем Microsoft Visual Studio або будь-яким іншим.
3. Мова розробки транслятора: C/C++.
4. Реалізувати графічну оболонку або інтерфейс з командного рядка.
5. На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.
6. На виході розробленого транслятора мають створюватись такі файли:

*файл з лексемами;*

*файл з повідомленнями про помилки (або про їх відсутність);*

*файл на мові С або асемблера;*

*об’єктний файл; виконуваний файл.*

1. Назва вхідної мови програмування утворюється від першої букви у прізвищі студента та останніх двох цифр номера його варіанту. Саме таке розширення повинні мати текстові файли, написані на цій мові програмування.

Деталізація завдання на проектування:

1. В кожному завданні передбачається блок оголошення змінних; змінні зберігають значення цілих чисел і, в залежності від варіанту, можуть бути 16/32 розрядними. За потребою можна реалізувати логічний тип даних.
2. Необхідно реалізувати арифметичні операції - додавання, віднімання, множення, ділення, залишок від ділення; операції порівняння - перевірка на рівність і нерівність, більше і менше; логічні операції - заперечення, “логічне І” і “логічне АБО”.

Пріоритет операцій наступний - круглі дужки (), логічне заперечення, мультиплікативні (множення, ділення, залишок від ділення), адитивні (додавання, віднімання), відношення (більше, менше), перевірка на рівність і нерівність, логічне І, логічне АБО.

1. За допомогою оператора вводу можна зчитати з клавіатури значення змінної; за допомогою оператора виводу можна вивести на екран значення змінної, виразу чи цілої константи.
2. В кожному завданні обов’язковим є оператор присвоєння за допомогою якого можна реалізувати обчислення виразів з використанням заданих операцій і операції круглі дужки (); у якості операндів можуть бути цілі константи, змінні, а також інші вирази.
3. В кожному завданні обов’язковим є оператор типу “блок” (складений оператор), його вигляд має бути таким, як і блок тіла програми.
4. Необхідно реалізувати задані варіантом оператори, синтаксис операторів наведено у таблиці 1.1. Синтаксис вхідної мови має забезпечити реалізацію обчислень лінійних алгоритмів, алгоритмів з розгалуженням і циклічних алгоритмів. Опис формальної мови студент погоджує з викладачем.
5. Оператори можуть бути довільної вкладеності і в будь-якій послідовності.
6. Для перевірки роботи розробленого транслятора, необхідно написати три тестові програми на вхідній мові програмування.

Деталізований опис власної мови програмування:

Блок тіла програми: **STARTPROGRAM**

**STARTBLOK**

VARIABLE …;

ENDBLOK

Оператори вводу-виводу: **INPUT, OUTPUT**

Оператор присвоєння: **<-**

Оператори:

**IF [- ELSE ]**

**GOTO**

**FOR – TO**

**FOR-DOWNTO**

**WHILE**

**REPEAT - UNTIL**

Регістр ідентифікаторів: **Up-Low2, перший символ \_**

Операції:

* арифметичні: **+ , -, \*, DIV, MOD**
* порівняння: **=, !=, <<, >>**
* логічні: **!! , && , ||**

Тип даних: **INT16**

Коментар: { ... }

Програма на вхідній мові програмування має починатись з ключового слова **STARTPROGRAM**, далі має відкриватись блок коду програми з ключового слова **STARTBLOK**, опісля має йти розділ опису змінних **VARIABLE**. Між розділом **VARIABLE** і ключовим словом **ENDBLOK** розміщуються оператори програми. Операторів є 9: оператор вводу даних **INPUT**, оператор виводу даних **OUTPUT**, оператор присвоєння **<-,** умовний оператор **IF [- ELSE ]**, оператор безумовного переходу **GOTO** та оператори циклів: **FOR-TO**, **FOR-DOWNTO**, **WHILE**, **REPEAT-UNTIL**. Кожен оператор має завершуватись символом крапка з комою **;**. Оператор присвоєння дозволяє присвоїти деякій змінній значення арифметичного виразу. Допустимі арифметичні операції - **+ , \*, DIV, MOD**. Операндами можуть бути змінні, цілі додатні константи і інші вирази, взяті в дужки. В умовному операторі використовуються логічні вирази, допустимі такі операції порівняння **>> , << , = , !=** і такі логічні операції **!! , && , ||**. Ідентифікатори (імена змінних) можуть бути довжиною до 6-х символів, складаються лише з малих та великих латинських літер або цифр та починаються з символу **\_**. Тип даних лише один – **INT16**, при оголошення декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою **;**. Коментарі починаються з лівої фігурної дужки { та закінчуються правою } **.** .

Приклади оголошення змінних:

VARIABLE INT16 \_A;

VARIABLE INT16 \_A, \_B, \_C;

VARIABLE INT16 \_A, \_M, \_M;

Приклади ідентифікаторів:

\_B, \_M, \_N, \_A

Приклади арифметичних виразів:

0 - 75

\_Y + 68

\_A + \_B \*\_C + 76

\_C \* (\_A + \_B) - 50 DIV \_B

Приклади логічних виразів:

\_A >> \_B

\_A >> \_B && \_A >> \_C

!! (\_A << \_C )

\_A != \_B

\_M < 0 - 100

АНОТАЦІЯ

У даному курсовому проекті розроблено програмне забезпечення - транслятор з вхідної мови програмування.

Для реалізації транслятора визначено граматику вхідної мови програмування у термінах розширеної нотації Бекуса-Наура.

Реалізовано лексичний, синтаксичний, семантичний аналізатор. На етапі синтаксичного і семантичного аналізу відбувається перевірка програми на вхідній мові програмування на наявність помилок.

Перед генеруванням вихідного коду програма на вхідній мові програмування перетворюється у двійкове абстрактне синтаксичне дерево, обходячи яке генератор коду будує вихідний код на мові програмування С.

Розроблене програмне забезпечення налаштоване і протестоване на тестових прикладах.

ЗМІСТ

ВСТУП 8

1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ ТРАНСЛЯТОРІВ 9
2. [ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ 10](#bookmark19)
   1. [Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура. 10](#bookmark21)
   2. [Опис термінальних символів та ключових слів 12](#bookmark26)
3. [РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ 13](#bookmark29)
   1. [Вибір технології програмування 13](#bookmark32)
   2. [Проектування таблиць транслятора та вибір структур даних 13](#bookmark35)
   3. [Розробка лексичного аналізатора 14](#bookmark38)
      1. Розробка алгоритму роботи лексичного аналізатора 16
      2. Опис програми реалізації лексичного аналізатора 18
   4. [Розробка синтаксичного та семантичного аналізатора 19](#bookmark43)
      1. Розробка дерева граматичного розбору 21
      2. [Розробка алгоритму роботи синтаксичного і семантичного аналізатора. 22](#bookmark46)
      3. [Опис програми реалізації синтаксичного та семантичного аналізатора. . 26](#bookmark47)
   5. [Розробка генератора коду 27](#bookmark49)
      1. Розробка алгоритму роботи генератора коду 28
      2. Опис програми реалізації генератора коду 31
4. [НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ТРАНСЛЯТОРА ... 36](#bookmark53)
   1. [Опис інтерфейсу та інструкції користувачу 36](#bookmark58)
   2. [Виявлення лексичних і синтаксичних помилок 37](#bookmark61)
   3. [Перевірка роботи транслятора за допомогою тестових задач 39](#bookmark64)

ВИСНОВКИ 43

[СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ 44](#bookmark69)

[ДОДАТКИ 45](#bookmark72)

**ВСТУП**

У сучасному світі програмне забезпечення є невід’ємною складовою практично всіх сфер діяльності — від бізнесу та освіти до медицини та індустрії розваг. Розробка системних модулів і компонентів систем програмування забезпечує створення нових технологій і оптимізацію існуючих процесів. Особливе місце у цьому займають транслятори — ключові інструменти, які дозволяють перетворювати код з однієї мови програмування в іншу, забезпечуючи його виконання на різноманітних платформах і в різних середовищах.

Цей курсовий проєкт спрямований на розробку транслятора для спеціально створеної вхідної мови програмування. Мета роботи полягає у створенні програмного забезпечення, яке здатне перетворювати вихідний код цієї мови у код мовою C. Для цього передбачено реалізацію ключових компонентів: лексичного, синтаксичного й семантичного аналізаторів, а також генератора коду.

**Актуальність**

Зростаючий попит на ефективні інструменти для розробки програмного забезпечення підкреслює важливість проекту. Транслятори дозволяють значно скоротити час і зусилля, необхідні для написання та тестування програмного коду. Крім того, вони забезпечують сумісність розробленого програмного забезпечення з різними операційними системами та платформами, що особливо важливо в умовах стрімкого розвитку технологій.

**Мета проєкту**

Метою проєкту є створення транслятора, який перетворює код, написаний вхідною мовою програмування, у код на мові C. Для досягнення цієї мети будуть реалізовані три етапи аналізу — лексичний, синтаксичний і семантичний, а також модуль генерації коду. Результатом стане програмне забезпечення, здатне створювати виконувані файли на основі текстового коду, що забезпечить практичне застосування розробленої системи.

1. **ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЄКТУВАННЯ**

**ТРАНСЛЯТОРІВ**

**1.1 Лексичний аналіз**

На цьому етапі текст програми розбивається на лексеми: ключові слова, ідентифікатори, літерали, роздільники та оператори. Основні методи реалізації:

* Регулярні вирази — для опису шаблонів лексем.
* Скінченні автомати — для автоматичного розпізнавання лексем.

**1.2 Синтаксичний аналіз**

Синтаксичний аналіз перевіряє структуру програми відповідно до граматики мови та формує дерево розбору. Основні методи:

* Зверху вниз (Top-down parsing):
* Рекурсивний спуск (Recursive Descent Parsing).
* LL-аналізатори (табличні методи).
* Знизу вгору (Bottom-up parsing):
* LR-аналізатори (LR(0), SLR, LALR, Canonical LR).

**1.3 Семантичний аналіз**

Семантичний аналіз перевіряє логічну коректність програми, враховуючи:

* Відповідність типів даних.
* Область видимості змінних.
* Коректність викликів функцій.

Методи реалізації:

* Атрибутні граматики.
* Таблиці символів — для збереження інформації про ідентифікатори.

**1.4 Генерація коду**

Цей етап перетворює вихідний код у проміжний або машинний код. Методи:

- Однопрохідна генерація — для простих мов.

- Двопрохідна генерація — створення проміжного коду з подальшою оптимізацією.

- Багатопрохідна генерація — використовується для складних мов, включає додаткові етапи оптимізації.

**1.5 Оптимізація коду**

Мета оптимізації — підвищити ефективність виконання програми та зменшити її розмір. Основні підходи:

* Локальна оптимізація — у межах одного блоку коду.
* Глобальна оптимізація — враховує взаємодію між блоками.
* Машинно-незалежна оптимізація — спрощення обчислень.
* Машинно-залежна оптимізація — призначення регістрів, налаштування інструкцій.

**Класифікація трансляторів**

1. За способом роботи:

- Компілятори — перетворюють код у машинний.

- Інтерпретатори — виконують код безпосередньо.

2. За кількістю проходів:

- Однопрохідні.

- Багатопрохідні.

3. За цільовим кодом:

- Машинний код.

- Проміжний код (LLVM IR, байт-код).

**Основні підходи до проєктування**

**1. Модульний підхід**

Розділення транслятора на незалежні компоненти, такі як лексичний і синтаксичний аналізатори, генератор коду тощо.

**2. Фазовий підхід**

Чітке розділення етапів із передачею проміжних результатів між ними.

**3. Однопрохідний дизайн**

Ефективний для простих мов, коли всі етапи виконуються в одному проході.

**4. Багатопрохідний дизайн**

Використовується для складних мов і передбачає глибоку оптимізацію коду через кілька етапів аналізу й генерації.

1. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ
   1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура.

Для задання синтаксису мов програмування використовують форму Бекуса- Наура або розширену форму Бекуса-Наура — це спосіб запису правил контекстно- вільної граматики, тобто форма опису формальної мови. Саме її типово використовують для запису правил мов програмування та протоколів комунікації.

БНФ визначає скінченну кількість символів (нетерміналів). Крім того, вона визначає правила заміни символу на якусь послідовність букв (терміналів) і символів. Процес отримання ланцюжка букв можна визначити поетапно: спочатку є один символ (символи зазвичай знаходяться у кутових дужках, а їх назва не несе жодної інформації). Потім цей символ замінюється на деяку послідовність букв і символів, відповідно до одного з правил. Потім процес повторюється (на кожному кроці один із символів замінюється на послідовність, згідно з правилом). Зрештою , виходить ланцюжок , що складається з букв і не містить символів. Це означає , що отриманий ланцюжок може бути виведений з початкового символу .

Нотація БНФ є набором «продукцій», кожна з яких відповідає зразку:

<символ> = <вираз, що містить символи>

де вираз, що містить символи це послідовність символів або послідовності символів, розділених вертикальною рискою |, що повністю перелічують можливий вибір символ з лівої частини формули.

У розширеній формі нотації Бекуса — Наура вирази, що можна пропускати або які можуть повторятись слід записувати у фігурних дужках { ... }:, а можлива поява може відображатися застосуванням квадратних дужок [ ... ]:.

Опис вхідної мови програмування у термінах розширеної форми Бекуса- Наура:

program = "**STARTPROGRAM**", "**STARTBLOK**", {"**VARIABLE**", variable\_declaration, "**;**"}, {statement, "**;**"}, "**ENDBLOK**";

variable\_declaration = "**INT16**", variable\_list;

variable\_list = identifier, {"**,**", **identifier**};

identifier = "**\_**", up;

up = "**A**" | "**B**" | "**C**" | "**D**" | "**E**" | "**F**" | "**G**" | "**H**" | "**I**" | "**J**" | "**K**" | "**L**" | "**M**" | "**N**" | "**O**" | "**P**" | "**Q**" | "**R**" | "**S**" | "**T**" | "**U**" | "**V**" | "**W**" | "**X**" | "**Y**" | "**Z**";

statement = input\_statement | output\_statement | assign\_statement | if\_else\_statement | goto\_statement | label\_point | for\_statement | while\_statement | repeat\_until\_statement | compound\_statement;

input\_statement = "**INPUT**", identifier;

output\_statement = "**OUTPUT**", arithmetic\_expression;

arithmetic\_expression = low\_priority\_expression {low\_priority\_operator, low\_priority\_expression};

low\_priority\_operator = "**+**" | "**-**";

low\_priority\_expression = middle\_priority\_expression {middle\_priority\_operator, middle\_priority\_expression};

middle\_priority\_operator = "**\***" | "**DIV**" | "**MOD**";

middle\_priority\_expression = identifier | number | "**(**", arithmetic\_expression, "**)**";

number = (["**-**"], non\_zero\_digit, {digit}) | “0”;

digit = "**0**" | non\_zero\_digit ;

non\_zero\_digit = "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9”;**

assign\_statement = identifier, "**<-**", arithmetic\_expression;

if\_else\_statement = "**IF**", "**(**", logical\_expression, "**)**", statement, [ "**ELSE**", statement];

logical\_expression = and\_expression {or\_operator, and\_expression};

or\_operator = "**||**";

and\_expression = comparison {and\_operator, and\_expression};

and\_operator = "**&&**";

comparison = comparison\_expression | [not\_operator] "**(**", logical\_expression, "**)**";

not\_operator = "**!!**";

comparison\_expression = arithmetic\_expression comparison\_operator arithmetic\_expression;

comparison\_operator = "**=**" | "**!=**" | "**<<**" | "**>>**";

goto\_statement = "**GOTO**", identifier;

label\_point = identifier, "**:**";

for\_to\_statement = "**FOR**", assign\_statement, "**TO**" | "**DOWNTO**", arithmetic\_expression, "**DO**", statement;

while\_statement = "**WHILE**", logical\_expression, {statement\_in\_while}, "**END WHILE**";

statement\_in\_while = statement | “CONTINUE WHILE” | “EXIT WHILE”

repeat\_until\_statement = "**REPEAT**", {statement}, "**UNTIL**", "**(**", logical\_expression, "**)**";

compoundStatement = "**STARTBLOK**", {statement}, "**ENDBLOK**";

* 1. Опис термінальних символів та ключових слів.

Визначаємо термінальні символи і ключові слова:

* **STARTPROGRAM** - початок програми
* **STARTBLOK** - початок блоку
* **VARIABLE** - оголошення змінних
* **ENDBLOK** - кінець програми
* **INT16** - тип даних
* **INPUT** - оператор вводу
* **OUTPUT** - оператор виводу
* **IF, ELSE** - умовний оператор
* **GOTO** – оператор безумного переходу
* **FOR, TO, DO** – оператор циклу for
* **FOR, DOWNTO, DO** – оператор циклу for
* **WHILE, END, CONTINUE, EXIT** – оператор циклу while
* **REPEAT, UNTIL** – оператор циклу repeat
* **<-** - оператор присвоєння
* **+** - додавання
* **-** - віднімання
* **\*** - множення
* **DIV** - ділення
* **MOD** - остача від ділення
* **>>** - більше
* **<<** - менше
* **=** - рівність
* **!=** - нерівність
* **!!** - заперечення
* **& &** - логічне І
* **||** - логічне АБО
* **;** - кінець оператора
* **,** - розділювач змінних
* **(**- відкрита дужка
* **)** - закрита дужка
* **##** - початок коментаря
* **a...z** - маленькі латинські букви
* **0...9** - цифри
* символи табуляції, переходу на новий рядок, пробіл

1. РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ  
   ПРОГРАМУВАННЯ
   1. Вибір технології програмування.

Перед тим як розпочинати створювати програму, для більш швидкого і ефективного її написання, необхідно розробити алгоритм її функціонування, та вибрати технологію програмування, середовище програмування.

Для виконання поставленого завдання найбільш доцільно буде використати середовище програмування Microsoft Visual Studio 2022, та мову програмування C/C++.

Для якісного і зручного використання розробленої програми користувачем, було прийнято рішення створення консольного інтерфейсу.

* 1. Проектування таблиць транслятора та вибір структур даних.

Використання таблиць значно полегшує створення трансляторів, а тому створимо необхідні структури даних для зберігання інформації про лексеми:

enum TypeOfToken {

StartProgram, // STARTPROGRAM

STARTBLOK, // STARTBLOK

Variable, // VARIABLE

Type, // INT16

ENDBLOK, // ENDBLOK

Input, // INPUT

Output, // OUTPUT

If, // IF

Else, // ELSE

Goto, // GOTO

For, // FOR

To, // TO

Downto, // DOWNTO

Do, // DO

While, // WHILE

End, // END

Continue, // CONTINUE

Exit, // EXIT

Repeat, // REPEAT

Until, // UNTIL

Identifier, // Identifier

Number, // number

Float, // float (incorrect)

Assign, // ==>

Add, // +

Sub, // -

Mul, // \*

Div, // DIV

Mod, // MOD

Equality, // =

NotEquality, // !=

Greate, // >>

Less, // <<

Not, // !!

And, // &&

Or, // ||

LBracket, // (

RBracket, // )

Semicolon, // ;

Colon, // :

Comma, // ,

Unknown\_

};

struct Token {

char name[16]; // ім'я лексеми

int value; // значення (для констант)

int line; // номер рядка

enum TypeOfToken type; // тип лексеми

};

struct id {

char name[16];

//unsigned int pos; //for labels

};

У програмі будемо зберігати таблицю лексем і таблицю та кількість лексем, ідентифікаторів та міток:

struct Token\* TokenTable; // Таблиця лексем

unsigned int TokensNum; // Кількість лексем

struct id\* idTable; // Таблиця ідентифікаторів

unsigned int idNum; // кількість ідентифікаторів

struct id\* labelTable; // Таблиця міток

unsigned int labelNum; // кількість міток

* 1. Розробка лексичного аналізатора.

Основна задача лексичного аналізу - розбити вихідний текст, що складається з послідовності символів, на послідовність слів, або лексем, тобто виділити ці слова з безперервної послідовності символів. Всі символи вхідної послідовності з цієї точки зору розділяються на символи, що належать яким-небудь лексемам, і символи, що розділяють лексеми. В цьому випадку використовуються звичайні засоби обробки рядків. Вхідна програма проглядається послідовно з початку до кінця. Базові елементи, або лексичнї одиниці, розділяються пробілами, знаками операцїй i спеціальними символами (новий рядок, знак табуляції), і таким чином виділяються та розпізнаються ідентифікатори, літерали і термінальні символи (операції, ключові слова).

При виділенні лексеми вона розпізнається та записується у таблицю лексем за допомогою відповідного номера лексеми, що є унікальним для кожної лексеми із усього можливого їх набору. Це дає можливість наступним фазам компіляції звертатись лексеми не як до послідовності символів, а як до унікального номера лексеми, що значно спрощує роботу синтаксичного аналізатора: легко перевіряти належність лексеми до відповідної синтаксичної конструкції та є можливість легкого перегляду програми, як вгору, так і вниз, від текучої позиції аналізу. Також в таблиці лексем ведуться записи, щодо рядка відповідної лексеми - для місця помилки - та додаткова інформація.

Лексична фаза відкидає коментарі, оскільки вони не мають ніякого впливу на виконання програми, отже ж й на синтаксичний розбір та генерацію коду.

Розділимо лексеми на типи або лексичні класи:

• Ключові слова (STARTPROGRAM, STARTBLOK, VARIABLE, ENDBLOK, INT16, INPUT, OUTPUT, IF, ELSE, FOR, TO, DOWNTO, DO, WHILE, WEND, REPEAT, UNTIL, DIV, MOD)

* Ідентифікатори (починається з символу \_, далі велика або маленька літера або цифра, максимум 6 символи)
* Числові константи (ціле число)
* Оператор присвоєння (<-)
* Знаки операції (+, -, \*, =, !=, >>, <<, !!, &&, ||)
* Розділювачі (,, ;)
* Дужки (**(,** **)**)
* Невідома лексема (символи і ланцюжки символів, які не підпадають під вищеописані правила).
  + 1. Розробка алгоритму роботи лексичного аналізатора.

Розробимо алгоритм роботи лексичного аналізатора на основі скінченного автомату. Лексичний аналізатор працює за принципом скінченного автомату з такими станами:

* **Start** - початок виділення чергової лексеми;
* **Finish** - кінець виділення чергової лексеми;
* **EndOfFile** - кінець файлу, завершення розпізнавання лексем;
* **Letter** - перший символ буква, розпізнавання слів (ключові слова і ідентифікатори);
* **Digit** - перший символ цифра, розпізнавання числових констант;
* **Separators** - видалення пробілів, символів табуляції і переходу на новий рядок;
* **Comment** - видалення тексту коментаря;
* **Another** - опрацювання інших символів.

У стані **Letter** читаємо по одному символи з файлу і виділяємо ланцюжок символів, який починається з букви чи символу \_, а далі можуть слідувати букви або цифри. Кінець ланцюжка - якщо прочитаний символ відмінний від букви чи цифри. Виділений ланцюжок порівнюємо з ключовими словами, якщо співпадінь немає, вважаємо його ідентифікатором при умові, що довжина ланцюжка не більше 2-х символів і ланцюжок складається з великих літер чи цифр, інакше це невизначена лексема. Переходимо до стану **Finish**.

У стані **Digit** читаємо по одному символи з файлу і виділяємо ланцюжок символів, який починається з крапки, мінуса або ж цифри, далі ж йдуть лише цифри або крапки, вважаємо цей ланцюжок числовою константою. Кінець ланцюжка - якщо прочитаний символ відмінний від цифри. Переходимо до стану **Finish**.

У стан **Comment** ми потрапляємо, якщо натикаємось на фігурну дужку, далі, перебуваючи у стані **Comment,** читаємо символи, поки не зустрінеться символ } після цього переходимо до виділення нової лексеми - до стану **Start**.

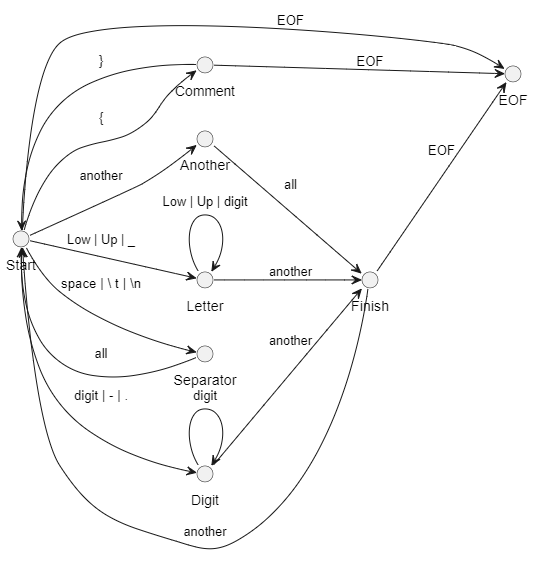
У стані **Separators** читаємо наступний символ і переходимо до виділення нової лексеми - до стану **Start**. Тобто пропускаємо усі пробіли, символи табуляції і переходу на новий рядок.

У стані **Another** порівнюємо поточний прочитаний символ з символами, що позначають знаки операцій, розділювачі і круглі дужки і визначаємо одну з лексем. Є кілька лексем, які вимагають ще читання наступного символу з файлу - це оператор присвоєння “<-” і операцій “!=”, “!!”, “||”, “&&’, “<<”, “>>”. Якщо співпадіння не виявлено, то поточний символ - невідома лексема, читаємо наступний символ і переходимо до стану **Finish**.

У стані **Finish** записуємо поточну лексему у таблицю лексем і переходимо до виділення нової лексеми, до стану **Start**.

У стані **EndOfFile** завершуємо обробку вхідного файлу, усі символи з файлу прочитані, усі лексеми записані у таблицю лексем.

Алгоритм роботи лексичного аналізатора можна зобразити у вигляді граф- схеми.

**

*Рис. 3.1. Граф-схема алгоритму роботи лексичного аналізатора.*

* + 1. Опис програми реалізації лексичного аналізатора.

В даному курсовому проекті реалізовано прямий лексичний аналізатор, який виділяє з вхідного тексту програми окремі лексеми і на основі цього формує таблицю лексем.

Створимо структуру даних для зберігання стану аналізатора:

enum States {

Start, // початковий стан

Finish, // фінальний стан

Letter, // опрацювання слів (ключові слова та ідентифікаторів)

Digit, // опрацювання цифр

Separator, // опрацювання роздільників

Another, // опрацювання інших символів

EndOFile, // кінець файлу

Comment // ігнорування коментаря

};

Напишемо функцію, яка реалізує лексичний аналіз:

unsigned int getTokens(FILE\* F);

І функції, які друкують список лексем:

void printTokens(void);

void fprintTokens(FILE\* F);

* 1. Розробка синтаксичного та семантичного аналізатора.

Синтаксичний аналіз - це процес, що визначає, чи належить деяка послідовність лексем граматиці мови програмування. В принципі, для будь-якої граматики можна побудувати синтаксичний аналізатор, але граматики, які використовуються на практиці, мають спеціальну форму. Наприклад, відомо, що для будь-якої контекстно-вільної граматики може бути побудований аналізатор, складність якого не перевищує O(n3) для вхідного рядка довжиною n, але в більшості випадків для заданої мови програмування ми можемо побудувати таку граматику, що дозволить сконструювати і більш швидкий аналізатор.

Аналізатори реальних мов зазвичай мають лінійну складність; це досягається за рахунок перегляду вхідної програми зліва направо із загляданням уперед на один термінальний символ (лексичний клас).

Вхід синтаксичного аналізатора - це послідовність лексем і таблиці представлень, які є виходом лексичного аналізатора.

На виході синтаксичного аналізатора отримуємо дерево граматичного розбору і таблиці ідентифікаторів та типів, які є входом для наступного перегляду компілятора (наприклад, це може бути перегляд, який здійснює контроль типів - семантичний аналіз).

* + 1. Розробка дерева граматичного розбору.

Схема дерева розбору виглядає наступним чином:

Program

├── "STARTPROGRAM"

├── "STARTBLOK"

├── {VariableDeclaration ";"}

│ ├── VariableDeclaration

│ │ ├── "INT16"

│ │ └── VariableList

│ │ ├── Identifier

│ │ │ ├── "\_"

│ │ │ └── Up

│ │ │ └── "A" | "B" | ... | "Z"

│ │ └── {"," Identifier}

├── {Statement ";"}

│ ├── Statement

│ │ ├── InputStatement

│ │ │ ├── "INPUT"

│ │ │ └── Identifier

│ │ ├── OutputStatement

│ │ │ ├── "OUTPUT"

│ │ │ └── ArithmeticExpression

│ │ │ ├── LowPriorityExpression

│ │ │ │ ├── MiddlePriorityExpression

│ │ │ │ │ ├── Identifier

│ │ │ │ │ ├── Number

│ │ │ │ │ │ ├── ["-"]

│ │ │ │ │ │ ├── NonZeroDigit

│ │ │ │ │ │ └── {Digit}

│ │ │ │ │ └── "(" ArithmeticExpression ")"

│ │ │ │ └── {MiddlePriorityOperator MiddlePriorityExpression}

│ │ │ └── {LowPriorityOperator LowPriorityExpression}

│ │ ├── AssignStatement

│ │ │ ├── Identifier

│ │ │ ├── "<-"

│ │ │ └── ArithmeticExpression

│ │ ├── IfElseStatement

│ │ │ ├── "IF"

│ │ │ ├── "(" LogicalExpression ")"

│ │ │ │ ├── AndExpression

│ │ │ │ │ ├── Comparison

│ │ │ │ │ │ ├── ComparisonExpression

│ │ │ │ │ │ │ ├── ArithmeticExpression

│ │ │ │ │ │ │ ├── ComparisonOperator

│ │ │ │ │ │ │ └── ArithmeticExpression

│ │ │ │ │ │ └── [NotOperator] "(" LogicalExpression ")"

│ │ │ │ │ └── {AndOperator AndExpression}

│ │ │ │ └── {OrOperator AndExpression}

│ │ │ ├── Statement

│ │ │ └── ["ELSE" Statement]

│ │ ├── GotoStatement

│ │ │ ├── "GOTO"

│ │ │ └── Identifier

│ │ ├── LabelPoint

│ │ │ ├── Identifier

│ │ │ └── ":"

│ │ ├── ForToStatement

│ │ │ ├── "FOR"

│ │ │ ├── AssignStatement

│ │ │ ├── "TO" | "DOWNTO"

│ │ │ ├── ArithmeticExpression

│ │ │ ├── "DO"

│ │ │ └── Statement

│ │ ├── WhileStatement

│ │ │ ├── "WHILE"

│ │ │ ├── LogicalExpression

│ │ │ ├── {StatementInWhile}

│ │ │ └── "END WHILE"

│ │ │ ├── Statement

│ │ │ ├── "CONTINUE WHILE"

│ │ │ └── "EXIT WHILE"

│ │ ├── RepeatUntilStatement

│ │ │ ├── "REPEAT"

│ │ │ ├── {Statement}

│ │ │ └── "UNTIL" "(" LogicalExpression ")"

│ │ └── CompoundStatement

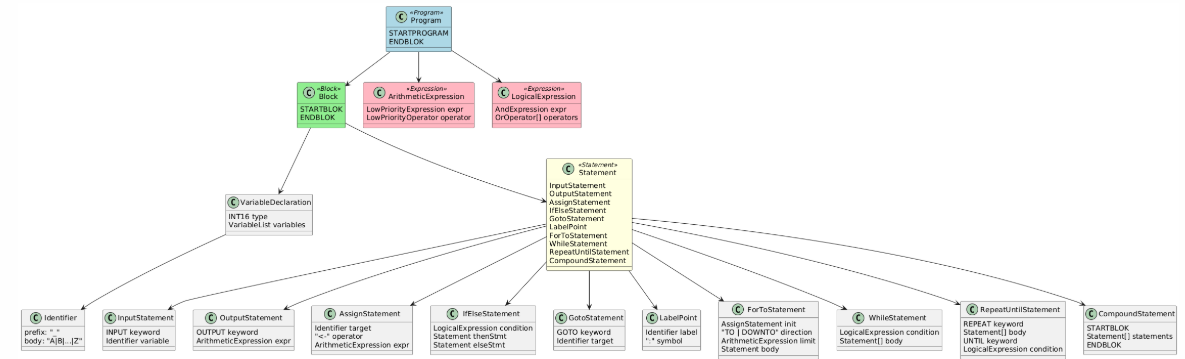
│ │ ├── "STARTBLOK"

│ │ ├── {Statement}

│ │ └── "ENDBLOK"

└── "ENDBLOK"

*Рис. 3.2. Дерево граматичного розбору.*



Рисю 3.3 Дерево граматичного розбору

3.4.2. Розробка алгоритму роботи синтаксичного і семантичного аналізатора.

Одним з найбільш простих і найбільш популярних методів низхідного синтаксичного аналізу є метод рекурсивного спуску (recursive descent method).

Метод заснований на тому, що в склад синтаксичного аналізатора входить множина рекурсивних процедур граматичного розбору, по одній для кожного правила граматики.

Визначимо назви процедур, що відповідають нетерміналам граматики таким чином:

// program :: = "STARTPROGRAM", "STARTBLOCK", { "VARIABLE", variable\_declaration, ";" }, { statement }, "ENDBLOCK";

void program();

// programBody ::= { statement };

void programBody();

// variable\_declaration ::= "INT16", variable\_list;

void variableDeclaration();

// variable\_list :: = identifier, { ",", identifier };

void variableList();

// statement ::= input\_statement | output\_statement | assign\_statement | if\_else\_statement | goto\_statement | label\_point | for\_statement | while\_statement | repeat\_until\_statement | compound\_statement;

void statement();

// input\_statement ::= "INPUT", identifier;

void inputStatement();

// output\_statement ::= "OUTPUT", arithmetic\_expression;

void outputStatement();

// arithmetic\_expression :: = low\_priority\_expression{ low\_priority\_operator, low\_priority\_expression };

void arithmeticExpression();

// low\_priority\_expression :: = middle\_priority\_expression{ middle\_priority\_operator, middle\_priority\_expression };

void lowPriorityExpression();

// low\_priority\_operator :: = "+" | "-";

//void lowPriorityOperator();

// middle\_priority\_expression :: = identifier | number | "(", arithmetic\_expression, ")";

void middlePriorityExpression();

// middle\_priority\_operator :: = "\*" | "DIV" | "MOD";

//void middlePriorityOperator();

// assign\_statement :: = arithmetic\_expression, "<-", identifier;

void assignStatement();

// if\_else\_statement :: = "IF", "(", logical\_expression, ")", statement, [ "ELSE", statement];

void ifStatement();

// logical\_expression :: = and\_expression{ or\_operator, and\_expression };

void logicalExpression();

// or\_operator :: = "||";

//void orOperator();

// and\_expression :: = comparison{ and\_operator, and\_expression };

void andExpression();

// and\_operator :: = "&&";

// void andOperator();

// comparison :: = comparison\_expression | [not\_operator] "(", logical\_expression, ")";

void comparison();

// not\_operator :: = "!!";

// void notOperator();

// comparison\_expression :: = arithmetic\_expression comparison\_operator arithmetic\_expression;

void comparisonExpression();

// comparison\_operator :: = "=" | "!=" | "<<" | ">>";

// void comparisonOperator();

// goto\_statement :: = "GOTO", identifier;

void gotoStatement();

// label\_point :: = identifier, ":";

void labelPoint();

// for\_to\_statement :: = "FOR", assign\_statement, "TO" | "DOWNTO", arithmetic\_expression, "DO", statement;

void forStatement();

// while\_statement :: = "WHILE", logical\_expression, { statement }, "END", "WHILE";

void whileStatement();

void statementInWhile();

// repeat\_until\_statement :: = "REPEAT", { statement }, "UNTIL", "(", logical\_expression, ")";

void repeatStatement();

// compoundStatement :: = "STARTBLOCK", { statement }, "ENDBLOCK";

void compoundStatement();

Блок-схема алгоритму роботи синтаксичного аналізатора виглядатиме наступним чином:



Рис. 3.4. Блок-сема алгоритму роботи синтаксичного аналізатора.

Синтаксичний аналізатор буде читати лексеми з таблиці лексем і аналізувати їх. Нам знадобиться допоміжна функція

void match(enum TypeOfToken expectedType);

яка перевіряє, чи поточна лексема збігається з очікуваною.

Блок-схема алгоритму функції variable\_declaration(), яка перевіряє чи правильно описані змінні виглядає наступним чином:



Рис. 3.5. Блок-сема алгоритму роботи функції variableDeclaration().

Блок-схема алгоритму функції programBody(), яка перевіряє чи правильно написані оператори вхідної мови програмування зображена на рисунку 3.5.

У наведених блок-схемах використано позначення ПЛ, яке позначає поточну лексему та НЛ, що позначає наступну лексему. У процесі перегляду таблиці лексем, після аналізу поточної лексеми, необхідно переміщатися на наступну лексему.



Рис. 3.6. Блок-сема алгоритму роботи функції programBody().

На етапі семантичного аналізу нам необхідно вирішити задачу ідентифікації ідентифікаторів. Алгоритм ідентифікації складається з двох частин:

* перша частина алгоритму опрацьовує оголошення ідентифікаторів;
* друга частина алгоритму опрацьовує використання ідентифікаторів.

Опрацювання оголошення ідентифікатора. Нехай лексичний аналізатор видав чергову лексему, що є ідентифікатором. Лексичний аналізатор сформував структуру, що містить атрибути виділеної лексеми, такі як ім’я ідентифікатора, його тип і лексичний клас. Далі вся ця інформація передається семантичному аналізатору. Припустимо, що в даний момент опрацьовується оголошення ідентифікатора. Основна семантична дія в цьому випадку полягає в занесенні інформації про ідентифікатор у таблицю ідентифікаторів.

Опрацювання використання ідентифікатора. Припустимо, що уже побудовано (цілком чи частково) таблицю ідентифікаторів. Далі вся ця інформація передається фазі використання ідентифікаторів. Таким чином, відомо, що опрацьовується використання ідентифікатора. Для того, щоб одержати інформацію про тип ідентифікатора нам достатньо прочитати певне поле таблиці ідентифікаторів.

3.4.3. Опис програми реалізації синтаксичного та семантичного аналізатора.

Структура синтаксичного аналізатора буде такою:

*//* Вхідна таблиця лексем

extern Token\* TokenTable;

int pos = 0;

void Parser() {

program();

printf("\nThe program is syntax correct.\n");

fprintf(errorFile, "\nThe program is syntax correct.\n");

}

Синтаксичний аналізатор працює за методом рекурсивного спуску, а отже функція parser() викликає функцію program(), яка в свою чергу викликає інші функції.

Семантичний аналіз у нашому випадку буде реалізований у функції, яка викликає функції, що отримують списки ідентифікаторів та міток:

void Semantic() {

idNum = IdIdentification(idTable, TokenTable, TokensNum);

labelNum = LabelIdentification(labelTable, TokenTable, TokensNum);

printf("\nThe program is semantic correct.\n");

printf("\n%d labels found\n", labelNum);

fprintf(errorFile, "\nThe program is semantic correct.\n");

fprintf(errorFile, "\n%d labels found\n", labelNum);

printIdentifiers(labelNum, labelTable);

fprintIdentifiers(errorFile, labelNum, labelTable);

printf("\n%d identifiers found\n", idNum);

fprintf(errorFile, "\n%d identifiers found\n", idNum);

printIdentifiers(idNum, idTable);

fprintIdentifiers(errorFile, idNum, idTable);

}

3.5. Розробка генератора коду.

Генерація вихідного коду передбачає спочатку перетворення програми у якесь проміжне представлення, а тоді вже генерацію з проміжного представлення у вихідний код. У якості проміжного представлення виберемо абстрактне синтаксичне дерево.

Абстрактне синтаксичне дерево (AST) — це структура даних, яка представляє синтаксичну структуру вихідного коду програми у вигляді дерева. AST використовується в компіляторах, інтерпретаторах та інструментах статичного аналізу для обробки коду.

AST представляє тільки важливу для аналізу і виконання інформацію, ігноруючи зайві деталі (наприклад, круглі дужки чи крапки з комою). Це спрощений, але точний опис логіки програми.

Вузли дерева представляють конструкції мови програмування (оператори, вирази, змінні, функції тощо). Гілки відповідають підконструкціям або елементам цих конструкцій.

Кожен вузол відповідає певному типу конструкції коду (наприклад, оператору додавання, виклику функції, оголошенню змінної).

AST є спрощеною версією синтаксичного дерева. Воно не включає зайві вузли, що відповідають елементам, які не впливають на логіку програми (наприклад, дужки чи крапки з комою).

* + 1. Розробка алгоритму роботи генератора коду.

Будемо використовувати бінарні дерева, а отже вузол у нас має два нащадки, відповідно нарисуємо типові варіанти побудови дерева.

Програма має вигляд:

Program

/ \

var statement

Оголошення змінних:

var

/ \

Id var

/ \

Id null

Тіло програми:

Statement

/ \

Statement Оператор

/ \

statement Оператор

Оператор вводу:

Input

/ \

Id null

Оператор виводу:

Output

/ \

Id null

Також оператор виводу може мати за лівого нащадка різні арифметичні вирази, наприклад:

Output

/ \

Add null

/ \

Id num

Умовний оператор (IF() оператор;):

If

/ \

Умова оператор

Умовний оператор (IF() оператор1; else оператор2;):

If

/ \

Умова else

/ \

Оператор1 оператор2

Оператор безумовного переходу:

Goto

/ \

Id null

Оператор циклу for:

For

/ \

(to | downto) оператор

/ \

Оп. Прис. ариф. вир.

Оператор циклу while:

While

/ \

Умова statement

/ \

Statement оператор

/ \

Оператор оператор

Оператор циклу repeat:

Repeat

/ \

Statement умова

/ \

Оператор оператор

Оператор присвоєння:

<-

/ \

Id арифметичний вираз

Арифметичний вираз:

(+ або -)

/ \

Id id

Доданок:

(\*, DIV або MOD)  
/ \  
множник множник

Множник:

фактор  
/ \

id або number або (арифм. вираз) null

Складений оператор:

compount

/ \

statement null

Генератор коду буде обходити створене дерево і, маючи усію необхідну інформацію, генерувати вихідний код на мові програмування С у текстовий файл. Кожен вузол у дереві буде позначати якусь конструкцію, для якої генерується певний код на мові програмування С. Опрацювання кожного з вузлів дерева передбачає рекурсивний виклик функції генерування коду для лівого і правого нащадків.

Блок-схема алгоритму роботи генератора коду зображена на рисунку 3.6.

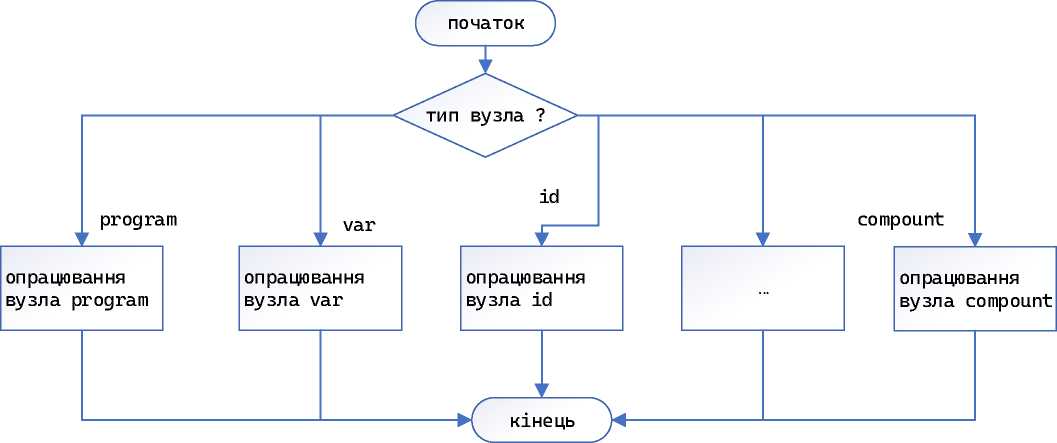


Рис. 3.7. Блок-сема алгоритму роботи генератора коду.

Розглянемо на прикладі вузла program детальніше алгоритм обходу дерева, який зображено на рисунку 3.7. Вузол позначає програму, зліва будемо зберігати інформацію про оголошені змінні, справа про оператори програми. Опрацювання вузла полягає у друці у файл необхідних шаблонів на мові програмування С, а також рекурсивного виклику для опрацювання лівого і правого нащадків. Лівий нащадок - оголошення змінних (вузол var), правий - тіло програми (вузол statement).

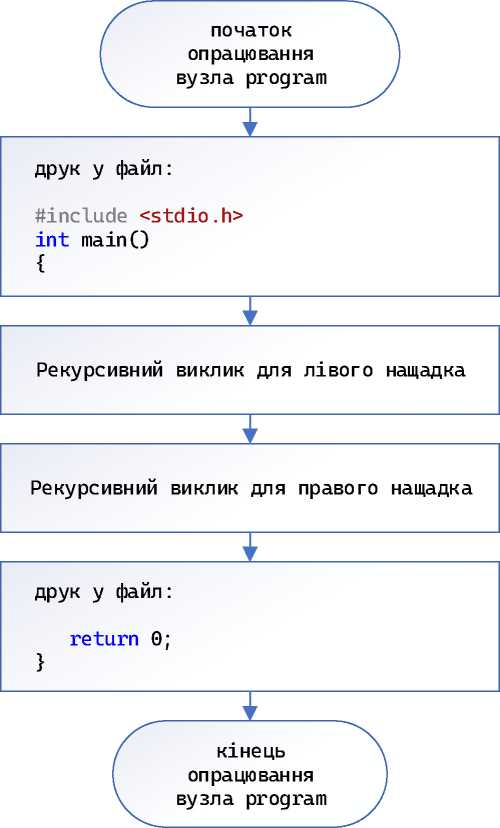


Рис. 3.8. Блок-сема алгоритму опрацювання вузла program.

* + 1. Опис програми реалізації генератора коду.

Генерувати вихідний код будемо з абстрактного синтаксичного дерева.

Створимо таку структуру даних для зберігання вузлів дерева:

enum TypeOfNode {

program\_node,

id\_node,

var\_node,

statement\_node,

input\_node,

output\_node,

add\_node,

sub\_node,

mul\_node,

div\_node,

mod\_node,

number\_node,

assign\_node,

if\_node,

else\_node,

or\_node,

and\_node,

not\_node,

eq\_node,

neq\_node,

gr\_node,

ls\_node,

goto\_node,

label\_node,

for\_node,

to\_node,

downto\_node,

while\_node,

continue\_node,

exit\_node,

repeat\_node,

compound\_node

};

struct astNode {

enum TypeOfNode type;

char name[16];

struct astNode\* left;

struct astNode\* right;

};

Функція створення вузла дерева:

// функція створення вузла AST

struct astNode\* createNode(enum TypeOfNode type, const char\* name, struct astNode\* left, struct astNode\* right);

Функція створення абстрактного синтаксичного дерева реалізована методом рекурсивного спуску:

// функція синтаксичного аналізу і створення абстрактного синтаксичного дерева

struct astNode\* astParser() {

pos = 0;

struct astNode\* tree = program();

printf("AST created.\n");

return tree;

}

Також напишемо функції для друку абстрактного синтаксичного дерева:

*//* функція для друку AST у вигляді дерева на екран

void printAST(struct astNode\* node, int level)

// функція для друку AST у вигляді дерева у файл

void fPrintAST(FILE\* outFile, struct astNode\* node, int level);

Напишемо рекурсивну функцію, яка буде генерувати код на мові програмування С з абстрактного синтаксичного дерева:

// Рекурсивна функція для генерації коду з AST

void codegen(FILE\* outFile, struct astNode\* node){

if (node == NULL) return;

switch (node->type) {

case ...

case ...

default: {

exit(1);

printf("Undescribed node type: %d\n", node->type);

break;

}

}

Тепер розглянемо варіанти генерації коду для можливих вузлів дерева.

Отже опрацювання вузла program\_node буде виглядати таким чином:

{

fprintf(outFile, "#include <stdio.h>\n");

fprintf(outFile, "int main() {\n");

codegen(outFile, node->left); // for declaration

fprintf(outFile, "\n");

codegen(outFile, node->right); // for statements

fprintf(outFile, "return 0;\n}\n");

break;

}

При генерації вихідного коду для блоку оголошення змінних будемо опрацьовувати вузли var\_node:

{

fprintf(outFile, "int ");

codegen(outFile, node->left);

fprintf(outFile, ";\n");

codegen(outFile, node->right);

break;

}

Опрацювання вузлів id\_node і number\_node буде полягати у друці імені вузла (ім’я ідентифікатора):

{

fprintf(outFile, "%s", node->name);

break;

}

При генерації вихідного коду для блоку тіло програми будемо опрацьовувати вузли statement\_node:

{

codegen(outFile, node->left);

codegen(outFile, node->right);

break;

}

Опрацювання вузла input\_node буде виглядати таким чином:

{

fprintf(outFile, "printf(\"Enter ");

codegen(outFile, node->left);

fprintf(outFile, ": \");\n");

fprintf(outFile, "scanf(\"%%d\", &");

codegen(outFile, node->left);

fprintf(outFile, ");\n");

break;

}

А вузла output\_node буде виглядати таким чином:

{

fprintf(outFile, "printf(\"%%d\\n\", ");

codegen(outFile, node->left);

fprintf(outFile, ");\n");

break;

}

Опрацювання вузла if\_node буде виглядати таким чином:

{

fprintf(outFile, "if (");

codegen(outFile, node->left);

fprintf(outFile, ") ");

codegen (outFile, node->right);

break;

}

Опрацювання вузла else\_node буде виглядати таким чином:

{

codegen(outFile, node->left);

fprintf(outFile, "else ");

codegen(outFile, node->right);

break;

}

Отже опрацювання вузла assign\_node буде виглядати таким чином:

{

codegen(outFile, node->left);

fprintf(outFile, " = ");

codegen(outFile, node->right);

fprintf(outFile, ";\n");

break;

}

Отже опрацювання вузла not\_node буде виглядати таким чином:

{

fprintf(outFile, "!(");

codegen(outFile, node->left);

fprintf(outFile, ")");

break;

}

Опрацювання вузлів or\_node, and\_node, eq\_node, neq\_node, gr\_node, ls\_node, а також вузлів add\_node, sub\_node, mul\_node, div\_node, mod\_node буде полягати у друці знаку операції і круглих дужок справа і зліва від знаку операції.

{

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, <необхідний знак операції>);

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

Опрацювання вузла for\_node буде виглядати таким чином:

{

fprintf(outFile, "for(\n");

codegen(outFile, node->left);

fprintf(outFile, "\n) ");

codegen(outFile, node->right);

break;

}

Опрацювання вузла to\_node буде виглядати таким чином:

{

codegen(outFile, node->left);

codegen(outFile, node->left->left);

fprintf(outFile, " <= ");

codegen(outFile, node->right);

fprintf(outFile, ";\n++");

codegen(outFile, node->left->left);

break;

}

Опрацювання вузла downto\_node буде виглядати таким чином:

{

codegen(outFile, node->left);

codegen(outFile, node->left->left);

fprintf(outFile, " >= ");

codegen(outFile, node->right);

fprintf(outFile, ";\n--");

codegen(outFile, node->left->left);

break;

}

Опрацювання вузла while\_node буде виглядати таким чином:

{

fprintf(outFile, "while(");

codegen(outFile, node->left);

fprintf(outFile, ") {\n");

codegen(outFile, node->right);

fprintf(outFile, "}\n");

break;

}

Опрацювання вузла repeat\_node буде виглядати таким чином:

{

fprintf(outFile, "do {\n");

codegen(outFile, node->left);

fprintf(outFile, "} while(");

codegen(outFile, node->right);

fprintf(outFile, ");\n");

break;

}

Опрацювання вузла compound\_node буде виглядати таким чином:

{

fprintf(outFile, "{\n");

codegen(outFile, node->left);

codegen(outFile, node->right);

fprintf(outFile, "}\n");

break;

}

4. НАЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО

ТРАНСЛЯТОРА

Будь-яке програмне забезпечення необхідно протестувати і налагодити. Після опрацювання синтаксичних і семантичних помилок необхідно переконатися, що розроблене програмне забезпечення функціонує так, як очікувалось.

Для перевірки коректності роботи розробленого транслятора необхідно буде написати тестові задачі на вхідній мові програмування, отримати код на мові програмування С і переконатись, що він працює правильно.

* 1. Опис інтерфейсу та інструкції користувачу.

Розроблений транслятор має простий консольний інтерфейс.

При запуску програми необхідно ввести ім’я файлу з текстом програми на вхідній мові програмування:

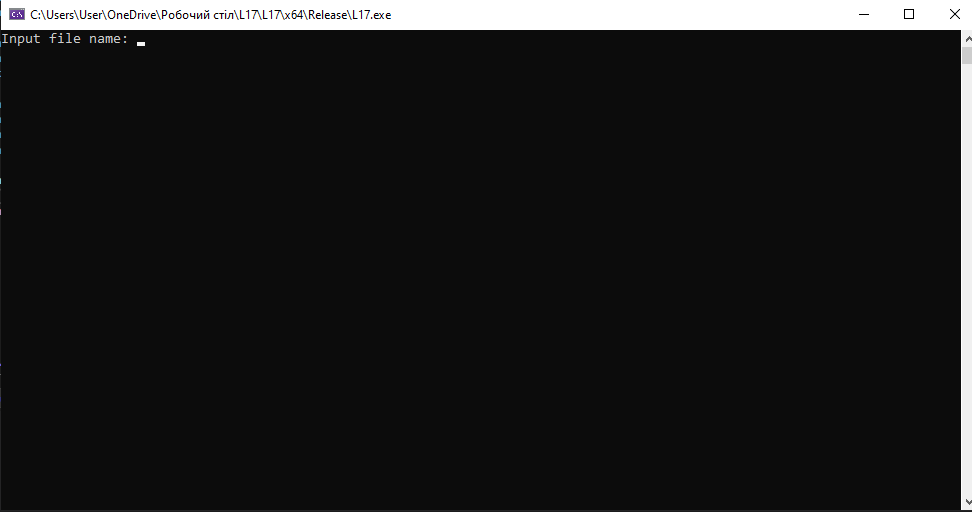


Рис. 4.1. Інтерфейс розробленого транслятора.

Після вводу імені програми отримуємо результати роботи транслятора:

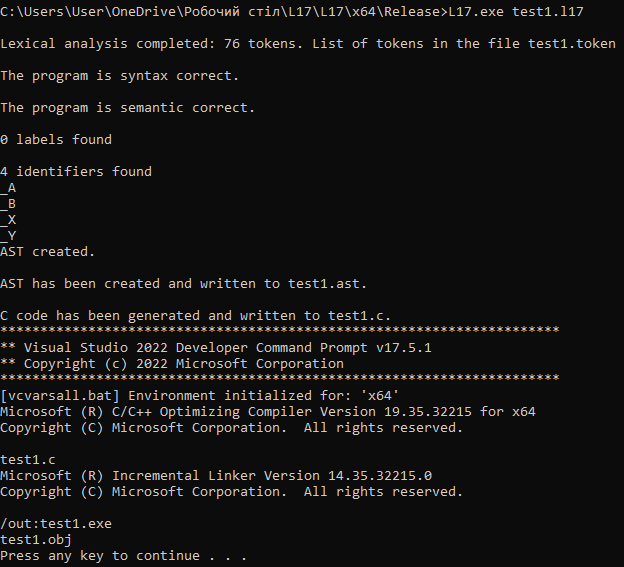


Рис. 4.2.Результати роботи розробленого транслятора.

* 1. Виявлення лексичних і синтаксичних помилок.

Помилки у вхідній програмі виявляються на етапі синтаксичного і семантичного аналізу.

Наприклад, у програмі зробимо синтаксичну помилку - у четвертому рядку введемо некоректну назву змінної:

{ Лінійний алгоритм }

STARTPROGRAM

STARTBLOK

VARIABLE INT16 a, \_B, \_X, \_Y;

{ 1. Ввести два числа А і В }

INPUT \_a

INPUT \_B

{ 2. Вивести на екран: ... }

OUTPUT \_a + \_B

OUTPUT \_a - \_B

OUTPUT \_a \* \_B

OUTPUT \_a DIV \_B

OUTPUT \_a MOD \_B

{ 3. Обрахувати значення виразів }

\_X <- (\_a - \_B) \* 10 + (\_a + \_B) DIV 10

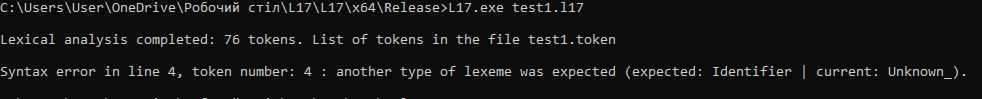
\_Y <- \_X + \_X MOD 10

{ 4. Вивести значення Х і У на екран. }

OUTPUT \_X

OUTPUT \_Y

ENDBLOK



*Рис. 4.3. Вивід інформації про синтаксичну помилку.*

Зробимо семантичну помилку - не оголосимо змінну “\_X”:

{ Лінійний алгоритм }

STARTPROGRAM

STARTBLOK

VARIABLE INT16 \_a, \_B, \_Y;

{ 1. Ввести два числа А і В }

INPUT \_a

INPUT \_B

{ 2. Вивести на екран: ... }

OUTPUT \_a + \_B

OUTPUT \_a - \_B

OUTPUT \_a \* \_B

OUTPUT \_a DIV \_B

OUTPUT \_a MOD \_B

{ 3. Обрахувати значення виразів }

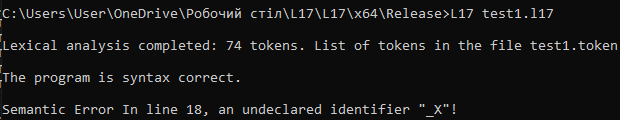
\_X <- (\_a - \_B) \* 10 + (\_a + \_B) DIV 10

\_Y <- \_X + \_X MOD 10

{ 4. Вивести значення Х і У на екран. }

OUTPUT \_X

OUTPUT \_Y

ENDBLOK

*Рис. 4.4. Вивід інформації про семантичну помилку.*

* 1. Перевірка роботи транслятора за допомогою тестових задач.

Тестова програма «*Лінійний алгоритм*»

1. Ввести два числа А і В (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання).

2. Вивести на екран:

А + В (результат операції додавання);

А - В (результат операції віднімання);

А \* В (результат операції множення);

А / В (результат операції ділення);

А % В (результат операції отримання залишку від ділення).

3. Обрахувати значення виразів

Х = (А - В) \* 10 + (А + В) / 10

Y = Х + Х % 10

1. Вивести значення Х і У на екран.

Напишемо програму на вхідній мові програмування:

{ Лінійний алгоритм }

STARTPROGRAM

STARTBLOK

VARIABLE INT16 \_A, \_B, \_X, \_Y;

{ 1. Ввести два числа А і В }

INPUT \_A

INPUT \_B

{ 2. Вивести на екран: ... }

OUTPUT \_A + \_B

OUTPUT \_A - \_B

OUTPUT \_A \* \_B

OUTPUT \_A DIV \_B

OUTPUT \_A MOD \_B

{ 3. Обрахувати значення виразів }

\_X <- (\_A - \_B) \* 10 + (\_A + \_B) DIV 10

\_Y <- \_X + \_X MOD 10

{ 4. Вивести значення Х і У на екран. }

OUTPUT \_X

OUTPUT \_Y

ENDBLOK

В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

int main() {

int \_Y;

int \_X;

int \_B;

int \_A;

printf("Enter \_A: ");

scanf("%d", &\_A);

printf("Enter \_B: ");

scanf("%d", &\_B);

printf("%d\n", (\_A + \_B));

printf("%d\n", (\_A - \_B));

printf("%d\n", (\_A \* \_B));

printf("%d\n", (\_A / \_B));

printf("%d\n", (\_A % \_B));

\_X = (((\_A - \_B) \* 10) + ((\_A + \_B) / 10));

\_Y = (\_X + (\_X % 10));

printf("%d\n", \_X);

printf("%d\n", \_Y);

return 0;

}

Також на виході маємо виконуваний файл test1.exe, запустивши який маємо:

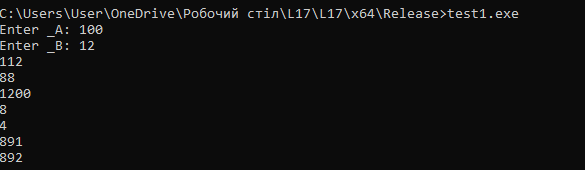


Рис. 4.5. Результати виконання тестової задачі 1.

**Тестова програма «*Алгоритм з розгалуженням*»**

1. Ввести три числа А, В, С (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання). Використання вкладеного умовного оператора:
2. Знайти найбільше з них і вивести його на екран. Використання простого умовного оператора:
3. Вивести на екран число 1, якщо усі числа однакові (логічний вираз в умовному операторі має виглядати так: «(А=В) і (А=С) і (В=С)»), інакше вивести 0.
4. Вивести на екран число -1, якщо хоча б одне з чисел від’ємне (логічний вираз в умовному операторі має виглядати так: «(А<0) або (В<0) або (С<0)»), інакше вивести 0.
5. Вивести на екран число 10, якщо число А більше за суму чисел В і С (логічний вираз в умовному операторі має виглядати так: «!(А<(В+С))»), інакше вивести 0

Напишемо програму на вхідній мові програмування:

{Алгоритм з розгалуженням}

STARTPROGRAM

STARTBLOK

VARIABLE INT16 \_A, \_B, \_C;

{ Ввести три числа А, В, С}

INPUT \_A

INPUT \_B

INPUT \_C

{ Використання вкладеного умовного оператора:

2. Знайти найбільше з них і вивести його на екран}

IF ( \_A >> \_B)

IF ( \_A >> \_C)

OUTPUT \_A

ELSE

OUTPUT \_C

ELSE

IF (\_B >> \_C)

OUTPUT \_B

ELSE

OUTPUT \_C

{ Використання простого умовного оператора:}

{ 3. Вивести на екран число 1, якщо усі числа однакові (логічний вираз в умовному операторі

має виглядати так: «(А=В) і (А=С) і (В=С)»), інакше вивести 0.}

IF ((\_A = \_B) && (\_A = \_C) && (\_B = \_C))

OUTPUT 1

ELSE

OUTPUT 0

{ 4. Вивести на екран число -1, якщо хоча б одне з чисел від’ємне (логічний вираз в умовному

операторі має виглядати так: «(А<<0) або (В<<0) або (С<<0)»), інакше вивести 0.}

IF ((\_A << 0) || (\_B << 0) || (\_C << 0))

OUTPUT -1

ELSE

OUTPUT 0

{ 5. Вивести на екран число 10, якщо число А більше за суму чисел В і С (логічний вираз в

умовному операторі має виглядати так: «!(А<<(В+С))»), інакше вивести 0}

IF (!!(\_A << (\_B + \_C)))

OUTPUT 10

ELSE

OUTPUT 0

ENDBLOK

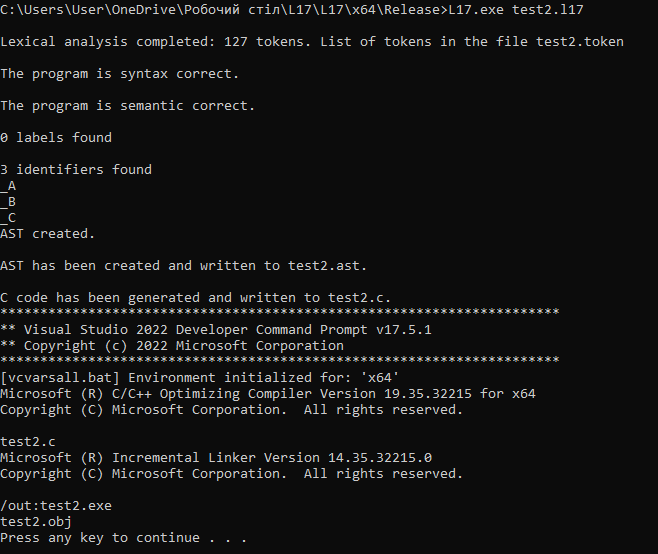


Рис. 4.6. Лексичний аналіз тестової задачі 2.

В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

int main() {

int \_C;

int \_B;

int \_A;

printf("Enter \_A: ");

scanf("%d", &\_A);

printf("Enter \_B: ");

scanf("%d", &\_B);

printf("Enter \_C: ");

scanf("%d", &\_C);

if ((\_A > \_B)) if ((\_A > \_C)) printf("%d\n", \_A);

else printf("%d\n", \_C);

else if ((\_B > \_C)) printf("%d\n", \_B);

else printf("%d\n", \_C);

if (((\_A == \_B) && ((\_A == \_C) && (\_B == \_C)))) printf("%d\n", 1);

else printf("%d\n", 0);

if (((\_A < 0) || ((\_B < 0) || (\_C < 0)))) printf("%d\n", -1);

else printf("%d\n", 0);

if (!((\_A < (\_B + \_C)))) printf("%d\n", 10);

else printf("%d\n", 0);

return 0;

}

Також на виході маємо виконуваний файл test2.exe, запустивши який маємо:

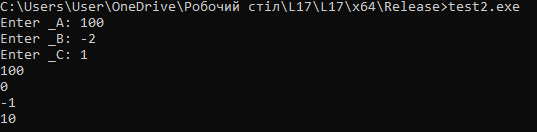


Рис. 4.7. Результати виконання тестової задачі 2.

**Тестова програма «*Циклічний алгоритм*»**

1. Ввести два числа А і В, причому А<В (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання).

Використання простого оператора циклу:

2. Вивести на екран квадрати чисел від А до В включно.

Використання вкладеного оператора циклу:

3. Обрахувати Х=А\*В за наступним алгоритмом:

Х = 0

Цикл від 1 до А з кроком 1

Цикл від 1 до В з кроком 1

Х = Х + 1

4. Вивести значення Х на екран.

Напишемо програму на вхідній мові програмування:

{ Тестова програма «Циклічний алгоритм» }

STARTPROGRAM

STARTBLOK

VARIABLE INT16 \_A, \_B, \_X, \_I, \_J;

{ 1. Ввести два числа А і В, причому А<В }

INPUT \_A

INPUT \_B

{ Використання простого оператора циклу: }

{ 2. Вивести на екран квадрати чисел від А до В включно. }

FOR \_X <- \_A TO \_B DO

OUTPUT \_X \* \_X

{ Використання вкладеного оператора циклу: }

{ 3. Обрахувати Х=А\*В за наступним алгоритмом:... }

\_X <- 0;

FOR \_I <- 1 TO \_A DO

FOR \_J <- 1 TO \_B DO

\_X <- \_X + 1

{ 4. Вивести значення Х на екран }

OUTPUT \_X

ENDBLOK

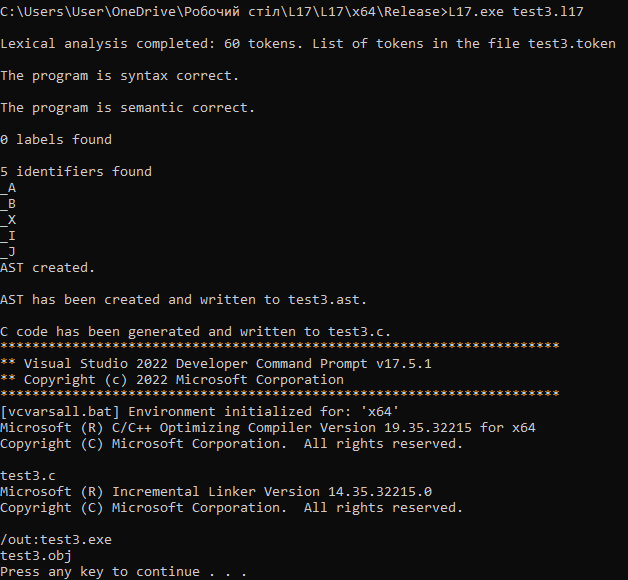


Рис. 4.8. Лексичний аналіз тестової задачі 3.

В результаті отримаємо файл з програмою на мові програмування С:

#include <stdio.h>

int main() {

int \_J;

int \_I;

int \_X;

int \_B;

int \_A;

printf("Enter \_A: ");

scanf("%d", &\_A);

printf("Enter \_B: ");

scanf("%d", &\_B);

for(

\_X = \_A;

\_X <= \_B;

++\_X

) printf("%d\n", (\_X \* \_X));

\_X = 0;

for(

\_I = 1;

\_I <= \_A;

++\_I

) for(

\_J = 1;

\_J <= \_B;

++\_J

) \_X = (\_X + 1);

printf("%d\n", \_X);

return 0;

}

Також на виході маємо виконуваний файл test3.exe, запустивши який маємо:

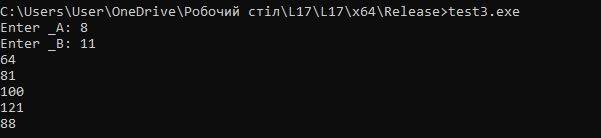


Рис. 4.9. Результати виконання тестової задачі 3.

**ВИСНОВКИ**

У межах курсового проєкту було розроблено транслятор для вхідної мови програмування, що реалізує такі основні функції:

1. **Лексичний аналіз**:
   * Алгоритм лексичного аналізу ділить текст програми на лексеми, формуючи таблицю з інформацією про їх тип, значення та номер рядка.
   * Лексичний аналізатор побудований на основі скінченного автомату, який розпізнає ключові слова, ідентифікатори, константи, оператори та роздільники.
2. **Синтаксичний і семантичний аналіз**:
   * Синтаксичний аналізатор перевіряє структуру програми відповідно до заданої граматики, будує дерево розбору, а також формує таблиці ідентифікаторів і типів.
   * Семантичний аналізатор забезпечує логічну коректність програми, перевіряючи відповідність типів даних, області видимості змінних та правильність викликів функцій.
3. **Генерація коду**:
   * Генератор коду перетворює абстрактне синтаксичне дерево у вихідний код мовою С, обробляючи дерево розбору й генеруючи код для кожного його елемента.
4. **Тестування**:
   * Транслятор був протестований на програмах різної структури (лінійних, з розгалуженнями, циклічних), що дозволило виявити й виправити лексичні, синтаксичні та семантичні помилки.
   * Результатом є коректний код на мові С, згенерований на основі вхідних програм.

**Переваги проєкту:**

* Повна реалізація всіх основних етапів трансляції.
* Модульна структура, що спрощує подальше розширення та модифікацію.
* Надійність роботи, підтверджена ретельним тестуванням.

**Недоліки проєкту:**

* Підтримка лише базових конструкцій вхідної мови.
* Відсутність графічного інтерфейсу, що ускладнює використання для непідготовлених користувачів.

Проєкт демонструє базову функціональність транслятора та слугує основою для подальшого вдосконалення й розширення.

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 - «Комп’ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. - Київ: КПІ ім. Ігоря Сікорського, 2021. - 108 с.
2. Формальні мови, граматики та автомати: Навчальний посібник / Гавриленко С.Ю. - Харків: НТУ «ХПІ», 2021. - 133 с.
3. Сопронюк Т.М. Системне програмування. Частина І. Елементи теорії формальних мов: Навчальний посібник у двох частинах. - Чернівці: ЧНУ, 2008. - 84 c.
4. Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. - Чернівці: ЧНУ, 2008. - 84 c.
5. Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. - 1038 c.
6. Системне програмування (курсовий проект) [Електронний ресурс] - Режим доступу до ресурсу: <https://vns.lpnu.ua/course/view.php?id=11685>.
7. MIT OpenCourseWare. Computer Language Engineering [Електронний ресурс] - Режим доступу до ресурсу: [https://ocw.mit.edu/courses/6-035-computer- language-engineering-spring-2010](https://ocw.mit.edu/courses/6-035-computer-language-engineering-spring-2010).

ДОДАТКИ

А. Таблиці лексем для тестових прикладів

Тестова програма «Лінійний алгоритм»

---------------------------------------------------------------------------

| TOKEN TABLE |

---------------------------------------------------------------------------

| line number | token | value | token code | type of token |

---------------------------------------------------------------------------

| 2 | STARTPROGRAM | 0 | 0 | StartProgram |

---------------------------------------------------------------------------

| 3 | STARTBLOK | 0 | 1 | StartBlock |

---------------------------------------------------------------------------

| 4 | VARIABLE | 0 | 2 | Variable |

---------------------------------------------------------------------------

| 4 | INT16 | 0 | 3 | Type |

---------------------------------------------------------------------------

| 4 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_Y | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 7 | INPUT | 0 | 5 | Input |

---------------------------------------------------------------------------

| 7 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 8 | INPUT | 0 | 5 | Input |

---------------------------------------------------------------------------

| 8 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 11 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 11 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 11 | + | 0 | 24 | Add |

---------------------------------------------------------------------------

| 11 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 12 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 12 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 12 | - | 0 | 25 | Sub |

---------------------------------------------------------------------------

| 12 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 13 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | \* | 0 | 26 | Mul |

---------------------------------------------------------------------------

| 13 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 14 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | DIV | 0 | 27 | Div |

---------------------------------------------------------------------------

| 14 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 15 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 15 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 15 | MOD | 0 | 28 | Mod |

---------------------------------------------------------------------------

| 15 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 18 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 18 | <- | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 18 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 18 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 18 | - | 0 | 25 | Sub |

---------------------------------------------------------------------------

| 18 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 18 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 18 | \* | 0 | 26 | Mul |

---------------------------------------------------------------------------

| 18 | 10 | 10 | 21 | Number |

---------------------------------------------------------------------------

| 18 | + | 0 | 24 | Add |

---------------------------------------------------------------------------

| 18 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 18 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 18 | + | 0 | 24 | Add |

---------------------------------------------------------------------------

| 18 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 18 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 18 | DIV | 0 | 27 | Div |

---------------------------------------------------------------------------

| 18 | 10 | 10 | 21 | Number |

---------------------------------------------------------------------------

| 19 | \_Y | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | <- | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 19 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | + | 0 | 24 | Add |

---------------------------------------------------------------------------

| 19 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | MOD | 0 | 28 | Mod |

---------------------------------------------------------------------------

| 19 | 10 | 10 | 21 | Number |

---------------------------------------------------------------------------

| 22 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 22 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 23 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 23 | \_Y | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 25 | ENDBLOK | 0 | 4 | EndBlock |

---------------------------------------------------------------------------

Тестова програма «Алгоритм з розгалуженням»

---------------------------------------------------------------------------

| TOKEN TABLE |

---------------------------------------------------------------------------

| line number | token | value | token code | type of token |

---------------------------------------------------------------------------

| 2 | STARTPROGRAM | 0 | 0 | StartProgram |

---------------------------------------------------------------------------

| 3 | STARTBLOK | 0 | 1 | StartBlock |

---------------------------------------------------------------------------

| 4 | VARIABLE | 0 | 2 | Variable |

---------------------------------------------------------------------------

| 4 | INT16 | 0 | 3 | Type |

---------------------------------------------------------------------------

| 4 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 7 | INPUT | 0 | 5 | Input |

---------------------------------------------------------------------------

| 7 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 8 | INPUT | 0 | 5 | Input |

---------------------------------------------------------------------------

| 8 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 9 | INPUT | 0 | 5 | Input |

---------------------------------------------------------------------------

| 9 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 13 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 13 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | >> | 0 | 31 | Greate |

---------------------------------------------------------------------------

| 13 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 14 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 14 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 14 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | >> | 0 | 31 | Greate |

---------------------------------------------------------------------------

| 14 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 15 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 15 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 16 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 17 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 17 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 18 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 19 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 19 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 19 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | >> | 0 | 31 | Greate |

---------------------------------------------------------------------------

| 19 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 20 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 20 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 21 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 22 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 22 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 28 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 28 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 28 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 28 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 28 | = | 0 | 29 | Equality |

---------------------------------------------------------------------------

| 28 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 28 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 28 | && | 0 | 34 | And |

---------------------------------------------------------------------------

| 28 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 28 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 28 | = | 0 | 29 | Equality |

---------------------------------------------------------------------------

| 28 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 28 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 28 | && | 0 | 34 | And |

---------------------------------------------------------------------------

| 28 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 28 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 28 | = | 0 | 29 | Equality |

---------------------------------------------------------------------------

| 28 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 28 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 28 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 29 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 29 | 1 | 1 | 21 | Number |

---------------------------------------------------------------------------

| 30 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 31 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 31 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 35 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 35 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 35 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 35 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 35 | << | 0 | 32 | Less |

---------------------------------------------------------------------------

| 35 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 35 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 35 | || | 0 | 35 | Or |

---------------------------------------------------------------------------

| 35 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 35 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 35 | << | 0 | 32 | Less |

---------------------------------------------------------------------------

| 35 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 35 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 35 | || | 0 | 35 | Or |

---------------------------------------------------------------------------

| 35 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 35 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 35 | << | 0 | 32 | Less |

---------------------------------------------------------------------------

| 35 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 35 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 35 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 36 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 36 | -1 | -1 | 21 | Number |

---------------------------------------------------------------------------

| 37 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 38 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 38 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 42 | IF | 0 | 7 | If |

---------------------------------------------------------------------------

| 42 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 42 | !! | 0 | 33 | Not |

---------------------------------------------------------------------------

| 42 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 42 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 42 | << | 0 | 32 | Less |

---------------------------------------------------------------------------

| 42 | ( | 0 | 36 | LBracket |

---------------------------------------------------------------------------

| 42 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 42 | + | 0 | 24 | Add |

---------------------------------------------------------------------------

| 42 | \_C | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 42 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 42 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 42 | ) | 0 | 37 | RBracket |

---------------------------------------------------------------------------

| 43 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 43 | 10 | 10 | 21 | Number |

---------------------------------------------------------------------------

| 44 | ELSE | 0 | 8 | Else |

---------------------------------------------------------------------------

| 45 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 45 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 48 | ENDBLOK | 0 | 4 | EndBlock |

---------------------------------------------------------------------------

Тестова програма «*Циклічний алгоритм*»

---------------------------------------------------------------------------

| TOKEN TABLE |

---------------------------------------------------------------------------

| line number | token | value | token code | type of token |

---------------------------------------------------------------------------

| 2 | STARTPROGRAM | 0 | 0 | StartProgram |

---------------------------------------------------------------------------

| 3 | STARTBLOK | 0 | 1 | StartBlock |

---------------------------------------------------------------------------

| 4 | VARIABLE | 0 | 2 | Variable |

---------------------------------------------------------------------------

| 4 | INT16 | 0 | 3 | Type |

---------------------------------------------------------------------------

| 4 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_I | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | , | 0 | 40 | Comma |

---------------------------------------------------------------------------

| 4 | \_J | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 4 | ; | 0 | 38 | Semicolon |

---------------------------------------------------------------------------

| 7 | INPUT | 0 | 5 | Input |

---------------------------------------------------------------------------

| 7 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 8 | INPUT | 0 | 5 | Input |

---------------------------------------------------------------------------

| 8 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | FOR | 0 | 10 | For |

---------------------------------------------------------------------------

| 13 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | <- | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 13 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | TO | 0 | 11 | To |

---------------------------------------------------------------------------

| 13 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 13 | DO | 0 | 13 | Do |

---------------------------------------------------------------------------

| 14 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 14 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 14 | \* | 0 | 26 | Mul |

---------------------------------------------------------------------------

| 14 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 19 | <- | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 19 | 0 | 0 | 21 | Number |

---------------------------------------------------------------------------

| 20 | FOR | 0 | 10 | For |

---------------------------------------------------------------------------

| 20 | \_I | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 20 | <- | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 20 | 1 | 1 | 21 | Number |

---------------------------------------------------------------------------

| 20 | TO | 0 | 11 | To |

---------------------------------------------------------------------------

| 20 | \_A | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 20 | DO | 0 | 13 | Do |

---------------------------------------------------------------------------

| 21 | FOR | 0 | 10 | For |

---------------------------------------------------------------------------

| 21 | \_J | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 21 | <- | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 21 | 1 | 1 | 21 | Number |

---------------------------------------------------------------------------

| 21 | TO | 0 | 11 | To |

---------------------------------------------------------------------------

| 21 | \_B | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 21 | DO | 0 | 13 | Do |

---------------------------------------------------------------------------

| 22 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 22 | <- | 0 | 23 | Assign |

---------------------------------------------------------------------------

| 22 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 22 | + | 0 | 24 | Add |

---------------------------------------------------------------------------

| 22 | 1 | 1 | 21 | Number |

---------------------------------------------------------------------------

| 25 | OUTPUT | 0 | 6 | Output |

---------------------------------------------------------------------------

| 25 | \_X | 0 | 20 | Identifier |

---------------------------------------------------------------------------

| 27 | ENDBLOK | 0 | 4 | EndBlock |

---------------------------------------------------------------------------

Б. Абстрактне синтаксичне дерево для тестових прикладів

Тестова програма «Лінійний алгоритм»

|-- Program(0)

| |-- var(2)

| | |-- \_Y(1)

| | |-- var(2)

| | | |-- \_X(1)

| | | |-- var(2)

| | | | |-- \_B(1)

| | | | |-- var(2)

| | | | | |-- \_A(1)

| |-- statement(3)

| | |-- statement(3)

| | | |-- statement(3)

| | | | |-- statement(3)

| | | | | |-- statement(3)

| | | | | | |-- statement(3)

| | | | | | | |-- statement(3)

| | | | | | | | |-- statement(3)

| | | | | | | | | |-- statement(3)

| | | | | | | | | | |-- statement(3)

| | | | | | | | | | | |-- input(4)

| | | | | | | | | | | | |-- \_A(1)

| | | | | | | | | | | |-- input(4)

| | | | | | | | | | | | |-- \_B(1)

| | | | | | | | | | |-- output(5)

| | | | | | | | | | | |-- Add(6)

| | | | | | | | | | | | |-- \_A(1)

| | | | | | | | | | | | |-- \_B(1)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- Sub(7)

| | | | | | | | | | | |-- \_A(1)

| | | | | | | | | | | |-- \_B(1)

| | | | | | | | |-- output(5)

| | | | | | | | | |-- Mul(8)

| | | | | | | | | | |-- \_A(1)

| | | | | | | | | | |-- \_B(1)

| | | | | | | |-- output(5)

| | | | | | | | |-- Div(9)

| | | | | | | | | |-- \_A(1)

| | | | | | | | | |-- \_B(1)

| | | | | | |-- output(5)

| | | | | | | |-- Mod(10)

| | | | | | | | |-- \_A(1)

| | | | | | | | |-- \_B(1)

| | | | | |-- assign(12)

| | | | | | |-- \_X(1)

| | | | | | |-- Add(6)

| | | | | | | |-- Mul(8)

| | | | | | | | |-- Sub(7)

| | | | | | | | | |-- \_A(1)

| | | | | | | | | |-- \_B(1)

| | | | | | | | |-- 10(11)

| | | | | | | |-- Div(9)

| | | | | | | | |-- Add(6)

| | | | | | | | | |-- \_A(1)

| | | | | | | | | |-- \_B(1)

| | | | | | | | |-- 10(11)

| | | | |-- assign(12)

| | | | | |-- \_Y(1)

| | | | | |-- Add(6)

| | | | | | |-- \_X(1)

| | | | | | |-- Mod(10)

| | | | | | | |-- \_X(1)

| | | | | | | |-- 10(11)

| | | |-- output(5)

| | | | |-- \_X(1)

| | |-- output(5)

| | | |-- \_Y(1)

Тестова програма «Алгоритм з розгалуженням»

|-- Program(0)

| |-- var(2)

| | |-- \_C(1)

| | |-- var(2)

| | | |-- \_B(1)

| | | |-- var(2)

| | | | |-- \_A(1)

| |-- statement(3)

| | |-- statement(3)

| | | |-- statement(3)

| | | | |-- statement(3)

| | | | | |-- statement(3)

| | | | | | |-- statement(3)

| | | | | | | |-- input(4)

| | | | | | | | |-- \_A(1)

| | | | | | | |-- input(4)

| | | | | | | | |-- \_B(1)

| | | | | | |-- input(4)

| | | | | | | |-- \_C(1)

| | | | | |-- if(13)

| | | | | | |-- Greate(20)

| | | | | | | |-- \_A(1)

| | | | | | | |-- \_B(1)

| | | | | | |-- else(14)

| | | | | | | |-- if(13)

| | | | | | | | |-- Greate(20)

| | | | | | | | | |-- \_A(1)

| | | | | | | | | |-- \_C(1)

| | | | | | | | |-- else(14)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- \_A(1)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- \_C(1)

| | | | | | | |-- if(13)

| | | | | | | | |-- Greate(20)

| | | | | | | | | |-- \_B(1)

| | | | | | | | | |-- \_C(1)

| | | | | | | | |-- else(14)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- \_B(1)

| | | | | | | | | |-- output(5)

| | | | | | | | | | |-- \_C(1)

| | | | |-- if(13)

| | | | | |-- and(16)

| | | | | | |-- Equality(18)

| | | | | | | |-- \_A(1)

| | | | | | | |-- \_B(1)

| | | | | | |-- and(16)

| | | | | | | |-- Equality(18)

| | | | | | | | |-- \_A(1)

| | | | | | | | |-- \_C(1)

| | | | | | | |-- Equality(18)

| | | | | | | | |-- \_B(1)

| | | | | | | | |-- \_C(1)

| | | | | |-- else(14)

| | | | | | |-- output(5)

| | | | | | | |-- 1(11)

| | | | | | |-- output(5)

| | | | | | | |-- 0(11)

| | | |-- if(13)

| | | | |-- or(15)

| | | | | |-- Less(21)

| | | | | | |-- \_A(1)

| | | | | | |-- 0(11)

| | | | | |-- or(15)

| | | | | | |-- Less(21)

| | | | | | | |-- \_B(1)

| | | | | | | |-- 0(11)

| | | | | | |-- Less(21)

| | | | | | | |-- \_C(1)

| | | | | | | |-- 0(11)

| | | | |-- else(14)

| | | | | |-- output(5)

| | | | | | |-- -1(11)

| | | | | |-- output(5)

| | | | | | |-- 0(11)

| | |-- if(13)

| | | |-- not(17)

| | | | |-- Less(21)

| | | | | |-- \_A(1)

| | | | | |-- Add(6)

| | | | | | |-- \_B(1)

| | | | | | |-- \_C(1)

| | | |-- else(14)

| | | | |-- output(5)

| | | | | |-- 10(11)

| | | | |-- output(5)

| | | | | |-- 0(11)

Тестова програма «*Циклічний алгоритм*»

|-- Program(0)

| |-- var(2)

| | |-- \_J(1)

| | |-- var(2)

| | | |-- \_I(1)

| | | |-- var(2)

| | | | |-- \_X(1)

| | | | |-- var(2)

| | | | | |-- \_B(1)

| | | | | |-- var(2)

| | | | | | |-- \_A(1)

| |-- statement(3)

| | |-- statement(3)

| | | |-- statement(3)

| | | | |-- statement(3)

| | | | | |-- statement(3)

| | | | | | |-- input(4)

| | | | | | | |-- \_A(1)

| | | | | | |-- input(4)

| | | | | | | |-- \_B(1)

| | | | | |-- for(24)

| | | | | | |-- to(25)

| | | | | | | |-- assign(12)

| | | | | | | | |-- \_X(1)

| | | | | | | | |-- \_A(1)

| | | | | | | |-- \_B(1)

| | | | | | |-- output(5)

| | | | | | | |-- Mul(8)

| | | | | | | | |-- \_X(1)

| | | | | | | | |-- \_X(1)

| | | | |-- assign(12)

| | | | | |-- \_X(1)

| | | | | |-- 0(11)

| | | |-- for(24)

| | | | |-- to(25)

| | | | | |-- assign(12)

| | | | | | |-- \_I(1)

| | | | | | |-- 1(11)

| | | | | |-- \_A(1)

| | | | |-- for(24)

| | | | | |-- to(25)

| | | | | | |-- assign(12)

| | | | | | | |-- \_J(1)

| | | | | | | |-- 1(11)

| | | | | | |-- \_B(1)

| | | | | |-- assign(12)

| | | | | | |-- \_X(1)

| | | | | | |-- Add(6)

| | | | | | | |-- \_X(1)

| | | | | | | |-- 1(11)

| | |-- output(5)

| | | |-- \_X(1)

В. С код, отриманий на виході транслятора для тестових прикладів

Тестова програма «Лінійний алгоритм»

#include <stdio.h>

int main() {

int \_Y;

int \_X;

int \_B;

int \_A;

printf("Enter \_A: ");

scanf("%d", &\_A);

printf("Enter \_B: ");

scanf("%d", &\_B);

printf("%d\n", (\_A + \_B));

printf("%d\n", (\_A - \_B));

printf("%d\n", (\_A \* \_B));

printf("%d\n", (\_A / \_B));

printf("%d\n", (\_A % \_B));

\_X = (((\_A - \_B) \* 10) + ((\_A + \_B) / 10));

\_Y = (\_X + (\_X % 10));

printf("%d\n", \_X);

printf("%d\n", \_Y);

return 0;

}

Тестова програма «Алгоритм з розгалуженням»

#include <stdio.h>

int main() {

int \_C;

int \_B;

int \_A;

printf("Enter \_A: ");

scanf("%d", &\_A);

printf("Enter \_B: ");

scanf("%d", &\_B);

printf("Enter \_C: ");

scanf("%d", &\_C);

if ((\_A > \_B)) if ((\_A > \_C)) printf("%d\n", \_A);

else printf("%d\n", \_C);

else if ((\_B > \_C)) printf("%d\n", \_B);

else printf("%d\n", \_C);

if (((\_A == \_B) && ((\_A == \_C) && (\_B == \_C)))) printf("%d\n", 1);

else printf("%d\n", 0);

if (((\_A < 0) || ((\_B < 0) || (\_C < 0)))) printf("%d\n", -1);

else printf("%d\n", 0);

if (!((\_A < (\_B + \_C)))) printf("%d\n", 10);

else printf("%d\n", 0);

return 0;

}

Тестова програма «*Циклічний алгоритм*»

#include <stdio.h>

int main() {

int \_J;

int \_I;

int \_X;

int \_B;

int \_A;

printf("Enter \_A: ");

scanf("%d", &\_A);

printf("Enter \_B: ");

scanf("%d", &\_B);

for(

\_X = \_A;

\_X <= \_B;

++\_X

) printf("%d\n", (\_X \* \_X));

\_X = 0;

for(

\_I = 1;

\_I <= \_A;

++\_I

) for(

\_J = 1;

\_J <= \_B;

++\_J

) \_X = (\_X + 1);

printf("%d\n", \_X);

return 0;

}

Г. Креслення алгоритму транслятора



Д. Лістинги коду

Main.cpp

#include <iostream>

#include <windows.h>

#include "LexicAnalyzer.hpp"

#include "Parser.hpp"

#include "Ast.hpp"

#include "Codegen.hpp"

#define LANGUAGE ".l17"

struct Token\* TokenTable; // Таблиця лексем

unsigned int TokensNum; // Кількість лексем

struct id\* idTable; // Таблиця ідентифікаторів

unsigned int idNum; // кількість ідентифікаторів

struct id\* labelTable; // Таблиця міток

unsigned int labelNum; // кількість міток

FILE\* errorFile;

int main(int argc, char\* argv[])

{

// виділення пам'яті під таблицю лесем

TokenTable = (struct Token\*)malloc(MAX\_TOKENS \* sizeof(struct Token));

// виділення пам'яті під таблицю ідентифікаторів

idTable = (struct id\*)malloc(MAX\_IDENTIFIER \* sizeof(struct id));

labelTable = (struct id\*)malloc(MAX\_IDENTIFIER \* sizeof(struct id));

char InputFile[32] = "";

FILE\* InFile, \*TokenFileP;

if (argc != 2)

{

printf("Input file name: ");

gets\_s(InputFile);

}

else

{

strcpy\_s(InputFile, argv[1]);

}

if (!strstr(InputFile, LANGUAGE)) {

free(TokenTable);

free(idTable);

free(labelTable);

\_fcloseall();

printf("Program file name should contain \"%s\"", LANGUAGE);

exit(1);

}

if ((fopen\_s(&InFile, InputFile, "rt")) != 0)

{

printf("Error: Can not open file: %s\n", InputFile);

return 1;

}

char NameFile[32] = "";

int i = 0;

while (InputFile[i] != '.' && InputFile[i] != '\0')

{

NameFile[i] = InputFile[i];

i++;

}

NameFile[i] = '\0';

char TokenFile[32], errorFileName[32];

strcpy\_s(TokenFile, NameFile);

strcat\_s(TokenFile, ".token");

strcpy\_s(errorFileName, NameFile);

strcat\_s(errorFileName, ".errorlist");

// лексичний аналіз

if ((fopen\_s(&TokenFileP, TokenFile, "wt")) != 0)

{

printf("Error: Can not create file: %s\n", TokenFile);

free(TokenTable);

free(idTable);

free(labelTable);

\_fcloseall();

return 1;

}

if ((fopen\_s(&errorFile, errorFileName, "wt")) != 0)

{

printf("Error: Can not create file: %s\n", errorFileName);

free(TokenTable);

free(idTable);

free(labelTable);

\_fcloseall();

return 1;

}

TokensNum = LexicAnalyzer::getTokens(InFile);

LexicAnalyzer::fprintTokens(TokenFileP);

fclose(InFile);

fclose(TokenFileP);

printf("\nLexical analysis completed: %d tokens. List of tokens in the file %s\n", TokensNum, TokenFile);

//PrintTokens(TokenTable, TokensNum);

// синтаксичний аналіз

Parser::Parser();

Parser::Semantic();

// створення абстрактного синтаксичного дерева

struct astNode\* ast = AST::astParser();

//printf("\nAbstract Syntax Tree:\n");

//PrintAST(ASTree, 0);

char AST[32];

strcpy\_s(AST, NameFile);

strcat\_s(AST, ".ast");

// Open output file

FILE\* ASTFile;

fopen\_s(&ASTFile, AST, "w");

if (!ASTFile)

{

printf("Failed to open output file.\n");

free(TokenTable);

free(idTable);

free(labelTable);

AST::deleteNode(ast);

exit(1);

}

AST::fPrintAST(ASTFile, ast, 0);

printf("\nAST has been created and written to %s.\n", AST);

char OutputFile[32];

strcpy\_s(OutputFile, NameFile);

strcat\_s(OutputFile, ".c");

// Open output file

FILE\* outFile;

fopen\_s(&outFile, OutputFile, "w");

if (!outFile)

{

printf("Failed to open output file.\n");

free(TokenTable);

free(idTable);

free(labelTable);

AST::deleteNode(ast);

exit(1);

}

// Generate C code from AST

Codegen::codegen(outFile, ast);

// генерація вихідного С коду

printf("\nC code has been generated and written to %s.\n", OutputFile);

// Close the file

\_fcloseall();

free(TokenTable);

free(idTable);

free(labelTable);

char setVar[256] = "\"C:\\Program Files\\Microsoft Visual Studio\\2022\\Community\\VC\\Auxiliary\\Build\\vcvars64.bat\"";

char createExe[128];

sprintf\_s(createExe, "cl %s", OutputFile);

strcat\_s(setVar, " && ");

strcat\_s(setVar, createExe);

system(setVar);

//\*/

system("pause");

return 0;

}

LexicAnalyzer.hpp

#pragma once

#include "header.hpp"

#include <stdio.h>

namespace LexicAnalyzer {

unsigned int getTokens(FILE\* F);

void printTokens(void);

void fprintTokens(FILE\* F);

//const char\* lexemeTypeName(enum TypeOfToken type);

}

LexicAnalyzer.cpp

#include "LexicAnalyzer.hpp"

#include <stdbool.h>

#include <string.h>

#include <stdlib.h>

extern struct Token\* TokenTable; // Таблиця лексем

extern unsigned int TokensNum; // Кількість лексем

unsigned int LexicAnalyzer::getTokens(FILE\* F) {

enum States state = Start;

struct Token tempToken;

char ch, buf[16];

unsigned int tokenCount = 0;

int line = 1;

int tokenLength = 0;

ch = getc(F);

while (true) {

switch (state) {

case Start: {

if (ch == EOF) {

state = EndOFile;

}

else if (('0' <= ch && ch <= '9') || ch == '-' || ch == '.') {

state = Digit;

}

else if (ch == '{') {

state = Comment;

}

else if (('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z') || ch == '\_') {

state = Letter;

}

else if (ch == ' ' || ch == '\t' || ch == '\n') {

state = Separator;

}

else {

state = Another;

}

break;

}

case Digit: {

buf[0] = ch;

int j = 1;

ch = getc(F);

while (((ch <= '9' && ch >= '0') || ch == '.') && j < 10) {

buf[j++] = ch;

ch = getc(F);

}

buf[j] = '\0';

if (!strcmp(buf, "-")) {

strcpy\_s(tempToken.name, "-");

tempToken.type = Sub;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

if (!strcmp(buf, ".")) {

strcpy\_s(tempToken.name, ".");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

short dotCounter = 0, currCharIndex = 0;

while (buf[currCharIndex] != '\0') {

if (buf[currCharIndex] == '.') {

++dotCounter;

}

++currCharIndex;

}

if (dotCounter > 1) {

strcpy\_s(tempToken.name, buf);

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

if (dotCounter == 1) {

strcpy\_s(tempToken.name, buf);

tempToken.type = Float;

tempToken.value = atof(buf);

tempToken.line = line;

state = Finish;

break;

}

if ((buf[0] == '0' && buf[1] != '\0') || (buf[0] == '-' && buf[1] == '0')) {

strcpy\_s(tempToken.name, buf);

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

strcpy\_s(tempToken.name, buf);

tempToken.type = Number;

tempToken.value = atoi(buf);

tempToken.line = line;

state = Finish;

break;

}

case Separator: {

if (ch == '\n') {

line++;

}

ch = getc(F);

state = Start;

break;

}

case Comment: {

while (ch != '}' && ch != EOF) {

if (ch == '\n')

++line;

ch = getc(F);

}

if (ch == EOF)

state = EndOFile;

else {

state = Start;

ch = getc(F);

}

break;

}

case Finish: {

if (tokenCount < MAX\_TOKENS) {

TokenTable[tokenCount++] = tempToken;

if (ch != EOF) {

state = Start;

}

else {

state = EndOFile;

}

}

else {

printf("\n\t\t\ttoo many tokens !!!\n");

return TokensNum = tokenCount - 1;

}

break;

}

case EndOFile: {

return TokensNum = tokenCount;

}

case Letter: {

buf[0] = ch;

int j = 1;

ch = getc(F);

while ((('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z') || ('0' <= ch && ch <= '9')) && j < 15) {

buf[j++] = ch;

ch = getc(F);

}

buf[j] = '\0';

enum TypeOfToken tempType = Unknown\_;

if (!strcmp(buf, "STARTPROGRAM")) {

tempType = StartProgram;

}

else if (!strcmp(buf, "STARTBLOK")) {

tempType = StartBlock;

}

else if (!strcmp(buf, "VARIABLE")) {

tempType = Variable;

}

else if (!strcmp(buf, "INT16")) {

tempType = Type;

}

else if (!strcmp(buf, "ENDBLOK")) {

tempType = EndBlock;

}

else if (!strcmp(buf, "INPUT")) {

tempType = Input;

}

else if (!strcmp(buf, "OUTPUT")) {

tempType = Output;

}

else if (!strcmp(buf, "IF")) {

tempType = If;

}

else if (!strcmp(buf, "ELSE")) {

tempType = Else;

}

else if (!strcmp(buf, "GOTO")) {

tempType = Goto;

}

else if (!strcmp(buf, "FOR")) {

tempType = For;

}

else if (!strcmp(buf, "TO")) {

tempType = To;

}

else if (!strcmp(buf, "DOWNTO")) {

tempType = Downto;

}

else if (!strcmp(buf, "DO")) {

tempType = Do;

}

else if (!strcmp(buf, "WHILE")) {

tempType = While;

}

else if (!strcmp(buf, "END")) {

tempType = End;

}

else if (!strcmp(buf, "EXIT")) {

tempType = Exit;

}

else if (!strcmp(buf, "CONTINUE")) {

tempType = Continue;

}

else if (!strcmp(buf, "REPEAT")) {

tempType = Repeat;

}

else if (!strcmp(buf, "UNTIL")) {

tempType = Until;

}

else if (!strcmp(buf, "DIV")) {

tempType = Div;

}

else if (!strcmp(buf, "MOD")) {

tempType = Mod;

}

else if (strlen(buf) <= 2 && buf[0] == '\_' && (('A' <= buf[1] && buf[1] <= 'Z') || ('0' <= buf[1] && buf[1] <= '9'))) {

tempType = Identifier;

}

strcpy\_s(tempToken.name, buf);

tempToken.type = tempType;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

break;

}

case Another: {

switch (ch) {

case '(': {

strcpy\_s(tempToken.name, "(");

tempToken.type = LBracket;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case ')': {

strcpy\_s(tempToken.name, ")");

tempToken.type = RBracket;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case ',': {

strcpy\_s(tempToken.name, ",");

tempToken.type = Comma;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case ';': {

strcpy\_s(tempToken.name, ";");

tempToken.type = Semicolon;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case ':': {

strcpy\_s(tempToken.name, ":");

tempToken.type = Colon;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case '|': {

ch = getc(F);

if (ch == '|') {

strcpy\_s(tempToken.name, "||");

tempToken.type = Or;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

}

else {

strcpy\_s(tempToken.name, "|");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

}

break;

}

case '&': {

ch = getc(F);

if (ch == '&') {

strcpy\_s(tempToken.name, "&&");

tempToken.type = And;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

}

else {

strcpy\_s(tempToken.name, "&");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

}

break;

}

case '!': {

ch = getc(F);

if (ch == '!') {

strcpy\_s(tempToken.name, "!!");

tempToken.type = Not;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

}

else if (ch == '=') {

strcpy\_s(tempToken.name, "!=");

tempToken.type = NotEquality;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

}

else {

strcpy\_s(tempToken.name, "!");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

}

break;

}

case '+': {

strcpy\_s(tempToken.name, "+");

tempToken.type = Add;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case '\*': {

strcpy\_s(tempToken.name, "\*");

tempToken.type = Mul;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case '/': {

strcpy\_s(tempToken.name, "/");

tempToken.type = Mul;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case '%': {

strcpy\_s(tempToken.name, "%");

tempToken.type = Mod;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

case '<': {

ch = getc(F);

if (ch == '<') {

strcpy\_s(tempToken.name, "<<");

tempToken.type = Less;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

}

else if (ch == '-') {

strcpy\_s(tempToken.name, "<-");

tempToken.type = Assign;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

}

else {

strcpy\_s(tempToken.name, "<");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

}

break;

}

case '>': {

ch = getc(F);

if (ch == '>') {

strcpy\_s(tempToken.name, ">>");

tempToken.type = Greate;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

}

else {

strcpy\_s(tempToken.name, ">");

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

}

break;

}

case '=': {

strcpy\_s(tempToken.name, "=");

tempToken.type = Equality;

tempToken.value = 0;

tempToken.line = line;

state = Finish;

ch = getc(F);

break;

}

default: {

tempToken.name[0] = ch;

tempToken.name[1] = '\0';

tempToken.type = Unknown\_;

tempToken.value = 0;

tempToken.line = line;

ch = getc(F);

state = Finish;

break;

}

}

}

}

}

return TokensNum = tokenCount;

}

// Функція друкує таблицю лексем на екран

void LexicAnalyzer::printTokens(void) {

char type\_tokens[16];

printf("---------------------------------------------------------------------------\n");

printf("| TOKEN TABLE |\n");

printf("---------------------------------------------------------------------------\n");

printf("| line number | token | value | token code | type of token |\n");

printf("---------------------------------------------------------------------------\n");

for (unsigned int i = 0; i < TokensNum; i++) {

strcpy\_s(type\_tokens, lexemeTypeName(TokenTable[i].type));

printf("|%12d |%16s |%11d |%11d | %-13s |\n",

TokenTable[i].line,

TokenTable[i].name,

TokenTable[i].value,

TokenTable[i].type,

type\_tokens);

printf("---------------------------------------------------------------------------\n");

}

}

void LexicAnalyzer::fprintTokens(FILE\* F) {

char type\_tokens[16];

fprintf(F, "\n\n---------------------------------------------------------------------------\n");

fprintf(F, "| TOKEN TABLE |\n");

fprintf(F, "---------------------------------------------------------------------------\n");

fprintf(F, "| line number | token | value | token code | type of token |\n");

fprintf(F, "---------------------------------------------------------------------------\n");

for (unsigned int i = 0; i < TokensNum; i++) {

strcpy\_s(type\_tokens, lexemeTypeName(TokenTable[i].type));

fprintf(F, "|%12d |%16s |%11d |%11d | %-13s |\n",

TokenTable[i].line,

TokenTable[i].name,

TokenTable[i].value,

TokenTable[i].type,

type\_tokens);

fprintf(F, "---------------------------------------------------------------------------\n");

}

}

Codegen.hpp

#pragma once

#include "header.hpp"

#include <stdio.h>

#include <stdlib.h>

namespace Codegen {

void codegen(FILE\* outFile, struct astNode\* node);

}

Parser.cpp

#include "Parser.hpp"

#include <stdbool.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

extern struct Token\* TokenTable;

extern unsigned int TokensNum;

extern struct id\* idTable;

extern unsigned int idNum;

extern struct id\* labelTable;

extern unsigned int labelNum;

extern FILE\* errorFile;

int pos = 0;

namespace Parser {

void Parser() {

program();

printf("\nThe program is syntax correct.\n");

fprintf(errorFile, "\nThe program is syntax correct.\n");

}

void Semantic() {

idNum = IdIdentification(idTable, TokenTable, TokensNum);

labelNum = LabelIdentification(labelTable, TokenTable, TokensNum);

printf("\nThe program is semantic correct.\n");

printf("\n%d labels found\n", labelNum);

fprintf(errorFile, "\nThe program is semantic correct.\n");

fprintf(errorFile, "\n%d labels found\n", labelNum);

printIdentifiers(labelNum, labelTable);

fprintIdentifiers(errorFile, labelNum, labelTable);

printf("\n%d identifiers found\n", idNum);

fprintf(errorFile, "\n%d identifiers found\n", idNum);

printIdentifiers(idNum, idTable);

fprintIdentifiers(errorFile, idNum, idTable);

}

void match(enum TypeOfToken expectedType) {

if (TokenTable[pos].type == expectedType)

pos++;

else {

printf("\nSyntax error in line %d, token number: %d : another type of lexeme was expected (expected: %s | current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(expectedType), lexemeTypeName(TokenTable[pos].type));

fprintf(errorFile, "\nSyntax error in line %d, token number: %d : another type of lexeme was expected (expected: %s | current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(expectedType), lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

// функція записує оголошені ідентифікатори в таблицю ідентифікаторів idTable

// повертає кількість ідентифікаторів

// перевіряє чи усі використані ідентифікатори оголошені

unsigned int IdIdentification(struct id idTable[], struct Token TokenTable[], unsigned int tokenCount) {

unsigned int idCount = 0;

unsigned int i = 0;

while (TokenTable[i++].type != Variable && TokenTable[i].type != EndBlock);

while (TokenTable[i++].type == Type) {

while (TokenTable[i].type != Semicolon) {

if (TokenTable[i].type == Identifier) {

int yes = 0;

for (unsigned int j = 0; j < idCount; j++) {

if (!strcmp(TokenTable[i].name, idTable[j].name)) {

yes = 1;

break;

}

}

if (yes == 1) {

printf("\nidentifier \"%s\" is already declared !\n", TokenTable[i].name);

return idCount;

}

if (idCount < MAX\_IDENTIFIER) {

strcpy\_s(idTable[idCount++].name, TokenTable[i++].name);

}

else {

printf("\nToo many identifiers !\n");

return idCount;

}

}

else

++i;

}

++i;

}

for (; i < tokenCount; ++i) {

if (

TokenTable[i].type == Identifier

&& TokenTable[i - 1].type != Goto

&& TokenTable[i + 1].type != Colon

) {

bool yes = 0;

for (unsigned int j = 0; j < idCount; ++j) {

if (!strcmp(TokenTable[i].name, idTable[j].name)) {

yes = 1;

break;

}

}

if (!yes) {

printf("\nSemantic Error In line %d, an undeclared identifier \"%s\"!", TokenTable[i].line, TokenTable[i].name);

fprintf(errorFile, "\nSemantic Error In line %d, an undeclared identifier \"%s\"!", TokenTable[i].line, TokenTable[i].name);

exit(1);

}

}

}

return idCount; // Повертає кількість ідентифікаторів

}

void program() {

match(StartProgram);

match(StartBlock);

//match(Variable);

if (TokenTable[pos].type == Variable) {

++pos; // for VARIABLE

variableDeclaration();

match(Semicolon);

}

programBody();

match(EndBlock);

}

void variableDeclaration() {

match(Type);

variableList();

}

void variableList() {

match(Identifier);

while (TokenTable[pos].type == Comma) {

++pos; // for Comma

match(Identifier);

}

}

void statement() {

switch (TokenTable[pos].type) {

case Input: inputStatement(); break;

case Output: outputStatement(); break;

case If: ifStatement(); break;

case Goto: gotoStatement(); break;

case For: forStatement(); break;

case While: whileStatement(); break;

case Repeat: repeatStatement(); break;

case StartBlock:compoundStatement(); break;

default: {

if (TokenTable[pos + 1].type == Colon)

labelPoint();

else

assignStatement();

}

}

}

void inputStatement() {

++pos; // for Input

match(Identifier);

}

void outputStatement() {

++pos; // for Output

arithmeticExpression();

}

void arithmeticExpression() {

lowPriorityExpression();

//lowPriorityOperator

while (TokenTable[pos].type == Add || TokenTable[pos].type == Sub) {

++pos; // for + or -

lowPriorityExpression();

}

}

void lowPriorityExpression() {

middlePriorityExpression();

// middlePriorityOperator

while (TokenTable[pos].type == Mul || TokenTable[pos].type == Mod || TokenTable[pos].type == Div) {

++pos; // for \* or Mod or Div

middlePriorityExpression();

}

}

void middlePriorityExpression() {

switch (TokenTable[pos].type) {

case Identifier: ++pos; break;

case Number: ++pos; break;

case LBracket: {

++pos; // for (

arithmeticExpression();

match(RBracket);

break;

}

default: {

printf("\nSyntax error in line %d, token number: %d : middle priority operation was expected (current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));

fprintf(errorFile, "\nSyntax error in line %d, token number: %d : middle priority operation was expected (current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

}

void assignStatement() {

match(Identifier);

match(Assign);

arithmeticExpression();

}

void ifStatement() {

match(If);

match(LBracket);

logicalExpression();

match(RBracket);

statement();

if (TokenTable[pos].type == Else) {

//match(Semicolon);

++pos; // for else

statement();

}

}

void logicalExpression() {

andExpression();

while (TokenTable[pos].type == Or) {

++pos; // for ||

andExpression();

}

}

void andExpression() {

comparison();

while (TokenTable[pos].type == And) {

++pos; // for &&

andExpression();

}

}

void comparison() {

if (TokenTable[pos].type == Not) {

++pos; // for !

match(LBracket);

logicalExpression();

match(RBracket);

}

else if (TokenTable[pos].type == LBracket) {

++pos; // for (

logicalExpression();

match(RBracket);

} else {

comparisonExpression();

}

}

void comparisonExpression() {

arithmeticExpression();

if (TokenTable[pos].type == Equality ||

TokenTable[pos].type == NotEquality ||

TokenTable[pos].type == Greate ||

TokenTable[pos].type == Less

) {

++pos; // for "=" | "!=" | "<<" | ">>"

}

else {

printf("\nSyntax error in line %d : Comparison operator was Expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

fprintf(errorFile, "\nSyntax error in line %d : Comparison operator was Expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

arithmeticExpression();

}

void gotoStatement() {

match(Goto);

match(Identifier);

}

void labelPoint() {

match(Identifier);

match(Colon);

}

void forStatement() {

match(For);

assignStatement();

if (TokenTable[pos].type == To || TokenTable[pos].type == Downto) {

++pos; // for to or downto

}

else {

printf("\nSyntax error in line %d : TO or DOWNTO was expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

fprintf(errorFile, "\nSyntax error in line %d : TO or DOWNTO was expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

arithmeticExpression();

match(Do);

statement();

}

void whileStatement() {

match(While);

logicalExpression();

while (TokenTable[pos].type != End) {

statementInWhile();

//match(Semicolon);

}

//match(WEnd);

++pos; // for END

match(While);

}

void statementInWhile() {

switch (TokenTable[pos].type) {

case Continue: {

++pos; // for CONTINUE

match(While);

break;

}

case Exit: {

++pos; // for EXIT

match(While);

break;

}

default: {

statement();

break;

}

}

}

void repeatStatement() {

match(Repeat);

while (TokenTable[pos].type != Until) {

statement();

//match(Semicolon);

}

//match(Until);

++pos; // for UNTIL

match(LBracket);

logicalExpression();

match(RBracket);

}

void compoundStatement() {

match(StartBlock);

programBody();

match(EndBlock);

}

void programBody() {

while (TokenTable[pos].type != EndBlock) {

statement();

//match(Semicolon);

}

}

void printIdentifiers(int num, struct id\* table) {

for (int i = 0; i < num; i++) {

printf("%s\n", table[i].name);

}

}

void fprintIdentifiers(FILE\* F, int num, struct id\* table) {

for (int i = 0; i < num; i++) {

fprintf(F, "%s\n", table[i].name);

}

}

unsigned int LabelIdentification(struct id labelTable[], struct Token TokenTable[], unsigned int tokenCount) {

unsigned int labelNum = 0;

unsigned int i = 0;

unsigned int start = 0;

while (TokenTable[start++].type != StartProgram);

i = start;

while (TokenTable[i].type != EndBlock) {

if (TokenTable[i].type == Identifier && TokenTable[i + 1].type == Colon) {

strcpy\_s(labelTable[labelNum++].name, TokenTable[i].name);

}

++i;

}

i = start;

while (TokenTable[i].type != EndBlock) {

if (TokenTable[i].type == Identifier && TokenTable[i - 1].type == Goto) {

bool found = false;

for (unsigned int j = 0; j < labelNum; j++) {

if (strcmp(labelTable[j].name, TokenTable[i].name) == 0) {

found = true;

break;

}

}

if (!found) {

printf("\n Semantic error: In line %d label %s is not defined\n", TokenTable[i].line, TokenTable[i].name);

fprintf(errorFile, "\n Semantic error: In line %d label %s is not defined\n", TokenTable[i].line, TokenTable[i].name);

exit(1);

}

}

++i;

}

return labelNum;

}

}

Header.hpp

#pragma once

#define MAX\_TOKENS 1000

#define MAX\_IDENTIFIER 10

enum TypeOfToken {

StartProgram, // STARTPROGRAM

StartBlock, // STARTBLOCK

Variable, // VARIABLE

Type, // INT16

EndBlock, // ENDBLOCK

Input, // INPUT

Output, // OUTPUT

If, // IF

Else, // ELSE

Goto, // GOTO

For, // FOR

To, // TO

Downto, // DOWNTO

Do, // DO

While, // WHILE

End, // WEND

Continue, // CONTINUE

Exit, // EXIT

Repeat, // REPEAT

Until, // UNTIL

Identifier, // Identifier

Number, // number

Float, // float (incorrect)

Assign, // ==>

Add, // +

Sub, // -

Mul, // \*

Div, // DIV

Mod, // MOD

Equality, // =

NotEquality, // !=

Greate, // >>

Less, // <<

Not, // !!

And, // &&

Or, // ||

LBracket, // (

RBracket, // )

Semicolon, // ;

Colon, // :

Comma, // ,

Unknown\_

};

struct Token {

char name[16]; // ім'я лексеми

int value; // значення (для констант)

int line; // номер рядка

enum TypeOfToken type; // тип лексеми

};

struct id {

char name[16];

//unsigned int pos; //for labels

};

enum States {

Start, // початковий стан

Finish, // фінальний стан

Letter, // опрацювання слів (ключові слова та ідентифікаторів)

Digit, // опрацювання цифр

Separator, // опрацювання роздільників

Another, // опрацювання інших символів

EndOFile, // кінець файлу

Comment // ігнорування коментаря

};

enum TypeOfNode {

program\_node,

id\_node,

var\_node,

statement\_node,

input\_node,

output\_node,

add\_node,

sub\_node,

mul\_node,

div\_node,

mod\_node,

number\_node,

assign\_node,

if\_node,

else\_node,

or\_node,

and\_node,

not\_node,

eq\_node,

neq\_node,

gr\_node,

ls\_node,

goto\_node,

label\_node,

for\_node,

to\_node,

downto\_node,

while\_node,

continue\_node,

exit\_node,

repeat\_node,

compound\_node

};

struct astNode {

enum TypeOfNode type;

char name[16];

struct astNode\* left;

struct astNode\* right;

};

const char\* lexemeTypeName(enum TypeOfToken type);

Parser.hpp

#pragma once

#include "header.hpp"

#include <stdbool.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

extern struct Token\* TokenTable;

extern unsigned int TokensNum;

extern struct id\* idTable;

extern unsigned int idNum;

extern struct id\* labelTable;

extern unsigned int labelNum;

namespace Parser {

void Parser();

void Semantic();

void match(enum TypeOfToken expectedType);

unsigned int IdIdentification(struct id idTable[], struct Token TokenTable[], unsigned int tokenCount);

unsigned int LabelIdentification(struct id labelTable[], struct Token TokenTable[], unsigned int tokenCount);

void printIdentifiers(int num, struct id\* table);

void fprintIdentifiers(FILE\* F, int num, struct id\* table);

// program :: = "STARTPROGRAM", "STARTBLOCK", { "VARIABLE", variable\_declaration, ";" }, { statement }, "ENDBLOCK";

void program();

// programBody ::= { statement };

void programBody();

// variable\_declaration ::= "INT16", variable\_list;

void variableDeclaration();

// variable\_list :: = identifier, { ",", identifier };

void variableList();

// statement ::= input\_statement | output\_statement | assign\_statement | if\_else\_statement | goto\_statement | label\_point | for\_statement | while\_statement | repeat\_until\_statement | compound\_statement;

void statement();

// input\_statement ::= "INPUT", identifier;

void inputStatement();

// output\_statement ::= "OUTPUT", arithmetic\_expression;

void outputStatement();

// arithmetic\_expression :: = low\_priority\_expression{ low\_priority\_operator, low\_priority\_expression };

void arithmeticExpression();

// low\_priority\_expression :: = middle\_priority\_expression{ middle\_priority\_operator, middle\_priority\_expression };

void lowPriorityExpression();

// low\_priority\_operator :: = "+" | "-";

//void lowPriorityOperator();

// middle\_priority\_expression :: = identifier | number | "(", arithmetic\_expression, ")";

void middlePriorityExpression();

// middle\_priority\_operator :: = "\*" | "DIV" | "MOD";

//void middlePriorityOperator();

// assign\_statement :: = arithmetic\_expression, "==>", identifier;

void assignStatement();

// if\_else\_statement :: = "IF", "(", logical\_expression, ")", statement, [ "ELSE", statement];

void ifStatement();

// logical\_expression :: = and\_expression{ or\_operator, and\_expression };

void logicalExpression();

// or\_operator :: = "||";

//void orOperator();

// and\_expression :: = comparison{ and\_operator, and\_expression };

void andExpression();

// and\_operator :: = "&&";

// void andOperator();

// comparison :: = comparison\_expression | [not\_operator] "(", logical\_expression, ")";

void comparison();

// not\_operator :: = "!!";

// void notOperator();

// comparison\_expression :: = arithmetic\_expression comparison\_operator arithmetic\_expression;

void comparisonExpression();

// comparison\_operator :: = "=" | "!=" | "<<" | ">>";

// void comparisonOperator();

// goto\_statement :: = "GOTO", identifier;

void gotoStatement();

// label\_point :: = identifier, ":";

void labelPoint();

// for\_to\_statement :: = "FOR", assign\_statement, "TO" | "DOWNTO", arithmetic\_expression, "DO", statement;

void forStatement();

// while\_statement :: = "WHILE", logical\_expression, { statement }, "END", "WHILE";

void whileStatement();

void statementInWhile();

// repeat\_until\_statement :: = "REPEAT", { statement }, "UNTIL", "(", logical\_expression, ")";

void repeatStatement();

// compoundStatement :: = "STARTBLOCK", { statement }, "ENDBLOCK";

void compoundStatement();

}

Ast.hpp

#pragma once

#include "header.hpp"

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

extern struct Token\* TokenTable;

extern int pos;

namespace AST {

// program :: = "STARTPROGRAM", "STARTBLOCK", { "VARIABLE", variable\_declaration, ";" }, { statement }, "ENDBLOCK";

struct astNode\* program();

// programBody ::= { statement };

struct astNode\* programBody();

// variable\_declaration ::= "INT16", variable\_list;

struct astNode\* variableDeclaration();

// variable\_list ::= identifier, {",", identifier};

struct astNode\* variableList();

// statement ::= input\_statement | output\_statement | assign\_statement | if\_else\_statement | goto\_statement | label\_point | for\_statement | while\_statement | repeat\_until\_statement | compound\_statement;

struct astNode\* statement();

// input\_statement ::= "INPUT", identifier;

struct astNode\* inputStatement();

// output\_statement ::= "OUTPUT", arithmetic\_expression;

struct astNode\* outputStatement();

// arithmetic\_expression :: = low\_priority\_expression{ low\_priority\_operator, low\_priority\_expression };

struct astNode\* arithmeticExpression();

// low\_priority\_expression :: = middle\_priority\_expression{ middle\_priority\_operator, middle\_priority\_expression };

struct astNode\* lowPriorityExpression();

// low\_priority\_operator :: = "+" | "-";

//void lowPriorityOperator();

// middle\_priority\_expression :: = identifier | number | "(", arithmetic\_expression, ")";

struct astNode\* middlePriorityExpression();

// middle\_priority\_operator :: = "\*" | "DIV" | "MOD";

//struct astNode\* middlePriorityOperator();

// assign\_statement :: = arithmetic\_expression, "==>", identifier;

struct astNode\* assignStatement();

// if\_else\_statement :: = "IF", "(", logical\_expression, ")", statement, [ "ELSE", statement];

struct astNode\* ifStatement();

// logical\_expression :: = and\_expression{ or\_operator, and\_expression };

struct astNode\* logicalExpression();

// or\_operator :: = "||";

//struct astNode\* orOperator();

// and\_expression :: = comparison{ and\_operator, and\_expression };

struct astNode\* andExpression();

// and\_operator :: = "&&";

// struct astNode\* andOperator();

// comparison :: = comparison\_expression | [not\_operator] "(", logical\_expression, ")";

struct astNode\* comparison();

// not\_operator :: = "!!";

// struct astNode\* notOperator();

// comparison\_expression :: = arithmetic\_expression comparison\_operator arithmetic\_expression;

struct astNode\* comparisonExpression();

// comparison\_operator :: = "=" | "!=" | "<<" | ">>";

// struct astNode\* comparisonOperator();

// goto\_statement :: = "GOTO", identifier;

struct astNode\* gotoStatement();

// label\_point :: = identifier, ":";

struct astNode\* labelPoint();

// for\_to\_statement :: = "FOR", assign\_statement, "TO" | "DOWNTO", arithmetic\_expression, "DO", statement;

struct astNode\* forStatement();

// while\_statement :: = "WHILE", logical\_expression, { statement\_in\_while }, "END", "WHILE";

struct astNode\* whileStatement();

// statement\_in\_while :: = statement | "CONTINUE WHILE" | "EXIT WHILE";

struct astNode\* statementInWhile();

struct astNode\* whileBody();

// repeat\_until\_statement :: = "REPEAT", repeat\_body, "UNTIL", "(", logical\_expression, ")";

struct astNode\* repeatStatement();

// repeat\_body :: = { statement };

struct astNode\* repeatBody();

// compoundStatement :: = "STARTBLOCK", { statement }, "ENDBLOCK";

struct astNode\* compoundStatement();

void deleteNode(struct astNode\* node);

struct astNode\* createNode(enum TypeOfNode type, const char\* name, struct astNode\* left, struct astNode\* right);

void printAST(struct astNode\* node, int level);

void fPrintAST(FILE\* outFile, struct astNode\* node, int level);

void match(enum TypeOfToken expectedType);

struct astNode\* astParser();

}

Codegen.cpp

#include "Codegen.hpp"

namespace Codegen {

void codegen(FILE\* outFile, struct astNode\* node) {

if (node == 0) {

return;

}

switch (node->type) {

case program\_node: {

fprintf(outFile, "#include <stdio.h>\n");

fprintf(outFile, "int main() {\n");

codegen(outFile, node->left); // for declaration

fprintf(outFile, "\n");

codegen(outFile, node->right); // for statements

fprintf(outFile, "return 0;\n}\n");

break;

}

case var\_node: {

fprintf(outFile, "int ");

codegen(outFile, node->left);

fprintf(outFile, ";\n");

codegen(outFile, node->right);

break;

}

case number\_node:

case id\_node: {

fprintf(outFile, "%s", node->name);

break;

}

case statement\_node: {

codegen(outFile, node->left);

codegen(outFile, node->right);

break;

}

case input\_node: {

fprintf(outFile, "printf(\"Enter ");

codegen(outFile, node->left);

fprintf(outFile, ": \");\n");

fprintf(outFile, "scanf(\"%%d\", &");

codegen(outFile, node->left);

fprintf(outFile, ");\n");

break;

}

case output\_node: {

fprintf(outFile, "printf(\"%%d\\n\", ");

codegen(outFile, node->left);

fprintf(outFile, ");\n");

break;

}

case add\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " + ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case sub\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " - ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case mul\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " \* ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case div\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " / ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case mod\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " %% ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case assign\_node: {

codegen(outFile, node->left);

fprintf(outFile, " = ");

codegen(outFile, node->right);

fprintf(outFile, ";\n");

break;

}

case if\_node: {

fprintf(outFile, "if (");

codegen(outFile, node->left);

fprintf(outFile, ") ");

codegen (outFile, node->right);

break;

}

case else\_node: {

codegen(outFile, node->left);

fprintf(outFile, "else ");

codegen(outFile, node->right);

break;

}

case or\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " || ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case and\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " && ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case not\_node: {

fprintf(outFile, "!(");

codegen(outFile, node->left);

fprintf(outFile, ")");

break;

}

case eq\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " == ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case neq\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " != ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case gr\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " > ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case ls\_node: {

fprintf(outFile, "(");

codegen(outFile, node->left);

fprintf(outFile, " < ");

codegen(outFile, node->right);

fprintf(outFile, ")");

break;

}

case goto\_node: {

fprintf(outFile, "goto ");

codegen(outFile, node->left);

fprintf(outFile, ";\n");

break;

}

case label\_node: {

codegen(outFile, node->left);

fprintf(outFile, ":\n");

break;

}

case for\_node: {

fprintf(outFile, "for(\n");

codegen(outFile, node->left);

fprintf(outFile, "\n) ");

codegen(outFile, node->right);

break;

}

case to\_node: {

codegen(outFile, node->left);

codegen(outFile, node->left->left);

fprintf(outFile, " <= ");

codegen(outFile, node->right);

fprintf(outFile, ";\n++");

codegen(outFile, node->left->left);

break;

}

case downto\_node: {

codegen(outFile, node->left);

codegen(outFile, node->left->left);

fprintf(outFile, " >= ");

codegen(outFile, node->right);

fprintf(outFile, ";\n--");

codegen(outFile, node->left->left);

break;

}

case while\_node: {

fprintf(outFile, "while(");

codegen(outFile, node->left);

fprintf(outFile, ") {\n");

codegen(outFile, node->right);

fprintf(outFile, "}\n");

break;

}

case continue\_node: {

fprintf(outFile, "continue;\n");

break;

}

case exit\_node: {

fprintf(outFile, "break;\n");

break;

}

case repeat\_node: {

fprintf(outFile, "do {\n");

codegen(outFile, node->left);

fprintf(outFile, "} while(");

codegen(outFile, node->right);

fprintf(outFile, ");\n");

break;

}

case compound\_node: {

fprintf(outFile, "{\n");

codegen(outFile, node->left);

codegen(outFile, node->right);

fprintf(outFile, "}\n");

break;

}

default: {

exit(1);

printf("Undescribed node type: %d\n", node->type);

break;

}

}

}

}

Ast.cpp

#include "Ast.hpp"

extern struct Token\* TokenTable;

extern int pos;

namespace AST {

void deleteNode(struct astNode\* node) {

if (node == nullptr) return;

deleteNode(node->left);

deleteNode(node->right);

free(node);

}

struct astNode\* createNode(enum TypeOfNode type, const char\* name, struct astNode\* left, struct astNode\* right) {

struct astNode\* node = (struct astNode\*)malloc(sizeof(struct astNode));

node->type = type;

strcpy\_s(node->name, name);

node->left = left;

node->right = right;

return node;

}

void printAST(struct astNode\* node, int level) {

if (node == nullptr)

return;

for (int i = 0; i < level; i++)

printf("| ");

printf("|-- %s(%d)", node->name, node->type);

printf("\n");

if (node->left || node->right)

{

printAST(node->left, level + 1);

printAST(node->right, level + 1);

}

}

void fPrintAST(FILE\* outFile, struct astNode\* node, int level) {

if (node == nullptr)

return;

for (int i = 0; i < level; i++)

fprintf(outFile, "| ");

fprintf(outFile, "|-- %s(%d)", node->name, node->type);

fprintf(outFile, "\n");

if (node->left || node->right)

{

fPrintAST(outFile, node->left, level + 1);

fPrintAST(outFile, node->right, level + 1);

}

}

void match(enum TypeOfToken expectedType) {

if (TokenTable[pos].type == expectedType)

pos++;

else {

printf("\nSyntax error in line %d : another type of lexeme was expected (expected: %s | current: %s).\n", TokenTable[pos].line, lexemeTypeName(expectedType), lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

struct astNode\* astParser() {

pos = 0;

struct astNode\* tree = program();

printf("AST created.\n");

return tree;

}

struct astNode\* program() {

match(StartProgram);

match(StartBlock);

struct astNode\* declaration = nullptr;

if (TokenTable[pos].type == Variable) {

++pos; // for VARIABLE

declaration = variableDeclaration();

match(Semicolon);

}

struct astNode\* body = programBody();

match(EndBlock);

return createNode(program\_node, "Program", declaration, body);

}

struct astNode\* variableDeclaration() {

match(Type);

return variableList();

}

struct astNode\* variableList() {

match(Identifier);

struct astNode\* id = createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr);

struct astNode\* list = list = createNode(var\_node, "var", id, nullptr);

while (TokenTable[pos].type == Comma)

{

match(Comma);

match(Identifier);

id = createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr);

list = createNode(var\_node, "var", id, list);

}

return list;

}

struct astNode\* programBody() {

if (TokenTable[pos].type != EndBlock) {

struct astNode\* stmt = statement();

//match(Semicolon);

struct astNode\* body = stmt;

while (TokenTable[pos].type != EndBlock)

{

struct astNode\* nextStmt = statement();

//match(Semicolon);

body = createNode(statement\_node, "statement", body, nextStmt);

}

return body;

}

return nullptr;

}

struct astNode\* statement() {

switch (TokenTable[pos].type) {

case Input: return inputStatement();

case Output: return outputStatement();

case If: return ifStatement();

case Goto: return gotoStatement();

case For: return forStatement();

case While: return whileStatement();

case Repeat: return repeatStatement();

case StartBlock:return compoundStatement();

default: {

if (TokenTable[pos + 1].type == Colon)

return labelPoint();

else

return assignStatement();

}

}

}

struct astNode\* inputStatement() {

match(Input);

match(Identifier);

return createNode(input\_node, "input", createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr), nullptr);

}

struct astNode\* outputStatement() {

match(Output);

return createNode(output\_node, "output", arithmeticExpression(), nullptr);

}

struct astNode\* arithmeticExpression() {

struct astNode\* left = lowPriorityExpression();

if (TokenTable[pos].type == Add || TokenTable[pos].type == Sub) {

enum TypeOfToken op = TokenTable[pos].type;

++pos; // for add or sub

struct astNode\* right = arithmeticExpression();

return createNode(op == Add ? add\_node : sub\_node, lexemeTypeName(op), left, right);

}

return left;

}

struct astNode\* lowPriorityExpression() {

struct astNode\* left = middlePriorityExpression();

if (TokenTable[pos].type == Mul || TokenTable[pos].type == Mod || TokenTable[pos].type == Div) {

enum TypeOfToken op = TokenTable[pos].type;

++pos; // for mul or mod or div

struct astNode\* right = lowPriorityExpression();

return createNode(op == Mul ? mul\_node : op == Div ? div\_node : mod\_node, lexemeTypeName(op), left, right);

}

return left;

}

struct astNode\* middlePriorityExpression() {

switch (TokenTable[pos].type) {

case Identifier: match(Identifier); return createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr);

case Number: match(Number); return createNode(number\_node, TokenTable[pos - 1].name, nullptr, nullptr);

case LBracket: {

++pos; // for (

struct astNode\* expr = arithmeticExpression();

match(RBracket);

return expr;

}

default: {

printf("\nSyntax error in line %d, token number: %d : middle priority operation was expected (current: %s).\n", TokenTable[pos].line, pos, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

}

struct astNode\* assignStatement() {

match(Identifier);

struct astNode\* left = createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr);

match(Assign);

struct astNode\* right = arithmeticExpression();

return createNode(assign\_node, "assign", left, right);

}

struct astNode\* ifStatement() {

match(If);

match(LBracket);

struct astNode\* expr = logicalExpression();

match(RBracket);

struct astNode\* stmt = statement();

if (TokenTable[pos].type == Else) {

//match(Semicolon);

++pos; // for ELSE

struct astNode\* elseStmt = statement();

stmt = createNode(else\_node, "else", stmt, elseStmt);

}

return createNode(if\_node, "if", expr, stmt);

}

struct astNode\* logicalExpression() {

struct astNode\* left = andExpression();

if (TokenTable[pos].type == Or) {

++pos; // for Or

struct astNode\* right = logicalExpression();

return createNode(or\_node, "or", left, right);

}

return left;

}

struct astNode\* andExpression() {

struct astNode\* left = comparison();

if (TokenTable[pos].type == And) {

++pos; // for And

struct astNode\* right = andExpression();

return createNode(and\_node, "and", left, right);

}

return left;

}

struct astNode\* comparison() {

struct astNode\* comp;

switch (TokenTable[pos].type) {

case Not: {

++pos; // for Not

match(LBracket);

comp = createNode(not\_node, "not", logicalExpression(), nullptr);

match(RBracket);

break;

}

case LBracket: {

++pos; // for (

comp = logicalExpression();

match(RBracket);

break;

}

default: {

comp = comparisonExpression();

}

}

return comp;

}

struct astNode\* comparisonExpression() {

struct astNode\* left = arithmeticExpression();

if (TokenTable[pos].type == Equality ||

TokenTable[pos].type == NotEquality ||

TokenTable[pos].type == Greate ||

TokenTable[pos].type == Less

) {

enum TypeOfToken op = TokenTable[pos].type;

++pos; // for Equality or NotEquality or Greate or Less

struct astNode\* right = arithmeticExpression();

return createNode(op == Equality ? eq\_node : op == NotEquality ? neq\_node : op == Greate ? gr\_node : ls\_node, lexemeTypeName(op), left, right);

}

else {

printf("\nSyntax error in line %d : Comparison operator was Expected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

struct astNode\* gotoStatement() {

match(Goto);

match(Identifier);

return createNode(goto\_node, "goto", createNode(id\_node, TokenTable[pos - 1].name, nullptr, nullptr), nullptr);

}

struct astNode\* labelPoint() {

match(Identifier);

match(Colon);

return createNode(label\_node, "label", createNode(id\_node, TokenTable[pos - 2].name, nullptr, nullptr), nullptr);

}

struct astNode\* forStatement() {

match(For);

struct astNode\* assign = assignStatement();

switch (TokenTable[pos].type) {

case To: {

++pos; // for To

struct astNode\* expr = arithmeticExpression();

match(Do);

return createNode(for\_node, "for", createNode(to\_node, "to", assign, expr), statement());

}

case Downto: {

++pos; // for Downto

struct astNode\* expr = arithmeticExpression();

match(Do);

return createNode(for\_node, "for", createNode(to\_node, "downto", assign, expr), statement());

}

default: {

printf("\nSyntax error in line %d : TO | DOWNTO operator was xpected, %s token gained).\n", TokenTable[pos].line, lexemeTypeName(TokenTable[pos].type));

exit(1);

}

}

}

struct astNode\* whileStatement() {

match(While);

struct astNode\* expr = logicalExpression();

struct astNode\* body = whileBody();

match(End);

match(While);

return createNode(while\_node, "while", expr, body);

}

struct astNode\* whileBody() {

if (TokenTable[pos].type != End) {

struct astNode\* stmt = statementInWhile();

//match(Semicolon);

struct astNode\* body = stmt;

while (TokenTable[pos].type != End)

{

struct astNode\* nextStmt = statementInWhile();

//match(Semicolon);

body = createNode(statement\_node, "statement", body, nextStmt);

}

return body;

}

return nullptr;

}

struct astNode\* statementInWhile() {

switch (TokenTable[pos].type) {

case Continue: {

++pos; // for CONTINUE

match(While);

return createNode(continue\_node, "continue", nullptr, nullptr);

}

case Exit: {

++pos; // for EXIT

match(While);

return createNode(exit\_node, "exit", nullptr, nullptr);

}

default: return statement();

}

}

struct astNode\* repeatStatement() {

match(Repeat);

struct astNode\* body = repeatBody();

match(Until);

match(LBracket);

struct astNode\* expr = logicalExpression();

match(RBracket);

return createNode(repeat\_node, "repeat", body, expr);

}

struct astNode\* repeatBody() {

if (TokenTable[pos].type != Until) {

struct astNode\* stmt = statement();

match(Semicolon);

struct astNode\* body = stmt;

while (TokenTable[pos].type != Until)

{

struct astNode\* nextStmt = statement();

//match(Semicolon);

body = createNode(statement\_node, "statement", body, nextStmt);

}

return body;

}

return nullptr;

}

struct astNode\* compoundStatement() {

match(StartBlock);

struct astNode\* body = programBody();

match(EndBlock);

return createNode(compound\_node, "compound", body, nullptr);

}

}