**Завдання на курсову роботу для варіанту 18**

Цільова мова транслятора – асемблер для 32/64-розрядного процесора (x86).

Для отримання виконавчого файлу на виході розробленого транслятора скористатися програмами ml.exe і link.exe (або ml64.exe і link64.exe).

Мова розробки транслятора: ANSI C або C++.

Інтерфейс: Реалізувати оболонку або інтерфейс з командного рядка.

Вхідні дані: На вхід розробленого транслятора має подаватися текстовий файл, написаний на заданій мові програмування.

Вихідні файли транслятора:

файл з лексемами;

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

файл на мові асемблера;

об’єктний файл;

виконавчий файл.

Назва вхідної мови програмування:

Утворюється від першої букви у прізвищі студента та останніх двох цифр номера його варіанту.

У моєму випадку це .p18.

Опис вхідної мови програмування:

Типи даних: Int32 (замість Integer\_2 для твого варіанту).

Оператор вводу: Scan().

Оператор виводу: Print().

Блок тіла програми: StartProgram <name> StartBlok Variable… EndBlok.

Оператор циклу: While (<умова>) Do.

Регістр ключових слів: Початковий символ — велика літера, інші — малі.

Регістр ідентифікаторів: Початкова велика літера, наступні — малі.

Арифметичні операції: ++, --, \*\* (піднесення до степеня), Div, Mod.

Операції порівняння: Eq (==), Neq (!=), Gr (>=), Ls (<=).

Логічні операції: Not, And, Or.

Коментарі: Використовувати формат коментарів { ... } (замість ##...##).

Оператор присвоєння: ==> (замість <-).

Програмне забезпечення:

Для отримання виконавчого файлу на виході розробленого транслятора скористатися програмами ml.exe (асемблер) та link.exe (редактор зв’язків).

# ВСТУП

Транслятором називається програма перекладу (трансляції) початкової програми, записаною вхідною мовою, в еквівалентну їй об`єктну програму. Якщо мова високого рівня є вхідною, а мова асемблера чи машинна – вихідною, то такий транслятор називається компілятором. [1](5-7 ст.)

Транслятори бувають двох типів: компілятори і інтерпретатори. Процес компіляції складається з двох частин: аналізу і синтезу. На етапі аналізу вхідну програму розбивають на окремі елементи (лексеми[2]), перевіряють її на відповідність правилам граматики і створюють проміжне представлення вхідної програми. На етапі синтезу із проміжної форми створюють програму в машинних кодах. Така програма називається об’єктною програмою. В подальшому об’єктну програму можна виконати на комп’ютері без перетрансляції.

Інтерпретатор — це тип транслятора, який виконує текст програми, написаної мовою програмування, безпосередньо на комп'ютері. На відміну від компіляторів, інтерпретатор не створює нової програми, а просто виконує – інтерпретує кожен оператор вхідної мови програмування. Подібно компілятору, інтерпретатор аналізує вхідну програму, створює проміжне представлення, однак не створює об’єктної програми, а зразу виконує передбачені вхідною програмою команди.

Компілятор[3] перекладає програму з однієї мови на іншу. На вхід компілятора поступає ланцюг символів, який є вхідною програмою на певній мові програмування. На виході компілятора (об’єктна програма) також представляє собою ланцюг символів, який вже належить іншій мові програмування, наприклад машинній мові деякого комп’ютера. При цьому сам компілятор може бути написаний третьою мовою.

# 1. Огляд методів та способів проектування трансляторів

Транслятор - обслуговуюча програма, що перетворює вихідну програму, надану на вхідній мові програмування, у робочу програму, представлену на об'єктному мовою. [2](7-8 ст).

Наведене визначення відноситься до всіх різновидів трансляторів програм. Однак у кожної з таких програм можуть бути свої особливості щодо організації процесу трансляції. В даний час транслятори поділяються на три основні групи: асемблери, компілятори та інтерпретатори.

Асемблер — це програма, яка перетворює текст програми, написаної мовою асемблера, на машинний код. Мова асемблера — це низькорівнева мова програмування, яка використовує мнемонічні коди для позначення машинних інструкцій. Специфічною рисою асемблером є те, що вони здійснюють дослівну трансляцію однієї символічної команди в одну машинну. [2](10-11 ст).

Компілятор - це обслуговуюча програма, що виконує трансляцію на машинну мову програми, записаної мовою оригіналу програмування. Також як і асемблер, компілятор забезпечує перетворення програми з однієї мови на іншу (найчастіше, в мову конкретного комп'ютера). [2](13-14 ст).

Інтерпретатор - програма або пристрій, що здійснює пооператорну трансляцію і виконання вихідної програми. На відміну від компілятора, інтерпретатор не породжує на виході програму на машинній мові. Розпізнавши команду вихідного мови, він тут же виконує її. Як у компіляторах, так і в інтерпретатора використовуються однакові методи аналізу вихідного тексту програми. Але інтерпретатор дозволяє почати обробку даних після написання навіть однієї команди. Це робить процес розробки і налагодження програм більш гнучким. [2](16-17 ст).

Процес трансляції складається з наступних фаз: лексичний аналіз, синтаксичний та семантичний аналіз, оптимізація коду, генерація коду.

На фазі лексичного аналізу вхідна програма, що представляє собою потік літер, розбивається на лексеми - слова у відповідності з визначеннями мови. Основними формалізму, які лежать в основі реалізації лексичних аналізаторів, є кінцеві автомати та регулярні вирази. Лексичний аналізатор може працювати в двох основних режимах: або як підпрограма, що викликається синтаксичним аналізатором для отримання чергової лексеми, або як повний прохід, результатом якого є файл лексем.

У процесі виділення лексем лексичний аналізатор може як самостійно будувати таблиці об'єктів (ідентифікаторів, рядків, чисел і т.д.), так і видавати значення для кожної лексеми при черговому до нього зверненні. У цьому випадку таблиці об'єктів будуються в наступних фазах (наприклад, в процесі синтаксичного аналізу).

На етапі лексичного аналізу виявляються деякі (найпростіші) помилки (неприпустимі символи, неправильний запис чисел, ідентифікаторів та ін.) Основне завдання синтаксичного аналізу - розбір структури програми. Як правило, під структурою розуміється дерево, відповідне розбору в контекстно-вільної граматики мови. В даний час найчастіше використовується або LL (1)-аналіз (і його варіант - рекурсивний спуск), або LR (1) - аналіз та його варіанти (LR (0), SLR (1),

LALR (1) та інші) . Рекурсивний спуск частіше використовується при ручному програмуванні синтаксичного аналізатора, LR (1) - при використанні систем автоматичного побудови синтаксичних аналізаторів. Результатом синтаксичного аналізу є синтаксичне дерево з посиланнями на таблиці об'єктів. У процесі синтаксичного аналізу також виявляються помилки, пов'язані зі структурою програми.

На етапі контекстного аналізу виявляються залежності між частинами програми, які не можуть бути описані контекстно-вільним синтаксисом. Це в основному зв'язку «опис-використання», зокрема, аналіз типів об'єктів, аналіз областей видимості, відповідність параметрів, мітки та інші. У процесі контекстного аналізу таблиці об'єктів поповнюються інформацією про описах (властивостях) об'єктів.

Основним формалізмом, що використовується при контекстному аналізі, є апарат атрибутних граматик. Результатом контекстного аналізу є атрибутоване дерево програми. Інформація про об'єкти може бути як розосереджена в самому дереві, так і зосереджена в окремих таблицях об'єктів. У процесі контекстного аналізу також можуть бути виявлені помилки, пов'язані з неправильним використанням об'єктів.

Потім програма може бути переведена у внутрішнє представлення. Це робиться для цілей оптимізації та / або зручності генерації коду. Ще однією метою перетворення програми у внутрішнє представлення є бажання мати переносимий компілятор. Тоді тільки остання фаза (генерація коду) є машинно-залежною. В якості внутрішнього подання може використовуватися префіксний або постфіксній запис, орієнтований граф, трійки, четвірки та інші.

Фаз оптимізації може бути декілька. Оптимізації зазвичай ділять на машиннозалежні та машинно-незалежні, локальні і глобальні. Частина машинно-залежною оптимізації виконується на фазі генерації коду. Глобальна оптимізація намагається взяти до уваги структуру всієї програми, локальна - тільки невеликих її фрагментів. Глобальна оптимізація грунтується на глобальному потоковому аналізі, який виконується на графі програми і представляє по суті перетворення цього графа. При цьому можуть враховуватися такі властивості програми, як міжпроцедурний аналіз, міжмодульний аналіз, аналіз галузей життя змінних і т.д.

Нарешті, генерація коду - остання фаза трансляції. Результатом її є або асемблерний модуль, або об'єктний (або завантажувальний) модуль. У процесі створення коду можуть виконуватися деякі локальні оптимізації, такі як розподіл регістрів, вибір довгих або коротких переходів, облік вартості команд при виборі конкретної послідовності команд. Для генерації коду розроблені різні методи, такі як таблиці рішень, зіставлення зразків, що включає динамічне програмування, різні синтаксичні методи.

Звичайно, ті чи інші фази транслятора можуть або відсутні зовсім, або об'єднуватися. У найпростішому випадку однопрохідного транслятора немає явної фази генерації проміжного представлення та оптимізації, інші фази об'єднані в одну, причому немає і явно побудованого синтаксичного дерева.

# 2. Формальний опис вхідної мови програмування

2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура

Опис вхідної мови програмування в термінах форми Бекуса-Наура:

Однією з перших задач, що виникають при побудові компілятора, є визначення вхідної мови програмування. Для цього використовують різні способи формального опису, отож я застосувала розширену нотацію Бекуса-Наура (Backus/Naur Form - BNF).

<program> ::= StartProgram <ident> StartBlok <var\_block> <code\_block> EndBlok

<var\_block> ::= Variable <ident> [{, <ident>}]

<ident> ::= <letter> [{<letter> | <digit>}]

<letter> ::= a | b | c | ... | z | A | B | C | ... | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<code\_block> ::= <statement> [{<statement>}]

<statement> ::= <assignment>

| <if\_statement>

| <loop\_statement>

| <input>

| <output>

| <goto\_statement>

<assignment> ::= <ident> ==> <expression>

<if\_statement> ::= If (<expression\_l>) Then <code\_block> [Else <code\_block>]

<loop\_statement> ::= For <ident> = <expression> To <expression> Do <code\_block>

| For <ident> = <expression> Downto <expression> Do <code\_block>

| While (<expression\_l>) Do <code\_block>

| Repeat <code\_block> Until (<expression\_l>)

<input> ::= Scan(<ident>)

<output> ::= Print(<expression>)

<goto\_statement> ::= Goto (<label>)

<expression> ::= <term> [{<operation\_a> <term>}]

<expression\_l> ::= <term> [{<operation\_l> <term>}]

<term> ::= <ident> | <number>

<number> ::= <digit> [{<digit>}]

<operation\_a> ::= ++ | -- | \*\* | Div | Mod

<operation\_l> ::= Eq | Neq | Gr | Ls | And | Or | Not

<label> ::= <ident>

**Опис ключових слів:**

StartProgram: Початок програми.

StartBlok: Початок блоку основного коду програми.

EndBlok: Кінець блоку коду.

Variable: Початок блоку оголошення змінних.

Scan(): Оператор для вводу даних.

Print(): Оператор для виводу даних.

If-Else: Умовний оператор.

For-To, For-Downto, While, Repeat-Until: Циклічні оператори.

==>: Оператор присвоєння.

Goto: Оператор безумовного переходу.

Eq, Neq, Gr, Ls: Оператори порівняння (дорівнює, не дорівнює, більше, менше).

And, Or, Not: Логічні оператори.

## **2.2. Опис термінальних символів та ключових слів**

Визначимо окремі термінальні символи та нерозривні набори термінальних символів (ключові слова):

*Таблиця 2.2.1*

|  |  |
| --- | --- |
| Термінальний символ або ключове слово | Значення |
| StartProgram | Початок програми. |
| StartBlok | Початок блоку основного коду програми. |
| EndBlok | Кінець блоку коду. |
| Variable | Початок блоку оголошення змінних. |
| Scan | Оператор для вводу даних. |
| Print | Оператор для виводу даних. |
| ==> | Оператор присвоєння. |
| If | Оператор умовного переходу. |
| Else | Оператор альтернативної умови. |
| For | Оператор циклу з параметром. |
| To | Використовується в циклах для задання верхньої межі. |
| Downto | Використовується в циклах для задання нижньої межі. |
| While | Оператор циклу з умовою. |
| Repeat | Оператор початку циклу. |
| Until | Оператор завершення циклу. |
| Goto | Оператор безумовного переходу. |
| Eq | Оператор порівняння (дорівнює). |
| Neq | Оператор порівняння (не дорівнює). |
| Gr | Оператор порівняння (більше або дорівнює). |
| Ls | Оператор порівняння (менше або дорівнює). |
| And | Логічний оператор (І). |
| Or | Логічний оператор (АБО). |
| Not | Логічний оператор заперечення. |
| ++ | Оператор інкременту (збільшення на 1). |
| -- | Оператор декременту (зменшення на 1). |
| \*\* | Оператор піднесення до степеня. |
| Div | Оператор цілочисельного ділення. |
| Mod | Оператор залишку від ділення. |
| {...} | Коментарі. |
| 0-9 | Цифри для числових констант. |
| a-z, A-Z | Літери для ідентифікаторів. |
| ( | Відкриваюча кругла дужка. |
| ) | Закриваюча кругла дужка. |
| ; | Кінець оператора. |
| , | Розділювач у списках змінних. |