МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЦЕНТРАЛЬНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

до виконання лабораторних робіт з навчальної дисципліни «Системне програмне забезпечення» для студентів денної та заочної форми навчання за спеціальностями
123 «Комп’ютерна інженерія»,
122 «Комп’ютерні науки»

ЗАТВЕРДЖЕНО
на засіданні кафедри кібербезпеки та програмного забезпечення,
протокол №1 від 25.08.2025 року

КРОПИВНИЦЬКИЙ
2025

Методичні рекомендації до виконання лабораторних робіт з навчальної дисципліни «Системне програмне забезпечення» для студентів денної та заочної форми навчання за спеціальностями 123«Комп'ютерна інженерія», 122«Комп'ютерні науки» / уклад. Дрєєва Г.М., Смірнов С.А. — Кропивницький: ЦНТУ, 2025. — 99 с.

Укладачі: Дрєєва Г.М., Смірнов С.А.

Рецензенти: Смірнов О. А., д-р техн. наук, професор;
Коваленко О.В., д-р техн. наук, професор.

© Дрєєва Г.М., Смірнов С.А., укладання, 2025
© Центральноукраїнський національний технічний університет, 2025

Вступ

В методичних рекомендаціях представлено принципи розробки трансляторів та компіляторів. Викладено основи аналізу, оптимізації та синтезу програмного коду.

Програма – це послідовність команд, яку виконує комп’ютер в процесі обробки інформації. Для того, щоб комп’ютер міг виконувати ту чи іншу програму, вона має завантажитися в оперативну пам’ять, оскільки саме з нею працює процесор. Всі програми, які містяться в комп’ютері називаються програмним забезпеченням або програмною конфігурацією відповідного комп’ютера.

Програмне забезпечення можна розподілити за наступними рівнями:

- базовий рівень;
- системний рівень;
- службовий рівень;
- прикладний рівень.

Програми системного рівня керують згодженою роботою всіх елементів системи як апаратного, так і програмного забезпечення. Системні програми, як правило, орієнтовані на кваліфікованих користувачів-фахівців в комп’ютерній галузі: системних програмістів, адміністраторів тощо.

До програм системного рівня відносяться: ·

операційні системи – це комплекс програм, які здійснюють діалог з користувачем, забезпечують керування ресурсами комп’ютера, запускають інші програми до виконання; ·

інтерфейсні оболонки – виконують посередницькі функції, забезпечують ефективну взаємодію користувача та комп’ютера; ·

драйвери – спеціальні програми для керування зовнішніми пристроями. Під час під’єднання будь-якого нового пристрою обов’язково має бути встановлений драйвер, інакше система не буде знати, які функції виконує пристрій і як ним керувати. ·

компілятори - програми, що перетворюють команди мови програмування

у машинний код.

Теми лабораторних робіт, що розглядаються та оцінювання знань

Дані методичні рекомендації містять у собі матеріал з 4 лабораторних робіт за наступними темами:

Теми лабораторних робіт
1. **Організація таблиць ідентифікаторів**
2. Проектування лексичного аналізатора
3. Побудова простого дерева виведення
4. Генерація і оптимізація об'єктного коду

Здобувачам потрібно виконати завдання до лабораторних робіт, а також відповісти на запитання, що знаходяться в кінці кожної лабораторної роботи.Звіт повинен містити хід виконання завдань а також графічні матеріали, що підтверджують виконання цих завдань. Приклад звіту знаходиться у додатку 1.

Максимальну кількість балів студент може одержати у випадку відвідування всіх лекцій, лабораторних занять, виконання і захисту виконаних завдань для самостійної роботи у встановлений термін та проходження контролю вчасно.

У разі виконання й захисту лабораторних робіт після встановленого терміну, одержані бали вираховуються з коефіцієнтом: для самостійної роботи студента -0,3; лабораторної роботи -0,7.

Звіт готується в паперовому варіанті й представляється під час захисту лабораторної роботи. В окремих випадках надсилається електронний варіант звіту за електронною адресою викладачу. Ім’я файлу звіту повинно містити прізвище, ім’я та групу студента, дисципліну та номер лабораторної роботи, що захищається.

Шкала оцінювання: національна та ECTS

Сума балів за всі види навчальної діяльності	Оцінка ECTS	Оцінка за національною шкалою
		для екзамену, курсової роботи
90 – 100	**A**	відмінно
82-89	**B**	добре
74-81	**C**	
64-73	**D**	задовільно
60-63	**E**	
35-59	**FX**	незадовільно з можливістю повторного складання
0-34	**F**	незадовільно з обов'язковим повторним вивченням дисципліни

Лабораторна робота №1

Тема: ОРГАНІЗАЦІЯ ТАБЛИЦЬ ІДЕНТИФІКАТОРІВ

Мета: Вивчити основні методи організації таблиць ідентифікаторів, одержати уявлення про переваги і недоліки, властиві різним методам організації таблиць ідентифікаторів.

Завдання:

Для виконання лабораторної роботи потрібно написати програму, яка одержує на вході набір ідентифікаторів, організовує таблиці ідентифікаторів за допомогою заданих методів, дозволяє здійснити багатократний пошук довільного ідентифікатора в таблицях і порівняти ефективність методів організації таблиць. Список ідентифікаторів вважати заданим у вигляді текстового файлу. Довжина ідентифікаторів обмежена 32 символами.

Теоретичні відомості:

Призначення таблиць ідентифікаторів

При виконанні семантичного аналізу, генерації коду і оптимізації результуючої програми компілятор повинен оперувати характеристиками основних елементів початкової програми — змінних, констант, функцій і інших лексичних одиниць вхідної мови. Ці характеристики можуть бути одержані компілятором на етапі синтаксичного аналізу вхідної програми (найчастіше при аналізі структури блоків описів змінних і констант), а також доповнені на етапі підготовки до генерації коду (наприклад при розподілі пам'яті).

Набір характеристик, відповідний кожному елементу початкової програми, залежить від типу цього елементу, від його сенсу (семантики) і, відповідно, від тієї ролі, яку він виконує в початковій і результуючій програмах. У кожному конкретному випадку цей набір характеристик може бути свій залежно від синтаксису і семантики вхідної мови, від архітектури цільової обчислювальної системи і від структури компілятора. Але є типові характеристики, які найчастіше властиві тим або іншим елементам початкової програми. Наприклад для змінної — це її тип і адреса елементу пам'яті, для константи — її значення, для функції - кількість і типи формальних аргументів, тип результату, що

повертається, адреса виклику коду функції. Докладнішу інформацію про характеристики елементів початкової програми, їх аналіз і використання можна знайти в [1,3, 7].

Головною характеристикою будь-якого елементу початкової програми є його ім'я. Саме з іменами змінних, констант, функцій і інших елементів вхідної мови оперує розробник програми - тому і компілятор повинен уміти аналізувати ці елементи по їх іменах.

Ім'я кожного елементу повинне бути унікальним. Багато сучасних мов програмування допускають збіги (неунікальність) імен змінних і функцій залежно від їх області видимості і інших умов початкової програми. В цьому випадку унікальність імен повинен забезпечувати сам компілятор - про те, як вирішується ця проблема, можна дізнатися в [1 - 3, 7], тут же вважатимемо, що імена елементів початкової програми завжди є унікальними.

Таким чином, завдання компілятора полягає в тому, щоб зберігати деяку інформацію, пов'язану з кожним елементом початкової програми, і мати доступ до цієї інформації по імені елементу. Для вирішення цього завдання компілятор організовує спеціальні сховища даних, звані *таблицями ідентифікаторів,* або *таблицями символів.* Таблиця ідентифікаторів складається з набору полів даних (записів), кожне з яких може відповідати одному елементу початкової програми. Запис містить всю необхідну компілятору інформацію про даний елемент і може поповнюватися у міру роботи компілятора. Кількість записів залежить від способу організації таблиці ідентифікаторів, але у будь-якому випадку їх не може бути менше, ніж елементів в початковій програмі. В принципі, компілятор може працювати не з однією, а з декількома таблицями ідентифікаторів – їх кількість і структура залежать від реалізації компілятора [1,2].

Принципи організації таблиць ідентифікаторів

Компілятор поповнює записи в таблиці ідентифікаторів по мірі аналізу початкової програми і виявлення в ній нових елементів, що вимагають розміщення в таблиці. Пошук інформації в таблиці виконується кожен раз, коли компілятору необхідні відомості про той або інший елемент програми. Причому

слід відмітити, що пошук елементу в таблиці виконуватиметься компілятором істотно частіше, ніж переміщення в неї нових елементів. Так відбувається тому, що описи нових елементів в початковій програмі, як правило, зустрічаються набагато рідше, ніж ці елементи використовуються. Крім того, кожному додаванню елементу в таблицю ідентифікаторів у будь-якому випадку передуватиме операція пошуку — щоб переконатися, що такого елементу в таблиці немає.

На кожну операцію пошуку елементу в таблиці компілятор витрачатиме час, і оскільки кількість елементів в початковій програмі велика (від одиниць до сотень тисяч залежно від об'єму програми), цей час істотно впливатиме на загальний час компіляції. Тому таблиці ідентифікаторів повинні бути організовані так, щоб компілятор мав можливість максимально швидко виконувати пошук потрібного йому запису в таблиці по імені елементу, з яким пов'язаний цей запис.

Можна виділити наступні способи організації таблиць ідентифікаторів:

- прості і впорядковані списки;
- бінарне дерево;
- хеш-адресація з рехешуванням;
- хеш-адресація по методу ланцюжків;
- комбінація хш-адресації із списком або бінарним деревом.

Далі буде дано короткий опис всіх вище перелічених способів організації таблиць ідентифікаторів. Докладнішу інформацію можна знайти в [3.7].

Прості методи побудови таблиць ідентифікаторів

У простому випадку таблиця ідентифікаторів є лінійним неврегульованим списком, або масивом, кожен осередок якого містить дані про відповідний елемент таблиці. Розміщення нових елементів в такій таблиці виконується шляхом запису інформації в чергову чарунку масиву або списку по мірі виявлення нових елементів в початковій програмі. Пошук потрібного елементу в таблиці в цьому випадку виконуватиметься шляхом послідовного перебору всіх елементів і порівняння їх імені з ім'ям шуканого елементу, поки не буде знайдений елемент з таким же ім'ям. Тоді якщо за одиницю часу прийняти час,

що витрачається компілятором на порівняння двох рядків (у сучасних обчислювальних системах таке порівняння найчастіше виконується однією командою), то для таблиці, що містить N елементів, в середньому буде виконано N/2 порівнянь.

Час, потрібний на додавання нового елементу в таблицю ($T_д$), не залежить від числа елементів в таблиці (N). Але якщо N велике, то пошук буде вимагати значних витрат часу. Час пошуку ($T_п$) в такій таблиці можна оцінити як $T_п$=O(N). Оскільки саме пошук в таблиці ідентифікаторів є найбільш часто виконуваною компілятором операцією, такий спосіб організації таблиць ідентифікаторов є неефективним. Він застосовується тільки для найпростіших компіляторів, що працюють з невеликими програмами.

Пошук може бути виконаний ефективніше, якщо елементи таблиці відсортовані (впорядковані) природним чином. Оскільки пошук здійснюється по імені, найбільш природним рішенням буде розташувати елементи таблиці в прямому або зворотному алфавітному порядку. Ефективним методом пошуку у впорядкованому списку з N елементів є *бінарний,* або *логарифмічний пошук.*

Алгоритм логарифмічного пошуку полягає в наступному: шуканий символ порівнюється з елементом (N+1)/2 в середині таблиці; якщо цей елемент не є шуканим, то ми повинні проглянути тільки блок елементів, пронумерованих від 1 до (N + 1) /2-1, або блок елементів від (N+1) /2+1 до N залежно від того, менше або більше шуканий елемент того, з яким його порівняли. Потім процес повторюється над потрібним блоком в два рази меншого розміру. Так продовжується до тих пір, поки або шуканий елемент не буде знайдений, або алгоритм не дійде до чергового блоку, що містить один або два елементи (з якими можна виконати пряме порівняння шуканого елементу). Оскільки на кожному кроці число елементів, які можуть містити шуканий елемент, скорочується в два рази, максимальне число порівнянь рівне 1 + 1og2N. Тоді час пошуку елементу в таблиці ідентифікаторів можна оцінити як Тп=O(1og2N). Для порівняння: при N=128 бінарний пошук вимагає найбільше 8 порівнянь, а пошук в неврегульованій таблиці — в середньому 64 порівняння. Метод називають «бінарним пошуком», оскільки на кожному кроці об'єм даної інформації

скорочується в два рази, а

«логарифмічним» — оскільки час, що витрачається на пошук потрібного елементу в масиві, має логарифмічну залежність від загальної кількості елементів в ньому. Недоліком логарифмічного пошуку є вимога впорядковування таблиці ідентифікаторів. Оскільки масив інформації, в якому виконується пошук, повинен бути впорядкований, час його заповнення вже залежатиме від числа елементів в масиві. Таблиця ідентифікаторів часто є видимим компілятором ще до того, як вона наповнена, тому потрібно, щоб умова впорядкованості виконувалася на всіх етапах звернення до неї. Отже, для побудови такої таблиці можна користуватися тільки алгоритмом прямого впорядкованого включення елементів.

Якщо користуватися стандартними алгоритмами, які використовуються для організації впорядкованих масивів даних, то середній час, необхідний на занесення всіх елементів в таблицю, можна оцінити таким чином:

$T_д = O(N \cdot \log_2 N) + k \cdot O(N^2)$.

Тут k - деякий коефіцієнт, що відображає співвідношення між часом, що витрачається комп'ютером на виконання операції порівняння і операції занесення даних.

При організації логарифмічного пошуку в таблиці ідентифікаторів забезпечується істотне скорочення часу пошуку потрібного елементу за рахунок збільшення часу на додавання нового елементу в таблицю. Оскільки додавання нових елементів в таблицю ідентифікаторів відбувається істотно рідше, ніж звернення до них, цей метод слід визнати ефективнішим, ніж метод організації неврегульованої таблиці. Проте в реальних компіляторах цей метод безпосередньо також не використовується, оскільки існують ефективніші методи.

Побудова таблиць ідентифікаторів по методу бінарного дерева

Можна скоротити час пошуку шуканого елементу в таблиці ідентифікаторів, не збільшуючи час, необхідний на її заповнення. Для цього треба відмовитися від організації таблиці у вигляді безперервного масиву даних.

Існує метод побудови таблиць, при якій таблиця має форму бінарного дерева. Кожен вузол дерева є улументом таблиці, причому корневим вузол стає перший елемент, який зустрінеться компілятором при заповненні таблиці. Дерево називається бінарним, оскільки кожна вершина в ньому може мати не більше двох гілок. Для визначеності називатимемо дві гаілки: «права» і «ліва».

Розглянемо алгоритм заповнення бінарного дерева. Вважатимемо, що алгоритм працює з потоком вхідних даних, що містить ідентифікатор. Перший ідентифікатор, як вже було сказано, поміщається у вершину дерева. Всі подальші ідентифікатори потрапляють в дерево по наступному алгоритму:

1. Вибрати черговий ідентифікатор з вхідного потоку даних. Якщо чергового ідентифікатора немає, то побудова дерева закінчена.
2. Зробити поточним вузлом дерева кореневу вершину.
3. Порівняти ім'я чергового ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.
4. Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, якщо рівне — припинити виконання алгоритму (двох однакових ідентифікаторів бути не повинно!), інакше — перейти до кроку 7.
5. Якщо у поточного вузла існує ліва вершина, то зробити її поточним вузлом і повернутися до кроку 3, інакше - перейти до кроку 6.
6. Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину лівою вершиною поточного вузла і повернутися до кроку 1.
7. Якщо у поточного вузла існує права вершина, то зробити її поточним вузлом і повернутися до кроку 3, інакше — перейти до кроку 8.
8. Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину правою вершиною поточного вузла і повернутися до кроку 1.

Розглянемо як приклад послідовність ідентифікаторів Ga, D1, M22, E, A12, BC, F. На Рис. 1.1 проілюстрований весь процес побудови бінарного дерева для цієї послідовності ідентифікаторів.

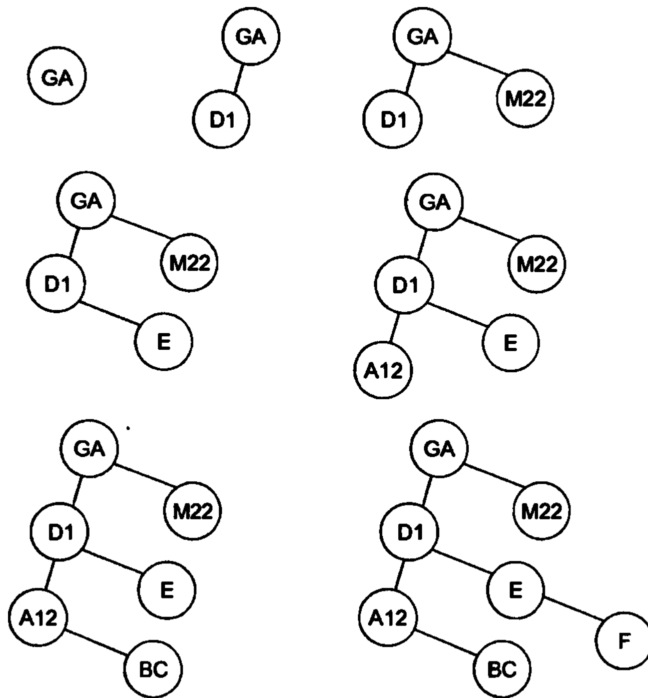

Рис. 1.1. Заповнення бінарного дерева для послідовності ідентифікаторів GA, D1, M22, E, A12, BC, F

Пошук елементу в дереві виконується по алгоритму, схожому з алгоритмом заповнення дерева:

1. Зробити поточним вузлом дерева кореневу вершину.

2. Порівняти ім'я шуканого ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.

3. Якщо імена співпадають, то шуканий ідентифікатор знайдений, алгоритм завершується, інакше треба перейти до кроку 4.

4. Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, інакше — перейти до кроку 6.

5. Якщо у поточного вузла існує ліва вершина, то зробити її поточним вузлом і повернутися до кроку 2, інакше - шуканий ідентифікатор не знайдений, алгоритм завершується.

6. Якщо у поточного вузла існує права вершина, то зробити її поточним вузлом і повернутися до кроку 2, інакше - шуканий ідентифікатор не знайдений, алгоритм завершується.

Для даного методу число необхідних порівнянь і форма дерева, що вийшло,

залежать від того порядку, в якому надходять ідентифікатори. Наприклад, якщо в розглянутому вище прикладі замість послідовності ідентифікаторів Ga, D1, M22, E, A12, BC, F узяти послідовність A12, BC, D1, E, F, Ga, M22, то дерево виродиться у впорядкований однонаправлений зв'язний список. Ця особливість є недоліком даного методу організації таблиць ідентифікаторів. Іншими недоліками методу є: необхідність зберігати два додаткові посилання на ліву і праву гілку в кожному елементі дерева і робота з динамічним виділенням пам'яті при побудові дерева.

Якщо припустити, що послідовність ідентифікаторів в початковій програмі є статистично неврегульованою (що в цілому відповідає дійсності), то можна вважати, що побудоване бінарне дерево буде невиродженим. Тоді середній час на заповнення дерева (Тд) і на пошук елементу в нім (Тп) можна оцінити таким чином [3, 7]:

$Тд=N•O(1og_2N)$; $Тп=O(1og_2N)$.

Не дивлячись на вказані недоліки, метод бінарного дерева є досить вдалим механізмом для організації таблиць ідентифікаторів. Він знайшов своє застосування у ряді компіляторів. Іноді компілятори будують декілька різних дерев для ідентифікаторів різних типів і різної довжини [1,2,3,7].

Хеш-функції і хеш-адресація

У реальних початкових програмах кількість ідентифікаторів така велика, що навіть логарифмічну залежність часу пошуку від їх числа не можна визнати задовільною. Необхідні ефективніші методи пошуку інформації в таблиці ідентифікаторів. Кращих результатів можна досягти, якщо застосувати методи, зв'язані з використанням хеш-функцій і хеш- адресації.

Хеш-функцією F називається деяке відображення множинаі вхідних елементів R на множина цілих ненегативних чисел Z: F(r)=n, r є R, N є Z. Сам термін «хеш-функція» походить від англійського терміну «hash function» (hash — «заважати», «змішувати», «плутати»).

Множина допустимих вхідних елементів R називається областю визначення хеш-функції. Множиначаю значень хеш-функції F називається підмножина M з

множинаі цілих ненегативних чисел Z: M є Z, що містить всі можливі значення, які повертаються функцією F*:* " rIR: F(r) є M і " m є M: $r є R: F(r)=m. Процес відображення області визначення хеш-функції на множина значень називається *хешуванням.*

При роботі з таблицею ідентифікаторів хеш-функція повинна виконувати відображення імен ідентифікаторів на множина цілих ненегативних чисел. Областю визначення хеш-функції буде множина всіх можливих імен ідентифікаторів.

Хеш-адресація полягає у використанні значення, яке повертається хеш-функцією, як адреса осередку з деякого масиву даних. Тоді розмір масиву даних повинен відповідати області значень використовуваної хеш- функції.

Отже, в реальному компіляторі область значень хеш-функції ніяк не повинна перевищувати розмір доступного адресного простору комп'ютера.

Метод організації таблиць ідентифікаторів, заснований на використанні хеш-адресації, полягає в додаванні кожного елементу таблиці в чарунку, адресу якої повертає хеш-функція, обчислена для цього елементу. Тоді в ідеальному випадку для додавання будь-якого елементу в таблицю ідентифікаторів досить тільки обчислити його хеш-функцію і звернутися до потрібної чарунки масиву даних. Для пошуку елементу в таблиці також необхідно обчислити хеш-функцію для шуканого елементу і перевірити, чи не є задана нею чарунка масиву порожньою (якщо вона не порожня — елемент знайдений, якщо порожня — не знайдений). Спочатку таблиця ідентифікаторів повинна бути заповнена інформацією, яка дозволила б говорити про те, що всі її чарунки є порожніми. Цей метод вельми ефективний, оскільки як час розміщення елементу в таблиці, так і час його пошуку визначаються тільки часом, що витрачається на обчислення хеш- функції, яке в загальному випадку незіставно менше часу, необхідного для багатократних порівнянь елементів таблиці. Метод має два очевидні недоліки. Перший з них — неефективне використання об'єму пам'яті під таблицю ідентифікаторів: розмір масиву для її зберігання повинен відповідати всій області значень хеш-функції, тоді як ідентифікаторів, що реально зберігаються в таблиці, може бути істотно менше. Другий недолік —

необхідність відповідного розумного вибору хеш-функції. Цей недолік є настільки істотним, що не дозволяє безпосередньо використовувати хеш-адресацію для організації таблиць ідентифікаторів.

Проблема вибору хеш-функції не має універсального рішення. Хешування зазвичай відбувається за рахунок виконання над ланцюжком символів деяких простих арифметичних і логічних операцій. Найпростішою хеш-функцією для символу є код внутрішнього уявлення в комп'ютері літери символу. Цю хеш-функцію можна використовувати і для ланцюжка символів, вибираючи перший символ в ланцюжку.

Очевидно, що така примітивна хеш-функція буде незадовільною: при її використанні виникне проблема - двом різним ідентифікаторам, що починаються з однієї і тієї ж букви, відповідатиме одне і те ж значення хеш- функції. Тоді при хеш-адресації в одну і ту ж чарунку таблиці ідентифікаторів повинні бути поміщені два різні ідентифікатори, що явно неможливо. Така ситуація, коли двом або більше ідентифікаторам відповідає одне і те ж значення хеш-функції, називається *колізією.*

Природно, що хеш-функція, що допускає колізії, не може бути використана для хеш-адресації в таблиці ідентифікаторів. Причому досить одержати хоча б один випадок колізії на всій множині ідентифікаторів, щоб такою хеш-функцією не можна було користуватися. Але чи можливо побудувати хеш-функцію, яка б повністю виключала виникнення колізій?

Для повного виключення колізій хеш-функція повинна бути взаємно однозначною: кожному елементу з області визначення хеш-функції повинне відповідати одне значення з її множини значень, і навпаки — кожному значенню з множини значень цієї функції повинен відповідати тільки один елемент з її області визначення. Тоді будь-яким двом довільним елементам з області визначення хеш-функції завжди відповідатимуть два різних її значення. Теоретично для ідентифікаторів таку хеш-функцію побудувати можна, оскільки і область визначення хеш-функції (всі можливі імена ідентифікаторів), і область цих значень (цілі ненегативні числа) є нескінченними рахунковими множинами, тому можна організувати взаємнооднозначне відображення однієї множини на

іншу.

Але на практиці існує обмеження, що робить створення взаємнооднозначної хеш-функції для ідентифікаторів неможливим. Річ у тому, що в реальності область значень будь-якої хеш-функції обмежена розміром доступного адресного простору комп'ютера. Множина адрес будь- якого комп'ютера з традиційною архітектурою може бути велика, але завжди скінченна. Організувати взаємно однозначне відображення нескінченної множини на кінцеве навіть теоретично неможливо. Можна, звичайно, врахувати, що довжина частини імені ідентифікатора, що приймається до уваги, в реальних компіляторах на практиці також обмежена — зазвичай вона лежить в межах від 32 до 128 символів (тобто і область визначення хеш- функції кінцева). Але і тоді кількість елементів в кінцевій множині, що становить область визначення хеш-функції, перевищуватиме їх кількість в кінцевій множині області її значень (кількість всіх можливих ідентифікаторів більше кількості допустимих адрес в сучасних комп'ютерах). Таким чином, створити взаємно однозначну хеш-функцію на практиці неможливо. Отже, неможливо уникнути виникнення колізій.

Тому не можна організувати таблицю ідентифікаторів безпосередньо на основі однієї тільки хеш-адресації. Але існують методи, що дозволяють використовувати хеш-функції для організації таблиць ідентифікаторів навіть за наявності колізій.

Хеш-адресація з рехешуванням

Для вирішення проблеми колізії можна використовувати багато способів. Одним з них є метод *рехешування* (або розстановки). Згідно цього методу, якщо для елементу A адреса n0=h(A), обчислена за допомогою хеш- функції h, вказує на вже зайняту чарунку, то необхідно обчислити значення функції n1=h1(A) і перевірити зайнятість чарунки за адресою n1. Якщо і вона зайнята, то обчислюється значення h2(A), і так до тих пір, поки або не буде знайдена вільна чарунка, або чергове значення hi(A) не співпаде з h(A). У останньому випадку вважається, що таблиця ідентифікаторів заповнена і місця в ній більше немає —

видається інформація про помилку розміщення ідентифікатора в таблиці. Тоді пошук елементу А в таблиці ідентифікаторів, організованій таким чином, виконуватиметься але наступному алгоритму:

1. Обчислити значення хеш-функції n=h(A) для шуканого елементу *A*.
2. Якщо чарунка за адресою n порожня, то елемент не знайдений, алгоритм завершений, інакше необхідно порівняти ім'я елементу в чарунці n з ім'ям шуканого елементу *A*. Якщо вони співпадають, то елемент знайдений і алгоритм завершений, інакше i:=1 і перейти до кроку 3.
3. Обчислити ni = hi(A). Якщо чарунка за адресою ni порожня або n=ni, то елемент не знайдений і алгоритм завершений, інакше - порівняти ім'я елементу в чарунці ni з ім'ям шуканого елементу *A*. Якщо вони співпадають, то елемент знайдений і алгоритм завершений, інакше i:=i + 1 і повторити крок 3.

Алгоритми розміщення і пошуку елементу схожі по виконуваних операціях. Тому вони матимуть однакові оцінки часу, необхідного для їх виконання.

При такій організації таблиць ідентифікаторів у разі виникнення колізії алгоритм поміщає елементи в порожні елементи таблиці, вибираючи їх певним чином. При цьому елементи можуть потрапляти в чарунки з адресами, які потім співпадатимуть із значеннями хеш-функції, що приведе до виникнення нових, додаткових колізій. Таким чином, кількість операцій, необхідних для пошуку або розміщення в таблиці елементу, залежить від заповненої таблиці.

Для організації таблиці ідентифікаторів по методу рехешування необхідно визначити всі хеш-функції hi для всіх i. Найчастіше функції hi визначають як деякі модифікації хеш-функцій h. Наприклад, найпростішим методом обчислення функції hi(A) є її організація у вигляді hi(A)=(h(A)+pi)modNm, де pi — деяке обчислюване ціле число, а Nm — максимальне значення з області значень хеш-функції h. У свою чергу, найпростішим підходом тут буде покласти pi=i. Тоді одержуємо формулу hi(A)=(h(A)+i)modNm . В цьому випадку при збігу значень хеш-функції для яких-небудь елементів пошук вільної чарунки в таблиці починається послідовно від поточної позиції, заданою хеш-функцією h(A).

Цей спосіб не можна визнати особливо вдалим: при збігу хеш-адрес

елементи в таблиці починають групуватися навколо них, що збільшує число необхідних порівнянь при пошуку і розміщенні. Але навіть такий примітивний метод рехешування є достатньо ефективним засобом організації таблиць ідентифікаторів при неповному заповненні таблиці. Середній час на додавання одного елементу в таблицю і на пошук елементу в таблиці можна понизити, якщо застосувати більш довершений метод рехешування. Одним з таких методів є використання як pi для функції hi(A)=(h(A)+pi)modNm послідовності псевдовипадкових цілих чисел p1, p2…p_k. При хорошому виборі генератора псевдовипадкових чисел довжина послідовності k=Nm .

Існують і інші методи організації функцій рехешування hi(A), засновані на квадратичних обчисленнях або, наприклад, на обчисленні піднесення за формулою: hi(A)=(h(A)N•i)mod N'm де N'm —найближче просте число, менше Nm. В цілому рехешування дозволяє добитися непоганих результатів для ефективного пошуку елементу в таблиці (кращих, ніж бінарний пошук і бінарне дерево), Але ефективність методу сильно залежить від заповненої таблиці ідентифікаторів і якості використовуваної хеш-функції — чим рідше виникають колізії, тим вище ефективність методу. Вимога неповного заповнення таблиці веде до неефективного використання об'єму доступної пам'яті.

Оцінки часу розміщення і пошуку елементу в таблицях ідентифікаторів при використанні різних методів рехешування можна знайти в [1,3,7].

Хеш-адресація з використанням методу ланцюжків

Неповне заповнення таблиці ідентифікаторів при застосуванні рехешування веде до неефективного використання всього об'єму пам'яті, доступного компілятору. Причому об'єм невживаної пам'яті буде тим вище, чим більше інформації зберігається для кожного ідентифікатора. Цього недоліку можна уникнути, якщо доповнити таблицю ідентифікаторів деякою проміжною хеш-таблицею.

В чарунках хеш-таблиці може зберігатися або порожнє значення, або значення вказівника на деяку область пам'яті з основної таблиці ідентифікаторів. Тоді хеш-функція обчислює адресу, по якій відбувається звернення спочатку до

хеш-таблиці, а потім вже через неї за знайденою адресою — до самої таблиці ідентифікаторів. Якщо відповідний елемент таблиці ідентифікаторів порожній, то чарунка хеш-таблиці міститиме порожнє значення. Тоді зовсім не обов'язково мати в самій таблиці ідентифікаторів чарунку для кожного можливого значення хеш-функції - таблицю можна зробити динамічною, так щоб її об'єм ріс по мірі заповнення (спочатку таблиця ідентифікаторів не містить жодної чарунки, а всі чарунки хеш-таблиці мають порожнє значення).

Такий підхід дозволяє добитися двох позитивних результатів: по- перше, немає необхідності заповнювати порожніми значеннями таблицю ідентифікаторів - це можна зробити тільки для хеш-таблиці; по-друге, кожному ідентифікатору відповідатиме строго одна чарунка в таблиці ідентифікаторів. Порожні чарунки у такому разі будуть тільки в хеш-таблиці, і об'єм невживаної пам'яті не залежатиме від об'єму інформації, що зберігається для кожного ідентифікатора, — для кожного значення хеш- функції витрачатиметься тільки пам'ять, необхідна для зберігання одного вказівника на основну таблицю ідентифікаторів.

На основі цієї схеми можна реалізувати ще один спосіб організації таблиць ідентифікаторів за допомогою хеш-функції, який називається *методом ланцюжків.* В цьому випадку в таблицю ідентифікаторів для кожного елементу додається ще одне поле, в якому може міститися посилання на будь-який елемент таблиці. Спочатку це поле завжди порожнє (нікуди не вказує). Також необхідно мати одну спеціальну змінну, яка завжди вказує на перший вільний елемент основної таблиці ідентифікаторів (спочатку вона вказує на початок таблиці). Метод ланцюжків працює по наступному алгоритму:

1. У всі чарунки хеш-таблиці помістити порожнє значення, таблиця ідентифікаторів порожня, змінна FreePtr (вказівник першої вільної чарунки) вказує на початок таблиці ідентифікаторів.

2. Обчислити значення хеш-функції n для нового елементу A. Якщо чарунка хеш-таблиці за адресою n порожня, то помістити в неї значення змінної *FreePtr* і перейти до кроку 5; інакше перейти до кроку 3.

3. Вибрати з хеш-таблиці адресу елементу таблиці ідентифікаторів m і

перейти до кроку 4.

4. Для елементу таблиці ідентифікаторів за адресою m перевірити значення поля посилання. Якщо воно порожнє, то записати в нього адресу змінної *FreePtr* і перейти до кроку 5; інакше вибрати з поля посилання нову адресу m і повторити крок 4.

5. Додати в таблицю ідентифікаторів нову чарунку, записати в неї інформацію для елементу *A* (поле посилання повинне бути порожнім), в змінну *FreePtr* помістити адресу за кінцем доданої чарунки. Якщо більше немає ідентифікаторів, які треба помістити в таблицю, то виконання алгоритму закінчене, інакше перейти до кроку 2.

Пошук елементу в таблиці ідентифікаторів, організованій таким чином, виконуватиметься по наступному алгоритму:

1. Обчислити значення хеш-функції n для шуканого елементу *A*. Якщо чарунка хеш-таблиці за адресою n порожня, то елемент не знайдений і алгоритм завершений, інакше вибрати з хеш-таблиці адресу елементу таблиці ідентифікаторів m.

2. Порівняти ім'я елементу в елементі таблиці ідентифікаторів за адресою m з ім'ям шуканого елементу A. Якщо вони співпадають, то шуканий елемент знайдений і алгоритм завершений, інакше перейти до кроку 3.

3. Перевірити значення поля посилання в чарунці таблиці ідентифікаторів за адресою m. Якщо воно порожнє, то шуканий елемент не знайдений і алгоритм завершений; інакше вибрати з поля посилання адресу m і перейти до кроку 2.

При такій організації таблиць ідентифікаторів у разі виникнення колізії алгоритм поміщає елементи в елементи таблиці, пов'язуючи їх один з одним послідовно через поле посилання. При цьому елементи не можуть потрапляти в чарунки з адресами, які потім співпадатимуть із значеннями хеш-функції. Таким чином, додаткові колізії не виникають. У результаті в таблиці виникають своєрідні ланцюжки зв'язаних елементів, звідки і походить назва даного методу – «метод ланцюжків».

На Рис. 1.2 проілюстровано заповнення хеш-таблиці і таблиці

ідентифікаторів для ряду ідентифікаторів: A_1, A_2, A_3, A_4, A_5 за умови що $h(A_1)=h(A_2)=h(A_5)=n_1$; $h(A_3)=n_2$; $h(A_4)=n_4$. Після розміщення в таблиці для пошуку ідентифікатора A_1 буде потрібно одне порівняння, для A_2 — два порівняння, для A_3 —одне порівняння, для A_4 - одне порівняння і для A_5 — три порівняння (спробуйте порівняти ці дані з результатами, одержаними з використанням простого рехешування для тих же ідентифікаторів).

Метод ланцюжків є дуже ефективним засобом організації таблиць ідентифікаторів. Середній час на розміщення одного елементу і на пошук елементу в таблиці для нього залежить тільки від середнього числа колізій, що виникають при обчисленні хеш-функції. Накладні витрати пам'яті, пов'язані з необхідністю мати одне додаткове поле вказівника в таблиці ідентифікаторів на кожен її елемент, можна визнати цілком виправданими, оскільки виникає економія використовуваної пам'яті за рахунок проміжної хеш-таблиці. Цей метод дозволяє економніше використовувати пам'ять, але вимагає організації роботи з динамічними масивами даних.

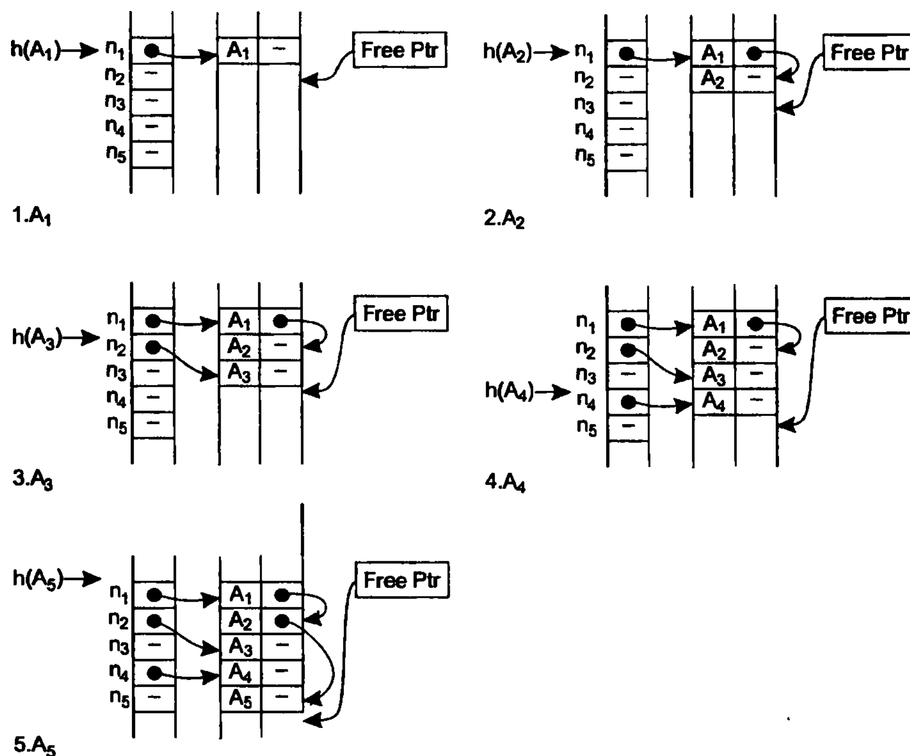

Рис. 1.2. Заповнення таблиці ідентифікаторів при використанні методу ланцюжків

Комбіновані способи побудови таблиць ідентифікаторів

Окрім рехешування і методу ланцюжків можна використовувати комбіновані методи для організації таблиць ідентифікаторів за допомогою хеш-адресації. В цьому випадку для виключення колізій хеш-адресація поєднується з одним з раніше розглянутих методів — простим списком, впорядкованим списком або бінарним деревом, який використовується як додатковий метод впорядковування ідентифікаторів, для яких виникають колізії. Причому, оскільки при якісному виборі хеш-функції кількість колізій зазвичай невелика (одиниці або десяти випадків), навіть простій список може бути цілком задовільним рішенням при використанні комбінованого методу.

При такому підході можливі два варіанти: у першому випадку, як і для методу ланцюжків, в таблиці ідентифікаторів організовується спеціальне додаткове поле посилання. Але на відміну від методу ланцюжків воно має декілька інше значення: за відсутності колізій для вибірки інформації з таблиці використовується хеш-функція, поле посилання залишається порожнім. Якщо ж виникає колізія, то через поле посилання організовується пошук ідентифікаторів, для яких значення хеш-функції співпадають, - це поле повинне указувати на структуру даних для додаткового методу: початок списку, перший елемент динамічного масиву або кореневий елемент дерева.

У другому випадку використовується хеш-таблиця, аналогічна хеш- таблиці для методу ланцюжків. Якщо за даною адресою хеш-функції ідентифікатор відсутній, то осередок хеш-таблиці порожній. Коли з'являється ідентифікатор з даним значенням хеш-функції, то створюється відповідна структура для додаткового методу, в хеш-таблицю записується посилання на цю структуру, а ідентифікатор поміщається в створену структуру за правилами вибраного додаткового методу.

У першому варіанті за відсутності колізії пошук виконується швидше, але другий варіант переважно, оскільки за рахунок використання проміжної хеш-таблиці забезпечується ефективніше використання пам'яті.

Як і для методу ланцюжків, для комбінованих методів час розміщення і час пошуку елементу в таблиці ідентифікаторів залежить тільки від середнього числа колізій, що виникають при обчисленні хеш-функції. Накладні витрати

пам'яті при використанні проміжної хеш-таблиці мінімальні. Очевидно, що якщо як додатковий метод використовувати простій список, то вийде алгоритм, повністю аналогічний методу ланцюжків. Якщо ж використовувати впорядкований список або бінарне дерево, то метод ланцюжків і комбіновані методи матимуть приблизно рівну ефективність при незначному числі колізій (одиничні випадки), але із зростанням кількості колізій ефективність комбінованих методів в порівнянні з методом ланцюжків зростатиме.

Недоліком комбінованих методів є складніша організація алгоритмів пошуку і розміщення ідентифікаторів, необхідність роботи з динамічно розподілюваними областями пам'яті, а також великі витрати часу на розміщення нового елементу в таблиці ідентифікаторів в порівнянні з методом ланцюжків.

То, який конкретно метод застосовується в компіляторі для організації таблиць ідентифікаторів, залежить від реалізації компілятора. Один і той же компілятор може мати навіть декілька різних таблиць ідентифікаторів, організованих на основі різних методів. Як правило, застосовуються комбіновані методи. Створення ефективної хеш-функції - це окреме завдання розробників компіляторів, і отримані результати, як правило, тримаються у секреті. Хороша хеш-функція розподіляє ідентифікатори, що на її вхід, рівномірно на адреси, щоб звести до мінімуму кількість колізій. В даний час існує множина хеш-функцій, але, як було показано вище, ідеального хешування досягти неможливо.

Хеш-адресація — це метод, який застосовується не тільки для організації таблиць ідентифікаторів в компіляторах. Даний метод знайшов своє застосування і в операційних системах, і в системах управління базами даних [5,6,11].

Вимоги до виконання роботи

Порядок виконання роботи

1. Одержати варіант завдання у викладача.
2. Вибрати і описати хеш-функцію.
3. Описати структури даних, використовувані для заданих методів

організації таблиць ідентифікаторів.

4. Підготувати і захистити звіт.
5. Написати і відлагодити програму на ПК.
6. Здати працюючу програму викладачеві.

Варіанти завдань

У табл. 1.1 дані варіанти завдань на основі методів організації таблиць ідентифікаторів, перерахованих в табл. 1.2.

Таблиця 1-1- Варіантів завдань

№ варіанту	I-й метод	II-й метод організації таблиць
1	1	5
2	1	6
3	1	7
4	2	1
5	2	5
6	2	6
7	3	5
8	3	6
9	3	7
10	7	5
11	4	6
12	4	7
13	1	4
14	2	4
15	3	4
16	2	3

У табл. 1.2 перераховані методи організації таблиць ідентифікаторів, що використовуються в завданнях.

Таблиця 1.2. Методи організації таблиць ідентифікаторів

№ методу	Спосіб вирішення колізій
1	Просте рехешування
2	Рехешування з використанням псевдовипадкових чисел
3	Рехешування за допомогою твору
4	Метод ланцюжків
5	Простий список
6	Впорядкований список
7	Бінарне дерево

Вимоги до оформлення звіту:

Звіт по лабораторній роботі повинен містити наступні розділи:

1) завдання по лабораторній роботі;

2) опис вибраної хеш-функції;

3) схеми організації таблиць ідентифікаторів (відповідно до варіанту завдання);
4) опис алгоритмів пошуку в таблицях ідентифікаторів (відповідно до варіанту завдання);
5) текст програми (оформляється після виконання програми на ПК**);**
6) результати обробки заданого набору ідентифікаторів (вхідного файлу) за допомогою методів організації таблиць ідентифікаторів, вказаних у варіанті завдання;
7) аналіз ефективності використовуваних методів організації таблиць ідентифікаторів і висновки по виконаній роботі.

Контрольні питання

1) Що таке таблиця символів і для чого вона призначена? Яка інформація може зберігатися в таблиці символів?

2) Які цілі переслідуються при організації таблиці символів?

3) Якими характеристиками можуть володіти лексичні елементи початкової програми? Які характеристики є обов'язковими?

4) Які існують способи організації таблиць символів?

5) В чому полягає алгоритм логарифмічного пошуку? Які переваги він дає в порівнянні з простим перебором і які він має недоліки?

6) Розкажіть про деревовидну організацію таблиць ідентифікаторів. У чому її переваги і недоліки?

7) Що таке хеш-функції і для чого вони використовуються? У чому суть хеш-адресації?

8) Що таке колізія? Чому вона відбувається? Чи можна повністю уникнути колізії?

9) Що таке рехешування? Які методи рехешування існують?

10) Розкажіть про переваги і недоліки організації таблиць ідентифікаторів за допомогою хеш-адресації і рехешування.

11) В чому полягає метод ланцюжків?

12) Розкажіть про переваги і недоліки організації ідентифікаторів за допомогою хеш-адресації і методу ланцюжків.

13) Як можуть бути скомбіновані різні методи організації хеш- таблиць?

14) Розкажіть про переваги і недоліки організації таблиць ідентифікаторів за допомогою комбінованих методів.

Лабораторна робота №2

Тема: ПРОЕКТУВАННЯ ЛЕКСИЧНОГО АНАЛІЗУ

Мета: Вивчити основні поняття теорії регулярних граматик, ознайомлення з призначенням і принципами роботи лексичних аналізаторів (сканерів), отримання практичних навиків побудови сканера на прикладі заданої простої вхідної мови.

Завдання:

Написати програму, яка виконує лексичний аналіз вхідного тексту відповідно до завдання і породжує таблицю лексем з вказівкою їх типів і значень. Текст на вхідній мові задається у вигляді символьного (текстового) файлу. Програма повинна видавати повідомлення про наявність у вхідному тексті помилок, які можуть бути виявлені на етапі лексичного аналізу.

Довжину ідентифікаторів і строкових констант можна вважати обмеженою 32 символами. Програма повинна допускати наявність коментарів необмеженої довжини у вхідному файлі. Форму організації коментарів пропонується вибрати самостійно.

Теоретичні відомості:

Призначення лексичного аналізатора

Лексичний аналізатор (або сканер) - це частина компілятора, яка читає літери програми на початковій мові і будує з них слова (лексеми) початкової мови. На вхід лексичного аналізатора поступає текст початкової програми, а вихідна інформація передається для подальшої обробки компілятором на етапі синтаксичного аналізу і розбору.

Лексема (лексична одиниця мови) - це структурна одиниця мови, яка складається з елементарних символів мови і не містить в своєму складі інших структурних одиниць мови. Лексемами мов програмування є ідентифікатори, константи, ключові слова мови, знаки операцій 11 т.п. Склад можливих лексем кожної конкретної мови програмування визначається синтаксисом цієї мови.

З теоретичної точки зору лексичний аналізатор не є обов'язковою,

необхідною частиною компілятора. Його функції можуть виконуватися На етапі синтаксичного аналізу. Проте існує декілька причин, виходячи з яких до складу практично всіх компіляторів включають лексичний аналіз.

Це наступні причини:

- спрощується робота з текстом початкової програми на етапі синтаксичного розбору і скорочується об'єм оброблюваної інформації, оскільки лексичний аналізатор структурує початковий текст програми, що поступає на вхід, і видаляє всю незначущу інформацію;
- для виділення в тексті і розбору лексем можливо застосовувати просту, ефективну і техніку аналізу, що добре пропрацювала теоретично, тоді як на етапі синтаксичного аналізу конструкцій початкової мови використовуються достатньо складні алгоритми розбору;
- лексичний аналізатор відокремлює складний по конструкції синтаксичний аналізатор від роботи безпосередньо з текстом початкової програми, структура якого може варіюватися залежно від версії вхідної мови – при такій конструкції компілятора при переході від однієї версії мови до іншої достатньо тільки перебудувати відносно простий лексичний аналізатор.

Функції, що виконуються лексичним аналізатором, і склад лексем, які він виділяє в тексті початкової програми, можуть мінятися залежно від версії компілятора. В основному лексичні аналізатори виконують вилученн з тексту початкової програми коментарів і незначущих пропусків, а також виділення лексем наступних типів: ідентифікаторів, строкових, символьних і числових констант, знаків операцій, роздільників і ключових (службових) слів вхідної мови

В більшості компіляторів лексичний і синтаксичний аналізатори – це взаємозв'язані частини. Де провести межу між лексичним і синтаксичним аналізом, які конструкції аналізувати сканером, а які – синтаксичним засобом розпізнавання, вирішує розробник компілятора. Як правило, будь-який аналіз прагнуть виконати на етапі лексичного розбору вхідної програми, якщо він може бути там виконаний. Можливості лексичного аналізатора обмежені в порівнянні з синтаксичним аналізатором, оскільки в його основі лежать простіші механізми. Детальніше про роль лексичного аналізатора в компіляторі і про його взаємодію

з синтаксичним аналізатором можна дізнатися в [1-4, 7].

Проблема визначення меж лексем

В найпростішому випадку фази лексичного і синтаксичного аналізу можуть виконуватися компілятором послідовно. Але для багатьох мов програмування інформації на етапі лексичного аналізу може бути недостатньо для однозначного визначення типу і меж чергової лексеми.

Ілюстрацією такого випадку може служити приклад оператора програми на мові Фортран, коли по частині тексту D0 10 I=1. неможливо визначити тип оператора (а відповідно, і межі лексем). У разі DO 10 I=1.15 це буде привласнення речовинної змінної D010I значення константи 1.15 (пробыли у Фортрані ігноруються). а у разі DO 10 I=1.15 це цикл з переліком від 1 до 15 по цілочисельній змінній I до мітки 10.

Інша ілюстрація з сучаснішої мови програмування С++ - оператор привласнення k=i+++++j:, який має тільки одну вірну інтерпретацію (якщо операції розділити пропусками): k=i+++++j:.

Якщо неможливо визначити межі лексем, то лексичний аналіз початкового тексту повинен виконуватися поетапно. Тоді лексичний і синтаксичний аналізатори повинні функціонувати паралельно, по черзі звертаючись один до одного. Лексичний аналізатор. знайшовши чергову лексему, передає її синтаксичному аналізатору, той намагається виконати аналіз ліченої частини початкової програми і може або запитати у лексичного аналізатора наступну лексему, або зажадати від нього повернутися на декілька кроків назад і спробувати виділити лексеми з іншими межами. При цьому він може повідомити інформацію про те, яку лексему слід чекати. Детальніше така схема взаємодії лексичного і синтаксичного аналізаторів описана [3, 7].

Паралельна робота лексичного і синтаксичного аналізаторів, очевидно, складніша в реалізації, чим їх послідовне виконання. Крім того, такий підхід вимагає більше обчислювальних ресурсів і в загальному випадку більшого часу на аналіз початкової програми, оскільки допускає повернення назад і повторний аналіз вже прочитаної частини початкового коду. Проте складність синтаксису деяких мов програмування вимагає саме такого підходу - розглянутий раніше приклад програми на мові Фортран не може бути проаналізований інакше.

Щоб уникнути паралельної роботи лексичного і синтаксичного аналізаторів, розробники компіляторів деяких мов програмування часто йдуть на розумні обмеження синтаксису вхідної мови. Наприклад. для мови С++ прийнято угоду, що при виникненні проблем з визначенням меж лексеми завжди вибирається лексема максимально можливої довжини.

В розглянутому вище прикладі для оператора k=i+++++j: це приведе до того, що при читанні четвертого знаку + з двох варіантів Лексем (+ - знак складання в С++, а ++ - оператор інкременту) лексичний аналізатор вибере щонайдовшу - ++ (оператор інкременту) - і в цілому весь оператор буде розібраний як k=i++ ++ +j: (знаки операцій розділені пропусками), що невірно, оскільки семантика мови С++ забороняє двох операторів інкременту підряд. Звичайно, невірний аналіз операторів, аналогічних приведеному в прикладі (охочі можуть переконатися в цьому на будь-якому доступному компіляторі мови С++), - незначна платня за збільшення ефективності роботи компілятора і не обмежує можливості мови (той же самий оператор може бути записаний у вигляді k=i++ + ++j:, що виключить будь-які неоднозначності в його аналізі). Проте таким же шляхом для мови Фортран піти не можна - різниця між оператором привласнення і оператором циклу дуже велика, щоб нею можна було нехтувати.

Надалі виходитимемо з припущення, що всі лексеми можуть бути однозначно виділені сканером на етапі лексичного аналізу. Для всіх сучасних мов програмування це дійсно так, оскільки їх синтаксис розроблявся з урахуванням можливостей компіляторів.

Таблиця лексем і інформація, що міститься в ній

Результатом роботи лексичного аналізатора є перелік всіх знайдених в тексті початкової програми лексем з урахуванням характеристик кожної лексеми. Цей перелік лексем можна представити у вигляді таблиці, званою таблицею лексем. Кожній лексемі в таблиці лексем відповідає якийсь унікальний умовний код, залежний типу лексеми, і додаткова службова інформація. Таблиця лексем в кожному рядку повинна містити інформацію про вид лексеми, її типі і, можливо, значенні. Звичайно структури даних, службовці для організації такої таблиці, мають два поля: перше - тип лексеми, друге - покажчик на інформацію

про лексему.

Крім того, інформація про деяких типів лексем, найдених в початковій програмі, повинна поміщатися таблицю ідентифікаторів (або в одну з таблиць ідентифікаторів, якщо компілятор передбачає різні таблиці ідентифікаторів для різних типів лексем).

Увага Не слід плутати таблицю лексем і таблицю ідентифікаторів - це дві принципово різні таблиці, що обробляються лексичним аналізатором.

Таблиця лексем фактично містить весь текст початкової програми, оброблений лексичним аналізатором. У неї входять всі можливі типи лексем, крім того, будь-яка лексема може зустрічатися в ній будь-яка кількість раз. таблиця ідентифікаторів містить тільки певні твані лексем - ідентифікатори і константи. У неї не потрапляють такі лексеми, як ключові (службові) слова вхідної мови, знаки операцій і роздільники. Крім того, кожна лексема (ідентифікатор або константа) може зустрічатися в таблиці ідентифікаторів тільки один раз. Також можна відзначити, що лексеми в таблиці лексем обов'язково розташовуються в тому ж порядку, що і в початковій програмі (порядок лексем в ній не міняється), а в таблиці ідентифікаторів лексеми розташовуються у будь-якому порядку так, щоб забезпечити зручність пошуку.

Як приклад можна розглянути деякий фрагмент початкового коду на мові Object Pascal і відповідну йому таблицю лексем, представлену табл. 2.1:

```
.
Begin
for i=1 to N do
fg=fg*0.5
..
```

Поле «значення» в табл. 2.1 має на увазі якесь кодове значення, яке буде поміщене в підсумкову таблицю лексем в результаті роботи лексичного аналізатора. Звичайно, значення, які записані в прикладі, є умовними. Конкретні коди вибираються розробниками при реалізації компілятора. Важливо відзначити також, що встановлюється зв'язок таблиці лексем з таблицею ідентифікаторів (у прикладі це відбито деяким індексом, наступним після ідентифікатора за знаком «:», а в реальному компіляторі визначається його реалізацією).

Таблиця 2.1. Лексеми фрагменту програми на мові Pascal

Лексема	Тип лексемы	Значение
begin	Ключове слово	X_1
for	Ключове слово	X_2
i	Ідентифікатор	i:1
:=	Знак присвоєння	S_1
1	Цілочисельна константа	1
to	Ключове слово	X_3
N	Ідентифікатор	N:2
do	Ключове слово	X_4
fg	Ідентифікатор	fg:3
:=	Знак присвоєння	S,
fg	Ідентифікатор	Fg:3
*	Знак арифметичної операції	A,
0,5	Речова константа	0,5

Побудова лексичних аналізаторів (сканерів)

Лексичній аналізатор має справу з такими об'єктами, як різного роду константи і ідентифікатори (до останніх відносяться і ключові слова). Мова опису констант і ідентифікаторів в більшості випадків є регулярною, тобто може бути описаний за допомогою регулярних граматик [1-4,7]. Пристроями розпізнавання для регулярних мов є кінцеві автомати (КА). Існують правила, за допомогою яких для будь-якої регулярної граматики може бути побудований КА. що розпізнає ланцюжки мови, заданої цією граматикою.

Детальніше про побудову КА на основі граматик для регулярних мов можна дізнатися в [3,7,16].

Будь-який КА може бути заданий за допомогою п'яти параметрів: $M(Q, \Sigma, \delta, q_0, F)$

де:

Q - кінцева множина станів автомата;

Σ - кінцева множина допустимих вхідних символів (вхідний алфавіт КА);

δ - задане відображення множини $Q \bullet \Sigma$ у Множина підмножин P(Q) д:

$Q \bullet \Sigma \rightarrow P(Q)$ (іноді д називають функцією переходів автомата);

q_o є Q - початковий стан автомата;

F є Q - множина завершальних СТАНІВ автомата.

Іншим способом описи КА є граф переходів - графічне представлення множинаі станів і функції переходів КА. Граф переходів КА - це навантажений однонаправлений граф, в якому вершини представляють стани КА, дуги відображають переходи з одного стану В інший. а символи навантаження (позначки) дуг відповідають функції переходу КА. Якщо функція переходу КА передбачає перехід із стану q в q' по декількох символах, то між ними будується одна яка позначається всіма символами, по яких відбувається перехід з q в q'.

Недетермінований КА незручний для аналізу ланцюжків, оскільки в ньому можуть зустрічатися стани, що допускають неоднозначність, тобто такі, з яких виходить дві або більш дуги, помічені одним і тим же символом. Очевидно, що програмування роботи такого КА - нетривіальне завдання. Для простого програмування функціонування КА $M(Q, \Sigma, \delta, q_0, F)$ він повинен бути детермінованим - в кожному з можливих станів цього КА для будь-якого вхідного символу функція переходу повинна містити не більше одного стану:

Доведено, що будь-який недетермінований КА може бути перетворений в детермінований КА так, щоб їх мови співпадали [3, 7. 16] (говорять, що ці НО еквівалентні).

Окрім перетворення в детермінований КА будь-який КА може бути мінімізований - для нього може бути побудований еквівалентний йому детермінований КА з мінімально можливою кількістю стані. Алгоритми перетворення КА в детермінований КА і мінімізації КА детально описані [3, 7, 26]. Можна написати функцію, що відображає функціонування будь-якого детермінованого КА. Щоб запрограмувати таку функцію, досить мати змінну, яка б відображала поточний стан КА, а переходи з одного стану в інший на основі символів вхідного ланцюжка можуть бути побудовані за допомогою операторів вибору. Робота функції повинна продовжуватися до тих пір, поки не буде досягнутий кінець вхідного ланцюжка. Для обчислення результату функції необхідно після її завершення проаналізувати стан КА Якщо це один з кінцевих станів, то функція виконана успішно і вхідний ланцюжок приймається, якщо ні,

то вхідний ланцюжок не належить заданій мові. Проте в загальному випадку завдання лексичного аналізатора ширше, ніж просто перевірка ланцюжка символів лексеми на відповідність її вхідній мові. Він повинен правильно визначити кінець лексеми (про це було сказано вище) і виконати ті або інші дії по запам'ятовуванню розпізнаної лексеми (занесення її в таблицю лексем). Набір виконуваних дій визначається реалізацією компілятора. Звичайно ці дії виконуються відразу ж при виявленні кінця розпізнаваної лексеми.

У вхідному тексті лексеми не обмежені спеціальними символами. Визначення меж лексем - це виділення тих рядків в загальному потоці вхідних символів для яких треба виконувати розпізнавання. Якщо межі лексем завжди визначаються (а вище була прийнята саме така угода), то їх можна визначити по заданих термінальних символах і по символах початку наступної лексеми. Термінальні символи - це пропуски, знаки операцій, символи коментарів, а також роздільники (коми, крапки з комою і ін.). Набір таких термінальних символів може варіюватися залежно від вхідної мови. Важливо відзначити, що знаки операцій самі також є лексемами і необхідно не пропустити їх при розпізнаванні тексту.

Таким чином, алгоритм роботи простого сканера можна описати так:

- є видимим вхідний потік символів програми на початковій мові до виявлення чергового символу, що обмежує лексему;
- для вибраної частини вхідного потоку виконується функція розпізнавання лексеми;
- при успішному розпізнаванні інформація про виділену лексему заноситься в таблицю лексем, і алгоритм повертається до першого етапу;
- при неуспішному розпізнаванні видається повідомлення про помилку, а подальші дії залежать від реалізації сканера: або його виконання припиняється, або робиться спроба розпізнати наступну лексему (йде повернення до першого етапу алгоритму).

Робота програми-сканера продовжується до тих пір, поки не будуть переглянуті всі символи програми на початковій мові з вхідного потоку.

Вимоги до виконання роботи

Порядок виконання роботи

1. Одержати варіант завдання у викладача.

2. Побудувати опис КА, лежачого в основі лексичного аналізатора (у вигляді набору множин і функції переходів або у вигляді графа переходів).

3. Поготувати і захистити звіт.

4. Написати і відлагодити програму на ПК.

5. Здати працюючу програму викладачу.

Вимоги до оформлення звіту

- Звіт повинен містити наступні розділи:
- Завдання по лабораторній роботі.
- Опис КС-граматики вхідної мови у формі Бекуса-Наура.
- Опис алгоритму роботи сканера або граф переходів НО для розпізнавання ланцюжків (відповідно до варіанту завдання).
- Текст програми (оформляється після виконання програми на ПК). СJ Висновки по виконаній роботі.

Варіанти завдань

1. Вхідна мова містить арифметичні вирази, розділені символом; (крапка з комою). Арифметичні вирази складаються з ідентифікаторів десяткових чисел з плаваючою крапкою (у звичайній і логарифмічній формі), знаку привласнення (:=), знаків операцій +, - *, / і круглих дужок.

2. Вхідна мова містить логічні вирази, розділені символом; (крапка з комою). Логічні вирази складаються з ідентифікаторів, констант true і false, знаку привласнення (:=), знаків операцій or, xor, and, not і круглих дужок.

3. Вхідна мова мова містить операторів умови типу if…then...else і if…then, розділені символом; (крапка з комою). Оператори умови містять ідентифікатори, знаки порівняння <, >, =, десяткові числа з плаваючою крапкою (у звичайній і логарифмічній формі), знак привласнення (:=).

4. Вхідна мова містить операторів циклу типу for (...;.;...) do, розділених символом ; (крапка з комою). Оператори циклу містять ідентифікатори, знаки порівняння <, >, =, Десяткові числа з плаваючою крапкою (у звичайній і логарифмічній формі), знак привласнення (:=).

5. Вхідна мова містить арифметичні вирази, розділені символом ; (крапка з комою). Арифметичні вирази складаються з ідентифікаторів, римських чисел, знаку привласнення (:=). знаків операцій +, - *, / і круглих дужок.

6. Вхідна мова містить логічні вирази, розділені символом; (крапка з

комою). Логічні вирази складаються з ідентифікаторів, констант 0 і 1. знаку привласнення (:=), знаків операцій or, xor, and, not і круглих дужок.

7. Вхідна мова містить операторів умови типу if...tben...else й if…then, розділені символом ; (крапка з комою). Оператори умови містять ідентифікатори, знаку порівняння <, >, =, римські числа, знак привласнення (:=).

8. Вхідна мова містить операторів циклу типу for(...;...;...) do, розділених символом ; (крапка з комою). Оператори циклу містять ідентифікатори, знаки порівняння <, >, =, римські числа, знак привласнення (:=).

9. Вхідна мова містить арифметичні вирази, розділені символом ; (крапка з комою). Арифметичні вирази складаються з ідентифікаторів, шістнадцятирічних чисел, знаку привласнення (:=), знаків операції " +. - *, / і круглих дужок

10. Вхідна мова містить логічні вирази, розділені символом; (крапка з комою). Логічні вирази складаються з ідентифікаторів, шістнадцятирічних чисел, знаку привласнення (:=). знаків операцій or, xor, and, not і круглих дужок

11. Вхідна мова містить операторів умови типу if...tben…else й if...then, розділені символом; (крапка з комою). Оператори умови містять ідентифікатори, знаки порівняння <, >, =, шістнадцятирічні числа, знак привласнення (:=).

12. Вхідна мова містить операторів циклу типу for(...;...;...) do, розділених символом; (крапка з комою). Оператори циклу містять ідентифікатори, знаки порівняння <, >, =, шістнадцятирічні числа, знак привласнення (:=).

13. вхідна мова містить арифметичні вирази, розділені символом; (крапка з комою). арифметичні вирази складаються з ідентифікаторів, символьних констант (один символ в одинарних лапках), знаку привласнення (:=), знаків операцій +, - *, / і круглих дужок.

14. вхідна мова містить логічні вирази, розділені символом; (крапка з комою). Логічні вирази складаються з ідентифікаторів, символьних констант 'T' і 'F', знаку привласнення (:=), знаків операцій or, xor, and. not і круглих дужок.

15. вхідна мова містить операторів умови типу if…then…else і if…then, розділені символом ; (крапка з комою). Оператори умови містять ідентифікатори, знаки порівняння <, >, =, строкові константи (послідовність символів в подвійних лапках), знак привласнення (:=).

16. Вхідна мова містить операторів циклу типу for(...;...;.) do, розділених символом ; (крапка з комою). Оператори циклу містять ідентифікатори, знаки порівняння <, >, =, строкові константи (послідовність символів в подвійних лапках), знак привласнення (:=).

ПРИМІТКА

- *Римськими числами вважати послідовності заголовних латинських букв X, V і I;*
- *шістнадцятирічними числами вважають послідовність цифр і символів «a», «b», «c», «d», «e», і «f», що починаються з цифри (наприклад: 89, 45ac9. 0abc4);*
- *завдання по лабораторній роботі № 2 взаємозв'язане із завданням по лабораторній роботі № 3, для уточнення складу вхідної мови можна подивитися граматику, задану в роботі № 3 по відповідному варіанту.*

Контрольні запитання:

1) Що таке трансляція, компіляція, транслятор, компілятор?
2) З яких процесів складається компіляція? Розкажіть про загальну структуру компілятора.
3) Яку роль виконує лексичний аналіз в процесі компіляції?
4) Що таке лексема? Розкажіть, які типи лексем існують в мовах програмування.
5) Як можуть бути зв'язані між собою лексичний і синтаксичний аналіз?
6) Які проблеми можуть виникати при визначенні меж лексем в процесі лексичного аналізу? Як розв'язуються ці проблеми?
7) Що таке таблиця лексем? Яка інформація зберігається в таблиці лексем?
8) У чому різниця між таблицею лексем і таблицею ідентифікаторів?
9) Що таке граматика? Дайте визначення граматики. Як виглядає опис граматики у формі Бекуса-Наура.
10) Які класи граматик існують? Що таке регулярні граматики?
11) Що таке кінцевий автомат? Дайте визначення детермінованого і недетермінованого кінцевих автоматів.
12) Опишіть алгоритм перетворення недетермінованого кінцевого автомата в

детермінований.

13) Які проблеми необхідно вирішити при побудові сканера на основі кінцевого автомата?

14) Поясніть загальний алгоритм функціонування лексичного аналізатора.

Лабораторна робота №3

Тема: ПОБУДОВА НАЙПРОСТІШОГО ДЕРЕВА ВИВОДУ

Мета: Вивчення основних понять теорії граматик простого і операторного передування, ознайомлення з алгоритмами синтаксичного аналізу (розбору) Для деяких класів КС-граматик, отримання практичних навичок створення простого синтаксичного аналізатора для заданої граматики операторного передування.

Завдання:

Написати програму, яка виконує лексичний аналіз вхідного тексту відповідно до завдання, породжує таблицю лексем і виконує синтаксичний розбір тексту по заданій граматиці з побудовою дерева розбору. Текст на вхідній мові задається у вигляді символьного (текстового) файлу. Синтаксис вхідної мови і перелік допустимих лексем вказані в завданні. Допускається, що текст містить не більш за одну пропозицію вхідної мови.

За наявності у вхідному файлі тексту, відповідного заданій мові, програма повинна будувати і відображати дерево синтаксичного розбору. Якщо ж текст у вхідному файлі містить помилки (лексичні або синтаксичні), програма повинна видавати повідомлення про наявність помилок у вхідному тексті і коректно завершувати своє виконання.

Рекомендується розбити програму на три складові частини: лексичний аналіз, побудова ланцюжка виводу та побудову дерева виводу. Лексичний аналізатор повинен виділяти в тексті лексеми мови і замінювати їх на термінальний символ граматики (який в завданні позначений як a). Одержаний після лексичного аналізу ланцюжок повинен розглядатися в другій частині програми відповідно до алгоритму розбору При невдалому завершенні алгоритму видається повідомлення про помилку, при вдалому - будується ланцюжок виводу. Після побудови ланцюжка виводу на її основі будується дерево розбору, в якому символи a послідовно замінюються на лексеми з таблиці лексем. Для виконання лексичного аналізу рекомендується використовувати

програмні модулі, створені в результаті виконання лабораторної роботи № 2. Довжину ідентифікаторів і строкових констант можна вважати обмеженою 32 символами. Програма повинна допускати коментарі необмеженої довжини у вхідному файлі. Форму організації коментарів пропонується обрати самостійно.

Теоретичні відомості:

Призначення синтаксичного аналізатора

За ієрархією граматик Хомського виділяють чотири основні групи мов (і граматик, що описують їх) [1, 3, 4, 7]. При цьому найбільший інтерес представляють регулярні і контекстно-свободні (КС) граматики і мови. Вони використовуються при описі синтаксису мов програмування. За допомогою регулярних граматик можна описати лексеми мови - ідентифікатори. константи, службові слова та інші. На основі КС-граматик будуються крупніші синтаксичні конструкції: описи типів і змінних. арифметичні і логічні вирази, оператори, що управляють, і. нарешті, повністю вся програма на вхідній мові.

Вхідні ланцюжки регулярних мов розпізнаються за допомогою кінцевих автоматів (КА). Вони лежать в основі сканерів, що виконують лексичний аналіз і виділення слів в тексті програми на вхідній мові. Результатом роботи сканера є перетворення початкової програми в список або таблицю лексем. Подальшу її роботу виконує інша частина компілятора - синтаксичний аналізатор. Його робота заснована на використанні правил КС-граматики, що описують конструкції початкової мови.

Синтаксичний аналізатор (синтаксичний пристрій для розбору) - це частина компілятора. яка відповідає за виявлення і перевірку синтаксичних конструкцій вхідної мови. У завдання синтаксичного аналізатора входить:

знайти і виділити синтаксичні конструкції в тексті початкової програми;

встановити тип і перевірити правильність кожної синтаксичної конструкції;

представити синтаксичні конструкції у вигляді, зручному для подальшої

генерації тексту результуючої програми.

Синтаксичний аналізатор - це основна частина компілятора на етапі аналізу. Без виконання синтаксичного розбору робота компілятора безглузда, тоді як лексичний розбір, у принципі, не є обов'язковою фазою компіляції. Всі завдання по перевірці синтаксису вхідної мови можуть бути вирішені на етапі синтаксичного розбору. Лексичний аналізатор тільки дозволяє позбавити складний по структурі синтаксичний аналізатор від рішення примітивних задач по виявленню і запам'ятовуванню лексем початкової програми.

Виходом лексичного аналізатора є таблиця лексем. Ця таблиця утворює вхід синтаксичного аналізатора, який досліджує тільки один компонент кожної лексеми - її тип. Решта інформації про лексеми використовується на пізніших етапах компіляції при семантичному аналізі, підготовці до генерації і генерації коду результуючої програми.

Синтаксичний аналізатор сприймає вихід лексичного аналізатора і розбирає його відповідно до граматики вхідної мови. Проте в граматиці вхідної мови програмування звичайно не уточнюється, які конструкції слід вважати лексемами. Прикладами конструкцій, які звичайно розпізнаються під час лексичного аналізу, служать ключові слова, константи і ідентифікатори. Але ці ж конструкції можуть розпізнаватися і синтаксичним аналізатором. На практиці не існує жорсткого правила, що визначає, які конструкції повинні розпізнаватися на лексичному рівні, а які треба залишати синтаксичному аналізатору. Звичайне це визначає розробник компілятора виходячи з технологічних аспектів програмування, а також синтаксису і семантики вхідної мови. Принципи взаємодії лексичного і синтаксичного аналізаторів були розглянуті в лабораторній роботі №2.

У основі синтаксичного аналізатора лежить пристрій для розпізнавання тексту початкової програми, побудований на основі граматики вхідної мови. Як правило, синтаксичні конструкції мов програмування можуть бути описані за допомогою КС-граматик; рідше зустрічаються мови, які можуть бути описані за допомогою регулярних граматик.

Головну роль в тому, як функціонує синтаксичний аналізатор і який алгоритм лежить в його основі, грають принципи побудови пристрою для розпізнавання для КС-мов. Без застосування цих принципів неможливо виконати ефективний синтаксичний розбір пропозицій вхідної мови.

Проблема розпізнавання ланцюжків КС-мов

Взаємодія лексичного і синтаксичного аналізаторів розглядалася в попередній лабораторній роботі, тут же будуть розглянуті алгоритми, які лежать в основі синтаксичного аналізу. Перед синтаксичним аналізатором стоять два основні завдання: перевірити правильність конструкцій програми, яка представляється у вигляді вже виділених слів вхідної мови, і перетворити її у вигляд, зручний для подальшої семантичної (смислової) обробки і генерації коду. Одним із способів такого уявлення є дерево синтаксичного розбору.

Основою для побудови пристрою для розпізнавання КС-мов є автомати з магазинною пам'яттю - МП-автомати - односторонні недетерміновані пристрої для розпізнавання з лінійно-обмеженою магазинною пам'яттю (повна класифікація пристроїв для розпізнавання приведена [1, 4, 3. 7]). Тому важливо розглянути, як функціонує МП-автомат і як для КС-мов розв'язується завдання розбору - побудова пристрою для розпізнавання мови На основі заданої граматики. Далі розглянуті технічні аспекти, пов'язані з реалізацією синтаксичних аналізаторів.

МП-автомат на відміну від звичайного КА має стек (магазин). у який можна поміщати спеціальні – «магазинні» символи (звичайне це термінальні і нетермінальні символи граматики мови). Перехід МП-автомата з одного стану в інший залежить не тільки від вхідного символу, але і від одного або декількох верхніх символів стека. Таким чином, конфігурація автомата визначається трьома параметрами: станом автомата, поточним символом вхідного ланцюжка (положенням покажчика в ланцюжку) і вмістом стека.

При виконанні перехода МП-автомата з однієї конфігурації в іншу із стека віддаляються верхні символи, відповідні умові переходу, і додається ланцюжок, відповідний правилу переходу. Перший символ ланцюжка стає верхівкою

стека. Допускаються переходи, при яких вхідний символ ігнорується (і тим самим він буде вхідним символом при наступному переході). Ці переходи називаються л-переходами. Якщо при закінченні ланцюжка автомат знаходиться в одному із заданих кінцевих станів, а стек порожній, ланцюжок вважається прийнятим (після закінчення ланцюжка можуть бути зроблені л-переходи). Інакше ланцюжок символів не приймається.

МП-автомат називається таким, що не термінує, якщо при одній і тій же його конфігурації можливий більш ніж один перехід. Інакше (якщо з будь-якої конфігурації МП-автомата по будь-якому вхідному символу можливо не більше одного переходу в наступну конфігурацію) МП-автомат вважається детермінованим (ДМП-автоматом). ДМП-автомати задають клас детермінованих КС-мов, для яких існують однозначні КС-граматики. Саме цей клас мов лежить в основі синтаксичних конструкцій всіх мов програмування, оскільки будь-яка синтаксична конструкція мови програмування повинна допускати тільки однозначне трактування [1-4, 7].

По довільній КС-граматиці G(VN,VT,P,S), V - VT∪VN завжди можна побудувати недетермінований МП-автомат, який допускає ланцюжки мови, заданої цією граматикою [1-3,7]. А на основі цього МП-автомата можна створити пристрою розпізнавання для заданої мови.

Проте при алгоритмічній реалізації функціонування такого пристрою розпізнавання можуть виникнути проблеми. Річ у тому, що: побудований МП-автомат буде, як правило, недетермінованим, а для МП-автоматів, на відміну від звичайних НО, не існує алгоритму, який дозволяв би перетворити довільний МП-автомат в ДМП-автомат. Тому програмування функціонування МП-автомата - нетривіальне завдання. Якщо моделювати його функціонування по кроках з перебором всіх можливих станів, то може опинитися, що побудований для тривіального МП-автомата алгоритм ніколи не завершиться на кінцевому вхідному ланцюжку символів за певних умов. Приклади таких МП-автоматів можна знайти в [1,3,7].

Тому для побудови пристрою розпізнавання для мови, заданої КС-

граматикою, рекомендується скористатися відповідним математичним апаратом і одним з існуючих алгоритмів.

Види пристроїв розпізнавання для КС-мов

Існують нескладні перетворення КС-граматик, виконання яких гарантує, що побудований на основі перетвореної граматики МП-автомат можна буде промоделювати за кінцевий час на основі кінцевих обчислювальних ресурсів. Опис суті і алгоритмів цих перетворень можна знайти в [1, 3, 7].

Ці перетворення дозволяють будувати два основних типу простих пристроїв розпізнавання:

- пристрій розпізнавання з підбором альтернатив;
- пристрій розпізнавання на основі алгоритму «згортка зрушення».

Роботу пристрою розпізнавання з підбором альтернатив можна неформально описати таким чином: якщо на верхівці стека МП-автомата знаходиться нетермінальний символ А, то його можна замінити на ланцюжок символів о. за умови, що в граматиці мови є правило A→α не зрушуючи при цьому читаючу головку автомата (цей крок роботи називається «підбір альтернативи»); якщо ж на верхівці стека знаходиться термінальний символ би, який співпадає з поточним символом вхідного ланцюжка, то цей символ можна викинути із стека і пересунути ту, що прочитує головку на одну позицію управо (цей крок роботи називається викид). Даний МП-автомат може бути недетермінованим. оскільки при підборі альтернативи в граматиці мови може опинитися більше одного правила виду A→α, тоді функція δ(q,λ,A) міститиме більше одного наступного стану - у МП-автомата буде декілька альтернатив.

Рішення про те, чи виконувати на кожному кроці роботи МП-автомата викид або підбір альтернативи, ухвалюється однозначно. Моделюючий алгоритм повинен забезпечувати вибір однієї з можливих альтернатив і зберігання інформації про те, які альтернативи на якому кроці вже були вибрані. щоб мати можливість повернутися до цього кроку і підібрати інші альтернативи.

Пристрій розпізнавання з підбором альтернатив є низхідним пристроєм розпізнавання: він читає вхідний ланцюжок символів зліва направо і будує лівобічний висновок. Назва .• низхідний. дано йому тому. що дерево висновку в цьому випадку слід будувати зверху вниз, від кореня до кінцевих вершин – «листю» (на відміну від звичайних дерев, корінь у синтаксичного дерева висновку знаходиться вгорі, а листя - внизу).

Роботу пристрою розпізнавання на основі алгоритму «згортка зрушення». можна описати так: якщо на верхівці стека МП-автомата знаходиться ланцюжок символів У. то її можна замінити на нетермінальний символ А за умови, що в граматиці мови існує правило вигляду А У. не зрушуючи при цьому читаючу головку автомата (цей крок роботи називається «згортка»); з іншого боку, якщо читаюча головка автомата оглядає деякий символ вхідного ланцюжка а, то його можна помістити в стек, зрушивши при цьому головку на одну позицію управо (цей крок роботи називається «зрушення» або «перенесення»).

Цей пристрій розпізнавання потенційно має більше за неоднозначностей чим розглянутий вище за пристрій розпізнавання, заснований на алгоритмі підбору альтернатив. На кожному кроці роботи автомата треба вирішувати наступні питання:

- що необхідно виконувати: зрушення або згортку;
- якщо виконувати згортку, то такий ланцюжок у вибрати для пошуку правил (ланцюжок у повинна зустрічатися в правій частині правил граматики);
- яке правило вибрати для згортки, якщо опиниться, що існує декілька правил вигляду Ару (дещо правил з однаковою правою частиною).

Для моделювання роботи цього розширеного МП-автомата треба на кожному кроці запам'ятовувати всі зроблені дії, щоб мати можливість повернутися до вже зробленого кроку і виконати ці ж дії по-іншому. Цей процес повинен повторюватися до тих пір, поки не будуть перебрані всі

можливі варіанти.

Пристрій розпізнавання на основі алгоритму «згортка зрушення» є висхідним пристроєм розпізнавання: він читає вхідний ланцюжок символів зліва направо і будує правосторонній висновок. Назва висхідний дано йому тому, що дерево висновку в цьому випадку слід будувати від низу до верху, від кінцевих вершин до кореня.

Функціонування обох розглянутих пристроїв розпізнавання реалізується достатньо простими алгоритмами, які можна знайти в [3. 7]. Проте обидва вони мають один істотний недолік - час їх функціонування експоненціально залежить від довжини вхідного ланцюжка n=|α|, що неприпустимо для компіляторів, де довжина вхідних програм складає від десятків до сотень тисяч символів. Так відбувається тому, що обидва алгоритми виконують розбір вхідного ланцюжка символів методом простого перебору, підбираючи правила граматики довільним чином, а у разі невдачі повертаються до вже прочитаної частини вхідного ланцюжка і намагаються підібрати інші правила.

Існують ефективніші табличні пристрої розпізнавання, побудовані на основі алгоритмів Эрлі і Кока-Янгера-Касамі [1, 3]. Вони забезпечують поліноміальну залежність часу функціонування від довжини вхідного ланцюжка (n^3 для довільного МП-автомата і n^2 для ДМП-автомата), Це найефективніші з універсальних пристроїв розпізнавання для КС-мов. Але і поліноміальну залежність часу розбору від довжини вхідного ланцюжка не можна визнати задовільною.

Кращих універсальних пристроїв розпізнавання не існує. Проте серед всього типу КС-мов існує множина класів і підкласів мов, для яких можна побудувати пристрої розпізнавання, функціонування, що мають лінійну залежність часу, від довжини вхідного ланцюжка символів. Такі пристрої розпізнавання називають лінійними пристроями розпізнавання КС-мов.

В теперішній час відома множина лінійних пристроїв розпізнавання і відповідних їм класів КС-мов. Кожний з них має свій алгоритм функціонування, але всі відомі алгоритми є модифікацією двох базових

алгоритмів - алгоритму з підбором альтернатив і алгоритму згортка зрушення, розглянутих вище. Модифікації полягають в тому, що алгоритми виконують підбір правил граматики для розбору вхідного ланцюжка символів не довільним чином, а керуючись встановленим порядком, який створюється заÈ раніше на основі заданої КС-граматика. Такий підхід дозволяє уникнути повернень до вже прочитаної частини ланцюжка і істотно скорочує час, потрібний на її розбір.

Серед всієї множини можна виділити наступні найбільш часто використовувані пристрої розпізнавання:

- пристрої розпізнавання на основі рекурсивного спуску (модифікація алгоритму з підбором альтернатив);
- пристрої розпізнавання на основі LL(1) - і LL(k) -граматик (модифікація алгоритму з підбором альтернатив);
- пристрої розпізнавання на основі LR(0) - і LR(1) -граматик (модифікація алгоритму згортка зрушення);
- пристрої розпізнавання на основі SLR(1) - і LALR(1) -граматик (модифікація алгоритму згортка зрушення);
- пристрої розпізнавання на основі граматик передування (модифікація алгоритму згортка зрушення).

Алгоритми Функціонування всіх відомих і ряду інших лінійних пристроїв розпізнавання описані [1-4,7].

Побудова синтаксичного аналізатора

Синтаксичний аналізатор повинен розпізнавати весь текст початкової програми. Тому, на відміну від лексичного аналізатора, йому немає необхідності шукати

межі розпізнаваного рядка символів. Він повинен сприймати всю формацію. що поступає йому на вхід, і або підтвердити її приналежність вхідному мова або повідомити про помилку в початковій програмі.

Але, як і у разі лексичного аналізу, завдання синтаксичного аналізу не обмежується тільки перевіркою приналежності ланцюжка заданій мові. Необхідно оформити знайдені синтаксичні конструкції для подальшої генерації тексту результуючою програмою. Синтаксичний аналізатор повинен мати якусь вихідну мову, за допомогою якої він передає наступним фазам компіляції інформацію про знайдені і розібрані синтаксичні структури. У такому разі він вже є не різновидом МП-автомату, а перетворювачем з магазинною пам'яттю - МП-перетворювачем [1, 2, 7].

Питання, пов'язані з представленням інформації, роботи синтаксичного аналізатора, що є результатом, і з породженням на основі цієї інформації тексту результуючої програми, розглянуті в лабораторній роботі È 4, тому тут на них зупинятися не будемо.

Побудова синтаксичного аналізатора - це більш творчий процес, ніж побудова лексичного аналізатора. Цей процес не завжди може бути повністю формалізований.

Маючи граматику вхідної мови, розробник синтаксичного аналізатора повинен в першу чергу виконати ряд формальних перетворень над цією граматикою, що полегшують побудову пристрою розпізнавання. Після цього він повинен перевірити, чи відноситься одержана граматика до одного з відомих класів КС-мов, для яких існують лінійні пристрої розпізнавання. Якщо такий клас знайдений, можна будувати пристрій розпізнавання (якщо знайдено декілька класів, слід вибрати той, для якого побудова пристрою розпізнавання простіше або побудований пристрій розпізнавання володітиме кращими характеристиками). Якщо ж такий клас КС-мов знайти не вдалося, то розробник повинен спробувати виконати над граматикою деякі перетворення, щоб привести її до одного з відомих класів. Ці перетворення не можуть бути описані формально, і у кожному конкретному випадку розробник повинен спробувати знайти їх сам (іноді перетворення має сенс шукати навіть у тому випадку, коли граматика підпадає під один з відомих класів КС-мов, з метою знайти інший клас, для якого можна побудувати кращий по характеристиках пристрій

розпізнавання).

Складнощів з побудовою синтаксичних аналізаторів не існувало б, якби для КС-граматик були дозволені проблеми перетворення і еквівалентності. Але оскільки в загальному випадку це не так, то одним класом КС-граматик, для якого існують лінійні пристрої розпізнавання, обмежитися не вдається. З цієї причини для всіх класів КС-граматик існує принципово важливе обмеження: У загальному випадку неможливо перетворити довільну КС-граматику до вигляду, потрібного даним класом КС-граматик, або ж довести, що такого перетворення не існує. Те, що проблема нерозв'язна в загальному випадку, не говорить про те, що вона не розв'язується в кожному конкретному окремому випадку, і часто вдається знайти такі перетворення. І чим ширше набір класів КС-граматик з лінійними пристроями розпізнавання, тим простіше за них шукати.

Тільки, коли в результаті всіх цих дій не вдалося знайти відповідний клас КС-мов, розробник вимушений будувати універсальний пристрій розпізнавання. Характеристики такого пристрою розпізнавання будуть істотно гірші. чим у лінійного пристрою розпізнавання: в кращому разі вдається досягти квадратичної залежності часу роботи пристрою розпізнавання від довжини вхідного ланцюжка. Таке рідкісне, тому всі сучасні компілятори побудовані на основі лінійних пристроїв розпізнавання (інакше час їх роботи був би неприпустимо великий).

Часто одна і та ж КС-граматика може бути віднесена не до одного. а відразу до декількох класів КС-граматик, що допускають побудову лінійних распознавателей. Тоді необхідно вирішити, який з декількох можливих пристроїв розпізнавання вибрати для практичної реалізації.

Відповісти на це питання не завжди легко, оскільки можуть бути побудовані два принципово різних пристрою розпізнавання, алгоритми роботу яких неможна співставити. В першу чергу йдеться саме про висхідних і низхідних пристроїв розпізнавання: у основі перших лежить алгоритм підбору альтернатив, в основі других - алгоритм згортка зрушення.

На питання про те, якої пристрій розпізнавання - низхідний або висхідний - вибрати для побудови синтаксичного аналізатора, немає однозначної відповіді. Цю проблему необхідно вирішувати, спираючись на якусь додаткову інформацію про те, як будуть використані або яким чином будуть оброблені результати роботи пристрою розпізнавання. Детальніше обговорення цього питання можна знайти в [1, 7].

ПОРАДА

Слід пригадати, що синтаксичний аналізатор - це один з етапів компіляції. І з цієї точки зору результати роботи пристрою розпізнавання служать початковими даними для наступних етапів компіляції. Тому вибір того або іншого пристрою розпізнавання багато в чому залежить від реалізації компілятора, від того, які принципи покладені в його основу.

Бажання використовувати простіший клас граматик для побудови пристрою розпізнавання може зажадати якихось маніпуляцій із заданою граматикою, необхідних для її перетворення до необхідного класу. При цьому нерідко граматика стає неприродною і малозрозумілою, що надалі утрудняє її використання для генерації результуючого коду. Тому буває зручним використовувати початкову граматику такої, яка вона є, не прагнучи перетворити її до простішого класу

В цілому слід зазначити, що, з урахуванням всього сказаного. інтерес представляють як лівобічний, так і правосторонній аналіз, Конкретний вибір залежить від реалізації конкретного компілятора, а також від складності граматики вхідної мови програмування.

У загальному вигляді процес побудови синтаксичного аналізатора можна описати таким чином:

1. Виконати прості перетворення над заданою КС-граматикою.
2. Перевірити приналежність КС-граматики, що вийшла в результаті перетворень, до одного з відомих класів КС-граматик, для яких існують лінійні пристрої розпізнавання.

3. Якщо відповідний клас знайдений, узяти за основу для побудови пристрою розпізнавання алгоритм розбору вхідних ланцюжків, відомий для цього класу, якщо знайдено декілька класів лінійних пристроїв розпізнавання - вибрати з них один на свій розсуд.
4. Інакше, якщо відповідний клас по п. 2 не був знайдений чи ж знайдений клас КС-граматик не влаштовує розробників компілятора - спробувати виконати над граматикою неформальні перетворення з метою підвести її під клас КС-граматик, що цікавить, для лінійних пристроїв розпізнавання і повернутися до п. 2.
5. Якщо ж ні в п. 3, ні в п. 4 відповідний пристрій розпізнавання знайти не вдалося (що для сучасних мов програмування практично неможливо). необхідно використовувати один з універсальних пристроїв розпізнавання.
6. Визначити, в якій формі синтаксичний пристрій розпізнавання передаватиме результати своєї роботи іншим фазам компілятора (ця форма називається внутрішнім представленням програми в компіляторі).

Реалізувати вибраний в п. 3 або 5 алгоритм з урахуванням структур даних, відповідних п. 6.

У даній лабораторній роботі в завданнях пропонуються граматики, що не вимагають додаткових перетворень. Крім того, гарантовано, що всі вони відносяться до масу КС-граматик операторного передування, для яких існує відомий алгоритм лінійного пристрою розпізнавання. Тому створення синтаксичного пристрою розпізнавання для виконання лабораторної роботи істотно спрощується.

Для граматик, запропонованих в завданнях, відомо, що вони відносяться також до класів КС-граматик LR(1) і LALR(1), для яких також існує відомий алгоритм лінійного пристрою розпізнавання, по, на думку автора, цей алгоритм складніший (його опис можна знайти в [1,2,7]). Проте охочі можуть не погодитися з автором і використовувати для виконання лабораторної роботи будь-якої з цих класів.

Після нескладних перетворень ці ж граматики можуть бути приведені до вигляду, що задовольняє вимогам алгоритму рекурсивного спуску (або алгоритму аналізу для LL(1) -грамматик). Цей алгоритм тривіально простий, але для його реалізації треба виконати достатньо нескладні неформальні перетворення Над заданими граматиками - автор залишає ці перетворення для охочих спробувати свої сили.

Що виконують лабораторну роботу можуть піти будь-яким з рекомендованих шляхів або побудувати інший синтаксичний аналізатор на свій розсуд - в цьому напрямі їх ніщо не обмежує.

Як основний шлях виконання лабораторної роботи автор пропонує пристрій розпізнавання на основі граматик операторного передування, тому саме цей клас КС-граматик далі розглянутий детальніше (описи решти відомих класів і підкласів КС-граматик можна знайти в [1-3. 7]).

Граматики передування

КС- мови діляться на класи відповідно до структури правил їх граматик. В кожному з класів накладаються додаткові обмеження на допустимі правила граматики. Одним з таких класів є клас граматик передування. Вони використовуються для синтаксичного розбору ланцюжків за допомогою модифікацій алгоритму «зсув-згортка».

Принцип організації розпізнавача на основі граматики передування виходить з того, що для кожної впорядкованої пари символів в граматиці встановлюється відношення, зване відношенням передування. В процесі розбору МП - автомат порівнює поточний символ вхідного ланцюжка з одним з символів, що знаходяться на верхівці стека автомата. В процесі порівняння перевіряється, яке з можливих відносин передування існує між цими двома символами. Залежно від знайденого відношення виконується або зрушення, або згортка За відсутності відношення передування між символами алгоритм сигналізує про помилку.

Завдання полягає в тому. щоб мати можливість несуперечливим чином

визначити відносини передування між символами граматики. Якщо це можливо, то граматика може бути віднесена до одного з класів граматик передування.

Відносини передування позначатимемо знаками «-.», «<.» і «.>». Відношення передування єдине для кожної впорядкованої пари символів. При цьому між якими-небудь двома символами може і не бути відношення передування це означає, що вони не можуть знаходитися поряд або в одному елементі розбору синтаксично правильного ланцюжка. Відносини передування залежать від порядку, і якому коштують символи, і в цьому сенсі їх не можна плутати із знаками математичних операцій (хоча але зовнішньому вигляду вони дуже схожі) - вони не володіють ні властивістю комутативності, ні властивістю асоціативності. Наприклад, якщо відомо, що В1 .> В, то не обов'язково виконується В <. В1 тому знаки передування позначають спеціальною крапкою: «=.» «<.» «.>»). Метод передування заснований на тому факті, що відносини передування між двома сусідніми символами розпізнаваного рядка відповідають трьом наступним варіантам:

- Bi <. В $_{i+1}$ якщо символ В $_{i+1}$ - крайній лівий символ деякої основи (це відношення між символами можна назвати «передує основі» або просто «передує»);
- Bi. > В $_{i+1}$, якщо символ Bi — крайній правий символ деякої основи (це відношення між символами можна назвати «слідує за основою» або просто «слідує»);
- Bi =. В $_{i+1}$ якщо символи В $_i$ і В $_{i+1}$ належать одній основі (це відношення між символами можна назвати «складають основу»).

Виходячи з цих співвідношень виконується розбір вхідного рядка для граматик передування.

Суть принципу такого розбору пояснює Рис. 3.1. На ньому зображений вхідний ланцюжок символів **α β γ δ** в той момент, коли виконується згортка ланцюжка **γ**. Символ **α** є останнім символом підланцюжка α , а символ β—

першим символом під ланцюжка β. Тоді, якщо в граматиці вдасться встановити несуперечливі відносини передування, то в процесі виконання розбору по алгоритму «зсув-згортка» можна завжди виконувати зрушення до тих пір, поки між символом на верхівці стека і поточним символом вхідного ланцюжка існує відношення <. або =. А як тільки між цими символами буде виявлено відношення .>, відразу треба виконувати згортку. Причому для виконання згортки із стека треба вибирати всі символи, зв'язані відношенням =. Всі різні правила в граматиці передування повинні мати різні праві частини — це гарантує несуперечність вибору правила при виконанні згортки.

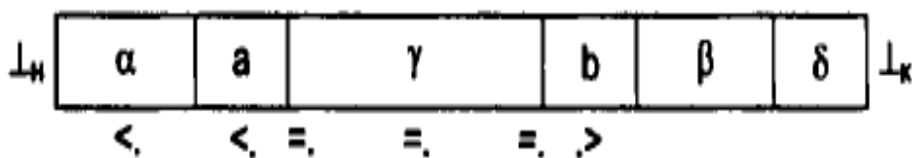

Рис. 3.1. Відношення між символами вхідного ланцюжка в граматиці передування

Таким чином, встановлення несуперечливих відносин передування між символами граматики в комплексі з неспівпадаючими правими частинами різних правил дає відповіді на всі питання, які треба вирішити для організації роботи алгоритму «зсув-згортка» без повернень.

На підставі відносин передування будують матрицю передування граматики. Рядки матриці передування позначаються першими (лівими) символами, стовпці - другими (правими) символами відносин передування. У клітки матриці на перетині відповідного стовпця і рядка поміщаються знаки відносин. При цьому порожні клітки матриці говорять про те. що між даними символами немає жодного відношення передування. Існує декілька видів граматик передування. Вони розрізняються по тому, які відносини передування в них визначені і між якими типами символів (термінальними або нетермінальними) можуть бути встановлені ці відносини. Крім того, можливі незначні модифікації функціонування самого алгоритму «зсув-згортка» в розпізнавачах для таких граматик (в основному на етапі вибору правила для

виконання згортки, коли можливі неоднозначності) (1).

Виділяють наступні види граматик передування:

- простого передування;
- розширеного передування;
- слабкого передування;
- змішаній стратегії передування;
- операторного передування.

Далі будуть розглянуті обмеження на структуру правив і алгоритми розбору для граматик операторного передування.

Матрицю операторного передування КС- граматики можна побудувати, спираючись безпосередньо на визначення відносин передування (1,3,7), але простіше і зручніше скористатися двома додатковими типами множин — множиначю крайніх лівих і крайніх правих символів, а також множиначю крайніх лівих термінальних і крайніх правих термінальних.

Якщо є КС- граматика G(VT,VN,P,S), V = VT U VN. то множина крайніх лівих і крайніх правих символів визначається таким чином:

- L(U)={T|Э U =>*Tz}множина крайніх лівих символів щодо нетермінального символу U;
- R(U)={T|Э U =>* zT} множина крайніх правих символів щодо нетермінального символу U.

де U — заданий нетермінальний символ (U є VN), T — будь-який символ граматики (T є V).a z — довільний ланцюжок символів(z є V*). Ланцюжок z може бути і порожнім ланцюжком).

Множина крайніх лівих і крайніх правих термінальних символів визначається таким чином:

$L_t(U) = \{t \mid \exists U \Rightarrow {}^*tz \text{ или } \exists U \Rightarrow {}^*Ctz\}$ — множина крайніх лівих термінальних символів щодо нетермінального символу U;

$\mathbf{R}_t(U) = \{t \mid \exists\, U \Rightarrow {}^*zt \text{ или } \exists\, U \Rightarrow {}^*ztC\}$ —множина крайніх правих термінальних символів щодо нетермінального символу U

де t— термінальний символ ($t \in \mathbf{VT}$), U і C — нетермінальні символи ($C \in \mathbf{VN}$), а z довільний ланцюжок символів($z \in \mathbf{V}^*$ланцюжок z може бути і пустим ланцюжком).

Множини L(U) і R(U) можуть бути побудовані для кожного нетермінального символу $U \in \mathbf{VN}$по дуже простому алгоритму:

1. Для кожного нетермінального символу U шукаємо всі правила, що містять U в лівій частині. В множині L(U) включаємо самий лівий символ з правої частини правила, а в множині R(U) – самий правий символ з правої частини (тобто в множині L (U) записуємо всі символи, з яких починаються правила для символу U, а в множині R(U) — символи, якими ці правила закінчуються). Якщо в правій частині правила для символу U є тільки один символ, то він повинен бути записаний в обидві множини L(U) і R(U).
2. Для кожного нетермінального символу U виконуємо наступне перетворення: якщо множина L(U) містить нетермінальні символи граматики U' U".... то його потрібно доповнити символами, що входять у відповідну множину $\mathbf{L}(U')$, $\mathbf{L}(U'')$,і що не входять в $\mathbf{L}(U)$., Ту ж операцію потрібно виконати для R(U). Фактично, якщо якийсь символ U входить в одну з множин для символу U, то потрібно об'єднати множини для U" та U, а результат записати в множину для символу U.

3. Якщо на попередньому кроці хоч би одна множина L(U)або R(U) для деякого символу граматики змінилося, то треба повернутися до кроку 2, інакше –побудова закінчена.

Для знаходження множини $\mathbf{L}_t(U)$ и $\mathbf{R}_t(U)$використовується наступний алгоритм:

1. Для кожного нетермінального символу граматики U будуються множини L(U) та R(U).

2. Для кожного нетермінального символу граматики U шукаються правила виду $U \to tz$ и $U \to Ctz$, где $t \in \mathbf{VT}, C \in \mathbf{VN}, z \in \mathbf{V}^*$;термінальні символи t включаються в множину $\mathbf{L}_t(U)$. Аналогічно для множини $\mathbf{R}_t(U)$шукаються правила вигляду $U \to zt$ и $U \to ztC$(тобто в множину $\mathbf{L}_t(U)$записуються всі крайні зліва термінальні символи з правих частин правил для символу U, а в множину $\mathbf{R}_t(U)$— всі крайні справа термінальні символи цих правил). Не виключено, що один і той же термінальний символ буде записаний в обидві множини $\mathbf{L}_t(U)$ и $\mathbf{R}_t(U)$.

3. Проглядається множина L(U), в яку входять символи $U', U''...$. Множина $\mathbf{L}_t(U)$доповнюється термінальними символами, що входять в $\mathbf{L}_t(U')...$.. і що не входять в $\mathbf{L}_t(U)$. Аналогічна операція виконується і для множини $\mathbf{R}_t(U)$на основі множини R (U).

Для практичного використання матрицю передування доповнюють термінальними символами $\perp_н$ и $\perp_к$(початок і кінець ланцюжка). Для них визначені наступні відносини передування:

$$\perp_н <\cdot\ a, \text{ если } a \in \mathbf{L}_t(S);$$
$$\perp_к \cdot> a, \text{ если } a \in \mathbf{R}_t(S).$$

Маючи побудовану множину $\mathbf{L}_t(U)$ и $\mathbf{R}_t(U)$, заповнення матриці операторного передування для КС- граматики G(VT.VN,P,S) можна виконати по наступному алгоритму:

1. Беремо перший символ з множини термінальних символів граматики VT: $a_i \in$VT. i=1. Вважатимемо цей символ поточним термінальним символом.

2. У всій множині правил P шукаємо правила $C \to xa_ib_jy$ или $C \to xa_iUb_jy$, де a_i поточний термінальний символ, b_j— довільний термінальний символ $(b_j \in \mathbf{VT})$,U і C —довільні нетермінальні символи($U, C \in \mathbf{VN}$). а x та y — довільні ланцюжки символів, можливо порожні ($x, y \in \mathbf{V}^*$). Фактично проводиться пошук таких правил, в яких в правій частині символи a, і b,

стоять поряд або ж між ними, та є не більш за один нетермінальний символ (причому символ обов'язково стоїть зліва від b_j.

3. Для всіх символів b_j знайдених на кроці 2, виконуємо наступне: ставимо знак «=» («основа») у клітинки матриці операторного передування на перетині рядка, поміченого символом a_i і стовпця, поміченого символом b_j

4.В всій множині правил P шукаємо правила вигляду $C \to xa_iU_jy$, де a, — поточний термінальний символ, U_j і C— довільні нетермінальні символи (U_j),а x та y- довільні ланцюжки символів, можливо порожні($x, y \in V^*$). Фактично шукаємо правила, в яких в правій частині символ стоїть зліва від нетермінального символу Uj/

5. Для всіх символів U знайдених на кроці 4, беремо множину символів Lt(Uj). Для всіх термінальних символів AAA що входять в цю множину, виконуємо наступне: ставимо знак «<.» («передує») в клітинки матриці операторного передування на перетині рядка, поміченого символом AAA і стовпця, поміченого символом AA.
6. У всій множині правил P шукаємо правила виду $C \to xU_ja_iy$, де a_i— поточний термінальний символ, U_J і C - довільні нетермінальні символи ($U_j, C \in VN$), а x та y— довільні ланцюжки символів, можливо порожні ($x, y \in V^*$). Фактично шукаємо правила, в яких правої частини символу a о стоїть праворуч від нетермінального символу U_J .
7. Для всіх символів U_J знайдених на кроці 6 беремо множину символів $R_t(U_J)$. Для всіх термінальних символів c_k що входять в цю множину, виконуємо наступне: ставимо знак «.>» («слідує») в клітинці матриці операторного передування на перетині рядка, поміченого символом c_K, і стовпця, поміченого символом a_i.
8. Якщо розглянуті всі термінальні символи з множини VT, то переходимо до кроку 9, інакше - беремо черговий символ $a_i \in VT$ з множини VT, i=i+1, робимо його поточним термінальним символом і повертаємося до кроку 2.
9. Беремо множину $L_t(S)$ для цільового символу граматики S. Для всіх

термінальних символів $c_к$ що входять в цю множину, виконуємо наступне: ставимо знак «<.» («передує») в клітинці матриці операторного передування на перетині рядка, поміченого символом ┴ₙ («початок рядку»), та стовпця, поміченого символом $c_к$.

10. Беремо множину R(S) для цільового символу граматики S. Для всіх термінальних символів $c_к$ що входять в цю множину, виконуємо наступне: ставимо знак «.>» («слідує») в клітинці матриці операторного передування на перетині рядка, поміченого символом і стовпця, поміченого символом кінця рядка. Побудова матриці закінчена.

Якщо на всіх кроках алгоритму побудови матриці операторного передування не виникло суперечностей, коли в одну і ту ж клітинку матриці треба записати два або три різні символи передування, то матриця побудована правильно (у кожній клітці такої матриці присутній одні з символів передування — «.>, «<.» або «=» — або ж клітка порожня). Якщо на якомусь кроці виникла суперечність, значить, початкова КС- граматика не є граматикою операторного передування. В цьому випадку можна спробувати перетворити граматику так, що вона стане задовольняти вимогам операторного передування (що не завжди можливо), або необхідно використовувати інший тип розпізнавача.

Детальніше робота з граматиками передування і іншими типами розпізнавачів описана в [1-4,7).

Алгоритм «зсув-згортка» для граматик операторного передування

Алгоритм «зсув-згортка» для граматики операторного передування виконується МП- автоматом з одним станом. Для моделювання його роботи необхідний вхідний ланцюжок символів та стек символів, в якому автомат може звертатися не тільки до самого верхнього символу, але і до деякого ланцюжка символів на вершині стека.

Цей алгоритм для заданої КС- граматики $G(VT,VN,P,S)$ за наявності побудованої матриці передування можна описати таким чином:

1. Помістити у верхівку стека символ початку рядка, зчитуючу голівку МП-автомата помістити в початок вхідного ланцюжка (поточним вхідним символом стає перший символ вхідного ланцюжка). У кінець вхідного ланцюжка треба дописати символ кінця рядка.
2. У стеку шукається самий верхній термінальний символ (якщо на вершині стека лежать нетермінальні символи, вони ігноруються і береться перший термінальний символ, що знаходиться під ними), при цьому сам символ s залишається в стеку. З одного ланцюжка береться поточний символ (справа зчитуючою головки МП- автомата).
3. Якщо символ s_j — це символ початку рядка, а символ a_i — символ кінця рядка, то алгоритм завершений, вхідний ланцюжок символів розібраний.
4. У матриці передування шукається клітинка на перетині рядка, поміченого символом a_i і стовпця, поміченого символом a_i виконується порівняння поточного вхідного символу і термінального символу на верхівці стека).
5. Якщо клітинка, знайдена на кроці 3, порожня, то значить, вхідний рядок символів не приймається МП- автоматом, алгоритм переривається і видасть повідомлення про помилку.
6. Якщо клітинка, знайдена з кроці 3, містить символ «=.» («складає основу») або «<.» («передує»), то необхідно виконати перенесення (зрушення). При виконанні перенесення поточний вхідний символ поміщається на верхівку стека, зчитуюча головка МП- автомату у вхідному ланцюжку символів зрушується на одну позицію вправо (після чого поточним вхідним символом стає наступний символ a_{i+1}, i:=i+1). Після цього треба повернутися до кроку 2.
7. Якщо клітинка, знайдена на кроці 3, містить символ «=.» («слідує»), то необхідно провести згортку. Для виконання згортки із стека вибираються всі термінальні символи, зв'язані відношенням «.>»(«складає основу»), починаючи від вершини стека, а також всі нетермінальні символи, лежачі в стеку поряд з ними. Ці символи виймаються із стека і збираються в

ланцюжок (якщо в стеку немає символів, зв'язаних відношенням «=», то з нього виймається один самий верхній термінальний символ та лежачі поряд з ним нетермінальні символи).

8. У всій множині правил P граматики G(VT.VN,P,S) шукається правило, у якого права частина співпадає з ланцюжком (за умовами граматик передування всі праві частини правил повинні бути різні, тому може бути знайдено або одне таке правило, або жодного). Якщо правило знайдене. то в стек надходить нетермінальний символ з лівої частини правила, інакше, якщо правило не знайдене, це означає, що вхідний рядок символів не приймається МП- автоматом, алгоритм переривається та видасть повідомлення про помилку. Слід зазначити, що при виконанні згортки зчитувана головка автомата не зрушується і поточний вхідний символ залишається незмінним. Після виконання згортки необхідно повернутися до кроку 2.

Після завершення алгоритму рішення про ухвалення ланцюжка залежить від вмісту стека. Автомат приймає ланцюжок, якщо в результаті завершення алгоритму він знаходиться в стані, коли в стеку знаходяться початковий символ граматики S . Виконання алгоритму може бути перерване, якщо на одному з його кроків виникне помилка. Тоді вхідний ланцюжок не приймається.

Алгоритм «зсув-згортка» для граматики операторного передування ігнорує нетермінальні символи. Тому має сенс перетворити початкову граматику так, щоб залишити в ній тільки один нетермінальний символ. Це перетворення полягає в тому, що всі нетермінальні символи в правилах граматики замінюються на один нетермінальний символ (найчастіше — цільовий символ граматики).

Побудована в результаті такого перетворення граматика називається остовною граматикою, а саме перетворення - остовним перетворенням [1,7|.

Остовне перетворення не веде до створення еквівалентної граматики і виконується тільки для спрощення роботи алгоритму (який при виборі правил все одно ігнорує нетермінальні символи) після побудови матриці передування.

Одержана в результаті остовного перетворення граматика може не бути однозначною, але всі необхідні дані про порядок застосування правил містяться в матриці передування і розпізнавач залишається детермінованим. Тому остовне перетворення може виконуватися без втрат інформації тільки після побудови матриці передування. При цьому також необхідно стежити, щоб в граматиці не виникло неоднозначностей із-за однакових правих частин правил, які можуть з'явитися в остовній граматиці.

Висновок, одержаний при розкладі на основі остовної граматики, називають результатам остовного розкладу, або остовним висновком. За наслідками остовного розкладу можна побудувати відповідний йому вивід на основі правил початкової граматики. Проте це завдання не представляє практичного інтересу, оскільки остовний висновок відрізняється від висновку на основі початкової граматики тільки тим, що в нім відсутні кроки, пов'язані з застосуванням кільцевих правил, і не враховуються типи нетермінальних символів.

Для компіляторів розпізнавання ланцюжків вхідної мови полягає не в знаходженні того чи іншого висновку, а в виявленні основних синтаксичних конструкцій початкової програми з метою побудови на їх основі ланцюжків мови результуючої програми. У цьому сенсі типи нетермінальних символів і цінні правила не несуть ніякої корисної інформації, а навпроти, тільки ускладнюють обробку ланцюжка висновку. Тому для реального компілятора знаходження остовного висновку є більш корисним ніж знаходження висновку на основі початкової граматики. Знайдений остовний результат подальших перетворень вже не потребує.

У загальному вигляді послідовність побудови розпізнавача для КС-граматики операторного передування G(VT,VN,P,S) можна описати слідуючим чином:

1. На основі множини правил граматики P побудувати множену крайніх лівих і крайніх правих символів для всіх нетермінальних символів граматики.

2. На основі множини правил граматики P і побудованих на кроці 1 множини побудувати множини крайніх лівих і крайніх правих термінальних символів для всіх нетермінальних символів граматики.

3. На основі побудованих на кроці 2 множини для всіх термінальних символів граматики заповнюється матриця операторного передування.

4. Початкова граматика G(VT,VN,P,S) переходить в остовну граматику з одним нетермінальним символом.

5. На основі побудованої матриці передування та остовної граматики будується розпізнавач на базі, алгоритму «сзсув-згортка», для граматик операторного передування.

Важливо, що алгоритм розпізнавача може бути реалізований незалежно від матриці передування і правил початкової граматики. Тоді, міняючи матрицю і правила, один і той же алгоритм можна використовувати для розпізнавання вхідних ланцюжків будь-якої граматики операторного передування.

Далі в прикладі виконання роботи проілюстрований саме такий підхід до побудови розпізнавача.

Порядок виконання лабораторної роботи:

1. Одержати варіант завдання у викладача.
2. Побудувати множина крайніх лівих і крайніх правих символів, множина крайніх правих і крайніх лівих термінальних символів і матрицю операторного передування для заданої граматики (якщо для побудови синтаксичного розпізнавача передбачається використовувати інший механізм, відмінний від граматик операторного передування, то форму його треба заздалегідь погоджувати з викладачем).
3. Виконати розбір простого прикладу вручну по правилах заданої граматики, переконатися, що розбір виконується коректно.
4. Підготувати і захистити звіт.
5. Написати і відлагодити програму на ПК.
6. Здати працюючу програму викладачеві.

Варіанти завдань.

Варіанти початкових граматик

Далі приведені варіанти граматик. У всіх варіантах символ 5 є початковим символом граматики S,T,F і E означають нетермінальні символи.

Термінальні символи виділені жирним шрифтом. Замість символу **a** повинні підставлятися лексеми.

1. *S* → **a :=** *F*;
   F → *F*+*T* | *T*
   T → *T*·*E* | *T*/*E* | *E*
   E → (*F*) | –(*F*) | *a*
2. *S* → **a :=** *F*;
   F → *F* **or** *T* | *F* **xor** *T* | *T*
   T → *T* **and** *E* | *E*
   E → (*F*) | **not** (*F*) | **a**
3. *S* → *F*;
   F → **if** *E* **then** *T* **else** *F* | **if** *E* **then** *F* | **a := a**
   T → **if** *E* **then** *T* **else** *T* | **a := a**
   E → **a<a** | **a>a** | **a=a**
4. *S* → *F*;
   F → **for (T) do F** | **a := a**

Початкові граматики і типи допустимих лексем

Нижче в табл. 3.1 приведені номери завдань. Для кожного завдання вказана відповідна йому граматика і типи допустимих лексем.

Таблиця 3.1. Номери завдань для виконання лабораторної роботи

№	№ вар. граматики	Допустимі лексеми вхідної мови
1	1	Ідентифікатори, десяткові числа з плаваючою крапкою
2	2	Ідентифікатори, константи true та false
3	3	Ідентифікатори, десяткові числа з плаваючою крапкою
4	4	Ідентифікатори, десяткові числа з плаваючою крапкою
5	1	Ідентифікатори, римські числа
6	2	Ідентифікатори, константи 0 та 1
7	3	Ідентифікатори, римські числа
8	4	Ідентифікатори, римські числа
9	1	Ідентифікатори, шістнадцяткові числа
10	2	Ідентифікатори, шістнадцяткові числа
11	3	Ідентифікатори, шістнадцяткові числа
12	4	Ідентифікатори, шістнадцяткові числа
13	1	Ідентифікатори, символьні константи (в одинарних лапках)
14	2	Ідентифікатори, символьні константи 'T' та 'F'
15	3	Ідентифікатори, символьні константи (в подвійних лапках)
16	4	Ідентифікатори, символьні константи (в подвійних лапках)

- *ПРИМІТКА*
- *Римськими числами вважати послідовності великих латинських букв X. V і I.*
- *Шістнадцятковими числами вважати послідовність цифр і символів «a», «b» … «f», які починаються з цифри (наприклад 89, 45ac9, 0aCc4).*
- *Для виконання роботи рекомендується використовувати лексичний аналізатор, побудований в ході виконання лабораторної роботи № 2*

Звіт повинен містити наступні розділи:

1. Завдання по лабораторній роботі.
2. Короткий виклад мети роботи.
3. Запис заданої граматики вхідної мови у формі Бекуса—Наура (якщо для побудови синтаксичного розпізнавача використовується механізм який потребує перетворення початкової граматики вхідної мови, то ці зміни і одержана в результаті їх граматика повинні бути відбиті в звіті).
4. Множина крайніх правих і крайніх лівих символів з вказівкою кроків побудови.
5. Множини крайніх правих та крайніх лівих термінальних символів.
6. Заповнену матрицю передування для граматики (якщо для побудови синтаксичного розпізнавача використовується інший механізм, відмінний

від граматик операторного передування, то форму його відображення в звіті треба погоджувати з викладачем).

7. Приклад виконання розбору простої пропозиції вхідної мови.
8. Текст програми (оформляється після виконання програми на ПК).

Контрольні запитання:

1) Яку роль виконує синтаксичний аналіз в процесі компіляції?
2) Які проблеми виникають при побудові синтаксичного аналізатора і як вони можуть бути вирішені?
3) Які типи граматик існують? Що таке КС- граматики? Розкажіть про їх використання в компіляторі.
4) Які типи розпізнавачів для КС- граматики існують? Розкажіть про недоліки та переваги різних типів розпізнавачів.
5) Поясніть правила побудови дерева виведення граматики.
6) Що таке граматика простого передування?
7) Як обчислюються відносини передування для граматик простого передування ?
8) Що таке граматика операторного передування ?
9) Як обчислюються відношення для граматик операторного передування ?
10) Розкажіть про завдання розбору. Що таке розпізнавач мови?
11) Розкажіть про загальні принципи роботи розпізнавача мови.
12) Що таке перенесення, згортка? Для чого необхідний алгоритм «перенесення-згортка»?
13) Розкажіть, як працює алгоритм «перенесення-згортка» в загальному випадку (з поверненнями).
14) Як працює алгоритм «перенесення -згортка» без повернень (поясніть на своєму прикладі)?

Лабораторна робота №4

Тема: ГЕНЕРАЦІЯ І ОПТИМІЗАЦІЯ ОБ'ЄКТНОГО КОДУ

Мета: Вивчення основних принципів генерації компілятором його коду, ознайомлення з методами оптимізації результуючого об'єктного кола для лінійної ділянки програми з допомогою згортки і виключення зайвих операцій

Завдання:

Написати програму, що на підставі дерева синтаксичного розбору породжує об'єктний код і виконує потім його оптимізацію методом згортки об'єктного коду й методом виключення зайвих операцій. Як вихідне дерево синтаксичного розбору рекомендується взяти дерево, що породжує програма, побудована за завданням лабораторної роботи № 3.

Програму рекомендується побудувати із трьох основних частин: перша частина - породження дерева синтаксичного розбору (за результатами лабораторної роботи №3), друга частина - реалізація алгоритму породження об'єктного коду по дереву розбору й третя частина - оптимізація породженого об'єктного коду (якщо в результуючій програмі присутні лінійні ділянки коду). Результатом роботи повинна бути побудована на основі заданої пропозиції граматики програма на об'єктній мові або побудованій послідовності тріад (за узгодженням з викладачем вибирається форма подання кінцевого результату).

Як об'єктний код пропонується обрати мову асемблера для процесорів типу Intel 80x86 у реальному режимі (можливий вибір іншої об'єктної мови за узгодженням з викладачем). Всі ідентифікатори, що зустрічаються у вихідній програмі, уважати простими скалярними змінними, не потребуючими виконання перетворення типів. Обмеження на довжину ідентифікаторів і констант відповідають вимогам лабораторної роботи № 3.

Теоретичні відомості:

Загальні принципи генерації коду

Генерація об'єктного коду — це переклад компілятором внутрішнього представлення початкової програми в ланцюжок символів вихідної мови. Оскільки вихідною мовою компілятора (на відміну від транслятора) може бути тільки або мова «асемблер», або мова машинних кодів, то генерація коду

породжує результуючу об'єктну програму на мові асемблера або безпосередньо на машинній мові (у машинних кодах).

Генерація об'єктного коду виконується після того, як виконані лексичний і синтаксичний аналіз програми і всі необхідні дії по підготовці до генерації коду: перевірені семантичні угоди вхідної мови (семамтичний аналіз), виконана ідентифікація імен змінних і функцій, розподілений адресний простір під функції і змінні.

У даній лабораторній роботі використовується гранично проста вхідна мова, тому немає необхідності виконувати всі перераховані перетворення. Вважатимемо, що всі вони вже виконані. Детальніше всі ці фази компіляції описані в (1-4,7), тут буде йти мова про самі примітивні прийоми семантичного аналізу, які будуть проілюстровані на прикладі виконання лабораторної роботи.

Внутрішнє представлення програми може мати будь-яку структуру залежно від реалізації компілятора, в той час як результуюча програма завжди є лінійною послідовністю команд. Тому генерація об'єктного кола (об'єктної програми) в любому випадку повинна виконувати дії, пов'язані з перетворенням складних синтаксичних структур в лінійні ланцюжки.

Генерацію коду можна вважати функцією, визначеною на синтаксичному дереві, побудованому в результаті синтаксичного аналізу, і на інформації, що міститься в таблиці ідентифікаторів. Характер відображення вхідної програми в послідовність команд, виконуваного генерацією, залежить від вхідної мови, архітектура цільової обчислювальної системи, на яку орієнтована результуюча програма, а також від якості бажаного об'єктного кола.

В ідеалі компілятор повинен виконати синтаксичний аналіз всієї вхідної програми, потім провести її семантичний аналіз, після чого приступати до підготовки генерації і безпосередньо генерації коду. Проте така схема роботи компілятора практично майже ніколи не застосовується. Річ у тому, що в загальному випадку жоден семантичний аналізатор і жоден компілятор не здатні проаналізувати і оцінити сенс всієї початкової програми в цілому. Формальні методи аналізу семантики застосовні тільки до дуже незначної частини можливих початкових програм. Тому у компілятора немає практичної можливості породжувати еквівалентну результуючу програму на основі всієї

початкової програми.

Як правило, компілятор виконує генерацію результуючого коду поетапно, на основі закінчених синтаксичних конструкцій вхідної програми. Компілятор виділяє закінчену синтаксичну конструкцію з тексту початкової програми, породжує для неї фрагмент результуючого коду і поміщає його в текст результуючої програми. Потім він переходить до наступної синтаксичної конструкції. Так продовжується доти, поки не буде розібрана вся початкова програма. В якості аналізованих закінчених синтаксичних конструкцій виступають блоки операторів, опису процедур і функцій. Їх конкретний склад залежить від вхідної мови і реалізації компілятора.

Сенс (семантику) кожній такій синтаксичній конструкції вхідної мови можна визначити, виходячи з її типу, а тип визначається синтаксичним аналізатором на основі граматики вхідної мови. Прикладами типів синтаксичних конструкцій можуть служити оператори циклу, умовні оператори, оператори вибору. Одні і ті ж типи синтаксичних конструкцій характерні для різних мов програмування, при цьому вони розрізняються синтаксисом (який задасться граматикою мови), але мають схожий сенс (який визначається семантикою). Залежно від типу синтаксичної конструкції виконується генерація кола результуючої програми, відповідного даній синтаксичній конструкції. Для семантично схожих конструкції різних вхідних мов програмування може породжуватися типовий результуючий код.

Синтаксично керований переклад

Щоб компілятор міг побудувати код результуючої програми для синтаксичної конструкції вхідної мови, часто використовується метод, званий синтаксично керованим перекладом, — СУ-переклодом.

Ідея СУ-перекладу заснована на тому. що синтаксис та семантика мови взаємозв'язані. Це означає, що сенс пропозиції мови залежить від синтаксичної структури цієї пропозиції. Теорія синтаксично керованого перекладу була запропонована американським лінгвістом Ноамом Хомським. Вона справедлива як для формальних мов, так і для мов природного спілкування: наприклад, сенс пропозиції російської мови залежить від вхідних в нього частин мови (підмета, присудка, доповнень і ін.) і від взаємозв’язку між ними. Проте природні мови

допускають неоднозначності в граматиках — звідси відбуваються різні двозначні фрази, значення яких чоловік зазвичай розуміє з того контексту, в якому ці фрази зустрічаються (і те він не завжди може це зробити). У мовах програмування неоднозначності в граматиках виключені, тому будь-яка пропозиція мови має чітко певну структуру і однозначний сенс, безпосередньо пов'язаний з цією структурою. Вхідна мова компілятора має нескінченну множену допустимих пропозицій, тому неможливо задати сенс кожної пропозиції. Але всі вхідні пропозиції будуються на основі кінцевої множини правил граматики, які завжди можна знайти. Оскільки цих правил кінцеве число, то для кожного правила можна визначити його семантику (значення).

Але абсолютне те ж саме можна стверджувати і для вихідної мови компілятора. Вихідна мова містить нескінченну множину допустимих пропозицій, але всі вони будуються на основі кінцевої множини відомих правил, кожне з яких має певну семантику (сенс). Якщо по відношенню до початкової програми компілятор виступає в ролі розпізнавача, то для результуючої програми він є генератором додатків вихідної мови. Завдання полягає в тому, щоб знайти порядок правил вихідної мови, по яких необхідно виконати генерацію.

Грубо кажучи, ідея СУ-перекладу полягає в тому, що кожному правилу вхідної мови компілятора зіставляється одне чи декілька(або жодного) правил вихідної мови відповідно до семантики вхідних і вихідних правил. Тобто при зіставленні треба вибирати правила вихідної мови, які несуть той же сенс, що і правила вхідної мови.

СУ-переклад - це основний метод породження коду результуючої програми на підставі результатів синтаксичного аналізу. Для зручності розуміння суті методу можна вважати, що результат синтаксичного аналізу приставлений у вигляді дерева синтаксичного аналізу, хоча в реальних компіляторах це не завжди так. Суть принципу СУ-перекладу полягає в наступному: з кожною вершиною дерева синтаксичного розкладу N зв'язується ланцюжок деякого проміжного коду C(N). Код для вершини N будується шляхом зчеплення (конкатенації) у фіксованому порядку послідовності коду C(N) і послідовностей кодів, пов'язаних зі всіма вершинами, прямими нащадками N. У

свою чергу, для побудови послідовностей коду прямих нащадків вершини N потрібно буде знайти послідовності коду для їх нащадків — нащадків другого рівня вершини N — і т.д. Процес перекладу йде, таким чином, від низу до верху в строго встановленому порядку, визначуваному структурою дерева. Для того, щоб побудувати СУ-переклад в заданому дереву синтаксичного розбору, необхідно знайти послідовність коду для кореня дерева. Тому для кожної вершини дерева породжуваний ланцюжок коду треба вибирати так, щоб код. приписуваний кореню дерева, виявився шуканим кодом для всього оператора, представленого цим деревом. У загальному випадку необхідно мати одноманітну інтерпретацію коду C(N), яка б зустрічалася в всіх ситуаціях, де присутня вершина N. В принципі, це завдання може виявитися нетривіальним, оскільки вимагає оцінки сенсу (семантики) кожної вершини дерева. При застосуванні СУ-переклада задача оцінки смислового навантаження для кожної вершини дерева вирішується розробником компілятора.

Можлива модель компілятора, в якій синтаксичний аналіз початкової програми і генерація коду результуючої програми об'єднані в одну фазу. Таку модель можна представити у вигляді компілятора, у якого операції генерації коду суміщені з операціями виконання синтаксичного розбору. Для опису компіляторів такого типу часто використовується термін СУ-компіляція (синтаксично керована компіляція).

Схему СУ-компіляції можна реалізувати не для всякої вхідної мови програмування. Якщо принцип СУ-перекладу застосуємо до всіх вхідних КС-мов, то застосувати СУ-компіляцію виявляється не завжди можливим (1. 2. 7). В процесі СУ-перекладу і СУ-компіляції не тільки виробляються ланцюжки тексту вихідної мови, але і здійснюються деякі додаткові дії, що виконуються самим компілятором. У загальному випадку схеми СУ-перекладу можуть, передбачати виконання наступних дій:

• приміщення у вихідний потік даних машинних кодів або команд асемблера, що є результатом роботи (вихід) компілятора;

• видача користувачеві повідомлень про виявлені помилки і попередження (які повинні поміщатися у вихідний потік, відмінний від потоку, використовуваного для команд результуючої програми);

• породження і виконання команд, вказуючих, що деякі дії повинні бути проведені самим компілятором (наприклад операції, виконувані над даними, розміщеними в таблиці ідентифікаторів).

Нижче розглянуті деякі основні технічні питання, що дозволяють реалізувати схеми СУ-перекладу для даної лабораторної роботи. Детальніше з механізмами СУ-перекладу і СУ-компіляції можна ознайомитися в (1, 2. 7).

Способи внутрішнього представлення програм

Результатом роботи синтаксичного аналізатора на основі КС- граматики вхідної мови є послідовність правил граматики, застосованих для побудови вхідного ланцюжка. По знайденій послідовності, знаючи тип розпізнавача, можна побудувати ланцюжок висновку або дерево висновку. В цьому випадку дерево висновку виступає як дерево синтаксичного розбору і є результатом роботи синтаксичного аналізатора в компіляторі. Проте ні ланцюжок висновку, ні дерево синтаксичного розбору не є метою роботи компілятора. Для повного уявленні про структуру розібраної синтаксичної конструкції вхідної мови в принципі достатньо знати послідовність номерів правил граматики, застосованих для її побудови. Проте форма уявлення цієї інформації може бути різною в залежності від реалізації самого компілятора, так і від фази компіляції. Ця форма називається внутрішнім **представленням програми** (іноді використовуються також терміни проміжне уявлення, або проміжна програма).

Всі внутрішні представлення програми зазвичай містять в собі два принципово різних елемента - оператори і операнди. Відмінності між формами внутрішнього уявлення полягають лише в тому, як оператори та операнди з'єднуються між собою. Також оператори і операнди повинні відрізнятися один від одного, якщо вони зустрічаються у будь-якому порядку. За розрізнення операндів і операторів, як вже було сказано вище, відповідає розробник компілятора, який керується семантикою вхідної мови. Відомі наступні форми внутрішнього представлення програм:

• структури зв'язних списків, що представляють синтаксичні дерева;
• багатоадресний код з явно іменованим результатом (тетради);
• багатоадресний код з неявно іменованим результатом (тріади);
• зворотний (постфіксна) польський запис операції;

• асемблерний код або машинні команди.

У кожному конкретному компіляторі може використовуватися одна з цих форм, вибрана розробниками. Але найчастіше компілятор не обмежується використанням тільки однієї форми для внутрішнього представлення програми.

На різних фазах компіляції можуть використовуватися різні форми, які у міру виконання проходів компілятора перетворяться одна в іншу. Деякі компілятори, які оптимізують результуючий код, генерують об'єктний код по мірі розкладу початкової програми. В цьому випадку застосовується схема СУ - компіляції, коли фази синтаксичного розбору, семантичного аналізу, підготовки і генерації об'єктного коду суміщені в одному проході компілятора. Тоді внутрішнє представлення програми існує тільки умовно у вигляді послідовності кроків алгоритму розбору.

Алгоритми, запропоновані для виконання даної лабораторної роботи, побудовані на основі використання форми внутрішнього представлення програми у вигляді тріад. Тому далі буде розглянута саме ця форма внутрішнього представлення програми. З рештою форм можна детальніше познайомитися в (1-3,7).

Багатоадресний код з неявно іменованим результатом (тріади)

Тріади є записом операції у формі з трьох складових: операція і два операнди. Наприклад, в строковому записі тріади можуть мати вигляд: <операція>(<операнд1>.<операнд2>). Спільністю тріад є те, що один або обидва операнди можуть бути посиланнями на іншу тріаду в тому випадку, якщо операнд даної тріади виступає результат виконання іншої тріади. Тому тріади при записі послідовно нумерують для зручності вказівки посилань одних тріад на інші (у реалізації компілятора як посилання можна використовувати не номери тріад, а безпосередньо посилання у вигляді покажчиків — тоді при зміні нумерації і порядку проходження тріад міняти посилання не потрібно).

Наприклад, вираз A:=B+C+D-B-10, записаний у вигляді тріад, матиме вигляд:

```
1:  * (  B.   C  )
2:  + ( ^1.   D  )
3:  * (  B.  10  )
4:  - ( ^2.  ^3  )
5:  := (  A.  ^4  )
```

Тут операції позначені відповідними знаками (при цьому привласнення також є операцією), а знак ^ означає посилання операнда однієї тріади на результат іншої.

Тріади є лінійною послідовністю команд. При обчисленні виразу, записаного у формі тріад, вони обчислюються одна за одною послідовно. Кожна тріада в послідовності обчислюється так: операція, задана тріадою, виконується над операндами, а якщо один з операндів (або обох операндів) виступає посилання на іншу тріаду, то береться результат обчислення тієї тріади. Результат обчислення тріади потрібно зберігати в тимчасовій пам'яті, оскільки він може зажадатися подальшими тріадами. Якщо якийсь з операндів в тріаді відсутній (наприклад, якщо тріада є унарною операцією), то він може бути опущений або замінений порожнім операндом (залежно від прийнятої форми запису і її реалізації). Порядок обчислення тріад може бути змінений, але тільки якщо допустити наявність тріад, що цілеспрямовано змінюють цей порядок (наприклад, тріади, що викликають безумовний, перехід на іншу тріаду із заданим номером або перехід на декілька кроків вперед або назад за якоїсь умови). Тріади є лінійною послідовністю, а тому для них нескладно написати тривіальний алгоритм, який перетворюватиме послідовність тріад в послідовність команд результуючої програми (або послідовність команд асемблера). В цьому їх перевага перед синтаксичними деревами. Проте для тріад потрібний також і алгоритм, що відповідає за розподіл пам'яті, необхідної для зберігання проміжних результатів обчислення, оскільки тимчасові змінні для цієї мети не використовуються (у цьому відмінність тріад від тетрад).

Тріади не залежать від архітектури обчислювальної системи, на яку орієнтована результуюча програма. Тому вони є машинно-незалежною формою внутрішнього представлення програми.

Тріади володіють наступними перевагами:

• лінійна послідовність операцій, на відміну від синтаксичного дерева, і тому простіше перетворюються в результуючий код;

• займають менше пам'яті, ніж тетради, дають більше можливостей по оптимізації програми, ніж зворотній польський запис;

• явно відображають взаємозв'язок операцій між собою, що робить їх застосування зручним, особливо при оптимізації внутрішнього представлення програми;

• проміжні результати обчислення тріад можуть зберігатися в регістрах процесора, що зручно при розподілі регістрів і виконанні машинно-залежної оптимізації;

• за формою представлення знаходяться ближчим до двохадресних машинних команд, ніж інші форми внутрішнього представлення програм, а саме ці команди понад усе поширені в наборах команд більшості сучасних комп'ютерів.

Необхідність створення алгоритму, що відповідає за розподіл пам'яті для зберігання проміжних результатів, є головним недоліком тріад. Але при грамотному розподілі пам'яті і регістрів процесора цей недолік може бути обернений на користь розробниками компілятора.

Схеми СУ-перекладу

Раніше був описаний принцип СУ-перекладу, що дозволяє одержати лінійну послідовність команд результуючої програми або внутрішнього представлення програми в компіляторі на основі результатів синтаксичного аналізу. Тепер побудуємо варіант алгоритму генерації коду, який одержує на вході дерево синтаксичного розбору і створить по ньому послідовність тріад (далі — просто «тріади») для лінійної ділянки результуючої програми. Розглянемо приклади схем СУ-перекладу для бінарних арифметичних операцій. Ці схеми достатньо прості, і на їх основі можна проілюструвати, як виконується СУ-переклад в компіляторі при генерації коду. Для побудови тріад по синтаксичному дереву може використовуватися проста рекурсивна процедура обходу дерева. Можна використовувати і інші методи обходу дерева — важливо, щоб дотримувався принцип, згідно якому операції які знаходяться нижче в дереві завжди виконуються перед вище розміщеними операціями (порядок

виконання операцій одного рівня не важливий, він не впливає на результат і залежить від порядку обходу вершин дерева).

Процедура генерації тріад по синтаксичному дереву перш за все повинна визначити тип вузла дерева. Для бінарних арифметичних операцій кожен вузол дерева має три нижче розміщених вершини (ліва вершина - перший операнд, середня вершина - операція і права вершина - другий операнд). При цьому тип вузла дерева відповідає типу операції, символом якої помічена середня з нижче розташована вершина. Після визначення типу вузла процедура будує тріади для вузла дерева відповідно до типу операції.

Фактично процедура генерації тріад повинна для кожного вузла дерева виконати конкатенацію тріади, пов'язаної з поточним вузлом, та ланцюгом тріади, пов'язаних з нижче розташованими вузлами. Конкатенація ланцюжків тріад повинна виконуватися так, щоб тріади, пов'язані з нижче розташованими вузлами, виконувалися до виконання операції, пов'язаної з поточним вузлом. Причому для арифметичних операцій важливо, щоб тріади, пов'язані з першим операндом, виконувалися раніше, ніж тріади, пов'язані з другим операндом (оскільки всі арифметичні операції за відсутності дужок і пріоритетів виконуються в порядку зліва направо). При чьому можливі чотири ситуації:

1. ліва і права вершини вказують на безпосередній операнд (це можна визначити, якщо у кожної з них є тільки один нижче розташований вузол, помічений символом якоїсь лексеми, - константи або ідентифікатора);

2. ліва вершина є безпосереднім операндом, а права вказує на іншу операцію;

3. ліва вершина вказує на іншу операцію, а права є безпосереднім операндом;

4. обидві, вершини вказують на іншу операцію.

Вважаємо, що на вхід процедури породження тріад по синтаксичному дереву подається список, в який потрібно додавати тріади, і посилання на вузол дерева, який треба обробити. Тоді процедура породження тріад для вузла синтаксичного дерева, пов'язаного з бінарною арифметичною операцією, може виконуватися по наступному алгоритму:

1. Перевіряється тип лівої вершини вузла. Якщо вона — простий операнд,

запам'ятовується ім'я першого операнда, інакше для цієї вершини рекурсивно викликається процедура породження тріад, побудовані нею тріади додаються в кінець загального списку запамятовується номер останньої тріади з цього списку як перший операнд.

2. Перевіряється тип правої вершини вузла. Якщо вона — простий операнд, запам'ятовується ім'я другого операнда, інакше для цієї вершини рекурсивно викликається процедура породження тріад, побудовані нею тріади додаються в конець загального списку і запам'ятовується номер останньої тріади як другий операнд.

3. Відповідно до типу середньої вершини в кінець загального списку додається тріада, відповідна арифметичній операції. Її першим операндом стає операнд, що був запам'ятований на кроці 1, а другим операндом — операнд, що був запам'ятований на кроці 2.

4. Процедура закінчена.

Процедури такого роду повинен створювати розробник компілятора, оскільки тільки він може співставити по сенсу вузли синтаксичного дерева і відповідні їм послідовності тріад. Для різних типів вузлів синтаксичного дерева можуть бути побудовані різні варіанти процедур, які викликатимуть один одного залежно від прийнятого порядку обходу синтаксичного дерева (у описаному вище варіанті — рекурсивно).

У розглянутому прикладі при породженні коду навмисно не були прийняті до уваги багато питань, що виникають при побудові реальних компіляторів. Це було зроблено для спрощення прикладу. Наприклад, фрагменти коду, відповідні різним вузлам дерева, приймають до уваги тип операції, але ніяк не враховують тип операндів. Всі ці вимоги ведуть до того, що в реальному компіляторі при генерації коду треба приймати до уваги дуже багато особливостей, залежні від семантики вхідної мови і від використовуваної форми внутрішнього представлення програми. У даній лабораторній роботі ці питання не розглядаються.

Крім того, у разі арифметичних операцій код, що породжується для вузлів синтаксичного дерева, залежить тільки від типу операції, тобто тільки від поточного вузла дерева. Такі схеми можна побудувати для багатьох операцій,

але не для всіх. Іноді код, що породжується для вузла дерева, може залежати від типу вищестоящого вузла: наприклад, код, що породжується для операторів типу Break і Continue (які є в мовах С, С++ і Objecl Pascal), залежить від того, всередині якого циклу вони знаходяться. Тоді при рекурсивній побудові коду по дереву вище розташований вузол, викликаючи функцію для нижче розташованого вузла, повинен передати їй необхідні параметри. Але код, що породжується для вище розташованого вузла, ніколи не повинен залежати від нижче розташованого вузла, інакше принцип СУ-перекладу непридатний. Далі в прикладі виконання роботи даються варіанти схем СУ-перекладу для різних конструкцій вхідної мови, які можуть служити хорошою ілюстрацією механізму застосування цього методу.

Загальні принципи оптимізації коду

Як вже говорилося, в переважній більшості випадків генерація коду виконується компілятором не для всієї початкової програми в цілому а послідовно для окремих її конструкцій. Для побудови результуючого кола різних синтаксичних конструкцій вхідної мови використовується метод Су-перекладу. Він поєднує ланцюжки побудованого коду за структурою дерева без обліку їхніх взаємозв'язків.

Побудований у такий спосіб код результуючої програми може містити зайві команди й дані. Це знижує ефективність виконання результуючої програми. У принципі, компілятор може завершити на цьому генерацію коду, оскільки результуюча програма побудована і є еквівалентною за змістом (семантиці) програмі вхідною мовою. Однак ефективність результуючої програми важлива для її розроблювача, тому більшість сучасних компіляторів виконують ще один етап компіляції - оптимізацію результуючої програми (або просто «оптимізацію»), щоб підвищити її ефективність, наскільки це можливо.

Важливо відзначити два моменти: по-перше, виділення оптимізації в окремий етап генерації коду - це змушений крок. Компілятор змушений робити оптимізацію побудованого коду, оскільки він не може виконати семантичний аналіз всієї вхідної програми в цілому, оцінити її зміст і виходячи з нього побудувати результуючу програму. По-друге, оптимізація - це необов'язковий етап компіляції. Компілятор може взагалі не виконувати оптимізацію, і при

цьому результуюча програма буде правильною, а сам компілятор буде повністю виконувати свої функції. Однак практично всі сучасні компілятори так чи інакше виконують оптимізацію, оскільки їхні розроблювачі прагнуть завоювати гарні позиції на ринку засобів розробки програмного забезпечення.

Тепер дамо визначення поняттю «оптимізація».

Оптимізація програми - це обробка, пов'язана з переупорядкуванням і зміною операцій у програмі, що компілюється, з метою одержання більш ефективної результуючої об'єктної програми. Оптимізація виконується на етапах підготовки до генерації й безпосередньо при генерації об'єктного коду.

Як показники ефективності результуючої програми можна використовувати два критерії: обсяг пам'яті, необхідний для виконання результуючої програми, і швидкість виконання (швидкодія) програми. Далеко не завжди вдається виконати оптимізацію так, щоб вона задовольняла обом цим критеріям. Найчастіше скорочення необхідного програмі обсягу даних веде до зменшення її швидкодії, і навпаки. Тому для оптимізації звичайно вибирається один зі згаданих критеріїв. Вибір критерію оптимізації звичайно виконується в налаштуваннях компілятора.

Але навіть вибравши критерій оптимізації, у загальному випадку практично неможливо побудувати код результуючої програми, який би був самим коротким або найшвидшим кодом, що відповідає вхідній програмі. Справа в тому, що немає алгоритмічного способу знаходження самої короткої або найшвидшої результуючої програми, еквівалентній заданій вихідній програмі. Це завдання в принципі нерозв'язне. Існують алгоритми, які можна прискорювати як завгодно багато разів для великого числа можливих вхідних даних, і при цьому для інших наборів вхідних даних вони виявляться неоптимальними [1,2]. До того ж компілятор має досить обмежені засоби аналізу семантики всієї вхідної програми в цілому. Усе. що можна зробити на етапі оптимізації - це виконати над заданою програмою послідовність перетворень у надії зробити її більше ефективною. Щоб оцінити ефективність результуючої програми, отриманої за допомогою того або іншого компілятора, часто прибігають до порівняння її з еквівалентною програмою (програмою, що реалізує той же алгоритм), отриманої з вихідної програми, написаної мовою

асемблера. Кращі оптимізуючі компілятори можуть одержувати результуючі об'єктні програми зі складних вихідних програм, написаних на мовах високого рівня, що майже не поступаються по якості програмам мовою асемблера. Звичайне співвідношення ефективності програм, побудованих за допомогою компіляторів з мов високого рівня, і програм, побудованих за допомогою асемблера, становить 1, 1-1,3. Тобто об'єктна програма, побудована за допомогою компілятора з мови високого рівня, звичайно містить на 10-30% більше команд, чим еквівалентна їй об'єктна програма, побудована за допомогою асемблера, а також виконується на 10-30% повільніше.

Це дуже непогані результати, досягнуті компіляторами з мов високого рівня, якщо зрівняти трудоємність розробки програм мовою асемблера й мовою високого рівня. Далеко не кожну програму можна реалізувати мовою асемблера в прийнятний термін (а значить і виконати прямо наведене вище порівняння можна тільки для вузького кола програм).

Оптимізацію можна реалізувати на будь-якій стадії генерації коду, починаючи від завершення синтаксичного розбору й аж до останнього етапу, коли породжується код результуючої програми. Якщо компілятор використовує кілька різних форм внутрішнього подання програми, то кожна з них може бути піддана оптимізації, причому різні форми внутрішнього подання орієнтовані на різні методи оптимізації (1-3, 7). Таким чином, оптимізація в компіляторі може виконуватися кілька разів на етапі генерації коду.

Принципово розрізняються два основних види оптимізуючих перетворень:

- перетворення вихідної програми (у формі її внутрішнього подання в компіляторі), що не залежать від результуючої об'єктної мови;
- перетворення результуючої об'єктної програми.

Перший вид перетворень не залежить від архітектури цільової обчислювальної системи, на якій буде виконуватися результуюча програма. Звичайно він заснований на виконанні добре відомих і обґрунтованих математичних і логічних перетворень, вироблених над внутрішнім поданням програми (деякі з них будуть розглянуті нижче).

Другий вид перетворень може залежати не тільки від властивостей

об'єктної мови (що очевидно), але й від архітектури обчислювальної системи, на якій буде виконуватися результуюча програма. Так, наприклад, при оптимізації може враховуватися обсяг кеш-пам'яті й методи організації конвеєрних операцій центрального процесора. У більшості випадків ці перетворення сильно залежать від реалізації компілятора і є «ноу-хау» виробників компілятора. Саме цей тип оптимізуючих перетворень дозволяє істотно підвищити ефективність результуючого коду.

Методи оптимізації, що використовуються, ні при яких умовах не повинні приводити до зміни «змісту» вихідної програми (тобто до таких ситуацій, коли результат виконання програми змінюється після її оптимізації). Для перетворень першого виду проблем звичайно не виникає. Перетворення другого виду можуть викликати складності, оскільки не всі методи оптимізації, що використовувались творцями компіляторів, можуть бути теоретично обґрунтовані й доведені для всіх можливих видів вихідних програм. Саме ці перетворення можуть вплинути на зміст вихідної програми. Тому в сучасних компіляторах існують можливості вибору не тільки загального критерію оптимізації, але й окремих методів, які будуть використовуватися при виконанні оптимізації.

Нерідко оптимізація веде до того, що зміст програми виявляється не зовсім таким, яким його очікував побачити розроблювач програми, але не через наявність помилки в оптимізуючій частині компілятора, а тому, що користувач не брав до уваги деякі аспекти програми, пов'язані з оптимізацією. Наприклад, компілятор може виключити із програми виклик деякої функції із заздалегідь відомим результатом, але якщо ця функція мала «побічний ефект» - змінювала деякі значення в глобальній пам'яті - зміст програми може змінитися. Найчастіше це говорить про поганий стиль програмування вихідної програми. Такі помилки важко помітити, для їхнього знаходження розроблювачеві програми варто звернути увагу на попередження, що видаються семантичним аналізатором, або відключити оптимізацію. Застосування оптимізації також недоцільно в процесі налагодження вихідної програми.

Методи перетворення програми залежать від типів синтаксичні конструкції вихідної мови. Теоретично розроблені методи оптимізації для багатьох типових конструкцій мов програмування.

Оптимізація може виконуватися для наступних типових синтаксичних конструкцій:

- лінійних ділянок програми;
- логічних висловів;
- циклів;
- викликів процедур і функцій;
- інших конструкцій вхідної мови.

У всіх випадках можуть використовуватися як машинно-залежні, так і машинно-незалежні методи оптимізації.

У лабораторній роботі використовуються два машинно-незалежних методи оптимізації лінійних ділянок програми. Тому тільки ці два методи будуть розглянуті далі. З іншими машинно-незалежними методами оптимізації можна більш докладно ознайомитися в [1,2,7]. Що стосується машинно-залежних методів, то вони, як правило, рідко згадуються в літературі. Деякі з них розглядаються в технічних описах компіляторів.

Принципи оптимізації лінійних ділянок

Лінійна ділянка програми - це виконувана один по одному послідовність операції, що має один вхід і один вихід. Найчастіше лінійна ділянка містить послідовність обчислень, що складаються з арифметичних операцій і операторів присвоювання значень змінним.

Будь-яка програма передбачає виконання обчислень і присвоювання значень, тому лінійні ділянки зустрічаються в будь-якій програмі. У реальних програмах вони становлять істотну частину програмного коду. Тому для лінійних ділянок розроблений широкий спектр методів оптимізації коду. Крім того, характерною рисою будь-якої лінійної ділянки є послідовний порядок виконання операції, що входять у його склад. Жодна операція в складі лінійної ділянки програми не може бути пропущена, жодна операція не може бути виконане більше число раз, чим сусідні з нею операції (інакше цей фрагмент програми просто не буде лінійною ділянкою). Це істотно спрощує завдання оптимізації лінійних ділянок програм. Оскільки всі операції лінійної ділянки виконуються послідовно, їх можна пронумерувати в порядку їхнього виконання

Для операцій, що становить лінійна ділянка програми, можуть

застосовуватися наступні види оптимізуючих перетворень:

• видалення непотрібних присвоювань;

• виключення надлишкових обчислень (зайвих операцій);

• згортка операцій об'єктного коду;

• перестановка операцій;

• арифметичні перетворення.

Далі розглянуті два методи оптимізації лінійних ділянок: виключення зайвих операцій і згортка об'єктного коду.

Згортка об'єктного коду

Згортка об'єктного коду — це виконання під час компіляції тих операцій вихідної програми, для яких значення операндів уже відомі. Немає необхідності багаторазово виконувати ці операції в результуючій програмі - цілком достатньо один раз виконати їх при компіляції.

УВАГА

Не слід плутати оптимізацію по методу згортки об'єктного коду з розглянутим у лабораторній роботі № 3 алгоритмом - «здвиг згортка». Згортка об'єктного коду й згортка за правилами граматики при виконанні синтаксичного розбору - це принципово різні операції.

Найпростіший варіант згортки - виконання в компіляторі операцій, операндами яких є константи. Трохи більше складний процес визначення тих операцій, значення яких можуть бути відомі в результаті виконання інших операцій. Для цієї мети при оптимізації лінійних ділянок програми використовується спеціальний алгоритм згортки об'єктного коду. Алгоритм згортки для лінійної ділянки програми працює зі спеціальною таблицею T, яка містить пари (<змінна>, <константа>) для всіх змінних, значення яких уже відомі. Крім того, алгоритм згортки позначає ті операції у внутрішнім поданні програми, для яких у результаті згортки вже не потрібна генерація коду. Тому що при виконанні алгоритму згортки враховується взаємозв'язок операцій, зручною формою подання для нього є тріади, оскільки в інших формах подання операцій (таких як тетради або команди асемблера) потрібні додаткові

структури, щоб відбити зв'язок результатів одних операцій з операндами інших. Розглянемо виконання алгоритму згортки об'єктного коду для тріад. Для позначки операцій, не потребуючі породження об'єктного коду, будемо використовувати тріади спеціального виду С(К,0).

Алгоритм згортки тріад послідовно переглядає тріади лінійної ділянки й для кожної тріади робить наступне:

1. Якщо операнд є змінна, котра втримується в таблиці Т, то операнд заміняється на відповідне значення константи.

2. Якщо операнд є посилання на особливу тріаду типу С(К,0), то операнд заміняється на значення константи К.

3. Якщо всі операнди тріади є константами, то тріада може бути згорнута. Тоді дана тріада виконується й замість її міститься особлива тріада виду С(К,0), де К - константа, що є результатом виконання згорнутої тріади. (При генерації коду для особливої тріади об'єктний код не породжується, а тому вона надалі може бути просто виключена.)

4. Якщо тріада є присвоюванням типу А:=В, тоді:

• Якщо В - константа, то А зі значенням константи заноситься в таблицю Т (якщо там уже було старе значення для А, то це старе значення виключається);

• Якщо В - не константа, то А взагалі виключається з таблиці Т, якщо воно там є.

Розглянемо приклад виконання алгоритму.

Нехай фрагмент вихідної програми (записаної мовою типу Pascal) має вигляд;

I:= 1*1;

I:=3;

J:= 6*I+I;

Її внутрішнє подання у формі тріад буде мати вигляд:

1. +(1.1)

2. =(I.^1)

3. =(I. 3)

4. *(6. I)

5. +(^4. I)

6. =(J. ^5)

Процес виконання алгоритму згортки показаний у табл. 4.1.

Таблиця 4.1. Приклад роботи алгоритму згортки

Тріада	*Крок 1*	*Крок 2*	*Крок 3*	*Крок 4*	*Крок 5*	*Крок 6*
1	*С (2,0)*	*С (2,0)*	*С (2,0)*	*С (2,0)*	*С (2,0)*	*С (2,0)*
2	*:=(I,^1)*	*:=(I, 2)*	*:=(I, 2)*	*:=(I, 2)*	*:=(I, 2)*	*:=(I, 2)*
3	*:=(I, 3)*	*:=(I, 3)*	*:=(I, 3)*	*:=(I, 3)*	*:=(I, 3)*	*:=(I, 3)*
4	**(6. I)*	**(6. I)*	**(6. I)*	*З (18,0)*	*З (18,0)*	*З (18,0)*
5	*+ (^4, I)*	*+ (^4, I)*	*+ (^4, I)*	*+ (^4, I)*	*З (21,0)*	*З (21,0)*
6	*:=(J,^5)*	*:=(J,^5)*	*:=(J,^5)*	*:=(J,^5)*	*:=(J,^5)*	*:=(J,21)*
Т	*(,)*	*(1,2)*	*(1,3)*	*(1,3)*	*(1,3)*	*(1,3) (J, 21)*

Якщо виключити особливі тріади типу С(К, 0) (які не породжують об'єктного коду), то в результаті виконання згортки одержимо наступну послідовність тріад:

1. :=(I, 2)
2. :=(I, 3)
3. :=(J,21)

Видно, що результуюча послідовність тріад може бути піддана подальшої оптимізації - у ній присутні зайві присвоювання, але інші методи оптимізації виходять за рамки даної лабораторної роботи (з ними можна ознайомитися в [1,2,7]).

Алгоритм згортки об'єктного коду дозволяє виключити з лінійної ділянки програми операції, для яких на етапі компіляції вже відомий результат. За рахунок цього скорочується час виконання, а також обсяг коду результуючої програми.

Згортка об'єктного коду, у принципі, може виконуватися не тільки для лінійних ділянок програми. Коли операндами є константи, логіка виконання програми значення не має - згортка може бути виконана в кожному разі. Якщо ж необхідно враховувати відомі значення змінних, то потрібно брати до уваги й логіку виконання результуючої програми. Тому для нелінійних ділянок програми (розгалужень і циклів) алгоритм буде більше складним, чим

послідовний перегляд лінійного списку тріад.

Виключення зайвих операцій

Виключення надлишкових обчислень (зайвих операцій) полягає в знаходженні й видаленні з об'єктного коду операцій, які повторно обробляють ті самі операнди.

Операція лінійної ділянки з порядковим номером і вважається *зайвою операцією,* якщо існує ідентична їй операція з порядковим номером J, J < і і ніякий операнд, оброблюваний операцією з порядковим номером і, не замінювався ніякою іншою операцією, що має порядковий номер між і і J.

Алгоритм виключення зайвих операцій переглядає операції в порядку їхнього проходження. Так само як і алгоритму згортки, алгоритму виключення зайвих операцій найпростіше працювати із тріадами, тому що вони повністю відбивають взаємозв'язок операцій.

Розглянемо алгоритм виключення зайвих операцій для тріад. Щоб стежити за внутрішньою залежністю змінних і тріад, алгоритм привласнює їм деякі значення, які називають *числами залежності,* за наступними правилами:

- завжди для кожної змінної її число залежності дорівнює 0, тому що на початку роботи програми значення змінної не залежить ні від якої тріади;
- після обробки і - ої тріади, у якій змінній А привласнюється деяке значення, число залежності А (dep(A)) одержує значення і, тому що значення А тепер залежить від даної і-ої тріади:
- при обробці і-ої тріади її число залежності (dep(i)) приймається рівним значенню 1 + (максимальне_з_чисел_залежності_операндів).

Таким чином, при використанні чисел залежності тріад і змінних можна затверджувати, що якщо і-а тріада ідентична J-ій тріаді (J<i), те і -а тріада вважається зайвою, в тім і тільки в тому випадку, коли dep(i)=dep(j). Алгоритм виключення зайвих операцій використовує у своїй роботі тріади особливого виду SAME(J,0). Якщо така тріада зустрічається в позиції з номером і, то це означає, що у вихідній послідовності тріад деяка тріада і ідентична тріаді j.

Алгоритм виключення зайвих операцій послідовно переглядає тріади лінійної ділянки. Він складається з наступних кроків, виконуваних для кожної тріади:

1. Якщо якийсь операнд тріади посилається на особливу тріаду виду SAME(J,0), то він заміняється на посилання на тріаду з номером j (^j).
2. Обчислюється число залежності поточної тріади з номером i, виходячи із чисел залежності її операндів.
3. Якщо в переглянутій частині списку тріад існує ідентична J-a тріада, причому J < i і dep(i)=dep(j), то поточна тріада i заміняється на тріаду особливого виду SAME(J,0)).
4. Якщо поточна тріада є присвоювання, то обчислюється число залежності відповідної змінної.

Розглянемо роботу алгоритму на прикладі:

D:=D+C*B;

A:=D+C*B;

C:=D+C*B.

Цьому фрагменту програми буде відповідати наступна послідовність тріад:

1: *(C,B)

2: +(D,^1)

3: :=(D,^2)

4: *(C,B)

5: +(D,^4)

6: :=(A,^5)

7: *(C,B)

8: +(D,^7)

9: :=(C,^8)

Видно, що в даному прикладі деякі операції обчислюються двічі над тими самими операндами, а виходить, вони є зайвими й можуть бути виключені. Робота алгоритму виключення зайвих операцій відображена в табл. 4.2.

Таблиця 4.2. Приклад роботи алгоритму виключень зайвих операцій

тріада I, що обробляється		Числа залежності змінних				Числа залежності тріад dep(i)	Отримані тріади після виконання алгоритму
		A	B	C	D		
1) *(C,B)	►	0	0	0	0	1	1) *(C,B)
2) +(D,^1)		0	0	0	0	2	2) +(D,^1)
3) :=(D,^2)	►	0	0	0	3	3	3) :=(D,^2)
4) *(C,B)	►	0	0	0	3	1	4) SAME (1,0)
5) +(D,^4)		0	0	0	3	4	5) +(D,^1)
6) :=(A,^5)		6	0	0	3	5	6) :=(A,^5)
7) *(C,B)	►	6	0	0	3	1	7) SAME (1,0)
8) +(D,^7)	►	6	0	0	3	4	8) SAME (5,0)
9) :=(C,^8)	►	6	0	9	3	5	9) := (C, ^5)

Тепер, якщо виключити тріади особливого виду SAME(J,0), то в результаті виконання алгоритму одержимо наступну послідовність тріад:

1: *(C,B)

2: +(D,^1)

3: :=(D,^2)

4: +(D,^1)

5: :=(A,^4)

6: :=(C,^4)

Зверніть увагу, що в підсумковій послідовності змінилася нумерація тріад і номера в посиланнях одних тріад на інші. Якщо в компіляторі як посилання використовувати не номери тріад, а безпосередньо покажчики на них, то зміни посилань у такому варіанті не буде потрібно.

Алгоритм виключення зайвих операцій дозволяє уникнути повторного виконання тих самих операцій над тими самими операндами. У результаті оптимізації по цьому алгоритмі скорочується й час виконання, і обсяг коду результуючої програми.

Загальний алгоритм генерації й оптимізації об'єктного коду

Тепер розглянемо загальний варіант алгоритму генерації коду, що одержує на вході дерево висновку (побудоване в результаті синтаксичного розбору) і створює по ньому фрагмент об'єктного коду результуючої програми. Алгоритм повинен виконати наступну послідовність дій:

- побудувати послідовність тріад на основі дерева висновку;

- виконати оптимізацію коду методом згортки для лінійних ділянок результуючої програми;
- виконати оптимізацію коду методом виключення зайвих операцій для лінійних ділянок результуючої програми;
- перетворити послідовність тріад у послідовність команд мовою асемблера (отримана послідовність команд і буде результатом виконання алгоритму).

Алгоритм перетворення тріад у команди мови асемблера - це єдина машинно-залежна частина загального алгоритму. При перетворенні компілятора для роботи з іншим результуючим об'єктним кодом буде потрібно поставити тільки цю частину, при цьому всі алгоритми оптимізації й внутрішнє подання програми залишаться незмінними.

У даній роботі алгоритм перетворення тріад у команди мови асемблера пропонується розробити самостійно. У тривіальному виді такий алгоритм заміняє кожну тріаду на послідовність відповідних команд, а результат її виконання запам'ятовується в часовий змінній з деяким ім'ям (наприклад TMPi, де i - номер тріади). Тоді замість посилання на цю тріаду в іншій тріаді буде підставлене значення цієї змінної. Однак алгоритм може передбачати й оптимізацію тимчасових змінних.

Вимоги до виконання роботи

Порядок виконання роботи

1. Варіанти завдань відповідають варіантам завдань для лабораторної роботи №3. Для виконання роботи рекомендується використовувати результати, отримані в ході виконання лабораторних робіт № 2 і 3.
2. Вивчити алгоритм генерації об'єктного коду по дереву синтаксичного розбору.
2. Розробити схеми Су-перекладу для операцій вихідної мови відповідно до заданої граматики.
3. Виконати генерацію послідовності тріад вручну для обраного найпростішого приклада. Перевірити коректність результату.
4. Вивчити й реалізувати (якщо потрібно) для заданої вхідної мови алгоритми оптимізації результуючого коду методом згортки й методом виключення зайвих операцій.
5. Розробити алгоритм перетворення послідовності тріад у заданий об'єктний код (за узгодженням з викладачем).
6. Підготувати й захистити звіт.
7. Написати й налагодити програму на ПК.
8. Здати працюючу програму викладачеві.

Вимоги до оформлення звіту

1. Звіт повинен містити наступні розділи:
2. Завдання по лабораторній роботі.
3. Короткий виклад мети роботи.
4. Запис заданої граматики вхідної мови у формі Бекуса-Наура.
5. Опис схем СУ-перекладу для операцій вихідної мови відповідно до заданої граматики.
6. Приклад генерації й оптимізації послідовності тріад на основі найпростішої вихідної програми.
7. Текст програми (оформляється після виконання програми на ПК).

Контрольні питання:

1. Що таке транслятор, компілятор і інтерпретатор? Розкажіть про загальну структуру компілятора.
2. Як будується дерево висновку (синтаксичного розбору)? Які вихідні дані необхідні для його побудови?
3. Яку роль виконує генерація об'єктного коду? Які дані необхідні компіляторою для генерації об'єктного коду? Які дії виконує компілятор перед генерацією?
4. Поясніть, чому генерація об'єктного коду виконується компілятором по окремих синтаксичних конструкціях, а не для всієї вихідної програми в цілому.
5. Розкажіть, що таке синтаксично керований переклад.
6. Поясніть роботу алгоритму генерації послідовності тріад по дереву синтаксичного розбору на своєму прикладі.
7. За рахунок чого забезпечується можливість генерації коду на різних об'єктних мовах по тому самому дереву?
8. Дайте визначення поняттю оптимізації програми. Для чого використовується оптимізація? Яким умовам повинна задовольняти оптимізація?
9. Поясніть, чому генерацію програми доводиться проводити у два етапи: генерація й оптимізація.
10. Які існують методи оптимізації об'єктного коду?
11. Що таке тріади й для чого вони використовуються? Які ще існують методи для подання об'єктних команд?
12. Поясніть роботу алгоритму згортки. Приведіть приклад виконання згортки об'єктного коду.
13. Що таке зайва операція? Що таке число залежності?
14. Поясніть роботу алгоритму виключення зайвих операцій. Приведіть приклад виключення зайвих операцій.

Додаток 1

Приклад звіту про виконаня лабораторної роботи

Міністерство освіти і науки України

Центральноукраїнський національний технічний університет

Механіко-технологічний факультет

Кафедра кібербезпеки та програмного забезпечення

ЗВІТ

ПРО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № _
з навчальної дисципліни

“ Системне програмне забезпечення”

на тему: ПОБУДОВА

НАЙПРОСТІШОГО ДЕРЕВА

ВИВОДУ

Виконав: студент/ка

групи КІ/КН-23М

Морозко Т. Г.

Перевірив: ст.викладач

Дрєєва Г.М.

Кропивницький 2025

Тема: Побудова найпростішого дерева виводу

Мета: вивчення основних понять теорії граматик простого і операторного передування, ознайомлення з алгоритмами синтаксичного аналізу (розбору) для деяких класів КС-граматик, отримання практичних навиків створення простого синтаксичного аналізатора для заданої граматики операторного передування.

Варіант: 12

4. $S \rightarrow F;$
$F \rightarrow$ **for (T) do F | a := a**

Початкові граматики і типи допусти

Нижче в табл. 3.1 приведені номери завдань. Д
вказана відповідна йому граматика і типи допустимих ле

Таблиця 3.1. Номери завдань для виконання лаб

№	№ вар. граматики	Допустимі лексеми вхідної мови
1	1	Ідентифікатори, десяткові числа з плаваючою крапкою
2	2	Ідентифікатори, константи true та false
3	3	Ідентифікатори, десяткові числа з плаваючою крапкою
4	4	Ідентифікатори, десяткові числа з плаваючою крапкою

Рисунок 1 – Опис граматики

Завдання: Для виконання лабораторної роботи потрібно написати програму, яка виконує лексичний аналіз вхідного тексту відповідно до завдання, породжує таблицю лексем і виконує синтаксичний розбір тексту по заданій граматиці з побудовою дерева розбору. Текст на вхідній мові задається у вигляді символьного (текстового) файлу. Синтаксис вхідної мови і перелік допустимих лексем вказані в завданні. Допускається, що текст містить не більш за одну пропозицію вхідної мови.

За наявності у вхідному файлі тексту, відповідного заданій мові, програма повинна будувати і відображати дерево синтаксичного розбору. Якщо ж текст у вхідному файлі містить помилки (лексичні або синтаксичні), програма повинна видавати повідомлення про наявність помилок у вхідному тексті і коректно

завершувати своє виконання.

Рекомендується розбити програму на три складові частини: лексичний аналіз, побудова ланцюжка висновку та побудову дерева висновку Лексичний аналізатор повинен виділяти в тексті лексеми мови і замінювати їх на термінальний символ граматики (який в завданні позначений як a). Одержаний після лексичного аналізу ланцюжок повинен розглядатися в другій частині програми відповідно до алгоритму розбору При невдалому завершенні алгоритму видається повідомлення про помилку, при вдалому - будується ланцюжок висновку. Після побудови ланцюжка висновку на її основі будується дерево розбору, в якому символи a послідовно замінюються на лексеми з таблиці лексем. Для виконання лексичного аналізу рекомендується використовувати програмні модулі, створені в результаті виконання лабораторної роботи № 2. Довжину ідентифікаторів і строкових констант можна вважати обмеженою 32 символами. Програма повинна допускати коментарі необмеженої довжини у вхідному файлі. Форму організації коментарів пропонується вибрати самостійно.

Алгоритм виконання:

1. Ініціалізація: Алгоритм отримує список токенів і встановлює початкову позицію для відстеження поточного токена. Токени — це окремі частини вхідного виразу, що підлягають розбору.

2. Поточний токен: На кожному кроці алгоритм перевіряє поточний токен, використовуючи поточну позицію в списку. Якщо токени закінчились, функція повертає сигнал, що всі вони оброблені.

3. Порівняння токенів: На кожному етапі розбору алгоритм порівнює поточний токен із очікуваним. Якщо збігу немає, це сигналізує про синтаксичну помилку, і алгоритм зупиняється.

4. Розбір граматичних конструкцій: Алгоритм починає з аналізу основної граматичної конструкції, яка називається "S". Ця конструкція може включати інші підконструкції, такі як "F" і "T". Алгоритм спочатку розбирає "S", а потім переходить до наступних елементів.

5. Рекурсивний розбір: Кожна граматична конструкція може містити

підконструкції. Якщо алгоритм зустрічає певний ключовий токен (наприклад, цикл), він починає розбір підконструкцій. Наприклад, для циклу "for" він шукає вираз у дужках і тіло циклу, яке може включати інші конструкції.

6. Обробка присвоєнь: Якщо алгоритм зустрічає операцію присвоєння, він перевіряє правильність структури (лівої і правої частини виразу) і включає цю інформацію в дерево розбору.

7. Обробка виразів: Для виразів алгоритм перевіряє наявність правильних елементів, таких як порівняння або інші прості вирази, і додає їх у дерево розбору.

8. Побудова дерева розбору: На кожному етапі розбору алгоритм будує ієрархічне дерево, яке відображає структуру виразу, що розбирається. Це дерево повертається як результат синтаксичного аналізу.

```
Дерево розбору у вигляді списку:
['S', ['F', 'for', '(', ['T', 'i < 10'], ')', 'do', ['F', 'a', ':=', '0,75']]]

Дерево розбору з відступами:
S
  F
    for
    (
    T
      i < 10
    )
    do
    F
      a
      :=
      0,75

Process finished with exit code 0
```

Рисунок 2 – Результат: дерево висновку

Контрольні запитання

1. Яку роль виконує синтаксичний аналіз в процесі компіляції?
2. Які проблеми виникають при побудові синтаксичного аналізатора і як вони можуть бути вирішені?

3. Які типи граматик існують? Що таке КС- граматики? Розкажіть про їх використання в компіляторі.
4. Які типи розпізнавачів для КС- граматики існують? Розкажіть про недоліки та переваги різних типів розпізнавачів.
5. Поясніть правила побудови дерева виведення граматики.
6. Що таке граматика простого передування?
7. Як обчислюються відносини передування для граматик простого передування ?
8. Що таке граматика операторного передування ?
9. Як обчислюються відношення для граматик операторного передування ?
10.Розкажіть про завдання розбору. Що таке розпізнавач мови?
11. Розкажіть про загальні принципи роботи розпізнавача мови.
12. Що таке перенесення, згортка? Для чого необхідний алгоритм «перенесення-згортка»?
13. Розкажіть, як працює алгоритм «перенесення-згортка» в загальному випадку (з поверненнями).

…..

(відповіді)…..

……

ДОДАТОК А (лістинг програми)

main.py

```
class Parser:
def __init__(self, tokens):
self.tokens = tokens # список токенів
self.pos = 0 # поточна позиція в токенах
print("\nДерево розбору з відступами:")
parser.print_tree(parse_tree) # Виводимо дерево з відступами
except SyntaxError as e:
print(f"Синтаксична помилка: {e}")
```

Висновок:

У процесі роботи я ознайомився на практиці з ……. Навчився створювати …... Навчився працювати з …….(свої висновки)

Рекомендовані джерела інформації

1. Keith D. Cooper, Linda Torczon / Engineering a Compiler, 3rd Edition / Keith D. Cooper, Linda Torczon / Morgan Kaufmann, 2023 – 832 p.
2. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman / Compilers: Principles, Techniques, and Tools, 2nd Edition / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman / Pearson Education, 2006 – 1000 p.
3. Torben Ægidius Mogensen / Basics of Compiler Design / Torben Ægidius Mogensen / Springer, 2017 – 382 p.
4. Dick Grune, Ceriel J. H. Jacobs / Parsing Techniques: A Practical Guide, 2nd Edition / Dick Grune, Ceriel J. H. Jacobs / Springer, 2008 – 748 p.
5. Thain D. Introduction to Compilers and Language Design. — University of Notre Dame, 2019. — 254 p. https://dthain.github.io/books/compiler/
6. Mogensen T. Basics of Compiler Design. — University of Copenhagen, 2011 (rev. edition). — 210 p. https://hjemmesider.diku.dk/~torbenm/Basics/basics_lulu2.pdf
7. Основи теорії побудови компіляторів: Навч. метод. посібник/ Гавриленко С. Ю., Бельорін-Еррера О. М. – Харків: НТУ «ХПІ», 2024. – 138 с. https://repository.kpi.kharkov.ua/server/api/core/bitstreams/87123284-02a8-4b0e-993a-5ff1aa1ffd5d/content