Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Центр післядипломної освіти_____
(повна назва)

Кафедра _____Програмної інженерії_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

_____другий (магістерський)_____
(рівень вищої освіти)

Дослідження методів розробки компіляторів для мов програмування
(тема)

Виконав: студент 2 курсу, групи ІПЗмзд-17-1___
спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)
Освітньо-наукової програми
Інженерія програмного забезпечення_____
(повна назва освітньої програми)

_____Церінгер Б.К._____
(прізвище, ініціали)
Керівник _____доц. Чуприна А.С._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2019 р.

Харківський національний університет радіоелектроніки

Факультет_ Центр післядипломної освіти_____

Кафедра_Програмної інженерії_____

Рівень вищої освіти_____другий (магістерський)_____

Спеціальність _121-Інженерія програмного забезпечення_____
(код і повна назва)
освітньо-наукова програма _Інженерія програмного забезпечення_____
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові _Церінгер Борис Костянтинович_____
(прізвище, ім'я, по батькові)

1. Тема роботи _ Дослідження методів розробки компіляторів для мов програмування_____
затверджена наказом по університету від "22" квітня 2019 р № 65 Стз
2. Термін подання студентом роботи до екзаменаційної комісії
05 червня 2019 р.
3. Вихідні дані до роботи *методів розробки компіляторів для мов програмування. Використовувати ОС Windows, середовище розробки Eclipse.*_____
4. Перелік питань, що потрібно опрацювати в роботі *меиа роботи, дослідити концепцію розробки компілятора мови програмування, основні методи, моделі, інструменти реалізаії. Скронструювати компілятор язика miniJava.*_____
5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) *Мета завдання, аналіз предметної області, обґрунтування розробки, постановка задачі, базові моделі, методи й алгоритми, структура програмної системи, схема взаємодії даних, інтерфейс програмної системи, результати роботи програмної системи.*_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка*
1.	Аналіз предметної галузі		
2.	Огляд існуючих методів		
3.	Методи розв’язання задачі		
4.	Підготовка пояснювальної записки		
5.	Спецчастина		
6.	Підготовка презентації та доповіді		
7.	Попередній захист		
8.	Нормоконтроль, рецензування		
9.	Занесення диплома в електронний архів		
10.	Допуск до захисту у зав. Кафедри		

Дата видачі завдання _11__ ____лютого___2019 р.

Студент _____
(підпис)

Керівник роботи _____ __доц. Чуприна А.С._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Звіт з науково-дослідної практики магістрів: 75 с., 10 рис., 13 джерел.

КОМПІЛЯТОР, ЛЕКСИЧНИЙ АНАЛІЗ, МОВА ПРОГРАМУВАННЯ, СИНТАКСИЧНИЙ АНАЛІЗ, СЕМАНТИЧНИЙ АНАЛІЗ.

Об’єкт – аналіз методів розробки компілятора мови програмування.

Мета роботи – дослідити концепцію розробки компілятора мови програмування, основні методи, моделі, інструменти реалізаії.

В роботі проведений аналіз предметної галузі, досліджені основні методи, моделі, інструменти реалізаії, концепція розробки компілятора мови програмування.

COMPILATOR, LANGUAGE PROGRAMMING, LEXICIAL ANALYSIS, SYNTAXIC ANALYSIS, SEMANTIC ANALYSIS.

The object is an analysis of the methods of programming a compiler programming language.

The purpose of the work is to investigate the concept of developing a programming language compiler, basic methods, models, implementation tools.

In this work the analysis of the subject area was carried out, the main methods, models, implementation tools, concept of development of the programming language compiler was studied.

ЗМІСТ

ВСТУП

Через швидкі темпи розвитку галузі інформаційних технологій глибоке розуміння основних принципів роботи різноманітних програмних засобів стає все більш актуальним. Велике різноманіття високорівневих програмних мов та фреймворків вимагає створення відповідної кількості програмних засобів компіляції[1] та інтерпретації коду. Архітектура цих засобів напряму впливає на швидкість та безпомилковість їх роботи. Тут швидкість є критично-важливою, адже в процесі розробки програмних продуктів компілювання виконується дуже часто і напряму впливає на його вартість.

В даній роботі буде розглянуто процес побудови компілятору на прикладі спрощеної мови програмування MiniJava та з використанням генератора синтаксичних аналізаторів ANTLR. Розробка виконувалась у середовищі Eclipse IDE.

Серед ключових етапів розробки можна виділити:

- проектування граматики для MiniJava;
- створення таблиці символів;
- перевірка відповідності типів (TypeCheckVisitor);
- генерація псевдокоду та його збереження на диску (CodeGenerationVisitor);
- виконання псевдокоду.

Метою роботи є дослідження концепції розробки компілятора мови програмування, основних методів, моделей, інструментов реалізаії.

В роботі проведений аналіз предметної галузі, досліджені основні методи, моделі, інструменти реалізаії, концепція розробки компілятора мови програмування.

1 АНАЛІЗ ПРОБЛЕМНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Основні поняття процесу компіляції.

Програмне забезпечення можна створювати за допомогою багатьох мов програмування з різними парадигмами (процедурною, об'єктно-орієнтованою, функціональною, візуальною тощо). Завдання компілятора полягає в перетворенні подання програмного забезпечення, орієнтованого на користувача, в машинно-орієнтоване, у якому відбувається безпосереднє виконання програми комп'ютером. Компілятори – це спеціалізовані системи обробки тексту, що мають багато спільного з іншими інструментальними засобами обробки текстів, написаних мовою програмування або природною мовою.

Задача компілятора звичайно розглядається на двох етапах.

1. Етап аналізу, на якому аналізується вихідний текст.
2. Етап синтезу, на якому генерується машинно-орієнтоване подання.

Вхід етапу аналізу називається вихідним текстом чи вихідним кодом, а вихід етапу синтезу – цільовим текстом чи цільовим кодом. Перетворення вихідного коду в цільовий звичайно називається процесом компіляції. Процес компіляції здійснює компілятор. Компілятор мови можна також назвати реалізацією мови. Породжений компілятором цільовий код може мати вид машинного коду для деякої машини (комп'ютера) чи деякого проміжного коду, що надалі буде перетворений (уже за допомогою інших інструментальних засобів) у машинний код. Можливий також варіант, коли проміжний код безпосередньо використовується за допомогою інтерпретатора. Процес компіляції являє собою перетворення однієї мови на іншу, перехід від вихідного коду до цільового коду, що виконується на машині, можливо, після деяких перетворень:

Процес компіляції також включає третю мову – мову реалізації, під якою

розуміють мову написання компілятора (зазвичай, це мова С). Нею може бути та ж мова, що і вихідний чи цільовий код, але це необов’язково.

1.2 Типова структура компілятора.

Логічно процес компіляції розділяється на етапи, що, у свою чергу, діляться на фази. Фізично компілятор розділяється на проходи.

Основними етапами компіляції є аналіз (визначення структури і значення вихідного коду) і синтез (побудова цільового коду). Крім того, може бути етап попередньої обробки (препроцесінгу). Цей етап в основному пов’язаний з мовами С та С++.

Етап аналізу розділяють на три окремі фази:

– лексичний аналіз.
– синтаксичний аналіз.
– семантичний аналіз.

Лексичний аналіз – це відносно проста фаза, у якій формуються символи (чи лексеми) мови. Слова мови, наприклад, if, for, do чи ідентифікатори, наприклад, count, name, чи послідовності знаків, наприклад, ++, ==, зручно сприймати як один символ, як це робиться на етапі аналізу. Задачею фази лексичного аналізу чи лексичного аналізатора є перехід від послідовності знаків до символів мови, з якими надалі будуть працювати синтаксична і семантична фази. Такий підхід копіює поведінку людини при читанні програми: адже ми сприймаємо текст програми не як простий набір знаків, а як набір символів, що складаються з цих знаків.

Тут важливо відзначити, що лексичний аналізатор усього лише формує символи – їхня послідовність не має для нього ніякого значення, тобто,

лексичний аналізатор звичайно не працює з контекстом.

У процесі синтаксичного аналізу визначається загальна структура програми, що включає розуміння порядку проходження символів у програмі. Це означає, що синтаксичний аналізатор повинний мати інформацію про контекст, у якому він працює. Результатом роботи синтаксичного аналізатора є подання програми в деревоподібній формі, що називають синтаксичним деревом. Наприклад, вираз (a + b) * (c – d) може бути поданий у вигляді:

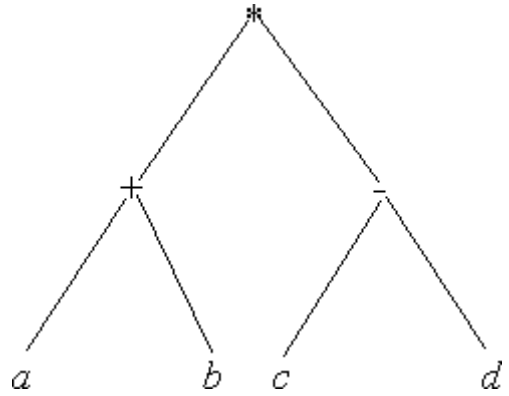


Рисунок 1.1 – Приклад синтаксичного дерева

Це подання називається абстрактним синтаксичним деревом. У такий же спосіб уся програма може бути представлена за допомогою абстрактних синтаксичних дерев.

Фаза синтаксичного аналізу є ключовою на етапі аналізу. Вона безпосередньо взаємодіє з лексичною фазою, а результати її роботи надалі будуть використовуватися семантичною фазою. Синтаксичний аналізатор зчитує символи в програмі зліва направо. У процесі зчитування він повинен уміти визначати, чи є послідовність вже прочитаних символів початком програми. Наприклад, перших п’ять вхідних символів можуть бути початком деякої програми, а перші шість – ні. У цьому випадку подальші дії компілятора будуть залежати від прийнятого способу відновлення після помилок.

Деякі властивості мов програмування не можуть бути перевірені простим скануванням зліва направо без створення таблиць довільного розміру. Перевірка таких властивостей мов програмування (що називають статичною семантикою)

виконується в семантичній фазі аналізу.

Етап синтезу процесу компіляції складається з наступних основних фаз:

– Генерація машинно-незалежного коду.
– Оптимізація машинно-незалежного коду.
– Розподіл пам’яті.Генерація машинного коду.
– Оптимізація машинного коду.

В окремих випадках деякі з цих фаз можуть бути відсутніми. Наприклад, якщо компілятор безпосередньо компілює в машинний код, перші дві фази можуть пропускатися. Оптимізація коду може відбуватися на рівні машинно-незалежного коду, на рівні машинного коду, на обох рівнях чи на жодному.

Існують причини, чому спочатку необхідно створювати машинно-незалежний код: це сприяє портабельності компілятора й служить для відокремлення залежності від мови та залежності від машини. Багато компіляторів також генерують деякі проміжні коди, що можуть бути незалежні від вихідної мови, машинної мови чи від обох. Прикладами таких проміжних мов є наприклад, P-код для Pascal, Diana для Ada, байт-код для Java.

Потреба в оптимізації генерованого коду може бути різною. Якщо потрібен ефективний код, компілятор зобов’язаний забезпечити значну оптимізацію. У той же час в багатьох середовищах швидкість роботи програмного забезпечення не є критичним параметром, отже, необхідна лише незначна оптимізація. Деякі типи оптимізації реалізувати просто, і тому їх часто включають у компілятори, тоді як інші форми оптимізації, особливо глобальні (на відміну від локальних), трудомісткі і вимагають значних витрат часу при компіляції, а тому застосовуються рідко. Багато компіляторів дозволяють користувачу самому визначити, що саме потрібно оптимізувати.

У фазі розподілу пам’яті кожна стала та змінна в програмі отримують зарезервоване місце в пам’яті для збереження свого значення. Дана область пам’яті може бути одного з трьох типів:

– статична пам’ять, якщо час життя змінної дорівнює часу життя

програми (не може бути звільнена до завершення виконання програми);

– динамічна пам’ять, якщо час життя змінної дорівнює часу життявизначеного блоку, функції чи процедури (може бути звільнена після виконання даного фрагмента програми);

– глобальна пам’ять, якщо на момент компіляції час життя змінної невідомо, а пам’ять повинна виділятися і звільнятися в процесі виконання.

Результатом роботи фази розподілу пам’яті є створення адреси.

Якщо логічно компілятор складається з етапів і фаз, фізично він складений із проходів. Компілятор здійснює прохід щоразу при зчитуванні вихідного коду чи його подання. Ранні компілятори були багатопрохідними через недостатній обсяг пам’яті машин того часу. Сучасні компілятори є переважно однопрохідними, тобто повний процес компіляції цілком виконується при однократному зчитуванні коду. У цьому випадку різні описані фази будуть виконуватися паралельно (що, як правило, є найбільш зручним), що усуває необхідність складного зв’язку між різними проходами.

1.3 Інтегровані середовища розробки

Сучасні компілятори часто є не окремими, автономними інструментальними засобами, а являють собою частину інтегрованих середовищ розробки (Integrated Development Environment – IDE), що іноді називають середовищами програмування. Крім надання засобу компіляції, сучасне середовище IDE пропонує засоби мовно-орієнтованого редагування,

налагодження, визначення робочих профілів програми, керування конфігурацією і т.д. Гарним прикладом такого середовища є IDE Borland C++, Eclipse, Intelij IDEA для Windows та Linux, що пропонує такі основні групи операці:

– редагування з засобами вирізання, вставки, скасування операції тощо;

– пошук із засобами заміни тексту та локалізації функцій в процесі налагодження;

– перегляд різних вікон, що містять засоби діагностики та іншу інформацію, пов’язану з поточним проектом (точки переривання програми, зміст регістрів, розташування змінних, використання класів і т.ін.);

– керування проектом, включаючи запуск нових проектів, компіляцію і зв’язування різних компонентів проекту;

– налагодження з можливістю запуску програми в режимі покрокового виконання, установки точок переривання, відстеження значень виразів, перегляду таблиць символів і т.д.

– засоби виконання, пов’язані з IDE.

1.4Вимоги до розробки компіляторів.

Загальна структура компілятора багато в чому залежить від його фазової структури і структури синтаксичного аналізатора, а структура синтаксичного аналізатора відбиває властивості вихідної мови. Звичайно при проектуванні

компілятора керуються такими вимогами:

– ефективна компіляція;

– мінімальний розмір компілятора;

– мінімальна довжина цільового коду;

– створення ефективного цільового коду;

– портабельність;

– простота використання;

– практичність.

Одночасно задовольнити всім цим вимогам неможливо, тому деяким з них доводиться віддавати перевагу. У навчальних середовищах, наприклад, ефективність компіляції й гарні засоби діагностики можуть бути більш важливими, ніж створення ефективного цільового коду, тоді як для вбудованих систем першочергове значення має розмір і ефективність цільового коду. Багато компіляторів дозволяють користувачу самому визначати режим роботи компілятора – ступінь оптимізації, виконання перевірок часу виконання і т.д.

Наведена структура вивчення теми дозволяє наповнити поняття компілятора змістом на основі вивчення загальних властивостей процесу компіляції, що найбільш повно відповідає вимогам до фундаментальної підготовки майбутнього вчителя інформатики.

1.5 Постановка задачі

На основі аналізу предметної галузі основною задачею є дослідженн концепції розробки компілятора мови програмування, основні методи, моделі, інструменти реалізаії.

Це дозволить побудувати компілятор на прикладі спрощеної мови програмування MiniJava та з використанням генератора синтаксичних аналізаторів ANTLR. Розробку виконати у Eclipse.

2 ДОСЛІДЖЕННЯ МЕТОДІВ, МОДЕЛЕЙ, АЛГОРИТМІВ РОЗРОБКИ КОМПІЛЯТОРА МОВИ ПРОГРАМУВАННЯ

2.1 Основні завдання компіляторів

Комп'ютери самі по собі здатні виконувати тільки дуже обмежений набір операцій, які називаються машинними кодами. У старі часи, коли з'явилися перші комп'ютери, програми писалися в машинних кодах, що представляють собою послідовності двійкових чисел, однозначно сприймаються комп'ютером. В кінці 50-х кодів минулого століття з'явилися перші мови програмування, такі як мова асемблера і Фортран. Для того, щоб комп'ютер міг зрозуміти програму, написану на якійсь мові програмування, необхідний перекладач (транслятор) такої програми в машинні коди. Відзначимо, що, якщо оператор мови асемблера відображається при трансляції частіше всего1 в одну машинну інструкцію, пропозиції мов більш високого рівня відображаються, взагалі кажучи, в кілька машинних інструкцій.

Транслятори бувають двох типів: компілятори (compiler) і інтерпретатори (interpreter). Процес компіляції складається з двох частин: аналізу (analysis) і синтезу (synthesis). Анализирующая частина компілятора розбиває вихідну програму на складові її елементи (конструкції мови) і створює проміжне представлення вихідної програми. Синтезує частина з проміжного представлення створює нову програму, яку комп'ютер в змозі зрозуміти. Така програма називається об'єктної програмою. Об'єктна програма може в подальшому виконуватися без перетрансляції. Як проміжний уявлення зазвичай

використовуються дерева, зокрема, так звані дерева розбору. Під деревом розбору розуміється дерево, кожен вузол якого є певною операції, а сини цього вузла - операндам.

2.1.1 Інтерпретатор

На відміну від компілятора, інтерпретатор не створює ніякої нової програми, а просто виконує кожне речення мови програмування. Можна сказати, що результатом роботи інтерпретатора є "число".

Взагалі кажучи, інтерпретатор, так само, як і компілятор, аналізує програму на вхідній мові, створює проміжне представлення, а потім виконує операції, що містяться в тексті цієї програми. Наприклад, інтерпретатор може побудувати дерево розбору, а потім виконати операції, якими позначені вузли цього дерева.

У тому випадку, якщо вихідний мова досить проста (наприклад, якщо це мова асемблера або Basic), то ніяке проміжне представлення не потрібно, і тоді інтерпретатор - це простий цикл. Він вибирає чергову інструкцію мови з вхідного потоку, аналізує і виконує її. Потім вибирається така інструкція. Цей процес триває до тих пір, поки не будуть виконані всі інструкції, або поки не зустрінеться інструкція, що означає закінчення процесу інтерпретації.

2.1.2 Компілятор

Компілятор переводить програми з однієї мови на іншу. Входом компілятора служить ланцюжок символів, складова вихідну програму на мові програмування L_1. Вихід компілятора (об'єктна програма) також є ланцюжком символів, але належить іншій мові L_2, наприклад, мови деякого комп'ютера. При цьому сам компілятор написаний на мові L_3, можливо, відрізняється від перших

двох. Будемо називати мову L_1 вихідним мовою, мова L_2 - цільовим мовою, а мова L_3 - мовою реалізації. Таким чином, можна говорити про компіляторі як про відображення множини L_1 в множина L_2 , тобто $K_3 : L_1 -> L_2$.

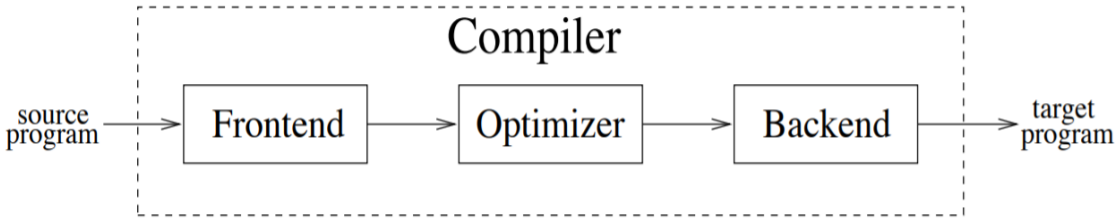


Рисунок 2.1 – Основні складові компілятора

Відзначимо, що далеко не завжди вихідні програми коректні з точки зору вихідної мови. Більш того, некоректні програми подаються на вхід компілятору значно частіше, ніж коректні - такий вже сучасний процес розробки програм. Тому вкрай важливою частиною процесу трансляції є точна діагностика помилок, допущених у вхідній програмі.

Існує величезна кількість різних мов програмування, починаючи з таких традиційних мов програмування як Fortran і Pascal і закінчуючи сучасними об'єктно-орієнтованими мовами такими, як C # і Java. Практично кожна мова програмування має якісь особливості з точки зору творця транслятора. Однак ми почнемо з розгляду різноманітних цільових мов компіляторів.

2.2 Завдання визначення мови

Одна з перших завдань, що виникають в процесі компіляції - це визначення розглянутого мови програмування. При розгляді мов, що складаються з кінцевого множини ланцюжків, найпростіше явно перерахувати всі допустимі вхідні ланцюжка. Але що робити з мовами, що не вводять ніяких обмежень на довжину

вхідного ланцюжка? Для потенційно нескінченних мов нам буде потрібно ввести якийсь конструктивний спосіб опису, який дозволить нам поставити правила, що описують породжуваний ними мову. Такий опис має задовольняти деяким властивостям: сам опис має мати кінцеву довжину, для даного опису мови повинен існувати алгоритм, який міг би перевірити приналежність деякій вхідний ланцюжка мови

Існує цілий ряд математичних формалізмов, в тій чи іншій мірі зручних для завдання мов - взагалі, етап аналізу вхідної програми найбільш розроблений і найкраще підтриманий математичними теоріями. Найбільш поширеним механізмом є граматики, які задають всі відповідні ланцюжки мови за допомогою деяких породжують правил. Очевидна гідність граматик полягає в тому, що існує безліч систем, які по заданій граматиці генерують програму, яка перевіряє відповідність вхідний ланцюжка визначається мови. Більш того, корисну роботу синтаксичного аналізатора (наприклад, побудова дерева розбору) можна проводити паралельно з самим розпізнаванням мови.

Інша часто використовувана ідея полягає в тому, що створюється певний узагальнений алгоритм, який перевіряє за кінцеве число кроків приналежність цього ланцюжка мови. Такий алгоритм або зупиняється після кінцевого числа кроків і каже "так", або зупиняється і каже "ні". Теоретично, нас могло б влаштувати і зациклення алгоритму на невідповідних вхідних ланцюжках, але на практиці така поведінка не зовсім зручно.

2.2.1 Визначення граматики

Граматики є найбільш поширений клас описів мов. При описі граматики необхідно почати з визначення алфавіту мови, який задається як набір допустимих термінальних символів. Крім того, необхідно визначити набір правил виведення вигляду $\alpha \rightarrow \beta$, за допомогою яких будуються всі ланцюжки мови. У

лівій і правій частині цих правил можуть зустрічатися спеціальні нетермінальні символи; в процесі виведення нетермінальні символи замінюються за допомогою відповідних правил до повної заміни на відповідні термінали. Нарешті, граматика повинна включати в себе початковий символ, або аксіому, з якої починається отримання будь-якої пропозиції мови.

Для формального визначення граматики нам будуть потрібні наступні позначення. Якщо A є алфавіт, то A^* позначає безліч всіх рядків (включаючи порожній рядок ε), складених із символів, що входять в A. Аналогічно, A^+ визначає безліч всіх рядків, складених із символів, що входять в A, але без порожнього рядка. Нетермінали ми будемо позначати прописними буквами, а термінали - малими.

Отже, граматика G визначається як наступна четвірка:

$$G = (V_T, V_N, P, S)$$

де V_T - кінцева множина термінальних символів; V_N - не перетиналися з V_ {T} кінцева множина нетермінальних символів; P - кінцевий набір породжують правил виду (α, β), де $\alpha \in V^+$, $\beta \in V^*$; S - початковий символ, де $S \in V_N$

2.2.2 Розпізнавачі для різних класів граматик

Під розпізнавачем ми будемо розуміти узагальнений алгоритм, що дозволяє визначити деяку множину (в нашому випадку - мова) і використовує в своїй роботі такі компоненти: вхідну стрічку, керуючий пристрій з кінцевої пам'яттю і додаткову робочу пам'ять. Зазвичай вважається, що керуючий пристрій може тільки читати інформацію, записану на вхідних стрічці (читання проводиться за допомогою вхідних головки, що вказує на поточний символ) і просуватися по ній вперед і, можливо, тому. Розпізнавач також може змінювати стан пам'яті, яка може бути організована як кінцевий лінійний список осередків або як стек (в

російській літературі званий також магазином). Як приклади распознавателей можна назвати машину Тьюринга, кінцеві і магазинні автомати, які повинні бути відомі студентам з попередніх курсів.

Мова визначається шляхом завдання деякої множини допустимих заключних станів распознавателя: якщо ланцюжок, подана на вхідну стрічку, дозволяє розпізнавачів виконати послідовність кроків і зупинитися в заключному стані, то ланцюжок належить мові.

Виявляється, кожному класу граматик з ієрархії Хомського відповідає клас распознавателей, що визначає той же клас мов:

- мова L праволінейний тоді і тільки тоді, коли він визначається (одностороннім детермінованим) кінцевим автоматом
- мова L контекстно-вільну тоді і тільки тоді, коли він визначається (одностороннім недетермінованим) автоматом з магазинною пам'яттю
- мова L контекстно-залежний тоді і тільки тоді, коли він визначається (двостороннім недетермінованим) автоматом з магазинною пам'яттю
- мова L рекурсивно перелічувальний тоді і тільки тоді, коли він визначається машиною Тьюринга (цими поняттями ми оперувати будемо; формально вони визначаються в курсі "обчислюваності" або в базовому курсі "Комп'ютерні науки").

2.2.3 Кінцеві автомати

Кінцевий автомат - це один з найпростіших механізмів, використовуваних при роботі з мовами. У цього засобу розв'язання є тільки фіксований набір елементів пам'яті, а керуючий пристрій може тільки зрушуватися вправо по вхідних стрічці і змінювати стан телефону. Основна частина кінцевого автомата - це функція переходу, що визначає можливі переходи для будь-якого поточного стану і будь-якого вхідного символу. Мається на увазі, що допускається

можливість переходу відразу в кілька різних станів автомата, тобто керуючий пристрій распознавателя може бути і недетермінованим. Недетермінірованность треба розуміти наступним чином: якщо можливо відразу кілька переходів з даного стану, то створюється кілька копій нашого автомата, по одному на кожне нове стан. Ланцюжок вважається, що належить мові, якщо хоча б одна з послідовностей кроків завершується в заключному стані.

Після такого короткого пояснення основних ідей ми можемо дати формальне визначення кінцевого автомата.

Визначення. Кінцевий автомат - це п'ятірка

$$M = (Q, \Sigma, \delta, q_0, F), \text{F})$$

де Q - кінцевої множини станів, Σ - кінцевої множини допустимих вхідних символів, δ - відображення множини Q x Σ в множині P(Q), що визначає поведінку керуючого пристрою (цю функцію часто називають функцією переходів), $q_0 \in Q$ - початковий стан керуючого пристрою, $F \subsetneq Q$ - множина заключних станів

В принципі, для визначення подальших дій кінцевого автомата досить знати поточний стан керуючого пристрою плюс послідовність все ще необроблених символів на вхідних стрічці. Цей набір даних називається конфігурацією автомата.

2.2.3 Детерміновані кінцеві автомати

Визначимо детермінований кінцевий автомат як окремий випадок загального (недетермінірованного) кінцевого автомата і введемо деякі додаткові визначення та угоди, які стануть в нагоді нам надалі:

Визначення. Автомат називається детермінованим, якщо множина $\delta(q,a)$ містить не більше одного стану для будь-яких q, a. Якщо $\delta(q,a)$ завжди містить рівно один стан (тобто не має невизначених переходів), то автомат називається повністю визначеним.

Визначення. Слово $\omega = \alpha_1 \dots \alpha_\kappa$ над алфавітом Σ допускається кінцевим автоматом $M = (Q, \Sigma, \delta, q_0, F)$, якщо існує послідовність станів $q_1, q_2, \dots, q_n$ така, що $q_1 = q_0, q_n \in F$ для $\forall \imath \forall j: 1 \leq i < n, 1 \leq j < k \;\; \delta(q_i, a_j) = q_{i+1}$.

Визначення. Мова L розпізнається кінцевим автоматом, якщо кожне слово мови L допускається цим кінцевим автоматом.

Позначення. Кінцеві автомати зручно ілюструвати за допомогою діаграм переходів, див. Приклади нижче (подвійним кружком позначені кінцеві стану):

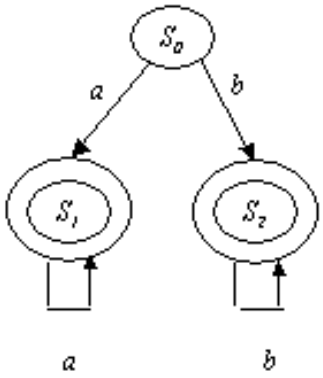
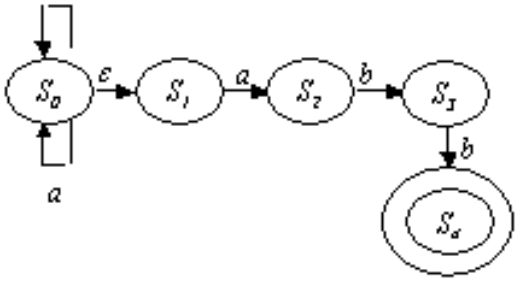


Рисунок 2.2 Приклади автоматів що розпізнають різні мови



2.3 Недетермінірованние і кінцеві автомати

Має місце наступна теорема: якщо L=L(M) для деякого недетермінірованного кінцевого автомата M, то L=L(M') для деякого детермінованого автомата M'.

Ця теорема доводиться конструктивним чином, тобто шляхом вказівки загального алгоритму побудови детермінованого автомата M', що визначає ту саму мову, що і M. Нехай $M = (Q, \Sigma, \delta, q_0, F)$; тоді ми визначимо

$$M' = (Q', \Sigma', \delta', q'_0, F'),$$

Де Q'збігається з безліччю станів автомата M, $q'_1 = q_0$, $F' = \{S \in Q | S \cap F \neq 0\}$, $\delta'(S, a) = S', S' = \{p | (q, a)$ містить p для деякого $q \in S\}$

Можна показати, що M' задає ту саму мову, що і M. Таким чином, класи мов, що задаються детермінованими і недетермінованими кінцевими автоматами, повністю збігаються. Природно, детерміновані кінцеві автомати зручніше, і в подальшому ми будемо мати справу тільки з ними.

2.3 Лексичний аналіз

Початкове текстове представлення програми не дуже придатне для роботи компілятора, тому під час аналізу програма перш за все розбивається на послідовність рядків, або, як прийнято говорити, лексем (lexeme). Безліч лексем розбивається на підмножини (лексичні класи). Лексеми потрапляють в один лексичний клас, якщо вони невиразні з точки зору синтаксичного аналізатора. Наприклад, під час синтаксичного аналізу все ідентифікатори можна вважати однаковими.

Розміри лексичних класів різні. Наприклад, лексичний клас ідентифікаторів, взагалі кажучи, нескінченний. З іншого боку, є лексичні класи, що складаються тільки з однієї лексеми, наприклад, підмножина, що складається з лексеми if. У

більшості мов програмування є такі лексичні класи: ключові слова (по одному на кожне ключове слово), ідентифікатори, рядкові літерали, числові константи. Кожному підмножині зіставляється деякий число, зване ідентифікатором лексичного класу (token) або, коротше, лексичним класом.

Приклад. Розглянемо оператор мови Pascal const pi = 3.1416; Цей оператор складається з наступних лексем:

- *const* - лексичний клас Const_LC
- *pi* - лексичний клас Identifier_LC
- = - лексичний клас Relation_LC
- *3.1416* - лексичний клас Number_LC
- *;* - лексичний клас Semicolon_LC

Лексичний аналіз різних мов програмування

Деякі мови мають особливості, істотно ускладнюють лексичний аналіз. Такі мови, як Фортран і Кобол, вимагають розміщення конструкцій мови в фіксованих позиціях вхідного рядка. Таке розміщення лексем могло бути дуже важливим при з'ясуванні коректності програми. Наприклад, при перенесенні рядки в Коболе необхідно поставити спеціальний символ в 6-й колонці, інакше наступний рядок буде розібрана неправильно. Основною тенденцією сучасних мов програмування є вільне розміщення тексту програми.

Від однієї мови до іншої варіюються правила використання символів мови, зокрема, прогалин. У деяких мовах, таких як Алгол 68 і Фортран, прогалини є значущими тільки в строкових літералах. Розглянемо популярний приклад, який ілюструє потенційну складність розпізнавання лексем у Фортране. В операторі DO 5 I = 1.25 ми не можемо визначити, що DO не є ключовим словом до тих пір, поки не зустрінемо десяткову точку.

З іншого боку, в операторі DO 5 I = 1,25 ми маємо сім лексем: ключове слово DO, мітку 5, ідентифікатор I, оператор =, константу 1, кому і константу 25. Причому, до тих пір поки ми не зустрінемо кому, ми не можемо бути впевнені в тому, що DO - це ключове слово. Щоб якось вирішити цю ситуацію, Fortran 77 дозволяє використовувати необов'язкову кому між міткою і індексом DO

оператора. Використання такої коми позвляет зробити DO оператор зрозуміліше і більш читабельним.

У більшості сучасних мов програмування ключові слова є зарезервованими, тобто їх зміст зумовлений і не може бути змінений користувачем. Якщо ключові слова не є зарезервованими, то лексичний аналіз повинен вміти розрізняти ключові слова і певні користувачем ідентифікатори. Природно, що це сильно ускладнює лексичний аналіз; наприклад, в PL / I цілком легальний наступний оператор:

IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;

При розборі такого оператора необхідно постійно перемикатися з режиму "THEN, ELSE як ключові слова" на трактування "THEN, ELSE як ідентифікатори", і назад.

У таблиці уявлень зберігається по одному примірнику всіх зовнішніх уявлень ідентифікаторів (і, можливо, також для всіх констант). Потім ідентифікатори замінюються на посилання в цю таблицю - цей процес називається згортанням.

Одна з найпростіших форм організації таблиці - це масив покажчиків на рядки. Однак при такому підході сповільнюються два основних процеси, пов'язаних з таблицею уявлень: пошук ідентифікатора в таблиці і додавання нового елемента. При цьому пошук ідентифікатора в таблиці є, напевно, найбільш масовою завданням в процесі компіляції, так як виконується для кожного використовує входження ідентифікатора. Тому хотілося б домогтися максимальної швидкодії для цієї операції.

Тому більш поширена інша форма організації таблиці уявлень - у вигляді набору хешірованних списків. Для цього вибирається деяка хеш-функція (в російській літературі іноді також звана функцією розстановки), що видає по цим ідентифікатором деяке число від 0 до H-1, де H - константа, яка називається довжиною змісту. Потім все ідентифікатори з однаковим хеш-значенням зв'язуються в список. Таким чином, для того, щоб перевірити, чи зустрічався вже

новий ідентифікатор в програмі чи ні, досить порівняти його тільки з ідентифікаторами з таблиці уявлень, що мають однакове хеш-значення.

Помічено, що більшість використань ідентифікатора знаходиться недалеко від місця його опису, тому рекомендується додавати нові ідентифікатори в початок хеш-списку, а не в кінець. Це підвищує швидкість пошуку, а також спрощує підтримку реалізації стандартних правил видимості в мовах з блочною структурою. Наприклад, перед входом в блок можна запам'ятовувати поточний стан хеш-таблиці, а потім при пошуку ідентифікатора всередині даного блоку вважати активним ідентифікатором першу змінну з такою назвою, зустрінуту в хеш-таблиці. Потім при виході з блоку необхідно відновлювати попередній стан хеш-таблиці.

2.3.2 Використання граматик для лексичного аналізу

Неважко помітити, що більшість розпізнаються лексем носять чітко задану структуру і тому виникає природне бажання застосувати до задачі лексичного аналізу теорію мов, тобто описати за допомогою будь-якого формалізму характер ланцюжків, прийнятих на вхід, а потім автоматично згенерувати по цьому опису лексичний аналізатор.

Дійсно, легко виписати, наприклад, праволінейную граматику для розпізнавання ідентифікаторів:

- *letter* -> 'a' .. 'z' | 'A' .. 'Z' | '_'
- *digit* -> '0' .. '9'
- *ident* -> letter | letter tail
- *tail* -> letter | digit | letter tail | digit tail

Як ми вже бачили, за такою граматиці можна згенерувати кінцевий автомат, який розпізнавав би правильно сформовані ідентифікатори.

Однак історично склалося, що для опису лексичних властивостей частіше використовується інший формалізм - регулярні вирази, до розгляду яких ми зараз і перейдемо.

2.3.3 Регулярні вирази

Визначимо наступні операції над множинами:

- $Q = \{x | x = yz, x \in P, y \in Q\}$
- $P^k = \begin{cases} \{\varepsilon\}, k = 0 \\ P^{k-1}P, k > 0 \end{cases}$
- $P^* = \bigcup_{i=0}^{\infty} P^i$

Тоді регулярні вирази над алфавітом T і мови, що подаються ними, можуть бути визначені наступним чином:

Символ Λ, який представляє порожня множина, є регулярним виразом, ε є регулярним виразом і представляє безліч $\{\varepsilon\}$.

Для кожного символу $a \in T$ a є регулярним виразом і представляє безліч {a}.

Якщо p і q - регулярні вирази, що представляють безлічі P і Q, то (pq), (p + q) і (p *) є регулярними виразами, що представляють безлічі PQ, $P \cup Q$ та P * відповідно.

Відзначимо, що в цьому визначенні мається на увазі наступна система пріоритетів: знак * володіє найвищим пріоритетом, за ним слідує символ конкатенації, за яким слід символ |. Пріоритети можна змінювати за допомогою використання дужок.

Приклад. Регулярний вираз 1 (0 + 1) * 1 + 1 представляє безліч ланцюжків, що починаються і закінчуються символом 1.

Згадаємо без докази, що регулярні вирази еквівалентні праволінейним граматика. Таким чином, регулярними виразами також відповідає природний клас распознавателей у вигляді кінцевих автоматів.

Приклад. Розглянута вище граматика для ідентифікатора може бути записана за допомогою наступного регулярного виразу: Letter (Letter | Digit) *.

2.4 Синтаксичний аналіз

Синтаксичний аналіз - це процес, який визначає, чи належить певна послідовність лексем мови, породжуваному граматикою. В принципі, з будь-якої граматики можна побудувати синтаксичний аналізатор, але граматики, використовувані на практиці, мають спеціальну форму. Наприклад, відомо, що для будь-якої контекстно-вільної граматики може бути побудований аналізатор, складність якого не перевищує O (n^3) для вхідного рядка довжини n, але в більшості випадків по заданому мови програмування ми можемо побудувати таку граматику, яка дозволить сконструювати і швидший аналізатор. Аналізатори реально використовуваних мов зазвичай мають лінійну складність; це досягається, наприклад, за рахунок перегляду вихідної програми зліва направо з загляданням вперед на один термінальний символ (лексичний клас).

Вхід синтаксичного аналізатора - послідовність лексем і таблиці, наприклад, таблиця зовнішніх уявлень, які є виходом лексичного аналізатора.

Вихід синтаксичного аналізатора - дерево розбору і таблиці, наприклад, таблиця ідентифікаторів і таблиця типів, які є входом для наступного перегляду компілятора (наприклад, це може бути перегляд, який здійснює контроль типів).

Відзначимо, що зовсім необов'язково, щоб фази лексічекого і синтаксичного аналізу виділялися в окремі перегляди. Зазвичай ці фази взаємодіють один з одним на одному перегляді. Основний фазою такого перегляду вважається фаза синтаксичного аналізу, при цьому синтаксичний аналізатор звертається до

лексичному аналізатору кожен раз, коли у нього з'являється потреба в черговому термінальному символі.

2.4.1 Класи синтаксичних аналізаторів

Більшість відомих методів аналізу належать одному з двох класів, один з яких об'єднує спадні (top-down) алгоритми, а інший - висхідні (bottom-up) алгоритми. Походження цих термінів пов'язане з тим, яким чином будуються вузли синтаксичного дерева: або від кореня (аксіоми граматики) до листів (термінальним символам), або від листя до кореня.

Спадні аналізатори будують висновок, починаючи від аксіоми граматики і закінчуючи ланцюжком термінальних символів. З тих, що сходять аналізаторами пов'язані так звані LL-граматики, які мають такі властивості:

- вони можуть бути проаналізовані без повернень
- перша буква L означає, що ми переглядаємо вхідні ланцюжок зліва направо (left-to-right scan)
- друга буква L означає, що будується лівий висновок ланцюжка (leftmost derivation).

Популярність низхідних аналізаторів пов'язана з тим, що ефективний спадний аналізатор досить легко може бути побудований вручну, наприклад, методом рекурсивного спуску. Крім того, LL-граматики легко узагальнюються: граматики, які не є LL-граматиками, зазвичай можуть бути проаналізовані методом рекурсивного спуску з поверненнями.

З іншого боку, висхідні аналізатори можуть аналізувати більшу кількість граматик, ніж низхідні, і тому саме для таких методів існують програми, які вміють автоматично будувати аналізатори. З висхідними аналізаторами пов'язані LR-граматики. У цьому позначенні буква L як і раніше означає, що вхідні ланцюжок проглядається зліва направо (left-to-right scan), а буква R означає, що

будується правий висновок ланцюжка (rightmost derivation). За допомогою LR-граматик можна визначити більшість використовуються в даний час мов програмування.

2.4.2 Метод рекурсивного спуску

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

Для пояснення принципів, що лежать в основі методу рекурсивного спуску, розглянемо задачу обчислення значення арифметичної формули. Будемо розглядати формули, що складаються з цілочислових значень, бінарних операцій додавання (+), віднімання (-), множення (*) і ділення без остачі (/), а також круглих дужок. Як завжди, пріоритети операцій множення і ділення рівні і їх пріоритет більше, ніж пріоритети операцій додавання і віднімання, причому пріоритети цих операцій є рівними. Будемо називати операції + і - операціями типу додавання, а операції * і / - операціями типу мно`ження. Круглі дужки використовуються для зміни стандартного порядку виконання операцій. Наше завдання полягає в написанні програми, що обчислює значення формули.

Досліджувані нами формули можна представити таким чином:

T1 + T2 + ... + Ti

де Ti - це формула виду F1i * F2i * ... * Fmi. У свою чергу, Fji - це або число, або довільна формула, укладена в круглі дужки.

Уявімо собі процес обчислення значення формули. Спочатку обчислюється F11, далі ми з'ясовуємо, яка операція слідує за F11. Якщо це операція типу множення, то ми, знаючи її лівий операнд, обчислюємо правий операнд і виконуємо операцію. Тим самим, ми отримаємо лівий операнд для можливих наступних операцій типу множення. Коли ми закінчимо обчислення формули F1 *

F2 * ... * Fn, то, можливо, побачимо далі операцію типу додавання, і процес обчислення такої формули буде аналогічний тільки що описаного процесу.

2.4.3 Леворекурсівние граматики

LL (k) -властивість накладає сильні обмеження на граматику. Іноді є можливість перетворити граматику так, щоб вийшла граматика мала властивість LL (1). Таке перетворення далеко не завжди вдається, але якщо нам вдалося отримати LL (1) -граматики, то для побудови аналізатора можна використовувати метод рекурсивного спуску без повернень.

Припустимо, що ми хочемо побудувати аналізатор мови, що породжується наступною граматикою:

$$E \to E + T | E - T | T$$

$$T \to T * F | T / F | F$$

$$F \to num | (E)$$

Зауважимо, що термінали безлічі FIRST (T) належать також безлічі FIRST (E + T). В силу цього ми не зможемо однозначно визначити послідовність викликів процедур, яку ми повинні виконати при аналізі вхідного ланцюжка. Проблема полягає в тому, що нетермінал E зустрічається на першій позиції правій частині правила, ліва частина якого також E. У такій ситуації нетермінал E називається безпосередньо леворекурсівним.

Определеніе. Нетермінал A КС-граматики G називається леворекурсівним, якщо в граматиці існує висновок A => * Aw.

Граматика, що має хоча б одне леворекурсівное правило, не може бути LL(1) -граматики. З іншого боку, відомо, що кожен КС-мова визначається хоча б однієї нелеворекурсівной граматикою.

2.5 Семантичний аналіз.

2.5.1 Ідентифікація

Ідентифікація ідентифікаторів - одне із завдань, вирішення якої необхідно для перевірки правильності використання типів.

Зрозуміло, що ми не можемо переконатися в правильності використання типів в який-небудь конструкції до тих пір, поки не визначимо типи всіх її складових частин. Наприклад, для того, щоб з'ясувати правильність оператора присвоювання ми повинні знати типи його одержувача (лівій частині) і джерела (правій частині). Для того, щоб з'ясувати, який тип ідентифікатора, що є, наприклад, одержувачем привласнення, ми повинні зрозуміти, яким чином цей ідентифікатор був оголошений в програмі.

Кожне входження ідентифікатора в програму є або визначальним, або використовують. Під визначальним входженням ідентифікатора розуміється його входження в опис, наприклад, int i. Всі інші входження є використовують, наприклад, i = 5 або i + 13.

Мета ідентифікації ідентифікаторів, визначити тип використовує входження ідентифікатора. Це завдання може бути повністю або частково вирішена на фазі синтаксичного аналізу. Все залежить від того, чи може що використовує входження ідентифікатора зустрітися в програмі до визначального входження чи ні. Якщо все визначають входження ідентифікаторів повинні бути розташовані текстуально перед використовують входженнями, то ми можемо виконати ідентифікацію на фазі синтаксичного аналізу. Якщо ж ні, то на фазі синтаксичного аналізу ми можемо обробити визначають входження ідентифікаторів і тільки на наступному перегляді тексту програми виконати власне ідентифікацію.

Незалежно від того, на якому перегляді буде виконуватися ідентифікація ідентифікаторів, при обробці визначає входження ідентифікатора необхідно запам'ятати інформацію про тип цього ідентифікатора. Це можна зробити декількома шляхами:

створювати вузол в синтаксичному дереві для конструкції "опис ідентифікатора" і запам'ятовувати інформацію про тип ідентифікатора в цьому вузлі;

створити таблицю ідентифікаторів (IdTab) і в ній запам'ятовувати інформацію про тип ідентифікатора. Чому нам може знадобитися нова таблиця? Зрозуміло, що якщо транслюється програма може мати блочну структуру, і / або мову допускає створення і використання перевантажених 1 ідентифікаторів (overloading), то в таблиці уявлень (таблиця уявлень зіставляє деякого використовуваному в компіляторі позначенню ідентифікатора його уявлення в програмі) інформацію про тип ідентифікатора зберігати не можна, оскільки в цій таблиці кожна лексема зустрічається тільки один раз. Таким чином, нам буде потрібно нова таблиця для зберігання інформації про визначальні входженнях ідентифікаторів.

Насправді, між перерахованими способами багато спільного і ми зупинимося на обговоренні другого способу.

2.5.2 Структура таблици ідентифікаторів

Наше завдання - розробити таку структуру таблиці ідентифікаторів, щоб мінімізувати пошук в ній. При цьому слід пам'ятати про те, що кожен ідентифікатор ми заносимо в таблицю уявлень, тому бажано мати прямий доступ з таблиці уявлень в таблицю ідентифікаторів для кожного ідентифікатора. Додамо до елементу таблиці уявлень поле, яке буде містити посилання в таблицю ідентифікаторів, якщо знаходиться в цьому елементі сутність є ідентифікатором.

Перейдемо до формулювання алгоритму ідентифікації, який складається з двох частин:

– перша частина алгоритму обробляє визначають входження ідентифікаторів
– друга частина алгоритму обробляє використовують входження ідентифікаторів

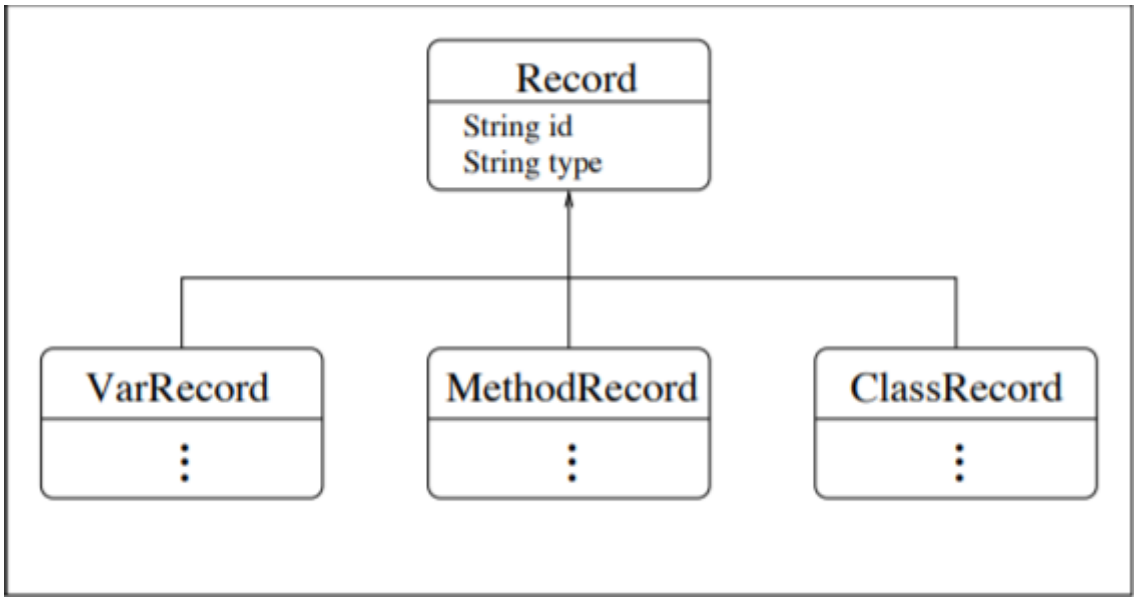


Рисунок 2.4 Структура таблиці ідентифікаторів

Обробка визначає входження ідентифікатора відбувається на фазі синтаксичного аналізу. Нехай лексичний аналізатор обробив чергову лексему, яка виявилася ідентифікатором. Лексичний аналізатор сформував структуру типу LEXEME, яка містить атрибути виділеної лексеми, такі як посилання в таблицю зовнішніх уявлень, лексичний клас і лексична марка. Далі вся ця інформація передається синтаксичному аналізатору. Припустимо, що в даний момент синтаксичний аналізатор обробляє визначальне входження ідентифікатора. Таким чином, він знає посилання в таблицю зовнішніх уявлень на оброблюваний ідентифікатор і тип ідентифікатора. Основне семантичне дію, яку повинен виконати аналізатор, полягає в занесенні інформації про ідентифікатор в таблицю ідентифікаторів. Це відбувається наступним чином: створюємо новий елемент таблиці ідентифікаторів, в поле toRepr таблиці ідентифікаторів поміщаємо rpr.

Поле toId елемента таблиці уявлень поміщаємо в поле toId нового елемента таблиці ідентифікаторів. В поле toId елемента таблиці уявлень поміщаємо посилання на новий елемент таблиці ідентифікаторів.

Відзначимо, що така організація таблиці ідентифікаторів майже повністю виключає пошук в цій таблиці, оскільки в потрібний елемент таблиці ми потрапляємо за прямим посиланням з таблиці уявлень.

Тепер обговоримо обробку використовує входження ідентифікатора, яка відбувається на фазі ідентифікації ідентифікаторів. Припустимо, що вже побудована (повністю або частково) таблиця ідентифікаторів. Нехай лексичний аналізатор обробив чергову лексему, яка виявилася ідентифікатором. Лексичний аналізатор сформував структуру типу LEXEME, яка містить атрибути виділеної лексеми, такі як посилання в таблицю зовнішніх уявлень, лексичний клас і лексична марка. Далі вся ця інформація передається фазі ідентифікації ідентифікаторів. Таким чином, ця фаза знає, що вона обробляє використовує входження ідентифікатора, причому їй відома крім іншого посилання в таблицю зовнішніх уявлень на оброблюваний ідентифікатор. Тепер для того, щоб отримати інформацію про тип ідентифікатора нам досить просто взяти її з того елемента таблиці ідентифікаторів, на який вказує поле toId поточного елемента таблиці уявлень.

2.5.3 Конструювання типів

Типи мов програмування конструюються з примітивних типів, таких як boolean, char, integer, real і void, за допомогою конструкторів типів. До примітивним типам природно віднести і тип, який не використовується при програмуванні, але вельми корисний для сигналізації про виниклу помилку в типах; це тип - invalid. Для побудови більш складних типів з примітивних зазвичай використовуються наступні конструктори:

– Масиви. Якщо T - тип, то array (I, T) - тип, що позначає тип масиву з елементами типу T і індексним безліччю I. Наприклад, опис мови Pascal: var A: array [1..10] of integer, пов'язує вираз над типами array (1..10, T) з A.

– Добуток. Якщо T1 і T2 - типи, то їх декартовій добуток T1 * T2 також є типом.

– Структури. Конструктор struct застосовується до кортежу пар (ім'я поля, тип поля). Наприклад, фрагмент програми на мові Pascal:

type row = record

address: integer;

lexeme: array [1..15] of char

end;

var table: array [1..13] of row;

Тип row будується з примітивних типів наступним чином:

struct ((address * integer) (lexeme * array (1..15, char))).

– Покажчики. Якщо T - тип, то і pointer (T) - тип, який визначає "покажчик на об'єкт типу T". Наприклад, опис мови Pascal: var p: ^ row, визначає змінну p, що має тип pointer (row).Функції.

– Якщо T1, T2 - типи, то proc (T1, T2) - тип, який визначає процедуру, типи формальних параметрів якої є T1, а тип результату - T2. Наприклад, функція mod, що обчислює залишок, має тип proc (int * int, int), а функція, певна як

function f (a, b: char): ^ integer;

має тип proc (char char, pointer (integer)).

2.5.4 Конструювання типів

Зручним шляхом подання виразів над типами є графи. Ми можемо конструювати дерева або DAG'i (орієнтовані ациклічні графи), листям яких будуть примітивні типи. наприклад:

Рисунок 2.5 – Приклад дерев типів

Використання dag'ов більш переважно, оскільки в цьому випадку відбувається іноді вельми значна економія пам'яті (замість самих типів в цьому випадку зберігається посилання на них).

Ми можемо використовувати і лінійне уявлення дерев або dag'ов, наприклад,

proc, 2, m1, m2, m2,

де m1 - покажчик в таблицю на тип pointer, integer,

а m2 - покажчик в таблицю на тип char.

Розглянемо ще один спосіб кодування типів, який був використаний в компіляторі C, розробленому Річі (D.M.Ritchie). Обмежимося трьома конструкторами типів: покажчиками, функціями і масивами: pointer (t) позначає покажчик на тип t, freturns (t) позначає функцію від деяких аргументів, яка повертає значення типу t і, нарешті, array (t) позначає масив деякої невизначеної довжини елементів типу t. Наведемо приклади типів:

- char
- freturns (char)
- pointer (freturns (char))
- array (pointer (freturns (char))).

Кожен з цих типів може бути представлений послідовністю бітів. Оскільки у нас є тільки три конструктора типу, ми можемо використовувати для кодування два біта:

- pointer 01
- array 10
- freturns 11

Примітивні типи кодуються чотирма бітами:

- boolean 0000
- char 0001
- integer 0010
- real 0011

Використовуючи такий спосіб кодування, наведені вище типи ми можемо закодувати таким чином:

- char 0000 0001
- freturns (char) 0011 0001
- pointer (freturns (char)) 0111 0001
- array(pointer(freturns(char))) 0011 10001

Тепер повернемося до обговорення структури таблиці ідентифікаторів. Одне з полів (toMode) ми збиралися використовувати для визначення типу ідентифікатора. Визначивши уявлення типів, які можуть з'явитися в програмі, ми можемо тепер зафіксувати, що поле toMode в - це або покажчик на дерево або dag, або посилання в таблицю типів ModeTab, що зберігає лінійне уявлення типів.

2.5.5 Контроль типів

Якщо контроль типів здійснюється під час трансляції програми, то ми говоримо про статичному контролі типів, в іншому випадку, тобто якщо контроль типів проводиться під час виконання об'єктної програми, ми говоримо про

динамічному контролі типів. В принципі, контроль типів завжди може виконуватися динамічно, якщо в об'єктному коді разом зі значенням буде розміщуватися і тип цього значення. Зрозуміло, що динамічний контроль типів призводить до збільшення розміру та часу виконання об'єктної програми і зменшення її надійності. Мова програмування називається мовою зі статичним контролем типів або строго універсальна мова (strongly typed language), якщо тип будь-якого виразу може бути визначений під час трансляції, тобто якщо можна гарантувати, що об'єктна програма виконується без типових помилок. До числа строго типізованих мов відноситься, наприклад, Pascal. Однак навіть для такої мови як Pascal деякі перевірки можуть бути виконані тільки динамічно. наприклад,

```
table: array [0..255] of char;
i: integer;
```

Компілятор не може гарантувати, що при виконанні конструкції table [i] значення і дійсно буде не менше нуля і не більше 255. В деяких ситуаціях здійснити таку перевірку може допомогти техніка, подібна data flow analysis, але далеко не завжди. Зрозуміло, що насправді цей приклад демонструє ситуацію загальну для більшості мов програмування, тобто тут мова йде про контроль індексів вирізки. Звичайно, в більшості випадків така перевірка виконується динамічно.

2.5.6 Атрибутное дерево разбора

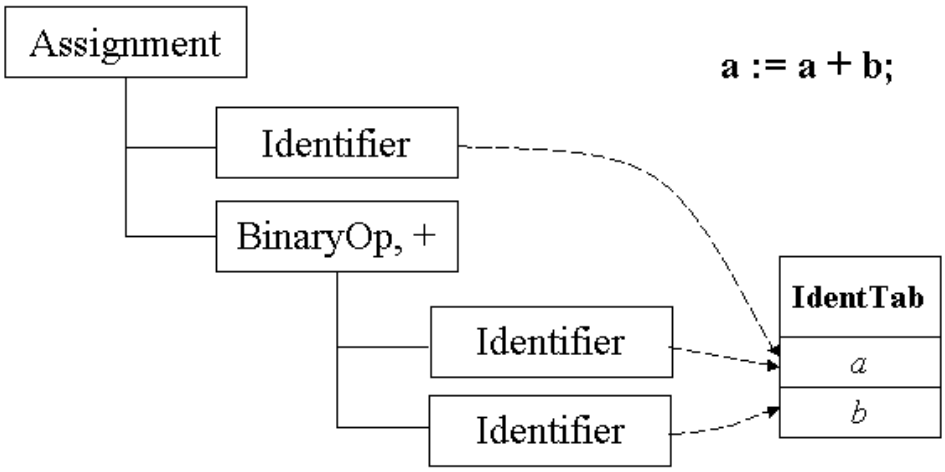


Рисунок 2.6 – Приклад атрибутнуго дерева розбору

Атрибутне дерево розбору є, напевно, найпоширенішою формою організації внутрішнього представлення програми. При такому підході кожна вихідна конструкція мови представляється у вигляді вузла дерева, що містить посилання на всі можливі елементи цієї конструкції (природно, кожний окремий елемент теж може мати складну структуру і, таким чином, також може бути піддерево). Крім того, кожен вузол дерева може навантажувати додатковими атрибутами, такими, як посилання в таблиці уявлень або таблиці ідентифікаторів. У підсумку, вся програма представляється у вигляді єдиного дерева розбору.

На слайді як приклад наведено атрибутного дерево розбору, породжене по наступному оператору вихідного мови a: = a + b; Відзначимо, що форма подання дерева, використана на слайді, є типовою при компіляції, так як дозволяє зобразити практично як завгодно складні дерева на екрані комп'ютера.

Дерева розбору привабливі насамперед своєю гнучкістю і можливістю використання в самих різних етапах компіляції - їх можна спроектувати таким чином, щоб вони мало залежали від вихідної мови і цільової платформи. Дерева розбору легко будувати під час аналізу вихідної програми, а всі наступні перегляди компілятора можуть бути реалізовані у вигляді самостійних обходів

цього дерева. Крім того, деякі перегляди, такі, як оптимізація програми, найзручніше виконувати саме над деревами розбору.

2.5.6 Польський запис

Польська запис була запропонована польським логіком Лукасевичем. У цій формі записи всі оператори безпосередньо передують операндам. Так, звичайне вираз (a + b) * (cd) в польській записи може бути представлено як * + ab-cd.

Таку форму записи називають також префіксной. Аналогічним чином вводиться зворотна або Постфіксний польський запис, в якій всі оператори виписуються після операндів. Скажімо, приклад, наведений вище, в зворотної польської записи буде записаний у такий спосіб: ab + cd- *. Для уявлення унарних операцій в польській записи можна скористатися еквівалентними виразами, які використовують бінарні операції, як в наступному прикладі: -b -> 0 - b, а можна ввести новий знак операції, скажімо, @b. Польська запис може бути поширена не тільки на арифметичні вирази, а й на інші конструкції мови. Наприклад, оператор a: = a + b; може бути записаний в польській записи як: = a + ab, а умовний оператор if <expr> then <instr1> else <instr2> може бути записаний як наступна послідовність операторів:

<Expr> <c1> BZ <instr1> <c2> BR <instr2>,

де c1 вказує на першу інструкцію <instr2>, а c2 - на першу інструкцію, наступну за <instr2>, BR - безумовний перехід на адресу <c2>, а BZ - перехід на <c1> за умови рівності нулю виразу <expr1>.

Користуючись такою термінологією, ми можемо називати традиційну форму запису виразів інфіксной, так як в ній знаки операцій розташовані між операндами. Зрозуміло, що будь-який вираз може бути переведено з інфіксной форми в польську запис і навпаки. Польська запис чудова тим, що при її використанні зникає потреба в пріоритетах операцій - кожна операція

виконується в порядку появи в вихідної ланцюжку (хоча очевидно, що пріоритет операцій необхідно враховувати при перетвореннях з інфіксной форми).

Польська запис (особливо зворотна) дуже добре накладається на стекову модель: кожен зустрінутий операнд завантажується в стек, а операції здійснюються лише на вершині стека: кожна операція знімає необхідну кількість операндів з вершини стека і кладе на стек свій результат. Саме така модель використовується в MSIL для реалізації більшості операцій.

2.6 Управління пам'яттю

Будь-який компілятор є всього лише одним з чисельних додатків, що працюють під управлінням даної операційної системи, і тому при зверненні до системних ресурсів компілятор змушений покладатися на надані стандартні функції і примітиви. Таким чином, управління пам'яттю з точки зору компілятора істотно обмежена можливостями цільової архітектури і операційної системи. З іншого боку, велика кількість рішень з управління пам'яттю робиться вже на етапі створення мови програмування (докладніше про це нижче).

Отже, управління пам'яттю при розробці компілятора є питанням одночасно і машинно-залежним, і мовно-залежним. У зв'язку з цим розробник компілятора повинен знайти найбільш ефективне відображення засобів управління пам'яттю, пропонованих мовою програмування, на задану апаратуру і ОС.

При цьому найчастіше виникає ситуація, коли доводиться миритися з існуванням відразу декількох паралельних механізмів управління пам'яттю. Навіть в тих мовах, в яких програміст має можливість явного управління пам'яттю, це ніяк не скасовує стандартних системних механізмів, так як розробник компілятора зобов'язаний забезпечити коректну роботу багатьох елементів програми, прихованих від кінцевого програміста (наприклад, компілятор повинен виділяти пам'ять під саму оттранслировать програму,

системні програми часу виконання, точки входу і повернення з підпрограм, тимчасову пам'ять для обчислення виразів і т.п.).

2.6.1 Основні фази роботи з пам'яттю

Пам'ять - це один з найважливіших ресурсів комп'ютера. Так як сучасні мови програмування не зобов'язують програміста працювати безпосередньо з фізичними осередками пам'яті, на компілятор мови програмування покладається відповідальність за забезпечення доступу до фізичної пам'яті, її розподіл і утилізацію. В якості ресурсу можуть виступати найрізноманітніші логічні і фізичні одиниці: звичайні змінні примітивного типу, масиви, структури, об'єкти, файли і т.д. З усіма цими об'єктами необхідно працювати і, отже, забезпечити виділення пам'яті під пов'язані з ними змінні в програмах.

Для цього компілятор повинен послідовно виконати наступні завдання:

- виділити пам'ять під змінну;
- форматувати виділену пам'ять деяким початковим значенням;
- надати програмісту можливість використання цієї пам'яті;
- як тільки пам'ять перестає використовуватися, необхідно її звільнити (можливо, попередньо очистивши)
- нарешті, необхідно забезпечити можливість подальшого повторного використання звільнення пам'яті.

З точки зору програміста, описана вище схема здається дуже простою. Проте, акуратно реалізувати ці дії в компіляторі і домогтися коректної роботи програм, що використовують цей механізм, досить складно через різні проблеми. Про це свідчить і той факт, що більшість помилок, що виникають в сучасних програмах, пов'язано з некоректним використанням пам'яті. На наступних слайдах ми розглянемо наібоелее поширені проблеми, пов'язані з управлінням пам'яттю.

2.6.2 Проблеми управління пам'яттю

Найбільша неприємність управління пам'яттю полягає в тому, що пам'ять не нескінченна і тому доводиться постійно враховувати можливість вичерпання вільної пам'яті. Звичайно, ця проблема поступово втрачає свою гостроту в зв'язку з постійним здешевленням апаратури комп'ютерів, але враховувати цю небезпеку доведеться завжди.

Інша проблема виникає в тих випадках, коли мова програмування надає програмістові явний механізм управління пам'яттю (такий, як malloc / free в мові C або new / delete в C ++). У цих випадках компілятор не може гарантувати правильність роботи оброблюваних їм програм і ця відповідальність покладається на програміста. На жаль, люди значно менш надійні, мають тенденцію помилятися і навіть повторювати свої помилки, а в багатьох випадках і просто ігнорують надані їм механізми. Тому при такому підході зазвичай виникає безліч помилок, що, в свою чергу, веде до необхідності кропіткої налагодження програм і суттєво ускладнює роботу програміста.

Особлива неприємність помилок, що виникають при некоректній роботі з пам'яттю, полягає в тому, що ці помилки щодо непередбачувані, можуть виникати у вкрай рідкісних випадках, можуть залежати від порядку виконання попередніх операторів програми і тому значно складніше у виявленні і виправленні, ніж звичайні "алгоритмічні" помилки. Наприклад, типовою помилкою є виділення ресурсу лише в одній з можливих гілок умовного оператора з подальшим безумовним використанням або звільненням цього ресурсу в наступних частинах програми.

Однак в деяких випадках важко обійтися без участі програміста. Наприклад, звільнення ресурсів, асоційованих з будь-якими зовнішніми сутностями (файлами на диску, записами баз даних, мережевими з'єднаннями і т.п.), як правило, вимагає явних операцій по закриттю. У таких випадках просте звільнення пам'яті,

займаної змінної в програмі, вирішить тільки частину проблеми, так як після цього файл або запис в базі даних залишаться недоступними для інших додатків.

З точки зору програміста, пам'ять стає вільною як тільки виконується оператор явного звільнення пам'яті (free / delete) або в момент закінчення часу життя останньої змінної, що використовує дану область пам'яті. Ці операції, що роблять структуру даних логічно недоступною, називаються знищенням пам'яті. Однак з точки зору розробника компілятора в цей момент вся робота тільки починається.

У випадку з явним звільненням пам'яті все більш-менш очевидно, хоча, як ми бачили вище, і пов'язано з проблемами для програміста. Але все одно більшість змінних звільняється автоматично (в кінці блоку, процедури і т.д.). Тому момент закінчення використання пам'яті ще необхідно відстежити, тобто зрозуміти, що даний фрагмент пам'яті дійсно ніхто більше не використовує. Це не завжди тривіально, так як в програмі може існувати кілька елементів, пов'язаних з даною галуззю пам'яті. У таких випадках говорять про існування різних шляхів доступу до структури. Найбільш простий приклад - це два покажчика, що вказують на один і той же адресу. Інший приклад - передача масиву параметром в процедуру. У загальному випадку відстеження всіх шляхів доступу до структури важко піддається реалізації і дорого.

Потім звільнену пам'ять необхідно повернути системі як вільну - утилізувати. Відзначимо, що операції знищення пам'яті та утилізації можуть бути сильно рознесені за часом. Більш того, в більшості мов у програміста немає можливості форсувати утилізацію даного конкретного об'єкта (хоча в C # така операція передбачена для великих об'єктів).

Зрозуміло, що утилізація пам'яті сильно утруднена через проблеми з визначенням єдиності доступу до знищуваної області пам'яті. На наступному слайді ми розглянемо проблеми, які можуть виникнути при різних помилках в цьому процесі.

2.6.3 Статична і динамічна пам'ять

При створенні компілятора необхідно розрізняти два важливих класу інформації про програму:

Статична інформація, тобто інформація, відома під час компіляції

Динамічна інформація, тобто відомості, невідомі під час компіляції, але які стануть відомі під час виконання програми

Наприклад, значення констант в строго типізованих мовах відомі вже під час компіляції, в той час як значення змінних в загальному випадку стають відомими вже тільки під час виконання програми. Статично ми знаємо кількість гілок в операторі switch, але визначити, яка з них виконається, ми зможемо вже тільки під час виконання програми.

У додатку до управління пам'яттю поділ всієї інформації на статичну і динамічну дозволяє визначити, яким механізмом розподілу пам'яті необхідно користуватися для тієї або змінної, структури або процедури. Наприклад, розмір пам'яті, необхідної під прості змінні, можна обчислити (і, відповідно, виділити необхідну пам'ять) вже під час компіляції, а ось пам'ять, запитувану користувачем з розміром, заданим за допомогою змінної, доведеться виділяти вже під час виконання програми. Зрозуміло, що статичний розподіл пам'яті за інших рівних умов краще ("дешевше").

Особливо цікаві "прикордонні" випадки, такі, як виділення пам'яті під масиви. Справа в тому, що розмір пам'яті, необхідної під масиви фіксованого розміру, в більшості сучасних мовах програмування можна порахувати статично. І тим не менше, іноді розподіл пам'яті під масиви відкладають на етап виконання програми. Це може бути осмислено, наприклад, для мов, які дозволяють опис динамічних масивів, тобто масивів з кордоном, невідомої під час компіляції. До таких мов відносяться Алгол 68, PL / I, C #. У цьому випадку механізм розподілу пам'яті буде однаковим для всіх масивів. А ось в Паскалі або C / C ++ пам'ять під масиви завжди можна виділяти статично.

2.7 Оптимізація

Під оптимізацією розуміють послідовність еквівалентних перетворень вихідної програми, що зменшують її вартість. Як набір, так і порядок виконання цих перетворень залежать від того, що вважається вартістю програми. В якості такої вартості можуть виступати, наприклад, середній час роботи, обсяг коду і т.д. Ефективність оптимізації також залежить від відносини еквівалентності і від розміру ділянки економії, на якому ця оптимізація проводиться (зазвичай оптимізованої програмі дозволяється мати велику область визначення, ніж вихідної). За рахунок оптимізації неможливо домогтися істотного поліпшення алгоритму програми, можна тільки говорити про поліпшення реалізації цього алгоритму. У вдалих випадках оптимізація може прискорити програму в кілька разів. Корисність застосування оптимізації обумовлена наступними причинами:

- поширенням мов надвисокого рівня, мов специфікації і систем проектування. Як правило, подібні системи породжують програми на деякій мові високого рівня. В цьому випадку оптимізація може нейтралізувати надмірність такого породження, наблизивши якість згенерованої програми до ручного програмування.
- необхідністю підтримки і супроводу готової програми. Такі вимоги часто призводять до того, що програмісти використовують не найефективніші рішення з метою поліпшення наочності або легкості супроводу програм (дуже часто міркування ефективності і сопровождаемости суперечать один одному). У той же час недоліки ефективності можуть бути виправлені оптимізатором при генерації остаточної програми.
- посиленням контролю семантичних помилок. Такі помилки вимагають спеціального аналізу вихідної програми, який може бути побічним результатом дій оптимізатора.

2.7.1 Види оптимізації

Існує багато різних класифікацій оптимізують перетворень. Тут ми розглянемо класифікації за рівнем представлення програми і за розміром ділянки економії.

Залежно від рівня представлення програми розрізняють наступні види оптимізації:

- Оптимізацію на рівні вихідної мови. При цьому в результаті трансформації виходить програма, записана в тій же самій мові.
- Машинно-незалежну оптимізацію. В цьому випадку перетворенню піддається програма на рівні машинно-незалежного проміжного представлення, загального для групи вхідних або машинних мов.
- Машинно-залежну оптимізацію, тобто оптимізацію на рівні машинної мови.

З точки зору ефективності найкращою є машинно-залежна оптимізація, оскільки саме з її допомогою можна врахувати особливості конкретної обчислювальної середовища, однак машинно-залежний оптимізатор нестерпний. З іншого боку, перетворення програми на рівні вихідної мови дозволяє отримати більш ефективну програму, яка допускає подальший розвиток і супровід. Нарешті, машинно-незалежна оптимізація на рівні проміжного представлення є компромісом між цими двома крайніми випадками.

Іншим важливим для якості оптимізації міркуванням є також розмір ділянки економії, тобто того фрагмента програми, в рамках якого проводиться оптимизирующее перетворення. Чим більше ділянку економії, тим більше інформації про властивості програми є оптимізатора.

3 МОДЕЛЮВАННЯ СИСТЕМИ ДЛЯ РОЗРОБКИ КОМПІЛЯТОРА

3.1 Проектування системи

В результаті проведеного аналізу було обрано багатопрохідний схема перегляду компілятора. На кожному етапі (лексичний аналіз, синтаксичний аналіз, формування проміжного коду, формування ассемблерного коду) відбувається новий перегляд (прохід) за програмою, представленою в різному вигляді. На першому етапі (сканер) - у вигляді тексту програми, на другому (парсер) - у вигляді кодів лексем, на третьому - дерево граматичного розбору, на четвертому - таблиця проміжного коду. Це зроблено для наочного поетапного процесу компіляції і можливості роботи з внутрішнім поданням програми.

При виборі мови високого рівня, як вхідна мова для аналізу була прийнята мова miniJava, заснована на спрощеному варіанті мови Java. Мова Java є поширеним, зрозумілим і простим для сприйняття, до того ж його структури досить зручні для розбору.

Проаналізовані і описані в попередньому розділі моделі і методи конструювання компіляторів реалізовані за допомогою технології LLVM, з використанням мови Java.

3.1.1 Технологія ANTLR

ANTLR (від англ. ANother Tool for Language Recognition - «ще один засіб розпізнавання мов») - генератор низхідних аналізаторів для формальних мов. ANTLR перетворює контекстно-вільну граматику у вигляді РБНФ в програму на C ++, Java, C #, JavaScript, Go, Swift, Python. Використовується для розробки компіляторів, інтерпретаторів і трансляторів.

Переваги ANTLR перед аналогами:

- Вільне програмне забезпечення.
- Використання єдиної нотації для опису лексичних і синтаксичних аналізаторів.
- Застосування спадного, а не висхідного аналізу.
- Зручність роботи з абстрактним синтаксичним деревом.
- Надання повідомлень про помилки і відновлення після них.
- Наявність візуальних середовищ розробки (ANTLR Works, ANTLR Studio, плагінів до Eclipse і IntellliJ IDEA), які дозволяють створювати і налагоджувати граматики, підтримують підсвічування синтаксису, автодоповнення, візуальне Отображаніе граматик, що будується в реальному часі під час введення, відладчик, рефакторинг.

ANTLR широко використовується в великих відомих продуктах:

- Реалізація мов програмування Groovy, Jython, Processing, Apex
- Аналіз мов запитів в системах баз даних Hibernate HQL, Cassandra, Hive, Pig
- Аналіз мов програмування в середовищах розробки: NetBeans C ++, Oracle SQL Developer IDE, IntelliJ IDEA Clion.

Коротка довідка елементів мови ANTLR:

- (...) підправити
- (...) * повторення підправила 0 або більше разів
- (...) + Повторення підправила 1 або більше разів
- (...)? підправити, може бути відсутнім
- {...} семантичні дії (на мові, що використовується в якості вихідної - напр., Java)
- [...] параметри правила
- | оператор альтернативи
- .. оператор діапазону
- ~ заперечення
- . будь-який символ
- = присвоювання
- : Мітка початку правила

– ; мітка кінця правила

Приклад найпростішої програми

```
grammar T; // ім'я граматики, має збігатися з назвою файлу
// нетермінальні символи:
msg: 'name' ID ';'
{
System.out.println ("Hello," + $ ID.text + "!");
};
// термінальні символи
ID: 'a' .. 'z' +; // довільне (але> = 1) кількість букв
WS: (" | '\ n' | '\ r') + {$ channel = HIDDEN;}; // пробіл, перенесення рядка, табуляція
```

3.1.2 Мова Java

Java - об'єктно-орієнтована мова програмування, що розробляється компанією Sun Microsystems з 1991 року і офіційно випущений 23 травня 1995 року. Спочатку нову мову програмування називався Oak (James Gosling) і розроблявся для побутової електроніки, але згодом був перейменований в Java і став використовуватися для написання додатків і серверного програмного забезпечення.

Програми на Java можуть бути трансльовані в байт-код, що виконується на віртуальній java-машині (JVM) - програмою, обробній байт-код і передавальної інструкції обладнанню, як інтерпретатор, але з тією відмінністю, що байт-код, на відміну від тексту, обробляється значно швидше.

Мова Java зародився як частина проекту створення передового програмного забезпечення для різних побутових приладів. Реалізація проекту була почата на мові C ++, але незабаром виник ряд проблем, найкращим засобом боротьби з

якими була зміна самого інструмента - мови програмування. Стало очевидним, що необхідний платформо-незалежний мова програмування, що дозволяє створювати програми, які не доводилося б компілювати окремо для кожної архітектури і можна було б використовувати на різних процесорах під різними операційними системами.

Мова Java потрібен для створення інтерактивних продуктів для мережі Internet. Фактично, більшість архітектурних рішень, прийнятих при створенні Java, було продиктовано бажанням надати синтаксис, схожий з C і C ++. В Java використовуються практично ідентичні угоди для оголошення змінних, передачі параметрів, операторів і для управління потоком виконанням коду. В Java додані всі хороші риси C ++.

Ключові елементи об'єдналися в технології мови Java

Java вивільняє міць об'єктно-орієнтованої розробки додатків, поєднуючи простий і знайомий синтаксис з надійним і зручним в роботі середовищем розробки. Це дозволяє широкому колу програмістів швидко створювати нові програми і нові аплети

Java надає програмісту багатий набір класів об'єктів для ясного абстрагування багатьох системних функцій, використовуваних при роботі з вікнами, мережею і для введення-виведення. Ключова риса цих класів полягає в тому, що вони забезпечують створення незалежних від використовуваної платформи абстракцій для широкого спектра системних інтерфейсів.

3.1.3 Мова програмування MiniJava

MiniJava є підмножиною мови програмування Java.

Кожна програма MiniJava має окремий MainClass, який містить один головний метод, що містить одну операцію. Метою MainClass є почати виконання, створивши об'єкт іншого класу. Таким чином, найпопулярніші уявлення в граматиці MiniJava виглядають так:

MiniJavaProgram --> MainClass (ClassDeclaration)*

MainClass --> "class" Identifier "{" MainMethod "}"

MainMethod --> "public" "static" "void" "main" "(" "String" ("[" "]" | "...") Identifier ")" "{" Statement "}"

Оголошення класів ніколи не мають модифікаторів доступу, вони завжди починаються з класу ключових слів. Оголошення класу (типу ClassDeclaration) містять ряд декларацій поля (без модифікаторів доступу, без ініціалізації), за якими йде число оголошень методів (відсутні модифікатори доступу або відкриті, як у прикладі вище). Немає конструкторів. Немає спадкування. Немає перевантаження методів і не змінює перемін.

Методи в MiniJava мають тіло з трьома послідовними розділами. Вони завжди починаються з числа оголошень змінної, за якими слідує ряд операторів, за якими слідує одна операція повернення. Усі змінні, що використовуються в методі, повинні бути оголошені в розділі оголошення змінної.

MiniJava містить лише шість різних типів операцій:

- Statement blocks (e.g. "{" Statement* "}")
- While statements (+ break and continue)
- If statements
- Assign statements
- Array assign statements (e.g. a[5] = ...)
- Print statements (e.g. "System.out.println" "(" Expression ")" ";")

Висновки друку включені, щоб дозволити програмам MiniJava друкувати інформацію без залучення будь-яких бібліотечних класів. Заява print може друкувати тільки вирази типів int, boolean, char і String.

Реалізована підтримкуа операцій break і continue тільки у звичайній немеченой формі, не підтримуэ позначену форму, таку як "continue label1;".

MiniJava включає підтримку типових посилань (Identifier), цілих чисел (int), символів (char), булевих (boolean), цілочисельних масивів (int []) і рядків (String).

Цілочисельний масив - це єдиний тип масивів, дозволений у MiniJava.

Рядки (подібні масивам) розглядаються як напівпримитивні типи лише з декількома дозволеними операціями: конкатенація рядків (+), перевірка довжини (.length ()) і доступ до символів (наприклад, .charAt (7)).

MiniJava має повну підтримку цілочисельних арифметичних і булевих виразів із залученням таких операторів:

+, -, *, /, <, ==, &&, ||,!

Дужки, напр. a * (3 + b), також дозволені в арифметичних і булевих виразах.

Єдиними застосовними операторами для символів є ==, <, і конкатенація з рядками.

Виклики, напр. this.ComputeFac (n-1) завжди має явне цільове вираз (це). Ланцюги виклику, такі як a.m (...). N (..) дозволені.

Приклад типової програми MiniJava:

```
class Factorial{
   public static void main(String[] a){
       System.out.println(new Fac().ComputeFac(10));
   }
}
class Fac {
   public int ComputeFac(int num){
       int num_aux ;
       if (num < 1)
          num_aux = 1 ;
```

```
        else
            num_aux = num * (this.ComputeFac(num-1)) ;
        return num_aux ;
    }
}
```

3.2 Конструювання компілятора для мови програмування

3.2.1 Побудова дерева синтаксису

Зараз існує достатня кількість генераторів компіляторів, таких як ANTLR, GPPG, Coco / R, GOLD Parsing System і безліч інших.

У даній роботі був використаний вище опіссанний ANTLR, оскільки він зручний для написання граматики, має графічну оболонку з текстовим редактором і дозволяє проводити візуалізацію абстрактного синтаксичного дерева.

Генератор компіляторів дозволяє отримати набір токенів і абстрактне синтаксичне дерево (Abstract Syntax Tree, AST), відповідне граматиці.

З опису граматики(Рисунок) в формі Бекуса-Наура був складений файл граматики, призначений для обробки ANTLR. Привівши граматику до LL-виду, видаливши ліву рекурсію і вписавши у відповідні частини створення символів граматики на Java, був підготовлений файл для подальшого процесу кодогенераціі (файл граматики MiniJava.g4 генеруєте в директорії проекту).

```
Program ::= (ClassDeclaration)* eot

ClassDeclaration ::=
        class id {
          (FieldDeclaration | MethodDeclaration)*
        }

FieldDeclaration ::= Declarators id;

MethodDeclaration ::=
        Declarators id (ParameterList? ) {
          Statement* (return  Expression ;)?
        }

Declarators ::= (public | private)? static? Type

Type ::=  PrimType | ClassType | ArrType

PrimType ::=   int | boolean  | void

ClassType ::= id

ArrType ::=  ( int | ClassType ) []

ParameterList ::= Type id (, Type id)*

ArgumentList ::= Expression ( ,  Expression)*

Reference ::= ( this | id) ( . id)*

Statement ::=
          { Statement* }
        | Type id = Expression ;
        | Reference ([ Expression ])? = Expression ;
        | Reference ( ArgumentList? ) ;
        | if ( Expression ) Statement (else Statement)?
        | while ( Expression ) Statement

Expression ::=
          Reference ( [ Expression ] )?
        | Reference ( ArgumentList? )
        | unop Expression
        | Expression binop Expression
        | ( Expression )
        | num | true | false
        | new (id ( ) | int [ Expression ]  | id [ Expression ] )
```

Рисунок 3.1 – Граматика язика miniJava

Тут граматика представлена правилами виведення нетерміналів в ланцюжки мови.

Термінал - це кінцевий символ граматики. Нетермінал - це символ граматики, для якого знайдеться правило виведення, що переводить його в ланцюжок з комбінацій терміналів і / або нетерміналів, тобто $\alpha \rightarrow \beta$, де $\alpha \in V$, $\beta \in (N \cup V)^*$. N - алфавіт терміналів, V - алфавіт нетерміналів.

Всі термінали в описі граматики позначені послідовністю символів в подвійних лапках, нетермінали - послідовністю символів, а також в кутових лапках. Аксіома граматики - нетермінал Program.

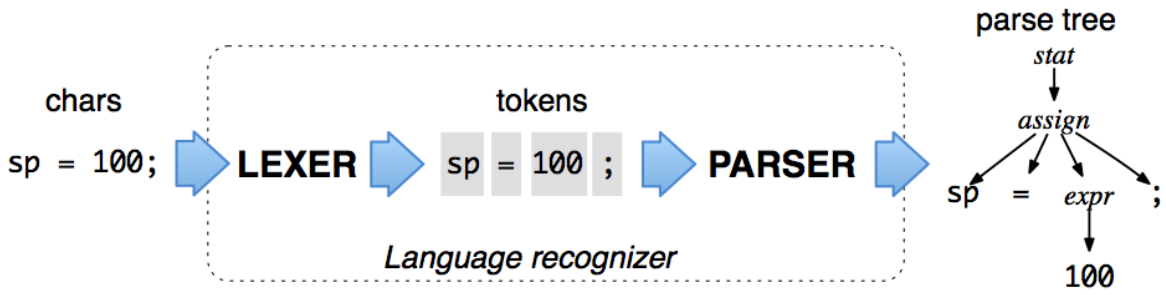


Рисунок 3.2 – Візуалізація роботи лексера та парсера

3.2.2 Конструювання таблиці символів

Таблиця символів являє собою структуру даних, в яких зберігається інформація, що стосується окремих ідентифікаторів. Вона організована так, що легко додати та шукати інформацію в таблицю символів.

Який ідентифікатор представив запису, яка зберігається в таблицях символів. Для різних типів ідентифікаторів (наприклад, класів, методів і перемінних) використовуються різні типи записів. Реалістична з допомогою ієрархии типов записей. Так як MiniJava не підтримує перевантаження методів,

ми можемо безпечно використовувати ім'я методу в якості ідентифікатора методу в таблиці / записі символів.

Який ідентифікатор в програмі MiniJava має область видимості, в якій він працює. Наприклад, всі ідентифікатори класу мають глобальну видимість, тоді як перехідні, певні й інші способи. Таблица символов написана с учётом областей действий.

Дерево областей, где каждая область являє собою структуру Карта від символів (ідентифікаторів) до записів. Ви можете скористатися будь-яким підходом, якщо він достатньо швидкий (по крайней мере, швидше, ніж O (N), де N - це число різних областей).

Здійснивши таблицю символів, ми обходимо полученое синтаксичне дерево і додаємо нову запису в таблицю символів для кожного оголошення, найменування програми. На даний момент ми маємо справу тільки з деклараціями. Використання і визначення будуть розглянуті в наступному семантичному аналізі. Тем не менш, конструкція таблиць символів є хорошим місцем для зв'язку кожного ідентифікатора з типом. Наступне, ми додамо інформацію про типи в каждую нову створену запись (тип класса - це клас, тип методу - його тип повертаємого значення).

Таблиця символів відповідно до наведених вище описами має наступний інтерфейс:

```
public interface SymbolTable {
  public void enterScope();
  public void exitScope();
  public void put(Symbol key, Record item);
  public Record lookup(Symbol key);
  public void printTable();         // Used during development
}
```

Був реалізований паттерн "Відвідувач" (за допомогою згенерованих базових служб ANTLR), щоб створити таблицю символів для одного з проміжних

сторінок. Реалізація таблиць символів через паттерн "слухача" є життєздатною опцією (за допомогою згенерованого слухача в цьому випадку).

Було реалізовано метод друку, який відображає вміст (назва і тип кожного з елементів) і структуру таблиць символів. Дана реалізація аналізує всю початкову програму і повідомляє про повну інформацію про семантичні помилки. Реализован счетчик виявлених помилок (ноль виявлений помилок, що вказує на дію програмної системи).

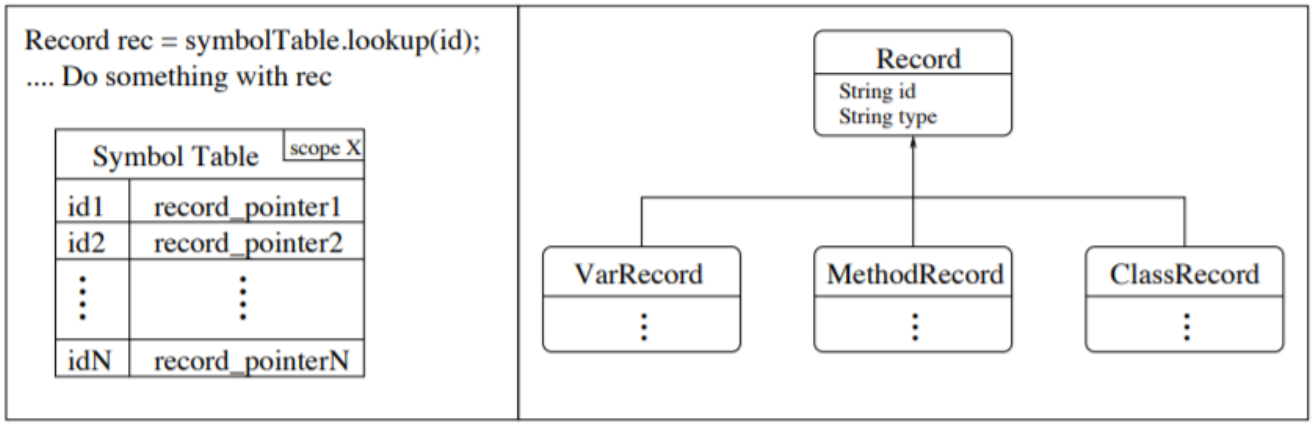


Рисунок 3.3 - Структура таблици символів

3.2.3 Семантичний аналіз

Семантичний аналізатор виконує наступні основні дії:

- перевірка соблючения вхідної програми;
- доповнення внутрішніх показників програм в компіляторі операторів і дій, явно передбачених семантикою вхідного мови;
- перевірка елементіврних семантичних (смыслових) норм мови программирования, напрямую не пов'язаних з вхідним языком.

Програма згортання вхідної програми полягає в тому, що первинні мови вхідного мови полягають у виборі програмного забезпечення з вимогами

семантичних вхідних мов програмування. Имен их в первую очередь проверяет семантичний аналізатор.

Проверяемые соглашения:

- каждая метка, на якій є посилання;
- кожен ідентифікатор повинен бути описаний один раз, і ні один ідентифікатор не може бути описаний більш ніж один раз (з описаним блоком структури);
- все операнды в выражениях і операціях повинні мати типи, \ t
- типи перемінних в вираженнях повинні бути узгодженими між собою;
- при виході процедур і функцій число і типи фактичних параметрів повинні бути узгоджені з числом і типами формальних параметрів.

3.2.4 Перевірка типу “Type-checking”

Примітивні типи в MiniJava - це int і boolean; всі інші типи - це цілочисельний масив, написані int [] або імена класів. Для простоти ми вибираємо, що кожен тип представлятимемо як рядок, а не як символ; це дозволяє нам перевіряти рівність типу, виконуючи порівняння рядків.

Перевірка типу програми MiniJava здійснюється у два етапи. По-перше, ми будуємо таблицю символів, а потім вводимо перевірку операторів і виразів. Під час другого етапу для кожного знайденого ідентифікатора розглядається таблиця символів. Зручно використовувати дві фази, оскільки в Java і MiniJava класи взаємно рекурсивні. Якщо ми спробували виконати перевірку типу в одній фазі, то нам, можливо, знадобиться набрати перевірку виклику методу, який ще не введено в таблицю символів. Щоб уникнути таких ситуацій, ми використовуємо підхід з двома фазами.

Перший етап перевірки типу може бути реалізований відвідувачем, який відвідує вузли в синтезаторі MiniJava і створює таблицю символів. Наприклад, метод відвідування в в нижче представленому коді обробляє декларації змінних. Він додасть ім'я змінної та тип до структури даних для поточного класу, який пізніше буде додано до таблиці символів. Зверніть увагу, що метод візиту перевіряє, чи змінна оголошена більше одного разу, і якщо так, то виводить відповідне повідомлення про помилку.

```
class ErrorMsg {
  boolean anyErrors;
  void complain(String msg) {
    anyErrors = true;
    System.out.println(msg);
  }
```

```
}
public void visit(VarDecl n) {

  Type t = n.t.accept(this);
  String id = n.i.toString();

  if (currMethod == null) {
     if (!currClass.addVar(id,t))
        error.complain(id + "is already defined in " + currClass.getId());
  } else if (!currMethod.addVar(id,t))
        error.complain(id + "is already defined in "
            + currClass.getId() + "." + currMethod.getId());
}
```

Друга фаза типу перевірки може бути реалізована відвідувачем, який перевіряє тип всіх операторів і виразів. Тип результату кожного методу відвідування - String, для представлення типів MiniJava. Ідея полягає в тому, що коли відвідувач відвідує вираз, він повертає тип цього виразу. Якщо вираз не перевіряє тип, то перевірка типу завершується повідомленням про помилку.

Візьмемо простий випадок: доповнення виразу e1 + e2. У MiniJava обидва операнда повинні бути цілими числами (перевірка типу має перевірити це), а результат буде цілим (перевірка типу поверне цей тип). Метод візиту для додавання легко реалізувати.

ВИСНОВКИ

В результаті написання атестаційної роботи був проведений аналіз предметної галузі, досліджені основні методи, моделі, інструменти реалізаії, концепція розробки компілятора мови програмування. Були досліджені їх переваги у тій чи іншій області та загальні недоліки. Згідно з цим було сформовано мету атестаційної роботи.

Серед ключових етапів розробки були виділені:

- проектування граматики;
- створення таблиці символів;
- перевірка відповідності типів;
- генерація проміжного коду та його збереження;
- виконання проміжного коду стек-машиною.

Вцілому був розроблений спосіб написання компілятора мови програмування MiniJava на мові Java, використовуючи існуючі засоби розробки. Зрозуміло, як низький рівень якості, так і високої ефективності, необхідні для написання компілятора.

Написаний компілятор успішно справляється з приєднанням з домашньої сторінки проекту MiniJava такими, як: обчислення факториалу, бінарного пошуку, сортування, обхідного дерева, лінійного пошуку, побудови святкового списку, побудови бінарного дерева.

Це дослідження є важливим для вибору та ефективного використання методів розробки компілятора мови програмування відповідно до вимог та поставлених завдань.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Wirth N. Compiler Construction. Addison Wesley, 1996. — 176 с.

2. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, техно-логии и инструменты.: Пер. с англ. — М.: Издательский дом «Вильямс», 2001. —768 с.: ил.

3. Ахо А., Сети Р., Лам М., Ульман Д. Компиляторы: принципы, технологии и инструментарий.: Пер. с англ. — М.: ООО «И. Д. Вильямс», 2008. —1184 с.: ил.

4. Баранов С. Н., Ноздрунов Н. Р. Язык Форт и его реализации. —Л.: Машиностроение. Ленингр. от-ние, 1988. —157 с., ил.

5. Бек Л. Введение в системное программирование: Пер. с англ. —М.: Мир, 1988. —448 с., ил.

6. Богданов В. В. и др. Программирование на языке АЛМО. Под общ. ред. С. С. Камынина и Э. З. Любимского. М., «Статисти-ка», 1976. 118 с. с ил.

7. Вирт Н. Алгоритмы + структуры данных = программы./ Пер. с англ. — М.: Мир, 1985.

8. Вирт Н. Построение компиляторов/ Пер. с англ. Борисов Е. В., Чернышов Л. Н. – М.: ДМК Пресс, 2010. – 192 с.: ил.

9. Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л. Алгебра, языки, программирование. Киев, «Наукова думка», 1974.

10. Гордеев А. В. Молчанов А. Ю. Системное программное обеспечение. СПб.: Питер, 2001. —736 с. ил.

11. Грис Д. Конструирование компиляторов для цифровых вы-числительных машин.: Пер. с англ. — М.: Мир, 1975.

12. Залогова Л. А. Разработка Паскаль-компилятора: Учебное пособие по спецкурсу / Перм. ун-т. Пермь, 1993. 120 с.

13. Зелковиц М. Шоу А., Гэннон Дж. Принципы разработки про-граммного обеспечения: Пер. с англ. — М.: Мир, 1982 — 368 с., ил.