Лекція №1 Системне програмне забезпечення для компіляції програмного коду.

Місце компілятора в програмному забезпеченні

Компілятори складають одну з найважливіших частин системного програмного забезпечення. Це зв'язано з тим, що мови високого рівня стали основним засобом розробки програм. Тільки дуже незначна частина програмного забезпечення, що вимагає особливої ефективності, програмується за допомогою асемблеру. У дійсний час поширені досить багато мов програмування. З іншого боку, постійно зростаюча потреба в нових компіляторах зв'язана з бурхливим розвитком архітектури ЕОМ. Цей розвиток йде по різних напрямках. Удосконалюються старі архітектури як у концептуальному відношенні, так і по окремим, конкретним лініям. Це можна проілюструвати на прикладі мікропроцесора Intel-80X86. Послідовні версії цього мікропроцесора 8086, 80186, 80286, 80386, 80486, 80586, Pentium і т.д. відрізняються не тільки технічними характеристиками, але і, що більш важливо, новими можливостями і, виходить, зміною (розширенням) системи команд. Природно, це вимагає нових компіляторів (чи модифікації старих). .компілятори ля багатьох мов програмування. Тут необхідно також відзначити, що нові архітектури вимагають розробки зовсім нових підходів до створення компіляторів, так що поряд із власне розробкою компіляторів ведеться і велика наукова праця по створенню нових методів трансляції.

Структура компилятора

На фазі лексичного аналізу (ЛА) вхідна програма, яка представляє собою потік символів, розбивається на лексеми - слова відповідно до визначень мови. Основним формалізмом, що лежить в основі реалізації лексичних аналізаторів, є кінцеві автомати і регулярні вираження. Лексичний аналізатор може працювати в двох основних режимах: або як підпрограма, викликувана синтаксичним аналізатором за черговою лексемою, або як повний прохід, результатом якого є файл лексем. У процесі виділення лексем ЛА може як самостійно будувати таблиці імен і констант, так і видавати значення для кожної лексеми при

черговому звертанні до нього. У цьому випадку таблиця імен будується в наступних фазах (наприклад, у процесі синтаксичного аналізу.

На етапі ЛА виявляються деякі (найпростіші) помилки (неприпустимі символи, неправильний запис чисел, ідентифікаторів і ін.). Основна задача синтаксичного аналізу - розбір структури програми. Як правило, під структурою розуміється дерево, відповідне розбору в контекстно-вільній граматиці мови. В даний час найчастіше використовується або LL(1)-аналіз (і його варіант - рекурсивний спуск), або LR(1)-аналіз і його варіанти (LR(0), SLR(1), LALR(1) і інші). Рекурсивний спуск частіше використовується при ручному програмуванні синтаксичного аналізатора, LR(1) - при використанні систем автоматизації побудови синтаксичних аналізаторів. Результатом синтаксичного аналізу є синтаксичне дерево з посиланнями на таблицю імен. У процесі синтаксичного аналізу також виявляються помилки, зв'язані зі структурою програми. На етапі контекстного аналізу виявляються залежності між частинами програми, що не можуть бути описані контекстно- вільним синтаксисом. Це в основному зв'язку "опис- використання", зокрема аналіз типів об'єктів, аналіз областей видимості, відповідність параметрів, мітки й інші. У процесі контекстного аналізу будується таблиця символів, яку можна розглядати як таблицю імен, поповнену інформацією про описи (властивостях) об'єктів. Основним формалізмом, що використовується при контекстному аналізі, є атрибутные граматики. Результатом роботи фази контекстного аналізу є атрибутированное дерево програми. Інформація про об'єкти може бути як розосереджена в самім дереві, так і зосереджена в окремих таблицях символів. У процесі контекстного аналізу також можуть бути виявлені помилки, зв'язані з неправильним

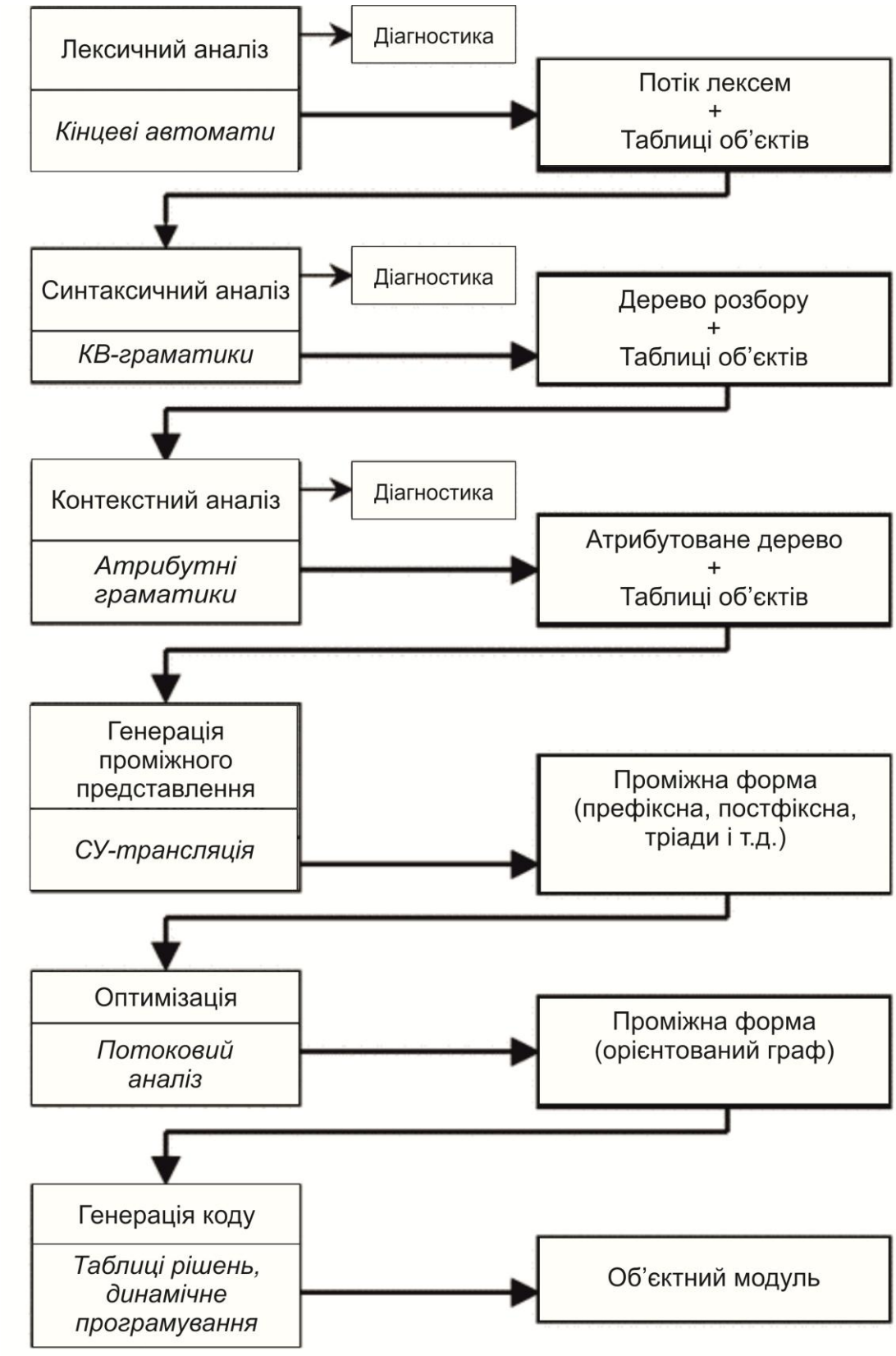

Рисунок 1.1 – Структура компілятору

використанням об'єктів. Потім програма може бути переведена у внутрішнє представлення. Це робиться для цілей оптимізації і/чи зручності генерації коду. Ще однією метою перетворення програми у внутрішнє

представлення є бажання мати стерпний компілятор. Тоді тільки остання фаза (генерація коду) є машинно-залежною. У якості внутрішнього представлення може використовуватися префиксний чи постфиксний запис, орієнтований граф, трійки, четвірки й інші. Фаз оптимізації може бути кілька. Оптимізації звичайно поділяють на машинно-залежні і машинно-незалежні, локальні і глобальні. Частина машинно-залежної оптимізації виконується на фазі генерації коду. Глобальна оптимізація намагається прийняти в увагу структуру всієї програми, локальна - тільки невеликих її фрагментів. Глобальна оптимізація ґрунтується на глобальному потоковий аналізі, що виконується на графі програми і представляє власне кажучи перетворення цього графа. При цьому можуть враховуватися такі властивості програми, як межпроцедурный аналіз, межмодульный аналіз, аналіз областей життя перемінних і т.д. Нарешті, генерація коду - остання фаза трансляції. Результатом її є або ассемблерный модуль, або об'єктний (чи завантажувальний) модуль. У процесі генерації коду можуть виконуватися деякі локальні оптимізації, такі як розподіл регістрів, вибір довгих чи коротких переходів, облік вартості команд при виборі конкретної послідовності команд. Для генерації коду розроблені різні методи, такі як таблиці рішень, зіставлення зразків, що включає динамічне програмування, різні синтаксичні методи. Звичайно, ті чи інші фази транслятора можуть бути або відсутні зовсім, або поєднуватися. У найпростішому випадку однопрохідного транслятора немає явної фази генерації проміжного представлення й оптимізації, інші фази об'єднані в одну, причому немає і явно побудованого синтаксичного дерева.