МЕТОДИЧНІ АСПЕКТИ ВИВЧЕННЯ ТЕМИ “ОСНОВИ КОМПІЛЯЦІЇ” У ПІДГОТОВЦІ МАЙБУТНЬОГО ВЧИТЕЛЯ ІНФОРМАТИКИ

Семеріков С.О., Теплицький І.О.
Криворізький державний педагогічний університет

Вивчення будь-якої мови програмування тісно пов’язане з інструментальним програмним забезпеченням для розробки програм – оболонкою, текстовим редактором та компілятором, що звичайно відносять до системного програмного забезпечення. Інтеграція системного програмного забезпечення у середовища для розробки програм та сучасні операційні системи поступово викликають розмивання поняття системного програмного забезпечення. Так, опитування студентів I–II курсів спеціальності “Інформатика та основи економіки” Криворізького педуніверситету показало, що 62% студентів недостатньо чітко розрізняють основні компоненти системного програмного забезпечення (з них 40% їх плутають), 24% утруднюються з визначенням функцій компілятора, 13% не дали жодного означення.

В процесі навчання мов програмування поняття компілятора поступово збагачується, наповнюючись досвідом практичної роботи, тому виникає необхідність його формалізації, як це було зроблено, наприклад, у навчальному посібнику М.І. Жалдака та Ю.С. Рамського [1].

На жаль, у багатьох студентів поняття компілятора залишається досить розмитим, і цьому є об’єктивні причини. По-перше, компілятори розроблені для великої кількості мов програмування і мають різні форми та методи застосування. По-друге, в багатьох системах компіляція є інтегрованим у середовище розробки процесом. По-третє, продовжується інтенсивний розвиток усіх напрямків досліджень, що відносяться до системного програмного забезпечення: в будові компіляторів з’явилися нові важливі компоненти, технології та розробки [2].

Незважаючи на таку різноманітність систем та постійні зміни, фундаментальні положення процесу компіляції залишаються незмінними, і саме на них варто зосередити увагу студентів. Для цього ми пропонуємо вивчення теми “Основи компіляції” за таким планом:

1. Основні поняття процесу компіляції.
2. Типова структура компілятора.
3. Інтегровані середовища розробки.
4. Вимоги до розробки компіляторів.

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.

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:

1. Лексичний аналіз.

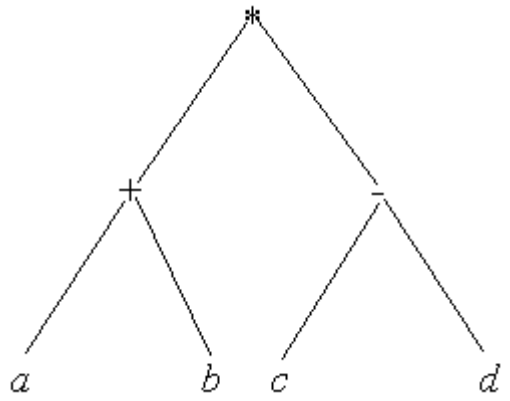
2. Синтаксичний аналіз.

3. 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:



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.

1. Генерація машинно-незалежного коду.

2. Оптимізація машинно-незалежного коду.
3. Розподіл пам’яті.
4. Генерація машинного коду.
5. 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:

1) *статична пам’ять*, якщо час життя змінної дорівнює часу життя програми (не може бути звільнена до завершення виконання програми);
2) *динамічна пам’ять*, якщо час життя змінної дорівнює часу життя

визначеного блоку, функції чи процедури (може бути звільнена після виконання даного фрагмента програми);

3) *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.

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++ 5.02 для Windows, що пропонує такі основні групи операцій.

– *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.

4. Вимоги 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: 1) ефективна компіляція; 2) мінімальний розмір компілятора; 3) мінімальна довжина цільового коду; 4) створення ефективного цільового коду; 5) портабельність; 6) простота використання; 7) 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.

Література:

1. Жалдак М.І., Рамський Ю.С. Інформатика: Навч. посібник / За ред. М.І. Шкіля. – К.: Вища шк., 1991. – 319 с.
2. Хантер Р. Основные концепции компиляторов. – М.: Вильямс, 2002. – 256 с.