ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 1

Історичні аспекти теорії мов програмування

Цей курс про *теорію мов програмування* призначений ознайомити вас з новими способами мислення щодо програмування. Як правило, студенти інформатики починають навчання програмуванню в обов'язковій моделі програмування, де змінні створюються та оновлюються під час виконання програми. Є й інші способи програмування. Після того як навчитися програмуванню в цих нових парадигмах ви почнете розуміти, що існують різні способи мислення щодо вирішення проблем. Кожна парадигма корисна в деяких контекстах.

Цей курс не призначений для охвату багатьох різних мов. Швидше, його мета - познайомити вас із трьома стилями мов програмування, використовуючи їх для реалізації

нетривіальних мов програмування. Ці три стилі програмування:

- Імперативне/Об’єктно-орієнтоване програмування в мовах подібних **Java, C++, Python** й інших.
- Функціональне програмування на мовах подібних **Standard ML, Haskell, Lisp, Scheme**, та інших.
- Логічне програмування на мові **Prolog**.

Багато з того, що ми приписуємо інформатиці, насправді походить від математики. Багато математиків - це програмісти, які написали свої програми або докази словами математики, використовуючи математичні позначення.

У середині 1800-х років абстрактна алгебра та геометрія були гарячими темами досліджень серед математиків. На початку 1800-х років Нільс Генрік Абель, норвезький математик, був зацікавлений у вирішенні задачі, яка називається квінтічним рівнянням.

Врешті-решт він розробив нову галузь математики під назвою *Теорія груп*, за допомогою якої йому вдалося довести, що загального алгебраїчного рішення рівняння квінтіка не було. Враховуючи доказ цього, потрібна нова галузь математики, велика частина роботи Абеля стосувалася розвитку математичної нотації або мови, щоб описати його роботу. На жаль, Авель помер від туберкульозу у двадцять шість років.

Софус Лі , був ще одним норвезьким математиком, який жив з 1842–1899 років.

Він розпочав там, де закінчилися дослідження Абеля, і дослідив зв’язок абстрактної алгебри та теорії груп з геометрією. З цієї роботи він розробив набір групових теорій, зрештою названих Групами Лі. З цього відкриття він знайшов способи вирішення звичайних диференціальних рівнянь, використовуючи властивості симетрії в рівняннях. Одна група Лі, група E8, була надто складною, щоб її можна було відобразити за часів Лі. Насправді лише в 2007 році структуру групи E8 можна було нанести на карту, оскільки рішення дало в шістдесят разів більше даних, ніж проект геному людини.

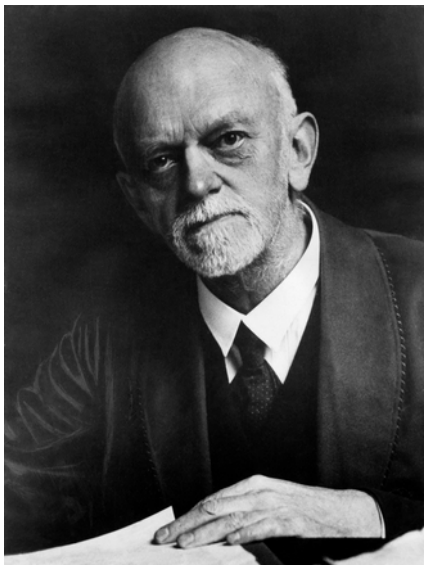
Хоча техніку, яку відкрили Лі та Абель, у той час було важко вивчити та використовувати, сьогодні комп'ютерні програми, здатні до символічних маніпуляцій, використовують методи Лі для вирішення цих та інших не менш складних проблем. І рішення цих проблем є дуже актуальними у світі сьогодні. Наприклад, роботи Софуса Лі використовується при проектуванні літаків.

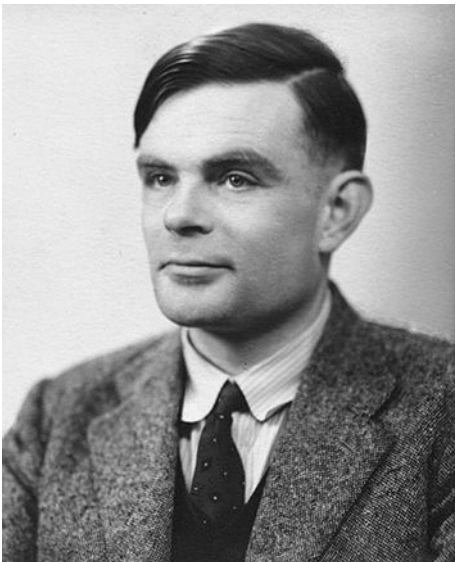
У міру того, як методи розв’язування задач математиків ускладнювались, а проблеми, які вони вирішували, ускладнювались, вони були зацікавлені в пошуку автоматизованих способів вирішення цих проблем.

Чарльз Беббідж (1791–1871) вбачав необхідність комп'ютера для обчислень, занадто схильних до помилок, щоб їх можна було виконувати людям.

Він розробив механізм різниці для обчислення математичних таблиць, коли виявив, що комп'ютери людини не дуже точні. Однак його комп’ютер був механічним і не міг бути побудований за допомогою інженерних методів, відомих на той час. Насправді він не був завершений до 1990 року, але він працював так само, як він сказав, що це зробить понад сто років тому.

Розбіжність механізму Чарльза Беббіджа була ранньою спробою автоматизувати вирішення проблеми, але інші, звичайно, дотримувались цього. Алан Тьюрінг був британським математиком і одним з перших інформатиків. Жив з 1912–1954 років. У 1936 р. Він написав роботу під назвою "Про обчислювальні номери з заявою до *Entscheidungsproblem*». *Проблема Entscheidungs*, або проблема прийняття рішень, була запропонована десятиліттям раніше німецьким математиком на ім'я Девід Гільберт. Ця проблема задається питанням: чи можна визначити алгоритм, який вирішує, чи можна довести твердження, подане в логіці першого порядку, із набору аксіом та відомих значень істини?





Пізніше проблема була узагальнена до наступного питання: Чи можемо ми придумати загальний набір кроків, які, враховуючи будь-який алгоритм та його дані, вирішать, чи припиняється він? У своїй роботі Алан Тьюрінг була запропанована абстрактна машина під назвою *Машина Тьюрінга*.

Ця машина Тьюрінга була дуже загальною і простою. Він складався з набору держав і стрічки. Набір штатів визначав програміст. Машина запускається у стартовому стані за рішенням програміста. З цього стану він міг прочитати символ із стрічки. На основі символу він міг записати символ на стрічку і переміщатися вліво або вправо, переходячи в інший стан. Коли машина Тьюрінга працювала, дія, яку вона здійснила, була продиктована поточним станом та символом на стрічці. Програміст повинен був вирішити, скільки станів є частиною машини, що повинен робити кожен стан і як переходити з одного стану в інший.

У статті Тьюрінг довів, що така машина може бути використана для вирішення будь-якої обчислюваної функції і що проблема рішення не вирішується цією машиною. Більш загальне твердження цієї проблеми було названо проблемою зупинки. Це був дуже важливий результат у галузі теоретичної інформатики.

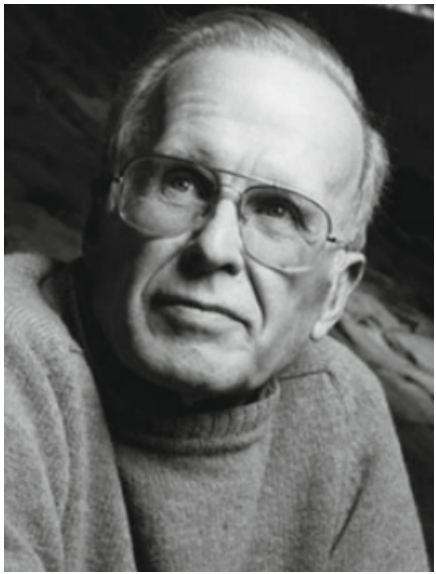
У 1939 році Джон Атанасов з Університету штату Айова спроектував, мабуть, перший комп’ютер - ABC (Atanasoff-Berry Computer). Кліффорд Беррі був одним із його аспірантів. Комп’ютер не мав центрального процесора, але виконував логічні та інші математичні операції. Еккерт та Моклі з Університету Пенсільванії були зацікавлені у побудові комп'ютера під час Другої світової війни. Їх фінансувало Міністерство оборони для побудови машини для розрахунку таблиць траєкторій для запуску снарядів з кораблів.

Комп’ютер під назвою ENIAC для електронно-цифрового інтегратора та комп’ютера був представлений у 1946 році, відразу після закінчення війни. ENIAC було складно запрограмувати, оскільки програма була написана шляхом підключення кабелів до комутатора, подібного до старого телефонного щита.

Приблизно в той же час розроблявся новий комп'ютер, який називався EDVAC. У 1945 році Джон фон Нейман запропонував зберігати в пам'яті комп'ютерні програми на EDVAC разом із даними програм. Алан Тьюрінг уважно стежив за статтями Джона фон Неймана, опублікувавши власну статтю в 1946 р., В якій описав більш повний дизайн комп'ютерів із збереженими програмами. Донині комп’ютери, які ми будуємо і користуються комп'ютерами із збереженою програмою. Архітектуру називають *архітектурою фон Неймана* через внески Джона фон Неймана та Алана Тьюрінга. Хоча Тьюрінг не отримав архітектуру, названу на його честь, він відомий в галузі комп'ютерних наук з інших причин, таких як машина Тьюрінга та проблема зупинки.

На початку інформатики багатьох програмістів цікавило написання інструментів, які полегшували програмування комп’ютерів. Значна частина програмування базувалася на концепції комп’ютера із збереженою програмою, і багато ранніх мов програмування були продовженнями цієї моделі обчислень. У моделі збереженої програми програма та дані зберігаються в пам'яті. Програма обробляє дані на основі певного вводу. Потім він виробляє результат.

Близько 1958 року було створено Algol, і другий перегляд цієї мови, який називався Algol 60, був першою сучасною структурованою імперативною мовою програмування. Хоча мова була розроблена комітетом, значна частина успіху проекту відбулася завдяки внеску Джона Бакуса. Він описав структуру алгольської мови, використовуючи математичні позначення, які пізніше називатимуть Форматом Бакуса-Наура (Пітера Наура) або БНФ. За останні роки дуже мало змінилося з базовою комп'ютерною архітектурою. Звичайно, відбулося багато змін у розмірах, швидкості та вартості комп'ютерів! Крім того, мови, якими ми користуємося, з роками стали ще більш структурованими. Але принципи, які запровадив Algol 60, використовуються і сьогодні.



Пам'ятаючи, що більшість ранніх вчених-інформатиків були математиками, не слід надто дивуватись, дізнавшись, що існували й інші, хто підходив до проблеми програмування інакше. Більша частина початкового інтересу до комп’ютерів була спричинена винаходом комп’ютера із збереженою програмою, і багато ранніх мов відображали це хвилювання. Імперативний стиль був тісно пов'язаний з архітектурою комп'ютера, що зберігається. Дані зчитувались із пристрою введення, і програма діяла на ці дані, оновлюючи пам’ять у міру виконання програми. Одночасно розвивався інший підхід.

Ще в 1936 році Алонцо Черч, американський математик, який жив з 1903–1995 рр., також був зацікавлений у проблемі рішення, запропонованій Девідом Гільбертом.

Щоб спробувати вирішити проблему, він розробив мову, яка називається лямбда-численням, зазвичай записаним як λ-числення. За допомогою своєї дуже простої мови він зміг описати обчислення як маніпулювання символами. Алан Тьюрінг був докторантом Церкви, і хоча вони самостійно придумали два способи довести, що проблема рішення не розв'язувана, пізніше вони довели, що їх дві моделі обчислень, машини Тьюрінга та λ-числення, були рівноцінними. Зрештою їх робота призвела до дуже важливого результату, який отримав назву Теза Черчінга-Тьюрінга. Неофіційно в дисертації зазначено, що всі обчислювані задачі можуть бути вирішені за допомогою машини Тьюрінга або λ-числення. Дві моделі еквівалентні за потужністю.

Ідеї λ-числення привели до розвитку Ліспа Джона Маккарті. Λ-числення та Lisp не були розроблені за принципом комп'ютера із збереженою програмою. На відміну від Algol 60, ці мови були зосереджені на функціях і на тому, що можна обчислити за допомогою функцій. Lisp був розроблений приблизно в 1958 році, в той самий час, коли розроблявся Algol 60.

Логіка важлива як в інформатиці, так і в математиці. Логіки також були зацікавлені у вирішенні проблем у перші дні інформатики. Багато проблем у логіці виражаються мовами логіки пропозицій або предикатів.

Звичайно, розвиток логіки сягає аж до Давньої Греції. Деякі логіки 20 століття цікавились розумінням природної мови, і вони шукали спосіб використовувати комп'ютери для вирішення принаймні деяких проблем, пов'язаних з обробкою тверджень на природній мові. Прагнення використовувати комп'ютери для вирішення логічних задач призвело до розробки Prolog - потужної мови програмування, заснованої на логіці предикатів.

Витоки декількох мов програмування

В цій частині дисципліни ми дослідимо реалізацію мови програмування за допомогою декількох малих мов, які ілюструють кожну з моделей обчислень. Крім того, наші приклади зажадають реалізації в чотирьох різних мовах: мові асемблера, Java (або в якості альтернативи C ++), Standard ML і Prolog.

Але звідки взялися ці мови і чому ми зацікавлені дізнатися, як ними користуватися?

Стислий екскурс в C/C++

Операційна система Unix була задумана, розроблена і написана приблизно в 1972 році. Кен Томпсон працював над дизайном Unix разом з Деннісом Річі. Саме їх проект спонукав Річі створити мову C. C був більш структурованим, ніж мова асемблер, на яку писали більшість операційних систем на той час, він був портативним і міг компілюватися в ефективний машинний код. Томпсон і Річі хотіли мати операційну систему, яка була переносною, невеликою та добре організованою. Хоча C був ефективним, існували й інші мови, які або були розроблені, або розроблялись, що заохочувало більш структурований підхід до програмування.

Протягом декількох років ходили ідеї про те, як писати код в об'єктно-орієнтованій формі. Simula, створена Оле-Йоханом Далем та Крістен Нігаардом близько 1967 року, була першим прикладом мови, яка підтримувала об’єктно-орієнтований дизайн. Modula-2, створена Ніклаусом Віртом близько 1978 року, також скористалася цими ідеями. Інтерпретована мова Smalltalk була об’єктно-орієнтована, а також була розроблена в середині 1970-х років і випущена в 1980 році.

У 1980 році Бьярн Страуструп почав працювати над проектом C++, працюючи в Bell Labs.

Він задумав С ++ як мову, яка дозволить програмістам З зберігати свій старий код, тоді як новий код можна писати за допомогою цих об'єктно-орієнтованих концепцій. У 1983 році він назвав цю нову мову С ++, як і при наступному прирості С, і з великим очікуванням, у 1985 році ця мова була випущена. Приблизно в той же час доктор Страуструп випустив книгу під назвою "Мова програмування С ++", де описується мова. Мова все ще розвивається. Наприклад, шаблони, важлива частина С ++, були вперше описані Stroustrup у 1988 р., і лише в 1998 р. Він був стандартизований як ANSI С ++. Сьогодні комітет ANSI контролює подальший розвиток С ++.

Останній стандарт C ++ був випущений в 2014 році на момент написання статті. Попередній стандарт був випущений в 2011 році. C ++ є зрілою мовою, але все ще зростає та розвивається. В даний час розробляється стандарт 2019 року, коментарі яких зараз вимагає комітет зі стандартів.

Жолудева кава, або історія про Java

С ++ - це дуже потужна мова, але також вимагає, щоб програмісти були дуже обережними при написанні коду. Найбільша проблема програм на С ++ - це витік пам'яті. Коли об'єкти створюються в купі в С ++, вони залишаються в купі, доки не звільняться. Якщо програміст забуде звільнити об'єкт, тоді цей простір не можна використовувати повторно, поки програма запущена. Цей простір втрачається, доки програма не буде зупинена, навіть якщо жоден код вже не має вказівника на цей об’єкт. Це витік пам'яті. І для довготривалих програм на С ++ це проблема номер один. Деструктори - це особливість С + +, яка допомагає програмістам запобігати витоку пам'яті.

Залежно від структури класу у вашій програмі, йому може знадобитися деструктор, який подбає про очищення власних екземплярів (тобто об'єктів класу), коли вони звільнені.
Програми C ++ можуть створювати об'єкти в стеку часу виконання, у купі або в інших об'єктах. Це ще одна потужна особливість C ++. Але завдяки цій владі над створенням об’єктів стає більше відповідальності за програміста. Цей контроль над створенням об'єктів призводить до необхідності додаткового коду для вирішення способу створення копій об'єктів. У C ++ кожен клас може містити конструктор копіювання, щоб програміст міг контролювати спосіб створення копій об'єктів.

У 1991 році команда під назвою Green Team працювала в компанії Sun Microsystems. Ця група інженерів програмного забезпечення хотіла розробити мову програмування та систему виконання, яка могла б бути використана в персональних пристроях наступного покоління. Групу очолював чоловік на ім'я Джеймс Гослінг. Щоб підтримати своє бачення, вони розробили віртуальну машину Java (тобто JVM), програму для інтерпретації файлів байт-коду. JVM був розроблений як система виконання для мови програмування Java. Програми Java під час компіляції перекладаються у файли байт-коду, що працюють на JVM.

1995 рік приніс всесвітню павутину, а разом з нею і один із перших веб-браузерів, Netscape Navigator, який згодом став

Mozilla Firefox. У 1995 році було оголошено, що Netscape включить технологію Java у браузер. Це призвело до певного початкового інтересу до мови, але мова зросла далеко за межі веб-браузерів. Насправді Java насправді вже не є технологією веб-браузера. Він використовується в багатьох веб-серверних системах, де програми Java чекають з'єднань з веб-браузерів, але в наші дні він не запускає програми у веб-браузерах. Інша мова, Javascript, зараз є основною мовою веб-браузерів. Javascript схожий на Java за назвою, але не за своєю технологією. Javascript був зареєстрований як назва від Sun Microsystems у перші дні завдяки популярності Java [22].

Спочатку намір Java полягав у тому, щоб служити засобом для запуску програмного забезпечення для персональних пристроїв. У цьому відношенні Java стала дуже важливою. Зараз це основа для операційної системи Android, яка працює на багатьох телефонах та інших персональних пристроях, таких як планшети. Отже, у певному сенсі первісна мета “Зеленої команди” була реалізована лише через п’ятнадцять років тому.

Коли оригінальна команда Green розробляла Java, вони хотіли використати найкраще з C ++, залишаючи позаду деяку його складність. В Java об'єкти можна створювати лише в одному місці, у купі. Дотримання однієї і тієї самої моделі пам’яті для об’єктів спрощує багато аспектів Java. Об'єкти ніколи не копіюються мовою. Отже, конструктори копіювання непотрібні в Java. Коли об'єкт передається функції, посилання на об'єкт передається без копіювання об'єкта. Коли один об'єкт хоче містити інший об'єкт, він зберігає посилання на цей об'єкт. Об'єкти Java ніколи не зберігаються в інших об'єктах. Спрощення моделі пам'яті для об'єктів означає, що в програмах Java нам не доведеться турбуватися про копіювання об'єктів.

Об'єкти все ще можна копіювати на Java, але копіювання об'єктів відповідає програміст. Мова Java не робить копії. Програмісти роблять копії, викликаючи спеціальний метод, який називається клоном. Java також включає збір сміття. Це означає, що віртуальна машина Java бере на себе рішення про те, коли простір, в якому знаходиться об’єкт, може бути витребовано. Він може бути витребований, коли жоден інший об'єкт або код вже не має на нього посилання. Це означає, що програмістам не потрібно писати деструктори. JVM управляє цим для них.

Отже, хоча C ++ та Java мають багато синтаксису, є також багато відмінностей. Java має простішу модель пам'яті. Збір сміття знімає страх витоків пам’яті в програмах Java. Віртуальна машина Java також надає інші переваги для написання програм Java. Це жодним чином не робить C ++ поганою мовою. Просто Java та C ++ мають різні цілі. JVM та Java керують великою кількістю складності написання об'єктно-орієнтованих програм, звільняючи програміста від цих обов'язків. З іншого боку, C ++ дає вам можливість керувати всіма деталями програми, аж до апаратного інтерфейсу. І той, і інший не кращий за інший, вони просто служать різним цілям, тоді як дві мови також мають спільний синтаксис.

Стислий екскурс до мови Python

Python був розроблений і реалізований Гвідо ван Россумом.

Він розпочав роботу з Python як хобі-проект у зимові місяці 1989 року. Більш повна історія цієї мови доступна в Інтернеті за адресою http://python-history.blogspot.com. Python - це ще одна об'єктно-орієнтована мова, така як C ++ та Java. На відміну від C ++, Python - це інтерпретована мова. Пан ван Россум спроектував інтерпретатор Python як віртуальну машину, подібно до віртуальної машини Java (тобто JVM). Але віртуальна машина Python не доступна окремо, на відміну від JVM. Віртуальна машина Python - це внутрішня деталь реалізації інтерпретатора Python.

Віртуальні машини існують вже деякий час, включаючи операційну систему для основних комп'ютерів IBM, яка називається VM. Використання віртуальної машини під час реалізації мови програмування може зробити мову та її програми більш портативними на різних платформах. Python працює на багатьох різних платформах, таких як Apple Mac OS X, Linux та Microsoft Windows. Віртуальні машини також можуть надавати послуги, що полегшують впровадження мови.

Програмісти у всьому світі прийняли Python і розробили багато бібліотек для Python і написали багато програм. Python набув популярності серед розробників завдяки своїй мобільності та можливості надавати бібліотеки іншим. Гвідо ван Россум у своїй історії Python стверджує: «Велика складна система повинна мати кілька рівнів розширюваності. Це максимізує можливості для користувачів, досвідчених чи ні, допомогти собі ». Розширюваність відноситься до можливості визначати бібліотеки класів для вирішення проблем з багатьох різних областей застосування. Python використовується в Інтернет-програмуванні, сценаріях серверів, комп'ютерній графіці, візуалізації, математиці, освіті з інформатики та багатьох інших сферах застосування.

Пан ван Россум продовжує, кажучи: «Багато в чому філософія дизайну, яку я використав при створенні Python, є, мабуть, однією з головних причин його остаточного успіху. Замість того, щоб прагнути до досконалості, першоприймачі виявили, що Python працював "досить добре" для своїх цілей. У міру зростання кількості користувачів, пропонуються вдосконалення були поступово включені в мову ". Зростання бази користувачів було ключем до успіху Python. Оскільки кількість програмістів, які знають Python, зросла, і він зацікавлений у вдосконаленні мови. Зараз Python має дві основні версії, Python 2 і Python 3. Python 3 не є зворотно сумісним з Python 2. Цей розрив у сумісності дав розробникам Python можливість вдосконалити мову.

Історія мови програмування Standard ML

Стандарт ML зароджувався в 1986 р., Але був продовженням ML, який виник у 1973 р. [16]. Як і багато інших мов, ML було впроваджено з певною метою. ML розшифровується як Meta Language. Мета означає вище або приблизно. Тож метамова це мова про мову. Іншими словами, мова, що використовується для опису мови. ML спочатку був розроблений для системи доведення теорем. Доказ теореми називався LCF, що означає Логіка для обчислюваних функцій. Довідник теореми LCF був розроблений для перевірки доказів, побудованих за певним типом логіки, вперше запропонованим Даною Скотт в 1969 р., А тепер називаним Скотт Логік.

Робін Мілнер був головним розробником системи LCF. Мілнер розробив першу версію LCF, перебуваючи в Стенфордському університеті.

У 1973 році Мілнер переїхав до Единбурзького університету і найняв Локвуда Моріса та Малкольма Ньюї, а потім Майкла Гордона та Крістофера Вадсворта як наукових співробітників, які допомогли йому створити нову та кращу версію під назвою Единбурзький LCF.

Для Единбурзької версії LCF доктор Мілнер та його співробітники створили мову програмування ML, щоб дозволити розширити та налаштувати команди перевірки в новій системі LCF. ML був лише однією частиною системи LCF. Однак швидко стало зрозуміло, що ML може бути корисним як мова програмування загального призначення. У

1990 році Мілнер разом з Мадсом Тофте та Робертом Харпером опублікував перше повне офіційне визначення мови; до них приєднався Девід МакКвін, вони переглянули цей стандарт, щоб створити стандарт ML, який існує сьогодні.
На ML вплинули мови програмування Lisp, Algol та Pascal. Насправді, ML спочатку було впроваджено в Lisp. Зараз існує дві основні версії ML: Московська ML та Стандартна ML.

Сьогодні ML основним чином використовується в наукових колах для дослідження мов програмування. Але він успішно використовується у кількох інших типах додатків, включаючи реалізацію стеку протоколів TCP / IP та веб-сервер як частину проекту Fox. Метою Проекту Fox була розробка системного програмного забезпечення з використанням передових мов програмування.

ML - це дуже хороша мова для використання при навчанні реалізації інших мов. Він включає інструменти для автоматичного генерування частин реалізації мови, включаючи компоненти, що називаються сканером та синтаксичним аналізатором. Ці інструменти, поряд з поліморфною сильною перевіркою типу, що надається Standard ML, роблять реалізацію компілятора або інтерпретатора набагато простішим завданням. Значна частина роботи із впровадження програми в Standard ML витрачається на те, щоб усі типи програми були правильними. Ця сильна перевірка типу часто означає, що як тільки програма правильно набрана, вона запускатиметься вперше. Це цілком твердження, але тим не менше воно часто є правдою.

Важливі стандартні функції ML включають:

- ML - це допоміжні функції вищого порядку як першокласні значення. Це означає, що функції можуть передаватися як параметри функціям і повертатися як значення з функцій.
- Сильна перевірка типу (обговорена далі в цьому розділі) означає, що вам досить рідко потрібно налагоджувати свій код. Яка чудова річ!
- Зіставлення шаблонів використовується у специфікації функцій у ML. Зіставлення шаблонів зручно для запису рекурсивних функцій.

- Система обробки винятків, впроваджена Standard ML, виявилася безпечною для типу, тобто система типів охоплює всі можливі шляхи виконання в програмі ML.

Стислий екскурс до мови логічного програмування Prolog

Пролог був розроблений в 1972 році Аленом Колмерауером, зображеним на рис. 1.9, разом із Філіпом Русселем. Колмерауер та Руссель та їх дослідницька група працювали над обробкою природних мов для французької мови і вивчали логіку та автоматизовану теорему, що підтверджує відповіді на прості запитання французькою мовою. Їхні дослідження змусили їх запросити Роберта Ковальського, який працював у галузі логічного програмування та розробив алгоритм під назвою SL-Resolution, щоб працювати з ними влітку 1971 року.

Колмерауер і Ковальський, працюючи разом у 1971 р., виявили спосіб, коли формальні граматики можуть бути записані як речення в логіці предикатів.

Незабаром Колмерауер розробив спосіб, за допомогою якого логічні предикати можуть бути використані для вираження граматик, що дозволило б автоматизованим доводником теорем ефективно аналізувати речення на природній мові.
Влітку 1972 року Ковальський і Колмерауер знову працювали разом, і Ковальський зміг описати процедурне тлумачення того, що відомо як "клаузульні клаузули". Значна частина дискусій у той час велася навколо того, чи логічне програмування має зосереджуватися на процедурних поданнях чи декларативних поданнях.

Робота Ковальського показала, як логічні програми можуть мати подвійне значення, як процедурне, так і декларативне. Колмерауер і Руссель використовували цю ідею логічних програм як декларативних, так і процедурних, щоб розробити Пролог влітку і восени 1972 року. Перша велика програма Пролог, яка реалізувала систему запитань і відповідей на французькій мові, була написана також у 1972 році.

Пізніше транслятор мови Prolog був переписаний в Единбурзі для компіляції програм у машинний код DEC-10. Це призвело до абстрактної проміжної форми, яка тепер відома як абстрактна машина Уоррена або WAM. WAM - це проміжне представництво низького рівня, яке добре підходить для представлення програм Prolog. Віртуальна машина WAM може бути (і була впроваджена) на широкому спектрі обладнання. Це означає, що реалізації Prolog існують для більшості обчислювальних платформ.

Реалізація мов програмування

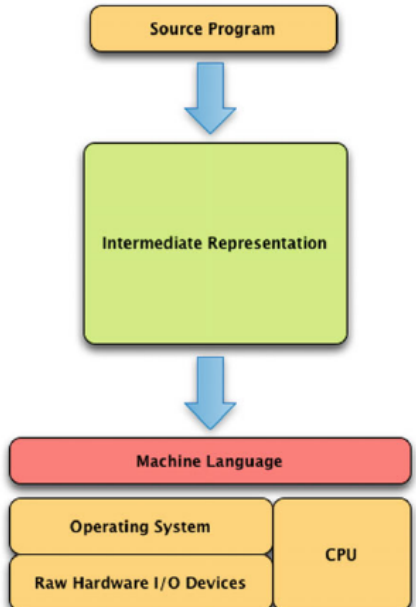
Існує три способи реалізації мов.

••• Мову можна інтерпретувати.

••• Мова може бути скомпільована до машинної мови.

••• Мова може бути реалізована за допомогою деякої комбінації перших двох методів.

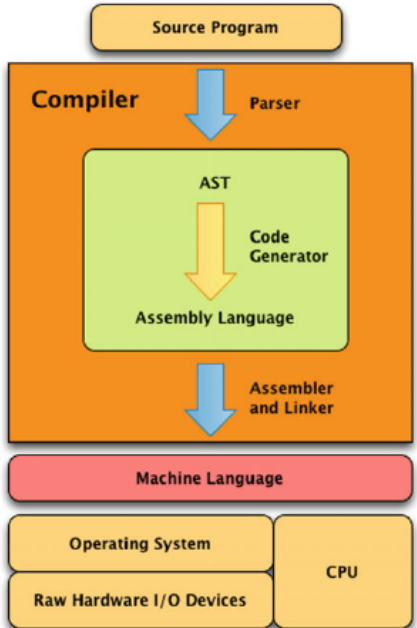
Комп’ютери здатні виконувати лише машинну мову. Машинна мова є мовою центрального процесорного блоку (ЦП) і дуже проста. Наприклад, типові інструкції отримують це значення в центральному процесорі, зберігають це значення в пам’яті з центрального процесора, додають ці два значення разом і порівнюють ці два значення, і якщо вони рівні, перейдіть сюди далі. Мета будь-якої реалізації мови програмування - перевести вихідну програму на цю простішу машинну мову, щоб вона могла виконуватися центральним процесором. Загальний процес зображений на рисунку нижче.


Source Program
Intermediate Representation
Machine Language
Operating System
Raw Hardware I/O Devices
CPU

Усі мовні реалізації перекладають вихідну програму на деяке проміжне представлення перед перекладом проміжного подання на машинну мову. Точно так, як упаковані ці два переклади, значно варіюється від однієї мови програмування до іншої, але, на щастя, більшість мовних реалізацій дотримуються однієї з небагатьох методологій. Надалі будуть представлені деякі приклади вивчення різних мов, щоб ми могли побачити, як виконано та упаковано цей переклад.

Компіляція

Найбільш безпосередній метод перекладу програми на машинну мову називається компіляцією. Процес показаний на рисунку нижче. Компілятор - це програма, яка внутрішньо складається з декількох частин. Синтаксичний аналізатор читає вихідну програму і переводить її в проміжну форму, яка називається абстрактним деревом синтаксису (AST). AST - це деревоподібна структура даних, яка внутрішньо представляє вихідну програму. Про абстрактні дерева синтаксису ми читатимемо у наступних розділах. Потім генератор коду проходить AST і створює іншу проміжну форму, яка називається програмою мови збірки. Ця програма не є машинною мовою, але вона набагато ближча.


Source Program
Compiler
Parser
AST
Code
Generator
Assembly Language
Assembler
and Linker
Machine Language
Operating System
Raw Hardware I/O Devices
CPU

Нарешті, асемблер та компонувальник перекладають програму мови збірки на машинну мову, роблячи програму готовою до виконання.

Весь цей процес інкапсульований інструментом, який називається компілятором. У більшості випадків асемблер і компоновник окремі від компілятора, але зазвичай компілятор запускає асемблер і компоновник автоматично, коли програма компілюється, тому як програмісти ми схильні думати про компілятор, який компілює наші програми, і не обов'язково думати про фази складання та зв'язку.

Програмування на компільованій мові - це триетапний процес.

- По-перше, ви пишете вихідну програму.
- Потім ви компілюєте вихідну програму, створюючи виконувану програму.
- Потім ви запускаєте виконувану програму.

Закінчивши, у вас є вихідна програма та виконувана програма, які представляють одне і те ж обчислення, одне на мові джерела, інше на машинній мові. Якщо ви внесете подальші зміни у вихідну програму, вихідна програма та програма машинної мови не синхронізуються. Після внесення змін у вихідну програму потрібно пам’ятати про перекомпіляцію, перш ніж запускати виконувану програму знову.

Машинна мова властива архітектурі центрального процесора та операційній системі. Компіляція вихідної програми на Linux означає, що вона буде працювати на більшості машин Linux із подібним процесором. Однак ви не можете взяти виконуваний файл Linux і покласти його на машину Microsoft Windows і очікувати його запуску, навіть якщо два комп'ютери мають однаковий процесор. Кожна операційна система Linux та Windows має власний формат виконуваних програм машинної мови. Крім того, скомпільовані програми використовують служби операційної системи для друку, читання вводу та виконання інших операцій введення/виводу. Ці служби по-різному викликаються між операційними системами.

Такі мови, як C ++, приховують ці деталі реалізації від вас у генераторі коду, але кінцевим результатом є те, що програма, скомпільована для однієї операційної системи, не буде працювати в іншій операційній системі без перекомпіляції.
C, C ++, Pascal, Fortran, COBOL та багато інших, як правило, компілюються мовами. В операційній системі Linux компілятор C називається gcc, а компілятор C ++ - g ++. G в обох назвах відображає той факт, що обидва компілятори вийшли з проекту GNU та Фонду вільного програмного забезпечення. Linux, gcc та g ++ є у вільному доступі кожному, хто хоче їх завантажити. Найкращий спосіб отримати ці інструменти - це завантажити дистрибутив Linux і встановити його на комп'ютер. Компілятори gcc та g ++ стандартно поставляються з Linux.

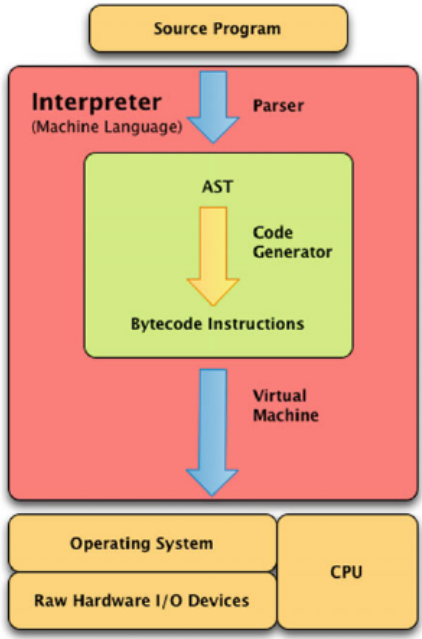
Існують реалізації C та C ++ для багатьох інших платформ. Веб-сайт http://gcc.gnu.org містить посилання на вихідний код та попередньо побудовані двійкові файли для компілятора g ++. Ви також можете завантажити компілятори C ++ від Apple та Microsoft. Для комп'ютерів Mac OS X ви можете отримати C + +, завантаживши інструменти розробника Xcode.

Ви також можете встановити g ++ та gcc для комп'ютерів Mac OS X за допомогою інструменту, який називається brew. Якщо ви запускаєте Microsoft Windows, ви можете встановити Visual C ++ Express від Microsoft. Він безкоштовний для навчального використання.

Інтерпретація

Інтерпретатор - це програма, яка написана якоюсь іншою мовою та скомпільована на машинну мову. Інтерпретатор - це програма машинної мови. Сам інтерпретатор написаний для читання вихідних програм з мови, що інтерпретується, та їх інтерпретації. Наприклад, Python - це інтерпретована мова. Інтерпретатор Python написаний на C і компілюється для певної платформи, такої як Linux, Mac OS X або Microsoft Windows. Щоб запустити програму Python, потрібно завантажити та встановити інтерпретатор Python, який підходить до вашої операційної системи та процесора.

Коли ви запускаєте інтерпретовану вихідну програму, як зображено на рисунку нижче, ви фактично запускаєте інтерпретатор. Ваша програма не запущена, оскільки вона ніколи не перекладається на машинну мову. Інтерпретатор - це програма машинної мови, яка виконує всі програми, які ви пишете мовою, що інтерпретується. Вихідна програма, яку ви пишете, контролює поведінку програми інтерпретатора. Програмування на інтерпретованій мові - це двоступеневий процес.


Source Program
Interpreter
(Machine Language)
Parser
AST
Code
Generator
Bytecode Instructions
Virtual
Machine
Operating System
Raw Hardware I/O Devices
CPU

Програмування на інтерпретованій мові - це двоступеневий процес.

- Спочатку ви пишете вихідну програму.
- Потім ви виконуєте вихідну програму, запускаючи інтерпретатор.

Кожного разу, коли ваша програма виконується, вона перекладається в AST частиною інтерпретатора, яка називається парсером. Може бути додатковий крок, який переводить AST на деяке представлення нижчого рівня, яке часто називають байт-кодом. У перекладачі це представництво нижчого рівня все ще є внутрішнім для програми перекладача. Тоді частина інтерпретатора, яку часто називають віртуальною машиною, виконує вказівки байт-коду.

Не кожен інтерпретатор переводить AST в байт-код. Іноді інтерпретатор безпосередньо інтерпретує AST, але часто зручно перекласти AST вихідної програми на якесь простіше уявлення перед його виконанням.

Усунення кроку компіляції має кілька наслідків.

- Оскільки у вас є один крок у розробці, вам може бути запропоновано запускати ваш код частіше під час розробки. Це загалом хороша річ і може скоротити цикл розвитку.
- По-друге, оскільки у вас немає виконуваної версії коду, вам не потрібно керувати двома версіями. У вас є лише програма вихідного коду, яку слід відстежувати.
- Нарешті, оскільки вихідний код не залежить від платформи, ви зазвичай можете легко переміщати програму між платформами. Інтерпретатор ізолює вашу програму від залежностей платформи.

Звичайно, вихідні програми для компільованих мов, як правило, теж не залежать від платформи. Але їх потрібно перекомпілювати для переміщення виконуваної програми з однієї платформи на іншу. Сам перекладач не є незалежним від платформи. Для кожної комбінації платформа / мова повинна існувати версія перекладача. Отже, існує інтерпретатор Python для Linux, інший для Microsoft Windows і ще один для Mac OS X. На щастя, оскільки інтерпретатор Python написаний на мові C, для кожної платформи можна скомпілювати ту саму програму інтерпретатора Python (з невеликими відмінностями). Доступно багато інтерпретованих мов, включаючи Python, Ruby, Standard ML, скриптові мови Unix, такі як Bash і Csh, Prolog та Lisp.

Переносимість інтерпретованих мов зробила їх дуже популярними серед програмістів, особливо при написанні коду, який повинен працювати на декількох платформах.

Однією з величезних проблем, яка спричинила дослідження інтерпретованих мов, є проблема управління пам'яттю купи. Нагадаємо, що купа - це місце, де динамічно розподіляється пам’ять. Як згадувалося раніше в главі, програми C і C ++ відомі тим, що вони мають витоки пам'яті. Кожного разу, коли програміст на C ++ резервує трохи місця в купі, він / вона повинен пам’ятати про звільнення цього місця. Якщо вони не звільнять місце після закінчення роботи, місце ніколи більше не буде доступним, поки програма продовжує виконуватися.

Купа - це великий простір, але якщо програма працює досить довго і продовжує виділяти, а не звільняти простір, з часом купа заповнюється, і програма закінчується аномально. Окрім того, навіть якщо програма не завершується аномально, продуктивність системи буде погіршуватися, оскільки все більше часу витрачається на управління великим кучевим простором.

Більшість, якщо не всі, інтерпретовані мови не вимагають від програмістів звільнення місця в купі. Натомість існує спеціальне завдання або потік, який періодично запускається як частина інтерпретатора, щоб перевірити купу, чи можна звільнити простір. Це завдання називається збирачем сміття. Програмісти можуть розподілити місце на купі, але не потрібно турбуватися про звільнення цього місця. Для того, щоб збирач сміття працював коректно, простір у купі повинен бути виділений та доступний належним чином. Багато інтерпретованих мов покликані забезпечити правильну роботу збирача сміття.

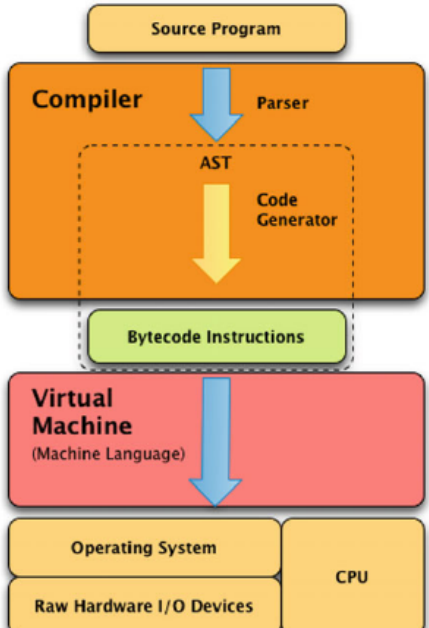
Недолік інтерпретованої мови полягає в швидкості виконання. Інтерпретовані програми зазвичай працюють повільніше, ніж скомпільовані програми. У компільованій програмі аналіз та генерація коду трапляються один раз, коли програма компілюється. При запуску інтерпретованої програми аналіз та генерація коду відбуваються кожного разу, коли програма виконується. Крім того, якщо програма має залежності в реальному часі, тоді збирач сміття, що працює через більш-менш випадкові інтервали, може бути небажаним. Як ви прочитаєте в наступному розділі, було зроблено кілька кроків, щоб зменшити різницю у часі виконання між компільованими та інтерпретованими мовами.

Віртуальні машини

Переваги інтерпретації перед компіляцією досить значні. Виявляється, однією з найбільших переваг є портативність програм. Приємно знати, коли ви витрачаєте час на написання програми, що вона буде працювати однаково на Linux, Microsoft Windows, Mac OS X або будь-якій іншій операційній системі. Це портативність

Це питання спонукало багато досліджень щодо того, щоб програми з інтерпретацією працювали так швидко, як компільовані мови.

Як обговорювалося раніше в цій лекції, концепція віртуальної машини існує досить давно. Віртуальна машина - це програма, яка забезпечує ізоляцію від фактичного обладнання та операційної системи машини, забезпечуючи послідовну реалізацію набору низькорівневих інструкцій, які часто називають байт-кодом. На рисунку нижче показано, як віртуальна машина сидить поверх операційної системи / центрального процесора, щоб виступати в ролі цього ізолятора.


Source Program
Compiler
Parser
AST
Code
Generator
Bytecode Instructions
Virtual
Machine
(Machine Language)
Operating System
Raw Hardware I/O Devices
CPU

Існує не одна специфікація інструкцій байт-коду. Вони специфічні для визначеної віртуальної машини. Python має віртуальну машину, поховану в інтерпретаторі. Prolog - ще один інтерпретатор, який використовує віртуальну машину як частину своєї реалізації. Деякі мови, такі як Java, зробили цю ідею на крок далі. Java має віртуальну машину, яка виконує інструкції байт-коду, як і Python. Творці Java відокремили віртуальну машину від компілятора. Замість того, щоб зберігати інструкції байт-коду всередині, як у інтерпретаторі, компілятор Java, званий javac, компілює програму вихідного коду Java у файл байт-коду. Цей файл не є машинною мовою, тому його неможливо виконати безпосередньо на апаратному забезпеченні.

Це файл байт-коду Java, який інтерпретується віртуальною машиною Java, що називається java у наборі інструментів Java. Усі файли байт-коду Java закінчуються розширенням .class. Можливо, ви помітили ці файли в якийсь момент після компіляції програми Java.

Компілюються програми, написані гібридною мовою, такою як Java. Однак складена програма байт-коду інтерпретується. Вихідні програми на мові не тлумачаться безпосередньо. Додаючи цей проміжний крок, перекладач може бути меншим і швидшим, ніж традиційні перекладачі. Для читання програми має відбутися дуже мало розбору, а виконання програми є простим, оскільки кожна інструкція байт-коду зазвичай має просту реалізацію.

Історія мов програмування захоплююча, і доступно набагато більше деталей, ніж було висвітлено в цій лекції. В Інтернеті є багато чудових ресурсів, де ви можете отримати більше інформації. Використовуйте Google або Вікіпедію та шукайте "Історії мов програмування" як відправний пункт. Однак будьте обережні. Ви не можете повірити усьому, що читаєте в Інтернеті, включаючи Вікіпедію. Хоча Інтернет є чудовим джерелом, вам завжди слід досліджувати свою тему настільки, щоб самостійно перевірити інформацію, яку ви там знайдете.

Під час вивчення нових мов та вивчення впровадження мови програмування стає важливим розуміння моделей обчислень. Компілятор перекладає мову програмування високого рівня в обчислення нижчого рівня. Ці обчислення на низькому рівні зазвичай виражаються в термінах машинної мови, але не завжди. Важливішою за фактичну мову низького рівня є модель обчислення. Деякі моделі базуються на реєстраційних машинах. Деякі моделі засновані на стекових машинах. Втім, інші моделі можуть базуватися на чомусь зовсім іншому.

На наступній лекції ми почнемо розглядати основи теорії мов програмування.