ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 10

Імплементація функціонального програмування (початок)

Весна 2021

Вступ

У лекціях 4-6 була введена мова асамблеру, яка була дуже вказівною мовою. Певні операнди повинні були бути у стеку операндів, перш ніж можна було виконати інструкцію. З цими деталями потрібно було розібратися, навіть незважаючи на те, що програміст намагався вирішити більшу проблему, ніж те, як виконати наступну інструкцію. Це було вирішено вивченням деяких зразків інструкцій асемблерної мови, які можна було б використовувати для вирішення більших проблем, таких як впровадження циклу. Звичайно, навіть написання циклу є більш розпорядчим, ніж спроба обчислити суму деякого списку цілих чисел.

Лекції 7-9 перейшли до Java та C ++, де програмування було менш вказівним. Більшість програмістів вчаться програмувати імперативно спочатку. Об’єктно-орієнтовані мови - це обов’язкові мови, де об’єкти створюються, а стан об’єктів оновлюється у міру виконання програми. Думати про підтримку та оновлення станів об’єктів є набагато менш розпорядчими, ніж думати про те, яку інструкцію виконати далі.

З цієї лекції представлене функціональне програмування. Функціональні мови, такі як Standard ML, очевидно, більше концентруються на написанні та виклику функцій. Однак термін функціональне програмування не говорить про те, чого не вистачає мовам функціонального програмування. Зокрема, чистим функціональним мовам не вистачає операторів присвоєння та ітерацій. Ітерація стосується можливості повторення або повторення коду, як у певному циклі. У чисто функціональній мові неможливо оголосити змінну, яка оновлюється під час виконання вашої програми! Якщо ви задумаєтесь, якщо немає змінних, тоді немає жодної причини для циклічної конструкції в мові. Ітерація та змінні йдуть рука об руку. Але, як отримати будь-яку роботу без змінних? Основним режимом програмування на функціональній мові є рекурсія.

Функціональні мови також містять функцію, якої немає в інших мовах. Вони дозволяють передавати функції функціям як параметри. Ми говоримо, що ці функції вищого порядку. Функції вищого порядку беруть інші функції як параметри і використовують їх. Є багато корисних функцій вищого порядку, які походять від загальних моделей обчислень. В окремих випадках цих зразків зазвичай є одна невелика різниця між ними. Якщо ця невелика різниця залишається як функція, яка буде визначена пізніше, ми маємо одну функцію, яка вимагає іншої функції, щоб завершити її реалізацію. Функції вищого порядку можна налаштувати, надавши деякі їх функції пізніше. Певним чином це функціональний еквівалент того, що наслідування або інтерфейси забезпечують нам в об'єктно-орієнтованих мовах.

Ці дві особливості, відсутність змінних та функції вищого порядку, кардинально змінюють спосіб, яким ви думаєте про програмування. Програмування рекурсивно займає деякий час, щоб звикнути, але врешті-решт це дуже хороший спосіб програмування. Програмування рекурсивно швидше є декларативним, ніж приписовим. Написання імперативних програм є розпорядчим. При декларативному програмуванні ми можемо зосередитись на тому, що хочемо сказати про проблему, а не на тому, як саме її вирішити.

Але чому ми хотіли б позбутися змінних в мові програмування? Проблема в тому, що змінні часто ускладнюють міркування щодо наших програм. Функціональні мови мають більше математичний характер і мають певні правила, такі як комутативність та асоціативність, яких вони дотримуються. Такі правила, як асоціативність та комутативність, можуть полегшити міркування щодо наших програм.

```
1  program P;
2    var b : integer;
3    function a() : integer;
4    begin
5      b:=b+2;
6      return 5
7     end;
8  begin
9    b:=10;
10   write(a()+b)
11   (* or write(b+a()) *)
12 end.
```

Рис.10.1 Комутативність (див. ЛР№5).

Імперативні версії функціонального програмування

Ви, мабуть, знайомі хоча б з однією імперативною мовою. Такі мови, як C, C ++, Java, Python та Ruby, вважаються імперативними мовами, оскільки основною конструкцією є оператор присвоєння. У кожній із цих мов ми оголошуємо змінні та присвоюємо їм значення, оновлюючи ці змінні у міру виконання програми.

На імперативні мови сильно впливає архітектура фон Неймана комп’ютерів, що включає сховище та лічильник програм; обчислювальна модель має структури управління, які перебирають інструкції, що вносять поступові модифікації пам'яті. Присвоєння значень змінним, для циклів і **`while`** цикли - це частина імперативних мов. Основна операція - це присвоєння значень змінним. Програми орієнтовані на оператори та виконують алгоритми з послідовним управлінням на рівні операторів. Іншими словами, обчислення здійснюються побічними ефектами.

Іноді проблеми з імперативними програмами виникають через ці побічні ефекти. Важко міркувати про програму, яка спирається на побічні ефекти. Якщо ми хочемо повторно використати код імперативної програми, тоді ми повинні бути впевнені, що ті самі умови виконуються перед тим, як повторно використаний код виконується, оскільки імперативний код покладається на певний стан машини. Як програмісти, ми іноді забуваємо, які передумови необхідні та які постінні умови виникають при виконанні сегмента коду. Це може призвести до помилок у наших програмах.

Функціональні мови базуються на математичному понятті функції і не відображають основну архітектуру фон Неймана. Ці мови стосуються об’єктів даних та значень замість змінних. Основною операцією є застосування функції.

Функції розглядаються як першокласні об’єкти, які можуть зберігатися в структурах даних, передаватися як параметри та повертатися як результати функцій. Примітивні функції, як правило, постачаються з мовною реалізацією. Функціональні мови дозволяють програмісту визначати нові функції. Функціональне виконання програми складається з оцінки виразу, а послідовне управління замінюється рекурсією.

Відсутня заява про призначення. Значення передаються переважно за допомогою використання параметрів і повернутих значень. Без змінних оператори циклу не мають призначення, а тому вони також не існують у чисто функціональних мовах.

Чисто функціональні мови не мають побічних ефектів, крім можливої читання певних введень від користувача. Схема - це чисто функціональна мова. Взагалі, функціональні мови уникають або принаймні ізолюють код із побічними ефектами. Навіть операції введення та виводу у функціональних мовах не оновлюють стан змінних у програмі.

Дивовижно те, що було доведено, що з функціональними мовами можна обчислити абсолютно ті самі речі, що і з імперативними мовами. Це відомо, оскільки машина Тьюрінга, теоретична основа імперативного програмування та конструкція комп'ютера, була доведена рівноцінною за потужністю Лямбда-численням, основою для всіх функціональних мов програмування.

Ви можете бути здивовані кількістю та типами мов, які підтримують функціональне програмування. Звичайно, Standard ML був розроблений як функціональна мова з нуля, але такі мови, як C ++, Java та Python, також підтримують функціональне програмування. Хоча C ++, Java та Python також є об'єктно-орієнтованими імперативними мовами, всі вони також підтримують функціональне програмування. Функціональне програмування залежить не стільки від мови, скільки від того, як ви використовуєте мову. Далі в цьому розділі буде представлено функціональний стиль програмування. Все почалося з λ-числення, яке коротко розглядається далі.

λ-числення

Усі мови функціонального програмування походять прямо чи опосередковано з роботи Алонцо Черчі та Стівена Кліне. Лямбда-числення було визначено Черчем і Кліне в 1930-х роках, до існування комп'ютерів. На той час математики були зацікавлені в формальному вираженні обчислень у письмовій формі, відмінній від англійської чи іншої неформальної мови. Лямбда-числення було розроблено як спосіб вираження тих речей, які можна обчислити. Це дуже маленька, функціональна мова програмування. У лямбда-числення функція - це відображення від елементів домену до елементів кодомену, заданого правилом.

Розглянемо функціональний **куб(x)=x³**. Яке значення куба ідентифікатора **куб** у визначенні **куб(x)=x³**? Чи можна визначити цю функцію, не даючи їй імені?

λx.x³ визначає функцію, яка відображає кожне **x** у домені до **x³**. Можна сказати, що це визначення або *лямбда-абстракція*, **λx.x³**, є значенням, прив'язаним до куба ідентифікатора. Ми говоримо, що **x³** - тіло лямбда-абстракції. Кожна лямбда-абстракція в ламбда-позначеннях є функцією одного ідентифікатора. Однак лямбда-вирази можуть містити більше одного ідентифікатора.

Вираз **y²+x** можна виразити як лямбда-абстракцію одним із двох способів:

$$\lambda x.\lambda y.y^2 + x$$
$$\lambda y.\lambda x.y^2 + x$$

У першій ламбда-абстракції **x** є першим параметром, який подається до виразу. У другій лямбда-абстракції параметр **y** є параметром для отримання значення першим. У будь-якому випадку абстракцію часто скорочують, викидаючи зайвий **λ**. У скороченій формі дві абстракції стали б **λxy.y²+x** та **λyx.y²+x**.

Нормальна форма

Сказати, що лямбда-числення або будь-яка мова має нормальну форму, означає, що кожен вираз, який можна скоротити, має найпростішу форму. Це означає, що ми можемо якимось механічним чином звести складніші вирази до більш простих. Лямбда-числення має властивість, що називається *злиттям*.

Злиття означає, що одна або кілька стратегій скорочення (або змішування їх) завжди призводять до однакової нормальної форми виразу, припускаючи, що вираз може бути зменшений стратегією скорочення. Ця властивість злиття була доведена в теоремі Черча – Россера.

Предикат **`(Ex) T(a,a,x)`** нерозв'язний, тобто функція:

$$\chi(a) = \begin{cases} 0, & \text{if}(Ex)T(a,a,x) \\ 1, & \text{else} \end{cases}$$

необчислювана.

Дане формулювання використовує поняття обчислюваності по Тьюрингу.

Застосування функції (тобто виклик функції) в лямбда-нотації записується з лямбда-абстракцією, за якою слідує значення, з яким потрібно викликати абстракцію. Таке поєднання називається *редекс*. Для виклику $\lambda x.x^3$ зі значенням 2 для x ми б написали

$(\lambda x.x^3)2$

Ця комбінація лямбда-абстракції та значення називається *редексом*.

Ця комбінація лямбда-абстракції та значення називається редексом. Редекс - це лямбда-вираз, який може бути зменшений. Зазвичай лямбда-вираз містить кілька редексів, які можна вибрати для зменшення. Застосування функції є лівоасоціативним, що означає, що якщо на одному рівні вкладених дужок доступно більше одного редекса, то спочатку слід зменшити крайній лівий редекс. Якщо крайній лівий зовнішній редекс завжди вибирається для зменшення першим, порядок зменшення називається нормальним зменшенням порядку. Коли редекс скорочується за допомогою застосування лямбда-числення, еквівалентного застосуванню функції, це називається *β-скороченням* (вираженим бета-скороченням).

Зменшення нормального порядку

(λx yz.x z (yz)) (λx.x) (λx y.x)

наведено на рис. 10.2. Редекс, який буде β-зменшено на кожному кроці, підкреслено.

$$\underline{(\lambda xyz.xz(yz))(\lambda x.x)(\lambda xy.x)}$$
$$\Rightarrow \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda xy.x)}$$
$$\Rightarrow \lambda z.\underline{(\lambda x.x)z}((\lambda xy.x)z)$$
$$\Rightarrow \lambda z.z\underline{((\lambda xy.x)z)}$$
$$\Rightarrow \lambda z.z(\lambda y.z)\square$$

Рис.10.2 Зниження нормального порядку

В лабі 5 у завданні 2 вам слід буде зменшити лямбда-вираз до того самого зменшеного лямбда-виразу, отриманого із зменшення нормального порядку на рис. 10.2. Якщо ви цього не зробили, ви зробили щось не так. Якщо ви хочете отримати більше досвіду зі зменшення лямбда-виразів, ви можете порадитися з перекладачем лямбда-виразів.

Одного чудового перекладача написав Пітер Сестофт і він доступний в Інтернеті. Він знаходиться за адресою http://www.itu.dk/people/sestoft/lamreduce/ . Обов'язково прочитайте його довідкову сторінку, щоб ознайомитись із синтаксисом, необхідним для введення лямбда-виразів у його інтерпретатор. Також пам'ятайте, що його перекладач не розуміє математичних символів, таких як **+**. Замість цього ви можете використовувати **p** для представлення додавання, якщо це необхідно. Інтерпретатор лямбда-числення Сестофта призначений для чистого лямбда-числення без знання математики чи будь-якої іншої мови.

Lambda calculus reduction workbench

This system implements and visualizes various reduction strategies for the pure untyped lambda calculus. It is intended as a pedagogical tool, and as an experiment in the programming of visual user interfaces using Standard ML and HTML.

- Start lambda calculus reducer. The lambda calculus reducer scripts now run on a tiny Raspberry Pi Linux server.
- Here is the implementation source code as a zip file.
- There is an old draft report describing the implementation, in PDF (645 KB).

Syntax

The letter lambda is written as a backslash `\'; otherwise the syntax is the usual one:

```
Lambda expression:

       e ::= x                        Variable
          |  (\x.e)                   Abstraction
          |  (e e)                    Application

Abbreviation:

       b ::= x = e                    Bind e to x

Program:
       p ::= e                        Expression to evaluate
          |  let b1; ...; bn in e     Abbreviations and expression
```

Problems with Applicative Order Reduction

Іноді додаткове зменшення замовлення може призвести до проблем. Наприклад, розглянемо вираз `(λx.y) ((λx.x x) (λx.x x))`.

Практична проблема 3 в ЛР№5 показує, чому визначення злиття включає фразу, припускаючи, що вираз можна зменшити за допомогою стратегії скорочення. Порядок застосування не завжди може призвести до зменшення виразу. Не бійтеся, якщо це станеться, ми можемо деякий час використовувати нормальне зменшення замовлення, оскільки перемішування стратегій зменшення не вплине на те, чи прийдемо ми до нормальної форми виразу чи ні.

Початок роботи зі Standard ML

Standard ML (або просто SML) - це функціональна мова, заснована на Lisp, яка, в свою чергу, базується на лямбда-числення. Важливі особливості ML вказані нижче.

- SML - це допоміжні функції вищого порядку як першокласні значення.
- Він набирається на зразок Pascal, але більш потужний, оскільки підтримує поліморфну перевірку типу. За допомогою цієї сильної перевірки типу досить рідко виникає потреба у налагодженні коду !! Яка чудова річ !!!

- Обробка винятків вбудована в Standard ML. Він забезпечує безпечне середовище для розробки та виконання коду. Це означає, що в ML немає традиційних покажчиків. Покажчики обробляються як посилання на Java.
- Оскільки традиційних покажчиків немає, збір сміття реалізовано в системі ML.
- Для зручного написання рекурсивних функцій передбачено узгодження шаблонів.
- Є вбудовані вдосконалені структури даних, такі як списки та рекурсивні структури даних.
- Доступна бібліотека загальновживаних функцій та структур даних, яка називається Бібліотека основ.

Існує кілька реалізацій StandardML. StandardML з Нью-Джерсі та MoscowML є найбільш повними та, безумовно, найпопулярнішими. Існує також реалізація SML.NET, яка націлена на бібліотеку виконання Microsoft .NET і може бути інтегрована з іншими мовами .NET. Існує реалізація MLj, орієнтована на віртуальну машину Java. Poly / ML - ще одна реалізація, яка включає підтримку програмування Windows. Хоча існує багато реалізацій, всі вони підтримують одне і те ж визначення SML. Якщо ви напишете стандартну програму ML, яка працює в одному середовищі, вона буде працювати в будь-якій іншій реалізації, якщо ви не використовуєте певні функції платформи.

SML успішно використовується у багатьох великих програмових проектах. Він був використаний для реалізації всього протоколу TCP у проекті FOX у Карнегі-Меллоні. Він використовувався для реалізації сценаріїв на стороні сервера на веб-серверах. Спочатку він був розроблений як мова написання доказів теорем і широко використовувався в цій галузі. Він був використаний при проектуванні та верифікації обладнання. Він також використовувався в дослідженнях мов програмування.

Решта цієї лекції присвячена SML. До кінця лекції ми повинні зрозуміти та мати змогу використовувати багато важливих особливостей мови. Цей текст заснований на StandardML з реалізації в Нью-Джерсі. Ви можете завантажити SML Нью-Джерсі з smlnj.org . SML з Нью-Джерсі доступний для більшості платформ, тому ви зможете знайти реалізацію для своїх потреб.

Після встановлення SML ви можете відкрити вікно терміналу та запустити інтерпретатор. Введення **sml** у командному рядку запустить інтерактивний режим інтерпретатора. Введення **ctl-d** припинить перекладача. Ви можете вводити вирази та програми безпосередньо в підказці перекладача, а можете вводити їх у файл і використовувати цей файл у SML. Для цього ви вводите слово вживання наступним чином:

```
Standard ML of New Jersey v110.59
- use "myfile.txt";
```

SML візьме все, що ви набрали у файлі, і оцінить так, як ніби ви ввели це безпосередньо в інтерпретатор.

Приклади та наступні завдання в ЛР№5 в цьому розділі вводять SML. Наступний матеріал представляє важливі аспекти SML і готують студента писати більш складні програми в наступних лекціях.

Вирази, типи, структури та функції

Функціональне програмування зосереджується на оцінці виразів. У SML ви можете оцінювати вирази прямо в інтерпретаторі. Під час оцінки виразу ви помітите, що інформація про тип відображається разом із результатом оцінки виразу. Наведене нижче діалогове вікно містить деякі оцінки виразів інтерпретатора SML.

У ідентифікаторі SML він прив'язаний до результату останнього успішно оціненого виразу. Це зручно, якщо ви хочете використовувати результат у наступному виразі. Результатом останнього виразу можна називати його в наступному, інтерактивно введеному виразі.

Взаємодія, представлена на рис. 10.3, містить негативну взаємодію, записану як `~1` у SML.
Хоча трохи нетрадиційний, "`~`" є унарним оператором заперечення в SML, що відрізняє його від двійкового оператора віднімання.

```
- 6;
val it = 6 : int
- 5*3;
val it = 15 : int
- ~1;
val it = ~1 : int
- 5.0 * 3.0;
val it = 15.0 : real
- true;
val it = true : bool
- 5 * 3.0;
Error: operator and operand don't agree
  operator domain: int * int
  operand:         int * real
  in expression:
5 * 3.0
-
```

Рис.10.3 Інтерактивний інтерпретатор

SML має дуже сувору систему типів. Насправді система типів для SML виявилася надійною. Це означає, що будь-яка правильно набрана програма гарантовано не містить помилок типу. SML набирається статично, як C ++ та Java. Це означає, що всі помилки типу виявляються під час компіляції, а не під час виконання. Робін Мілнер це довів
Стандартний ML. ML - це єдина широко розповсюджена мова, система типів якої була офіційно визначена та перевірена як правильна.

Хоча формально визначена та сувора, система типу ML надзвичайно гнучка. Він поліморфний. Невдовзі ми побачимо, що це означає для нас. Багато типів в ML також неявно виражені. У C++ та Java повинен бути оголошений тип кожної змінної та функції. Ви можете побачити на рис. 10.3, що програміст ніколи не вводив жодного типу для наведених там виразів. У більшості випадків система типових стандартів ML звільняє програміста від необхідності вказувати типи в програмі, оскільки вони в основному визначаються автоматично.

Ви також могли помітити, що на рис. 10.3 є помилка типу. ML поліморфний, але він також сильно типизований. Оскільки `5` - це ціле число в SML, а `3,0` - дійсне, їх не можна множити разом. Якщо у вас є потреба помножити ціле і дійсне це можна зробити, але ви повинні явно перетворити в один із типів. Наведена нижче взаємодія інтерпретатора показує деякий код для множення цілого і дійсного, створюючи дійсне число.

```
- Real.fromInt(5) * 3.0;
val it = 15.0 : real
-
```

Ціле число **`5`** перетворюється в **`5.0`** за допомогою виклику функції, що викликається **`fromInt`**, у структурі **`Real`**. Структура в SML - це групування функцій і типів. Структура схожа на модуль на Python або на включення в C ++. Існує декілька структур, що складають Бібліотеку основ для стандартної ML. Базова бібліотека доступна в SML під час запуску інтерпретатора. Структури в базовій бібліотеці включають **`Bool, Int, Real, Char, String`** та **`List`**. Веб-сайт http://standardml.org/Basis містять описи багатьох із цих структур.

Функція в SML приймає один або кілька аргументів і повертає значення. Підпис функції - це тип функції. Іншими словами, тип функції - це її підпис. Підпис функції **`fromInt`** у структурі **`Real`** є

```
val fromInt : int -> real
```

Introduction

These web pages contain the interface specifications for the modules of the **SML Basis Library**, which is a standard library for the 1997 Revision of SML**[CITE]**. The SML Basis Library provides interfaces and operations for basic types, such as integers and strings, support for input and output (I/O), interfaces to basic operating system interfaces, and support for standard datatypes, such as options and lists. The Library does *not* attempt to define higher-level APIs, such as collection types or graphical user-interface components. These APIs are left for other libraries.

This document may be distributed freely over the internet as long as the copyright notice and license terms below are prominently displayed within every machine-readable copy. The SML Basis Library is also published as a book by *Cambridge University Press*. In addition to the manual pages, the book also contains tutorial descriptions of programming techniques and idioms for effective use of the Library's interfaces.

The design philosophy of the SML Basis Library is to use the SML module system as an organizing tool. All type, exception, and value identifiers are bound in some module. A small number of these, which are called *pervasive identifiers*, are also bound at top-level (*i.e.*, without qualification). In addition, the top-level environment defines overloading of some identifiers.

Цей підпис вказує на те, що **`fromInt`** приймає **`int`** як аргумент і повертає **`real`**. З назви функції та того факту, що вона є частиною **`real`** структури, ми можемо встановити, що вона створює дійсне число з **`int`**.

Тип на лівій стороні стрілки (тобто **`->`**) - це тип аргументів, що надаються функції. Тип праворуч від стрілки - це тип значення, що повертається функцією. Функція **`fromInt`** приймає int як аргумент і повертає **`real`**.

Рекурсивні функції

Рекурсія - це спосіб зробити щось функціональною мовою. Рекурсія відбувається, коли функція викликає себе. Через принцип референційної прозорості функція ніколи не повинна викликати себе з однаковими аргументами. Якби це було зроблено, то функція зробила б саме те, що робила минулого разу, називаючи себе тим самим аргументи, які б тоді .... Ну, ви отримуєте картину!

Щоб позбавити себе цієї проблеми, ми наполягаємо на тому, щоб відбувалися дві речі. *По-перше*, кожна рекурсивна функція повинна мати базовий випадок. Базовий випадок - це проста підзадача, яку ми намагаємось вирішити, і не вимагає рекурсії. Ми повинні написати якийсь код, який перевіряє просту проблему і просто повертає відповідь у такому випадку.

Друге правило рекурсивних функцій вимагає від них закликати себе до якоїсь простішої чи меншої підзадачі. Певним чином кожен рекурсивний виклик повинен зробити крок до базового випадку проблеми. Якщо кожен рекурсивний виклик рухається до базового випадку, то за математичним принципом індукції ми можемо зробити висновок, що функція працюватиме для всіх значень, за якими функція визначена! Фокус не в тому, щоб думати над цим надто важко. Рекурсивний випадок часто називають індуктивним.

Написання функціональних програм набагато декларативніше, ніж директивне програмування збірки та імперативне програмування на таких мовах, як C ++, Python та Java. Насправді це твердження говорить про те, що під час написання рекурсивних функцій ми набагато менше думаємо про те, як це працює, а більше про структуру даних. Це призводить до кількох простих кроків, які можна застосувати до написання будь-якої рекурсивної функції. Запам’ятайте ці кроки і практикуйте їх, і ви можете написати будь-яку рекурсивну функцію.

1. Вирішіть, як називається функція, які аргументи їй передаються і що функція повинна повертати.

2. Принаймні один з аргументів повинен кожного разу зменшуватися. Здебільшого це лише один аргумент, який стає меншим. Вирішіть, який це буде.

3. Напишіть декларацію функції, оголосивши ім'я, типи аргументів та тип повернення, якщо це необхідно.

4. Напишіть базовий випадок для аргументу, який, на вашу думку, зменшиться. Виберіть найменше, найпростіше значення, яке можна було б передати функції, і просто поверніть результат для базового випадку.

5. Наступний крок - вирішальний. Ви не пишете наступне твердження зліва направо. У цей момент ви пишете зсередини.

6. Зробіть рекурсивний виклик функції з меншим значенням. Наприклад, якщо це список, який, як ви вирішили, зменшиться, викличте функцію з хвостом списку. Якщо ціле число є аргументом, що стає меншим, викличте функцію з цілочисельним аргументом мінус 1. Викличте функцію з необхідними аргументами і, зокрема, з меншим значенням аргументу, який ви вирішили зменшувати на кожному кроці.

7. Ось, ось стрибок віри. Той виклик, який ви зробили на останньому кроці, спрацював! Він повернув результат, який ви очікували від аргументів, які він наводив. Використовуйте цей результат для побудови результату для вихідних аргументів, переданих функції. На цьому етапі може бути корисно спробувати конкретний приклад. Припустимо, що рекурсивний виклик спрацював на конкретному прикладі. Що ви маєте робити з цим результатом, щоб отримати результат, який ви хотіли для початкового дзвінка? Напишіть код, який використовує результат для побудови кінцевого результату для вашого конкретного прикладу. Розглянувши конкретний приклад, це допоможе вам зрозуміти, які обчислення необхідні для отримання вашого кінцевого результату.

8. Ось і все! Ваша функція завершена, і вона буде працювати, якщо ви дотримуєтесь цих вказівок.

Щоб визначити функцію в SML, ми пишемо ключове слово **`fun`**, за яким слідує назва функції, параметри, знак рівності та тіло функції. Синтаксис дуже схожий на визначення функцій іншими мовами. Основна відмінність - це тіло функції. Замість того, щоб бути послідовністю операторів із присвоєнням змінної, тіло функції буде виразом.

Одним із важливих виразів у SML є вираз **`if-then-else`**. Це не твердження **`if-then-else`**. Натомість це вираз **`if-then-else`**. Вираз **`if-then-else`** дає одне з двох значень, і ці значення повинні бути сумісними за типом. Найпростіший спосіб зрозуміти вирази «**`if-then-else`**» - це побачити їх на практиці.

Вавілонський метод обчислення квадратного кореня числа, **x**, полягає в тому, щоб почати з довільного числа як здогадки. Якщо вгадайте **2 = x**, ми готові. Якщо ні, то нехай буде наступним відгадуванням (здогадка **+ x** / здогадка) /2.0. Щоб записати це як рекурсивну функцію, ми повинні знайти базовий випадок і бути впевненими, що наші послідовні припущення наближатимуться до базового випадку.

Оскільки вавилонський метод пошуку квадратного кореня є добре відомим алгоритмом, ми можемо бути впевнені, що він буде збігатися на квадратному корені. Базовий випадок повинен бути написаний так, що коли ми підійдемо досить близько, ми будемо готові. Нехай досить близький коефіцієнт складає одну мільйонну частину від початкового числа.

```
fun babsqrt(x,guess) =
  if Real.abs(x-guess*guess) <
     x/1000000.0 then
    guess
  else
    babsqrt(x,(guess + x/guess)/2.0);
```

Рис. 10.4 Square Root

Код SML на рис. 10.4 реалізує цю функцію. Дивлячись на код, слід спостерігати дві речі. Базовий випадок стоїть на першому місці. Якщо відгадка знаходиться в межах однієї мільйонної частки від правильного значення, функція повертає відгадування як квадратний корінь. Інше спостереження - рекурсивний виклик наближає нас до рішення.

Символи, рядки, і списки

SML має окремі типи символів і рядків. Буквальний символ починається зі знака фунта (тобто `#`). Потім персонажа оточують подвійні лапки. Отже, перший символ в алфавіті в SML представлений як "`a`". У структурі `Char` доступно кілька функцій для тестування та перетворення символів. Сигнатура функцій у структурі `Char` наведено на сайті.

Рядки в SML - це не просто послідовності символів, як це є в деяких мовах. Рядок у SML - це власний примітивний тип. Звичайно, є функції для перетворення між рядками та символами. Список цих функцій можна переглянути в Додатку B. Рядковий літерал - це текст, оточений подвійними лапками. Символ зворотної косої риски (тобто `\`) є символом екранування у рядках. Це означає включити подвійну лапку до рядка, який ви можете написати ”як частину рядка. `\N` - символ нового рядка у рядку, а `\t` - символ табуляції, як і в багатьох мовах.

Мабуть, найпотужніша структура даних в SML - це список. Список є поліморфним, що означає, що в SML існує багато типів списків. Однак усі функції списку працюють із списками будь-якого типу. Оскільки неможливо визначити всі типи в SML (оскільки програмісти можуть визначати власні типи), тип списку параметризується типом змінна. Тип списку записується як „список. Коли тип списку відомий, змінна типу `'a` замінюється типом, який вона представляє. Отже, список цілих чисел має тип `int list`. Можливо, ви вже це зрозуміли, але списки в SML повинні бути однорідними.

Це означає, що всі елементи списку повинні мати однаковий тип. Це не схоже на деякі мови, але для цього є вагома причина. Вимагання однорідності списків робить можливим статичну перевірку типів у SML, а перевірка типу звучить і завершена.

```
:: : 'a * 'a list -> 'a list
@ : 'a list * 'a list -> 'a list
hd : 'a list -> 'a
tl : 'a list -> 'a list
```

Рис.10.5 Сигнатури функцій


```
:: : 'a * 'a list -> 'a list
@ : 'a list * 'a list -> 'a list
hd : 'a list -> 'a
tl : 'a list -> 'a list
```

Список будується одним із кількох способів. По-перше, порожній список представляється як нуль або порожнім списком (тобто `[]`). Список може бути представлений як літерал, розмістивши ліву та праву дужки навколо вмісту списку, як у `[1,4,9,16]`. Список також може бути побудований за допомогою конструктора списку, який пишеться `::` та вимовляється проти. У функціональній мові Lisp той самий оператор побудови списку пишеться мінусами, тому його багато функціональні програмісти називають оператором мінус. Оператор `cons` бере елемент ліворуч від нього та список праворуч і створює новий список з двох аргументів. Список може бути побудований шляхом об'єднання двох списків разом.

Об’єднання списків представлено символом `@`. Нижче наведено всі допустимі конструкції списків у SML.
Ви можете вибрати елементи зі списку за допомогою функцій **`hd`** та **`tl`**. **`Hd`** (вимовляється заголовок) списку є першим елементом списку. **`Tl`** - це хвіст або всі інші елементи списку. Виклик функцій **`hd`** або **`tl`** у порожньому списку призведе до помилки. Використовуючи ці дві функції та рекурсію, можна отримати доступ до кожного елемента списку. Код на рис. 10.6 ілюструє функцію, яка називається **`implode`**, яка бере список символів як аргумент і повертає рядок, що складається з цих символів. Отже, імплодування (**`[# "H", # "e", # "l", # "l", # "o"]`**) дасть "**`Hello`**".

Під час написання рекурсивної функції фокус полягає в тому, щоб не надто думати про те, як це працює. Подумайте про базовий випадок чи випадки та про рекурсивні випадки окремо. Отже, на рис. 10.6 базовий випадок - це те, коли список порожній (оскільки список є параметром). Коли список порожній, рядок, який повинна повертати функція, також повинен бути порожнім.

```
fun implode(lst) =
  if lst = [] then ""
  else str(hd(lst))^implode(tl(lst))
```

Рис. 10.6 Функція Implode

```
fun length(x) =
    if null x then 0
    else 1+length(tl(x))
fun append(L1, L2) =
    if null L1 then L2
    else hd(L1)::append(tl(L1),L2)
```

Рис. 10.7 Дві функції списку

Рекурсивний випадок - це коли список не порожній. У цьому випадку у списку є принаймні один елемент. Якщо це правда, тоді ми можемо викликати **`hd`**, щоб отримати перший елемент, а **`tl`** - решту списку. Заголовок списку є символом і повинен бути перетворений у рядок. Решта списку перетворюється на рядок, викликаючи якусь функцію, яка перетворить список на рядок. Ця функція називається **`implode`**! Ми можемо просто припустити, що це спрацює. Така природа рекурсії. Фокус, якщо він є, полягає в тому, щоб довіряти, що рекурсія спрацює. Пізніше ми дослідимо, чому ми можемо довіряти рекурсії.

Код на рис. 10.7 містить ще пару прикладів функцій списку. Функція довжини підраховує кількість елементів у списку. Це повинен бути список, оскільки використовується функція `tl`. Функція додавання додає два списки, беручи кожен елемент із першого списку та консолідуючи його до результату додавання решти першого списку до другого списку.

Відповідність шаблону

Часто рекурсивні функції покладаються на кілька рекурсивних та кілька базових випадків. SML включає приємну можливість обробки цих різних випадків у рекурсивному визначенні, дозволяючи узгоджувати шаблони аргументів із функцією. Зіставлення шаблонів працює з такими буквальними значеннями, як `0`, порожній рядок та порожній список. Як правило, ви можете використовувати зіставлення шаблонів, якщо зазвичай використовуєте рівність для порівняння значень. Дійсні числа не є типами рівності. `Real` тип лише наближає дійсні числа. Код на рис. 10.4 показує, як порівнюються два дійсних числа для рівності.

Ви також можете використовувати конструктори в шаблонах. Тож конструктор списку `::` також працює у шаблонах. Такі функції, як функція додавання (тобто інфікс `@`) та об'єднання рядків (тобто `^`), не працюють у шаблонах. Ці функції не є конструкторами значень і не можуть ефективно або детерміновано узгоджуватися із шаблонами аргументів.

Додаток можна записати за допомогою зіставлення зразків, як показано на рис. 10.8. Потрібні додаткові патерни навколо рекурсивного виклику для додавання, оскільки конструктор `::` має вищий пріоритет, ніж додаток функції.

```
fun append(nil,L2) = L2
  | append(h::t,L2) = h::(append(t,L2))
```

Рис.10.7 Відповідність шаблону

Кортежі

Тип кортежу є перехресним продуктом типів. Дві кортежі - це поперечний добуток двох типів, три кортежі - це поперечний добуток трьох типів тощо. **`(5,6)`** є двою кортежем **`int * int`**. Три кортежі **`(5,6, "привіт")`** мають тип **`int * int * string`**.

Можливо, ви помітили підпис деяких функцій у цьому розділі. Наприклад, розглянемо підпис функції додавання. Його сигнатура

```
val append : 'a list * 'a list -> 'a list
```

Це вказує на те, що це функція, яка бере за аргумент «список *» кортеж списку. Насправді кожна функція приймає один аргумент і повертає одне значення. Єдиним аргументом може бути кортеж одного або декількох значень, але кожна функція приймає один параметр як параметр. Повернене значення функції також може бути кортежем.

У багатьох інших мовах ми думаємо про написання програми як про функцію, за якою слідує ліва лінія, за якою стоять аргументи, розділені комами, а за нею права. У StandardML (та більшості функціональних мов) додаток функції записується як ім'я функції, за яким слідує значення, до якого застосовується функція.

Це так само, як застосування функції в лямбда-числення. Отже, ми можемо подумати про виклик функції з нулем або більше значень, але насправді кожна функція в ML передається аргументу, який може бути кортежем.

Let вирази та сфера дії

Нехай вирази - це просто синтаксис для прив’язки значення до ідентифікатора, який згодом буде використовуватися у виразі. Вони корисні, коли ви хочете задокументувати свій код, присвоївши значущому імені значення. Вони також можуть бути корисними, коли одне і те ж значення потрібно більше одного разу у визначенні функції. Замість того, щоб викликати функцію двічі, щоб отримати одне і те ж значення, ви можете викликати її один раз і прив’язати значення до ідентифікатора. Тоді ідентифікатор можна використовувати стільки разів, скільки потрібно значення. Це ефективніше, ніж виклик функції кілька разів з однаковими аргументами.

Розглянемо функцію, яка обчислює суму перших n цілих чисел, як показано на рис. 10.9. Нехай вирази визначають ідентифікатори, які є локальними для функцій. Ідентифікатор, що називається **`sum`** на рис. 10.9, не видно поза визначенням функції **`sumupto`**. Ми говоримо, що область дії суми - це тіло виразу **`let`** (тобто вираз, поданий між ключовими словами **`in`** та **`end`**). Нехай вирази дозволяють нам оголошувати ідентифікатори з обмеженим обсягом.

```
fun sumupto(0) = 0
  | sumupto(n) =
    let val sum = sumupto(n-1)
    in
      n + sum
    end
```

Рис.10.9 Let вираз

Обмеження сфери застосування є важливим аспектом будь-якої мови. Визначення функцій також обмежують сферу застосування в SML та більшості мов. Формальні параметри визначення функції не видно поза тілом функції.

Значення прив’язки до ідентифікаторів не слід плутати з присвоєнням змінних. Прив'язка значення до ідентифікатора є одноразовою операцією. Значення ідентифікатора не можна оновити як змінну. Ілюструвати це допоможе практична проблема.

Прив’язки - це не те саме, що змінні. Прив'язки робляться один раз і лише один раз і не можуть бути оновлені. Змінні мають оновлюватися в міру просування коду. Прив’язки - це зв’язок між значенням та ідентифікатором, який не оновлюється.

SML та багато сучасних мов використовують статичні або лексичні правила обсягу. Це означає, що ви можете визначити область дії змінної, переглянувши структуру програми, не враховуючи її виконання. Слово лексичний відноситься до письмового слова, а лексичний або статичний обсяг відноситься до визначення обсягу шляхом перегляду того, як пишеться код, а не виконання коду. Спочатку LISP використовував правила динамічного обсягу. Щоб визначити динамічну область дії, ви повинні переглянути прив'язки, які були активними під час виклику виконуваного коду. Різницю між динамічною та статичною сферою можна побачити, коли функції можуть бути вкладені в мову, а також можуть передаватися як параметри або повертатися як результати функції.

Різницю між динамічним та статичним обсягом можна спостерігати в програмі на рис. 10.10. У цій програмі функція **a** при виклику оголошує локальне прив'язування **x** до **1** і повертає функцію **b**. Коли **c**, результат виклику **a**, називається, він повертає **a 1**, значення **x** у середовищі, де було визначено **b**, а не **2**. Цей результат є тим, що більшість людей очікує, що відбудеться. Це статичний або лексичний обсяг. Правильне значення **x** залежить не від значення **x**, коли воно було викликане, а від значення, де була записана функція **b**.

```
let fun a() =
        let val x = 1
            fun b() = x
        in
          b
        end
    val x = 2
    val c = a()
in
  c()
end
```

Рис.10.10 Область дії

Хоча статичний обсяг використовується багатьма мовами програмування, включаючи Standard ML, Python, Lisp та Scheme, він використовується не всіма мовами. Версія Emacs Lisp використовує динамічну область дії, і якщо еквівалентна програма Lisp для коду на рис. 10.10 обчислюється в Emacs Lisp, вона поверне значення `2`.

Насправді важче реалізувати статичну область, ніж динамічну. У мовах з динамічним масштабом, коли функція повертається як значення, повертане значення може містити вказівник на код функції. Коли функція `b` з рис. 10.10 виконується мовою з динамічним масштабом, вона просто шукає значення x у поточному середовищі.

Для реалізації статичного обсягу потрібно більше вказівника на код. Потрібен вказівник на поточне середовище, яке містить прив'язку **`x`** до значення на момент визначення функції. Це потрібно, щоб під час оцінки функції **`b`** можна було знайти правильне прив’язку **`x`**. Поєднання вказівника на код функції та її середовища називається закриттям. Закриття використовуються для представлення значень функцій у мовах зі статичним масштабом, де функції можуть повертатися як результати та можуть бути визначені вкладені функції. У попередніх лекціях було введено замикання, а на рис. 10.10 наведено приклад у StandardML, де показано, чому вони необхідні для мов зі статичним масштабом.

Типи даних

Слово тип даних часто широко використовується в інформатиці. У ML тип даних - це особливий тип типу. Тип даних - це позначена структура, яку можна визначити рекурсивно. Цей тип потужний тим, що ви можете визначати перелічені типи за допомогою нього, а також визначати рекурсивні структури даних, такі як списки та дерева.

Типи даних - це визначені користувачем типи і, як правило, визначаються рекурсивно. Типів даних у Standard ML існує нескінченно багато. Визначення типу даних подібно до створення класу в C++ без будь-яких методів і лише загальнодоступних даних. У C/C++ ми можемо створити перелічений тип, написавши декларацію, зображену на рис. 10.11. Це визначає тип, що називається **`TokenType`** з одинадцяти значень: ідентифікатор **`0`**, ключове слово **`1`**, число **`2`** тощо. Ви можете оголосити змінну цього типу наступним чином.

```
TokenType t = keyword;
```

Однак до C++ 11 ніщо не заважало вам виконати декларацію

```
t = 1; //this is the keyword value.
```

У цьому прикладі, навіть якщо t має тип **`TokenType`**, йому може бути присвоєно ціле число з компіляторами до C++ 11. Це тому, що тип **`TokenType`** був просто іншою назвою цілочисельного типу в C++ до C ++ 11. Присвоєння **`t`** до **`1`** нітрохи не заважало C++. Насправді, присвоєння **`t 99`** не буде турбувати C++ ні до C++ 11. У Standard ML та тепер у C++ ми не можемо використовувати цілі числа та типи даних (або перелічення) як взаємозамінні.

```
- datatype TokenType = Identifier | Keyword | Number |
    Add | Sub | Times | Divide | LParen | RParen | EOF |
    Unrecognized;
datatype TokenType = Identifier | Keyword | Number | ...
- val x = Keyword;
x = Keyword : TokenType
```

Типи даних дозволяють програмістам визначати власні типи. Зазвичай тип даних включає іншу інформацію. Типи даних використовуються для представлення певних структурованих даних. Додавши ключове слово, значення типу даних може включати кортеж інших типів як частину свого визначення. Тип даних може представляти будь-який тип рекурсивної структури даних. Сюди входять списки, дерева та інші структури, пов’язані зі списками та деревами. На рис. 10.12 ми маємо визначення дерева із поєднанням одинарних та двійкових вузлів. Типи даних дозволяють програмісту писати рекурсивну функцію, яка може обходити дані, що йому передаються. Функції можуть використовувати зіставлення зразків для обробки кожного випадку у типі даних із збігом зразків у функції.

```
enum TokenType {
    identifier,keyword,
    number,add,sub,times,
    divide,lparen,
    rparen,eof,unrecognized
};
```

Рис.10.11 C++ Enum Type

```
datatype
  AST = add' of AST * AST
      | sub' of AST * AST
      | prod' of AST * AST
      | div' of AST * AST
      | negate' of AST
      | integer' of int
      | store' of AST
      | recall';
```

Рис.10.12 AST Datatype

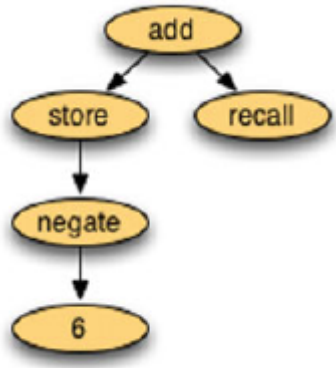

Рис.10.13 AST in SML

У типі даних, наведеному на рис. 10.12, значення додавання може розглядатися як вузол в AST, який має двох дочірніх об'єктів, кожен з яких є AST. Тип даних є рекурсивним, оскільки він визначається як сам по собі. Код на рис. 10.12 - це повне визначення абстрактних дерев синтаксису для виразів мовою калькулятора. Зберігання вузлів у дереві зберігає їх значення в одному місці пам’яті калькулятора. Вузли виклику нагадують місце пам’яті калькулятора. Вузол заперечення відображає одинарне заперечення значення, яке ми отримуємо при оцінці його дочірнього елемента. Отже `~6` є допустимим виразом, якщо дозволити знаку тильди представляти одинарне заперечення, як це робиться у StandardML.

Абстрактне дерево синтаксису для `~ 6S + R` зображено графічно на рис. 10.13. Додана вартість `'(store' (negate '(integer' (integer '(6))), recall')` - це SML-спосіб представлення AST, показаний на рис. 10.13. Функцію можна записати для оцінки такого абстрактного дерева синтаксису на основі шаблонів у такому значенні, і це робиться далі в наступній лекції.

```
1  fun evaluate(add'(e1,e2),min) =
2      let val (r1,mout1)= evaluate(e1,min)
3          val (r2,mout) = evaluate(e2,mout1)
4      in
5        (r1+r2,mout)
6      end
7
8    | evaluate(sub'(e1,e2),min) =
9      let val (r1,mout1)= evaluate(e1,min)
10         val (r2,mout) = evaluate(e2,mout1)
11     in
12       (r1-r2,mout)
13     end
```

Рис.10.14 Результати функції відповідності шаблонів

Ви можете використовувати зіставлення шаблонів для типів даних. Наприклад, для обчислення дерева виразів ви можете написати рекурсивну функцію за допомогою зіставлення шаблонів. Кожен шаблон, який відповідає такій функції, відповідає обробці одного вузла в дереві. Кожне піддерево може бути оброблене рекурсивним викликом тієї ж функції. На рис. 10.14, параметр **`min`** - це значення пам'яті перед обчисленням даного вузла в дереві абстрактного синтаксису. Значення **`mout`** - це значення пам'яті після обчислення вузла в дереві абстрактного синтаксису.

Цей приклад коду на рис. 10.14 ілюструє, як використовувати зіставлення шаблонів з типами даних та шаблонами в конструкції **`let`**. Це один із способів написати функцію оцінки для оцінки абстрактних дерев синтаксису, визначених на рис. 10.12. **`mout1`** - значення пам'яті після обчислення **`e1`**. Це передається оцінці **`e2`** як значення пам'яті перед оцінкою **`e2`**. Значення пам’яті після обчислення **`e2`** - це значення пам’яті після обчислення суми / різниці двох виразів. Цей шаблон передачі пам'яті через оцінку дерева називається однопоточною пам'яттю при обчисленні.

Передача параметрів у StandardML

Типи даних у стандартній ML включають цілі числа, дійсності, символи, рядки, кортежі, списки та типи даних, що визначаються користувачем, представлені в останньому розділі. Якщо ви розглянете ці типи в цій главі та у Додатку B, ви можете помітити, що немає функцій, які б змінювали існуючі дані. Функція підрядка, визначена для рядків, повертає новий рядок. Насправді більшість функцій типів даних, доступних у Standard ML, повертають нове значення без мутації переданих їм аргументів. Не всі дані Standard ML незмінні, але більшість із них є.

Існує один тип даних, який можна змінювати в Standard ML. Посилання - це посилання на значення визначеного типу. Посилання можуть мутувати, щоб програміст міг програмувати, використовуючи імперативний стиль програмування. Посилання розглядаються більш докладно далі в цьому розділі. Тип масиву в Standard ML - це список посилань, тому масиви, як правило, також вважаються змінними типами даних, але лише тому, що масиви є списками посилань. Відсутність змінних даних, за винятком посилань, має певний вплив на реалізацію мови. Значення передаються за посиланням у Standard ML.

Однак важливий єдиний момент, коли посилання передається як параметр або один із небагатьох змінних типів об'єктів передається функції. В іншому випадку незмінність усіх даних означає, що спосіб передачі даних функції не має значення. Це приємно для програмістів, оскільки їм не потрібно турбуватися про те, які функції мутують дані, а які створюють нові значення даних. Для більшості практичних цілей існує лише одна операція, яка мутує дані, оператор присвоєння (тобто `:=`), і єдиними даними, які він може мутувати, є посилання. Крім того, оскільки більшість даних незмінні та передаються за посиланням, параметри ефективно передаються в ML.

На наступній лекції буде продовжений розгляд імплементації функціонального програмування.