ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 11

Імплементація функціонального програмування (завершення)

Весна 2021

Ефективність рекурсії

Як тільки ви звикнете, писати рекурсивні функції не надто складно. Насправді це може бути простіше, ніж писати ітеративні рішення. Але те, що ви знайшли рекурсивне рішення проблеми, не означає, що це ефективне рішення проблеми. Розглянемо числа Фібоначчі. Рекурсивне визначення призводить до дуже прямого рекурсивного рішення. Однак, як виявляється, просте рекурсивне рішення є чим завгодно, але не ефективним. Насправді, враховуючи визначення на рис. 11.1, **`fib(42)`** займав шість секунд для обчислення на 2,66 ГГц MacBook Pro з 8 ГБ оперативної пам'яті. **`fib(43)`** складає на третину довше, стрибнувши до дев'яти секунд.

```
1  fun fib(0) = 0
2    | fib(1) = 1
3    | fib(n) = fib(n-1) + fib(n-2)
```

Рис.11.1 Функція fib

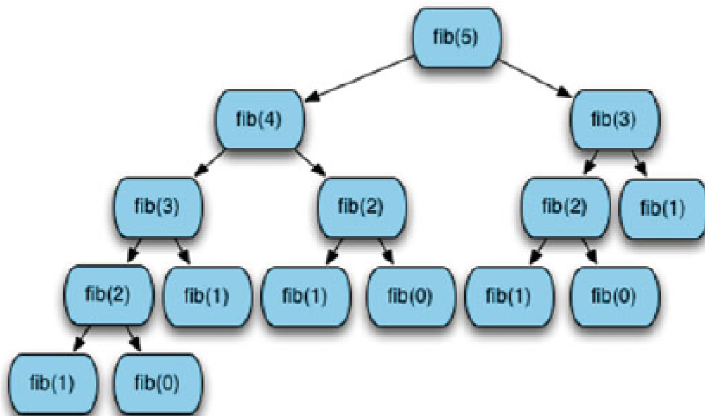

Рис.11.2 Виклики для обчислення fib(5)

Числа Фібоначчі можна обчислити з визначенням функції, наведеною на рис. 11.1. Це дуже неефективний спосіб обчислення чисел Фібоначчі. Кількість викликів до `fib` зростає в геометричній прогресії із збільшенням розміру `n`. Це можна побачити, подивившись на дерево викликів `fib`, як на рис. 11.2. Кількість викликів, необхідних для обчислення `fib(5)`, дорівнює `15`. Якби ми перерахували виклики, необхідні для обчислення `fib(6)`, це було б все в дереві викликів `fib(5)` плюс кількість вузлів у `fib(4)` дерево викликів, `15 + 9 = 25`. Кількість викликів зростає в геометричній прогресії.

```
1  fun fib(n) =
2    let fun fibhelper(count,current,previous) =
3          if count = n then previous
4          else fibhelper(count+1,previous+current,current)
5    in
6          fibhelper(0,1,0)
7    end
```
Рис.11.3 Ефективна функція fib

З цього аналізу ви, мабуть, помітили, що багато і тієї ж роботи роблять знову і знову. Можливо, можна буде усунути велику частину цієї роботи, якщо ми будемо розумнішими щодо способу написання функції Фібоначчі. Насправді це так. Ключ до цієї ефективної версії **`fib`** полягає у визнанні того, що ми можемо отримати наступне значення в послідовності, додавши два попередні значення. Якщо ми просто переносимо два значення, поточне та наступне значення в послідовності, ми можемо обчислити кожне число Фібоначчі лише за одним викликом. Код на рис. 11.3 демонструє, як це зробити. З новою функцією обчислення **`fib(43)`** відбувається миттєво.

Використання допоміжної функції може призвести до кращого впровадження в деяких ситуаціях. У випадку функції **`fib`** функція **`fibhelper`** перетворює експоненційно складну функцію в лінійну функцію часу. Код на рис. 11.3 використовує допоміжну функцію, яка є приватною для функції **`fib`**, оскільки ми не хочемо, щоб інші програмісти безпосередньо викликали функцію **`fibhelper`**. Він призначений для використання функцією **`fib`**. Ми також не хотіли б пам'ятати, як викликати функцію **`fibhelper`** кожного разу, коли ми її викликали. Приховуючи його у функції **`fib`**, ми можемо показати той самий інтерфейс, який був у нас з початковою реалізацією, але реалізувати набагато ефективнішу функцію.

Допоміжна функція використовує шаблон, званий накопичувачем. Допоміжна функція використовує акумулятор, щоб зменшити обсяг виконаної роботи. Робота зменшується, оскільки функція відстежує останні два значення, обчислені допоміжною функцією, щоб допомогти у обчисленні наступного числа.

Хвостова рекурсія

Одна з критик центрів функціонального програмування через інтенсивне використання рекурсії, яку деякі критики вважають надмірно неефективною. Проблема пов’язана з використанням кеш-пам’яті в сучасних процесорах. Залежно від розміру блоку кешу інструкцій, код, що оточує виконуваний код, може бути легко доступний у кеші. Однак, коли потік інструкцій переривається викликом функції, навіть тієї самої функції, кеш може не містити правильних інструкцій.

Отримання інструкцій з пам'яті відбувається набагато повільніше, ніж їх пошук у кеш-пам’яті. Однак розміри кешу продовжують збільшуватися, і навіть такі імперативні мови, як C ++ та Java, заохочують багато викликів до невеликих функцій або методів, враховуючи їх об'єктно-орієнтований характер. Отже, аргумент на користь меншої кількості викликів функцій, безумовно, зменшився за останні роки.

Усе ще буває, що виклик функції займає більше часу, ніж виконання простого циклу. Коли здійснюється виклик функції, виконуються додаткові інструкції для створення нового запису активації. Крім того, у конвеєрних процесорах конвеєр порушується через виклики функцій. StandardML Нью-Джерсі, Scheme та деяких інших функціональних мов мають механізм, за допомогою якого вони оптимізують певні рекурсивні функції за рахунок зменшення обсягу пам’яті в стеку часу виконання та виключення дзвінків.

У деяких випадках рекурсивні виклики можуть бути автоматично перетворені в код, який можна виконати за допомогою інструкцій про перехід або розгалуження. Щоб ця оптимізація стала можливою, рекурсивна функція повинна бути рекурсивною. Хвостова рекурсивна функція - це функція, де остання операція функції - це рекурсивний виклик до себе.

Функція факторіалу представлена на рис. 11.4. Чи є факторіальний хвіст рекурсивним? Відповідь - ні. Рекурсія хвоста трапляється тоді, коли останнє, що зроблено в рекурсивній функції - це заклик до себе. Останнє, що зроблено на рис. 11.4, це множення.
Коли викликається факторіал **6**, потрібні записи активації для семи викликів функції, а саме факторіал **6** через факторіал **0**. Без кожного з цих кадрів стеку локальні значення **n**, **n=6** крізь **n=0**, будуть втрачені, так що множення в кінці не може бути здійснено правильно.

На найглибшому рівні рекурсії вся інформація у виразі,

$$(6*(5*(4*(3*(2*(1*(factorial 0))))))$$

зберігається в стеці виконання під час виконання.

```
1  fun factorial 0 = 1
2    | factorial n = n * factorial (n-1);
```

Рис.11.4 Факторіал

```
1  fun factorial n =
2      let fun tailfac(0,prod) = prod
3            | tailfac(n,prod) = tailfac(n-1,prod*n)
4      in
5        tailfac(n,1)
6      end
```
Рис.11.5 Хвостова рекурсія факторіалу

Функцію факторіалу можна записати як хвостову рекурсивну. Рішення полягає у використанні методики, подібної до поліпшення функції `fib`, виконаної на рис. 11.3. До визначення функції додано акумулятор. Акумулятор - це додатковий параметр, який може використовується для накопичення значення, приблизно так, як ви накопичували б значення у циклі. Значенню накопичувача спочатку надається ідентичність операції, яка використовується для накопичення значення. На рис. 11.5 операція є множенням. Ідентичність, що надається як початкове значення, дорівнює 1.

Функція, представлена на рис. 11.5, є рекурсивною версією факторіальної функції. Рекурсивна функція хвоста - допоміжна функція `tailfac`. Зверніть увагу, що хоча `tailfac` є рекурсивним, немає необхідності зберігати це локальне середовище, коли воно викликає себе, оскільки після виклику не залишається обчислень. Результат рекурсивного виклику просто передається як результат поточного виклику функції. Функція є рекурсивною, якщо її рекурсивний виклик є останньою дією, яка відбувається під час будь-якого конкретного виклику функції.

Каррінг

Двійкова функція, наприклад, `+` або `@`, приймає обидва свої аргументи одночасно. `a + b` оцінить як `a`, так і `b`, щоб значення могли передаватися операції додавання. Може бути перевага в тому, що двійкова функція бере аргументи по черзі. Така функція називається каррінг за іменем Хаскелла Каррі. Функції ML беруть свої параметри по одному, оскільки всі функції приймають рівно один аргумент. Функція `curried` також приймає один аргумент. Однак ця функція одного параметра може, у свою чергу, повернути функцію, яка приймає один аргумент. Це, мабуть, найкраще проілюструвати на прикладі. Ось функція, яка приймає пару аргументів як свій вхід через один кортеж.

```
- fun plus(a:int,b) = a+b;
val plus = fn : int * int -> int
```

Функція плюс приймає один аргумент, який просто є кортежем. Виклик функції означає надання їй єдиного кортежу.

```
- plus (5,8);
val it = 13 : int
```

Функції ML можна визначити, маючи на увазі кілька параметрів:

```
- fun cplus (a:int) b = a+b;
val cplus = fn : int -> (int -> int )
```

Зверніть увагу на підпис функції **cplus**. Здається, потрібно два аргументи, але бере їх по одному. Власне, **cplus** бере лише один аргумент. Функція **cplus** повертає функцію, яка приймає другий аргумент. Друга функція не має назви.

```
- cplus 5 8;
val it = 13 : int
```

Застосування функції залишається асоціативним. У наведених нижче таблицях показано порядок операцій.

```
- (cplus 5) 8;
val it = 13 : int
```

Ми можемо дати цій функції назву.

```
- val add5 = cplus 5;
val add5 = fn : int -> int
- add5 8;
val it = 13 : int
```

Результатом (**cplus5**) є функція, яка додає **5** до свого аргументу. Функції **curried** дозволяють частково оцінити, дуже цікаву тему функціональними мовами, але виходять за рамки цього тексту. Слід зазначити, що StandardML з Нью-Джерсі широко використовує функції **curried** у своїй реалізації. Додаток B містить безліч функцій, сигнатури яких відображають те, що вони виконують.

Анонімні функції

На початку цієї глави описується лямбда-числення. У цьому розділі ми дізналися, що функції можна охарактеризувати як об'єкти першого класу. Функції можуть бути представлені лямбда-абстракцією, і їм не потрібно присвоювати назву. Це справедливо і для SML. Функції в SML не потребують імен. Анонімну функцію $\lambda xy.y^2+x$ можна представити в ML як

```
fn x => fn y => y*y + x;
```

Анонімну функцію можна застосувати до значення так само, як іменовану функцію застосувати до значення. Застосування функції - це завжди функція спочатку, а потім значення.

```
- (fn x => fn y => y*y + x) 3 4;
val it = 19 : int
```

Ми можемо визначити функцію, прив’язавши лямбда-абстракцію до ідентифікатора:

```
- val f = fn x => fn y => y*y + x;
val f = fn: int -> int -> int
- f 3 4;
val it = 19 : int
```

Цей механізм забезпечує альтернативну форму визначення функцій, якщо вони не є рекурсивними; у декларації **`val`** визначений ідентифікатор не видно у виразі праворуч від стрілки. Для рекурсивних визначень потрібен вираз **`val rec`**. Щоб визначити рекурсивну функцію за допомогою анонімної форми функції, ви повинні використати **`val rec`**, щоб оголосити її.

Функції вищого порядку

Унікальною особливістю функціональних мов є те, що функції обробляються як першокласні об'єкти з тими самими правами, що й інші об'єкти, а саме зберігаються в структурах даних, передаються як параметр і повертаються як результати функцій. Функції можуть бути прив'язані до ідентифікаторів за допомогою ключових слів **`fun`**, **`val`** і **`val rec`**, а також можуть зберігатися в структурах. Це приклади функцій, які розглядаються як значення.

```
- val fnlist = [fn (n) => 2*n, abs, ~, fn (n) => n*n];
val fnlist = [fn,fn,fn,fn] : (int -> int) list
```

Зверніть увагу, що кожна з цих функцій приймає **int** і повертає **int**. Функцію ML можна визначити для застосування кожної з цих функцій до числа. Функція побудови застосовує перелік функцій до значення.

```
- fun construction  nil n = nil
    | construction (h::t) n = (h n)::(construction t n);
val construction = fn : ('a -> 'b) list -> 'a -> 'b list
- construction [op +, op *, fn (x,y) => x - y] (4,5);
val it = [9,20,~1] : int list
```

Конструкція базується на функціональній формі, знайденій у FP, ранній функціональній мові програмування, розробленій Джоном Бекусом. Це ілюструє можливість передачі функцій як аргументів. Оскільки функції є першокласними об'єктами в ML, вони можуть зберігатися в будь-якій структурі. Можна уявити собі програму для стека функцій або навіть дерева функцій.

Функція називається вищим порядком, якщо вона приймає функцію як параметр або повертає функцію як результат. Функції вищого порядку іноді називають функціональними формами, оскільки вони дозволяють будувати нові функції з уже визначених функцій.

Корисність функціонального програмування походить від використання функціональних форм, що дозволяють розробляти складні функції з простих функцій із використанням абстрактних зразків. Функція побудови - одна з цих абстрактних закономірностей обчислень. Ці функціональні форми або закономірності обчислень знову і знову з'являються в програмах. Програмісти розпізнали ці закономірності та абстрагували деталі, щоб отримати кілька загальновживаних функцій вищого порядку. Наступні розділи представляють декілька з цих функцій вищого порядку.

Склад

Складання двох функцій - це природно операція вищого порядку, яку ви, мабуть, використовували в алгебрі. Ви коли-небудь писали щось на зразок `f(g(x))`? Ця операція може бути виражена в ML. Насправді ML має вбудований оператор, який називається o, який представляє склад. Цей приклад коду демонструє, як композицію можна писати та використовувати.

```
- fun compose f g x = f (g x);
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
- fun add1 n = n+1;
val add1 = fn : int -> int
- fun sqr n:int = n*n;
val sqr = fn : int -> int
- val incsqr = compose add1 sqr;
val incsqr = fn : int -> int
- val sqrinc = compose sqr add1;
val sqrinc = fn : int -> int
```

Зауважте, що ці дві функції, **`incsqr`** та **`sqrinc`**, визначаються без використання параметрів.

```
- incsqr 5;
val it = 26 : int
- sqrinc 5;
val it = 36 : int
```

ML має заздалегідь визначену функцію інфіксації, яка складає функції. Зверніть увагу на те, що o - це непророблений.

```
- op o;
val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- val incsqr = add1 o sqr;
val incsqr = fn : int -> int
- incsqr 5;
val it = 26 : int
- val sqrinc = op o(sqr,add1);
val sqrinc = fn : int -> int
- sqrinc 5;
val it = 36 : int
```

Відображення

У SML застосування функції до кожного елемента в списку називається **`map`** і заздалегідь визначеним. Він приймає унарну функцію та список як аргументи та застосовує функцію до кожного елемента списку, що повертає список результатів.

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- map add1 [1,2,3];
val it = [2,3,4] : int list
- map (fn n => n*n - 1) [1,2,3,4,5];
val it = [0,3,8,15,24] : int list
- map (fn ls => "a"::ls) [["a","b"],["c"],["d","e","f"]];
val it = [["a","a","b"],["a","c"],["a","d","e","f"]] :
             string list list
- map real [1,2,3,4,5];
val it = [1.0,2.0,3.0,4.0,5.0] : real list
```

```
fun map f nil = nil
  | map f (h::t) = (f h)::(map f t);
```

Рис.11.6 Функція map

Функція **map** заздалегідь визначена у структурі List, але наведена на рис. 11.6 для довідки.

Зменшення або згорнення

Функції вищого порядку розробляються шляхом абстрагування загальних шаблонів із програм. Наприклад, розглянемо функції, які знаходять суму або добуток списку цілих чисел. У цьому шаблоні результати попереднього виклику функції використовуються в двійковій операції з наступним значенням, яке буде використано при обчисленні.

Іншими словами, для складання списку значень ви починаєте з першого або останнього елемента списку, а потім додаєте його разом із значенням поруч із ним. Потім ви додаєте результат цього обчислення до наступного значення у списку тощо. Коли ми починаємо з кінця списку і рухаємось назад через список, операція іноді називається **`foldr`** (тобто **`foldright`**) або зменшення.

```
- fun sum nil = 0
    | sum ((h:int)::t) = h + sum t;

val sum = fn : int list -> int
- sum [1,2,3,4,5];
val it = 15 : int

- fun product nil = 1
    | product ((h:int)::t) = h * product t;

val product = fn : int list -> int
- product [1,2,3,4,5];
val it = 120 : int
```

Кожна з цих функцій має однаковий шаблон. Якщо ми абстрагуємо загальний шаблон як функцію вищого порядку, ми отримуємо загальну функцію вищого порядку, яка називається `foldr`. `foldr` - це абревіатура від `foldright`. Функція `foldr` продовжує застосовувати свою функцію до результату та наступного пункту у списку.

```
- fun foldr f init nil = init
    | foldr f init (h::t) = f(h, foldr f init t);

val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr op + 0 [1,2,3,4,5];
val it = 15 : int
- foldr op * 1 [1,2,3,4,5];
val it = 120 : int
```

Тепер суму і добуток можна визначити в термінах зменшення.

```
- val sumlist = List.foldr (op +) 0;
val sumlist = fn : int list -> int
- val mullist = List.foldr op * 1;
val mullist = fn : int list -> int
- sumlist [1,2,3,4,5];
val it = 15 : int
- mullist [1,2,3,4,5];
val it = 120 : int
```

SML включає дві заздалегідь визначені функції, які зменшують список, **foldr** та **foldl**, що означає **foldleft**. Вони поводяться дещо інакше.

```
- List.foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- List.foldl;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- fun abdiff (m,n:int) = abs(m-n);
val abdiff = fn : int * int -> int
- foldr abdiff 0 [1,2,3,4,5];
val it = 1 : int
- foldl abdiff 0 [1,2,3,4,5];
val it = 3 : int
```

Фільтр

Функція предикат - це функція, яка приймає значення і повертає `true` або `false` залежно від значення. Передаючи функцію предиката, можна фільтрувати лише ті елементи зі списку, які задовольняють предикат. Це загальновживана функція вищого порядку, яка називається `filter`. Якби нам довелося писати фільтр самому, так би це було написано. Цей приклад також показує, як його можна використовувати.

```
- fun filter bfun nil = nil
    | filter bfun (h::t) = if bfun h then h::filter bfun t
                           else filter bfun t;

val it = fn : ('a -> bool) -> 'a list -> 'a list
- even;
val it = fn : int -> bool
- filter even [1,2,3,4,5,6];
val it = [2,4,6] : int list
- filter (fn n => n > 3) [1,2,3,4,5,6];
val it = [4,5,6] : int list
```

Стиль проходження продовження

Стиль передачі продовження (або CPS) - це спосіб написання функціональних програм, де управління чітко визначене. Іншими словами, продовження представляє роботу, що залишилася, яку потрібно зробити. Цей стиль написання коду цікавий тим, що цей стиль використовується в компіляторі SML. Щоб зрозуміти CPS, найкраще подивитися приклад. Давайте розглянемо функцію **`len`** для обчислення довжини списку.

```
- fun len nil = 0
    | len (h::t) = 1+(len t);
val len = fn : 'a list -> int
```

Щоб перетворити це у форму cps, ми представляємо решту обчислень явно як параметр, який називається **k**. Таким чином, коли нам потрібно продовження обчислення, ми можемо просто записати ідентифікатор **k**. Ось форма cps **len** та приклад його виклику.

```
- fun cpslen nil k = k 0
    | cpslen (h::t) k = cpslen t (fn v => (k (1 + v)));
val cpslen = fn : 'a list -> (int -> 'b) -> 'b
- cpslen [1,2,3] (fn v => v);
val it = 3 : int
```

Зверніть увагу, що рекурсивний виклик **`cpslen`** - це останнє, що зроблено. Ця функція є рекурсивною. Однак усунення рекурсії хвоста не можна застосувати, оскільки функція повертає функцію і рекурсивно викликає себе з функцією як параметром. CPS все ще важливий, оскільки його може оптимізувати компілятор. Крім того, оскільки потік управління є явним (передається як **`k`**), виклики функцій можуть бути реалізовані за допомогою стрибків, а багато стрибків можуть бути усунені, якщо код організовано правильно.

Усунення дзвінків та стрибків важливо, оскільки дзвінки мають наслідком переривання конвеєрів у процесорах RISC. Оскільки функціональні мови здійснюють багато дзвінків, одна з критик функціональних мов полягає в тому, що вони були неефективними. З оптимізацією функцій CPS функціональні мови наближаються до настільки ж ефективних, як імперативні мови. Крім того, із збільшенням розміру кеш-пам’яті та швидкості процесора різниця в продуктивності стає дедалі меншою проблемою.

Введення і виведення

SML містить структуру **TextIO** як частину базової бібліотеки. Підпис функцій у структурі **TextIO** наведено в Додатку В. Можна читати та записувати рядки в потоки, використовуючи цю бібліотеку функцій. Звичайний стандартний вхід, стандартний вихід і стандартні потоки помилок заздалегідь визначені. Ось приклад читання рядка з клавіатури. **Explode** використовується на рядку, щоб показати, що векторний тип справді є рядковим. Він також показує, як надрукувати щось у потоці.

```
- val s = TextIO.input(TextIO.stdIn);
hi there
val s = "hi there\n" : vector
- explode(s);
val it = [#"h",#"i",#" ",#"t",#"h",#"e",
            #"r",#"e",#"\n"] : char list
- TextIO.output(TextIO.stdOut,s^"How are you!\n");
hi there
How are you!
- val it = () : unit
```

Оскільки потоки можуть бути спрямовані у файли, на екран або по всій мережі, насправді в SML не так багато введення та виведення. Звичайно, якщо ви відкриваєте власний потік, його слід закрити, коли ви закінчите з ним. Завершення програми також закриє будь-які відкриті потоки.

Є деякі функції **`TextIO`**, які можуть повертати значення, а може і не повертати. У цих випадках опція повертається. Варіант - це значення, яке має значення **`NONE`** або **`NOME`**. Варіант - це спосіб SML мати справу з функціями, які можуть або не матимуть успіху. Функції завжди повинні повертати значення або закінчуватись за винятком.

Щоб запобігти використанню механізму обробки винятків для операцій введення, які можуть або не матимуть успіху, було створено цю ідею опції. Параметри чудово вписуються в сильний набір тексту, який надає SML. Функція **`input1`** структури **`TextIO`** зчитує рівно один символ із вводу і в результаті повертає опцію. Причиною того, що він повертає опцію, а не символ безпосередньо, є те, що потік може бути не готовим до читання. Функцію **`valOf`** можна використовувати для отримання значення опції, яка не є **`NONE`**.

```
- val u = TextIO.input1(TextIO.stdIn);

hi there
val u = SOME #"h" : elem option
=
= ^C
Interrupt
- u;
val it = SOME #"h" : elem option
- val v = valOf(u);
val v = #"h" : elem
```

Програмування з побічними ефектами

StandardML - це не чисто функціональна мова. Можна писати програми з побічними ефектами, такими як читання та запис у потоки. Для написання імперативних програм мова повинна підтримувати послідовне виконання, змінні та, можливо, цикли. Усі ці три функції доступні в SML. У наступних розділах показано, як використовувати кожну з цих функцій.

Змінні в StandardML

У StandardML є лише один тип змінних. Змінні називаються посиланнями. Цікаво зауважити, що ви не можете оновити цілі числа, реальні, рядкові чи багато інших типів значень у SML. Всі ці значення незмінні. Після створення їх не можна змінити. Це приємна особливість мови, оскільки тоді вам не доведеться турбуватися про різницю між посиланням на значення та самим значенням. Об'єкти масиву можна змінювати, оскільки вони містять список посилань.

Посилання в StandardML набирається. Це або посилання на **`int`**, або рядок, або інший тип даних. Посилання можна мутувати. Тож посилання може бути оновлено, щоб вказати на нове значення під час виконання вашої програми. Оголошення та використання посильної змінної показано в цьому прикладі коду. У SML змінна оголошується шляхом створення посилання на значення певного типу.

```
- val x = ref 0;
val x = ref 0 : int ref
```

Знак оклику використовується для позначення значення, на яке вказує посилання. Це називається оператором розвідки. Це схоже на зірку (тобто *) в С ++, яка розмежовує покажчик.

```
- !x;
val it = 0 : int
- x := !x + 1;
val it = () : unit
- !x;
val it = 1 : int
```

```
1  let val x = ref 0
2  in
3    x:= !x + 1;
4    TextIO.output(TextIO.stdOut,"The new value of x is "^
5                  Int.toString(!x)^"\n");
6    !x
7  end
```

Рис. 11.7 Послідовне виконання

Оператор присвоєння (тобто `:=`) оновлює посилальну змінну, щоб вказати на нове значення. Результатом присвоєння є порожній кортеж, який має спеціальний тип, який називається unit. Обов'язкове програмування в SML часто призводить до типу одиниці. На відміну від звичайних ідентифікаторів, ви можете прив'язати до значень, використовуючи **`let val id = Expr in Expr end`**, посилання справді може бути оновлене, щоб вказати на нове значення.

Слід зазначити, що посилання в StandardML набираються. Коли посилання створюється, воно може вказувати лише на значення того самого типу, на яке спочатку було створено посилання. Це на відміну від посилань у Python, але подібне до посилань у Java. Посилання стосується певного типу даних.

Послідовне виконання

Якщо програма збирається присвоювати змінним нові значення або читати з потоків і писати їх, вона повинна мати можливість виконувати оператори або вирази послідовно. Існує два способи записати послідовність виразів у SML. Коли ви пишете **`let val id = Expr in end expr`**, **`Expr`** між входом і кінцем може бути послідовністю виразів. Послідовність виразів розділяється крапкою з комою. Код на рис. 11.7 демонструє, як записати послідовність виразів.

Обчислення цього виразу дає такий результат:

```
The new value of x is 1
val it = 1 : int
```

На рис. 11.7 крапки з комою відокремлюють вирази в послідовності. Зверніть увагу, що крапка з комою не закінчує кожен рядок, як у C++ або Java. Крапка з комою в SML - це роздільники виразів, а не термінатори тверджень. Останній вираз у послідовності виразів - це значення виразу. Усі раніше обчислені значення у послідовному виразі викидаються. `!x` - останній вираз у послідовності на рис. 11.7, тому 1 подається як значення виразу.

Бувають випадки, коли вам може знадобитися оцінити послідовність виразів за відсутності дозволеного виразу. У цьому випадку послідовність виразів може бути оточена дужками. Ліва дужка може розпочати послідовність виразів, закінчених правою дужкою. Послідовність виразів розділяється крапкою з комою в кожному випадку. Ось код, який виводить на екран значення **x**, а потім повертає **x + 1**.

```
(TextIO.output(TextIO.stdOut,"The value of x is" ^
 Int.toString(x);
 x+1)
```

Ітерація

Строго кажучи, змінні та ітерації не потрібні у функціональній мові. Параметри можна передавати замість оголошень змінних. Рекурсію можна використовувати замість ітерації. Однак бувають випадки, коли ітераційна функція може мати більше сенсу. Наприклад, під час читання з потоку може бути ефективнішим читати потік у циклі, особливо коли потік може бути великим. У цьому випадку рекурсивна функція може переповнювати стек, якщо тільки рекурсивна функція не є рекурсивною і може бути оптимізована для видалення рекурсивного виклику.

Цикл **`while`** у SML пишеться як **`while Expr do Expr`**. Як зазвичай у циклах **`while`**, перший вираз має бути оцінений як логічне значення. Якщо воно оцінюється як істина, тоді оцінюється другий вираз. Цей процес повторюється, доки перший вираз не поверне значення **`false`**.

Обробка винятків

Виняток виникає в SML, коли виникає стан, який вимагає спеціального поводження. Якщо спеціальна обробка для умови не визначена, програма завершується. Як і в більшості сучасних мов, SML має можливості для обробки цих винятків та для створення визначених користувачем винятків. Розглянемо функцію **`maxIntList`**, яку ви повинні написати в лабораторній роботі №5 у завданні 5.13. Вам, мабуть, довелося розібратися, що робити, якщо до функції передано порожній список. Один із способів вирішити це - створити виняток.

```
exception emptyList;

fun maxIntList [] = raise emptyList
  | maxIntList (h::t) = Int.max(h,maxIntList t) handle
                                     emptyList => h
```

Виклик **maxIntList** у порожньому списку можна обробити, використовуючи вираз обробки винятків. У реченні дескриптора використовується відповідність шаблону для відповідності правильному обробнику виключень. Для обробки будь-яких винятків може бути використаний шаблон **_**. Підкреслення відповідає будь-чому. Кілька винятків можна обробити, використовуючи вертикальну смужку (тобто **|**) між обробниками.

Інкапсуляція в ML

ML пропонує дві мовні конструкції, які дозволяють програмістам визначати нові типи даних і приховувати деталі їх реалізації. Перша з цих мовних конструкцій, яку ми розглянемо, - це сігнатура. Інша конструкція - це структура.

Сігнатури

Підпис - це засіб для визначення набору пов'язаних функцій і типів без надання деталей реалізації. Це аналогічно інтерфейсу в Java або шаблону в C ++. Розглянемо тип даних, що складається з набору елементів. Набір - це група елементів без повторюваних значень. Набори дуже важливі в багатьох областях інформатики та математики. Теорія множин - ціла галузь математики. Якби ми хотіли визначити набір в ML, ми могли б написати підпис для нього наступним чином.

Сігнатура групи набору функцій та набору типів даних наведено на рис. 11.8. Зверніть увагу, що цей тип даних параметризований змінною типу, тому це може бути сігнатурою `'` для набору чого-небудь. Ви також помітите, що, хоча параметр типу а, у сігнатурі є змінні типу з іменем **`a`**. Це тому, що деякі з цих функцій покладаються на оператор дорівнює. У ML оператор дорівнює поліморфним і не може бути інстанційований до типу. Коли ця сігнатура використовується на практиці, типи **`'a`** та **`''a`** будуть правильно інстанційовані до того самого типу.
Перед використанням підпису кожна з цих функцій повинна бути реалізована у структурі, що реалізує підпис. Ця інкапсуляція дозволяє програмісту писати код, який використовує цей набір функцій, незалежно від їх реалізації.

```
1  signature SetSig =
2  sig
3     exception Choiceset
4     exception Restset
5     datatype 'a set = Set of 'a list
6     val emptyset    : 'a set
7     val singleton   : 'a -> 'a set
8     val member      : ''a -> ''a set -> bool
9     val union       : ''a set -> ''a set -> ''a set
10    val intersect   : ''a set -> ''a set -> ''a set
11    val setdif      : ''a set -> ''a set -> ''a set
12    val card        : 'a set -> int
13    val subset      : ''a set -> ''a set -> bool
14    val simetdif    : ''a set -> ''a set -> ''a set
15    val forall      : ''a set -> (''a -> bool) -> bool
16    val forsome     : ''a set -> (''a -> bool) -> bool
17    val forsomeone  : 'a set -> ('a -> bool) -> bool
18 end
```

Рис.11.8 Набір сігнатури

Перед запуском програми необхідно надати реалізацію. Проте, якщо краще реалізація приходить пізніше він може бути замінений без зміни будь-якого коду, який використовує набір сигнатур.

Імплементація сігнатури

Для реалізації підпису ми можемо використовувати структуру struct, яку ми бачили раніше. У цьому випадку це робиться наступним чином. Часткова реалізація сігнатури `SetSig` представлена на рис. 11.9.

```
(***** An Implementation of Sets as a SML datatype *****)

structure Set : SetSig =
struct

exception Choiceset
exception Restset

datatype 'a set = Set of 'a list

val emptyset = Set []

fun singleton e = Set [e]

fun member e (Set [])      = false
  | member e (Set (h::t)) = (e = h) orelse member e (Set t)

fun notmember element st = not (member element st)

fun union (s1 as Set L1) (s2 as Set L2) =
   let fun noDup e = notmember e s2
    in
      Set ((List.filter noDup L1)@(L2))
   end

...
end
```

Рис.11.9 Множина структури

Тип виведення

Мабуть, найсильнішим місцем StandardML є формально доведена надійність системи виведення типу. Система виводу типу ML гарантовано запобігає виникненню будь-яких помилок типу виконання під час виконання програми. Це виявляється для запобігання виникненню багатьох помилок під час виконання у ваших програмах. Такі проекти, як Fox Project, показали, що ML можна використовувати для створення високонадійних великих програмних систем.

Витоки виведення типу включають Хаскелла Каррі та Роберта Фейса, які в 1958 році розробили алгоритм виведення типу для просто набраного лямбда-числення. У 1969 році Роджер Хіндлі працював над розширенням цього типу алгоритму умовиводу. У 1978 році Робін Мілнер незалежно від Хіндлі розробив систему висновків подібного типу, що підтверджує її надійність. У 1985 році Луїс Дамас довів, що алгоритм Міллера є повноцінним, і розширив його для підтримки поліморфних посилань. Цей алгоритм називається алгоритмом виведення типу Хіндлі-Мілнера або алгоритмом Мілнера-Дамаса. Система виведення типів базується на дуже потужній концепції, яка називається уніфікація.

Уніфікація - це процес використання правил виведення типу для прив'язки змінних типу до значень. Правила виведення типу виглядають так.

IfThen

$$\frac{\varepsilon \vdash e_1 : bool \quad \varepsilon \vdash e_2 : \alpha \quad \varepsilon \vdash e_3 : \alpha}{\varepsilon \vdash if\ e_1\ then\ e_2\ else\ e_3 : \alpha}$$

Це правило говорить, що для того, щоб вираз **if-then** був правильно введений, тип першого виразу повинен бути **bool**, а типи другого та третього виразів мають бути об'єднаними. Якщо ці передумови виконуються, то тип виразу **if-then** задається типом будь-якого з двох других виразів (оскільки вони однакові).
Об'єднання відбувається, коли **α** пишеться двічі у наведеному вище правилі. **ε** - це наявність інформації про тип, яка використовується при визначенні типів трьох виразів і називається середовищем типу.
Ось два приклади, які підказують, як працює механізм виведення типу. У цьому прикладі ми визначаємо тип наступної функції.

```
fun f(nil,nil) = nil
  | f(x::xs,y::ys) = (x,y)::f(xs,ys);
```

Функція **f** приймає один параметр, пару.

f : ’a * ’b -> ’c

З характеру аргументації ми робимо висновок, що три невідомі типи повинні бути списками.

```
f: ('p list) * ('s list) -> 't list
```

Функція не накладає обмежень на списки доменів, але список кодоменів повинен бути списком пар через операцію **cons(x,y)::**. Ми знаємо **x:’p** та **y:’s**. Тому **`t = `p * `s**.

```
f: 'p list * 's list -> ('p * 's) list
```

де **'p** та **'s** - це будь-які типи ML. У цьому прикладі виведено тип функції **g**.

```
fun g h x = if null x then nil
            else
              if h (hd x) then g h (tl x)
              else (hd x)::g h (tl x);
```

Функція **g** приймає два параметри, по одному.

g : 'a -> 'b -> 'c

Другий параметр, **x**, повинен служити аргументом для **null**, **hd** і **tl**; це повинен бути список.

g : 'a -> (' s list) -> 'c

Перший параметр **h** повинен бути функцією, оскільки він застосовується до **hd x**, а тип його домену повинен узгоджуватися з типом елементів у списку. Окрім того, **h** має дати логічний результат через його використання в умовному виразі.

`g : ( 's -> bool ) -> ( 's list ) -> 'c`

Результатом функції повинен бути список, оскільки базовий випадок повертає нуль. Список результатів будується за кодом `(hd x):: g h (tl x)`, який додає елементи типу `'s` до отриманого списку.

Отже, тип `g` повинен бути:

`g : ( 's -> bool ) -> 's list -> s list`

Створення інтерпретатора префіксного калькулятора

Визначення типу даних на рис. 10.12 з лекції №10 надало абстрактне визначення дерева синтаксису для мови калькулятора з одним місцем пам'яті. Мову вираження калькулятора, що пов’язана з префіксом, порівняно легко визначити, і з цього ми можемо створити інтерпретатор виразів калькулятора префіксів. Префіксні вирази складаються з оператора, за яким слідують вираз або вирази. Мова вираження калькулятора префіксів визначається цією граматикою LL (1).

$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$
$\mathcal{T} = \{S, R, number, \ , +, -, *, /\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow +\ E\ E \mid -\ E\ E \mid *\ E\ E \mid /\ E\ E \mid \sim E \mid S\ E \mid R \mid number$$

```
1  fun delimiter #" " = true
2    | delimiter #"\t" = true
3    | delimiter #"\n" = true
4    | delimiter _ = false
5
6  fun run() =
7    (TextIO.output(TextIO.stdOut,"Please enter a prefix calculator expression: ");
8     TextIO.flushOut(TextIO.stdOut);
9     let val line = TextIO.inputLine(TextIO.stdIn)
10        val tokens = String.tokens delimiter (valOf line)
11        val (ast,remainingTokens) = E(tokens)
12        val result = eval(ast)
13    in
14      if length(remainingTokens) <> 0 then
15        raise(eofException)
16      else ();
17      TextIO.output(TextIO.stdOut,"The answer is: "^Int.toString(result)^"\n")
18    end
19    handle eofException =>
20              TextIO.output(TextIO.stdOut,
21                "You entered an invalid prefix expression.\n")
22           | Option =>
23              TextIO.output(TextIO.stdOut,
24                "You entered invalid characters in the prefix expression.\n"))
```
Рис.11.11 Функція запуску інтерпретатора префіксного калькулятора

Єдиним нетерміналом у цій граматиці є **E**. **S** є оператором сховища, який зберігає вираз, що слідує за ним, у місці пам'яті. **R** - оператор відкликання. **~** Тильда (тобто) є одинарним оператором заперечення. Для реалізації інтерпретатора для цієї мови ми повинні спочатку проаналізувати вираз і побудувати абстрактне дерево синтаксису. Тоді можна оцінити дерево абстрактного синтаксису. Весь процес можна інкапсулювати у функцію запуску.

Функція запуску, яка забезпечує загальний дизайн інтерпретатора калькулятора префіксів, представлена на рис. 11.11.

Про цей код слід пояснити ряд речей. Рядок 8 змиває стандартний вихід. Без цього запит не друкується до того, як програма почне чекати введення. Рядок 9 отримує рядок вводу від користувача. Він повертається як опція рядка, тому у рядку 10 застосовується функція **valOf**, щоб отримати рядок або викликати виняток **Option**, якщо не було повернуто **NONE**.

Рядок 10 викликає функцію токенів. Усі маркери повинні бути розділені пробілами або вкладками, щоб програма могла правильно читати маркери. Ось приклад запуску цього коду.

```
- run();
Please enter a prefix calculator expression: + * S ~ 6 R 5
The answer is: 41
val it = () : unit
```

Рядок 11 викликає синтаксичний аналізатор для аналізу списку лексем. У цьому випадку список токенів передається функції аналізу. Синтаксичний аналізатор повертає кортеж з AST як перший елемент кортежу, а решта маркерів як другий результат. Після синтаксичного аналізу рядок 14 перевіряє, чи не залишилося більше маркерів після аналізу. Якщо такі є, тоді **eofException** піднімається.

Рядок 12 викликає функцію оцінки `eval` для інтерпретації AST. Функція `eval` повертає результат оцінки дерева. Рядок 17 друкує результат на екран. Є два винятки. Якщо `eofException` викинуто, вираз неправильно проаналізовано. Якщо вилучено виняток `Option`, у вході був поганий маркер. Зверніть увагу, що в цій реалізації для чисел дозволені лише цілі числа. Це було визначено визначенням AST на рис. 10.12 в лекції №10.

Prefix Calc Parser

Проаналізувати вираз легко завдяки граматиці LL (1) для виразів калькулятора префіксів. Функція **E** визначається за допомогою відповідності шаблонів на рис. 11.12. Щоразу, коли маркер споживається, його просто опускають із решти списку маркерів. Маркери однопотокові через функцію. Це просто означає, що залишені маркери завжди передаються наступному фрагменту, який потрібно проаналізувати, а решта маркери завжди повертаються разом з AST, коли повертається функція **E**.

Аналізатор не проводить жодної оцінки даних. Це просто працює над створенням AST для виразу. Оцінка AST приходить пізніше, оцінювачем. Зверніть увагу на рядок 39, що функція **valOf** використовується для результату функції **Int.fromString**. Якщо рядок, що перетворюється, не є дійсним значенням, **valOf** піднять виняток **Option**, завершуючи функцію запуску відповідним повідомленням про помилку.

Рядок 43 на рис. 11.12 обробляє потрапляння до кінця введення (тобто списку токенів) раніше, ніж очікувалося. Якщо парсер досягне цього випадку, початковий вираз був неправильно сформований, і викидання **eofException** є відповідною відповіддю.

```
1  exception eofException;
2
3  fun E ("+"::rest) =
4     let val (ast1,rest1) = E(rest)
5         val (ast2,rest2) = E(rest1)
6     in
7         (add'(ast1,ast2),rest2)
8     end
9     | E ("-"::rest) =
10    let val (ast1,rest1) = E(rest)
11        val (ast2,rest2) = E(rest1)
12    in
13        (sub'(ast1,ast2),rest2)
14    end
15    | E ("*"::rest) =
16    let val (ast1,rest1) = E(rest)
17        val (ast2,rest2) = E(rest1)
18    in
19        (prod'(ast1,ast2),rest2)
20    end
```

Рис.11.12. Парсер

```
21     | E ("/"::rest) =
22     let val (ast1,rest1) = E(rest)
23       val (ast2,rest2) = E(rest1)
24     in
25       (div'(ast1,ast2),rest2)
26     end
27     | E ("~"::rest) =
28     let val (ast,rest1) = E(rest)
29     in
30       (negate'(ast),rest1)
31     end
32      | E ("S"::rest) =
33     let val (ast,rest1) = E(rest)
34     in
35       (store'(ast),rest1)
36     end
37     | E ("R"::rest) = (recall',rest)
38     | E (x::rest) =
39       let val i = valOf(Int.fromString(x))
40       in
41         (integer'(i),rest)
42       end
43     | E nil = raise eofException;
```

Рис.11.13 Evaluator

```
1  val memory = ref 0;
2  fun eval(add'(t1,t2)) =
3          eval(t1) + eval(t2)
4     | eval(sub'(t1,t2)) =
5          eval(t1) - eval(t2)
6     | eval(prod'(t1,t2)) =
7          eval(t1) * eval(t2)
8     | eval(div'(t1,t2)) =
9          eval(t1) div eval(t2)
10    | eval(negate'(t)) =
11          ~1 * eval(t)
12    | eval(store'(t)) =
13       let val x = eval(t)
14       in
15          memory := x;
16          x
17       end
18    | eval(recall') = !memory
19    | eval(integer'(x)) = x
```

Огляд імперативного програмування

Є кілька проблем із синтаксисом StandardML, які на даний момент добре розпізнати. На рис. 11.11 рядок 7 починається з лівого рівня. Ліву дужку можна використовувати для побудови кортежу в StandardML, але він також використовується для початку послідовності виразів. Останній правий елемент у рядку 24 закінчує послідовність. Вирази відокремлюються крапкою з комою в послідовності виразів. Це відбувається в рядку 16 на рис. 5.24. Після виразу в рядку 17 не відображається крапка з комою, оскільки крапка з комою лише відокремлює вирази, вони не закінчують їх. У рядку 16 пропозиція `else` має одиницю (тобто ()) як результат.

Це пов’язано з тим, що тип, який генерується за допомогою винятку, є одиницею, і типи повернення пропозиції and `else` повинні збігатися.

Імплементація функціонального програмування. Висновки

У цьому розділі було представлено функціональне програмування. Для багатьох це новий спосіб думати про програмування. Рекурсія - основний шаблон, який використовується в обчислювальних процесах при написанні у функціональному стилі програмування. Функції вищого порядку є важливою частиною функціонального програмування. Певні закономірності часто з'являються у функціональних програмах і ці шаблони були реалізовані як деякі загальні функції вищого порядку, такі як **`map`**, **`filter`**, **`foldr`** та інші.

Важливо винести з цих лекцій, що функціональне програмування є більш декларативним і менш розпорядчим, ніж програмування на такій імперативній мові, як C ++ або Java. StandardML - це хороша мова функціонального програмування, але інші мови, такі як C ++, Java та Python, також підтримують функціональне програмування. StandardML має потужну перевірку типу, яка перевірена часом і надійна. Це означає, що хоча більше часу витрачається на видалення помилок типу з програм, набагато менше витрачається на налагодження програм StandardML. Експерименти, такі як FoxProject на Carnegie Mellon, показали, що це вірно і для великих програмних систем, написаних на StandardML.

Про StandardML можна дізнатись набагато більше, і наступний розділ не лише розглядає деякі інструменти StandardML для реалізації мови, але також описує реалізацію компілятора, який перекладає StandardML на мову збірки JCoCo.

Книга Джеффрі Уллмана про функціональне програмування в StandardML є дуже вдалим вступом та посиланням на StandardML. Він є більш ґрунтовним, ніж теми, наведені в цьому тексті, і містить безліч тем, які тут не висвітлюються, включаючи обговорення масивів, функторів та шейрінгів, а також кілька основних структур.
Теми, представлені тут і в наступному розділі, дають вам гарний вступ до ідей та концепцій, пов'язаних з функціональним програмуванням. Книги Улмана та он-лайн підручники, сторінки посібників - ще один чудовий ресурс для вивчення функціонального програмування.

На наступній лекції буде розглянута компіляція мови Standard ML.