ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 14

Компіляція Standard ML (закінчення)

Весна 2021

Одинарне заперечення

Виявляється, унарне заперечення не реалізовано належним чином у компіляторі **mlcomp**. В даний час можна надрукувати мінус 5. Однак програма на рис. 14.1 повинна скомпілювати та запуститись, але замість цього сканер видаляє **~** як поганий маркер, а замість нього на екран друкується 5. Це не поведінка Standard ML. Тильда служить одинарним оператором заперечення в Standard ML. Щоб це виправити, необхідно внести кілька змін. Починаючи зі сканера, тильда повинна бути розпізнана як власний маркер. Для цього тильда видаляється з маркера **Int** і додається як власний маркер у файл **mlcomp.lex**.

```
{tilde} => (Tokens.Negate(!pos,!pos));
{digit}+  => (Tokens.Int(yytext,!pos,!pos));
```

Наступним буде додавання маркера до специфікації аналізатора. Тож лексеми тепер визначені, як зазначено в **mlcomp.grm.**

```
%term EOF
    | Negate
    | ...
```

Останній біт у **mlcomp.grm** - це написати продукцію, яка використовує маркер **Negate**. Щоб заперечити вираз, ми просто пишемо вираз, який можливо заперечується, як у цьому розряді коду.

```
    | Negate Exp          (negate(Exp))
```

Написання цього продукту вимагає нового визначення вузла для AST у **mlast.sml**. Негативний вузол в AST - це інший вид вираження. Одинарне заперечення може бути представлений визначенням іншого виразу для заперечення наступним чином.

```
| negate of exp
```

```
1  let val x = 5
2  in
3      println ~x
4  end
```

Рис.14.1 test3.sml

Нарешті, щоб завершити правильну реалізацію унарного заперечення, модуль генератора коду повинен бути змінений. Файл **mlcomp.sml** потрібно відредагувати в декількох місцях, щоб додати підтримку унарного заперечення. Вираз **infixexp** - це вузол AST, як вузол заперечення. Пошук **infixexp** у файлі **mlcomp.sml** допомагає визначити, де потрібно внести зміни у **mlcomp.sml**. Перша зміна полягає у функції **nameOf**.

```
| nameOf(infixexp(operator,e1,e2)) = operator
| nameOf(negate(e)) = "~"
```

Наступний збіг знаходиться всередині функції констант, куди потрібно додати цей код.

```
| con(infixexp(operator,t1,t2)) = (con t1) @ (con t2)
| con(negate(e)) = "0" :: (con e)
```

Цей код додає нуль до списку констант. Це пояснюється тим, що для реалізації унарного заперечення згенерований код відніме значення з нуля. Функція **`bindingsOf`** - це наступне місце, де **`infixexp`** з'являється у файлі **`mlcomp.sml`**. Код для написання тут виглядає так.

```
| bindingsOf(infixexp(operator,exp1,exp2),bindings,scope) =
        (bindingsOf(exp1,bindings,scope); bindingsOf(exp2,bindings,scope))
| bindingsOf(negate(exp),bindings,scope) = bindingsOf(exp,bindings,scope)
```

Функція **bindingsOf** шукає будь-які нові прив'язки, введені новим унарним виразом заперечення. Немає нових прив’язок, створених запереченням Unary, тому він просто викликає функцію **bindingsOf** у своєму підвиразі. Функція **codegen** - це наступне місце, де знаходиться **infixexp**, і додається наступний код для створення коду для одинарного заперечення.

```
| codegen(negate(t),outFile,indent,consts,...) =
  let val _ = codegen(int("0"),outFile,indent,consts,...)
      val _ = codegen(t,outFile,indent,consts,...)
  in
    TextIO.output(outFile,indent^"BINARY_SUBTRACT\n")
  end
```

У функції кодегену створюється вузол “підроблений” **`int ("0")`** для завантаження нуля в стек. Потім значення підвиразу завантажується в стек, і інструкція **`BINARY_SUBTRACT`** призводить до обчислення одинарного заперечення. І вкладеним **`funs`**, і функції **`makeFunctions`** також потрібен рядок для одинарного заперечення. В обох випадках код ідентичний і виглядає так:

```
| functions(infixexp(operator,exp1,exp2)) = (functions exp1;functions exp2)
| functions(negate(exp)) = functions exp
```

Код nestedfuns шукає будь-які вкладені функції у виразі. Унарне заперечення не є вкладеною функцією, тому код просто викликає перевірку, викликаючи функцію функції у підвиразі. Функція **`makeFunctions`** генерує певний код для будь-яких вкладених функцій, щоб JCoCo створював об'єкти закриття або функції для будь-яких вкладених функцій. Нарешті, функцію **`writeTerm`** потрібно змінити. Функція **`writeTerm`**, хоча і не потрібна компілятору, корисна при наступних лекціях. Ось код для написання одинарного заперечного терміна.

```
| writeExp(indent,negate(exp)) =
            (print("negate(");
             writeExp(indent,exp);
             print(")"))
```

```
1  Function: main/0
2  Constants: None, 'Match Not Found', 5, 0
3  Locals: x@0
4  Globals: print, ...
5      LOAD_CONST 2
6      STORE_FAST 0
7      LOAD_GLOBAL 0
8      LOAD_CONST 3
9      LOAD_FAST 0
10     BINARY_SUBTRACT
11     CALL_FUNCTION 1
12     POP_TOP
13     LOAD_CONST 0
14     RETURN_VALUE
15 END
```

Рис. 14.2 test3.sml JCoCo код

Кінцевим результатом цих змін є код, як це показано на рис. 14.2. Значення `~ x` обчислюється відніманням від 0. Новий код складається з рядків 8 та 10 у коді JCoCo на рис. 14.2.

If-Then-Else вирази

Порівняти два значення в SML так само просто, як написати **`x<y`**. У JCoCo це передбачає натискання двох значень на стек операндів і виклик інструкції **`COMPARE_OP`**. При порівнянні значень у виразі **`if-then-else`** результат порівняння буде використаний для переходу до тієї чи іншої мітки. Розглянемо малу програму на рис. 14.3. Знову ж таки, цей код дещо відрізняється від Standard ML. Функція введення є унікальною для Small, як і функції print та **`println`**. Функція введення повертає рядок вводу від користувача. Функція друку друкується без символу нового рядка. **`Println`** друкує новий рядок у кінці рядка.

Компіляція коду на рис. 14.3 повинна призвести до появи коду JCoCo на рис. 14.4.
Однак генерація коду для виразів **`if-then-else`** наразі не реалізована. Абстрактне дерево синтаксису для програми на рис. 14.3 включає вузол для виразу **`if-then-else`**, як цей.

```
ifthen(infixexp(">",id("x"),id("y")),id("x"),id("y"))
```

```
let val x = Int.fromString(
              input("Please enter an integer: "))
    val y = Int.fromString(
              input("Please enter an integer: "))
in
  print "The maximum is ";
  println (if x > y then x else y)
end
```

Рис. 14. 3 test4.sml

Рис. 14.4 test4.sml JCoCo код

```
Function: main/0
Constants: None, 'Match Not Found',
  0, "Please enter an integer: ",
  "The maximum is "
Locals: y@1, x@0
Globals: print, fprint, input, int, len,
  type, Exception, funlist, concat
BEGIN
    LOAD_GLOBAL 3
    LOAD_GLOBAL 2
    LOAD_CONST 3
    CALL_FUNCTION 1
    CALL_FUNCTION 1
    STORE_FAST 1
    LOAD_GLOBAL 3
    LOAD_GLOBAL 2
    LOAD_CONST 3
    CALL_FUNCTION 1
    CALL_FUNCTION 1
    STORE_FAST 0
    LOAD_GLOBAL 1
    LOAD_CONST 4
    CALL_FUNCTION 1
    POP_TOP
    LOAD_GLOBAL 0
    LOAD_FAST 1
    LOAD_FAST 0
    COMPARE_OP 4
    POP_JUMP_IF_FALSE L0
    LOAD_FAST 1
    JUMP_FORWARD L1
L0:
    LOAD_FAST 0
L1:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Визначення AST для цієї програми вже є у файлі **`mlast.sml`**, і сканер та синтаксичний аналізатор вже можуть аналізувати вирази **`if-then-else`**. Генерування коду для цього AST передбачає ті самі зміни, які були необхідні для додавання унарного заперечення до генератора коду. Ці кроки можна виконати, щоб додати весь необхідний код для обробки виразів **`if-then-else`** у генераторі коду. Спробувавши скомпілювати код на рис. 14.3, ви виявите місця в компіляторі, де код відсутній. Компілятор написаний, щоб повідомити про відсутність коду. Спробуйте скомпілювати **`test4.sml`**, побачити, де проблема, виправити її та повторити стільки разів, скільки потрібно.

Реалізація кодового коду є найважчою частиною додавання підтримки виразів **`if-then-else`**, але це не надто складно. Вираз AST, наведений вище, має три підвирази: більший за порівняння, ідентифікатор (**`"x"`**) та id (**`"y"`**). Генерація коду вже виконана для ідентифікаторів, тому вузли ідентифікаторів для **`x`** та **`y`** вже оброблені. Генерування коду для виразу **`if-then-else`** передбачає генерацію коду для порівняння, а потім перехід в те чи інше місце залежно від результату порівняння.

Код **`if-then-else`** починається з рядка 26 на рис. 14.4 із кодом порівняння. Виклик кодегену у виразі з інфіксом генерує код у рядках 26–28. У рядку 29 починається частина коду для виразу **`if-then-else`**. Рядок 29 починається з переходу до L0, якщо умова хибна. Мітка L0 позначає речення **`else`** виразу.
Рядок 30 - це код, згенерований для ідентифікатора (**`"x"`**), який є тодішнім виразом. Рядок 31 генерується **`if-then-else`** знову, щоб пропустити минулий код у виразі **`else`**. Рядок 34 - останній біт коду, сформований виразом **`if-then-else`**.

Генератору коду потрібні дві мітки. Функція **`nextLabel`** у **`mlcomp.sml`** призначена саме для цієї мети. Зателефонувавши до нього, ви отримаєте унікальну мітку, яку можна використовувати в коді. Генерація коду для виразів **`if-then-else`** двічі викликає цю функцію. Підсумовуючи, є кілька дій, які мають відбутися для генерації коду для виразів **`if-then-else`**.

- Потрібно створити дві мітки.
- Код порівняння генерується.
- Інструкція **`POP_JUMP_IF_FALSE`** пишеться разом із міткою речення **`else`**.
- Код пропозиції **`then`** генерується.
- Написано перехід, щоб перейти через код пропозиції **`else`**.
- Написано мітку пропозиції **`else`**.
- Код пропозиції **`else`** генерується.
- Остаточна мітка записується у файл.

Успішне заповнення цього коду отримає вирази **`if-then-else`**, які правильно скомпілюються, і **`test4.sml`** запустить друк максимум двох чисел, введених на клавіатурі.

Логіка короткого замикання

Логіка короткого замикання - загальна риса мов програмування. Якщо у вас є два булевих вирази, E1 та E2, і ви хочете знати, чи є вони істинними чи хибними, існують ситуації, коли не потрібно перевіряти обидві умови. Наприклад, при тестуванні E1 та E2, якщо E1 хибний, немає причин оцінювати E2. Так само, якщо оцінювати E1 або E2, якщо E1 відповідає дійсності, немає причин оцінювати E2. Ця логіка називається логікою короткого замикання і зазвичай використовується операторами і та або в мовах програмування. C ++ та Java використовують цю логіку у своїх `&&` та `||` операторів. У StandardML оператори називаються також, а також `orelse`, щоб вказати їх характер короткого замикання.

Ні оператори **andalso**, ні **orelse** не реалізовані в компіляторі **mlcomp**. Підтримку можна додати досить легко, виконавши багато кроків із додавання одинарного заперечення до мови. Ці кроки включають:

- Додайте до сканера два маркери для **andalso** і **orelse**. Обидва вони є ключовими словами і їх слід додати до розділу ключових слів специфікації сканера в **mlcomp.lex**.
- Додайте лексеми до специфікації граматики в **mlcomp.grm** та визначте їх пріоритет. Обидва оператори мають однаковий пріоритет, який знаходиться на тому ж рівні, що і додавання. Вони обидва також лівоасоціативні.

- Додайте до граматики два утворення, щоб вирази могли бути проаналізовані. Продукції повинні повертати вузли AST, як описано далі.
- Впровадити генерацію коду для цих операторів.

Правильно сформований AST для цього коду включатиме такі вузли **`infixexp`**.

```
infixexp("orelse",id("x"),
    infixexp("div",id("y"),int("0")))
infixexp("andalso",id("y"),
    infixexp("*",id("x"),int("5")))
```

Код для рядка 4 на рис. 14.5 починається з рядка 14 на рис. 14.6. Спочатку завантажується функція друку. Звичайно, це вже реалізовано. У рядку 15 починається генерація коду для оператора **`orelse`**. Для виразу E1 або більше E2 спочатку генерується код для E1, потім **`DUP_TOP, POP_JUMP_IF_TRUE`** та інструкції **`POP_TOP`**. Ідея полягає в тому, що якщо перше значення істинне, залиште його в стеку і пропустіть оцінку E2. Однак, якщо значення E1 хибне, виведіть його значення і залиште значення E2 у стеку після виконання коду для E1 або більше E2.

```
1  let val x = true
2      val y = false
3  in
4    println (x orelse y div 0);
5    println (y andalso x * 5)
6  end
```

Рис. 14.5 test5.sml

Рис.14.6 test5.sml JCoCo кодового

```
Function: main/0
Constants: None,
   'Match Not Found',
   True, False, 0, 5
Locals: y@1, x@0
Globals: print, fprint, input,
   int, len, type, Exception,
   funlist, concat
BEGIN
    LOAD_CONST 2
    STORE_FAST 1
    LOAD_CONST 3
    STORE_FAST 0
    LOAD_GLOBAL 0
    LOAD_FAST 1
    DUP_TOP
    POP_JUMP_IF_TRUE L0
    POP_TOP
    LOAD_FAST 0
    LOAD_CONST 4
    BINARY_FLOOR_DIVIDE
L0:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_GLOBAL 0
    LOAD_FAST 0
    DUP_TOP
    POP_JUMP_IF_FALSE L1
    POP_TOP
    LOAD_FAST 1
    LOAD_CONST 5
    BINARY_MULTIPLY
L1:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Мітка потрібна як ціль для інструкції стрибка. Функція **`nextLabel`** повертає унікальну мітку, як було розглянуто в попередній лекції щодо складання виразів **`if-then-else`**.

Код для **`andals`**, що також з'являється в рядках 26–33 на рис. 14.6, є аналогічним перескакуванню **`orelse`**-коду, якщо перше значення хибне, та обчисленням E2, якщо E1 відповідає істині.

Програма на рис. 14.5 представляє певний інтерес, оскільки вона не є допустимою малою програмою, проте компілятор **`mlcomp`** сформує код, і програму можна запустити на віртуальній машині JCoCo. Оскільки логіка короткого замикання перешкоджає оцінці неправильно набраних виразів, помилка ніколи не зустрічається.

В останніх лекціях буде розглянуто, як програма на рис. 14.5 не проходить перевірку типу, розглядаючи, як реалізований алгоритм виведення типу StandardML.

Різниця між Python та StandardML полягає в тому, що Python дозволить запускати подібну програму до тих пір, поки не виникає помилка під час виконання, а Standard ML скаржиться, що вона не проходить перевірку типу та скасовує. Чи висновок типу StandardML кращий за динамічну перевірку типу Python? Висновок типу вловлює багато помилок у логіці. Налагодження більшості програм StandardML є тривіальним порівняно з налагодженням програм Python. Однак проходження перевірки типу часто є складнішим і часто вимагає нудного коду перетворення типу. StandardML трохи кращий у цьому плані, враховуючи його алгоритм поліморфного виведення.

Загалом, такі дослідження, як проект Fox у Карнегі-Меллоні, показали, що великі програмні системи надзвичайно корисні від сильної перевірки типу за рахунок скорочення часу, необхідного для тестування коду.

Компроміс полягає в зручності проти безпеки під час написання коду та кількості часу, витраченого на тестування та налагодження після написання коду. Стандартний ML дещо менш зручний для написання, але витрати на налагодження незначні. Python зручніше писати, але у великій програмній системі ви можете заплатити за нього пізніше. Інші фактори при виборі мови включають відповідність завдання, що розглядається, чи вже був написаний подібний код певною мовою, наявність бібліотек, що надають API, і наявність таких інструментів, як компілятори, інтерпретатори та IDE (тобто інтегровані середовища розробки) . Усі ці фактори необхідно зважити, щоб вирішити, яка мова є найбільш підходящою для проекту.

Визначення функцій

Визначення функцій у Standard ML можуть з’являтися буквально в будь-якому місці програми. Функції є значеннями першого класу і можуть з'являтися скрізь, де може з'являтися декларація. Крім того, анонімні функції можуть з’являтися скрізь, де вираз може з’являтися у програмі SML. У JCoCo не так. Визначення функції віртуальної машини JCoCo можуть надаватися на верхньому рівні, поза будь-якими іншими функціями, або можуть бути вкладеними всередину іншої функції, але повинні бути записані відразу після оператора функції їх зовнішньої функції. Крім того, у JCoCo всі функції мають бути названі.

Немає анонімних функцій. Функція **nestedfuns** проходить AST для виразу SML, шукаючи будь-які визначення функції. Якщо він знаходить його, він негайно генерує код для вкладеної функції. Розглянемо функцію компіляції модуля **mlcomp.sml**.

```
TextIO.output(outFile,"Function: main/0\n");
nestedfuns(ast,outFile,"   ",globals,[],globalBindings,0);
```

Цей код друкує оператор функції для основної функції. Потім він негайно викликав функцію **`nestedfuns`**, щоб шукати будь-які вкладені функції та генерувати їх код, перш ніж продовжувати генерацію коду для основної функції. Знову ж таки, це порядок, який вимагає віртуальна машина JCoCo. Коли визначення вкладеної функції знайдено в AST, функція вкладеної функції викликається для генерації коду для неї. Тут занадто багато коду, щоб включити сюди, але функція **`nestedfun`** збирає інформацію про константи, локалі, змінні клітинки та прив'язки внутрішньої функції перед тим, як викликати **`codegen`**, щоб сформувати її тіло. Звичайно, він також шукає будь-які вкладені функції всередині нього, перш ніж продовжувати.

Коли анонімну функцію знайдено, їй повинно бути присвоєно ім’я, оскільки це потрібно віртуальній машині JCoCo. Присвоєння імені анонімним функціям відбувається у парсері у виробництві анонімних функцій.

```
Fn MatchExp (func(nextIdNum(),MatchExp))
```

```
let fun factorial 0 = 1
      | factorial n = n * (factorial (n-1))
in
  println (factorial 5)
end
```

Рис.14.7 test6.sml

Рис.14.8 test6.sml JCoCo код

```
Function: main/0
    Function: factorial/1
    Constants: None,
        'Match Not Found', 0, 1
    Locals: factorial@Param, n@1
    FreeVars: factorial
    Globals: print, fprint, input,
        int, len, type, Exception,
        funlist, concat
    BEGIN
        LOAD_FAST 0
        LOAD_CONST 2
        COMPARE_OP 2
        POP_JUMP_IF_FALSE L0
        LOAD_CONST 3
        RETURN_VALUE
L0:
        LOAD_FAST 0
        STORE_FAST 1
        LOAD_FAST 1
        LOAD_DEREF 0
        LOAD_FAST 1
        LOAD_CONST 3
        BINARY_SUBTRACT
        CALL_FUNCTION 1
        BINARY_MULTIPLY
        RETURN_VALUE
L1:
        LOAD_GLOBAL 6
        LOAD_CONST 1
        CALL_FUNCTION 1
        RAISE_VARARGS 1
    END
...
```

У цьому коді функція **`nextIdNum`** повертає унікальне ціле число. У генераторі коду це унікальне ціле число використовується для формування імені анонімної функції **`anon@i`**, де **`i`** - унікальне ціле число, призначене парсером.
Визначення функцій завжди визначаються для функцій точно одного аргументу. Зіставлення зразків може бути використано для узгодження аргументу, як у StandardML. Параметр функції відповідає кожному шаблону у визначенні функції. Розглянемо код на рис. 14.7. У визначенні функції є два шаблони, шаблон числа і шаблон ідентифікатора, який завжди відповідає. Функція **`patMatch`** у **`mlcomp.sml`** дбає про генерацію коду, який відповідає аргументу шаблону.

Для шаблону чисел код у рядках 12–14 на рис. 6.33 перевіряє, чи відповідає число. Якщо ні, код переходить до кінця своєї справи. Немає коду для перевірки відповідності шаблону ідентифікатора, оскільки він завжди збігається. Зверніть увагу на код у рядках 28–32 рис. 14.8. Кожного разу, коли викликається код патча, йому передається мітка наступного шаблону, до якого потрібно перейти, якщо поточний шаблон не відповідає. У цьому випадку останній шаблон завжди збігається, але якби його не було, можливо, код перескочив до L1. У такому випадку, оскільки всі шаблони на той момент вичерпані, код викличе виняток. У цій конкретній функції рядки 28–32 є прикладом мертвого коду. Код ніколи не буде досягнутий, і його можна буде видалити.

Функція **`patmatch`** відповідає шаблонам для **`nil`**, **`numbers`**, **`true`** або **`false`**, рядків, ідентифікаторів, оператора **`::cons`** (тобто непустий шаблон списку) та кортежів. Шаблон кортежу, у свою чергу, відповідає кожному елементу шаблону кортежу елементам аргументу кортежу, викликаючи **`patmatch`**.

Функції каррірування

Раніше було сказано, що всі функції є функціями одного аргументу в Small (і в StandardML також), і це правда. Функції каррі - ще один приклад синтаксичного цукру. Виявляється, що функція, що функціонує, є функцією декількох аргументів, де аргументи можуть надаватися по одному. Істина полягає в тому, що функція, що викривляється, перетворюється на низку анонімних функцій, кожна з яких має один аргумент. Розглянемо програму на рис. 14.9.

Функція додавання записана у формі **`curried. appendOne`** - це функція одного аргументу. Під час запуску програми вони обидва роблять те саме, додаючи два списки разом. Виклик **`curried. appendOne`** виглядає однаково. Це тому, що дві функції однакові. Застосування функції залишається асоціативним, тому кожна функція застосовується до свого першого і єдиного аргументу, який повертає функцію, застосовану до другого аргументу.

Синтаксичний аналізатор **mlcomp** зменшує функції **curried**, як додавання до функції одного аргументу з однією анонімною функцією для кожного з **curried** аргументів. Це робиться за допомогою досить складної функції, яка збирає кожне з різних збігів шаблонів функції, що перекривається, і переписує код так, щоб кожен збіг за зразком відповідав зразку точно одного аргументу, що повертає функцію, яка приймає наступний аргумент. Ця функція називається **uncurryIt** і дана на рис. 14.10.

```
let
  fun append nil L = L
    | append (h::t) L =
        h :: (append t L)

  fun appendOne x =
    (fn nil => (fn L => L)
     | h::t => (fn L =>
         h :: (appendOne t L))) x
in
  println(append [1,2,3] [4]);
  println(appendOne [1,2,3] [4])
end
```

Рис.14.9 test7sml

```
fun uncurryIt nil = raise emptyDecList
  | uncurryIt (L as ((name,patList,exp)::t)) =
    let fun len nil = raise argumentMismatch
          | len [(n,p,e)] = length(p)
          | len ((n,p,e)::t) =
            let val size = length(p)
            in
              if size = len t then size else
                (TextIO.output(TextIO.stdOut,
                     "Syntax Error: Number of arguments does not match ...."
                 raise argumentMismatch)
            end

        val tupleList = List.map (fn x => "v"^Int.toString(nextIdNum())) patList
    in
      len(L); (* check that all patterns have same length *)
      (name,[match(idpat(hd(tupleList)),
                List.foldr (fn (x,y) => func(nextIdNum(),[match(idpat(x), y)]))
                   (apply (func(nextIdNum(),List.map (fn (n,p,e) =>
                       match(tuplepat(p),e)) L),
                          tuplecon(List.map
                          (fn x => id(x)) tupleList))) (tl tupleList))])
    end
```

Рис.14.10 Функція uncurryIt

Взаємно-рекурсивні функції

Функції в малому та SML часто є рекурсивними. Іноді функції можуть бути взаємно рекурсивними, як це має місце на рис. 6.36. Функція **`f`** викликає **`g`** і навпаки. У C ++ для написання двох таких функцій потрібна переадресація з використанням прототипу функції принаймні для **`g`**. У Standard ML використання ключового слова **`and`** між двома визначеннями функцій вказує на те, що вони є взаємно рекурсивними функціями. AST для цієї програми визначається так:

```
letdec(funmatches([funmatch("f",f's body),funmatch("g",g's body)]))
```

```
1  let fun f(0,y) = y
2        | f(x,y) = g(x,x*y)
3      and g(x,y) = f(x-1,y)
4  in
5    println (f(10,5))
6  end
```

Рис.14.11 test11.sml

```
| dec(funmatches(L)) =
  let val nameList = List.map (fn (name,matchlist) => name) L
  in
    List.map (fn (name,matchList) =>
    let val adjustedBindings =
        List.map (fn x => (x,x)) (listdiff nameList [name])
    in
      nestedfun(name,matchList,outFile,indent,globals,
          adjustedBindings@env,globalBindings,scope)
    end) L;
    ()
  end
```

Рис.14.12 Взаємно-рекурсивні оголошення функцій

Коли зустрічається вузол AST **`funmatches`**, прив'язки всіх функцій у списку **`funmatches`** передаються генерації коду кожної функції. Це видно у функції **`nestedfuns`** при зіставленні оголошення для фундаменту, як показано на рис. 14.12.
У цьому коді список усіх імен функцій збирається в **`nameList`**, а потім передається кожному рекурсивному виклику **`nestedfun`** після вилучення імені функції, для якої викликається **`nestedfun`**.
Взаємно-рекурсивні функції є більш поширеними, ніж ви можете подумати. Шукайте використання та у файлі **`mlcomp.sml`**, щоб побачити, коли це потрібно у реалізації компілятора.

Довідкові змінні

Додавання змінних до мови Small виявляється майже тривіальним. Розглядаючи рис. 14.13, новий код включає ключове слово **`ref`**, знак оклику, що використовується як оператор перенаправлення, та оператор **`: =`** (вимовляється набір рівним). Сканер включає підтримку деререференції та встановлення рівних операторів. Посилання буде розпізнано як ідентифікатор, що виявляється дуже добре.

Специфікація граматики в **`mlcomp.grm`** вже має підтримку і для розпізнавання, і для встановлення рівних операторів. Постановки для двох представляють певний інтерес.

На рис. 14.14 набір рівних показників вимагає, щоб ідентифікатор знаходився зліва. Змінна не може бути виразом. Якщо посилальна змінна повинна вказувати на нове значення, ліва сторона повинна називати посилальну змінну. Проте AST є **`infixexp`** шляхом створення вузла виразу з ідентифікатора за допомогою **`id (Id)`**. Виробництво розшифровки ще цікавіше створює підроблений вузол програми-функції за допомогою **`!`** ідентифікатор. Для додавання ключового слова **`ref`** виробництво не потрібне, оскільки граматика вже аналізує це як застосування функції функції **`ref`** до значення **`0`**.

Генерація коду для змінних обробляється низкою особливих випадків. Функцію **`decBindingsOf`** потрібно змінити, оскільки прив'язка змінної відрізняється від прив'язки звичайного ідентифікатора. Код на рис. 14.15 повинен бути розміщений перед шаблоном для звичайних ідентифікаторів.
Код на рис. 14.15 прив'язує ім'я змінної до унікального ідентифікатора в програмі JCoCo, і це додає ім'я змінної до списку ідентифікаторів, які будуть пов'язані зі змінними комірки. Змінна комірки - це посилання, а змінні - посилання в StandardML.

Оператор розмежування повинен оброблятися як особливий випадок у функції **bindingsOf**. Зазвичай ідентифікатор шукають, щоб визначити, чи він пов'язаний чи вільний у функції. Синтаксичний аналізатор, згенерований AST для оператора розстановки, робить його схожим на ідентифікатор на рис. 14.14. Щоб вирішити це, наступний код є особливим випадком і повинен з'явитися перед звичайним пошуком ідентифікаторів у функції **bindingsOf**.

```
| bindingsOf(id("!"),bindings,scope) = ()
```

```
1  let val x = ref 0
2  in
3    x := !x + 1;
4    println (!x)
5  end
```

Рис.14.13 test8.sml

```
| Exclaim Exp (apply(id("!"),Exp))
| Id SetEqual FuncExp
              (infixexp(":=",id(Id),FuncExp))
```

Рис.14.14 Set equal and deref operators

```
and decbindingsOf(bindval(idpat(name),apply(id(l+s+s2{r}ef"),exp)),bindings,scope) =
    let val newbindings = patBindings(idpat(name),scope)
    in
      bindingsOf(exp,newbindings@bindings,scope+1);
      addIt(name,cellVars);
      [addIt((name,name^l+s+s2{@}n{^}Int.toString(scope)),theBindings)]
    end
```

Рис.14.15 Прив'язки змінних посилань

```
| codegen(id(name),outFile,...) =
  load(name,outFile,...)
| codegen(apply(id("ref"),t2),outFile,...) =
  codegen(t2,outFile,...)
| codegen(infixexp(":=",id(name),t2),...) =
let val _ = codegen(t2,outFile,...)
      val noneIndex = lookupIndex("None",consts)
in
   store(name,outFile,...);
   TextIO.output(outFile,
       indent^"LOAD_CONST "^noneIndex^"\n")
end
```

Рис. 14.16 Генерація кода змінних

Нарешті, генерація коду повинна бути виконана для декларації **`ref`**, оператора розмежування та встановленого рівного оператора. Створення посилального коду - це ще один особливий випадок, який потрібно робити перед нормальним застосуванням функції. Цікаво, що вся робота з генерації коду фактично була виконана функцією **`decBindingsOf`**, коли змінна була додана до списку змінних комірок. У рядках 1–2 на рис. 14.16 код для виразу **`ref`** є ідентичним коду для виразу без посилання, оскільки функція **`store`** знайде змінну у змінних комірки, а потім згенерує відповідну інструкцію зберігання.

Рядки 3–4 генерують код для переназначення змінної. Побічно це викликає завантаження, яке автоматично генерує відповідну інструкцію завантаження, оскільки функція **`decBindingsOf`** помістила змінну в список змінних комірок. Нарешті, код для встановленого рівного оператора досить простий. Інструкція **`LOAD_CONST`** потрібна, оскільки кожен вираз у Standard ML має результат, а в кінці оператора присвоєння результат вискакується із стеку. Результатом призначення є одиниця, яка перетворюється на значення None у віртуальній машині JCoCo.

Коли прив'язка до ідентифікатора використовується у внутрішній функції, ідентифікатор повинен бути прив'язаний до змінної клітинки, щоб можна було побудувати закриття при виклику внутрішньої функції. Довідкові змінні також прив'язані до змінних комірок, щоб їх можна було оновити.
Наявність двох різних типів прив'язок, обидва вони відповідають одній і тій же реалізації, призводить до деяких цікавих можливостей у коді. Розглянемо програму на рис. 14.17. Ця програма не є правильною невеликою програмою. Прив'язка від `x` до `0` є постійним зв'язуванням. Не повинно бути можливо оновити вміст змінної. Однак оператор присвоєння у рядку 2 працює, оскільки `x` використовується у внутрішній функції `f`, а тому присвоюється змінній комірки.

Код на рис. 14.17 є прикладом того, коли перевірка типу необхідна для запобігання виконанню нелегальної програми. Програма неправильна. Програміст помилився і хотів би знати про цю помилку. Проте JCoCo не хвилює, як і компілятор **`mlcomp`**.

Переглядач типів повинен позначити це як помилку та припинити генератор коду до того, як буде створено будь-яку програму. Цей приклад та необхідність перевірки типу будуть детальніше вивчені в останніх лекціях.

```
let val x = 0
    fun f y = (x:=!x+1)
in
  f 0;
  println x
end
```

Рис.14.17 test9.sml

Деякі висновки

Метою останніх лекцій було надати ознайомлення з особливостями мови шляхом вивчення реалізації мови Small. Бажаючі дізнатися більше про побудову компілятора, можливо, захочуть ознайомитися з повним текстом на цю тему. Наприклад, книга Ахо, Сетхі та Ульмана. Є також багато інших хороших текстів про написання компіляторів.

Тематичне дослідження на цих останніх лекціях проілюструвало кілька особливостей мов програмування. Реалізація функцій у структурованих блоком мовах є чи не найскладнішою з представлених концепцій. Важливі концепції та навички, представлені в цих лекціях, включають обсяг прив'язок та те, як створюються прив'язки, взаємно рекурсивні функції, посилальні змінні, генерацію коду для декількох мовних особливостей, як розширити мову, як використовувати **`ML-lex`** та **`ML-yacc`** , синтаксичний цукор та його використання в малій мові та логіка короткого замикання. Обробка винятків не розглядалась у цьому розділі і є частиною компілятора **`mlcomp`**.

У міру зростання потреби у вбудованих системах зростатиме і попит на нові мови програмування, орієнтовані на ці платформи. Потреби стрімкого робочого середовища також викликали інтерес до проектування та розробки мови програмування. Це хвилюючий час для експертів з мов програмування, і цей текст лише торкається поверхні величезної та захоплюючої області вивчення.

На наступній лекції буде розглянута імплементація логічної мови програмування.