## Робота №09

Процедурна мова програмування

Традиційно інтерпретатор довільної мови програмування виконує декілька кроків в процесі інтерпретації. Протягом цих кроків початкова програма - послідовності символів перетворюється в проміжну форму, котра інтерпретується (виконується):

* **Синтаксичний аналіз**: якщо програма синтаксичну вірна то, в ній виділяються лексичні та синтаксичні конструкції і будується її представлення у вигляді проміжної форми (Абстрактне Синтаксичне Дерево AST - *Abstract Syntax Trees*) – об`єкт типу Program.
* **Перевірка типів**: перевірка чи вірно (*is well formed*) побудовані елементи проміжної форми. Функції ( *iswf-)*  перевіряють виконання різних контекстних умов, котрим повинна задовольняти проміжна форма, інколи при цьому її модифікуючи.
* **Інтерпретація**: виконання проміжної форми.

В цій роботі необхідно реалізувати етапи перевірки типів і інтерпретації для простої процедурної мови програмування, що містить чисті функції та «нечисті» імперативні процедури, котрі можуть змінювати глобальні змінні.

В мові працюють з даними двох типів: цілі або одновимірні масиви цілих. Логічних значень немає - вони кодуються цілими: True кодується довільним цілим, котре не дорівнює 0, а False – 0 (нулем).

Об»єкти, котрі мають імена (ідентифікатори, що визначаються користувачем), - це процедури, функції і змінні / параметри - скалярні або одновимірні масиви.

Всі головні компоненти мови (процедури, оператори, функції) використовують *вирази*. Вираз може бути:

* *Константою*. В програмі можна вживати лише константи – цілі числа.
* З*мінною.* Ідентифікатори змінних визначаються користувачем і починаються з букви , наприклад і x, yy, VAR, cameCase т.д. Є змінні глобальні і локальні, котрі є або параметри або об’являються в блоках.
* *Застосуванням бінарної операції*. Є бінарні операції «додавання», «віднімання», «множення», «менше-ніж», «дорівнює» і «індексування». Операндами перших п’яти, що відповідають операціям +, -, \*, < і == в Haskell, можуть бути лише цілі скаляри (два масиви не порівнюються) і іх результат – цілий (скалярне) значення. Для операцій «менше-ніж» і «дорівнює» результат буде або 0 або 1, котрі представляють False і True відповідно. Операція «індексування» реалізує доступ до елементів масиву: перший операнд - масив і другий - індекс (цілий скаляр).
* *Умовний* вираз має вигляд *тернарного умовного оператору* в Java - *P ? Q : R*, якщо значення *P* є ціле число відмінне від 0, то результат виразу – значення виразу *Q,* в іншому випадку значення виразу *R*. Вирази *Q* і *R* повинні повертати значення одного типу (або ціле або одновимірний масив цілих).
* *Застосування функції визначеної користувачем* – вираз виду f (e1, e2, …, en), де f – ім»я функції, а e1, e2, …, en – вирази, що задають аргументи функції.

Визначення функції включає ім’я функції f, список об’яв параметрів p1, p2, …, pn і вираз e з правого боку - на зразок *func* *f (p1, p2, …, pn) = e*. Об’ява параметру *x* або *x[]*, де x – ім’я параметру. В цілому скалярному виразі *e* можуть використовуватися лише параметри функції *p1, p2, …, pn* і невикористовуються глобальні змінні*.*

На абстрактному рівні вирази і визначення функцій в Haskell описуються типами:

***type*** Id = String

***data*** Value = I Int | A [(Int, Int)]

***deriving*** (Eq, Show)

***data*** Op = Add | Minus | Mul | Less | Equal | Index

***deriving*** (Eq, Show)

***data*** Exp = Const Int

| Var Id

| OpApp Op Exp Exp

| Cond Exp Exp Exp

| FunApp Id [Exp]

***deriving*** (Eq, Show)

***data*** VarDef = Arr Id | Int Id ***deriving*** (Eq, Show)

***type*** FunDef = (Id, ([VarDef], Exp))

Значення цілих і масивів розрізняються конструкторами I і A в типі даних Value*.* Елементи масиву задаються списком пар, що відображають індекс масиву в значення. Наприклад:

* Масив представлений A [(3,1), (0,3), (5,12)] має значення, що визначені в позиціях 0, 3 і 5
* Операція «індексування» цього масиву з індексами 0, 3 і 5 поверне значення 3, 1 і 12 відповідно
* Порядок пар в списку не обов»язковий. Індекси не утворюють неперервний сегмент.
* Не повинно бути дублікатів входів для окремих індексів: масив зі значеннями, що визначені в n (унікальних) індексах повинен бути представлений списком точно з n пар елементів.
* Значення масиву в індексах, що не вказані, припускається нуль, тому при спробі індексувати масив A, наведений раніше, в позиціях 1, 2, 4, 6, 7, … буде повернуто 0.
  + В цьому відношенні масиви в мові – *необмежені* навіть у випадку їх скінченного представлення.

Приклади означень функцій:

* Функція biggest, що повертає більше з двох чисел m і n

*func biggest(m, n) = (m < n ? n : m)*

* Ця функція представляється наступним FunDef:

biggest :: FunDef

biggest =

("biggest",

([Int "m", Int "n"],

Cond (OpApp Less (Var "m") (Var "n")) (Var "n") (Var "m")

)

)

* Функція fib, що обчислює число Фібоначчі

*func fib(n) = (n < 3 ? 1 : fib(n-1) + fib(n-2))*

* Її можна представити так

fib :: FunDef

fib =

("fib",

([Int "n"], Cond (OpApp Less (Var "n") (Const 3))

(Const 1)

(OpApp Add

(FunApp "fib" [OpApp Minus (Var "n") (Const 1)])

(FunApp "fib" [OpApp Minus (Var "n") (Const 2)]))

)

)

* Наступна функція демонструє індексування масиву - sumA додає елементи 0 …n масиву A

*func sumA(a[], n) = (n < 0 ? 0 : a[n] + sumA (a, n-1))*

* Її можна представити

sumA :: FunDef

sumA =

("sumA",

([Arr "a", Int "n"],

Cond (OpApp Less (Var "n") (Const 0))

(Const 0)

(OpApp Add (OpApp Index (Var "a") (Var "n"))

(FunApp "sumA"

[Var "a", OpApp Minus (Var "n")(Const 1)]))

)

)

Інтерпретатор виразів (функція, що обчислює значення вираз) повинен працювати з оточення – список означень функції (FunDef), введених користувачем, для того щоб реалізувати застосування функції, i *станом*, що надає значення аргументів функції – локальні дані.

Результат обрахування виразу – об’єкт типу Value, котрий може бути або ціле вигляду *I n* для деякого цілого n, або масив виду *A elems*, де elems - список відображення індексів у значення.

В загальному випадку, стан (стек даних) може містити значення як локальних так і глобальних змінних:

***type*** StateP = [(Id, Value)]

Залишок мови процедурний і його можна уявити як дуже спрощену версію Java, без об’єктів. Об’ява процедура (на зразок - *proc* *np(p1, p2, …, pn) stmt*) включає ім’я процедури *np* та список об’яв параметрів *p1, p2, …, pn*, як для функції. Різниця в тому, що права частина процедури є оператор *stmt.* Оператор може бути:

* *Присвоювання змінній x*, *x = e*, де e – вираз.
* *Присвоювання елементу масиву a*, *a[i] = e*, де *a* - масив, *i* - цілий індекс і *e* - вираз.
  + Змінна *x* (*присвоювання змінній*) і масив *a* (*присвоювання елементу масиву*) вже повинні бути об»явлені (в області дії імен) як параметр процедури або як змінна локальна або глобальна.
* *Оператор ‘if’*, еквівалентний оператору Java виду *if (P) {S1} else {S2}*, де P цілий вираз, значення якого *(I v)* інтерпретується як True, коли *v* не дорівнює 0, і як False, коли *v* є 0. S1 та S2 - оператори.
  + ‘if’ *оператори* не потрібно змішувати з умовними *виразами*.
* *Цикл while ,* *while (P) S*, де *P* інтерпретується як в операторі *if,* *S* - оператор. Оператор *S* виконується повторно поки в значенні виразу *P* *(I v)* *v* не дорівнює *0* (True).
* Виклик *процедури визначеної користувачем* з іменем *p* і виразами-аргументами *e1, …, en - call p(e1, …, en).*
* Блок *{v1;..vk: st1;…;stn}*, що містить об`яву нових локальних змінних *v1;..vk* (може не бути) і список операторів *st1;…;stn*. Блок виконується в три кроки:
  + В стеку даних заводяться нові пари для локальних змінних *v1;..vk* з початковими значеннями 0.
  + Виконується список операторів *st1;…;stn,* в яких використовуються локальні, включаючи введені на початку блоку, і глобальні дані.
  + По закінченню блоку заведені пари для локальні дані *v1;..vk* вилучаються із стеку.

Процедури представляються наступними типами в Haskell.

***data*** Stmt = Assign Id Exp

| AssignA Id Exp Exp

| If Exp Stmt Stmt

| While Exp Stmt

| Call Id [Exp]

| Block [VarDef] [Stmt]

***deriving*** (Eq, Show)

***type*** ProcDef = (Id, ([VarDef], Stmt))

Оскільки процедура може викликати, як функції визначені користувачем (протягом обчислення Expr), так і процедури визначені користувачем, то інтерпретатор операторів повинен мати доступ до

* списку функцій - FunDefs,
* списку процедур - ProcDef.
* значення змінних, що є в області дії – локальних та глобальних, - StateP.

Приклади означень процедур:

* Процедура gAdd додає два числа і присвоює результат глобальній змінній gSum

*gSum;*

*proc gAdd(x, y) gSum = x + y*

* Процедура представляється наступним ProcDef:

gAdd :: ProcDef

gAdd =

("gAdd",

([Int "x", Int "y"], Assign "gSum" (OpApp Add (Var "x") (Var "y"))

)

)

* Далі процедурна версія функції sumA, що використовує цикл while для отримання суми масиву в глобальній змінній sA.

*sA;*

*proc sumA1 (a[], n) {*

*i, limit;*

*sA = 0; i = 0; limit = n + 1;*

*while (i < limit) {*

*sA = sA + a[i]; i = i + 1*

*}*

*}*

* Її можна представити

sumA1 :: ProcDef

sumA1 =

("sumA1",

([Arr "a", Int "n"],

Block [Int “i”, Int “limit”]

[Assign "sA" (Const 0),

Assign "i" (Const 0),

Assign "limit" (OpApp Add (Var "n") (Const 1)),

While (OpApp Less (Var "i") (Var "limit"))

(Block [] [Assign "sA" (OpApp Add (Var "sA")

(OpApp Index (Var "a") (Var "i"))),

Assign "i" (OpApp Add (Var "i") (Const 1))

])

]

)

)

В загальному випадку програма складається з трьох списків: глобальних змінних, функцій визначених користувачем і процедур визначених користувачем.

Серед процедур завжди є одна процедура без параметрів з іменем main.

Виконання програми: встановлюються значення всіх глобальних змінних в 0 і викликається процедура main.

***type*** Program = ([VarDef], [FuncDef], [ProcDef])

Приклад програми

*gSum;*

*proc gAdd(x, y) gSum = x + y*

*proc main() call gAdd(5,10)*

* Після її виконання стан буде містити одну глобальну змінну gSum зі значенням 15.

На етапі перевірки типів необхідно впевнитися, що проміжна форма (об»єкт типу Program), котра отримується в результаті синтаксичного аналізу, задовольняє ряду контекстних умов.

* Всі данні (скалярні і масиви), що використовуються в операторах і функціях визначені (знаходяться в області дії імен).
* Всі процедури і функції, що використовуються – визначені, кожний раз застосовуються до вірної кількості аргументів, кожний з яких має відповідний тип.
* Вираз, що складає тіло функції вираховує скалярне значення.
* Операнди всіх операцій у виразах і операціях присвоювання мають відповідні типи.
* В кожному списку об`яв – функцій, процедур, параметрів, глобальних і локальних змінних всі ідентифікатори різні.
* Серед списку процедур є процедура без параметрів з ідентифікатором main.

При перевірці типів (контекстних умов) використовується середовища (списки пар), в яких зберігається інформація про типи змінних (VarEnv), типи функцій (FunEnv) і типи процедур (ProcEnv). Тип змінної Type може бути або скалярний It або масив At.

Середовища типів формуються при обробці об`яв змінних та параметрів (VarEnv), функцій (FunEnv), процедур (ProcEnv) і використовуються при перевірці типів (контекстних умов) виразів та операторів.

***data*** Type = At | It ***deriving*** (Eq, Show)

***type*** FunEnv = [(Id,[Type])]

***type*** ProcEnv = [(Id,[Type])]

***type*** VarEnv = [(Id,Type)]

Допоміжний файл містить ряд функцій, котрі можна використати при перевірці типів:

* iswfOp o ts – перевіряє відповідність бінарної операції (Op) і типів її операндів. Якщо вони коректні, то повертається тип результату - Just t, в іншому випадку Nothing.
* iswfCond ts – виконує перевірку типів ts операндів умовного виразу, повертаючи тип результату Just t, або Nothing.
* iswfAssignA ts – предикат, що перевіряє типи операндів операції присвоювання елементу масиву.

В допоміжному файлі, котрий включає визначення типів, допоміжних функцій і даних для тестування, надати визначення наступних функцій.

1. Функція *updateValue a b abs*, котра в списку пар *abs* знаходить *першу* пару *(a1,b1)*, у якої *a1 == a*, і замінює цю пару на пару *(a,b)*. Якщо такої пари немає, то додає пару *(a,b)* в список. Наприклад:
   * updateValue 3 4 [(2,0), (3,7), (1,6), (3,2)] = [(2,0), (3,4), (1,6), (3,2)]
   * updateValue 3 4 [(2,0), (1,6), (0,2)] = [(2,0), (1,6), (0,2), (3,4)]
2. Функція *updateArray a i v*, що повертає масив такий самий як *a* за винятком його *i* елементу, значення якого потрібно зв’язати з *v*. Наприклад:
   * updateArray (lookUp “a” sampleState) (I 2) (I 1) = A [(2,1), (0,4), (1,2)]
   * updateArray (lookUp “a” sampleState) (I 5) (I 1) = A [(2,3), (0,4), (1,2),(5,1)]
3. Функція *applyOp op v1 v2*, котра застосовує оператор *op* до аргументів *v1* і  *v2.* Для *op =* *Index* («індексування» масиву) результат повинен бути нуль (представляється I 0), якщо немає зв’язування для індексу *v2* в масиві *v1*. Наприклад.
   * applyOp Add (I 6) (I (-2)) = I 4
   * applyOp Mul (I 3) (I 4) = I 12
   * applyOp Less (I 7) (I 0) = I 0
   * applyOp Equal (I 2) (I 2) = I 1
   * applyOp Index (A [(1,1),(0,3)]) (I 0) = I 3
   * applyOp Index (A [(1,1),(0,3)]) (I 2) = I 0
4. Взаємно рекурсивні функції *evExp* і *evArgst* для обчислення виразів. Функція *evExp e dfx st* обчислює значення вираз *e* для списку функцій визначених користувачем *dfx* і стану *st.*  *evArgs ex dfx st* застосовує функцію *evExp* до кожного елементу списку виразів *ex*, повертаючи список значень. Функція *evExp e dfx st* використовує правила:
   * Значення константи (ціле число) (*Const c*) є (I c).
   * Значення змінної отримується з стану *st* (стан – це стек, що містить список пар (Id,Value) ).
   * Значення умовного виразу - значення одного з підвиразів, в залежності від значення першого (умовного) виразу, значення якого (I v) – False (v == 0) або – True (v/= 0).
   * Значення бінарної операції – результат *applyOp* на обчислених аргументах.
   * Щоб обчислити застосування функції *f* до списку виразів *es* потрібно
     + Знайти означення *f* в списку *dfx*
     + Виділити з означення список імен аргументів *as* і вираз *ef* - тіло функції.
     + Використовуючи *evArgs*, обчислити кожний вираз в *es*, отримуючи список значень *vs*.
     + Зв’язати імена з *as* зі значеннями з *vs,* утворюючи новий стан *new*.
     + Обрахувати вираз *ef* в стані *new.*

Наприклад:

* evExp (Const 1) [] sampleState = I 1
* evExp (Var "y") [] sampleState = I 2
* evExp (OpApp Add (Var "x") (Const 2)) [] sampleState = I 7
* evExp (Cond (Const 1) (Var "x") (Const 9)) [] sampleState = I 5
* evExp (FunApp "fib" [Const 6]) [fib] sampleState = I 8

1. Рекурсивна функція *evStmt s dfx dpx st,* котра виконує один оператор *s*. Результат виконання – це перетворення вхідного стану у вихідний стан. Параметри функцій – список функцій визначених користувачем *dfx,* список процедур визначених користувачем *dpx* і вхідний стан *st.* Особливості виконання:
   * Виконання присвоювання змінній (Assign) і елементу масиву (AssignA) припускає, що відповідні змінна (масив) - «відомі» (є в області дії).
   * При реалізації виклику процедури потрібно спочатку розширити стан, додаючи пари (параметр-значення), а по закінченню обчислення процедури вилучити зі стану ці пари.

Наприклад, використовуючи приклади і тестові функції з допоміжного файлу:

* + evStmt sampleBlock [] [sumA1] [(“sA”, I 0)] = [(“s”, I 22)]
  + evStmt (Assign “y” (FunApp “sumA” [Var “a”, Const 1])) [sumA] [] sampleState = [(“x”, I 5), (“y”, I 6), (“a”, A [(2,3),(0,4),(1,2)])]
  + evProgram pr2 = [(“sA”, I 22)]
  + evProgram pr1 = [("gSum", I 15)]

1. Функція iswfExp e ve fe, котра перевіряє контекстні умови виразу s, використову-ючи типи доступних змінних (середовище змінних ve) і типи визначених функцій (середовище функцій fe), і повертає тип результату виразу Just t, або Nothing.
   * iswfExp (Var "a") varEnv [] = Just At
   * iswfExp (Var "b") varEnv [] = Nothing
   * iswfExp (FunApp "fib" [ Var “x”]) varEnv [(“fib”,[It])] = Just It
   * iswfExp (FunApp "fib" [Const 6, Var “x”]) varEnv [(“fib”,[It])] = Nothing
2. Предикат iswfStmt s ve fe pe, що перевіряє контекстні умови оператору s в середовищі доступних змінних ve, функцій fe і процедур pe.
   * iswfStmt sampleBlock varEnv [] procEnv = True
   * iswfStmt (Assign “y” (FunApp “sumA” [Var “a”, Var “x”])) varEnv funEnv [] = True
   * iswfStmt (Assign “y” (FunApp “sumA” [Var “y”, Var “x”])) varEnv funEnv [] = False
3. Предикати iswfFunDef df fe і iswfProcDef dp ve fe pe, шо перевіряють коректність визначення функції df і процедури dp, використовуючи середовища змінних ve, функцій fe і процедур pe.
   * iswfFunDef fib funEnv = True
   * iswfProcDef sumA1 varEnv funEnv procEnv = False
   * iswfProcDef sumA1 [(“sA”,At)] funEnv procEnv = False
   * iswfProcDef sumA1 [(“sA”,It)] [] [] = True
4. Предикат iswfProgram pr, котрий перевіряє контекстні умови програми, формуючи середовища змінних, функцій і процедур.
   * iswfProgram pr1 = True
   * iswfProgram pr2 = True

*updateValue* :: Eq a => a -> b -> [(a,b)] -> [(a,b)]

*updateArray* :: Value -> Value -> Value -> Value

*applyOp* :: Op -> Value -> Value -> Value

*evExp*  :: Exp -> [FunDef] -> StateP -> Value

*evArgs*  :: [Exp] -> [FunDef] ->StateP -> [Value]

*evStmt*  :: Stmt -> [FunDef] -> [ProcDef] -> StateP -> StateP

*iswfExp*  :: Exp -> VarEnv -> FunEnv -> Maybe Type

*iswfStmt* :: Stmt -> VarEnv -> FunEnv -> ProcEnv -> Bool

*iswfFunDef*  :: FunDef -> FunEnv -> Bool

*iswfProcDef* :: ProcDef -> VarEnv -> FunEnv -> PrucEnv -> Bool

*iswfProgram* :: Program -> Bool

Зауваження:

Назва файлу Family09.hs (Family – прізвище студента). Файл включає модуль Family09 і створюється на основі файлу-заготовки HWP09.hs