## Робота №9

Регулярні вирази і скінченні автомати

Мета роботи пояснити деякі алгоритми і структури даних, що зв’язані з реалізацією регулярних виразів. Регулярний вираз визначається:

* *Термінальний* символ, що визначає зразок, котрий співставляється лише з таким символом.
  + Вираз *a* - зразок, котрий співставляється лише з рядком *“a”.*
* *Послідовність* двох рядом записаних регулярних виразів.
  + Регулярний вираз *ab* - послідовність двох терміналів, *a* і *b*, - зразок, котрий співставляється лише з рядком *“ab”* .
  + Круглі дужки можна використовувати для групування підвиразів регулярного виразу, *a(bc),* *(ab)c* і *abc* всі задають однаковий регулярний вираз (послідовність – асоціативна).
* *Альтернатива,*  *(e1 | e2)*, де *e1* і *e2* – регулярні вирази.
  + Вираз *(ab | c) -* зразок котрий співставляється *або* з рядком *“ab”* *або* з рядком *“c”*.
  + Оператор являється асоціативним і комутативним, вирази *(a | bb | ac), (ac | (bb | a))* і *((a | ac) | bb)* задають однаковий зразок.
* *Повторення* («Зірочка Кліні») *нуль* або *більше* випадків виразу *e* - *e\**.
  + *ab\*c*  - зразок, котрий відповідає символу *‘a’*, за яким слідує нуль або більше символів *‘b’*, за яким слідує єдиний символ *‘c*’.
* *Повторення* («Плюс Кліні») *одного* або *більше* випадків виразу *e* - *e+*.
  + *ab+с*  - зразок, котрий співставляється з *“abbc”* , але не з *“ac”*.
  + Операцію можна спростити *e+ ≅ ee\**
* *Необов’язкове* входження виразу *e* - *e?*.
  + *(ab)?d+* - зразок, котрий співставляється з *“d”, “dd”, “abd”, “abdd”* і так далі.
  + Операцію можна спростити *e? ≅ (e|)*
* *Нуль* вираз, котрий відповідає лише порожньому рядку (“”).
  + *a?* - зразок, що співставляється лише з “a” і “”.
  + За домовленістю порожній виразу позначається символом ϵ, як в (a|ϵ).

|  |  |  |
| --- | --- | --- |
| Регулярний вираз | Рядки, що відповідають | Рядки, що НЕ відповідають |
| (x|y)(1|2) | “x2”, “y1” | “x”, “x3” |
| x’\* | “x”, “x’”, “x’’’” | “y”, “’”, “x’x” |
| (ab|c)\* | “”, “c”, “abababccab” | “d”, “ac”, “cccb” |
| (a?)a | “a”, “aa” | “b”, “aaa” |
| (ab)?d+ | “d”, “abd”, “abddd” | “ab”, “bd”, “ababd” |

*Таб.1.Регулярні вирази та приклади рядків, що їм відповідають / НЕ відповідають.*

Регулярні вирази можна реалізувати в Haskell наступним типом даних:

***data*** RE = Null | -- Нульвираз

Term Char | -- Термінальнийсимвол

Seq RE RE | -- Послідовність

Alt RE RE | -- Альтернатива

Rep RE | -- Повторення (\*)

Plus RE | -- Повторення (+)

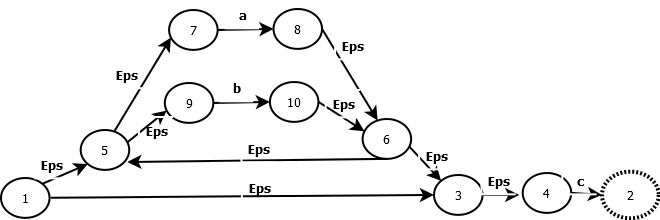
Opt RE | -- Необов’язковевходження (?)

deriving (Eq, Show)

|  |  |  |
| --- | --- | --- |
| Регулярий вираз | Представлення | Змінна |
| (x|y)(1|2) | Seq (Alt (Term ‘x’) (Term ‘y’))(Alt (Term ‘1’)(Term ‘2’)) | re1 |
| x’\* | Seq (Term ‘x’) (Rep (Term ‘’)) | re2 |
| (ab|c)\* | Rep (Alt (Seq (Term ‘a’)(Term ‘b’)) (Term ‘c’) | re3 |
| (a?)a | Seq (Opt (Term ‘a’)) (Term ‘a’) | re4 |
| (ab)?d+ | Seq (Opt (Seq (Term ‘a’)(Term ‘b’))) (Plus (Term ‘d’)) | re5 |

*Таб.2. Вирази і їх приклади представлення в допоміжному файлі*

Головна властивість регулярних виразів – їх можна задати за допомогою Недетермінованих (Скінченних) Автоматів (НСА).



*Мал.1. НСА для регулярного виразу (a|b)\*c (reFigure)*

НСА (ndaFigure) має один *початковий* стан (Мал.1 - 1) і один *заключний* стан (Мал.1 – 2 помічений пунктирною лінією). Послідовність міток (не Eps-переходів), що зустрічаються на шляху через НСА від початкового стану до заключного, визначає рядок, котрий розпізнається НСА («допускається» регулярним виразом, що відповідає НСА).

* Перехід, помічений символом Eps, відповідає регулярному виразу Null і його можна проходитися *без всяких умов* (не читаючи на вході символу).
* Якщо декілька Eps-переходів виходять з стану, то довільний з них може вести до успіху, тому загалом їх потрібно досліджувати *не детерміновано -* в гіршому випадку потрібно перебрати всі варіанти.
* Рядки, що розпізнаються НСА з Мал.1 - це точно ті, котрі співставляються («допускаються») регулярним виразом (a|b)\*c. Наприклад:
  + “ac” = шлях 1 -> 5 -> 7 -> 8 -> 6 -> 3 -> 4 -> 2), “bc” ( шлях 1 -> 5 -> 9 -> 10 -> 6 -> 3 -> 4 -> 2
  + “aac” = шлях 1 -> 5 -> 7 -> 8 -> 6 -> 5 -> 7 -> 8 -> 6 -> 3 -> 4 -> 2
* Вхідний рядок розпізнається НСА, якщо починаючи з початкового стану всі символи вхідного рядка будуть прочитані і автомат попаде в заключний стан. Наприклад:
  + Рядки “ac”, “bc” і “aac” розпізнаються.
  + Рядки “cc” і “ad” НЕ розпізнаються - відкидаються.

Автомат можна описати наступними типами Haskell:

***type*** State = Int

***data*** Label = C Char | Eps deriving (Eq, Ord, Show)

***type*** Transition = (State, State, Label)

***type*** Automation = (State, [State], [Transition])

Стан автомату (State) - унікальне ціле число.

Перехід (Transition) складається з вхідного стану, вихідного стану і мітки:

* Eps (безумовний перехід)
* C c, де c – символ, котрий автомат читає на вході при переході.

Автомат (Automaton) має 3 елементи

* Унікальний початковий стан.
* *Список* заключних станів.
* Список переходів.

НСА з Мал. 1 (ndaFigure в допоміжному файлі) можна представити :

(1, [2],

[(1,3,Eps), (1,5,Eps), (3,4,Eps), (4,2,C 'c'), (5,7,Eps),

(5,9,Eps), (6,3,Eps), (6,5,Eps), (7,8,C 'a'), (8,6,Eps),

(9,10,C 'b'), (10,6,Eps)])

Як перевірити, що НСА розпізнає рядок? Необхідно промоделювати робот у НСА, перебрати всі можливі шляхи з початкового стану, якщо *якийсь* з них, прочитавши рядок, переводить автомат в заключний стан, то рядок допускається.

Для цього можна визначити дві функції*: ss1= setStep naut bs mc-* визначає множину станів *ss1,* в які недетермінований автомат *naut* може перейти з множини станів *bs* за один крок прочитавши символ *mc = C ‘c’*, і *ss2 = closure naut ss1* – визначає множину станів *ss2*, в які недетермінований автомат *naut* може перейти з довільного стану з *ss1* нічого не читаючи на вході (виконуючи лише «порожні» переходи - Eps)

Розглянемо для НСА *naut* і слова *str = “c1c2…cn”* наступну послідовність множин станів:

*ss0  = closure naut {is} is – початковий стан НСА naut*

*ss1 = closure naut (setStep naut (C ‘c1’))*

*…………*

*ssn = closure naut (setStep naut (C ‘cn’))*

Список множин станів *[ss0, ss1, …, ssn]*, моделює ВСІ можливі варіанти поведінки НСА naut, коли він читає слово “*c1c2…cn”.*

НСА *naut* допускає слово “*c1c2…cn”* тоді і тільки тоді, коли множина *ssn* містить хоча б один заключний стан.

Як побудувати НСА за регулярним виразом *re*?

Перед побудовою НСА регулярний вираз можна спростити *simplify re*, замінивши в ньому на еквівалентні під-вирази, операції: *повторення* («Плюс Кліні») *одного* або *більше* та *необов’язкове* входження виразу.

|  |  |
| --- | --- |
| (x|y)(1|2) |  |
| x`\* |  |
| (ab|c)\* |  |
| (a?)a |  |
| (ab)?d+ |  |
| c?\* |  |

*Таб. 3: НСА для регулярних виразів в Таб.1.*

Функції верхнього рівня, котра наводиться в допоміжному файлі має вид:

makeNDA :: RE -> Automation

makeNDA re = (1, [2], sort transitions)

***where*** (transitions, k) = make (simplify re) 1 2 3

Головну роботу виконує допоміжна функція *make re beg fin nxt*, котра за регулярним виразом *re* будує НСА з початковим станом  *beg* (ціле число) і заключним  *fin* (ціле) використовуючи при необхідності нові ідентифікатори (цілі числа) станів починаючи з *nxt*. Повертає ця функція пару *(trx,nxt1)*: *trx* – список переходів, що реалізують регулярний вираз *re*, і *nxt1*- номер наступного вільного ідентифікатора стану.

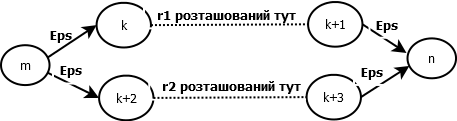
* Самий перший виклик має вид *make re 1 2 3*: 1- початковий, 2 – заключний і 3 – наступний вільний стани. Побудова ведеться по структурі регулярного виразу, викликаючи рекурсивно функцію *make* дляреалізації підвиразів згідно правил, що визначаються на Мал.2 .
* Результат в базовому випадку – це *один* перехід, позначений відповідним чином, ніяких додаткових станів не потрібно і наступний ідентифікатор стану залишається таким же.
* Якщо вираз Seq r1 r2 , то рекурсивно створюється НСА для r1 з початковим і заключним станами m і k, відповідно, потім створюється НСА для r2 з початковим і заключним станами k+1 і n, відповідно. Залишається додати один додатковий перехід, що помічений Eps, між станами k та k+1.



1. *Null та термінальний випадки: Eps і C c*



1. *Послідовність: Seq r1 r2*



1. *Альтернатива: Alt r1 r2*



1. *Повторення : Rep r1*

*Мал.2: Правила для побудови НСА*

Наприклад: *makeNDA re2*, де *re2 = x’\** створить наступний НСА

makeNDA Seq (Term 'x') (Rep (Term '\''))

result: (1, [2], [(1,3,C 'x'), (3,4,Eps), (4,2,Eps), (4,5,Eps), (5,6,C '\''), (6,2,Eps), (6,5,Eps)])

Скінченний автомат - Детермінований Автомат (ДА), якщо у нього з довільного стану завжди можливий лише один перехід, котрий однозначно визначається мітками переходу. Представлення як НСА так і ДА – однакові.

Ідея перетворення НСА в ДА в тім, що стани ДА, так називаємі «метастани», відповідають *множинам* станів початкового НСА.

* Спочатку, використовуючи допоміжну функцію *makeDA’ aut stx mstx mtrx*, будується список метастанів і переходів, що представляють ДА, разом з початковим метастаном.
* Потім необхідно відобразити метастани, множини цілих чисел, в звичайні стани, помічені цілими числами, стани і визначити *множину* заключних станів.

***type*** MetaState = [State]

***type*** MetaTransition = (MetaState, MetaState, Label)

makeDA :: Automation -> Automation

makeDA' :: Automation -> [State] -> [MetaState] -> [MetaTransition]

-> (MetaState, [MetaState], [MetaTransition])

Список нових метастанів *mstx* і нових переходів *mtrx* – це накопичувальні параметри допоміжної функції. Алгоритм побудови нових метастанів і нових переходів:

* Напочатку вхідною множиною станів *stx* початкового НСА є одноелементна множина, що містить лише початковий стан. Кожна множина станів представляється у виді списку. Наприклад, якщо початковий стан – 1, то [1].
* Для довільної вхідної множини станів початкового НСА можна побудувати *кордон* НСА - це список не-Eps переходів, котрих можна досягти ідучи лише Eps–переходами, якщо починати з кожного стану, що входить у вхідну множину.
  + Якщо кордон досягає заключного стану t, тоді спеціальний «фантомний» (dummy) перехід (t, t, Eps) додається в результуючий список (дивися далі).
  + З метою уникнення явного пошуку циклів, допускається ПЕРЕДУМОВА - кожен цикл в НСА включає хоча б один не-Eps перехід. Для автоматів, що будуються за наведеним раніше алгоритмом, ця передумова виконується.
  + Наприклад, кордон НСА nda3, що побудований за виразом (ab|c)\*, починаючи з початкової множини станів [1], є [(2,2,Eps), (5, 9,C ‘a’), (7,8, C ‘c’)].
  + Функцію *getFrontier st aut* будує кордон для НСА *aut,* починаючи зі стану *st*, як описано. Сигнатура типу цієї функції надається в допоміжному файлі.
  + Якщо вхідна множина має декілька станів, то всі кордони конкатенуються.
* Початкові стани всіх переходів з кордону (без повторів) визначає новий «метастан» ДА, що конструюється.
  + Наприклад: метастан для кордону вище буде [2,5,7].
  + Призначення фантомного переходу: цей метастан (пізніше) буде помічено, як термінальний, тому що 2 – заключний стан НСА.
  + Зауваження: скористайтесь функціями nub та sort з Data.List , щоб побудувати метастан.
* Якщо метастан *вже* *побудований* раніше, то початковий метастан і накопичений список метастанів та переходів не змінюється.
* Якщо метастан – *новий*, то будуються переходи з цього метастану.
  + Оскільки може бути декілька переходів з однаковою міткою, можливо, в різні стани, то всі переходи кордону *групуються* за міткою переходу. Якщо в кордон входить фантомний перехід, то він вилучається.
  + В прикладі вище будуть згруповані [(C ‘a’,[9]), (C ‘c’, [8])].
  + Але, кордон НСА для (a|)a зі стану 1 містить два ‘a’ переходи , тобто [(3,7,C ‘a’), (5,9,C ‘a’)], котрі будуть згруповані так: [(C ‘a’, [7,9]).
  + Для побудови функції *groupTransitions trx,*  котра групує переходи, як описано, можна використати функцію *labels*, котру необхідно побудувати в завданні далі. Сигнатура типів функції *groupTransitions* надається в допоміжному файлі.
  + Рекомендація: можна скористатися формувачами списків. Зауважимо, що функція labels вилучає фантомний перехід автоматично, якщо він є.
  + Кожна пара *(c, stx)-*  символ *c* і множина станів *stx,* що входить в множину згрупованих переходів, фактично визначає перехід під дією символу *с* з *нового* побудованого метастану в метастан, за кордоном отриманим з множини станів *stx.*
* Новий метастан додається до накопичувального списку станів і рекурсивно обробляється кожна цільова множину станів, додаючи новий перехід в накопичувальний список переходів після кожного рекурсивного виклику.
  + Наприклад, якщо згруповані переходи є [(C ‘a’,[5,6,7]), (C ‘c’, [8,9])], потрібно зробимо рекурсію з множинами станів [5,6,7] і [8,9], підтримуючи списки нових метастанів і переходів.

|  |  |
| --- | --- |
| Регулярний вираз | Детермінований Автомат |
| (x|y)(1|2) |  |
| x’\* |  |
| (ab|c)\* |  |
| (a?)a |  |
| (ab)?d+ |  |
| c?\* |  |

*Таб.4: ДА для прикладів в Таб.1.*

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

1. Функція *simplify re*, котра вилучає всі + або ? з виразу *re* використовуючи правила спрощення. Наприклад:

showRE (simplify re5) = “(ab|)dd\*”

1. Функції *startState aut, terminalStates aut, transitions aut,* котрі повертають початковий стан, список заключних станів та список переходів автомату *aut.* Наприклад:

terminalStates nda1 = [2]

1. Функція *isTerminal s aut*, що повертає значення True тоді і тільки тоді, коли стан *s* являється заключним станом автомату *aut*. Наприклад:

isTerminal 2 nda1 = True

isTerminal 3 da4 = True

isTerminal 5 nda2 = False

1. Функція *transitionsFrom s aut*, котра повертає список переходів, що виходять з стану *s* в автоматі *aut*. Наприклад

transitionsFrom 5 ndaFigure = [(5,7,Eps), (5,9,Eps)]

transitionsFrom 10 nda3 = [(10,6, C ‘b’)]

1. Функція *labels trx*, котра повертає всі мітки без дублікатів, що з’являються в списку переходів *trx*. Довільні Eps потрібно вилучити з результату. Наприклад.

labels [(1,2,Eps)] = []

labels (transitions nda3) = [C ‘a’, C ‘c’, C ‘b’]

Зауваження: можна використати функцію nub з Data.List, щоб вилучити дублікати.

1. Функція *stStep naut st mc*, котра обчислює множину станів, в які може перейти недетермінований автомат *naut* зі стану *st* за один крок, прочитавши символ ‘с’, якщо *mc == C ‘c’*, або по порожньому переходу, якщо *mc == Eps*.

Функція *setStep naut bs mc*, котра обчислює множину станів, в які може перейти недетермінований автомат *naut* з одного із стану *bs* за один крок, прочитавши символ ‘с’, якщо *mc == C ‘c’*, або по порожньому переходу, якщо *mc == Eps*.

Функція *closure naut ss*, котра обчислює множину станів, в які недетермінований автомат *naut* може перейти з довільного стану з *ss*, не читаючи на вході жодного символу (лише по порожнім переходам).

Наприклад.

stStep nda1 4 Nothing = [9, 11]

// setStep nda1 ([2,4,5]) (C ‘a’) = [6,2]

setStep nda4 ([2,4,5]) (C ‘a’) = [2,6]

// closure ndaTest ( [2]) = [1..5]

closure nda4 ( [1,7,8] ) = [5,7,8,3]

1. Функція *accepts* *aut st*, котра повертає True, в тому і тільки в тому випадку коли автомат *aut* допускає рядок *st.*  Для реалізації можна використати функції *setStep* i *closure.* Наприклад:

accepts ndaFigure “ac” = True

accepts ndaFigure “aac” = True

accepts ndaFigure “ad” = False

1. Функція *make re beg fin nxt*, котра за регулярним виразом *re* будує НСА з початковим станом  *beg* і заключним  *fin* використовуючи при необхідності нові стани починаючи з *nxt*. Результат функції - пара *(trx,nxt1)*: *trx* – список переходів, що реалізують регулярний вираз *re*, і *nxt1*- номер наступного стану. Наприклад, точністю до нумерації станів:

makeNDA reFigure = ndaFigure

makeNDA re1 = nda1

makeNDA re4 = nda4

1. Функція *makeDA nda,* котра перетворює НСА *nda* в еквівалентний детермінований автомат. Для реалізації можна визначити допоміжні функції *makeDA’, getFrontier* та *groupTransitions,* що визначені в допоміжному файлі.Наприклад (з точністю до нумерації станів детермінованого автомату):

makeDA ndaFigure = daFigure

makeDA nda1 = da1

makeDA nda3 = da3

simplify :: RE -> RE

startState :: Automation -> State

terminalStates :: Automation -> [State]

transitions :: Automation -> [Transition]

isTerminal :: State -> Automation -> Bool

transitionsFrom :: State -> Automation -> [Transition]

labels :: [Transition] -> [Label]

stStep :: Automation -> State -> Label -> [State]

setStep :: Automation -> [State] -> Label -> [State]

closure :: Automation -> [State] -> [State]

accepts :: Automation -> String -> Bool

make :: RE -> Int -> Int -> Int -> ([Transition], Int)

getFrontier :: State -> Automation -> [Transition]

groupTransitions :: [Transition] -> [(Label, [State])]

makeDA' :: Automation -> [State] -> [MetaState] -> [MetaTransition]

-> (MetaState, [MetaState], [MetaTransition])

makeDA :: Automation -> Automation

Зауваження:

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