## Робота №10

*Аналізатори*

Алфавіт – скінченна множина символів: букви, цифри, спеціальні символи. Як правило підмножина елементів типу Char.

Мова L в алфавіті V – це довільна підмножина слів в алфавіті V, як правило, нескінченна. Скінченна множина правил, яка описує як побудувати всі слова мови – синтаксис мови.

Задача синтаксичного аналізу – для заданих мови L і слова w визначити чи містить мова L слово w. Програми, що розв’язують цю задачу – аналізатори.

Перш ніж розв’язувати задачу аналізу, часто потрібно спочатку описати саму мову – задати синтаксис мови.

Існує багато формальних методів, котрі задають синтаксис мов: контекстно-вільні граматики, форми Наура-Бекуса, синтаксичні діаграми і т.і. Часто сам формальний метод задання мови визначає структуру і склад аналізатора.

Для опису мов буде використовуватися деяке розширення форми Наура-Бекуса.

В описі мови використовуються термінали – це символи алфавіту – і нетермінали, котрі позначають певні синтаксичні конструкції мови.

Термінали позначають самі себе і записуються в лапках – *‘a’, ‘n’, ‘=’, ‘\n’*. *‘eos’* – особливий термінал, котрий позначає кінець слова (end Of String).

Нетермінали – це слова в кутових дужках - *<sign>*, *<integer>*, *<digit>*.

Опис мови – це скінченна множина правил виду:

*нетермінал* ::= *правило*

Правило - слово з терміналів, нетерміналів і метасимволів, котре описує як побудувати всі слова синтаксичної конструкції *нетермінал.*

Наступні правила описують всі цілі десяткові числа <integer>, можливо зі знаком.

*<sign> ::= ‘+’ | ‘-‘*

*<digit> ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’*

*<integer> ::= [<sign] <digit> {<digit>}’eos’*

В цих правилах вживаються метасимволи:

* *a1 | a2* - вибір одного з варіантів a1 або a2
* *[a1]* – конструкцію a1 в середині [] потрібно використати 0 або 1 раз
* *{a1}* – конструкцію a1 в середині {} потрібно використати 0 або багато разів

Використання метасимволів {} надає можливостей одним правилом описати нескінченну

множину слів.

Ідея синтаксичного аналізу зверху вниз досить проста.

Створюється набір рекурсивних аналізаторів (функцій, процедур) кожний з яких розпізнає одну синтаксичну конструкцію (*нетермінал*). Структура аналізатора відображає правило для цього нетермінала:

* *термінал –* якщо наступний необроблений символ рядка співпадає з терміналом, то аналіз *продовжується* на наступному символі, якщо НЕ співпадає, то *помилка*.
* *нетермінал –* викликається аналізатор, що розпізнає цю синтаксичну конструкцію.

Проблема виникає про розпізнаванні правил виду *a1 | a2* - необхідно почати розпізнавати або конструкцію *a1* або конструкцію *a2.* Варіанти:

1. Завжди перебирати обидва варіанти. Можливо отримання двох варіантів успішного аналізу.
2. На основі якогось критерію, наприклад в залежності від наступного символу котрий потрібно розпізнати, вибрати один з варіантів і виконувати лише його. В цьому випадку можлива ситуація помилку, хоча насправді інший варіант приводить до успіху.
3. Вибрати один з варіантів, але у випадку його невдачі повернутися назад і перевірити інший*.*

Бібліотека Text.Parsec реалізує 2 варіант - будує прогнозуючий аналізатор *(predictive parser*), котрий приймає рішення на базі одного наступного нерозпізнаного символу (*the look ahead is 1*). Якщо одного символу НЕ досить для успішного вибору, то бібліотека дозволяє реалізувати 3 варіант (комбінатор *try*).

Використовуючи цілі числа, круглі дужки ‘(‘, ‘)’ та позначення цілочислених операцій ‘+’ (додавання), ‘-‘ (віднімання ) , ‘\*’ (множення), ‘/’ (ділення націло) ‘%’ (цілий залишок від ділення) можна будувати арифметичні вирази, котрі можна обрахувати, отримуючи результат ціле число. Наступні правила описують рядки, що містять такі вирази:

*<opMul> ::= '\*' | '/' | '%'*

*<opAdd> ::= '+' | '-'*

*<digit> ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’*

*<integer> ::= <digit> {<digit>}*

*<factor> ::= '(' <expr> ')' | <integer>*

*<term> ::= <factor> { <opMul> <factor> }*

*<expr> ::= [<opAdd>] <term> { <opAdd> <term> }*

*<evExpr> ::= <expr> 'eos'*

В мовах програмування вирази включають ідентифікатори змінних і для наочності символи проміжку ‘ ‘, ‘\t’, ‘\r’, ‘\n’, а в результаті успішного синтаксичного аналізу будується AST (Абстрактне Синтаксичне Дерево).

Наступні типи Haskel описують одне з таких представлень AST:

***data***  Expr = Add Expr Expr | Sub Expr Expr

| Mul Expr Expr | Mod Expr Expr | Div Expr Expr

| Var String | Lit Int

deriving (Show, Eq)

Наприклад: рядок *“ a1 / (17 + b ) “* містить вираз, котрий можна представити так:

*Div (Var “a1”) ( Add (Int 17) (Var “b”))*

Наступні правила описують рядки, що містять такі вирази зі змінними та проміжками:

*<digit> ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’*

*<letter> ::= ‘a’| … \ ‘w’ | ‘A’ | … | ‘W’*

*<spaces> ::= { ‘ ‘ | ‘\t’ | ‘\r’ | ‘\n’}*

*<integer> ::= <digit> {<digit>}*

*<iden> ::= <letter> {<digit> | <letter> }*

*<opMul> ::= '\*' <spaces> | '/' <spaces> | '%'<spaces>*

*<opAdd> ::= '+' <spaces> | '-'<spaces>*

*<factorT> ::= '(' <spaces> <expr> ')' <spaces>*

*| <integer> <spaces> | <iden> <spaces>*

*<termT> ::= <factorT> { <opMul> <factorT> }*

*<exprT> ::= [<opAdd>] <termT> { <opAdd> <termT> }*

*<fullExpr> ::= <spaces><exprT> 'eos'*

Для задання простих мов в програмуванні використовують регулярні вирази. Кожний регулярний вираз – це рядок, що містить символи алфавіту мови (термінальні символи) та мета-символи ‘(‘, ‘)’, ‘|’, ‘\*’, ‘+’, ‘?’ і описує деяку мову в алфавіті термінальних символів, наприклад букви та цифри.

Регулярний вираз:

* *Термінальний* символ, буква або цифра, 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”*.
* *Необов’язкове* входження виразу *e* - *e?*.
  + *(ab)?d+* - зразок, котрий співставляється з *“d”, “dd”, “abd”, “abdd”* і так далі.

|  |  |  |
| --- | --- | --- |
| Регулярний вираз | Рядки, що відповідають | Рядки, що НЕ відповідають |
| (x|y)(1|2) | “x2”, “y1” | “x”, “x3” |
| x1\* | “x”, “x1”, “x111” | “y”, “1”, “x1x” |
| (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 = 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*

Синтаксис регулярних виразів описують наступні правила:

*< reg> :: = <rexpr> ‘eos’*

*< rexpr> :: = <rterm> { '|' <rterm>}*

*< rterm> :: = <rfact> {<rfact>}*

*< rfact> :: = <prime> {'\*' | '+' | '?'}*

*< prime> ::= <rsymb> | '(' <rexpr> ')'*

*< rsymb> :: =<довільний символ крім ‘(‘, ‘)’, ‘|’, ‘\*’, ‘+’, ‘?’>*

Наступні правила описують синтаксис XML- документу *<fullXML>*:

*< wSp> :: = {символи, що задовольняють isSpace}*

*< textXML> ::= {довільні символи крім ‘<’ і ‘>’ }*

*<valueXML> :: = {довільні символи крім ‘”’ }*

*<nameXML> ::= <letter> {<letter> | <digit> | ‘.’ | ‘-‘}*

*<element> ::=‘<’ <nameXML> {<attribute>} ’>’ { <xmlXML> } ‘<’’/' <nameXML> ‘>’*

*<attribute> :: = <wSp> <nameXML> <wSp> ‘=’ <wSp> ‘”’ <valueXML> ‘”’*

*< xmlXML> ::= <element> | <textXML>*

*<fullXML> ::= <wSp> <element> <wSp> <eos>*

На основі допоміжного файлу, котрий включає визначення типів, допоміжних функцій і даних для тестування, створити файл, в якому надати визначення наступних функцій-аналізаторів. В разі потреби визначити необхідні додаткові функції.

1. Функція *evExpr str,* котра розпізнає і вираховує арифметичний вираз , що складається з цілих чисел, круглих дужок та операцій ‘+’, ‘-‘, ‘\*’, ‘/’, ‘%’, повертаючи результат – значення типу Maybe Integer. Наприклад:

evExpr “27-(6\*3+5%2)” = Just 8

evExpr “3-67+(7= 6)” = Nothing

1. Функція-аналізатор *fullExpr,* котра розпізнає арифметичний вираз, що містить змінні та проміжки, - значення типу Expr. Наприклад, використовуючи функцію *astrExpr* з допоміжного файлу:

astExpr “a1 / (17 + b )” =

Just (Div (Var “a1”) (Add (Lit 17) (Var “b”)))

astExpr “a1 (17 + b )” = Nothing

1. Функція-аналізатор *reg,* котрарозпізнає регулярний вираз, - значення типу RE. Наприклад, використовуючи функцію *regExp* з допоміжного файлу:

regExp “(a?)a” = Just (Seq (Opt (Term ‘a’)) (Term ‘a’))

regExp “ab(+)” = Nothing

1. Функція-аналізатор *fullXML,* котра розпізнає XML-документ , - значення типу XML. Наприклад, використовуючи функцію *anXML* і данні з допоміжного файлу:

anXML casablanca == Just casablancaParsed == True

*evExpr*  :: String -> Maybe Int

*fullExpr*  :: Parser Expr

*reg* :: Parser RE

*fullXML* :: Parser XML

Зауваження:

Назва файлу Family10.hs (Family – прізвище студента). Файл включає модуль Family10 і створюється на основі файла-заготовки HWP10.hs