Expressio

Ian Treyball (Language Guru)
Lalka Rieger (Tester)
Chengtian Xu (System Architect)
David Han (Team Manager)
{ict2102, ler2161, cx2168, dth2126 } @ columbia.edu

February 2, 2018

Figure 1: What we will need for this project.

1 Introduction

A regular expression is a sequence of characters that defines a search pattern. In theoretical computer science, it can express a regular language. Practically, there are countless times programmers and non-programmers use regular expressions in their daily lives, whether it be validating email or password formats on server-side web applications or string searching algorithms for "find" or "find and replace" operations on strings.

Within automata theory, the basic mathematical model of computation is known as a finite state machine. A finite state machine is an abstract machine that takes in an input string, and in response to the input characters moves through a finite set of states. An FSM can be in exactly one state at a time and moves through them according to a function of the current state and next input character. An FSM is said to "accept" an input if it finishes processing the input in an "accept state." An FSM is functionally equivalent to a regular expression. Meaning that any language that a regular language can "match," some FSM can also "accept."

We propose a language that allows for construction and manipulation of regular expressions which allows users to do normalization, pattern matching, construct finite state machine, etc.. The name of our language is Expressio, conceived after an unintentional misspelling of "expression."

2 Language Description and Features

Expressio is a language designed to create, manipulate and match literals to regular expressions. Users can build regular expressions from literals and other regular expressions with the usual operators: concatenation, union, and Kleene star. These operators would improve the composability and maintainability for the construction of much larger regular expressions by potentially organizing and dividing them to smaller blocks of regular expressions. In addition, Expressio offers the ability to decompose regular expressions into components using functional pattern matching.

Expressio also supports Deterministic Finite Automata (DFAs) and Nondeterministic Finite Automata(NFAs), which are equivalent in computational power to regular expressions. Since regular expressions can be converted to NFAs (vice-versa), Expressio empowers users to explore this connection, and implement related applications and algorithms.

Expressio is type-safe and mostly imperative, but handles regular expressions in a functional manner.

Algorithms that could be expressed with Expressio include Thompson's Construction, Glushkov's Construction Algorithm, Kleene's Algorithm, and Brzozowski derivatives

2.1 Deconstruction through pattern matching

Deconstruction via pattern matching allows the users of Expressio to recursively decompose the regular expressions also recursively built. This is important because it allows the users of Expressio to naturally express functions on regular expressions.

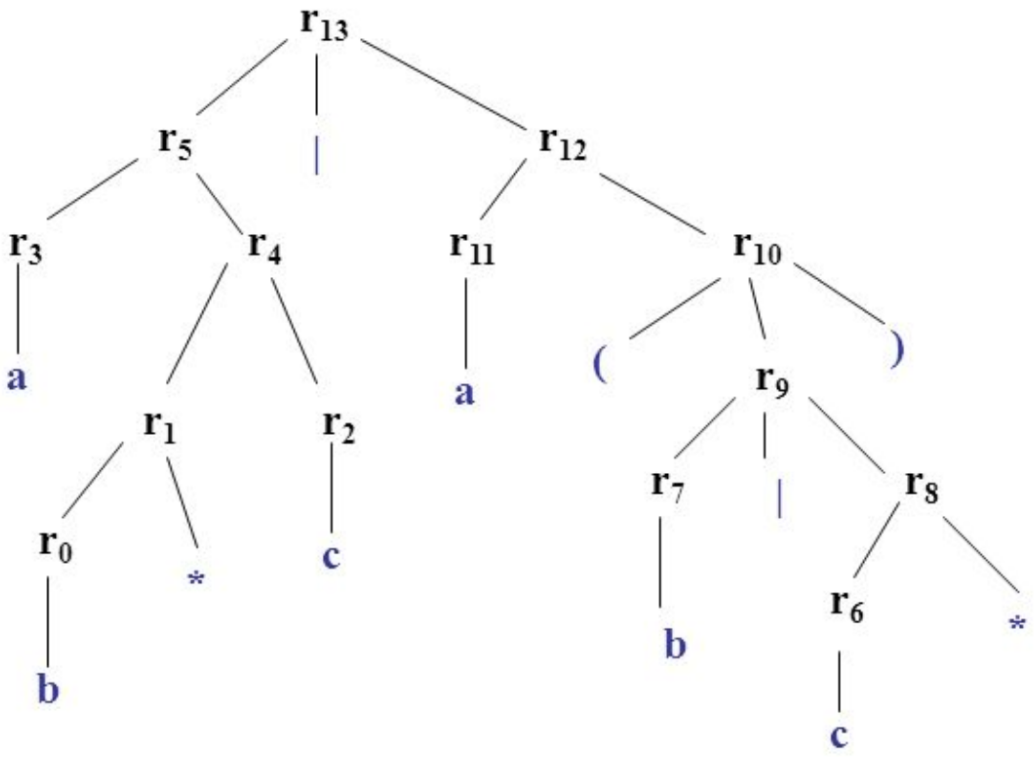

Figure 2: $(ab*c)|(a(b|c*))$

2.2 Matching

Matching informs users if an input string matches a specific regular expression. An input string followed by the word "matches" and then a regular expression returns a boolean type.

```
regexp char sample = (($"ab")** ($'c')) ^ (($'a')( ($'b')| ($'c')**))
print "abc" matches sample
~~ should print true
```

2.3 Polymorphism

Formalisms for defining regular languages (Regular Expressions, DFAs, NFAs) require the alphabet (typically denoted Σ) of each formalism to match.

e.g. let m_1 be the DFA (Q_1, Σ_1, δ_1, q_{10}, F_1) and m_2 be the DFA (Q_2, Σ_2, q_{20}, F_2), then the union of m_1 and m_2 would require $\Sigma_1 = \Sigma_2$.

This can be enforced by the compiler through the use of polymorphism and type safety by encoding the alphabet as part of the type of the formalism. This way a compiler error will be generated if any user is attempting to take the union of two DFAs, for example, which do not have the same alphabet.

3 Syntax

3.1 Types

Types	Name
unit	()
char	Character
int	Integer
boolean	Boolean
array	Array
string	String

3.2 Operators

3.2.1 Conventional Operators

Operator	Description
=	Assignment Operator
+, -, *, /	Arithmetic Operators
[ ]	Looping Statement
<, <=, ==, >=, >	Relational and equality comparison
()	Parentheses
{}	Grouping statements
&&, ||, ~	Logical Operators
#	Specify polymorphic type variable name
:	Specify input type definition
->	Specify function return type

3.2.2 Regexp Operators

Regex Operator	Description
$	Evaluate String to compound of Literals
**	Kleene Closure
^	Concatenation
|	Union
(.)	Nullary Operator for ϵ
{.}	Nullary Operator for $\emptyset$

3.3 Keywords

Keyword	Description
regexp	Regular Expression
dfa	Deterministic Finite Automata
nfa	Nondeterministic Finite Automata
if elif else	Conditional Statement
for in	Looping Statement
string	String
break	Exit out of loop
continue	Start next iteration
matches	check if string matches regexp

3.4 Comments

- ~~ *Single Line Comment*
- <~ *Block Comment* ~>

3.5 Control Flow

3.5.1 If Statements

```
if (condition){
        <~ CODE BLOCK ~>
} elif (condition2) {
        <~ CODE BLOCK ~>
} else {
        <~ CODE BLOCK ~>
}
```

3.5.2 Loops

```
~~ a "while" loop
for ; <condition> ; {
        <~ CODE BLOCK ~>
}

~~ a "for" loop
for <start> ; <condition> ; <increment> {
        <~ CODE BLOCK ~>
}

~~ a "for each" loop
for s in <Collection> {
        <~ CODE BLOCK ~>
}

~~ an infinite loop
for {
        <~ CODE BLOCK ~>
}
```

3.6 Declarations

3.6.1 Functions

```
<name> : <input_type> -> <return_type>
<name> <cond1> = <return1>
<name> <cond2> = { <~ CODE BLOCK ~>
```

```
        <return2>
        }
```

3.6.2 Finite State Machines

Deterministic Finite Automata (DFA):

```
dfa <type of states (Q)> <type of alphabet (Sigma)> {
    ~~ start state
    q0 = <value of type Q>
    ~~ Set of final states
    F = <Set of type Q>
    ~~ transition function
    delta = <function of type (Q , Sigma) -> Q>
}
```

Nondeterministic Finite Automata (NFA):

```
nfa <type of states (Q)> <type of alphabet (Sigma)> {
    ~~ start state
    q0 = <value of type Q>
    ~~ Set of final states
    F = <Set of type Q>
    ~~ transition function
    delta = <function of type (Q , Sigma) -> Q>
}
```

3.6.3 Regular Expressions

```
regexp <type> <name> = <value>
```

4 Library Features

We plan to provide support for Sets and tuples in an external library, which is used in the code samples below.

5 Code Sample

5.1 Accessing Subexpressions Example

```
main {
    string s0 = "01"
    string s1 = "10"
    regexp alternate = ($s0)**|($s1)**
    ~~ deconstruct function defined using pattern matching
    (op, exp1, exp2) = deconstruct alternate
    ~~ examine exp1 and exp2
}

~~ function size uses pattern matching to find the number of literals in regexp
size : RegExp #s -> int
size (.)       = 0
size {}        = 0
size ($ _)     = length 1
size (r1 ^ r2) = size r1 + size r2
size (r1 | r2) = size r1 + size r2
size (r **)    = size r
```

5.2 Matching Example

```
main {
    regexp char digit = $'0123456789'
    ~~ will match any string of digits ending in 0 or 5
    regexp char divBy5 = (digit)** ^ ($'0' | $'5')
    regexp number = "15"
    print number
    if (number matches divBy5) {
            print (" is divisible by 5")
    }
    else {
            print (" is not divisible by 5")
    }
}
```

5.3 DFA Example

```
~~ A number, n, either ends in 5 or 0 (when n % 5 = 0),
~~ or it doesn't (n % 5 != 0).
delta1 : (bool, char) -> bool
delta1 (_, '0') = true
delta1 (_, '5') = true
delta1 (_, _) = false

~~ states bool values, alphabet char values
dfa bool char {
    q0    = false
    F     = {true}
    delta = delta1
}

delta2 : (bool, char) -> Set bool
delta2 (_, '0') = {true}
delta2 (_, '5') = {true}
delta2 (_, _) = {false}

nfa bool char {
    q0 = false
    F  = {true}
    delta = delta2
}
```