Expressio LRM

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

February 26, 2018

Figure 1: What we will need for this project.

Contents

1 Introduction

Regular expressions, sequences of characters that define a pattern, are used widely in computer science and formal language theory. Search engines, web forms, word processors and text editors all make use of regular expressions. Expressio is a language designed to create, manipulate and match strings to regular expressions. For complex patterns, Expressio improves the composability and maintainability of the regular expression by organizing and dividing it to smaller blocks. Users build regular expressions with literals, regular expression variables and regular expression operators: concatenation, union, and Kleene star. Expressio offers the ability to decompose regular expressions into components using functional pattern matching.

In the realm of formal language theory, regular expressions are equally as expressive as the basic mathematical model of computation, 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. Expressio supports two types of finite state machines: Deterministic Finite Automata (DFA), and Non-Deterministic Finite Automata (NFA).

Expressio enables users to explore the relationship between NFAs, DFAs and regular expressions though basic construction and deconstruction operations on regular expressions.

1.1 Regular Expressions Background

A Regular Expression (RE) is a formalism for compactly representing a regular language. Originally invented by Stephen Cole Kleene, REs have become a widely used tool throughout computer science. Many modern implementations extend the original definitions typically by adding intersection and complement (as Expressio does) constructors, while managing to keep the formalism in direct correspondence with the class of regular languages. An RE is inductively defined over a finite set of symbols known as the alphabet (the alphabet which is typically denoted by Σ) by the following grammar in Backus Naur Form (BNF):

$$\alpha, \beta ::= \emptyset$$

$$|\epsilon$$
$$|\sigma$$
$$|\alpha|\beta$$
$$|\alpha \bullet \beta$$
$$|\alpha\&\beta$$
$$|\alpha^*$$
$$|\neg\alpha$$

where $\sigma \in \Sigma$.

A regular language can be thought of a (not necessarily finite) set of strings. Let $L(\alpha)$ represent the language of some RE α, then

$$L(\emptyset) = \emptyset$$
$$L(\epsilon) = \epsilon$$
$$L(\sigma) = \sigma, \forall\sigma \in \Sigma$$
$$L(\alpha \bullet \beta) = L(\alpha) \bullet L(\beta)$$
$$L(\alpha|\beta) = L(\alpha) \cup L(\beta)$$
$$L(\alpha\&\beta) = L(\alpha) \cap L(\beta)$$
$$L(\alpha^*) = L(\alpha)^*$$
$$L(\neg\alpha) = \Sigma^* \setminus L(\alpha)$$

A string of symbols (each of which belong to Σ), s, is said to *match* a regular expression, α, if and only if $s \in L(\alpha)$. The set Σ^* represents all possible strings of that alphabet. It is important to note that some modern implementations of REs add the back-reference feature which disqualifies the implementation from actually defining regular languages. For this reason, Expressio does not support the back reference feature.

1.2 DFA/NFA Background

A DFA and NFA are finite-state machines (FSM) that either *accept* or *reject* a string of symbols. The set of strings that the FSM M *accepts* is the language of the FSM denoted $L(M)$.

Formally, a **DFA** M is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, consisting of:

- a finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- transition function ($\delta : Q \times \Sigma \to Q$)
- initial or start state ($q_0 \in Q$)
- a set of accept states ($F \subseteq Q$)

Let $w = a_1a_2...a_n$ be a string over the alphabet Σ. The automaton M *accepts* the string w if a sequence of states, $r_1r_2...r_n$, exists in Q with the following conditions:

1. $r_0 = q_0$
2. $r_0 = \delta(r_i, a_{i+1})$, for $i = 0, ..., n-1$
3. $r_n \in F$.

In words, the first condition states that the machine starts in the start state q0. The second condition says that given each character of string w, the machine will transition from state to state according to the transition function δ. The last condition says that the machine *accepts* w if the last input of w takes the machine to one of its accepting states. Otherwise, it is said that the automaton *rejects* the string.

An **NFA** is the same 5-tuple except the transition function is $\delta : Q \times \Sigma \to P(Q)$. Where, $P(Q)$ denotes the power set of Q. This means an NFA can transition to any number of states on a single input. This automaton *accepts* w if one sequence of states covered by the transition function ends in an accepting state. If it is impossible to get from q_0 to a state in F by following w, it is said that the automaton *rejects* the string.

The languages of DFA and NFA are regular languages that can be expressed with regular expressions. Thompson's construction and Glushkov's construction are known algorithms that convert regular expressions to FSM.

2 Notation in this Document

Prose for this document will be in this font. Code examples will be formatted as following:

```
int x;
print("I am code!");
```

Grammar rules look like the following:

$$NON\text{-}TERMINALS \rightarrow SYMBOLS$$

3 Language Basics

3.1 Types, Variable Names, and Assignment

All variables in Expressio are statically typed. Thus, users must declare the type of a variable before its name is used anywhere in the program. Variable names are restricted to begin with a letter and contain only letters, numerals and underscore. There is no type promotion. Operations are only permitted on operands of the same type.

The assignment operator = applies to all types. It takes a variable name on the left and an expression on the right. The expression must evaluate to the same type as the variable name. The variable then assumes the value of the expression. For example:

Listing 1: Declare/Assignment example

```
int a;
a = 3;
```

Listing 1 is declaration and then assignment for variable a of type int. a now has the value of 3. Using operators on variables that have not been assigned a value will produce undefined behavior.

See section 3.4 for more information on expressions.

Expressio has three primitive types:

- `int` (4 bytes)
- `bool` (1 byte)
- `char` (1 byte)

As indicated, an int will be 4 bytes long and represent a non-decimal number. bool is one byte long and represents true/false values. char is one byte and represents an ASCII character. Expressio also has complex types:

- `string`

- `regexp`
- `dfa`
- `nfa`

3.2 Literals

3.2.1 Integer Literals

Integer literals are numbers represented by a sequence of 0 - 9 characters. Adding a prefix '-' sign makes an integer negative.

3.2.2 Boolean Literals

Boolean literals are either true or false. `true` and `false` are reserved as keywords.

3.2.3 String Literals

String literals are any character sequences surrounded by double quotation marks "".

3.2.4 Delimiters

White spaces are used to delimit tokens from code.

3.2.5 Parentheses and Braces

Parentheses and braces are used to force execution orders as well as limit the scope of variables.

3.2.6 Commas and Semicolons

Commas separate arguments in function declarations. Semicolons denote the termination of a sequence of expressions.

3.2.7 Comments

Single line comments start with ~~and end with a newline character.
Block comments start with <~and end with ~>

3.3 Operators

Tables include operators, description and associativity. Associativity fixes the order of evaluation for operators with the same precedence.

3.3.1 Arithmetic

Operator	Description	Associativity
+	integer addition; string concatenation	left
- (binary)	integer subtraction	left
- (unary prefix)	integer negation	nonassoc
*	integer multiplication	left
/	integer division	left

Operators - * / apply to integers only. + doubles as integer ADDITION and string CONCATENATION. On one operand - is a prefix operator and performs integer NEGATION. Examples:

```
−3;
```

has value negative 3.

```
−(−3);
```

has value positive 3.

```
4−;
```

is incorrect syntax.

3.3.2 Logical

Operator	Description	Associativity
&&	logical AND	left
||	logical OR	left
~(unary prefix)	logical negation	nonassoc

&& || ~are logical operators. They apply to bool operands only. Examples:

```
true && false;
```

has value false.

```
true || false;
```

has value true.

```
~true;
```

has value false.

3.3.3 Comparison

Comparison operations apply to `int`, `bool` and `char` types only. For chars, a `char` is "less than" another if it comes earlier in the ASCII table. For example `'a' < 'z'` is `true`.

Operator	Description	Associativity
==	structural equality	left
!=	structural inequality	left
>	greater than	left
<	less than	left
>=	greater than or equal	left
<=	less than or equal	left

3.3.4 Regular Expressions

These operations take `regexp` operands unless otherwise specified.

Operator	Description	Associativity
lit (unary prefix)	takes a single `char` and returns the regular expression equivalent	nonassoc
|	regexp union	left
&	regexp intersection	left
^	regexp concatenation	left
**	Kleene closure	nonassoc
‘	regexp complement	nonassoc
matches	takes a `string` on the left and a regexp on the right; returns true if the string is in the language of the regexp	right

3.3.5 DFA/NFA

These operations take two operands of the same type - either both `dfa` or both `nfa` - unless otherwise specified.

Operator	Description	Associativity
|	dfa/nfa union	left
^	dfa/nfa concatenation of languages	left
accepts	takes a string on the left and a dfa/nfa on the right; returns true if the string is in the language of the dfa/nfa	left
simulates	takes a string on the left and a dfa/nfa on the right; returns the state (integer) that the simulation terminates on	left

3.3.6 Precedence

Precedence in the table below goes from highest to lowest.

Operator Precedence
^
|
matches
accepts , simulates
! , - (NEGATION)
* , /
+ , -
< , > , <= , >=
== , !=
&&
||
=

3.4 Expressions

Expressions are sequences of, literals, variables, and operators that produce a value. The value of the expression is determined by applying operators in order of precedence and associativity. Valid expressions contain literals, variables, and operators that apply to the correct type. Some examples of expressions:

Listing 2: Expressions Example

```
a = 4; ~~where a was already declared an integer variable

a; ~~expression has value 4

1; ~~where 1 is an integer; has value 1

a+1; ~~expression value 5

b = a+1; ~~ where b is an integer variable; expression value 5

c = b = a+1; ~~ c also integer variable; expression value 5

<~ greetings is a regexp; expression evaluates to a bool ~>
true && "hello" matches greetings;
```

3.5 Case Matching

Case matching examines the structure or value of a data type and based on said case analysis returns a value.

To begin case matching, the input to be matched on must be followed by the `case` keyword and a colon symbol. Following the colon should be the different cases of input patterns. There are three parts of a case definition. First part of a case is the actual pattern the input should be matched upon. Second part is the arrow sign, pointing to what to return if the particular case is matched. Third part is the return value itself.

All patterns on the left side of the arrows must be of the same type. All return values on the right side of the arrow sign must also be of the same type.

Listing 3: Case Matching Example

```
regexp rTest;
regexp result;
rTest = lit 'a' | lit 'b' | lit 'c';
result =
  rTest case:
    r1 | r2 >> r1
```

```
r1 ^ r2 >> r1
r1**    >> r1
r1      >> r1
;
```

3.6 Keywords

Certain words in Expressio are reserved. They cannot be used as variable names:

Keywords	Description
true	**true** is a keyword that represents the corresponding boolean value.
false	**false** is a keyword that represents the corresponding boolean value.
dfa	The **dfa** keyword is the type name used to declare a new DFA in Expressio. Refer to Section 7 for an example.
nfa	The **nfa** keyword is the type name used to declare a new NFA in Expressio. Refer to section 7 for an example.
states	The user must indicate the number of states in a DFA/NFA when assigning a value to the DFA/NFA variable by using the **states** keyword. The **states** keyword takes in a finite natural number. Refer to Section 7 for an example.
alphabet	The user must define the alphabet in the DFA/NFA body expression when assigning a value to the DFA/NFA variable by using the **alphabet** keyword. The **alphabet** keyword takes in a list of characters contained in left and right square brackets. Refer to Section 7 for an example.
start	The user must define the start state in the DFA/NFA body expression when assigning to the DFA/NFA variable by using the **start** keyword. The **start** keyword takes in a single natural number less than or equal to the value of the states keyword. Refer to Section 7 for an example.
final	The user must define the set of final states in the DFA/NFA when assigning a value to the DFA/NFA variable by using the **final** keyword. The **final** keyword takes in a list of natural numbers less than or equal to the number of states. Refer to Section 7 for an example.
transitions	The user must define a set of transitions in the DFA/NFA when assigning a value to the DFA/NFA variable by using the **transitions** keyword. The **transitions** keyword takes in a list of 3-tuples delimited with spaces, surrounded by left and right parentheses. The tuple consists of: (initial state, input character, final state). The initial and final state are natural numbers less than or equal to the value of the **states** keyword. The input character is of type character. Refer to Section 7 for an example.
if,else	The **if** and **else** keywords will be used to construct conditional statements. Refer to Section 4 for an example.
for	The **for** keyword will be used to define loops. Refer to Section 4 for an example.
continue	The **continue** keyword will be used to move forward to the next iteration of the loop. Refer to Section 4 for an example.

break	The **break** keyword will be used to completely exit the current loop and move forward with the next code statement. Refer to Section 4 for an example.
return	The **return** keyword will be used to exit from a function with or without a value. Refer to Section 5 for an example.
case	The **case** keyword will be used to begin pattern matching. The input to pattern match on will be included right before the **case** keyword. Refer to Section **??** for an example.

simulates	The **simulates** keyword will be used to simulate an input string onto an FSM. **simulates** will be a binary operator that takes a FSM variable on the left, and an input string on the right. **simulates** returns an int value that represents the final state the FSM reached.
	Listing 4: Using "simulates"
	`dfa even_zeros;`
	`int i;`
	`even_zeros = {`
	`  states : 2`
	`  alphabet : ['0', '1', '2']`
	`  start : 0`
	`  final : [ 0 ]`
	`  transitions : [ (0 '0' 1), (1 '0' 0) ]`
	`};`
	`i = even_zeros simulates "0010"; ~~state 1`

accepts	The **accepts** keyword will be also be used to simulate an input string onto an FSM, similarly to the **simulates** keyword. The difference is the return value will be a bool value which represents if the FSM accepts or not.
	Listing 5: Using "accepts"
	`dfa even_zeros;`
	`bool b;`
	`even_zeros = {`
	`  states : 2`
	`  alphabet : ['0', '1', '2']`
	`  start : 0`
	`  final : [ 0 ]`
	`  transitions : [ (0 '0' 1), (1 '0' 0) ]`
	`};`
	`b = even_zeros accepts "0010"; ~~False`

4 Control Flow

4.1 If-Else Statements

If-else statements can be used to indicate whether or not to execute a code block. Code block inside an if statement will only execute if the condition defined after the `if` keyword evaluates to true. Else statements are optional after if statements. If an else statement is provided using the `else` keyword, the code block under the else statement will only execute if the condition after the if keyword evaluates to false. There can only be a single else statement associated with an if statement.

Listing 6: If/Else example

```
if(<condition>){
  ~~Code Block
}else{
  ~~Code Block
}
```

4.2 For Loops

For loops will be used to execute a code block more than once. The `for` keyword is followed by the code block encapsulated in left and right braces. Within for loops, the `break` and `continue` keywords can be used.

Listing 7: Using "break"

```
~~count to 5
int i;
for i = 0; i < 10; i = i + 1 {
  if(i == 5){
    break;
  }
  print(i);
}
```

Listing 8: Using "continue"

```
~~skip 5
int i;
for i = 0; i < 10; i = i + 1 {
  if(i == 5){
    continue;
  }
  print(i);
}
```

Expressio offers three different ways of using for loops.

The first and most common way is the `for` keyword followed by a start statement initializing a variable, conditional statement indicating if the loop should continue or exit, and increment statement to update the start variable for the next iteration. Each of these statements must be delimited by a semicolon.

Listing 9: For loop

```
int i;
for i = 0; i < 5; i = i + 1 {
  ~~Code Block
}
```

The second way of using the `for` keyword is by leaving the start and increment statements empty, but including a conditional statement. This type of for loop would simulate a "while" loop. The conditional statement is checked at the beginning of each iteration. The loop continues if the conditional evaluates to true and exits if it evaluates to false.

Listing 10: "While" loop

```
int i;
for ; i < 5; {
  ~~Code Block
}
```

The final way of using for loops is to only include the `for` keyword and none of the other statements. This type of for loop simulates an infinite loop. The code block will execute an unlimited number of times.

Listing 11: Infinite loop

```
for {
```

```
  ~~Code Block
}
```

5 Functions

5.1 Function Definition

Expressio offers users the ability to define functions. A function is a group of statements that together perform a task. To create a function, it must be declared syntactically as shown below:

Listing 12: Function syntax

```
<func_name> : (name1 : type1) (name2 : type2) ... -> <return type> {
  ~~CODE BLOCK
}
```

The function-name is the name of the function. This will be used to call the actual function. A colon must follow the function name.

Function arguments are optional and are defined as the name of the parameter followed by a colon and the type of the parameter. The argument must be surrounded by left and right parentheses. More than one function argument can be defined. An arrow sign must follow when all of the function arguments have been defined.

The return type is the last part of the function declaration, and is the data type of the value the function returns. If the return type is not `unit`, the `return` keyword following the actual value to be returned must be included. Following the end of the function declaration is the function body or code block that is encapsulated by left and right braces.

Listing 13: Function calling

```
<func_name> (name1,name2...)
```

Listing 14: Function with no arguments

```
main : -> int {
  ~~CODE BLOCK
}
```

Listing 15: Function with arguments

```
addTwoNums : (a : int) (b : int) -> int {
  int i;
  i = a + b;
  return i;
}
```

5.2 Built-in Functions

Expressio has a built-in function, the **print** function that comes in handy.

Listing 16: Using print function

```
print(input); ~~prints the input
```

6 Regular Expressions

Regular Expressions (REs) are first class citizens in Expressio. REs can be constructed through "constructors" which are just operators within Expressio:

- Nullary constructors:
  - The RE representing the empty language:

    ```
    {.}
    ```

  - The RE representing the empty string:

    ```
    {{.}}
    ```

- Unary constructors:
  - The RE representing a literal character, `'c'`:

    ```
    lit 'c'
    ```

  - The RE representing the Kleene star of `regexp x`:

    ```
    x **
    ```

  - The RE representing the complement of `regexp x` in Σ^*:

    ```
    `x
    ```

- Binary constructors using REs `regexp x` and `regexp y`:
  - The RE representing the union of `x` and `y`:

    ```
    x | y
    ```

  - The RE representing the concatenation of `x` and `y`:

    ```
    x ^ y
    ```

  - The RE representing the intersection of `x` and `y`:

    ```
    x & y
    ```

- Other operators involving REs:
  - A RE, `regexp x`, may be checked against a `string str` , to see if it matches, returning a `bool`:

    ```
    x matches str
    ```

Listing 17: Example RE code

```
main : -> int {
  regexp x;
  regexp y;
  regexp z;
  regexp t;
  string s;
  bool m;
  s = "c"
  x = lit 'c';
  y = x **;
  z = x | y;
  t = y ^ x;
  m = x matches s; ~~true
  return 0;
}
```

By default, the alphabet for any RE will be the `char` type.

7 State Machines

7.1 DFAs

To define a DFA, one must specify five arguments encapsulated around left and right braces. The five arguments are `states`, `alphabet`, `start`, `final`, and `transitions`. These arguments are further discussed in the keywords section and each keyword must be separated from its value by a colon.

DFAs and NFAs are first class objects in Expressio. They can be declared and initialized just as any other variables inside a function, using the "=" sign.
Example code:

Listing 18: DFA Declaration Inside a Function

```
main : -> int {
  dfa even_zeros;
  even_zeros = {
    states : 2
    alphabet : ['0', '1', '2']
    start : 0
    final : [ 0 ]
    transitions : [ (0 '0' 1), (1 '0' 0) ]
  };
  print("Example for locally declaring and initializing a dfa")
  return 0;
}
```

DFA and NFA also have extra features. A DFA or NFA can be defined (not just declared) outside a function declaration as a global variable, with the slight difference being the absence of the "=" sign and the trailing semicolon:
Example code:

Listing 19: DFA Global Scope Declaration and Assignment

```
dfa even_zeros;
even_zeros {
  states : 2
  alphabet : ['0', '1', '2']
  start : 0
  final : [ 0 ]
  transitions : [ (0 '0' 1), (1 '0' 0) ]
}
main : -> int {
  print("Example for globally declaring a dfa")
  return 0;
}
```

7.2 NFAs

Declaring and assigning to an NFA follows the same construction as a DFA with the only difference being the `nfa` keyword used instead of `dfa`. As per the definition of NFA, multiple transition tuples with the same start state and character can go to different end states. Note well that these are NFAs but not ϵ-NFAs, i.e. the transition function will be $\delta : Q \times \Sigma \to P(Q)$ and not $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to P(Q)$. Both NFAs and ϵ-NFAs (and REs for that matter) are equivalent in computational power.
Example code:

Listing 20: nfa Declaration and Assignment

```
nfa mult_trans;
mult_trans {
  states : 2
  alphabet : ['0', '1', '2']
  start : 0
```

```
  final : [ 0 ]
  transitions : [ (0 '0' 1), (1 '0' 0), (1 '0' 1) ]
}
main : -> int {
  print("Example for globally declaring a nfa")
  return 0;
}
```

8 Program Structure

An Expressio program consists of a collection of variable and function declarations along with the corresponding definitions. The entrypoint of an Expressio program starts at the main method. There can be only one main method. The return type of the main method must be an integer. The return value of the main method should indicate how the program exited. Since Expressio will not support command line arguments, the main method cannot have function arguments.

Listing 21: Main method

```
main : -> int {
  ~~CODE BLOCK
  return 0;
}
```

Variables in Expressio can have either a local or global scope. A local variable is one that has been declared inside of a function, and it can only be referenced within the block of code where it was declared. A global variable is one that has been declared outside of any function, and it can be referenced anywhere in an Expressio program after its declaration.

Listing 22: Global/Local Scope

```
dfa even_zeros; ~~Global variable
main : -> int {
  int i; ~~Local variable
  ~~CODE BLOCK
  return 0;
}
```

An Expressio program must conform to a specific structure. Any global variables must be declared at the beginning of the program. Following the declarations of global variables are function declarations. Any function declarations besides the main method must be declared before the main method. Within a function, all local variables must be declared in the beginning before any other code statements. Following all of the local variable declarations are any code statements including variable assignment. Finally, if the function does not return unit, there should be a return statement at the end.

Listing 23: Program Structure

```
~~Global variable declartions
~~Other function declarations
main : -> int {
  ~~Local variable declarations
  ~~CODE BLOCK
  return 0;
}
```

All Expressio programs are saved with the file extension *.xp*.

9 Standard Library Code

The Expressio Standard Library was constructed with the sole purpose of assisting users, whether it be for debugging or saving considerable development time. The Standard Library is imported to user programs by default.

Constructing a `regexp` could be tedious in Expressio. The reason being we do not allow users to specify an alphabet for each regular expression, which makes ASCII the default alphabet for any `regexp`. For example, when users try to define a regular expression that matches strings whose last two characters are digits, they have to manually define a `regexp` for all ASCII characters and another `regexp` for all digits. This is where our standard library is useful. The Standard Library of Expressio, defined in *.mk* (milk) files, contains pre-defined sets of commonly used regular expressions, among other helpful sets and functions. Code inside *.mk* files are written in Expressio. Example for code in *.mk* file:

Listing 24: <filename>.mk

```
regexp NUMERICS;
regexp DIGITS;
NUMERICS = lit '0' | lit '1' | lit '2' | lit '3' | lit '4'
         | lit '5' | lit '6' | lit '7' | lit '8' | lit '9';
DIGITS = NUMERICS **;
```

10 Sample Programs

This is an example for matching strings with regexp and testing if the last two digits of a string are digits.

Listing 25: Regexp example

```
~~ expressio comment
regexp re_last_2digits;
last_2digits : (s : string) -> bool {
  re_last_2digits = ASCII ** ^ NUMERICS ^ NUMERICS;
  return s matches re_last_2digits;
}

main : -> int {
  string s;
  s = "abcde10";

  if (s matches last_2digits) {
    print("Last two characters are digits!");
  }

  return 0;
}
```

This is an example for testing a `string` against a `dfa` and if that string contains even number of zeros:

Listing 26: DFA example

```
~~ comment
dfa even_zeros;
even_zeros {
  states : 2
  alphabet : ['0', '1', '2']
  start : 0
  final : [ 0 ]
  transitions : [ (0 '0' 1), (1 '0' 0) ]
}

main : -> int {
  string s;
  bool b;
  int t;
  s = "000000";
  b = even_zeros accepts s;
  t = even_zeros simulates s;
```

```
  if ((t == 0) == b) {
    print("DFA is working correctly!");
  }
  return 0;
}
```

11 Context Free Grammar

```
PROGRAM       → DECLS eof
DECLS         → DECLS VARDECL
              | DECLS DFADECL
              | DECLS FUNDECL
DFADECL       → id = {
                  states : int
                  alphabet : [ CHAR_OPT ]
                  start : int
                  final : [ INT_OPT ]
                  transitions: [ TFDECL_OPT ] }
FUNDECL       → id : FORMALS_OPT → TYP { VARDECL_LIST STMTDECL_LIST }
TYP           → int
              | bool
              | unit
              | char
              | regexp
              | string
              | dfa
FORMALS_OPT   → ϵ
              | FORMAL_LIST
FORMAL_LIST   → ( id : TYP )
              | FORMAL_LIST ( id : TYP )
INT_OPT       → ϵ
              | INT_LIST
INT_LIST      → int
              | INT_LIST , int
CHAR_OPT      → ϵ
              | CHAR_LIST
CHAR_LIST     → char
              | CHAR_LIST , char
VARDECL       → TYP id ;
VARDECL_LIST  → ϵ
              | VARDECL_LIST VARDECL
TFDECL        → ( int char int )
TFDECL_LIST   → TFDECL
              | TFDECL_LIST , TFDECL
TFDECL_OPT    → ϵ
              | TFDECL
STMT          → EXPR ;
              | return EXPR_OPT ;
              | { STMT_LIST }
              | if ( EXPR ) STMT
              | if ( EXPR ) STMT else STMT
              | for EXPR ; EXPR ; EXPR FOR_BODY
              | for ; EXPR ; FOR_BODY
              | for FOR_BODY
STMT_LIST     → ϵ
              | STMT_LIST STMT
FOR_BODY      → { STMT_LIST }
```

```
EXPR          → int
              | bool
              | char
              | string
              | id
              | reglit EXPR
              | {.}
              | {{.}}
              | EXPR + EXPR
              | EXPR - EXPR
              | EXPR * EXPR
              | EXPR / EXPR
              | EXPR == EXPR
              | EXPR != EXPR
              | EXPR < EXPR
              | EXPR <= EXPR
              | EXPR > EXPR
              | EXPR >= EXPR
              | EXPR && EXPR
              | EXPR || EXPR
              | EXPR | EXPR
              | EXPR ^ EXPR
              | EXPR matches EXPR
              | EXPR **
              | - EXPR
              | ! EXPR
              | id = EXPR
              | id ( ARGS_OPT )
              | ( EXPR )
              | { states : int
                  alphabet : [ CHAR_OPT ]
                  start : int
                  final : [ INT_OPT ]
                  transitions: [ TFDECL_OPT ] }
EXPR_OPT      → ϵ
              | EXPR
ARGS_OPT      → ϵ
              | ARGS_LIST
ARGS_LIST     → EXPR
              | ARGS_LIST , EXPT
```