XYZ Language Specification

Compilers Project

Authors

Vishal Vijay Devadiga (CS21BTECH11061)

Satpute Aniket Tukaram (CS21BTECH11056)

Mahin Bansal (CS21BTECH11034)

Harshit Pant (CS21BTECH11021)

Table of Contents

    * Assignment Statement
    * Function Declaration Statement
    * Function Call Statement
    * IO Statements
- References

Introduction

What is XYZ?

XYZ is a domain specific language that simplifies working with Finite State Machines(FSMs). Finite State Machines include Deterministic Finite Automata(DFAs), Non-Deterministic Finite Automata(NFAs), Pushdown Automata(PDAs).

It supports the following features:

- Defining Finite State Machines
- Regular Expressions
- Context Free Grammars

Why XYZ?

Finite State Machines are used in many applications, such as:

- Regular Expressions
- Lexical Analysis
- Compilers
- Network Protocols
- Digital Logic
- Artificial Intelligence
- Natural Language Processing
- etc.

Finite State Machines are used in many applications, but the syntax for defining a Finite State Machine is not very intuitive. XYZ aims to simplify the syntax for defining a Finite State Machine, making it easier for programmers to work with Finite State Machines.

Language Specifications

XYZ follows, making it easier for programmers to pick up XYZ easily and keep their focus on the logic rather than XYZ.

- XYZ is a **statically typed** language
- XYZ is a **strongly typed** language
- XYZ is a **procedural** language
- XYZ is case sensitive.

XYZ does not support Object Oriented Programming(**OOPs**).

Data Types

XYZ uses common data types found in most programming languages.

Primitive Data Types

Integer: Signed Integers are represented by the `int_x` keyword, where `x` is the number of bits used to represent the integer. XYZ supports 8, 16, 32 and 64 bit integers.

```
int_8 a;
int_8 b = 10;
int_16 c = 20;
int_32 d = 30;
int_64 e = 40;
```

Unsigned Integers are represented by the `uint_x` keyword, where `x` is the number of bits used to represent the integer. XYZ supports 8, 16, 32 and 64 bit integers.

```
unit_8 a;
uint_8 b = 10;
uint_16 c = 20;
uint_32 d = 30;
uint_64 e = 40;
```

Character: Characters are represented by the `char` keyword. XYZ supports 8 bit characters.

```
char a;
char b = 'a';
```

Float: Floats are represented by the `float_x` keyword, where `x` is the number of bits used to represent the float. XYZ supports 32 and 64 bit floats.

```
float_32 a;
float_32 b = 10.5;
float_64 c = 20.5;
```

Boolean: Booleans are represented by the `bool` keyword, which is similar to the `bool` keyword in C, C++, Java and Python.

```
bool a;
bool b = true;
bool c = false;
```

Composite Data Types

Strings: Strings are represented by the `string` keyword. Strings are immutable, and can be indexed using the `[]` operator.

```
string temp;
string a = "Hello World";
char b = a[0];
```

Finite-Sets: Sets are collections of elements of the same data type. XYZ supports two types of sets: Ordered Sets and Unordered Sets.

- Ordered Sets are represented by the `o_set` keyword.
- Unordered Sets are represented by the `u_set` keyword.

```
o_set<int_8> a;
o_set<int_8> b = {1, 2, 3};
```

Structs: Structs are represented by the `struct` keyword. Structs can contain any data type supported by XYZ.

```
struct Point {
    int_8 x;
    int_8 y;
    float_32 z;
    string name;
}
```

Regular Expressions: Regular Expressions are represented by the `regex` keyword. Regular Expression can contain definitions of other Regular Expressions, and can be used to define Finite State Machines.

```
regex alphabet = r'[a_z]';
regex Letter = r'{alphabet}';
regex Digit = r'[0_9]';
regex a = r'[ab]{2}';
regex b = r'{a}*';
regex c = r'{a}+';
regex d = r'{a}?';
regex e = r'^[ab]';
regex f = r'[ab]$';
regex g = r'{a}|{b}';
```

DFAs: DFAs are represented by the `dfa` keyword.

A DFA is defined by a 5-tuple:

$$(Q, \Sigma, \delta, q_0, F)$$

where:

- Q is a `o_set` of states
- Σ is a **set** of input symbols
- δ is the transition function, which maps $Q \times \Sigma$ to Q
- q_0 is the initial state
- F is a set of final states

A transition can be represented as:

state1, input_symbol -> state2

In case of multiple transitions from the same state on different input symbols to the same state, the transitions can be represented as:

state1, {input_symbol1, input_symbol2, ...} -> state2

This can also be done as:

state1, <regex> -> state2

state1 , <set> -> state2

δ is either a set of such transitions or it can be represented as a matrix of size $|Q| \times |\Sigma|$, where each element of the matrix is a state.

```
dfa a;
a.Q = {q0, q1, q2};
a.Sigma = {0,1,2};
a.delta = {
    q0, 0 -> q1,
    a.Q[0] , 1 -> a.Q[1],
    q1 , r'[12]' -> q2,
    q2, {0,2} -> q1
};
a.q0 = q0;
a.F = {q1,q2};
```

NFAs: NFAs are represented by the `nfa` keyword.

A NFA is defined by a 5-tuple:

$$(Q, \Sigma, \delta, q_0, F)$$

where:

- Q is a `o_set` of states
- Σ is a **set** of input symbols
- δ is the transition function, which maps $Q \times \Sigma$ to 2^Q
- q_0 is the initial state
- F is a set of final states

Here 2^Q represent the power set of Q.

A transition can be represented as:

- state1, input_symbol -> {state2, state3, . . . }
- state1, {input_symbol1, input_symbol2, . . . } -> {state2, state3, . . . }
- state1, <regex> -> {state2, state3, . . . }
- state1, <set> -> {state2, state3, . . . }

Here input_symbols can include ϵ which is represented by '\e'.

```
nfa a;
a.Q = {q0, q1, q2};
a.Sigma = {0,1,2};
a.delta = {
    q0, 0 -> {q1, q2},
```

```
    a.Q[0] , 1 -> a.Q[1],
    q1 , r'[12]' -> q2,
    q2, {0,2} -> q1,
    q2, \e -> q0
};
a.q0 = q0;
a.F = {q1,q2};
```

Comments

XYZ has only one type of comment, that can act as both single line and multi line comments. The comment starts with `<!--` and ends with `--!>`. Below is an example of a comment:

```
<!-- This is a comment --!>

<!-- This is a
multi line comment --!>
```

Operators

Operators supports by XYZ are similar to the operators supported by C.

Arithmetic Operators

Operator	Description	Associativity
`+`	Addition	Left to Right
`-`	Subtraction	Left to Right
`*`	Multiplication	Left to Right
`/`	Division	Left to Right
`%`	Modulo	Left to Right

Logical Operators

Operator	Description	Associativity
`&&`	Logical AND	Left to Right
`||`	Logical OR	Left to Right
`!`	Logical NOT	Right to Left

Comparison Operators

Operator	Description	Associativity
==	Equal to	Left to Right
!=	Not equal to	Left to Right
>	Greater than	Left to Right
<	Less than	Left to Right
>=	Greater than or equal to	Left to Right
<=	Less than or equal to	Left to Right

Assignment Operators

Operator	Description	Associativity
=	Assignment	Right to Left
+=	Addition Assignment	Right to Left
-=	Subtraction Assignment	Right to Left
*=	Multiplication Assignment	Right to Left
/=	Division Assignment	Right to Left
%=	Modulo Assignment	Right to Left
&=	Logical AND Assignment	Right to Left
|=	Logical OR Assignment	Right to Left

Set Operators

Operator	Description	Associativity
+	Union	Left to Right
-	Difference	Left to Right
*	Intersection	Left to Right
‘^2’	Power Set	Left to Right

```
o_set<int_8> a = {1, 2, 3};
o_set<int_8> b = {2, 3, 4};
o_set<int_8> c = a + b; <!-- c = {1, 2, 3, 4} --!>
o_set<int_8> d = a - b; <!-- d = {1} --!>
o_set<int_8> e = a * b; <!-- e = {2, 3} --!>
o_set<o_set<int_8>> f = a^2; <!-- f = {{}, {1}, {2}, {3}, {1, 2},
{1, 3}, {2, 3}, {1, 2, 3}} --!>
```

Automaton Operators

Operator	Description	Associativity
*	Kleene	Left to Right
@	Concatenation	Left to Right
+	Union	Left to Right
!	Negation	Right to Left

```
dfa a;
dfa b;

dfa c = a*; <!-- Kleene Star --!>
dfa d = a@b; <!-- Concatenation --!>
dfa e = a+b; <!-- Union --!>
dfa f = !a; <!-- Negation --!>
```

Misc Operators

Operator	Description	Associativity
.	Access Struct Member	Left to Right
[]	Access Set Element	Left to Right
()	Function Call	Left to Right

```
struct Point {
    int_8 x;
    int_8 y;
}

Point p;
p.x = 10;

o_set<int_8> a = {1, 2, 3};
int_8 b = a[0];

int_8 func(int_8 a, int_8 b) {
```

```
    return a + b;
}

int_8 a = func(10, 20);
```

Operator Precedence

Operator	Description
`()`	Parentheses
`!`	Logical NOT
`*, /, %`	Multiplication, Division, Modulo
`+, -`	Addition, Subtraction
`>, <, >=, <=`	Comparison
`==, !=`	Equality
`&&`	Logical AND
`||`	Logical OR
`=`	Assignment
`+=, -=, *=, /=, %=, &=, |=`	Assignment

Control Flow

XYZ enforce the use of curly braces for all control flow statements. XYZ does not support the use of indentation for control flow statements. XYZ supports the following control flow statements:

If-Else

Below is the syntax for the if-else statement:

```
if (condition) {
    statement;
}
elif (condition) {
    statement;
}
else {
    statement;
}
```

Loops

XYZ only supports the `while` loop. Below is the syntax for the `while` loop:

```
while (condition) {
    statements;
}
```

Constants

Constants are represented by the `const` keyword. Constants can be of any data type supported by XYZ.

Keywords

Keyword	Description
`int_x`	Integer
`uint_x`	Unsigned Integer
`char`	Character
`float_x`	Float
`bool`	Boolean
`const`	Constant
`struct`	Struct
`o_set`	Ordered Set
`u_set`	Unordered Set
`string`	String
`regex`	Regular Expression
`dfa`	DFA
`nfa`	NFA
`pda`	PDA
`cfg`	CFG
`if`	If
`elif`	Else If
`else`	Else
`while`	While
`break`	Break
`continue`	Continue
`return`	Return
`true`	True

Keyword	Description
`false`	False
`<!--`	Start of comment
`--!>`	End of comment

Identifiers

XYZ uses the following rules for identifiers:

- Identifiers can only contain alphanumeric characters and underscores.
- Identifiers cannot start with a number.
- Identifiers cannot be a keyword.
- Identifiers cannot contain spaces.
- Identifiers cannot contain special characters.

Regular Expressions for Identifiers:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Statements

XYZ supports the following statements:

Declaration Statement

Declaration statements are used to declare variables. Below is the syntax for declaration statements:

```
data_type identifier;
```

Multiple variables of the same data type can be declared in a single statement:

```
data_type identifier1, identifier2, ...;
```

Assignment Statement

Assignment statements are used to assign values to variables. Below is the syntax for assignment statements:

```
identifier = expression;
```

Function Declaration Statement

Function declaration statements are used to declare functions. Below is the syntax for function declaration statements:

```
data_type function_name(data_type1 arg1, data_type2 arg2, ...) {
    statements;
}
```

Function Call Statement

Function call statements are used to call functions. Below is the syntax for function call statements:

```
function_name(arg1, arg2, ...);
```

In case the function returns a value, the function call statement can be used as an expression:

```
data_type variable = function_name(arg1, arg2, ...);
```

IO Statements

Print statements are used to print values to the console. Below is the syntax for print statements:

```
out(expression);
```

Input statements are used to take input from the console. Below is the syntax for input statements:

```
inp(identifier);
```

In case multiple variables need to be inputted, the input statement can be used as:

```
inp(identifier1, identifier2, ...);
```

References

- Wikipedia: FSMs
- Wikipedia: DFAs
- Wikipedia: NFAs
- Wikipedia: PDAs
- Wikipedia: CFGs
- Wikipedia: Regular Expressions
- Michael Sipser: Introduction to the Theory of Computation
- C_Programming Language by Kernighan and Ritchie