Laboratory work nr.1
Regular Grammars

Course: Formal Languages and Finite Automata

Author: Perlog Boris FAF-221

Objectives

1. Implement a class to represent a given regular grammar.
2. Implement a function that generates a string according to the regular grammar rules.
3. Implement the conversion between an object of type RegularGrammar to one of type FiniteAutomaton.
4. Implement a function in the FiniteAutomaton class that checks if a string belongs to the language recognized by it.

1. Regular Grammar Class

The Regular Grammar class is going to be defined with the help of the **RegularGrammarDefinition** structure, it represents the mathematical definition of the grammar.

```
struct ProductionRule {
	char A;
	char terminal;
	char non_terminal;
};
struct RegularGrammarDefinition {
	std::vector<char> non_terminals;
	std::vector<char> terminals;
	std::string start_symbol;
	std::vector<ProductionRule> production_rules;
};
```

RegularGrammarDefinition structure is initialized by the **CreateGrammarDefinition()** function using the given laboratory grammar variant as input in the form of a vector of strings. The set of terminal, non-terminals and production rules are all initialized here.

```
void RegularGrammar::CreateGrammarDefinition(RegularGrammarDefinition&
grammar_def,
	const std::vector<std::string>& input)
{
	std::unordered_map<char, uint32_t> map;
	grammar_def.production_rules.resize(input.size());
	for (uint32_t i = 0; i < input.size(); i++) {
		// TERMINALS
		if (map.find(input[i][2]) == map.end()) {
			map[input[i][2]];
			grammar_def.terminals.emplace_back(input[i][2]);
		}
		// NON-TERMINALS
		if (map.find(input[i][0]) == map.end()) {
			map[input[i][0]];
			grammar_def.non_terminals.emplace_back(input[i][0]);
		}
		// PRODUCTION RULES
		grammar_def.production_rules[i].A = input[i][0];
		grammar_def.production_rules[i].terminal = input[i][2];
		grammar_def.production_rules[i].non_terminal = input[i][3];
	}
	// START SYMBOL
	grammar_def.start_symbol = input[0][0];
}
```

The **RegularGrammar** class uses two additional helper structures **NonTerminal** and **Rule**.

```
struct NonTerminal {
	char symbol = '\0';
	uint32_t rules_count = 0;
	Rule* rule_address = nullptr;

	const Rule* GetRandomRule() const;
};
struct Rule {
	char terminal;
	NonTerminal* NT_address = nullptr;
};
class RegularGrammar {
public:
	RegularGrammar(const std::vector<std::string> production_rules);

	void CreateGrammarDefinition(RegularGrammarDefinition& grammar_def,
		const std::vector<std::string>& input);

	std::string GenerateString();
private:
	void Derive(const NonTerminal* NT, std::string& word);
private:
	std::vector<NonTerminal> m_NT;
	std::vector<Rule> m_Rules;

	RegularGrammarDefinition m_GrammarDef;
};
```

The implementation creates an array of **Rule**s. The **Rule** structure contains a terminal character and a **NonTerminal***. An array of **NonTerminal**s represents the grammar's non-terminal symbols. Each **NonTerminal** entry contains the number of derivation rules available for the given non-terminal and the associated rules address. Note that **Rule**s associated with a non-terminal are adjacent in memory.

The **RegularGrammar** constructor initializes these arrays of structures as described above. These constructions (vectors of **NonTerminals** and **Rules**) are used for fast string generation using recursion.

```
RegularGrammar::RegularGrammar(const std::vector<std::string> production_rule)
{
	CreateGrammarDefinition(m_GrammarDef, production_rule);

	m_Rules.resize(m_GrammarDef.production_rules.size());
	m_NT.resize(m_GrammarDef.non_terminals.size());

	std::unordered_map<char, uint32_t> map;
	for (uint32_t i = 0; i < m_GrammarDef.production_rules.size(); i++) {
		map[m_GrammarDef.production_rules[i].A]++;
	}
	uint32_t pointer = 0;
	for (uint32_t i = 0; i < m_NT.size(); i++) {
		m_NT[i].symbol = m_GrammarDef.non_terminals[i];
		m_NT[i].rules_count = map[m_NT[i].symbol];
		m_NT[i].rule_address = &m_Rules[pointer];

		pointer += m_NT[i].rules_count;
	}
	for (uint32_t i = 0; i < m_Rules.size(); i++) {
		m_Rules[i].terminal = m_GrammarDef.production_rules[i].terminal;
		if (m_GrammarDef.production_rules[i].non_terminal == '\0') {
			m_Rules[i].NT_address = nullptr;
			continue;
		}
		for (uint32_t j = 0; j < m_NT.size(); j++)
			if (m_GrammarDef.production_rules[i].non_terminal ==
m_NT[j].symbol) {
				m_Rules[i].NT_address = &m_NT[j];
				break;
			}
	}
}
```

2. String Generation

The **GenerateString()** calls the **Derive()** function which recursively adds a terminal character at a time until it runs out of non-terminals on the left-hand side of a production rule.

```
std::string RegularGrammar::GenerateString() {
	std::string word;
	do {
		word = "";
		Derive(&m_NT[0], word);
	} while (word.length() < 5);

	return word;
}

void RegularGrammar::Derive(const NonTerminal* NT, std::string& word) {
	const Rule* rule = NT->GetRandomRule();
	const NonTerminal* next_NT = rule->NT_address;

	word += rule->terminal;
	if (next_NT != nullptr)
		Derive(next_NT, word);
}
```

The **GetRandomRule()** returns a random address of a **Rule** if a **NonTerminal** has more than one rule available. It takes into account the number of rules a **NonTerminal** has to generate a random offset in the **Rules** array.

```
const Rule* NonTerminal::GetRandomRule() const {
	uint32_t random_offset = uint32_t(rules_count * dist(e2));
	return rule_address + random_offset;
}
```

3. Regular Grammar to Finite Automaton Conversion

FiniteAutomatonDefinition structure is similar to **RegularGrammarDefinition** structure as it also represents a mathematical definition, in this case the definition of a finite automaton.

```
struct Transition {
	std::string src_state;
	std::string dst_state;
	char transition_symbol;
};
struct FiniteAutomatonDefinition {
	std::vector<std::string> Q;
	std::vector<char> sigma;
	std::vector<std::string> F;
	std::vector<Transition> delta;
};
```

The **CreateFADefFromRegularGrammar()** function uses the information from a **RegularGrammarDefinition** structure to create a **FiniteAutomatonDefinition** structure.

```
void FiniteAutomaton::CreateFADefFromRegularGrammar(FiniteAutomatonDefinition&
fa_def,
	const RegularGrammarDefinition& grammar)
{
	// Q
	const std::vector<char>& non_terminals = grammar.GetNonTerminals();
	fa_def.Q.resize(non_terminals.size() + 1);
	for (uint32_t i = 0; i < non_terminals.size(); i++) {
		fa_def.Q[i] = non_terminals[i];
	}
	fa_def.Q.back() = "*";
	// F
	fa_def.F.emplace_back("*");
	// SIGMA
	const std::vector<char>& terminals = grammar.GetTerminals();
	fa_def.sigma.resize(terminals.size());
	for (uint32_t i = 0; i < terminals.size(); i++) {
		fa_def.sigma[i] = terminals[i];
	}
	// TRANSITION FUNCTION
	const std::vector<ProductionRule>& production_rules =
grammar.GetProductionRules();
	fa_def.delta.resize(production_rules.size());
	for (uint32_t i = 0; i < production_rules.size(); i++) {
		fa_def.delta[i].src_state = production_rules[i].A;
		fa_def.delta[i].transition_symbol = production_rules[i].terminal;

		if (production_rules[i].non_terminal == '\0') {
			fa_def.delta[i].dst_state = fa_def.F[0];
		}
		else
			fa_def.delta[i].dst_state = production_rules[i].non_terminal;
	}
}
```

The **FiniteAutomaton** class uses the **State** helper structure that represents a node in the finite automaton's graph. The **m_Transitions** array of **State*** is a hash table that stores **State** addresses, a hash function named **TransitionFunction()** takes in the source **State** and the transition symbol as input and hashes it into a **State*** that represents the **State** the transition leads to. **State** addresses are inserted into **m_Transitions** using the **CreateTransition()** function.

```
struct State {
	std::string symbol;
	uint32_t index;
};
class FiniteAutomaton {
public:
	FiniteAutomaton(const RegularGrammar& grammar);

	void CreateFADefFromRegularGrammar(FiniteAutomatonDefinition& fa_def,
		const RegularGrammarDefinition& grammar);

	void CreateTransition(const std::string& st_state, const std::string&
nd_state, char symbol);
	const State& TransitionFunction(const State& state, char symbol) const;
	bool IsStringAccepted(const std::string& str);
private:
	std::vector<State> m_States;
	State** m_Transitions;

	FiniteAutomatonDefinition m_FADefinition;

	uint32_t m_FinalState_id;
};
```

The **FiniteAutomaton** constructor creates **States** using **FiniteAutomatonDefinition** structure information and inserts **State*** into **m_Transitions** using the **CreateTransition()** function.

```
FiniteAutomaton::FiniteAutomaton(const RegularGrammar& grammar)
{
	CreateFADefFromRegularGrammar(m_FADefinition,
grammar.GetGrammarDefinition());

	m_States.resize(m_FADefinition.Q.size());
	m_Transitions = (State**)calloc(m_States.size() * 26, sizeof(State*));
	for (uint32_t i = 0; i < m_FADefinition.Q.size(); i++) {
		m_States[i].symbol = m_FADefinition.Q[i];
		m_States[i].index = i;
	}
	m_FinalState_id = m_FADefinition.Q.size() - 1;

	for (uint32_t i = 0; i < m_FADefinition.delta.size(); i++) {
		CreateTransition(m_FADefinition.delta[i].src_state,
m_FADefinition.delta[i].dst_state,
			m_FADefinition.delta[i].transition_symbol);
	}
}
```

4. Finite Automaton String Recognition

The **IsStringAccepted()** function reads the input string to transition between the Finite Automaton's states. If it detects an invalid transition it returns false. If string reading finishes without invalid transitions the function returns true if the current **State** is a final **State**, otherwise returns false as the automaton is not in an accepting **State**.

```
bool FiniteAutomaton::IsStringAccepted(const std::string& str) {
	State* current_state = &m_States[0];
	for (uint32_t i = 0; i < str.length(); i++) {
		const State& next_state = TransitionFunction(*current_state,
str[i]);
		if (&next_state == nullptr)
			return false;
		current_state = (State*)&next_state;
	}
	return (current_state->index == m_FinalState_id);
}
```

Main Function

In the **main()** function a **RegularGrammar** object is instantiated with the given laboratory grammar information. Using a for loop 5 strings are generated and printed to the console.

A **FiniteAutomaton** object is instantiated with a **RegularGrammar** object. Previously generated strings are fed into the automaton. The automaton recognizes them and a “True” message is printed for each string.

Additionally, a string that does not correspond to our given grammar is fed into the automaton and a “False” message is printed as it is not recognized by the automaton.

```
int main()
{
    RegularGrammar grammar(production_rules_24);

    std::vector<std::string> strings;
    for (uint32_t i = 0; i < 5; i++) {
        strings.emplace_back(grammar.GenerateString());
        std::cout << strings.back() << "\n";
    }

    FiniteAutomaton daf(grammar);
    for (std::string& str : strings)
        std::cout << str << "\n" << (daf.IsStringAccepted(str) ? "True\n" :
"False\n");

    std::cout << (daf.IsStringAccepted("abc") ? "True\n" : "False\n");

    return 0;
}
```

```
abababadba
ababadabbbadaba
ababadabbbadaaaba
abadba
adabbdbbdaaaaaaabbbadbbbabababadaaaaaaabbbadba
abababadba
True
ababadabbbadaba
True
ababadabbbadaaaba
True
abadba
True
adabbdbbdaaaaaaabbbadbbbabababadaaaaaaabbbadba
True
False
```

Conclusion

A finite automaton and a regular grammar are two different concepts closely related. Even if their definitions are different one can easily be converted into another. Using a mathematical definition structure to describe both the finite automaton and the regular grammar helped to abstract away details about their implementation and make conversion between types much easier.

The implementation for both these types is slightly different than the mathematical definition, as when working with computers performance is desired and special types of constructions are required to achieve an efficient runtime.

When working on this laboratory work I had to make sure my implementation worked with different inputs. This way I ensured the system is robust and provides accurate results for a variety of edge cases.