Laboratory work nr.2
Finite Automata

Course: Formal Languages and Finite Automata

Author: Perlog Boris FAF-221

Objectives

1. Implement a function that classifies a grammar based on the Chomsky hierarchy.
2. Implement a function that determines if the automaton is DFA or NFA.
3. Implement a function that converts an NFA to a DFA.
4. Implement conversion of a finite automaton to a regular grammar.

1. Formal Grammar Classification

The **GetGrammarType()** function works by considering the grammar a type 3 at first and then doing various checks to verify if restrictions are met. If restrictions are not met then the type becomes one with fewer restrictions. If no restrictions are present then the grammar is of type 0.

```
uint32_t Grammar::GetGrammarType()
{
	uint32_t type = 3;
	for (uint32_t i = 0; i < m_GrammarDef.production_rules.size(); i++) {
		if (m_GrammarDef.production_rules[i].left_side.size() > 1) {
			type = 1;
			break;
		}
		uint32_t rs_terminal_count = 0;
		uint32_t rs_nonterminal_count = 0;
		for (uint32_t j = 0; j <
m_GrammarDef.production_rules[i].right_side.size(); j++) {
			if (m_GrammarDef.production_rules[i].right_side[j][0] > 91)
				rs_terminal_count++;
			else
				rs_nonterminal_count++;
		}
		if (rs_nonterminal_count > 1 || rs_terminal_count > 1)
			type = 2;
	}
	for (uint32_t i = 0; i < m_GrammarDef.production_rules.size(); i++) {
		if (m_GrammarDef.production_rules[i].left_side.size() >
			m_GrammarDef.production_rules[i].right_side.size()) {
			type = 0;
			break;
		}
	}
	return type;
}
```

This function is a method of the **Grammar** class which is a general type of grammar. It has a **GrammarDefinition** struct that represents it's theoretical definition.

```
struct GrammarProductionRule {
	std::vector<std::string> left_side;
	std::vector<std::string> right_side;
};
struct GrammarDefinition {
	std::vector<std::string> non_terminals;
	std::vector<std::string> terminals;
	std::string start;
	std::vector<GrammarProductionRule> production_rules;

};
```

2. IsNFA() function

The **IsNFA()** function is called before the finite automaton implementation if initialized as this implementation does not support non-determinism. The **IsNFA()** function uses the theoretical definition of the automaton and checks if there is a transition that leads to more than one destination state from a source state. If there are such transitions the function returns **true** and stops further checks.

```
bool FiniteAutomaton::IsNFA()
{
	std::unordered_map<std::string, uint32_t> map;
	for (uint32_t i = 0; i < m_FADefinition.delta.size(); i++) {
		std::string key = m_FADefinition.delta[i].src_state +
m_FADefinition.delta[i].transition_symbol;
		if (map.find(key) != map.end())
			return true;
		map[key];
	}
	return false;
}
```

3. NFA to DFA conversion

The **ConvertNFAtoDFA()** function uses the theoretical definition of the automaton contained in a **FiniteAutomatonDefinition** struct and creates a new set of states and a new transition function.

```
void FiniteAutomaton::ConvertNFAtoDFA()
{
	std::vector<std::vector<std::string>> new_Q;
	new_Q.resize(m_FADefinition.Q.size());
	std::unordered_map<std::string, uint32_t> map;
	for (uint32_t i = 0; i < m_FADefinition.Q.size(); i++) {
		map[m_FADefinition.Q[i]];
		std::vector<std::string> entry = { m_FADefinition.Q[i] };
		new_Q[i] = entry;
	}
	std::vector<Transition> new_delta;

	for (uint32_t i = 0; i < new_Q.size(); i++) {
		for (uint32_t j = 0; j < m_FADefinition.sigma.size(); j++) {
			std::vector<std::string> dst_set;

			GetDstStateSet(new_Q[i], m_FADefinition.sigma[j], dst_set);

			if (dst_set.size() == 0)
				continue;
			std::string src_set_name =
GetStringFromStateVector(new_Q[i]);
			std::string dst_set_name = GetStringFromStateVector(dst_set);

			if (map.find(dst_set_name) == map.end()) {
				map[dst_set_name];
				new_Q.emplace_back(dst_set);
				m_FADefinition.Q.push_back(dst_set_name);

				for (uint32_t k = 0; k < dst_set.size(); k++)
					if (IsStateFinal(dst_set[k])) {
						m_FADefinition.F.push_back(dst_set_name);
						break;
					}
			}
			// Creating the transition
			Transition transition;
			transition.src_state = src_set_name;
			transition.dst_state = dst_set_name;
			transition.transition_symbol = m_FADefinition.sigma[j][0];

			new_delta.push_back(transition);
		}
	}
	m_FADefinition.delta.clear();
	for (uint32_t i = 0; i < new_delta.size(); i++)
		m_FADefinition.delta.push_back(new_delta[i]);
}
```

It loops over every state and transition, if it finds a transition that leads to multiple states it creates a composite state and adds it to the new states set. Then transitions are analysed for these newly composed states to determine in which other states they transition. This process is repeated until there are no new composed states.

4. Finite automaton to regular grammar conversion

The **RegularGrammar** class has a constructor that takes in a **FiniteAutomaton** object. The **CreateGrammarDefinitionFromFA()** function transforms the automaton theoretical definition contained in a **FiniteAutomatonDefinition** struct into a regular grammar theoretical definition contained in a **RegularGrammarDefinition** struct. The **RegularGrammar** class already implements the functionality to create a **RegularGrammar** implementation from a theoretical definition, so all this function has to do is to create that definition and pass it further.

```
void RegularGrammar::CreateGrammarDefinitionFromFA(const
FiniteAutomatonDefinition& fa_def)
{
	// Terminals
	m_GrammarDef.terminals.resize(fa_def.sigma.size());
	for (uint32_t i = 0; i < m_GrammarDef.terminals.size(); i++) {
		m_GrammarDef.terminals[i] = fa_def.sigma[i][0];
	}
	// Non-terminals
	m_GrammarDef.non_terminals.resize(fa_def.Q.size());
	for (uint32_t i = 0; i < m_GrammarDef.non_terminals.size(); i++) {
		m_GrammarDef.non_terminals[i] = fa_def.Q[i];
	}
	// Production rules
	//m_GrammarDef.production_rules.resize(fa_def.delta.size());
	for (uint32_t i = 0; i < fa_def.delta.size(); i++) {
		ProductionRule rule;
		rule.A = fa_def.delta[i].src_state;
		rule.non_terminal = fa_def.delta[i].dst_state;
		rule.terminal = fa_def.delta[i].transition_symbol;

		m_GrammarDef.production_rules.push_back(rule);
		// if transition leads to a final state create an additional rule
without a non-terminal on the right side
		for (uint32_t j = 0; j < fa_def.F.size(); j++)
			if (fa_def.delta[i].dst_state == fa_def.F[j]) {
				rule.non_terminal = "\0";
				m_GrammarDef.production_rules.push_back(rule);
				break;
			}
	}
	// Start symbol
	m_GrammarDef.start_symbol = m_GrammarDef.non_terminals[0];
}
```

Every state of the automaton becomes a non-terminal, sigma set becomes the terminals set. Each state that leads into a final state will have an additional rule created for it meant to stop string generation. The start symbol will be the same as the start state of the automaton.

Main Function

In the **main()** function a **FiniteAutomaton** object is created using the input provided. Its states and transitions are printed.

Then a **RegularGrammar** object is created using the **FiniteAutomaton** as a parameter. Grammar non-terminals and production rules are printed.

1000 strings are generated using the grammar and checked using the automaton. If the grammar created from the automaton definition generates a word that is not recognized by the automaton a fail message will be displayed indicating that the implementation is wrong.

```
    FiniteAutomaton daf(automaton_22);

    daf.PrintStates();
    daf.PrintTransitions();

    RegularGrammar grammar(daf);

    grammar.PrintNonTerminals();
    grammar.GetGrammarDefinition().PrintProductionRules();

    for (uint32_t i = 0; i < 1000; i++) {
        std::string str = grammar.GenerateString();

        if (!daf.IsStringAccepted(str))
            std::cout << "fail!" << "\n";
    }
```

```
Automaton states:
0: State: q0
1: State: q1
2: State: q2 final
3: State: q1q2 final

Automaton transitions:
Src: q0 transition: a dst: q0
Src: q0 transition: b dst: q1
Src: q1 transition: a dst: q0
Src: q1 transition: b dst: q1q2
Src: q2 transition: b dst: q1
Src: q1q2 transition: a dst: q0
Src: q1q2 transition: b dst: q1q2

Grammar non-terminals:
q0 Rules count: 2
q1 Rules count: 3
q2 Rules count: 1
q1q2 Rules count: 3

Grammar production rules:
q0->aq0
q0->bq1
q1->aq0
q1->bq1q2
q1->b
q2->bq1
q1q2->aq0
q1q2->bq1q2
q1q2->b


D:\University\4th_Semester\limbaje_formale\LFA_lab\x64\Debug\LFA_lab.exe (process 20256) exited with code 0.
Press any key to close this window . . .
```

No fail message is displayed so we can conclude that the implementation is correct.

Conclusion

As regular grammar can be converted into a finite automaton, a finite automaton can also be converted into a regular grammar. Abstracting away the theoretical description of both the regular grammar and finite automaton in special structs proved very helpful, considering that the **RegularGrammar** class already had the functionality to implement the grammar using a definition struct all we had to do was to convert the automaton definition into a regular grammar definition and let the existing implementation do the work from there.

As the automaton implementation does not support non-determinism the conversion process from NFA to DFA took place on the automaton definition struct rather than on the **FiniteAutomaton** object.

The **IsNFA()** function is used to guard the automaton implementation initialization against a non-deterministic definition, as it is called to check the definition and is needed to call the conversion function. After the conversion, the implementation will receive a DFA as it expects.

The **GetGrammarType()** function receives a grammar of any type as long as the input form is compatible and returns the type of the grammar. The function is a method of the more general **Grammar** class. The grammar type can be used to create more specialized grammar classes like the already existing one **RegularGrammar**, or other classes for context-sensitive or context-free grammars.