An implementation approach of an deterministic finite automaton

Marcos Henrique Curtale Serafim

ABSTRACT

This article intents to approach an implementation of generator of deterministic finite automatons, given a set of tokens and regular grammars. Initially, there will be explored the basic concepts of formal languages and automatons and after that will be presented the main choices of the algorithm implementation. The language chosen to development is Javascript for no other reason than the facility to work with arrays manipulation.

1. INTRODUCTION

The study of formal languages is the study of mathematics models that makes possible the definition and recognizing of languages, their structure, characteristics and rules. A formal language is also composed by symbols and methods to combine these symbols on the formation of words and phrases. Before analysing these phrases, it is necessary verify if the given tokens (combination of symbols) are valid. This process is called the lexical analysis.

Formally, a language is formed by the alphabet and the grammar rules. With this informations is possible to construct the transition table, a way to represent the automaton itself. By the automaton generated, it's finally possible verify the productions that takes a token to the accept or denial state.

The notation used to represent the grammar rules on this work is the Backus-Naur formalism (BNF).

It is also important to clarify that this work covers only the Regular Grammars, the third type on Chomsky hierarchy.

2. DEVELOPMENT

The initial objective was to develop an deterministic finite automaton, with no unreachable or dead states and no epsilon transitions, but this work contempls only the determination process. In this way the development is divided in two parts being them the initial load and the determination process.

At the initial load an file input is provided, following some patterns, and after all the process an output file is generated with the final automaton. The algorithm is also capable to receive full tokens (e.g.: ‘if’, ‘else’, ‘then’’) and full regulars grammars. The first step is make the initial load and after execute independently the processes remaining.

2.1 Initial Load

The input file contains all the tokens to be included in the language and all grammar rules defined. To build the grammar rules, the Backus–Naur formalism was used and the following elements are used:

- < > - Usually used to contain the non-terminals/ rules productions symbols.
- ::= - Symbol definition. Before comes the rule name and after comes the productions rules.
- ε - Epsilon production, that indicates end of production.
- | - Pipe that separates each production definition

So, on a practical example, a production definition follows the model:
<ruleName> ::= alternative1<Transition> | alternative2<Transition>

Therefore, the input file contains mandatory all the tokens first and after all the grammar rules, using BNF formalism. Each token takes one line of the the file and they are separate from the regular grammars by a special line. This special line contains the symbol ‘ † ’ that the algorithm ignores and stop the reading tokens process and starts to run the reading grammars process. Following there is an input file example:

---

if
else
elseif
†
<S> ::= 1<A> | 2<A> | 3<A> | 4<A> | 5<A> | 6<A> | 7<A> | 8<A> | 9<A>

<A> ::= 1<A> | 2<A> | 3<A> | 4<A> | 5<A> | 6<A> | 7<A> | 8<A> | 9<A> | ε

The symbol ‘ † ’ is the separator symbol.

2.2 Reading token process

The algorithm initially loads the file and starts reading character by character of each line. For each character found, if it is the first char then it's added to the initial state of the automaton. If it's not, then a new state is created. When the line ends, the last rule created starts to be a end state.

On the initial load the automaton contains a lot of duplicate states but, after the determination process, this problem tends to be solved.

This process runs until the special line is found. When this occurs, the algorithm starts to run the reading regular grammar process.

2.3 Reading regular grammar process

This process also runs line by line until the file ends. To the grammar interpretation was necessary to create 3 distinct arrays , being them the terminals, non-terminals and the non deterministic automaton array itself.

What the algorithm does is read each element of the rule definition. If the element is nested between ‘<’ and ‘>’ then it is a non-terminal symbol and it is added to the non-terminals array. For each non-terminal found, a new empty space is allocated into the automaton matrix. Everything else, with exception of white spaces, pipe symbol (|) and definition symbol (::=) its considered a terminal symbol and becomes productive. For each terminal found, its added the matrix of the automaton with with its own transition. If the rule name is ‘S’, so it’s interpreted as initial state and all it´s productions are added to the state ‘0’ of the automaton. If an rule contains the epsilon symbol (ε), then this rule becomes a end state.

At the end of this process, the non deterministic automaton is generated and it will be used to run run the determination process.

2.4 Determination

This process intents to receive an non deterministic automaton, identify the rules which contains more than one transition by the same symbol and resolve it. For each rule of the automaton, the following steps to are done:

- Verify if there is to transitions by the same symbol.
- If true, creates a new state with the name of the transitions.
- Transform the symbol transition to only transitate by the new state created.
- Add to the new state all the transitions related to him.
- Verify again

The previous process is repeated until the new automaton has no transformation, being now a deterministic automaton.

2.5 Output

The output file is generated after the determination process and follows the BNF formalistm too. For the example given above, the following output were generated. It's interesting to note that the unreachable states were not removed, as the state <3>. Even it not being no longer used, it's still present on the automaton.

<0> ::= i<1> | 1<A> | 2<A> | 3<A> | 4<A> | 5<A> | 6<A> | 7<A> | 8<A> | 9<A> | e<3,7,> |

<1> ::= f<2> |

<*2> ::= <END> |

<3> ::= l<4> |

<4> ::= s<5> |

<5> ::= e<6> |

<*6> ::= <END> |

<7> ::= l<8> |

<8> ::= s<9> |

<9> ::= e<10> |

<10> ::= i<11> |

<11> ::= f<12> |

<*12> ::= <END> |

<*A> ::= 1<A> | 2<A> | 3<A> | 4<A> | 5<A> | 6<A> | 7<A> | 8<A> | 9<A> | <END>|

<*END> ::= ε<> |

<3,7,> ::= l<4,8,> |

<4,8,> ::= s<5,9,> |

<5,9,> ::= e<6,10,> |

<6,10,> ::= <END> | i<11> |

---

3. CONCLUSION

In this work was possible to understand the main techniques to create the deterministic automaton that recognize tokens of a given language. The creation of this automaton becomes useful on the process of lexical analysis to garant that all tokens are valid to before being processed on the next stages of compilation. Develop this work was a good opportunity to deep understand how to create and define formal languages and their particularities.

4. References

1. Linguagens de Programação 2006/07. Ficha 5. Gramáticas. 5.2.2 Notação BNF e EBNF (Extended BNF)
2. “BNF and EBNF: What are they and how do they work?” Avaible in: <http://www.garshol.priv.no/download/text/bnf.html>