**Varun Khadayate**

**Roll No.: A016**

**FORMAL AUTOMATA**

The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple (Q, ∑, δ, q0, F), where −

* Q is a finite set of states.
* ∑ is a finite set of symbols, called the alphabet of the automaton.
* δ is the transition function.
* q0 is the initial state from where any input is processed (q0 ∈ Q).
* F is a set of final state/states of Q (F ⊆ Q).

Automata theory is the basis for the theory of formal languages. A proper treatment of formal language theory begins with some basic definitions:

* A symbol is simply a character, an abstraction that is meaningless by itself.
* An alphabet is a finite set of symbols.
* A word is a finite string of symbols from a given alphabet.
* Finally, a language is a set of words formed from a given alphabet.

The set of words that form a language is usually infinite, although it may be finite or empty as well. Formal languages are treated like mathematical sets, so they can undergo standard set theory operations such as union and intersection. Additionally, operating on languages always produces a language. As sets, they are defined and classified using techniques of automata theory.

Formal languages are normally defined in one of three ways, all of which can be described by automata theory:

* regular expressions
* standard automata
* a formal grammar system

Regular Expressions Example

alphabet A1 = {a, b}  
alphabet A2 = {1, 2}  
language L1 = the set of all words over A1 = {a, aab, ...}  
language L2 = the set of all words over A2 = {2, 11221, ...}  
language L3 = L1 ∪ L2

language L4 = {an | n is even} = {aa, aaaa, ...}

language L5 = {anbn | n is natural} = {ab, aabb, ...}

Languages can also be defined by any kind of automaton, like a Turing Machine. In general, any automata or machine M operating on an alphabet A can produce a perfectly valid language L. The system could be represented by a bounded Turing Machine tape, for example, with each cell representing a word. After the instructions halt, any word with value 1 (or ON) is accepted and becomes part of the generated language. From this idea, one can defne the complexity of a language, which can be classified as P or NP, exponential, or probabilistic, for example.

Noam Chomsky extended the automata theory idea of complexity hierarchy to a formal language hierarchy, which led to the concept of formal grammar. A formal grammar system is a kind of automata specifically defined for linguistic purposes. The parameters of formal grammar are generally defined as:

* a set of non-terminal symbols N
* a set of terminal symbols Σ
* a set of production rules P
* a start symbol S

Grammar Example

start symbol = S  
non-terminals = {S}  
terminals = {a, b}  
production rules: S → aSb, S → ba

S → aSb → abab  
S → aSb → aaSbb → aababb

L = {abab, aababb, ...}

As in purely mathematical automata, grammar automata can produce a wide variety of complex languages from only a few symbols and a few production rules. Chomsky's hierarchy defines four nested classes of languages, where the more precise aclasses have stricter limitations on their grammatical production rules.

The formality of automata theory can be applied to the analysis and manipulation of actual human language as well as the development of human-computer interaction (HCI) and artificial intelligence (AI).

**Application Of Automata Theory**

1. Finite Automata (FA) –

* For the designing of lexical analysis of a compiler.
* For recognizing the pattern using regular expressions.
* For the designing of the combination and sequential circuits using Mealy and Moore Machines.
* Used in text editors.
* For the implementation of spell checkers.

2. Push down Automata (PDA) –

* For designing the parsing phase of a compiler (Syntax Analysis).
* For implementation of stack applications.
* For evaluating the arithmetic expressions.
* For solving the Tower of Hanoi Problem.

3. Linear Bounded Automata (LBA) –

* For implementation of genetic programming.
* For constructing syntactic parse trees for semantic analysis of the compiler.

4. Turing Machine (TM) –

* For solving any recursively enumerable problem.
* For understanding complexity theory.
* For implementation of neural networks.
* For implementation of Robotics Applications.
* For implementation of artificial intelligence.

**Other Application**

Many other branches of science also involve unbelievable levels of complexity, impossibly large degrees of variation, and apparently random processes, so it makes sense that automata theory can contribute to a better scientific understanding of these areas as well. The modern-day pioneer of cellular automata applications is Stephen Wolfram, who argues that the entire universe might eventually be describable as a machine with finite sets of states and rules and a single initial condition. He relates automata theory to a wide variety of scientific pursuits, including:

* Fluid Flow
* Snowflake and crystal formation
* Chaos theory
* Cosmology
* Financial analysis

**Tools for Formal Automata Theory**

* Automata simulators are pedagogical tools used to teach, learn and research automata theory.
* An automata simulator takes as input the description of an automaton and then simulates its working for an arbitrary input string.
* The description of the automaton can be entered in several ways. An automaton can be defined in a symbolic language or its specification may be entered in a predesigned form or its transition diagram may be drawn by clicking and dragging the mouse.
* Well known automata simulators include Turing's World, JFLAP, VAS, TAGS and SimStudio.
* JFLAP is for experimenting with automata, pushdown automata and Turing machines; LLparse and LRparse are for experimenting with top-down and bottom-up parsing; Pate is both a brute force parser for restricted and unrestricted grammars and a grammar transformer from a context-free grammar to CNF; and PumpLemma is a tool for experimenting with the pumping lemma.

**Enhancements to LLparse and LRparse**

LLparse and LRparse are instructional tools for constructing LL and SLR parse tables through a series of steps, and then animating the stack in parsing input strings.

In LLparse, the user enters a grammar and can proceed if the grammar is an LL grammar. The user then enters FIRST sets of variables and then enters FOLLOW sets of variables. If a set is incorrect, the set is highlighted and the user must either change the set or select Show to see the answer. Finally, the user fills in an LL parse table with the appropriate entries. Once correct, a parsing window appears. The user can enter any input string and then step through its parsing. Enhancements to LLparse include handling LL gram- This material is based upon work supported by the National Science Foundation’s Division of Undergraduate Education through grants DUE9596002 and DUE-9555084. mars and presenting the parse tree when parsing a string. When a user enters a grammar, he/she can select to construct either an LL or LL parse table In constructing the LL parse table, the user enters FIRST sets, FIRST2 sets (for 2 lookaheads), FOLLOW sets and FOLLOW2 sets. The LL parse table shown will automatically compress columns where only 1 lookahead instead of 2 are needed, reducing the size of the table.

In LRparse, the user enters an LR grammar, followed by FIRST and FOLLOW sets. Then in a drawing window, the user draws a transition diagram of a FA that models the stack. For each state in the FA, the user enters the item set (marked rules). As in all windows, the user must enter the correct information before proceeding. There is a Show button if the user wants to see the answer. Next, the user fills in the LR parse table. Finally, a parsing window appears and the user can step through the parsing of an input string. Enhancements to LRparse include parsing any context free grammar with conflict resolution and showing the parse tree when parsing an input string. If a grammar is not LR, there will be multiple items in at least one entry of the parse table. The first item in the entry will be selected in parsing. Thus the user can experiment with choosing different items as the first item. In the parsing window, the user enters an input string and steps through the parsing process. In addition to a stack trace, when a rule is reduced, the reduction is shown by joining the right hand side of the rule in the parse tree.

**JFLAP**

JFLAP is a Java implementation of FLAP. In JFLAP, one can graphically construct a transition diagram for nondeterministic versions of finite automata, pushdown automata, and 1-tape and 2-tape Turing machines. Figure shows a 1-tape Turing machine in JFLAP that adds unary numbers. Once constructed, an input string is entered and either fast or step run is selected. In fast run, a message quickly responds indicating the acceptance of the string. In step run mode, all current configurations (there is more than one if the machine is nondeterministic) are shown at each step and the user must control the trace by freezing or killing configurations if there are more than 15.

**Pate-Brute Force Parser**

The brute force parser part of Pâté is an exhaustive search parser for restricted (regular and context-free) and unrestricted grammars. Given a grammar and an input string, the parser builds a derivation tree (not displayed) of all possible derivations in a breadth-first manner.

* Each node in the tree contains a sentential form and the production number used to get from its parent node to itself. The start symbol S is the root of the tree. A derivation of the input string is found when the input string appears as a sentential form (a node).
* To speed up the creation of the derivation tree, the user can choose to allow only one such node for each sentential form in the tree. Thus, each new sentential form generated is first looked up to see if the sentential form already exists, and if so it is not added.
* For restricted grammars, additional pruning of nodes is accomplished by eliminating nodes whose prefix, suffix or substring of terminals do not match in the input string. Once a derivation is found, a message indicates the acceptance of the string and the size of the derivation tree.
* The user can choose to display the actual derivation in textual format or in the form of a parse tree (for restricted grammars only).
* If all nodes in the derivation tree are exhausted and the string is not found, a message indicates that the string is not in the language of the grammar.
* Pâté is an instructional tool to experiment with small grammars and small input strings, and works well for most assignments given to beginning students. Obviously, for some strings and grammars, the parsing may take a long time.
* For large sizes of the derivation tree, messages appear indicating to the user the size of the tree and asking if they want to continue. The user can also pause the parsing at any time for the same information.

We give an example of Pâté parsing a string in a grammar. Figure 1 shows a portion of the initial Pâté window. Either a restricted or unrestricted grammar is entered. In this case, a context-free grammar is entered (; represents the empty string). There are two buttons at the bottom of the window (not shown) for selecting either Parser or Transform Grammar, and a message window to indicate if the grammar is in the correct format. The Parser button is selected and the parsing window in Figure  2 appears (except the string and window are blank). The user enters an input string at the top and selects Parse. While the parser is busy, animations of fractals appear on the Parse button. When the parser completes, the derivation and information are displayed in the bottom half of the window. Figure 2 shows the derivation for the string abbcca. The derivation tree was quite large, containing 2114 nodes or sentential forms. Alternatively for restricted grammars only, one can select graphical output and the parse tree is shown.

Pate-Grammar Transformer

The grammar transformer part of Pâté is an instructional tool for converting a context-free grammar to Chomsky Normal Form (CNF) through a series of steps. At each step the user is requested to enter information and cannot proceed until the information is correct. There is a Show button to show the answer in case the user is stuck. To use the transformation tool, the user enters a context free grammar in the original Pâté window and selects Trans Figure 2: Derivation of abbcca form Grammar. The first step is to remove lambda productions. A window appears and the user first enters the variables that derive lambda productions. Once verified these are correct, the user enters the new grammar without lambda productions. No windows are removed, so the user can look at the grammar window to see the original grammar.The user would click on the button Next Transformation at the bottom of the window (not shown in the figure) to move to the next transformation. The next two steps remove unit productions and useless productions. Both of these steps include a window to draw a graph modeling how unit productions are connected (in the first case) and which variables can be replaced by other variables (in the second case). The final step is a window for constructing the CNF grammar. In this window, the user is informed of the format for additional variables, B(x) for a new variable deriving the single terminal x and D(#) (where # is an integer) for other new variables. Informative error messages tell the user which rules are typed incorrectly or have not yet been replaced. For some grammars, all steps in the conversion process may not be needed. A user is informed if a step does not apply and can skip the step (if a grammar does not have unit productions, none need to be removed).

**PumpLemma**

PumpLemma is a tool for experimenting with the regular pumping lemma. The user enters a non-regular language L and through steps, tries to prove that L is not regular. The user chooses the string w such that w 2 L and jwj  m. This string w should be partitioned into three parts: x, y and z such that jxyj  m, jyj > 0, and for all i  0, xyiz is in the language. If the string cannot be partitioned to meet these conditions, then the language L is not regular. PumpLemma begins with the user typing in a language. Figure 7 shows the top portion of the PumpLemma window in which the user has typed in the language an=2bn . Currently language selection is limited to ordered languages (i.e. an bn (ab)n , but not “the number of a’s equals the number of b’s”). When the user presses return, the language is parsed, and the user can continue if the language is legal. Next, the user can define the ranges of variables. For conventional purposes the only legal variables are n-s. If a language fails parsing due to poorly defined variables, an error message will appear.

The string w is entered in the same manner as the language, except the only legal exponents are combinations of m and integers. If the string is syntactically correct, it’s length is then checked before proceeding. If the string is long enough, then the case list is automatically generated. In Figure 7 the user entered the string w = am=2bm and three cases were generated. The case list represents all the possible values for the y field. A well chosen string will yield fewer cases. (Note that in Figure 7, if the user had chosen w = am b 2m there would be only one case.) The user now selects a case (the case is highlighted) and fills in values for the substrings x, y and z. The contents of the substrings are stored for each case, so the user can switch between cases without losing information.

The x, y, and z fields have their own set of exponents, ranging from h through l. After the y string is typed in, it is checked against all possible cases for a match. If it does not match any case, then the xyz cannot possibly equal w, and an error is generated. Otherwise, the case it matches becomes the current case. Figure 8 shows the bottom of the PumpLemma window for the example in Figure 7. After the final substring field is entered the substrings are concatenated together and verified that they form w. The user then chooses i, the value to pump, and presses the run button and xyiz is generated in the Resulting String box to the right. A message displays whether the resulting string is in the language. The case is colored red if the string is not in the language, and green if it is. The user then proceeds through all cases, trying to disprove them. When all cases are red, the language has been proven not to be regular.