ECS 120 Lesson 2 – Alphabets, Languages and Grammars, Pt. 2; Finite State Machines, Pt. 1

Oliver Kreylos

Monday, April 2nd, 2001

1 Languages

A *(formal) language* L over an alphabet Σ is a set of words over Σ; that is, L is a subset of Σ^*. Examples:

- The set of all english words over the latin alphabet: $\{\text{a}, \text{Aardvark}, \text{aback}, \ldots, \text{Zulu}, \text{zymotic}\}$.
- The set of all correctly formed arithmetic expressions over the alphabet $\{\texttt{a}, \texttt{(}, \texttt{)}, \texttt{+}, \texttt{*}\}$: $\{\texttt{a}, \texttt{(a)}, \texttt{a+a}, \texttt{a*(a+a)}, \ldots\}$.
- The set of all syntactically correct `C` programs over the ASCII alphabet: $\{\texttt{main() \{\}}, \texttt{int main() \{return 0;\}}, \ldots\}$.

In natural languages, words are only the building blocks for more complex constructs, e. g., phrases and sentences, whereas in formal languages words usually have no relationship to each other and stand on their own.

To specify a language, one has to specify exactly which words over the alphabet are included in the language. There are at least three different methods to do this:

- Provide a mathematical description of the set of words, either by listing all words explicitly or giving a property each word must have, e. g., $L = \{ w \in \text{ASCII}^* \mid w \text{ is a syntactically correct } \texttt{C} \text{ program} \}$.
- Provide a *grammar* that can construct all words in the language.

- Provide an *automaton* that can decide whether a given word is in the language or not.

The first method is very general – too general to be generated or checked by automatic means.

2 Grammars

A grammar describes how to create words of a language. Formally, a grammar G is a 4-tuple $G = (V, \Sigma, R, S)$, where

1. V is a finite set of *variables* or *nonterminals*.
2. Σ is an alphabet of *terminals* disjoint from V, i. e., $V \cap \Sigma = \emptyset$.
3. R is a finite set of *substitution rules* or *productions* of the form $u \to v$, where $u, v \in (V \cup \Sigma)^*$ and $u \notin \Sigma^*$, i. e., u contains at least one variable.
4. $S \in V$ is the *start symbol*.

A word is *generated* by a grammar using the following substitution algorithm:

1. Create a *current string* $c \in (V \cup \Sigma)^*$ and set it to the start symbol S.
2. Find any rule $u \to v \in R$, such that u is a substring of c, i. e., there exist $x, y \in (V \cup \Sigma)^*$ such that $c = xuy$.
3. Replace c with xvy.
4. Repeat from step 2 until $c \in \Sigma^*$, i. e., c contains only terminals.

A single application of steps 2 and 3 of the algorithm is called a *derivation* and denoted by the symbol $\Rightarrow$. Thus, any word w is generated by applying a finite number of derivations to the start symbol S: $S \Rightarrow c_1 \Rightarrow c_2 \Rightarrow \cdots \Rightarrow c_n \Rightarrow w$. Such a chain of derivations is often abbreviated by the symbol $\Rightarrow^*$, e. g., $S \Rightarrow^* w$. The set of all words that can be generated by a given grammar G form the language $L(G)$ of the grammar. Formally, $L(G) := \left\{ w \in \Sigma^* \;\middle|\; S \Rightarrow^* w \right\}$.

2.1 Example

Consider the grammar $G = \big(\{E, T, F\}, \{\texttt{a}, \texttt{(}, \texttt{)}, \texttt{+}, \texttt{*}\}, \{E \to E\texttt{+}T, E \to T, T \to T\texttt{*}F, T \to F, F \to \texttt{(}E\texttt{)}, F \to \texttt{a}\}, E\big)$. We can generate the following words from it:

- $E \Rightarrow T \Rightarrow F \Rightarrow \texttt{a}$
- $E \Rightarrow T \Rightarrow F \Rightarrow \texttt{(}E\texttt{)} \Rightarrow \texttt{(}T\texttt{)} \Rightarrow \texttt{(}F\texttt{)} \Rightarrow \texttt{(a)}$
- $E \Rightarrow E\texttt{+}T \Rightarrow T\texttt{+}T \Rightarrow F\texttt{+}T \Rightarrow \texttt{a+}T \Rightarrow \texttt{a+}F \Rightarrow \texttt{a+a}$
- $E \Rightarrow T \Rightarrow T\texttt{*}F \Rightarrow F\texttt{*}F \Rightarrow \texttt{a*}F \Rightarrow \texttt{a*(}E\texttt{)} \Rightarrow \texttt{a*(}E\texttt{+}T\texttt{)} \Rightarrow \texttt{a*(}T\texttt{+}T\texttt{)} \Rightarrow \texttt{a*(}F\texttt{+}T\texttt{)} \Rightarrow \texttt{a*(a+}T\texttt{)} \Rightarrow \texttt{a*(a+}F\texttt{)} \Rightarrow \texttt{a*(a+a)}$

Upon closer inspection, we recognize $L(G)$ as the language of correctly formed arithmetic expressions encountered earlier.

3 The Chomsky Hierarchy

Grammars, as a means to formalize the construction of languages, have traditionally been studied by linguists. The most important contribution to computer science by a linguist was made by Noam Chomsky, who around 1950 invented a classification of grammars into four levels:

- CH-0 is the set of all grammars, as defined above.
- Grammars in CH-1 are restricted by requiring that for each rule $u \to v \in R : |u| \leq |v|$, i. e., the right-hand side of each rule is at least as long as the left-hand side. Grammars in CH-1 are called *context-sensitive grammars*.
- Grammars in CH-2 are further restricted by requiring that for each rule $u \to v \in R : u \in V$, i. e., the left-hand side of each rule is a single variable. Grammars in CH-2 are called *context-free grammars*.
- Grammars in CH-3 are further restricted by requiring that for each rule $u \to v \in R : v \in \Sigma \cup \Sigma V$, i. e., the right-hand side of each rule is either a single terminal or a single terminal followed by a single variable. Grammars in CH-3 are called *regular grammars*.

From these definitions follows that CH-3 $\subset$ CH-2 $\subset$ CH-1 $\subset$ CH-0. This hierarchy, the *Chomsky Hierarchy*, fell short of its original goal to classify natural languages, but it is perfectly able to categorize formal languages and has been used by computer scientists ever since. By classifying grammars, the Chomsky Hierarchy also classifies the languages constructed by those grammars:

- Languages in L(CH-0) are called *recursively enumerable languages*.
- Languages in L(CH-1) are called *context-sensitive languages*.
- Languages in L(CH-2) are called *context-free languages*.
- Languages in L(CH-3) are called *regular languages*.

From these definitions follows that L(CH-3) $\subset$ L(CH-2) $\subset$ L(CH-1) $\subset$ L(CH-0).

3.1 Discussion Problems

- Are there languages which are *not* in L(CH-0)? In other words, are there languages which can *not* be specified by a grammar?

4 Automata

Automata are abstract models of computation, and are used to *decide* formal languages. Each automaton A has a language $L(A)$ associated with it, but as opposed to grammars, an automaton does not generate words from $L(A)$, but it reads a word $w \in \Sigma^*$ and decides whether $w \in L(A)$, see Figure 1. The most powerful type of automaton, the *Turing Machine*, was first introduced by the english mathematician Alan Turing in the 1930s, and less powerful types were introduced later. Interestingly enough, though automata and grammars are two very different ways to specify languages, they are also very closely related: Each class of languages, as defined by the Chomsky Hierarchy, has an associated type of automaton that recognizes the same class of languages:

- Languages in L(CH-0) are recognized by *Turing Machines*.
- Languages in L(CH-1) are decided by *Length-limited Turing Machines*.

- Languages in L(CH-2) are decided by *Push-down Automata* (PDAs) or *Stack Machines*.
- Languages in L(CH-3) are decided by *Finite State Machines* (FSMs).

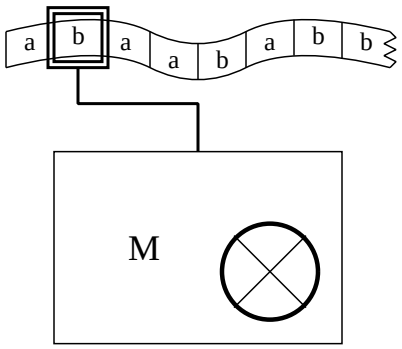

Figure 1: An automaton M decides its language $L(M)$ by reading a word $w \in \Sigma^*$ from its input tape and displaying its decision whether $w \in L(M)$ by some means, e. g., by lighting a lamp.

5 Finite State Machines

A Finite State Machine (FSM) is the simplest model of computation. It is an automaton that reads a word one character at a time and displays its decision after having read all characters. Since an FSM cannot go back in its input or read characters more than once, it needs to "remember" which characters it has already read. In a finite state machine, memory is represented by a (finite) set of *states*. A very simple example of an FSM is shown in Figure 2; a slightly more interesting example is shown in Figure 3.

A finite state machine works in the following way:

1. The machine starts out in the start state, denoted by an incoming arrow.
2. The machine waits for an event (reading of a character); as soon as a character $c \in \Sigma$ arrives, the machine will follow an arrow labeled by c to another state.
3. The machine repeats step 2 until all characters are read.

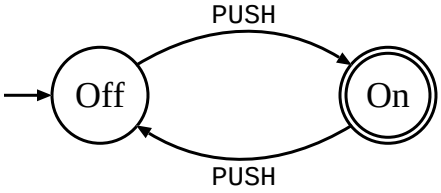

Figure 2: An FSM for a simple pushbutton mechanism. The machine starts out in the "Off" state, and remains there until a `PUSH` event happens. Then it moves to the "On" state, where it remains until another `PUSH` event happens.

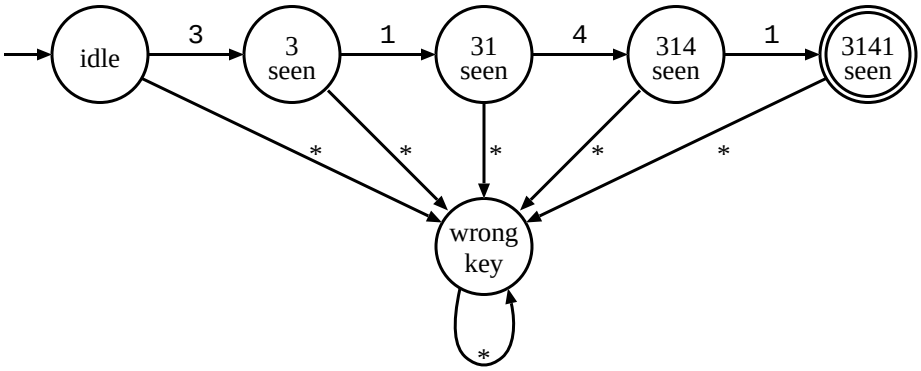

Figure 3: A simple four-digit door lock. The lock only accepts a single digit sequence, 3141. If any other sequence is keyed in, the lock will go to and remain in the "wrong key" state.

4. If the machine ends up in a final state (denoted by a double circle), it accepts the read word.

The language $L(M)$ defined by a finite state machine M is the set of words $L(M) := \left\{ w \in \Sigma^* \;\middle|\; M \text{ ends in a final state after reading word } w \right\}$. In this notation, the language over the alphabet $\Sigma_{\mathrm{PB}} = \{\texttt{PUSH}\}$ (the symbol `PUSH` is treated as a single character here) accepted by the pushbutton automaton is the set of all words containing an odd number of characters; and the language over the alphabet $\Sigma_{\mathrm{DL}} = \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$ accepted by the doorlock automation is $L(\mathrm{DL}) = \{\texttt{3141}\}$.