

Lecture 17. Automata and Grammars II. Push-Down Storage Automata and Context-Free Grammars

Note about Lecture 16: there is only an old handwritten handout for Lecture 16 (Nov 20 and 27) on semantic automata. The reading for that was:

van Benthem, Johan. 1986. Semantic automata. In *Studies in Discourse Representation Theory and the Theory of Generalized Quantifiers (GRASS 8)*, eds. J. Groenendijk, D. de Jongh and M. Stokhof, 1-25. Dordrecht: Foris.

Reading:

Chapter 18, "Pushdown Automata, Context Free Grammars and Languages" of PtMW, pp. 485-504.

1. Pushdown automata

A *pushdown automaton*, or *pda*, is essentially a finite state automaton augmented with an auxiliary tape on which it can read, write, and erase symbols. Its transitions from state to state can depend not only on what state it is in and what it sees on the input tape but also on what it sees on the auxiliary state, and its actions can include not only change of state but also operations on the auxiliary tape.

The auxiliary tape works as a *pushdown store*, "last in, first out", like a stack of plates in some cafeterias. You can't 'see' below the top item on the stack without first removing (erasing) that top item.

States: as in a fsa, a pda has a finite number of states, including a designated initial state and a set of "final" or "accepting" states.

Transitions: $(q_i, a, A) \rightarrow (q_j, \Gamma)$, where q_i, q_j are states, a is a symbol of the input alphabet, A is either the empty string e or a symbol of the stack alphabet (which need not be the same as the input alphabet), and Γ is a string of stack symbols, possibly the empty string.

Interpretation: When in state q_i , reading a on the input tape, and reading A at the top of the stack, go to state q_j , and *replace A by the string Γ* . The elements of the string Γ go onto the stack one at a time, so the last symbol of Γ ends up as the top symbol on the stack. In case Γ is the empty string e , the effect is to erase A that is to remove it ("pop" it) from the stack. ("Push" down and "pop" up.)

Examples of A and of Γ and their effects. First suppose that A is a particular symbol (and let's just call it A .) If Γ is BC , the result is to remove A , and add B on the top of the stack, and then add C on top of B . If Γ is e , the result is to remove A , and the new top symbol will be whatever was just below A , if anything. If Γ is AB , the result is to add a B on top of the A that was there. If Γ is A , then the stack will be unchanged.

If A is e , that means that the transition does not depend on what's on the stack; it does *not* mean that the stack must be empty. (We don't have any special way of indicating that the stack is

empty, though you can design an individual machine to start by putting some particular symbol on the stack, and to be sensitive to when it sees that symbol again as a way of knowing its stack is empty except for that one symbol.) If A is e and \sqsupset is also e , the stack stays unchanged. Otherwise \sqsupset is added to the stack.

Initial configuration: the stack is assumed to be empty at the beginning of a computation, with the pda in its initial state and the reading head looking at the first (leftmost) symbol on the input tape. A pda accepts an input tape if the computation leads to a situation in which all three of the following are simultaneously true:

- (i) the entire input has been read;
- (ii) the pda is in a final (accepting) state;
- (iii) the stack is empty.

[NB: one could define acceptance by final state, or by empty stack, or, as here, by both simultaneously, and the resulting classes of automata turn out to be equivalent.]

Example 1. A deterministic pda for the classic context-free and non-finite-state language $\{a^n b^n \mid n \geq 0\}$.

(on blackboard. See p. 486.)

Example 2: A pda for another classic context-free and non-finite-state language: $\{xx^R \mid x \in \{a, b\}^*\}$

(on blackboard. See pp 487-8.)

This machine is non-deterministic. Acceptance for non-deterministic vs deterministic machines is defined just as for finite state automata: a non-deterministic pda accepts a string iff there is *some* computation which leads to acceptance.

The natural question: are deterministic and non-deterministic pda's equivalent? Answer: NO. The proof is not simple and it isn't included in PtMW. Intuitively, in the given case, we can see that there is no deterministic way for a pda to know when it is at the middle of the string in Example 2, and it needs to know that in order to know when to stop adding symbols and start matching and erasing.

By contrast, the language $\{xcx^R \mid x \in \{a, b\}^*\}$, in which the center is marked by a distinctive symbol c , is acceptable by a deterministic pda.

Example 3: A pda for the language of exercise 1c, p. 503: $L_3 = \{x \mid x \in \{a, b\}^* \text{ and } x \text{ contains twice as many } b\text{'s as } a\text{'s}\}$

Note: both example 3 and example 1 can be done with a 1-symbol pushdown alphabet; these are sometimes called "counter machines", since they are not requiring the full power of a pushdown store with arbitrary alphabet. The mirror-image language, by contrast, requires as many distinct symbols in the pushdown alphabet as there are in the input alphabet.

2. Context-free grammars and languages

Non-deterministic pda's accept exactly the languages generated by context-free (Type 2) grammars. The regular languages are a proper subset of the context-free languages.

The proof of equivalence is fairly long and complex. PtMW give just one part: an algorithm for constructing from any context-free grammar an equivalent non-deterministic pda (but not the formal proof that the constructed automaton is indeed equivalent.)

[blackboard. See pp. 490-1]

The automaton M just constructed works as a simple parser for the grammar G . M starts by loading the symbol S onto its stack and then simulating a derivation of the string it finds on its input tape by manipulations that correspond to the rules of G . M in general has to be non-deterministic, since the grammar may have more than one way of rewriting a given non-terminal symbol A .

3. Pumping theorem for context-free languages

The pumping theorem for cfl's is similar in form to that for fal's. It is used primarily for showing that a given language is not context free.

Key facts used in the theorem: a derivation by a cfg can be naturally associated with a parse tree, and the maximum width of any such parse tree is constrained by its height.

For discussion and pictures, see pp 492-494.

Theorem 18.1: If L is an infinite context free language, then there is some constant K such that any string w in L longer than K can be factored into substrings $w = uvxyz$ such that v and y are not both empty and $uv^i xy^i z \in L$ for all $i \geq 0$.

Note that this pumping theorem is a conditional but not a biconditional.

Examples: Show that $a^n b^n c^n$ is not context-free.

Show that xx is not context-free (assuming an alphabet of more than one letter).

Interesting example: The language of first-order logic, with and without constraints on vacuous quantifiers or unbound variables.

4. Closure properties of context free languages.

to do

5. Decidability properties of context free languages.

to do

[No homework. If there were, it would be to do some or all of the questions at the end of Chapter 18.]