CS 361 Meeting 18 — 4/10/20

Announcements

1. Nothing today.

Building a PDA for a CFG

(Click for video)

1. The set of languages that can be described by a push down automaton is exactly the same as those that can be described using a context-free grammar. That is, both notations provide exactly the same expressive power.

2. To establish this fact, one has to show both that the language of any context-free grammar can be described by a push down automaton and that any language described by a push down automaton can be described by a context-free grammar.

3. Sipser's text provides the details of proofs for both these claims. Rather than reproducing these proofs, my goal will be to work examples that illustrate the intuition behind the constructions used to complete the proofs.

4. Oddly, in presenting these examples, I won't strictly follow the transformation algorithms included in Sipser's proofs. Algorithms can't use intuition. They have to follow simple rules for converting an input to an output. I believe it will be more useful for our understanding of the algorithms to perform the transformations using the intuitions underlying the algorithms but taking advantage of insights that will allow us to take shortcuts that will simplify the overall process enough to make understanding the resulting machines/grammars possible.

5. First, let't think about how one can design a PDA to accept the same language as some given context-free grammar, G.

   - To make things concrete, suppose we attempt to do this for the glob-blob grammar:

     $B \rightarrow$ x G B y
     $B \rightarrow$ z
     $G \rightarrow$ a G
     $G \rightarrow$ ϵ

   - Consider a simple derivation relative to this grammar:

     $$B \Longrightarrow xGBy \Longrightarrow xaGBy \Longrightarrow xaBy \Longrightarrow xaxGByy \Longrightarrow xaxByy \Longrightarrow xaxzyy$$

   - Now, let's look at the same derivation from a different angle/perspective.

     $$\begin{array}{ccccccccccccc}
      & & & & & & & & & & & & x \\
      & & & & & & & & & & & & a \\
      & & & & & & & & & & & & x \\
      & & & & & & & & x & & x & & z \\
      & & & & x & & x & & a & & a & & y \\
      & & x & & a & & a & & x & & x & & y \\
     B & \Longrightarrow & G & \Longrightarrow & G & \Longrightarrow & B & \Longrightarrow & G & \Longrightarrow & B & \Longrightarrow & \\
      & & B & & B & & y & & B & & y & & \\
      & & y & & y & & & & y & & y & & \\
      & & & & & & & & y & & & &
     \end{array}$$

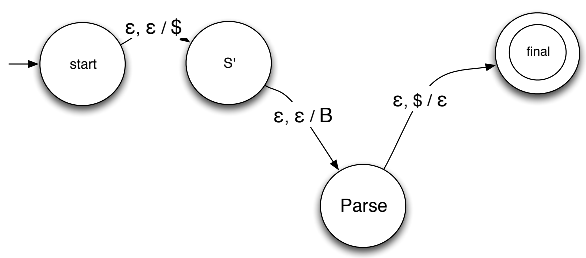
   - This is the same derivation, but each sentential form is:
     - written with its symbols running vertically rather than horizontally, and
     - aligned so that the first (i.e. left-most) variable found in each sentential form is on the same line as the derivation symbol ($\Longrightarrow$), or (more precisely) aligned so that all terminal symbols preceding the first variable are above the line holding the derivation symbols.

   - This presentation of the derivation is intended to suggest a process one might follow when looking for a derivation for a given string/input that can be emulated by a PDA.
     - Start with a sentential form consisting of just the start symbol.
     - Repeatedly:

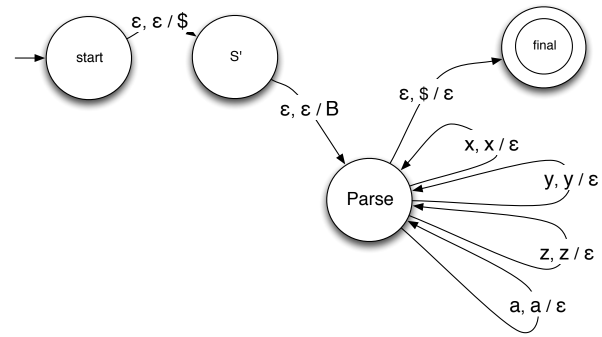
Click here to view the slides for this class

  * Make sure that the prefix of the current sentential form preceding the first variable matches the equal length prefix of the input string.
  * Based on the next few symbols of the string to be derived, choose a production to apply to the leftmost/topmost non-terminal in the current sentential form.
  * Apply the production to get the next sentential form
- Stop when the sentential form is a sentence that matches the target string.

- To build a PDA that implements this approach, we will keep the suffix of the current sentential form starting at the leftmost non-terminal on the PDA's stack.
- We must provide transitions to let the PDA replace a non-terminal at the top of the stack with the symbols on the right hand side of some rule for the symbol on top of the stack.
- We need to also provide transitions to let the PDA "cross off" any terminals in the prefix of the current sentential form that match the next symbols in the input as it reads those symbols.
- We begin with states and transitions to push the initial sentential form (the start symbol) and an end of stack marker onto the stack:

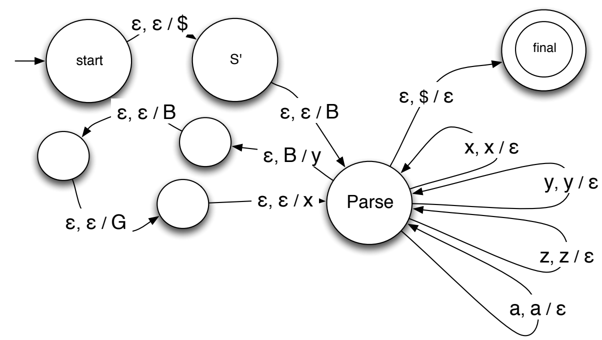


- Next, for all terminal symbols, we add transitions to allow the machine to match symbols in the input to symbols preceding the first variable in the current sentential form:

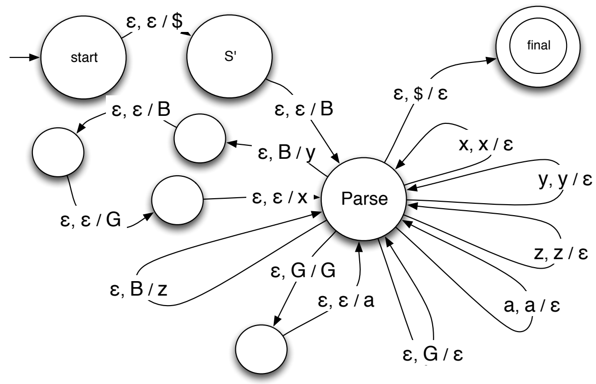


- Next, for each rule in the grammar, we add sequences of transitions through additional states (as necessary) that enable the machine to replace any variable on the top of the stack with the symbols found on the right side of one of that variable's productions.

  The diagram below shows the components added just to handle the production $B \to xGBy$:



- Doing the same for the other three productions in G gives us a complete PDA for the language:



- The text provides a detailed proof, but I hope it is clear that the same procedure can be used to produce a PDA for the language of any context-free grammar. Thus the languages accepted by PDAs includes all context-tree languages.

Building a CFG for a PDA

(Click for video)

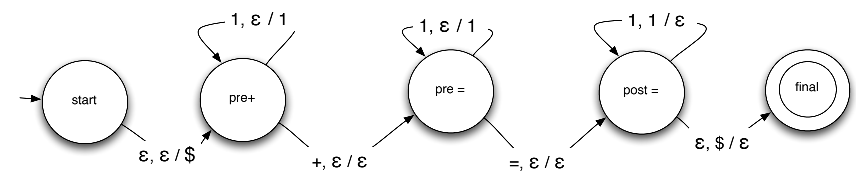
1. To complete the proof that PDAs and CFGs describe the same set of languages, we need to show that it is possible to construct a grammar that describes the same language as any, given PDA.

- The proof given in Sipser and the approach we will illustrate here depends on assuming that the PDA we start with has three special properties:
  - The machine has a single accept state.
  - The machine always empties its stack before entering the accept state.
  - Every transition either pushes or pops a stack symbol (but not both).
- It is easy to convert a given PDA to have all three of these properties. All of the machines we have considered have already had the first two!

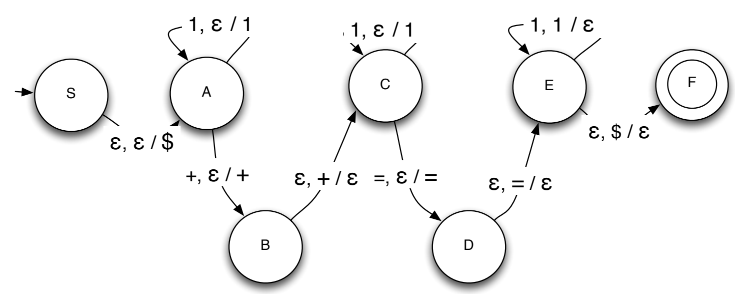
- As an example, we will use a machine for the language used as an example earlier:

$$L_{add} = \{1^i + 1^j = 1^{i+j} \mid i, j \geq 1\}$$

- The machine we discussed for this language earlier does not have the third required property:



- We can revise the machine to satisfy the third property by adding a few extra states turning single steps in which no symbol was pushed or popped in two steps that lead to the same destination, but push then pop some symbol:



- Given such a machine, Sipser's strategy for constructing a grammar for the machine's language is to define a grammar with a set of non-terminals $\{A_{pq} \mid p, q \in Q\}$ and productions chosen so that $L(A_{pq})$ = exactly the strings that take the PDA from state p with an empty stack to state q with an empty stack.
  - This has some of the flavor of the construction used to show that given a DFA, we could construct a regular expression for the machine's language.

- Here, things are a bit trickier because we have to account for the stack.

- To accomplish this, Sipser's proof include three types of rules in the grammar. The first collection of rules is probably the most important, but the description of the rules is a bit scary:

  For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $(r, t) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$, include the rule $A_{pq} \rightarrow aA_{rs}b$ in the set R for G.

- A more intuitive way to state this description is:

  If state p pushes t and goes to r on input x and s pops t and goes to q on input y include $A_{pq} \rightarrow xA_{rs}y$ in the set R for G.

- Given the machine we are working with, the description requires that we include the following rules in our grammar:
  - $A_{SF} \rightarrow \epsilon A_{AE}\epsilon$
  - $A_{AE} \rightarrow 1A_{AE}1$
  - $A_{CE} \rightarrow = A_{DD}\epsilon$
  - $A_{CE} \rightarrow 1A_{CE}1$
  - $A_{AC} \rightarrow +A_{BB}\epsilon$

  (Normally, the ϵ's include in these rules would be omitted, but we have included them to make it clear how each rule results from the process described for determining the rules to include.)

- It should already be clear that these rules might be useful pieces toward forming a complete grammar for L_{add}.

- The next requirement is to add ϵ-productions for the variables A_{pp} for every state p in the PDA. Clearly, on ϵ any PDA can "move" from any of its states back to that state without changing its stack.

- This rule leads us to add the productions:
  - $A_{SS} \rightarrow \epsilon$
  - $A_{AA} \rightarrow \epsilon$
  - $A_{BB} \rightarrow \epsilon$
  - $A_{CC} \rightarrow \epsilon$
  - $A_{DD} \rightarrow \epsilon$
  - $A_{EE} \rightarrow \epsilon$
  - $A_{FF} \rightarrow \epsilon$

- The main impact adding these rules has in this example, is that we can expand the epsilon productions "inline" to simplify the productions we derived from the first rule to obtain a version without any explicit or implicit ϵ's:
  - $A_{SF} \rightarrow A_{AE}$
  - $A_{AE} \rightarrow 1A_{AE}1$
  - $A_{CE} \rightarrow =$
  - $A_{CE} \rightarrow 1A_{CE}1$
  - $A_{AC} \rightarrow +$

- Finally, for any three states, p, q and r, it may be possible to get from p to q starting and ending with an empty stack by first going from p to r starting and ending with an empty stack and then going from r to q with an empty stack. To capture this, Sipser tells us to add rules of the form $A_{pq} \rightarrow A_{pr}A_{rq}$ for every such triple.

- Even for our simple machine with 7 states there would be $7^3 = 343$ such rules.

- For a construction algorithm in a proof, adding 343 (or 10 million) rules is no problem. Unfortunately, writing out all these rules tends to prevent mere mortals from getting any intuition about how the construction really works.

- Fortunately, there are some obvious situations in which rules of the form $A_{pq} \rightarrow A_{pr}A_{rq}$ that are certain to be useless[1] components of our grammar can be identified.
  - In our machine, all transitions are from left to right, so non-terminals of the form A_{pq} where p appears to the right of q in our state diagram are clearly useless.

[1] Technically, a non-terminal in a grammar is considered *useless* if it is impossible to derive any string of terminals from the non-terminal. Any rule in a grammar that contains a useless non-terminal on its right hand side must be useless in the sense that it cannot appear in any derivation that produces a sentence of the language of the grammar.

- In any machine, if all transitions out of a given state immediately pop a stack symbol, then there can be no inputs sequences that would take the machine from that state *starting with an empty stack* to any other state ending with an empty stack. Therefore, if p is such a state, for all q the non-terminals A_{pq} must be useless.
- In any machine, if all transitions into a given state q push a stack symbol, all variables of the form A_{pq} must be useless.

- As a result, there is really only one production of interest added to our grammar by this last rule

$$A_{AE} \rightarrow A_{AC}A_{CE}$$

- Putting this all together, eliminating the unnecessary start symbol A_{SF} by replacing it with A_{AE}, and reordering our rules to make things a bit clearer we get:
  - $A_{AE} \rightarrow 1A_{AE}1$
  - $A_{AE} \rightarrow +A_{CE}$
  - $A_{CE} \rightarrow 1A_{CE}1$
  - $A_{CE} \rightarrow =$

  It should be clear that this is indeed a grammar for the language accepted by our PDA!

2. Again, what we have done is intended to give you an intuitive appreciation of the constructions presented in Sipser. You should reread the details in Sipser carefully with the hope that this intuition will make the argument clearer.