Announcements

1. Our midterm will be a 72-hour take-home open-book/notes. You can take it any time between 10/22 (today) and 10/28 (Saturday) but you have to pick it up by 10/26 (Friday). You may spend no more than 7 hours working on the exam questions. If you choose to spread these 7 hours over more than 7 hours, you must work on one question at a time in such a way that the total time spent is less than 7 hours.

PDAs and CFLs

1. One of the main results we can show about context-free languages is that 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. Last class, I presented the basics of the construction that shows how to build a PDA for a language given a context-free grammar for the same language. I will review this construction in class, but our main goal today will be to consider the other direction of this proof ...

4. Now, we need to show that PDAs can only accept context-free languages. To do this, 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 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:
      \[ L_{\text{add}} = \{1^i + 1^j = 1^{i+j} \mid i, j \geq 1 \} \]

    • 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:

    • 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 \( a, b \in \Sigma \), 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:

  - \( A_{SF} \rightarrow \epsilon A_{AE} \)
  - \( A_{AE} \rightarrow 1A_{AE}1 \)
  - \( A_{CE} \rightarrow \epsilon \)
  - \( A_{CE} \rightarrow 1A_{CE}1 \)
  - \( A_{AC} \rightarrow + \)

(Normally, the \( \epsilon \)'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 \( \epsilon \)-productions for the variables \( A_{pp} \) for every state \( p \) in the PDA. Clearly, on \( \epsilon \) any PDA can “move” from any of its states back to that state without changing its stack. So, if the stack starts empty it will end empty.

* 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 \( \epsilon \)'s:

  - \( A_{SF} \rightarrow A_{AE} \)
  - \( A_{AE} \rightarrow 1A_{AE}1 \)
  - \( A_{CE} \rightarrow \epsilon \)
  - \( A_{CE} \rightarrow 1A_{CE}1 \)
  - \( A_{AC} \rightarrow + \)

- 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 components of our grammar can be identified.

\(^1\) Technically, a non-terminal in a grammar is considered useless if it is impossible
– 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.
– 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:
  
  - \( A_{AE} \rightarrow 1A_{AE}1 \)
  - \( A_{AE} \rightarrow A_{AC}A_{CE} \)
  - \( A_{AC} \rightarrow + \)
  - \( A_{CE} \rightarrow 1A_{CE}1 \)
  - \( A_{CE} \rightarrow = \)

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

5. 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.

---

### Chomsky Normal Form

1. Have you ever heard of Noam Chomsky?
2. He is best know these days for his political activism.
3. Before his political career, Chomsky played a significant role in the development of formal grammars (of which CFGs are one example).
4. One of his contributions was describing a restricted form of CFG that was powerful enough to describe any CFL. In particular, we say that a grammar is in Chomsky Normal Form, if all productions in the grammar are of the forms:
   - \( A \rightarrow BC \), where \( A, B, \) and \( C \) are all non-terminals,
   - \( A \rightarrow t \) for some non-terminal \( A \) and terminal \( t \), or
   - \( S \rightarrow \epsilon \) where \( S \) is the start symbol.
5. At some level, this is not all that surprising. In particular, if you take a “typical” production like

\[
S \rightarrow if(B) S
\]

it can be replace by a collection of productions of the second type to handle the terminals:

- \( T_1 \rightarrow i \)
- \( T_2 \rightarrow f \)
- \( T_3 \rightarrow ( \)
- \( T_4 \rightarrow ) \)

It is obvious that this grammar is in Chomsky Normal Form.

---

...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.
6. The tricky part is finding a replacement for productions like \( T \rightarrow \epsilon \) and \( T \rightarrow N \).

- The basic idea is that if such rules are present, we will make multiple copies of other productions that reference the variable on the left side (\( T \) in our examples) with multiple productions that reflect the option of using these simple rules.
- For example, if our grammar included the rules

\[
S \rightarrow if(B) S \\
S \rightarrow if(B) S \; else \; S \\
S \rightarrow \epsilon \\
S \rightarrow T
\]

- We would begin by adding a copy of the first rule that reflected the possibility of using the \( \epsilon \)-rule:

\[
S \rightarrow if(B)
\]

- This is a bit trickier for the second rule because we have to account for the fact that either one, both, or neither of the instance of \( S \) in the rule might be expanded using the \( \epsilon \)-rule:

\[
S \rightarrow if(B) \; else \; S \\
S \rightarrow if(B) \; S \; else \\
S \rightarrow if(B) \; else
\]

- Similar productions would be added to take the place of the \( S \rightarrow N \) rule:

\[
S \rightarrow if(B) \; N \\
S \rightarrow if(B) \; N \; else \; S \\
S \rightarrow if(B) \; S \; else \; N \\
S \rightarrow if(B) \; N \; else \; N
\]

- After all these additions are made, the offending rules can be removed.

7. The existence of Chomsky Normal Form means that if we are given a context-free grammar and a string we can determine whether the string belongs to the language of the grammar.

- You may already think this is obvious because we could just build a PDA from the grammar for the language, but PDAs are non-deterministic and simulating the non-determinism to see if there is any combination of production choices that leads to the desired string might go on forever if no such choices existed.
- Chomsky Normal form means we can find a grammar for the language with the property that the sentential forms we generate get larger and larger as we do more production steps.
- This means we can cut off the non-deterministic search once we reach any sentential form that is longer than the desired sentence and backtrack to explore other short derivations.