CS 536 Review

Important information for the midterm

- The exam will begin at 9:30 Sharp, please plan to be here at least 15 min early.
- The exam will finish at 10:45 Sharp as Prof Gleicher is in the room by 10:50
- One 8.5 x 11 inch page of handwritten notes is allowed. Apart from this, no books, sheets, electronic devices, or help from neighbors allowed during the exams.
- You **MUST** bring your student ID to identify yourself.
- **PROJECT 3:** There is an updated deadline for P3 if you need it, so you aren’t forced between completing p3 or studying. Check the website for info

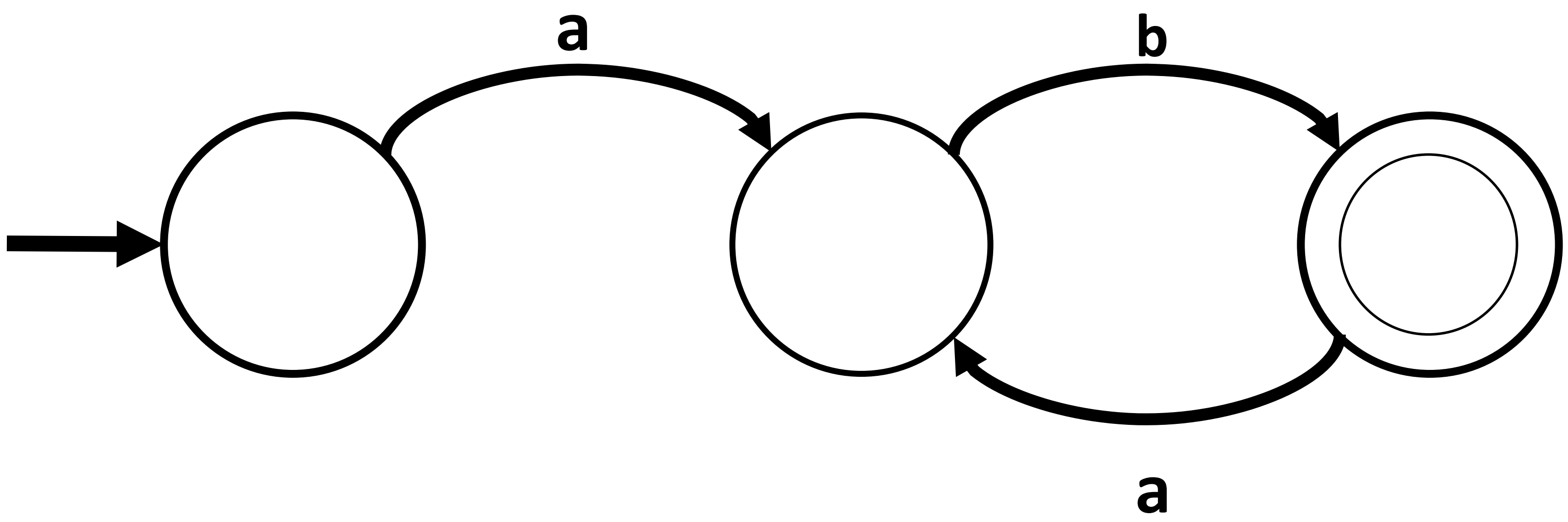
Finite State Automata

- A DFA can be defined by a quintuple (Q, Σ, δ, s, F) where
  - Q is a finite, non empty set of states.
  - Σ is the input alphabet.
  - δ is the transition function δ: Q x Σ -> Q
  - a ϵ Q is the initial state.
  - F c Q is a set of accepting states. Note this need not be non-empty!
- δ need can be a partial function, but δ as a total function is required for some algorithms in their default form. (See Project 2)

Finite State Automata Continued

- NFAs are similar to DFAs in that they are a quintuple (Q, Σ, δ, a, F) except δ: Q x (Σ U {ε}) -> P(Q) where P(Q) is the power set of Q
- Despite the added flexibility of epsilon transitions and non-determinism, they are no more powerful than DFAs!
- This symmetry is broken when moving to more expressive languages and complex automata
- Is the language $\{\ (ab)^n \mid n \geq 1\}$ a regular language?

Yes it is!



Another Example

- Is the language $\{ a^n b^n \mid n \geq 1\}$ a regular language?
- It is not! DFAs have no way to "store" information such as the number of a's written.
- If you don't believe me, I challenge you to come up with a DFA that does accept the above language.
- This language is context free however.

Context Free Grammars

- CFGs are defined to be 4 tuple G=(V, Σ, R,S) where:
  - V is a finite set where each v ϵ V is a *Variable*. *Variables* are non terminal characters than define a sublanguage of G.
  - Σ is the set of *Terminal Characters* of G, which are disjoint from V. This is the actual content of the grammar.
  - R is a relation (V, (V U Σ)*) known as the *Production Rules* of G
  - S is the start variable. Is analogous to S in DFA's
- CFGs are more sophisticated than Regular Languages as
  - Tokens become grammatical phrases
  - Structure in the program can be accounted for

Grammar for $\{ a^n b^n \mid n \geq 1\}$

- G = (V, Σ, R, S) where:
  - V = {T}
  - Σ = {“a”, “b”}
  - R = (T, aTb | ab). When written as a production rule: T -> aTb | ab
  - S = T

Parse Trees and Derivations

- Derivation: Starting with a beginning Nonterminal, expand out until there are no Nonterminals remaining.
- Examples:
  - T -> ab : T derives ab in one step
  - T -> aTb -> aabb : T derives aabb in two steps.
- A Parse Tree is graphical representation of the derivation.
- Example for the two strings

Useful and Useless Non Terminals

- A useless non terminal is one that can't be used in any derivations of the grammar.
  - We can find out how to eliminate these useless variables
- Generating: A nonterminal can derive a string
  - X is generating iff X -> w where w is all terminals or contains variables previously marked generating
- Reachable: The start symbol can derive a string that contains this nonterminal
  - Z is reachable from Y iff Y is reachable from X.
- We find the non generating nonterminals first and eliminate them, and then find non reachable nonterminals and eliminate them.

Example

- T -> aTb | ab | S
- S -> E | “eps”
- E-> aE
- D -> c
- Generating: E, T, S so eliminate E, E->aE
- Reachable: T so eliminate D

Syntax directed translation

- Consider the following grammar (non terminals upper case)
  - S -> L dot R | L
  - L -> B | L B
  - R -> B | B R
  - B -> 0 | 1
- What is an example string in this grammar?
  - 101.101
- So this is the grammar for binary decimal strings.
- Lets try develop a set of translations that will give us the value in binary.

Syntax Directed Translation

- Our basic scheme will be to start at the root, build down to leaves, and then compute the decimal value in reverse.
  - S -> L dot R : L.pos = R.pos = -1; S.trans = L.trans + R.trans
  - S -> L : L.pos = 0; S.trans = L.trans
  - L -> B : B.pos = L.pos ; L.trans = B.trans
  - L -> L B : L1.pos = L.pos +1; B.pos = L.pos; L.trans = L1.trans+B.trans
  - R -> B : B.pos = R.pos; R.trans = B.trans
  - R -> B R : R1.pos = R.pos-1; B.pos = R.pos; R.trans = R1.trans+B.trans
  - B -> 0 : B.trans = 0;
  - B -> 1 : B.trans = 1*2^(B.pos);
- Lets do out the string 101.101 on the board.

Lets discuss in more detail how a computer parses a CFG.

- You can always use the CYK Algorithm. It is a bottom up parser with an acceptable runtime O(n^3) and will work for any CFG in Chomsky Normal Form (CNF).
- To do this, you need to do three things:
  - Eliminate eps rules
  - Eliminate Unit rules
  - Fix Remaining Rules so that all rules have either a single terminal or exactly two nonterminals on the right.
- After this conversion, the algorithm works by considering every possible subsequence of increasing length to see if is a valid production.

Parsing Continued

- We can do better if our grammar is LL(1).
- LL(1) grammars are top down parsers than only require one symbol look ahead.
  - Thus at every step, we need to have a definite way to get from one state to the next.
- The main Idea
  - Keep track of : the scanned tokens, the stack contents, and the leaves of the current parse tree.
  - We need to use a parse or selector table to do this.
- Push EOF, Push start symbol, Expand via Selector Table and Scan when appropriate.
  - Expansion is guaranteed to be unique so there is no ambiguity.

LL(1) Grammars

- How do we know if we have a LL(1) grammar?
- We need to actually try to build the selector table.
  - If the selector table only allows one production per (symbol, state) pair then we have it!
- Unfortunately, this will always fail unless we make sure our grammar doesn’t have any left recursion or isn’t left factorable.

Remove Left Recursion

- Left Recursion: A +-> A string
  - After a sequence of derivations you end up with A going to A and then another string.
  - Immediate left recursion is a problem!
  - You don’t know if you should choose the first production or the second production without looking ahead.
- If A -> A a | b then change to
  - A -> b A’
  - A’-> a A’ | eps

Example

- Consider our previous example of binary decimal strings.
  - We had the production: L -> B | L B
  - This is immediately left recursive!
- Lets go ahead and change this.
  - L -> B L’
  - L’ -> B L’ | eps
- Does the syntax directed translation still work?
  - L -> B L’ : B.pos = L’.pos +1; L.trans = B.trans + L’.trans
  - L’ -> B L’ : L’.pos = B.pos = L1’.pos+1 ; L’.trans = B.trans+L’.trans
  - L’ -> eps : L’.pos = 0; L’.trans = 0

Left Factoring

- You need to left factor if any production you write leads to a common prefix.
  - Say A -> string string1 | string string2
  - This is not left factored because of the common prefix
  - You don’t know which production to choose based of the current symbol you see without looking ahead.
- We change this to
  - A -> string A’
  - A’ -> string1 | string2

Example

- Lets looks at our previous example again.
  - We have a production: R -> B | B R
  - This is not left factored
- So We'll insert another non terminal to fix this problem
  - R -> B R'
  - R' -> R | eps
- We'll have to change around the translations again
  - R -> B R' : R'.pos = R.pos -1; B.pos = R.pos; R.trans = B.trans + R.trans
  - R' -> R : R.pos = R'.pos; R'.trans = R.trans
  - R' -> eps : R.trans = 0;

Left Recursive and Left Factoring

- The situation is a little more complicated if you have a more than two productions with a common prefix or more than two cases of immediate left factoring.
- The type of process you follow is exactly the same however.

First and Follow Sets

- In order to truly decide if a grammar is LL(1) we actually have to build a selector table for it.
- The previous slides talked about sufficient conditions for a grammar to not be LL(1), they were not necessary conditions.
- Lets try to compute the first and follow set for our example of binary decimal strings.

Updated Grammar and First Set

- S -> L dot R .
- L -> B L'
- L' -> B L'
- L' -> eps
- R -> B R'
- R' -> R
- R' -> eps
- B -> 0
- B -> 1

- So to construct the FIRST set, lets consider what terminal could appear first for each nonterminal
- S.First = L.First = B.First = {0,1}
- L’.First = B.First = {0,1} U {eps}
- R.First = B.First = {0,1}
- R’.First = R.First = {0,1} U {eps}
- Nothing too surprising here.

Updated Grammar and Follow Set

- S -> L dot R .
- L -> B L'
- L' -> B L'
- L' -> eps
- R -> B R'
- R' -> R
- R' -> eps
- B -> 0
- B -> 1

- For the Follow Sets
- S.follow = { $} as S doesn’t appear in the RHS of **any** production
- L.follow = {dot} as L only appears on the LHS of the first production and it isn’t the last symbol in the production.
- L’.follow = {dot} as we add L.follow to L’.follow
- R.follow = {$} as we add S.follow to R.follow
- R.follow = {$} as we add R.follow to R’.follow
- B.follow = {0,1,$,dot} as we add L’.follow to B.follow, R’.follow to B.follow. Finally add L’.First to B.follow as L’ -> eps

Selector Table

	dot	0	1	$
S		S → L dot R	S → L dot R	
L		L → B L′	L → B L′	
L′	L′ → ε	L′ → B L′	L′ → B L′	
R		R → B R′	R → B R′	
R′		R′ → R	R′ → R	R′ → ε
B		B → 0	B → 1	

Updated Grammar and Action Numbers

• S -> L dot R	(1): S.trans = L.trans + R.trans, R.pos = -1
• L -> B L'	(2): L.trans = B.trans + L’.trans, B.pos = L’.pos+1
• L' -> B L'	(3): L’.trans = B.trans + L’.trans, B.pos = L’.pos = L1’.pos +1
• L' -> eps	(4): L’.trans = 0
• R -> B R'	(5): R.trans = B.trans + L’.trans, R’.pos = B.pos = R.pos-1
• R' -> R	(6): R’.trans = R.trans, R.pos = R’.pos
• R' -> eps	(7): R.trans = 0
• B -> 0	(8): B.trans = 0
• B -> 1	(9): B.trans = 2^(B.pos)

Example: Derive 10.0

Input seen so Far	Stack	Action
eps	L dot R EOF	Pop, push B L’
Eps	B L’ dot R EOF	Pop, push 1
Eps	1 L’ dot R EOF	Pop, scan
1	L’ dot R EOF	Pop, push B L’
1	B L’ Dot R EOF	Pop, push 0
1	0 L’ Dot R EOF	Pop, scan
10	L’ Dot R EOF	Pop, scan
10.	R EOF	Pop, push B R’
10.	B R’ EOF	Pop, push 0
10.	0 R’ EOF	Pop, scan
10.0	R’ EOF	Pop, scan

Cites

- SDT Example: http://www.isi.edu/~pedro/Teaching/CSCI565-Spring15/Practice/SDT-Sample.pdf