CMSC 330

Midterm Review

Administrative

Administrative

Midterm is Thursday

Project 3 is due Friday

Quiz 3 on July 6th

Context Free Grammars

Basics

Basics

A way of describing sets of strings

CFGs subsume REs, DFAs, and NFAs

CFGs are often used as the basis of parsers

Generating Strings

A grammar “generates” strings via rewriting

Example

    S -> 0S | 01 | ε

Generate the string 011

    S ->

Generating Strings

A grammar “generates” strings via rewriting

Example

    S -> 0S | 1S | ε

Generate the string 011

    S -> 0S -> 01S -> 011S -> 011

Accepting Strings

Algorithm - is s in the language?

    Find a rewriting of s starting from G’s start symbol

A sequence of rewrites is a derivation (or parse)

Discovering the derivation is called parsing

Creating Grammars

Write a grammar for $a^n b^n$

S ->

Write a grammar for $a^n b^n$

```
S -> aSb | ε
```

Write a grammar for $a^i b^j a^k$ where $k = i + j$ and $i,j,k \geq 0$

S ->

Wire a grammar for $a^i b^j a^k$ where k = i + j and i,j,k ≥ 0

———

```
S -> aSa | T

T -> bTa | ε
```

Ambiguity

Ambiguity

A grammar is ambiguous if there are multiple leftmost derivations of the same string

Give a leftmost derivation for (()())

S -> SS | (S) | ε

S -> (S) -> (SS) -> ((S)S) -> (()S) -> (()(S)) -> (()())

Prove that the grammar is ambiguous

Prove that the grammar is ambiguous by giving 2 different leftmost derivations of the same string

S -> aS | Sb | SS | ab

Prove that the grammar is ambiguous

Prove that the grammar is ambiguous by giving 2 different leftmost derivations of the same string

S -> aS | Sb | SS | ab

S -> aS -> aSb -> aabb

S -> Sb -> aSb -> aabb

Parsing

Provide a leftmost derivation for a * a + b

———

```
S -> S + E | E

E -> E * L | L

L -> a | b
```

S ->

Provide a leftmost derivation for a * a + b

S -> S + E | E

E -> E * L | L

L -> a | b

S -> S + E -> E + E -> E * L + E -> L * L + E -> a * L + E -> a * a + E -> a * a + L -> a * a + b

Provide the Parse Tree for the derivation

———

S -> S + E -> E + E

-> E * L + E -> L * L + E

-> a * L + E -> a * a + E

-> a * a + L -> a * a + b

Provide the Parse Tree for the derivation

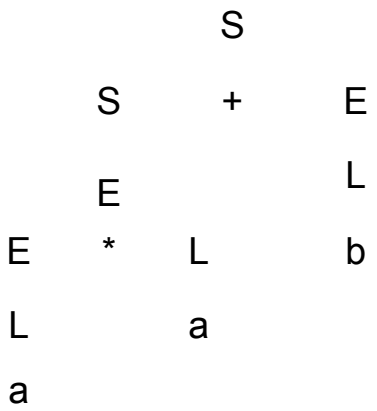
———

S -> S + E -> E + E

-> E * L + E -> L * L + E

-> a * L + E -> a * a + E

-> a * a + L -> a * a + b



Provide the AST

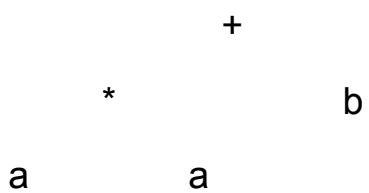
———

S -> S + E -> E + E

-> E * L + E -> L * L + E

-> a * L + E -> a * a + E

-> a * a + L -> a * a + b



Associativity

Associativity

Left-recursive = Left-associative

Right-recursive = right-associative

Make the Grammar Right Associative

———

S -> S + E | E

E -> E * L | L

L -> a | b

S ->

Make the Grammar Right Associative

———

```
S -> S + E | E

E -> E * L | L

L -> a | b
```

```
S -> E + S | E

E -> L * E | L

L -> a | b
```

Precedence

Precedence

Close to start symbol = lower precedence

Further from start symbol = higher precedence

Give + Higher Precedence Than *

```
S -> S + E | E

E -> E * L | L

L -> a | b
```

```
S -> S * E | E

E -> E + L | L

L -> a | b
```

First Sets

Provide The First Set For Each Production Rule

———

S -> aAB | Bc

A -> dB | Ac

B -> Cg

C -> f | d | ε

FIRST(S) =

FIRST(A) =

FIRST(B) =

FIRST(C) =

Provide The First Set For Each Production Rule

S -> aAB | Bc	FIRST(S) = {a, f, d, g}
A -> dB | Ac	FIRST(A) = {d}
B -> Cg	FIRST(B) = {f, d, g}
C -> f | d | ε	FIRST(C) = {f, d, ε}

Ruby Scan Method

Ruby Scan Method

Scan returns everything that the regex matches

Scan returns as a 2D array with each match as an array of all the capture groups

What is the output?

———

```
a = “CMSC 330 CMSC 351”

b = a.scan(/[A-Z]+/)

b
```

What is the output?

```
a = “CMSC 330 CMSC 351”

b = a.scan(/[A-Z]+/)

b
```

Output

```
b = [“CMSC”, “CMSC”]
```

What is the output

———

```
a = “CMSC 330 CMSC 351”

b = a.scan(/[0-9]+ [A-Z]+/)

b
```

What is the output

```
---
a = “CMSC 330 CMSC 351”
b = a.scan(/[0-9]+ [A-Z]+/)
b


Output
b = [“330 CMSC”]
```

What is the Output?

```
s = “To be, or not to be!”

a = s.scan(/(\S+) (\S+)/)

a.inspect
```

What is the Output?

```
s = “To be, or not to be!”

a = s.scan(/(\S+) (\S+)/)

a.inspect
```

Output

```
a = [["To", "be,"], ["or", "not"], ["to", "be!"]]
```

Closures

Closures

In OCaml, when you create a function, you create a “closure”

Function code associated with binding for variables

Closures

```
let f x y = x + y;;
```

When first created, this is the code with no bindings

Arguments can be applied one at a time

```
let g = f 6;;
```

This assigns g to be closure defined by f, with x=6

What would g 7 be? What would f 7 be?

Scoping

Scoping

Defined variables can be associated with a closure

Static Scoping

```
let x = 5;;

let f y = x + y;; (* f is a closure where x is bound to 5 *)

let x = 2;;

f 5;; (* What is the output? *)
```

Static Scoping

```
let x = 5;;

let f y = x + y;; (* f is a closure where x is bound to 5 *)

let x = 2;;

f 5;; (* What is the output? *)
```

10 is the output. This is static scoping.

Dynamic Scoping

```
let x = 5;;

let f y = x + y;; (* f is a closure where x is bound to 5 *)

let x = 2;;

f 5;; (* What is the output? *)
```

Dynamic Scoping

```
let x = 5;;

let f y = x + y;; (* f is a closure where x is bound to 5 *)

let x = 2;;

f 5;; (* What is the output? *)
```

7 is the output. Variables use the most recent declaration.

Midterm

Topics

PL Concepts

Ruby

OCaml

NFA/DFA