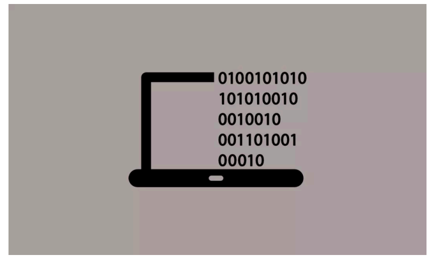
CECS 444 Compiler Constructions

Seminar Notes

August 28, 2018



Syllabus

Things to cover:

- Treewalking (binary)

Textbook:

Fisher, Cytron, Leblanc

- Crafting a Compiler (2009 ~720pg)

Grading:

Cumulative Exams

20% Exams I

20% Exams II

33% Final

20% Projects (Will build on each other)

7% Quiz, Paper, Participation

MGR Types: (Manager Types)

Good: 10% - Super people

Bad: 80% - Need people to do the job

- They buy programmers “By the Yard”

Ugly: 10% - Backstab

Mini- SWE (Software Engineering) Rules

** Reasonable Person STD (Standard)

- Due Diligence (Everybody has their own view)

  - Pace yourself

- AIO: (Adapt, Improvise, and Overcome)

** “Smart” Person STD

- Always be ready to show your work (Show your progress)

★ Most Important Things in SW(Software): **MORALE**

Rules:

0. Get to working Software Fast!

(Go ugly early)

Why!

1. You can see it work

* 2. Users can see it & tell you it sucks

- Get users feedback faster

(MVP = Minimum Viable Product)

1. Never Pre-Optimize (Usually 1% of code is too slow)

- Change this 1% and program increases more in speed

*** Optimize ONLY when proven needed

2. No “BUG HUNTS”

I. Compile-Time Errors $\leq$ 5 mins to fix

II. Usually 90% of DEV Time spend on Run-Time Bugs

- How to get rid of it?

• Force all bugs into small box (look there!)

★ Use “Add-A-Trick”

- Add 1-N Lines, Compile, then Test

3. EIO (Expected Input/Output)

*** Build Before Coding (Slice it into Itty-Bitty Stepping Stones)

- It focus design on what is important

*** Avoid “Gold-Platting”

Continued on August 30, but placed here since it continue ——>

- Making things look nice with nothing to functionality

4. Clean The Page. (~ 50 to n lines of code per page)

   - Usually one page for a Function so easy to read

August 30, 2018

Homework: Read Fischer

Chapter 1 Intro - 30pg

Chapter 2 Compiler Parts - 25pg

Chapter 3 Scanner/Lexer - 50pg

Mini Study Rules:

1. Textual Mean

   - Build/Use “Flash-Cards” (3x5)

2. Visual Memory

   IE: Charts, Graphs, etc

   - Draw it twice, looking

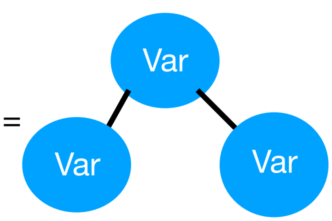
   - Draw it Blind

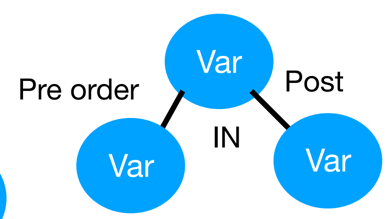
     • win 3x • include labels

TreeWalking:

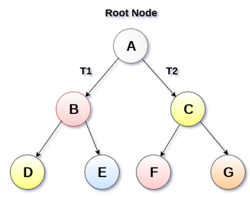
- Consist of:

  Left / Right / Lollypop =





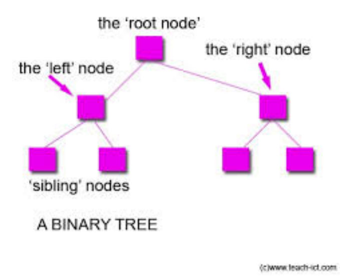

```
CLASS Node
{
    INT VAL;
    NODE LKid;
    NODE RKid;
}
```



Binary Tree

To Do For TreeWalking:
1. Header
2. Basic Step
   - Do manually
3. Left/Right Recur
4. Deal with Lollypop
5. Glue

```
Void printTree(NODE root)
{
        # Basic Step
        If (NULL == root)
        {
                RGT; #Abbr. for returning nothing
        }
        # Left Recur
        printTree(root.LKid);
        # Right Recur
        printTree(root.RKid);
        # Deal with LollyPOP
        System.out.println(root.VAL);
        # GLUE
        // None
}
```




```
Void countTree(NODE RP)
{
        # Basic Step
        If (NULL == root)
        {
                RGT; #Abbr. for returning nothing
        }
        # Left Recur
        INT Lx = countTree(RP.LKid);
        # Right Recur
        INT Rx = countTree(RP.RKid);
        # Deal with LollyPOP
```

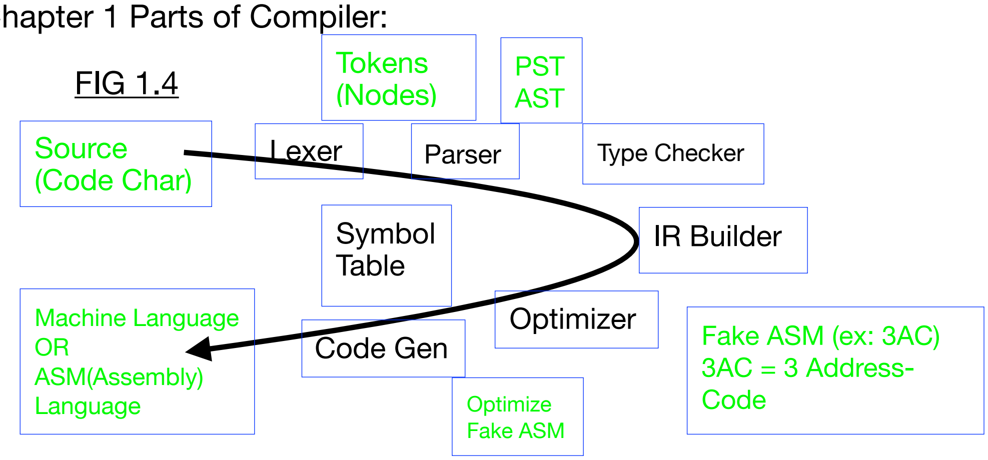
```
        Px = 1;
        # GLUE
        Return Lx + Rx + Px;
}

Void sumValTree(NODE RP)
{
        ….
        # Deal with LollyPOP
        Px = RP.VAL;
        …
}

Void sumValForKind(NODE RP, INT RK)
{
        ….
        # Left Recur
        …. RK
        # Right Recur
        ….. RK
        # Deal with LollyPOP
        Px = (RK == RP.kind
                ? RP.VAL
                : θ );
}
```

Chapter 1 Parts of Compiler:

FIG 1.4



Lexer = Lexical Analysis

- Lang. REGEXES

Parser = Syntactic Analysis

- CFG (Context Free Grammar) Rules

Type Checker & IR Builder = Semantic Analysis (Good meaning)

- IR Builder (Intermediate Representation Builder)
    - In each stages, since they are not source or final, they are IR
- AST + Decoration

Optimizer

Code Generation = Final representation (Emiter Phase)

- “Emits” Machine/ASM/Byte Code
    - Bytecode usually mean for JAVA since it is old
        - For interpreter/VM Architecture
- Machine Architecture Description

Symbol Table:

- Contains all user-define names (names = symbols)
- Are builded into debugger

Front End:

- Between beginning to Syntactics Analysis

Back End:

- After Syntactics Analysis to end

PST (Parse Tree): Convert to AST (through Parser)

AST (Abstract Syntax Tree): In one simple operation from PST —> AST

September 4, 2018

Homework: Read Fisher

Chapter 3 Scanner (Lexer)

Chapter 3.2 REGEX (Regular Expression)

- Regular Lang

1. LITERALS: “3”, “Hi”

2. Wildcard Character “operator”.

   - Uses Period

   IE: c.t

   All matches of period wildcard—> {Cat, Cbt, C7t, C$t, c t, c.t,…}

3. Escape (De-Opify)

   - Uses Backslash

   IE: c\.t —> {c.t}

   IE: c\\t —> {c\t}

4. Optional

   - Uses Question Mark

   IE: Ca?t —> {ct, cat}

5. Grouping

   - Uses Parenthesis

   IE: C(a)t —> {Cat}

   IE: (Ca)?t —> {t, Cat}

6. Zero or More (AKA: Kleene Star)

   - Uses Astris

   IE: (Ca)*t —> {t, Cat, CaCat,…}

   IE: 123*4 —> {124, 1234, 12334, …}

7. 1 or more (AKA: Kleene Plus, Positive Closure)

- Uses Plus

IE: 123+4 —> {1234, 12334, …}

IE: 12.+4 —> {12a4, 12b4, …, 12ab@724}

- Give me one or more "wildcard char op"

8. Any 1 Char: From the set

- Uses Brackets (Anything inside the bracket is auto escape)

IE: [BFC]at —> {Bat, Fat, Cat}

IE: [BFC]?at —> {at, Bat, Fat, Cat}

*9. Choose Sequence of (AKA "OR")

- Uses Vertical Stroke

IE: C(a|o+|u)LL —> {CaLL, CoLL, CuLL, CooLL, CoooLL}

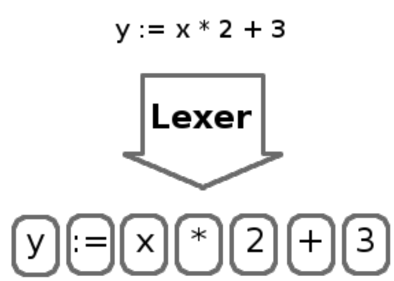
- a | o+ | u = a or o+ or u

10. In a Char Subset: A Range of..

- Uses Hyphen

IE: a[A-D]z —> {aAz, aBz, aCz, aDz}

IE: [_A-Za-z0-9]



Project 1 Lexer

Digits = [0.9]+

Leading Underscore = '_' | [A-Za-z]

FSM = Finite State Machine (AKA: DFA)

DFA = Deterministic(no choice) Finite Automaton

• States

2 Types:

SS = Start State

AS = Accept State(s) AKA Recognized

- Found a Match (Doesn't mean stop)

- Events (Words Event, Letters Event)
- Links/Moves (Labeled with Events)

    - moving from one state to another based on events

Input Event Sequence leading from SS to some AS

- A word/sentence in the “Language” of the Regex

IE: Regex, R = C(a | o | u)t

Lang, L(R) = {Cat, Cot, Cut}

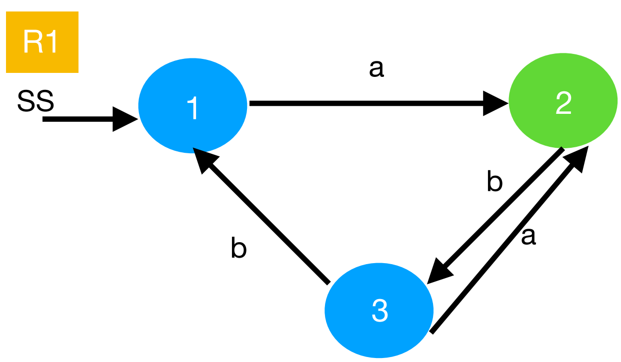
IE: R’ = See the (cat|dog|bear)\?

L(R’) = {“See the cat?”, “See the dog?”, “See the bear?”}

DFA Format/“Coding”

State transition

- Table/(Matrix)
- Diagram/(Graph)



- Circe per state
- Link per transition
    - One Table Cell

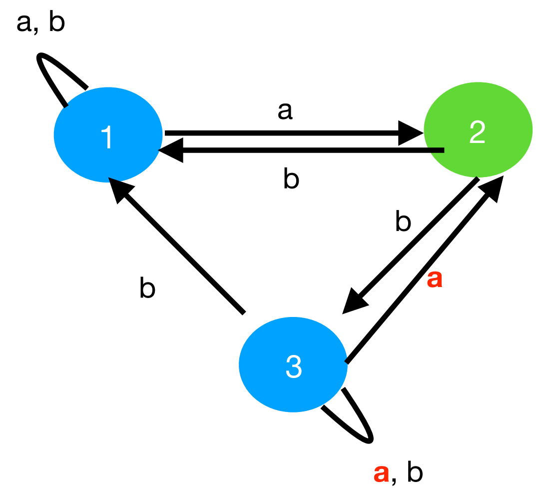
R1	a	b
1	2	-
(2)	-	3
3	2	1

= AS

- Row per state
- Column per event

- Empty Cell = no possible match

NFA



- Because of a, **we have choice so NFA**

R2	a	b
1	1,2	1
(2)	-	1,3
3	2,3	1,3

- NFA Choice, 2 Ways:
    1. Choice of Moves (From State, on Event)
    2. “Epsilon Move” - Greek E (ϵ) for empty

    *(FISCHER uses Lambda, λ)

★ Convert NFA to DFA:

“Path Graph”: BAABAB$ ($ = end of input)

SS

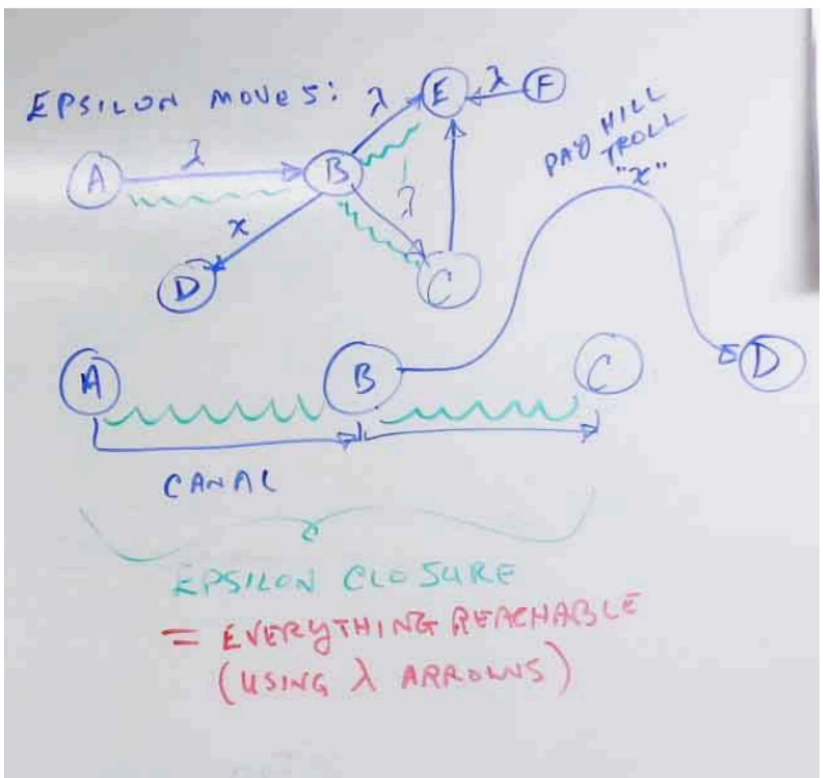
	B	A	A	B	A	B	
1	1	1	1	1	1	1	1
(2)			2	2		2	
3					3	3	3

To DFA	a	b
{1}	{1,2}	{1}
{1,2}	{1,2}	{1,3}
{1,3}	{1,2,3}	{1,3}
{1,2,3}	{1,2,3}	{1,3}
{2,3}	{2,3}	

Q: How many DFA States from “N” NFA States max?

Ans: $2^n - 1$

Epsilon Moves:

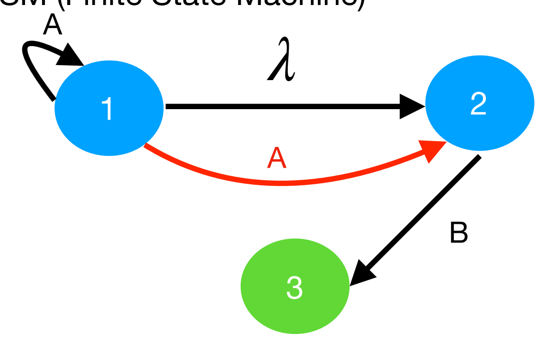


September 6, 2018

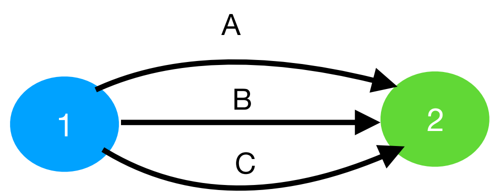
Why NFA Bother?

Regex to FSM (Finite State Machine)

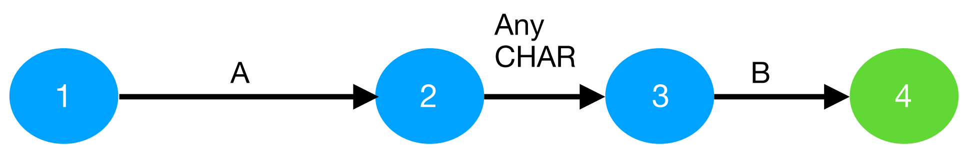
(A * B)



(A | B | C)

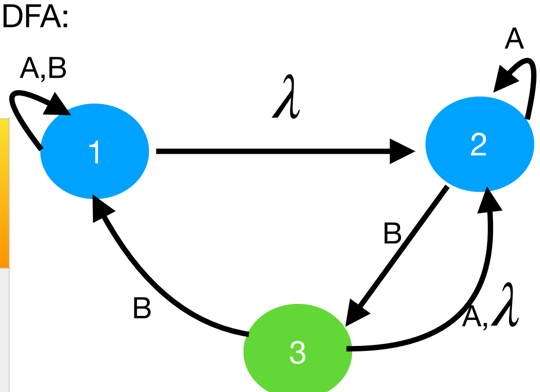


(A . B)



- These Scenario can be combined: (A . B)(A | B | C)

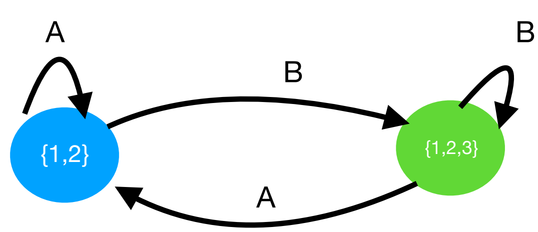
More NFA to DFA:



NFA	A	B	λ
1	1	1	2
2	2	3	-
3	2	1	2

DFA (USES EPSILON)	A	B
{1;2}	{1,2;}	{1,3;2}
{1,2,3}	{1,2;}	{1,3;2}

NEW SET



after semicolon is the lamda state

- The red semi colon is there even though on cell {1;2}A the lamda doesn't go anywhere beside 1 and 2
- On cell {1;2}B , we got a new set, {1,3,2} so we start a new row for {1,2,3}.
  - After that row, no more new set, so we stop
- Since 3 is the accept state, anything with a 3 will be green, aka accepted

READ FISHER

Chapter 4

Chapter 5.1 - 5.4

September 11, 2018

Read Chapter 3:

Skip 3.5 Lex

Read Chapter 4:

then 4.4

Previous: Regular Langs

- REGEX

CFG: Context-Free Grammar

CFG Rules

- LHS = RHS (left hand side = right hand side)

1 Symbol = Sequence 0 or more Symbols

::=

<— = LHS "expands to" RHS in A "Derivation"

GMR "G":

IE: S = X | Y   is an example of Combo - Rule

- Rule 1: S = X   is Simple - Rule

  OR

- Rule 2: S = Y

History: High-Level Langs

1957: FORTRAN (FTN)

1958: LISP (A.I)

1960: Cobol

1960: Algol 60

- Algol 68 (Euro ver.)

Popular Langs: (TIOBE website)

Market Shares:

Java - 16%
C - 14%
C++ - 8%
Python - 5%
C# - 4%
VB - 4%
PHP
Javascript
SQL
RUBY

IE: X = a | yxy

“a” = lower case
“yxy” = symbol sequence

- RHS = Terminal Symbol
    1. Can’t expand
    2. Not on LHS
- LHS = Non-Terminal Symbol

Lang “G”: All “sentence” described by GMR “G”

Q: is “bab” in L(G)?

Try to derive from Starting Symbol

A: * We can only “—>” into something that is rule in GMR “G”

```
S      // S -> X
X      // X -> yxy
yxy    // second y —> b
               - Right-Most Derivation
yxb    // X —> a
yab    // Y —> b
bab    // WIN
```

Parse Tree (PST):

```
          S
          |
          X
    |     |     |
    y     x     y
    |     |     |
    b     a     b
```

LR = Left-To-Right Scan & Right-Most Derivation

LL = Left-To-Right Scan & Left-Most Derivation

AST = Abstract-Syntax-Tree

IE: I See The Red Truck

S = Subj Verb Obj

Person = I | You | Fred | ….

```
             S

|            |            |

Subj         Verb         Obj

|            |            |   |   |

Man          SEE          Adj Adj Noun

|                         |    |    |

I                         the red truck
```

“GE”: Arith Expr GMR

E = E ‘+’ T | E ‘-‘ T | T

T = T ‘*’ F | T “÷” F | F

F = id | k | ‘(‘ E ‘)’

IE: B * B - 4 * A * C

PIC GOES HERE

“Recursive Descent” Parser

- Uses Depth First Search (DFS)

REG:

• Each Non-T gets a Function

• Each Rule gets a “Trial” in that function

- to match next input Sequence

```
BOOL E ( )   // Match the first E = … rule
{
    Input_Pos = Current;
    if (match (E) && match(‘+’) && match(’T’))
```

```
{
    return True;
}
Else ( E( ) && match('-') && T())
{
    return True;
}
Else ( T( ) ) return True;
return False;
```

Cons:

- Most Tries Fail
    - Very Slow
- If Error, tries everything first

LL Parse Machine

Machine	+	Table	+	Stack
4 Steps		Predict the rules		Partial Derivation

September 13, 2018

LL Parser:

Machine, Table, Stack

Table: (LHS, TERM) — LHS = top of stack TERM = Front of input

	Terminal 1	Terminal 2	…
LHS 1	###	-	
LHS 2	-	-	R#2
LHS 3	-	R#16	
…			

R#16 = Simple Rule (Not Combo)

$\boxed{N = N}$ A B C —- > Left Recursion

X = A B X C —-> Also recursion

Things we need to do:

1. Need simple rules
2. Get rid of left recursion

Combo —> Simple:

E = E + T | E - T

- Breaking it down : E = E + T

  E = E - T

A = B [C-DF] G

- Breaking it down: A = B C G

  A = B D G

  A = B F G

- The F is there as extra step

A = B ? C

- Breaking it Down: A = B C

  A = C

A = X * Y

- Breaking it down: A = N Y

  N = X N - Right Recursion

  - As much X as I want with N since *

  N = λ

A = B (X | Y) ? C

- Breaking it Down: A = B X C

  A = B Y C

A = B C

A = B X + Y or A = B X X * Y (C.F Ex: p 138 #3, 7)

- A = B X X * Y is similar to A = X * Y so just pretend B X is the front

LRE = Left Recursion Elimination

Direct LRE = $\boxed{\text{N = N}}$ - Directly calling itself, NOT foo calls bar and bar calls foo.

GE: * In GE, lowercase are TERM and uppercase are Non-TERM

E = E + T | E - T | T

T = T * F | T / F | F

F = i | K | ‘(‘ E ‘)’

- 9 Rules

3 Rules to do LRE for one NON-TERMINAL, N:

1. Add a <u>new</u> NON-TERMINAL (EX: “X”)

   - Which means X goes to nothing AKA X = λ

2. Append X to all N right hand sides.

3. Replace “Lefty Part” (N = N) with (X =)

   - so E = E —> X =

GE:

E = E + T | E - T | T

T = T * F | T / F | F

F = i | K | ‘(‘ E ‘)’

————————————————-

Q = λ	R = λ
$\boxed{\text{E = E}}$ + T Q	$\boxed{\text{T = T}}$ * F R
$\boxed{\text{E = E}}$ - T Q	$\boxed{\text{T = T}}$ / F R
E = T Q	T = F R
$\boxed{\text{Q = +}}$ T Q	$\boxed{\text{R = *}}$ F R
$\boxed{\text{Q = -}}$ T Q	$\boxed{\text{R = /}}$ F R

GE2:

$E = T\ Q$

$Q = +\ T\ Q \mid -\ T\ Q \mid \lambda$

$T = F\ R$

$R = *\ F\ R \mid /\ F\ R \mid \lambda$

F = i | K | ‘(‘ E ‘)’

L(GE2) == L(GE)

Next Chapter: 4.5.2

September 18, 2018

Disappearing Non-T’s

Direct Epsilon Rule: $N = \lambda$ <— 0th step

Indirect Epsilon Rule: N = AB | fg

$A = \lambda$ <— 0th step

B = C | h

$C = \lambda$ <— 0th step

Consult Further (CF) Fig 4.7, p 129

ALT: KEY: Bottom Up

ALGO - Direct Epsilon, First

1 Step Indirect, Next

2 Step …. , …..

ETC

Disappearing Set of Non-Ts:

"Dirty Bit"

Simple Rules:

(N, -) <— go to DS
(N, AB)
(N, fg) <- RHS, only Terms
(A, -) <— go to DS
(B,C)
(B,h) <- Take out
(C,-) <— go to DS

Step:

0 : (N, A, C) —> Disappearing Set

- Look at right hand side for N, A, C and eliminate
- Red = take out based on NAC
- for (B,C), right hand side became empty so add B

1: (N, A, C, B) —> DS

Point:

Add direct Epsilon Rule
For each indirect
Disappearing Non-T:
Simplify Table Build

Setup:

```
DS <— Empty
SR <— All Simple Rules

Dirty = True;
while (Dirty)
{ // One Step
        Dirty = False
        For each Pair in SR
                RHS = Pair.RHS
                For each N,A,C in DS
                        RHS -= N,A,C
                IF RHS is Empty
                {
                        DS += Pair.LHS
                        Dirty = True
                }
}
```

First Sets:

- Let RHS
- Look at L(RHS)   RHS = "Reduced Grammar"

    • Set of all Terms that start a sentence in L(RHS)

    $F(\lambda) = \{ \}$

    F(h) = { h }

    F(fg) = { f }

    F(C) = { }

        Union F(C=RHS)

    $F(AB) = A = F(\lambda) = \{ \}$

September 20, 2018

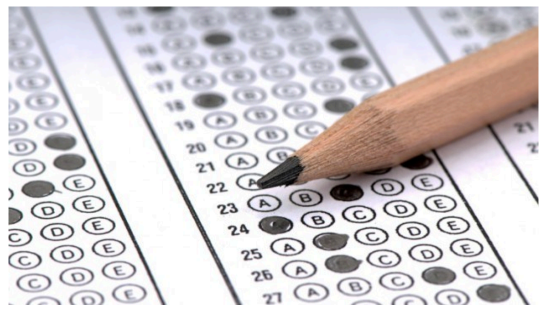
No Class

September 25, 2018

Exam Review:

NOX (not in exam): MINI - SWE Rules, PST

In exam:



- Treewalking
    - not just binary tree
- Spiderweb Diagram

    EX:

    - What is it?
    - What are the parts?
    - What are the links?
        - Leads to glossary terms
    - Front-End
    - Back-End

        What Parts?

Glossary Items:

- Bytecode
- (PST) = parse tree
    - non standard, but mirrors AST
- AST = abstract syntax tree

    AST vs PST

    - PST = every node and its kids are

- Non terminal = shows on left and right side
- Terminal doesn't expanse so it on only right
- AST = contains only Terminal symbol

Why AST?

- Twice as many nodes
- Messing with tree, mom and kids in PST will change

- LEXER
- Scanner
- Tokens
- ASM
  - Assembly language
- CFG
  - Context Free Grammar
- IR
  - Intermediate Representation
    - Source code is NOT IR
    - PST and AST are IR
    - Code coming out are NOT IR
- 3AC
  - 3 Address Code
- Semantic Analysis
  - Checking on meaning
  - Making well form sentences with reasonable meaning

    EX: The Bread ate the cat : Bad

    The Mouse ate the bread: good
- Syntax Analysis

- Check on Sentence Form by Grammar (part of speech)
- Glue words together to form proper sentence
- Well form coded
    - Ex: a statements, assignment, etc

      The Bread ate the cat : works here

        - Since Semantic will check meaning

- Lexical Analysis
    - Check if word exist in Dictionary

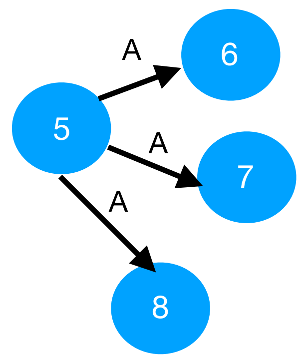
      IE: Is word in “Lexicon”

    - Glueing letters together to form word
    - Deals with stuff from below the level of words
        - EX: words are “+=“ which are tokens

- REGEX
    - What is t?
    - What are the Operations?
    - What are their actions?
    - What are legal tokens/words in L(R)
        - L(R) = Language of Regex
- DFA
    - What is it?
    - What Parts?
    - SS,(AS) = Start state, Accepted State
    - L(G) = language of grammar
    - State transition
        - TABLE (Matrix)
        - Diagram (Graph)

- Convert from Diagram to Table?

- NFA
    - What is it?
    - How to tell if NFA vs DFA?
        - Choice move?



        - Epsilon move λ
        - Extra Column in table for λ
- Convert NFA to DFA
    - Know how
        - Path graph + fill in
        - All subset of NFA states
        - Start with DFA SS. and use Reachability (row by row)
    - Epsilon Closure
        - CANALS
        - WETNESS
        - HILL TROLLS

Glossary Items:

- REGEX
- DFA
- FSM
- NFA
- EPS Closure

- Recognized
- Accepted
    - Recognized vs Accepted = SAME

- SS

- Why bother with NFA?
    - REGEX operations give NFA

      EX: A (AB | AC) D

- History
    - 1957 Fortran - Math Equation
    - 1958 LISP - Lambda Calc
    - 1960 ALGOL - Fortran-killer
    - 1960 COBOL - Don't need Programmers

- Arithmetic Expression Grammar
    - How to use it to build
        - A PST from an input Expression

    Recalls:

      PST mom + kids

      GMR: LHS + RHS

    Glossary:

    - LHS
    - RHS
        - L_1 = Left-To-Right Scan of Input Tokens
        - L_2 = Left-Most Derivation
        - R_2 = Right-Most Derivation

          EX:

          a b X c F a Y Z p Q a - Cap = Non Terminal

            - LL expands the Capitalized Letter from left to right
            - RR expands the Capitalized Letter from right to left

- Recursive Descent Parser
    - What is it?
    - How to build (given GMR)
        - It is LL (Left Recursion)
    - Mutual Function Call Recursion
        - A calls B and B calls A (Has to be a loop)
    - PROS
        - Rewind the input
        - Real simple to build the functions
    - CONS
        - Checking all possible Derivations
        - Could take a long time because it tries and fail to check
        - Very SLOW
- LL Parse Machine
    - What is it?
    - What parts?
        - Machines
        - Tables
            - What is it?
            - Parts?
            - Row Headers
                - Has LHS Symbol
            - Column Headers
                - Has Event/Terminal Symbol
            - Cells
                - One Simple Rules in a Cell

- LHS is Row headers
- How does it index?
  - (LHS,Token) = (Row, Column)
- Runtime Stack
- Any LL parser can't handle Left Recursion

- Left Recursion in a Simple Rule
  - What is it?
    - A = A ..
  - Convert Combo Rules to Simple Rules

Glossary:

- LRE (Left-Recursion Elimination)
  - Removes Left Recursion in Grammar

- Disappearing Non-Terminals
  - Direct (easy)
  - Indirect
  - How to find them all

Glossary:

- Terminal Symbol
- Non-Terminal Symbol

What is a Epsilon Rule?

- Rule where Terminal goes to nothing

September 27, 2018

MIDTERM

———————————————————————————————————————

October 2, 2018

GE2 : (LRE'd)

E = [1]TQ

Q = [2]+TQ | [3]-TQ | [4]esp

T = [5]FR <— IF "F" has an esp, then there is a ghost rule for T goes to R

R = [6]*FR | [7]÷FR | [8]esp

F = [9]i | [10]k | [11]'(' E ')'

* Build First Step, Each simple rule (RHS)

LL Table

only terminal —>	i	k	+	-	*	÷	(	)	$ (EOF)
E									
Q									
T									
R									
F	R#9 or F=i								

T (F,i)

stack	input stream
F	i
.	.
.	.
.	.
.	.
$	$

——————> F(R#9) = {i}

L(F=i & GE2) = {"i"}

F(f=i) = {i} F = First Set f=i = GE2 Non-T

F(#9) = {i}

F(#10) = {k}

F(#11) = {'('} <— every thing has to start with (for rule 11

F(#6) = {*}

F(#7) = {÷}

F(#2) = {+}

F(#3) = {-}

F(#4) = { } <— eps goes to nothing (eps = λ)

F(#8) = { } <— λ

F(#5) = {F(f)} = {i,k,'('}

- F(f) = F(#9) + F(#10) + F(#11) = {i,k,'('}

F(#1) = {F(T)} = {i,k,'('}

- F(T) = {F(#5)} = {i,k,'('}

New Table: to get rule, look at equation and see what rule applies

- EX: Q = rule 2 | 3 | 4 —> we we put 2 and 3 under (Q, +) & (Q, -)

only terminal —>	i	k	+	-	*	÷	(	)	$ (EOF)
E	1	1					1		
Q			2	3					
T	5	5					5		
R					6	7			
F	9						11		

First Set Algorithm:

1. If RHS starts with sym X, Add F(x) to F(RHS)
2. If X has an esp rule A = XYZ, then Add F(A=YZ) —> Ghost Rule

   And keep going with Y , etc

3. If like A = …Y & Y has eps rule

   —> Ignore it (Handle in Follow Set)

Follow Sets: W(.)

- . = Non Term

W-1. W(S.S) += $(EOF)

W-2. W(N) :

Find N in each RHS of Rule “Z = ….”

A. If Z = …… NXYZ ……

ADD F(X) to W(N)

B. IF X = eps is a Rule

Because “Ghost Rule” with X gone.

ADD F(Y) to W(N), etc. Z = …N [ ] YZ ….

C. IF Ghost Rule Z = …N <— RHS - Last Symbol

ADD W(Z) to W(N)

Continuing from Set of Rules

W(E) = {$,’)’}

W(Q) = {W(E),W(Q),W(Q)} <— Recursion W(Q)

W(Q) = {$,’)’}

W(T) = {F(Q), W(E),~~F(Q)~~,W(Q)~~,F(Q),W(Q)~~}

= {+ , -, $, ‘)’}

W(R) = {W(T),~~W(R), W(R)~~}

W(R) = {+, -, $, ‘)’}

W(F) = BUT WAIT!!!

Done for 2 reasons:

1. No F = λ Rule

2. No Further worry about other λ Rules

W(E) = Rule 1

W(Q) = Rule 2

F(Q) & W(E) = R#1

F(Q) & W(Q) = R#2

F(Q) & W(Q) = R#3

W(T) = R#5

LL TABLE:

only terminal —>	i	k	+	-	*	÷	(	)	$ (EOF)
E	1	1					1		
Q			2	3				4	4
T	5	5					5		
R			8	8	6	7		8	8
F	9	10					11		

- NO two rules in a single cell
    - It means Grammar Bug (Or your bug)

October 4, 2018

October 9, 2018
