CS 536 Announcements for Thursday, February 6, 2025

Programming Assignment 2

- has been released
- due Tuesday, February 18

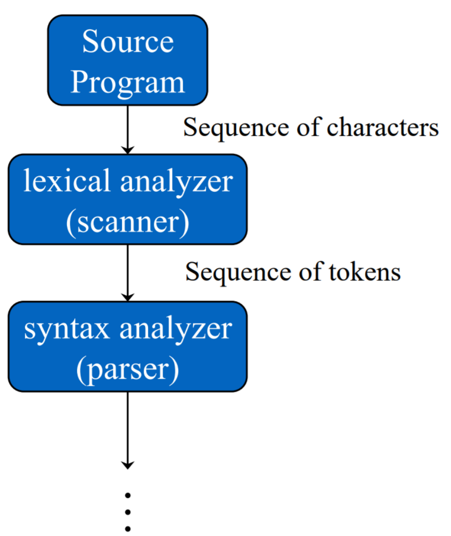
Last Time

- non-determinisitic FSMs
- equivalence of NFAs and DFAs
- regular expressions
- regular languages
- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

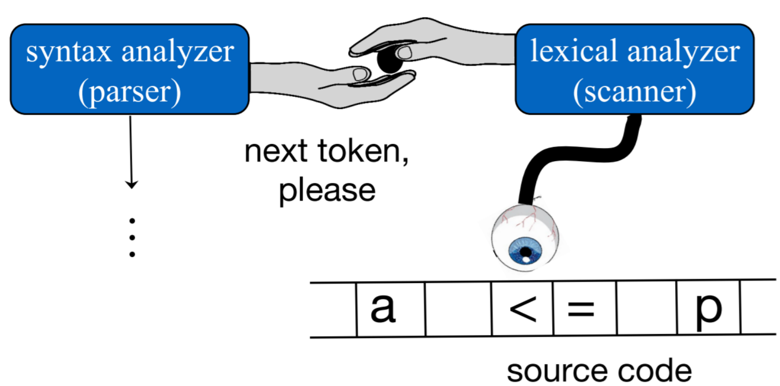
Today

- CFGs
- Makefiles
- resolving ambiguity
- expression grammars
- list grammars

Recall big picture



Conceptual organization



Why regular expressions are not good enough

Regular expression wrap-up

\+ perfect for tokenizing a language

– limitations

- define only limited family of languages
  - can't be used to specify all the programming constructs we need
- no notion of structure

Regexs cannot handle "matching"

Example: $L_{()} = \{ (^n)^n \text{ where } n > 0\}$ = { "( )", "(( ))", "((( )))", ... }

Theorem: No regex/DFA can describe the language $L_{()}$

Proof by contradiction: Suppose there exists a DFA A for $L_{()}$ where A has N states.

Then A has to accept the string $(^N)^N$ with some sequence of states

$$q_0 \xrightarrow{(} q_1 \xrightarrow{(} q_2 \rightarrow \dots \xrightarrow{(} q_N \xrightarrow{)} q_{N+1} \xrightarrow{)} q_{N+2} \xrightarrow{)} \dots \xrightarrow{)} q_{2N}$$

accepts $(^N$ accepts $)^N$

By the pigeonhole principle, there exists $i, j \leq N$ where $i < j$ such that $q_i = q_j$

So

$$q_0 \xrightarrow{(} q_1 \xrightarrow{(} q_2 \xrightarrow{(} \dots \xrightarrow{(} q_{i=j} \xrightarrow{(} q_{j+1} \xrightarrow{(} \dots \xrightarrow{(} q_N \xrightarrow{)} q_{N+1} \xrightarrow{)} \dots \xrightarrow{)} q_{2N}$$

accepts $(^{i+(N-j)}$ accepts $)^N$

In other words,

A accepts $(^{N-j+i})^N$ but $(^{N-j+i})^N \notin L_{()}$

which is a contradiction

No notion of structure

Consider the following stream of tokens: ID ASSIGN ID PLUS ID ($x = y + z$)

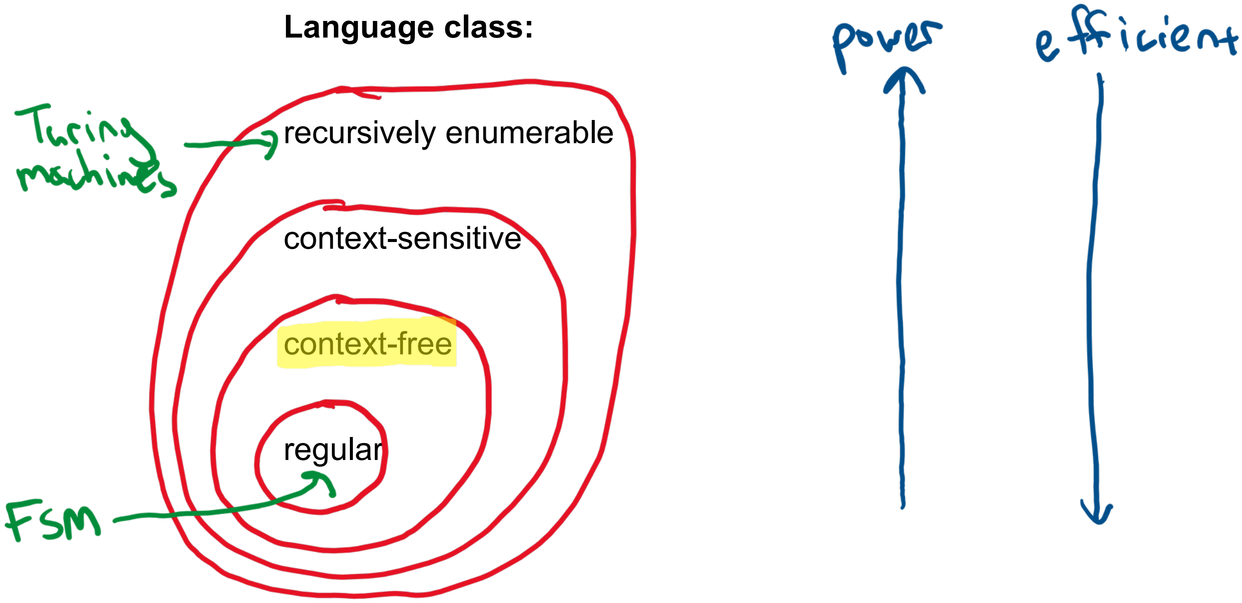
What should be done 1st? $(x = y) + z$ or $x = (y + z)$

What about precedence & associativity

(Noam)

The Chomsky Language Hierarchy

Language class:

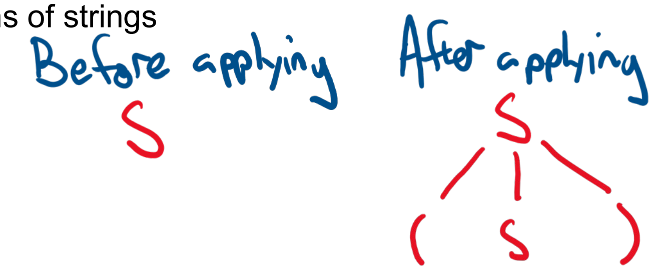


Context-free grammar (CFG)

= a set of recursive rewriting rules to generate patterns of strings

S → '(' S ')'

rule for rewriting S



Formal definition: A CFG is a 4-tuple (N, ∑, P, S)

- N = set of **non-terminals** – placeholders (interior nodes in parse tree)
- ∑ = set of **terminals** – tokens from scanner
- P = set of **productions** – rules for re-writing non-terminals (ie for deriving)
- S = initial non-terminal symbol ("start symbol"), S ϵ N

– if not otherwise specified, use non-term on LHS of 1st production as start symbol

Productions

Production syntax : LHS → RHS

single non-terminal ↑ ↑ expression (seq of terms & non-terms) or ε

non-term → expression
non-term → ε

$S \rightarrow$ '(' S ')'
$S \rightarrow \varepsilon$

or (assuming non-terms on LHS are the same)

non-term → expression
| ε

$S \rightarrow$ '(' S ')'
| ε

or
non-term → expression | ε

$S \rightarrow$ '(' S ')' | ε

Language defined by a CFG

= set of strings (i.e., sequences of terminals) that can be derived from the start non-terminal

To derive a string (of terminal symbols):

- set Curr_Seq to start symbol
- repeat
  - find a non-terminal x in Curr_Seq
  - find production of the form x → α
  - "apply" production: create new Curr_Seq by replacing x with α
- until Curr_Seq contains no non-terminals

Derivation notation

- derives $\Rightarrow$
- derives in one or more steps $\Rightarrow^+$ or $\stackrel{+}{\Rightarrow}$
- derives in zero or more steps $\Rightarrow^*$ or $\stackrel{*}{\Rightarrow}$

L(G) = language defined by CFG G

= { w | $S \stackrel{+}{\Rightarrow} w$ where S is start non-terminal & w is seq of terms or ε }

↑ "such that"

Example grammar

Terminals

BEGIN
END } program boundary

SEMICOLON - ";" to separate statements

ASSIGN - "=" in asg stmts

ID - identifier (variable name)

PLUS - "+" operator in expression

Non-terminals

prog - start non-term (root of parse tree)

stmts - list of statements

stmt - a single statement

expr - a (mathematical expression)

Productions - define syntax of legal programs

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID

Example derivation

Productions

1) prog → BEGIN stmts END
2) stmts → stmts SEMICOLON stmt
3) | stmt
4) stmt → ID ASSIGN expr
5) expr → ID
6) | expr PLUS ID

Derivation

prog

⟹ BEGIN stmts END (1)

⟹ BEGIN stmts SEMICOLON stmt END (2)

⟹ BEGIN stmt SEMICOLON stmt END (3)

⟹ BEGIN ID ASSIGN expr SEMICOLON stmt END (4)

⟹ BEGIN ID ASSIGN expr SEMICOLON ID ASSIGN expr END (4)

⟹ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr END (5)

⟹ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr PLUS ID END (6)

⟹ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN ID PLUS ID END (5)

BEGIN ID = ID; ID = ID + ID END

prog $\stackrel{+}{\Rightarrow}$ BEGIN ID = ID; ID = ID + ID END

Q: is BEGIN ID = ID; END in language?

No - but would be if we added stmt → ε

Start again at 7:35pm

Parse trees

= way to visualize a derivation

To derive a string (of terminal symbols):

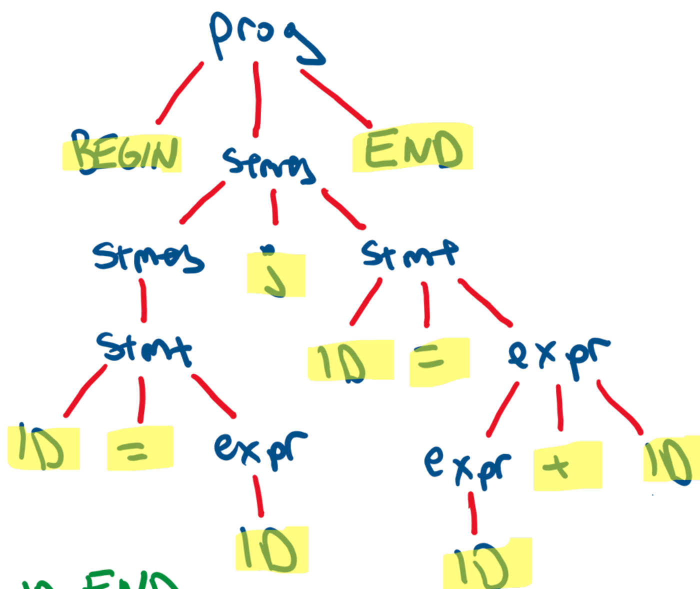
- set root of parse tree to start symbol
- repeat
  - find a leaf non-terminal x
  - find production of the form x → α
  - "apply" production: symbols in α become the children of x
- until there are no more leaf non-terminals

Derived sequence determined from leaves, from left to right

Productions

1) prog → BEGIN stmts END
2) stmts → stmts SEMICOLON stmt
3) | stmt
4) stmt → ID ASSIGN expr
5) expr → ID
6) | expr PLUS ID

↑ this notation is BNF (or extended BNF)
Backus-Naur Form



BEGIN ID = ID; ID = ID + ID END

Makefiles

Basic structure

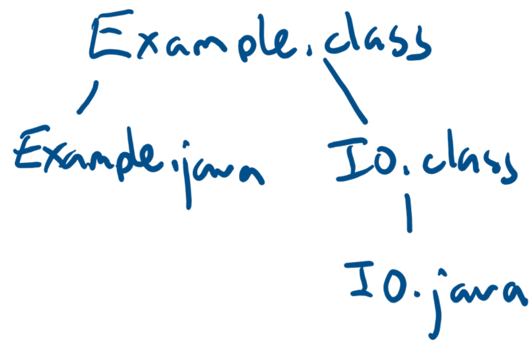
```
<target>: <dependency list>
        <command to satisfy target)
```

tab

Example

```
Example.class: Example.java IO.class
        javac Example.java

IO.class: IO.java
        javac IO.java
```



Make creates an internal **dependency graph**

- a file is rebuilt if one of its dependencies changes

Variables – for common configuration values to use throughout your makefile

Example

```
JC = /s/std/bin/javac
JFLAGS = -g
```

← build for use with debugger

```
Example.class: Example.java IO.class
        $(JC) $(JFLAGS) Example.java

IO.class: IO.java
        $(JC) $(JFLAGS) IO.java
```

Phony targets

- target with no dependencies = "phony"
- use make to run commands:

Example

force remove

```
clean:
        rm -f *.class
```

test:
java Example inFile.txt outFile.txt
java Example inErrFile.txt outErrFile.txt

Programming Assignment 2

Modify:

- bach.jlex
- P2.java
- Makefile

Makefile

```
###
# testing - add more here to run your tester and compare
# its results to expected results
###
test:
	java -cp $(CP) P2
	diff allTokens.in allTokens.out

###
# clean up
###

clean:
	rm -f *~ *.class bach.jlex.java

cleantest:
	rm -f allTokens.out
```

Run make to compile (by default make does 1st target in Makefile)

Running the tester

```
vm-instunix-07(53)% make test
java -cp ./deps:. P2
3:1 ****ERROR**** ignoring illegal character: a
diff allTokens.in allTokens.out
3d2
< a
make: *** [Makefile:40: test] Error 1
```

error msg produced by bach scanner when P2 is run

commands from Makefile

output of diff command

from running make

CFG review

terminals ≡ tokens

formal definition: CFG *G* = (N, ∑, P, S)

CFG generates a string by applying productions until no non-terminals remain

⟹+ means "derives in 1 or more steps"

q ⟹ (q) ⟹ (ε) ie ( )

language defined by a CFG *G*

L(*G*) = { w | s ⟹+ w} where

s = start is the start non-terminal of G, an

w = sequence consisting of (only) terminal symbols or ε

L(G) = { ε, (), (()), ((())), ... }

Example: nested parens

N = { q }

∑ = { '(', ')' }

P = q → (q)
 | ε

q ⟹ (q)
⟹ ((q))
⟹ (((q)))
⟹ (((ε)))

q ⟹+ ((( )))

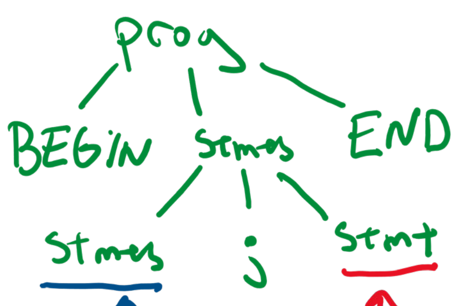
Derivation order

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID



Leftmost derivation : leftmost non-terminal is always expanded

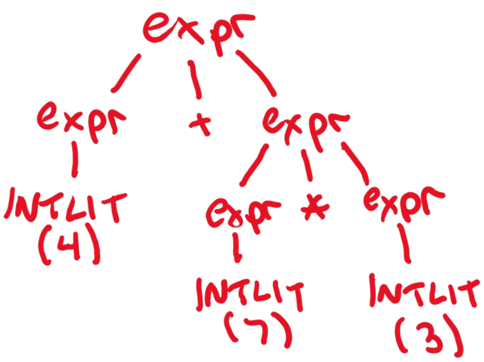
Rightmost derivation : rightmost non-terminal is always expanded

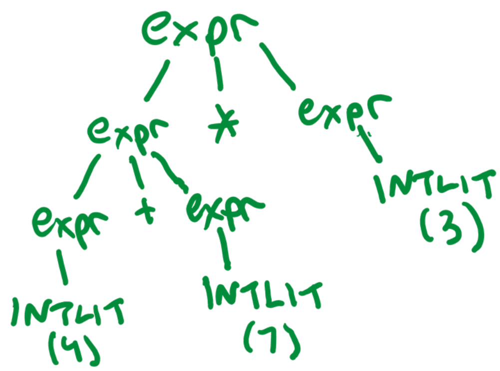
Expression Grammar Example

1) expr → INTLIT
2) | expr PLUS expr
3) | expr TIMES expr
4) | LPAREN expr RPAREN

Goal: create a CFG for arithmetic expressions involving only +, *, parens, & integer literals

Derive: 4 + 7 * 3





ambiguous grammar!

For grammar G and string w, G is **ambiguous** if there is

> 1 leftmost derivation of w

OR

> 1 rightmost derivation of w

OR

> 1 parse tree for w

these are all equivalent

Grammars for expressions

Goal: write a grammar that correctly reflects precedences and associativities

a+b*c ↔ a+(b*c)

↳ a+b+c ↔ (a+b)+c
a=b=c ↔ a=(b=c)

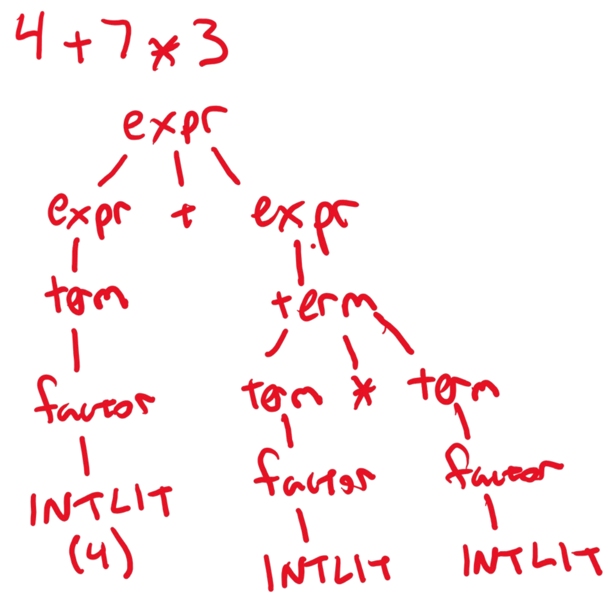
Precedence

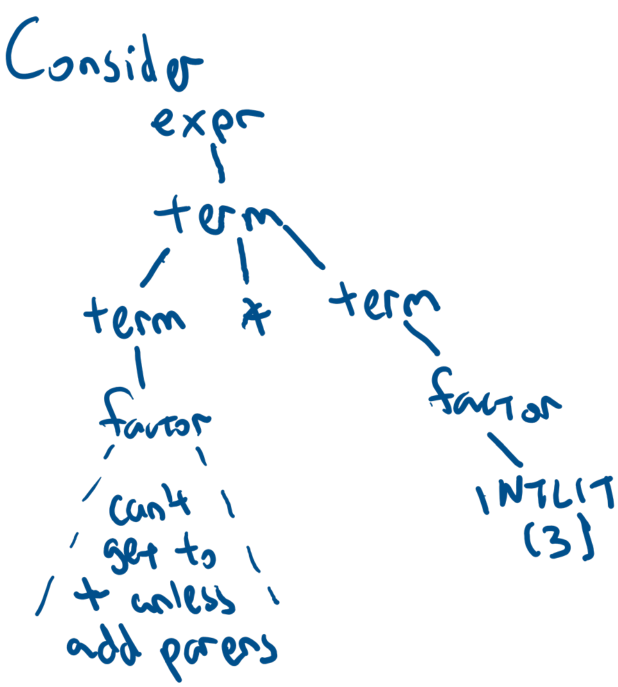
- use different non-terminal for each precedence level
- start by re-writing production for lowest precedence operator first

Example

1) expr → INTLIT
2) | expr PLUS expr ← + has lowest precedence
3) | expr TIMES expr
4) | LPAREN expr RPAREN

```
expr → expr + expr
     | term
term → term * term
     | factor
factor → INTLIT
       | ( expr )
```





Grammars for expressions (cont.)

What about associativity? Consider 1 + 2 + 3   equiv (1+2)+3

Definition: recursion in grammars

A grammar is ***recursive*** **in non-terminal** ***x*** if
x ⟹+ α *x* γ for non-empty strings of symbols α and γ

A grammar is ***left-recursive*** **in non-terminal** ***x***
if *x* ⟹+ *x* γ for non-empty string of symbols γ

A grammar is ***right-recursive*** **in non-terminal** ***x*** if
x ⟹+ α *x* for non-empty string of symbols α

left associative
\+ − * /

right associative
= ^ ← exponentiation

$2^3^4 \equiv 2^{3^4}$

In expression grammars

for left associativity, use left recursion

for right associativity, use right recursion

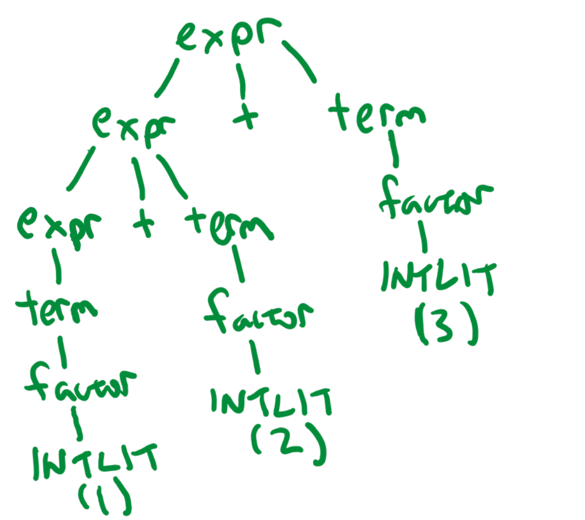
Example

expr → expr + ~~expr~~ term
  | term

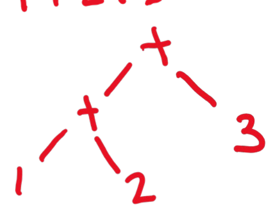
term → term * ~~term~~ factor
  | factor

factor → INTLIT
  | (expr)

1+2+3



AST for 1+2+3



Extend this grammar to add exponentiation (POW)

Add exponentiation (POW) to this grammar, with the correct precedence and associativity.

↳ highest ↳ right

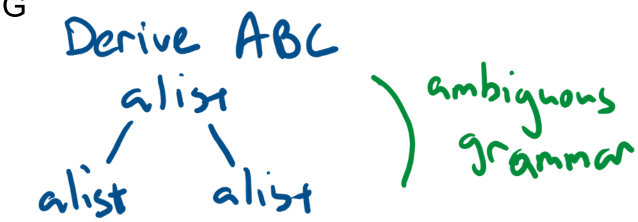
```
expr   →  expr PLUS term
       |  term
term   →  term TIMES factor
       |  factor
factor →  INTLIT
       |  LPAREN expr RPAREN
```

```
factor → exponent POW factor
       | exponent
exponent → INTLIT
       | LPAREN expr RPAREN
```

List grammars

Example a list with no separators, e.g., A B C D E F G

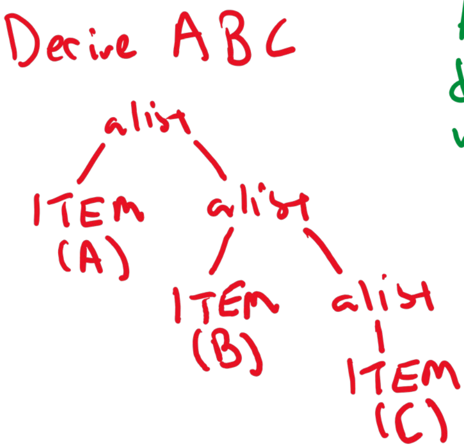
alist → ITEM
| alist alist



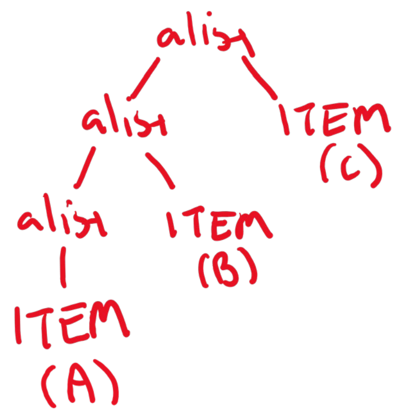
alist → ITEM
| ITEM alist

OR

alist → ITEM
| alist ITEM



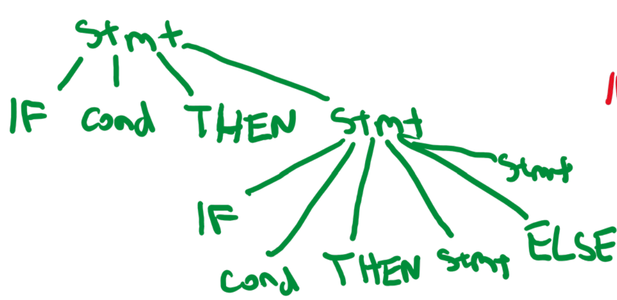
Associativity doesn't matter with lists so either grammar is fine

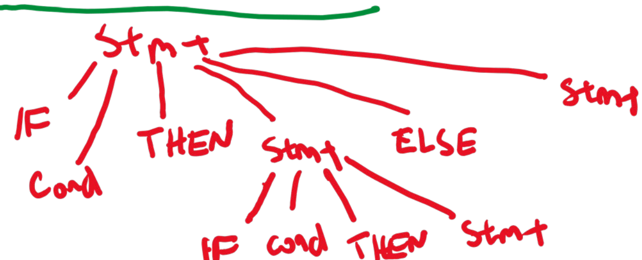


Another ambiguous example

stmt → IF cond THEN stmt
| IF cond THEN stmt ELSE stmt
| . . .

Given this word in this grammar: `if a then if b then s1 else s2`
How would you derive it?





stmt → IF cond THEN LCURLY stmt RCURLY
| IF cond THEN LCURLY stmt RCURLY ... ELSE LCURLY stmt RCURLY