CS 536 Announcements for Monday, February 12, 2024

Programming Assignment 2 – due Tuesday, February 20

Last Time

- why regular expressions aren't enough
- CFGs
  - formal definition
  - examples
  - language defined by a CFG

Today

- Makefiles
- parse trees
- resolving ambiguity
- expression grammars
- list grammars

Next Time

- syntax-directed translation

Makefiles

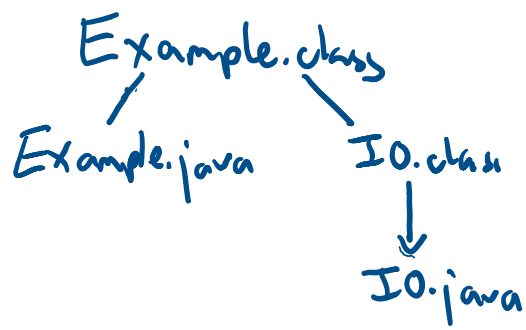
Basic structure

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

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

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

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

force remove

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

Programming Assignment 2

Modify:

- base.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 base.jlex.java

cleantest:
    rm -f allTokens.out
```

Run make to compile P2 (by default make does 1st target in Makefile)

Running the tester

```
royal-12(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 base scanner when P2 is run

commands from Makefile

output of diff command

from running make

CFG review

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

(∑ ← terminals ≡ tokens)

Example: nested parens
N = {q}
∑ = { (,) }
P = q → (q)
    | ε

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

⟹+ means "derives in 1 or more steps"

q ⟹ (q) ⟹ (ε) ie ( )

q ⟹ ε

q ⟹ (q) ⟹ ((q)) ⟹ ((ε))

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

q ⟹+ (( ))

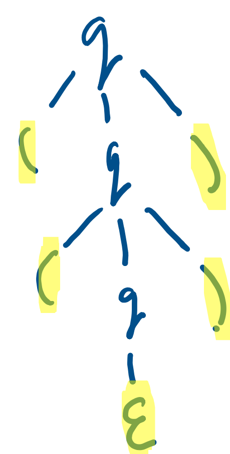
L(G) = { ε, ( ), (( )), ((( ))), ... }

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

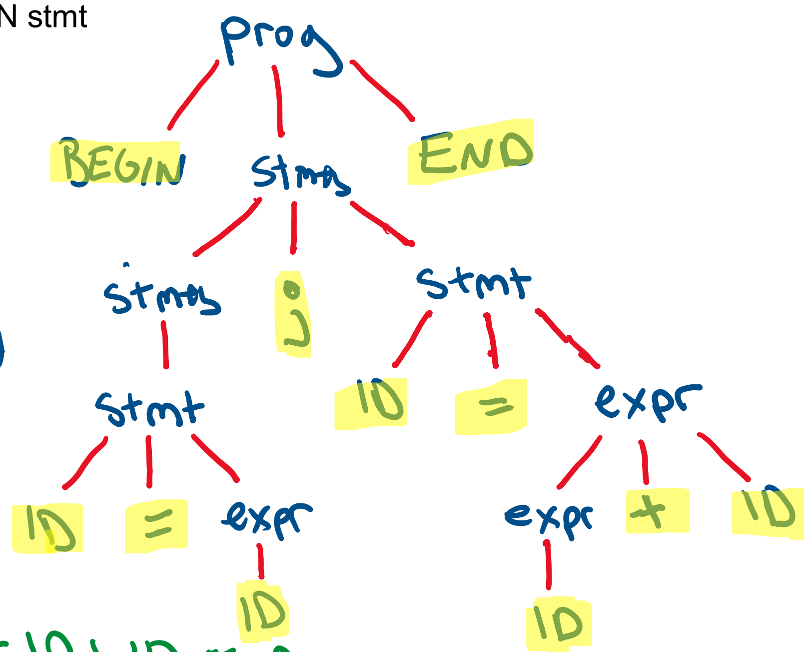
Sequence is: (( ))

Parse tree example

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

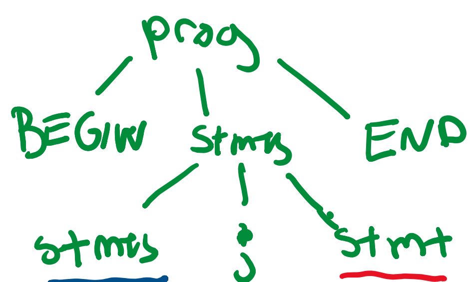


BEGIN ID=ID; ID = ID+ID END

Derivation order

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



Leftmost derivation : leftmost non-terminal is always expanded

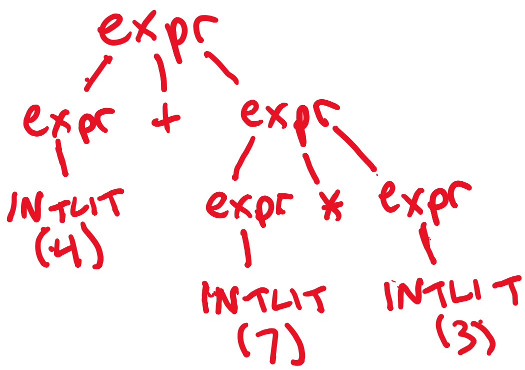
Rightmost derivation : rightmost non-terminal is always expanded

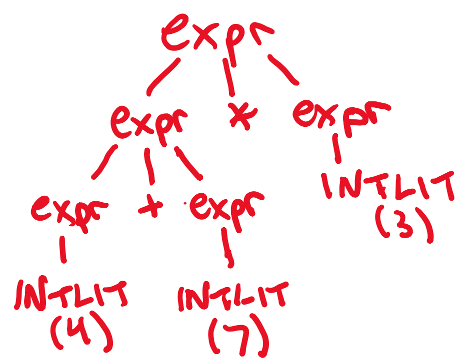
Expression Grammar Example

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

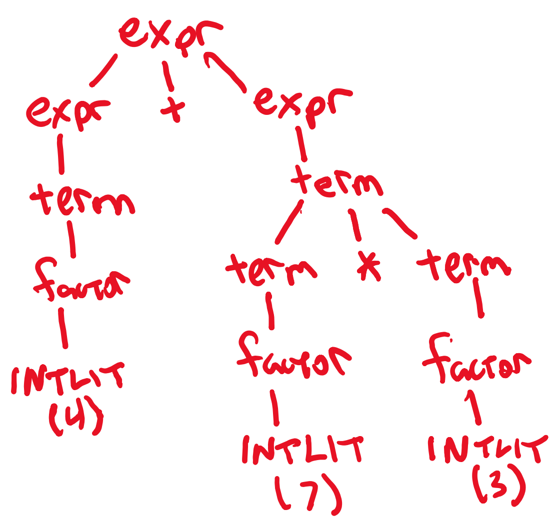
1) expr → INTLIT
2) | expr PLUS expr ← + has lowest precedence
3) | expr TIMES expr
4) | LPAREN expr RPAREN

expr → expr + expr
| term

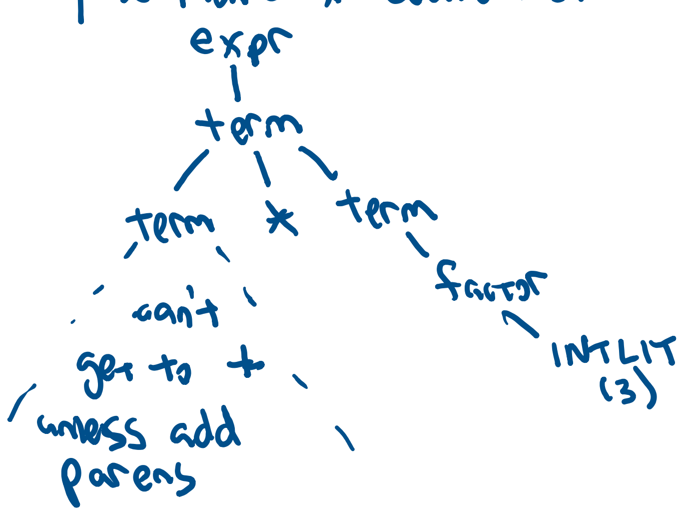
term → term * term
| factor

factor → INTLIT
| (expr)

4+7*3

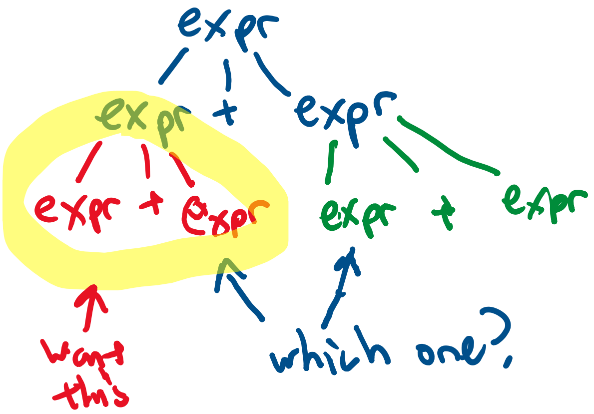


Try to make * eval'd last



Grammars for expressions (cont.)

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



left associative

+ - * /

right associative

= ^

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

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

In expression grammars

for left associativity, use left recursion

for right associativity, use right recursion

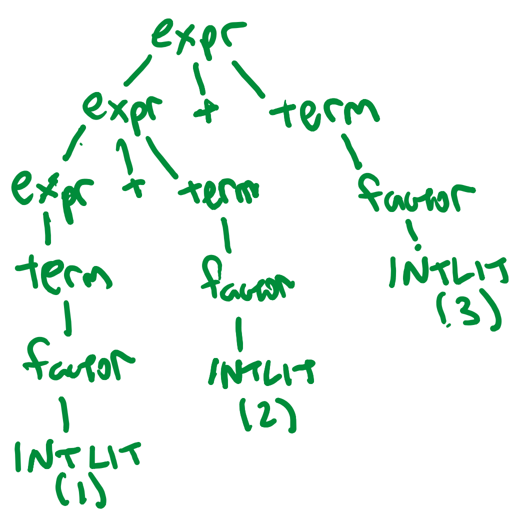
Example

expr → expr + term
| term

term → term * factor
| factor

factor → INTLIT
| (expr)

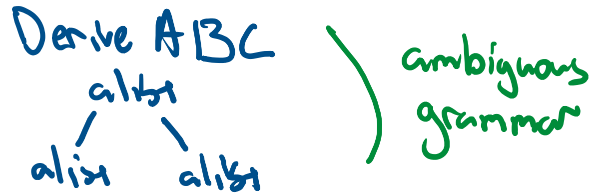
1+2+3



List grammars

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

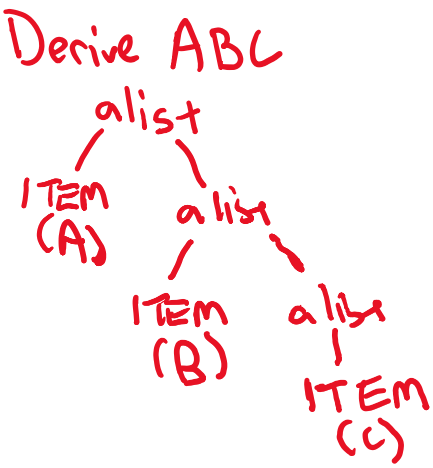
alist → ITEM
 | alist alist



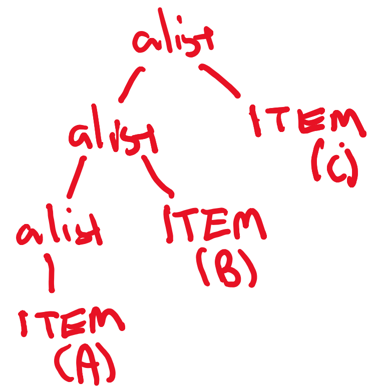
alist → ITEM
 | ITEM alist

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