EECS4302
Compilers and Interpreters

Winter 2020

Instructor: Jackie Wang

LECTURE 01

MONDAY JANUARY 06

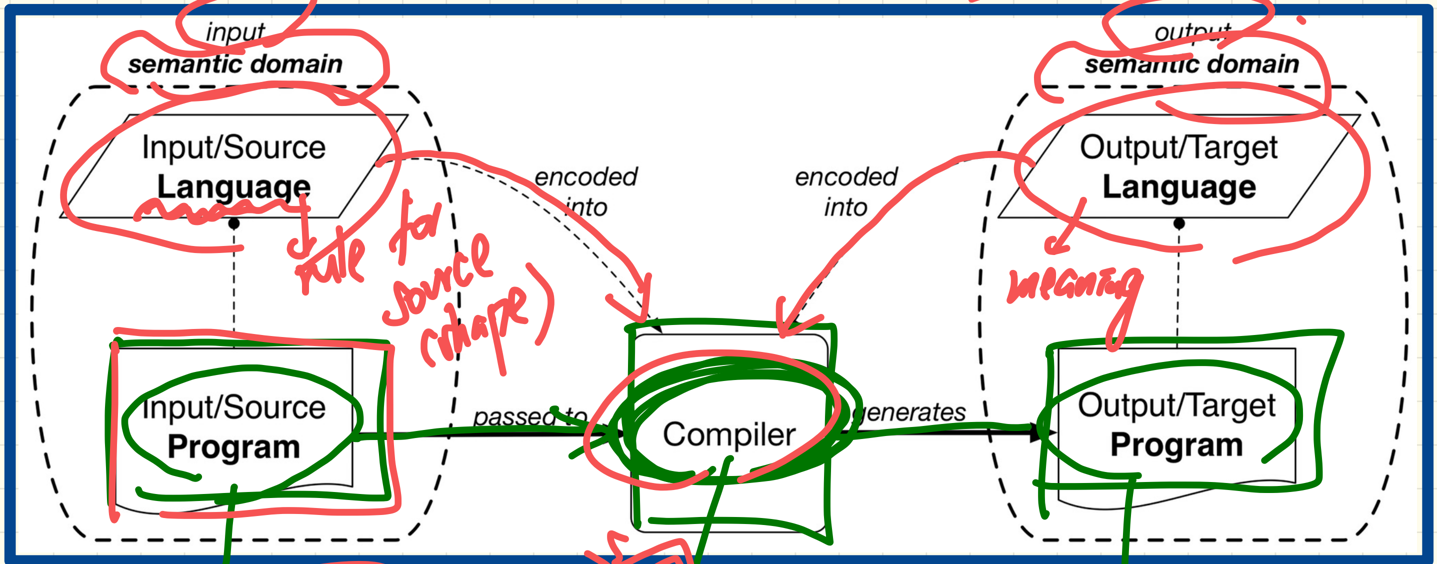
Course Learning Outcomes (CLOs)

Upon completion of the course, students are expected to develop their:

- **CLO1** Compare and contrast General-Purpose Languages (GPLs) and Domain-Specific Languages (DSLs).
- **CLO2** Apply the theoretical understanding of, and use the relevant tools to generate, a lexical scanner.
- **CLO3** Apply the theoretical understanding of, and use the relevant tools to generate, a grammatical parser.
- **CLO4** Construct an abstract syntax tree and perform various semantic operations on it.
- **CLO5** Communicate the design and implementation of a DSL and its associated tools.

What is a **Compiler**?

semantics-preserving



e.g. Java

for(int i=0; ...

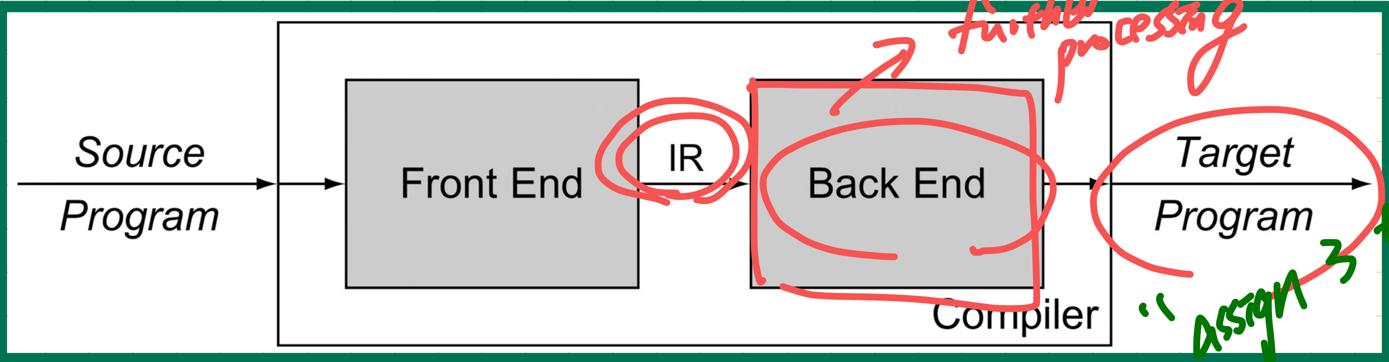
software system

if

byte code

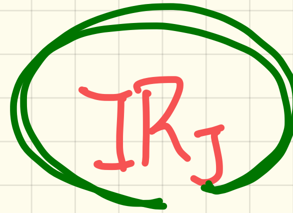
SQL database

Compiler: Typical Infrastructure (1)

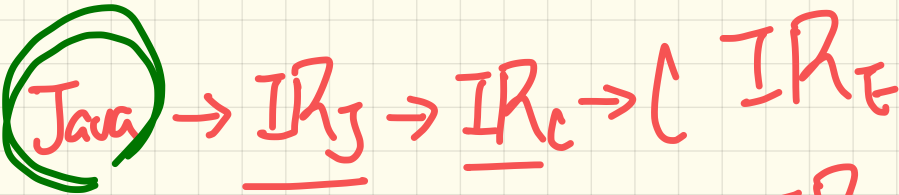


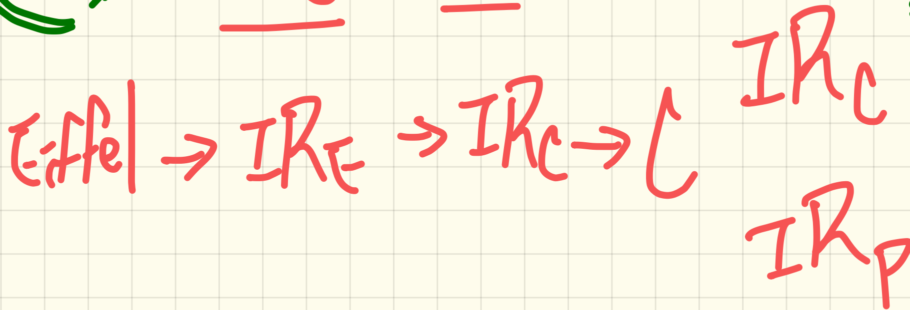
Q. How many IRs are necessary to build a number of compiler?

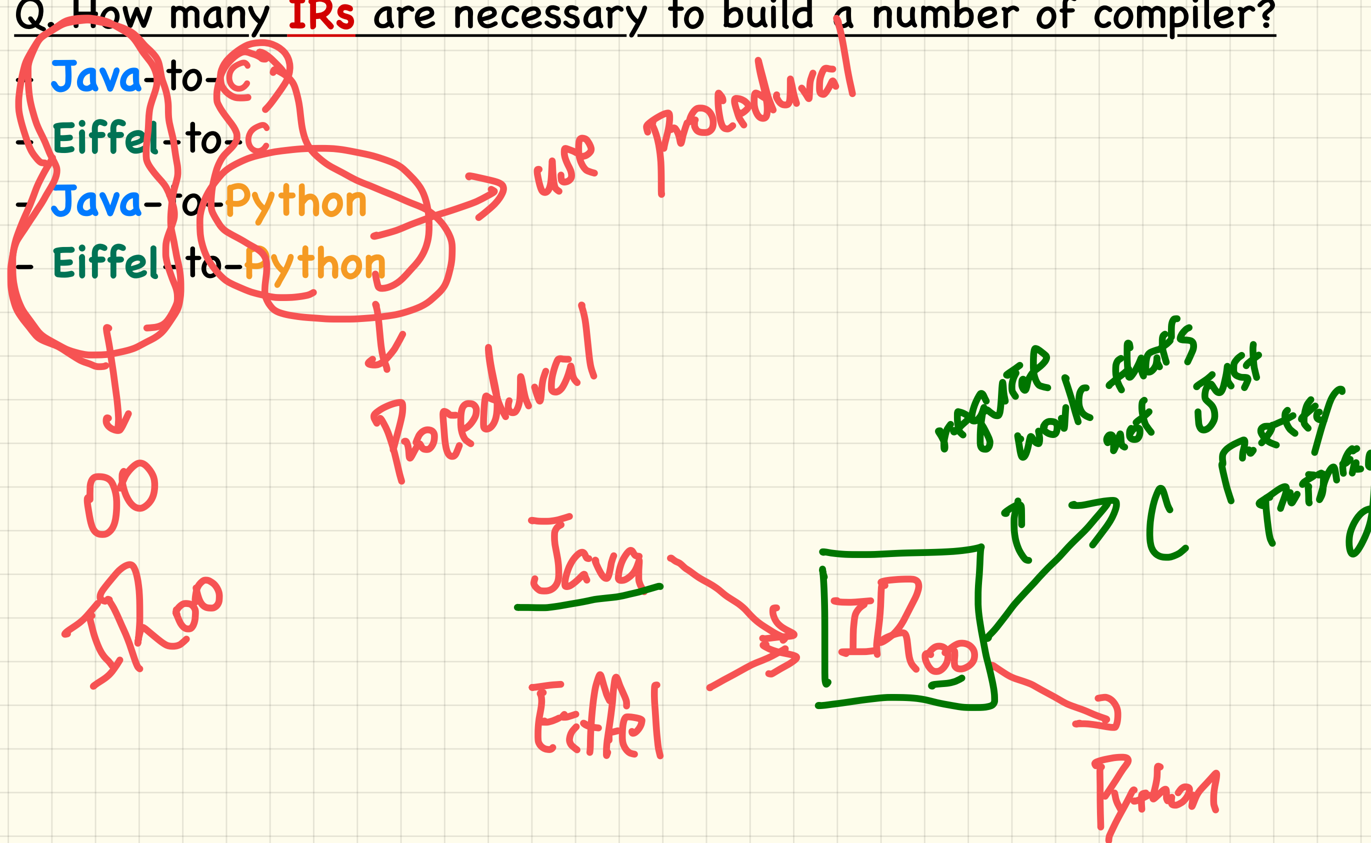
- Java-to-C
- Eiffel-to-C
- Java-to-Python
- Eiffel-to-Python



how many Java features should be covered.



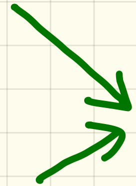




Q. How many IRs are necessary to build a number of compiler?
- Java-to-C
- Eiffel-to-C
- Java-to-Python
- Eiffel-to-Python
use Procedural
Procedural
OO
IR OO
Java
Eiffel
IR OO
C
Python

One possible way.

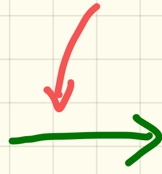
Java



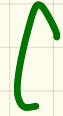
Eiffel

IRoo

strategy of using procedural to implement OO

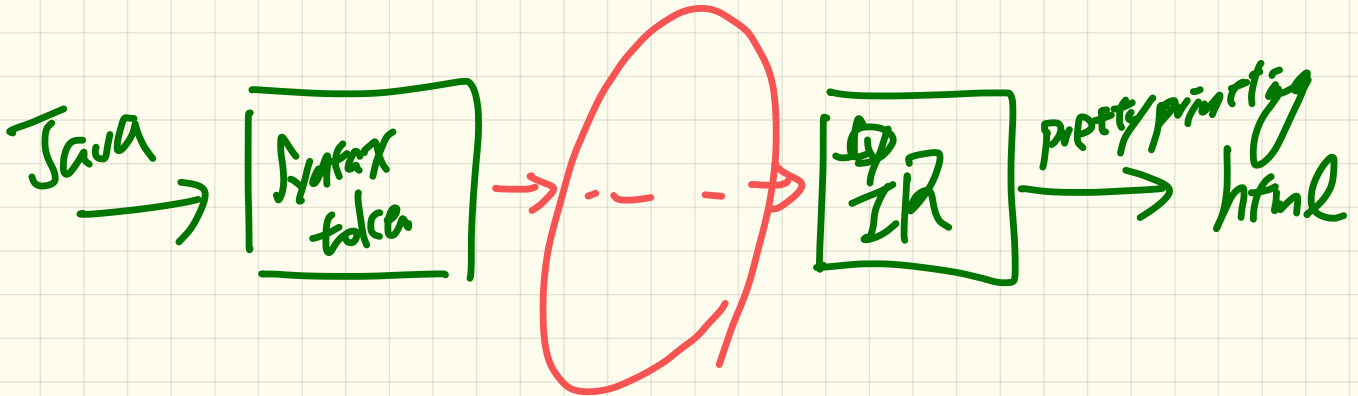


IRprocedural



C Python Java Eiffel

Java-to-html compiler?

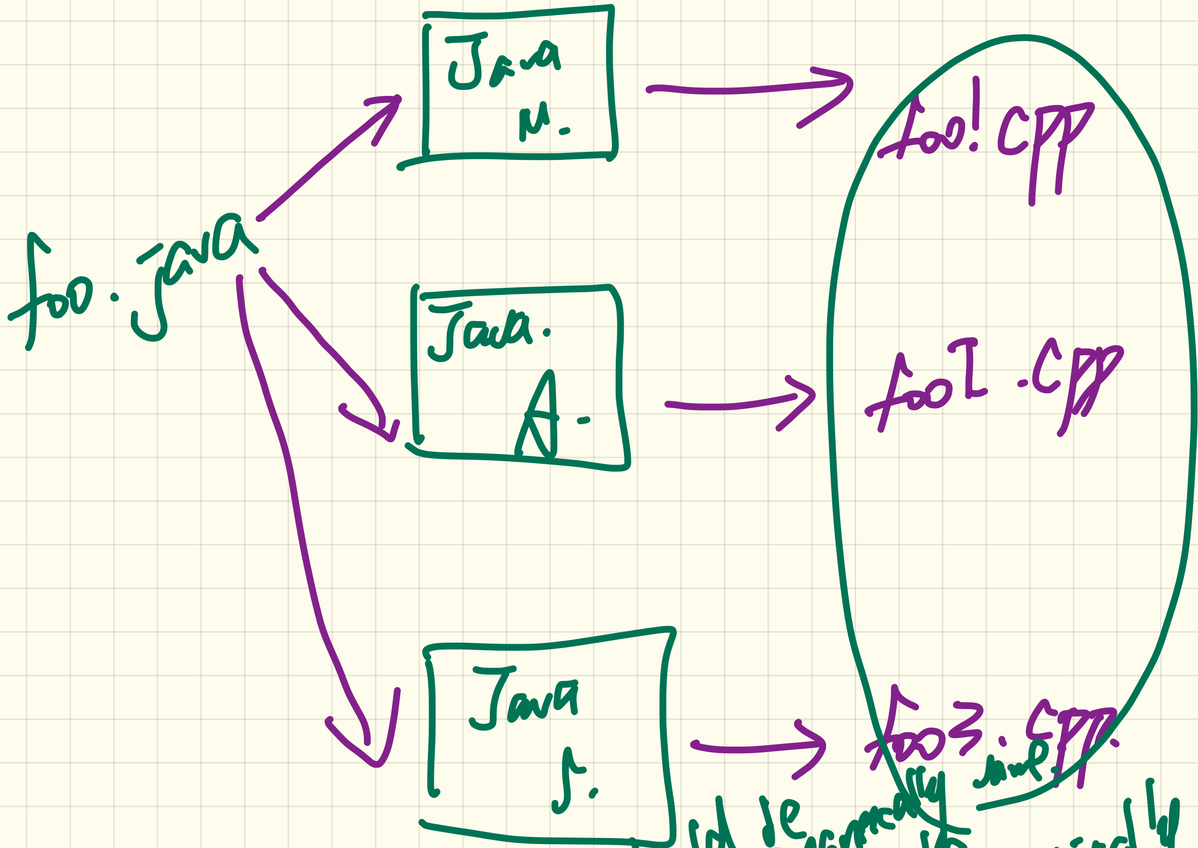


Compiler: Typical Infrastructure (2)

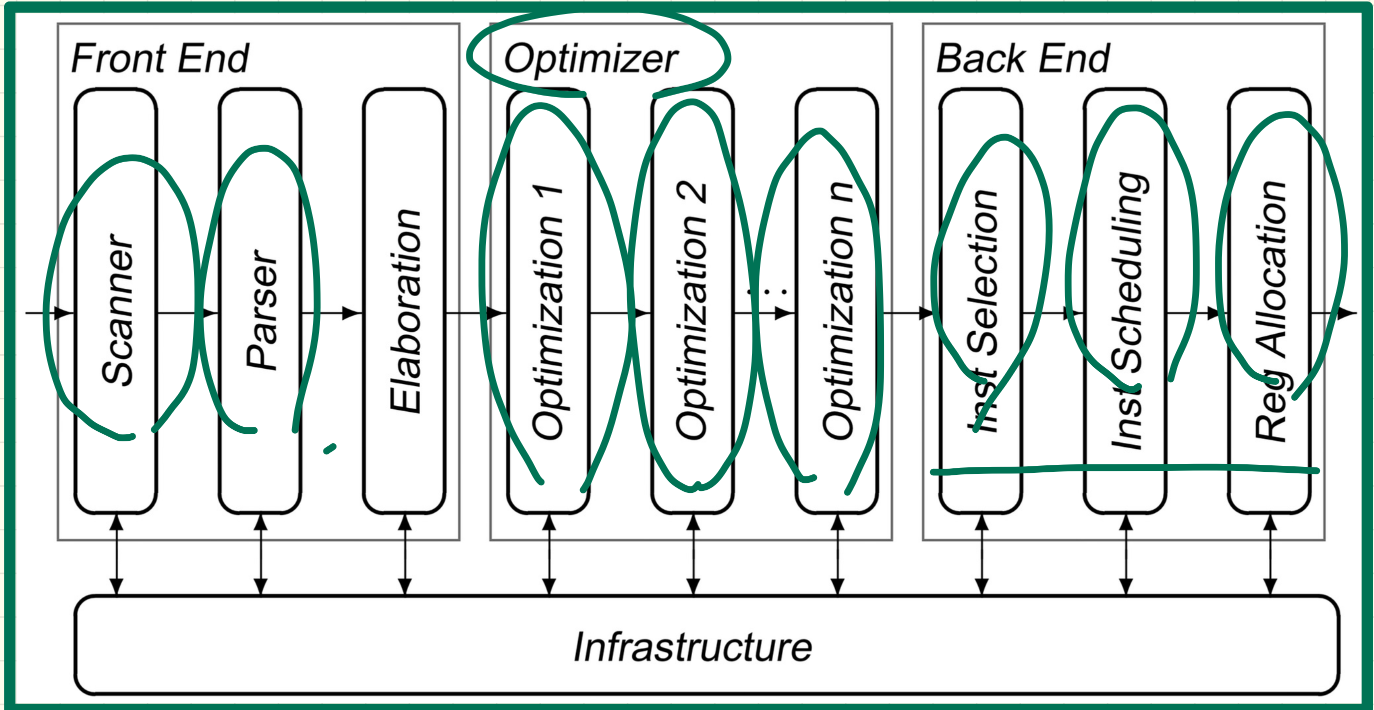
Source Program → Front End → IR → Optimizer → IR → Back End → Target Program

Compiler

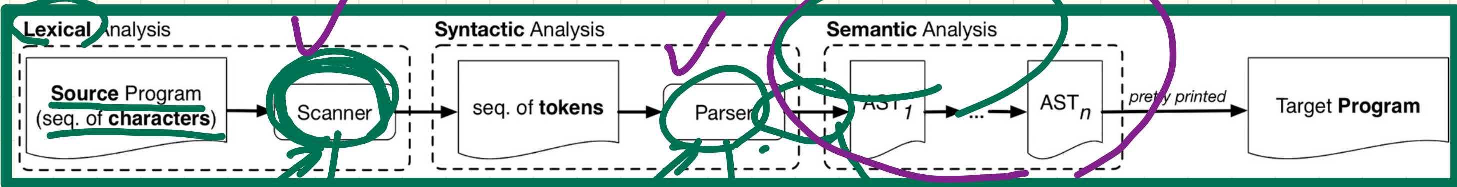
Q. What is the behaviour of the target program predicated upon?

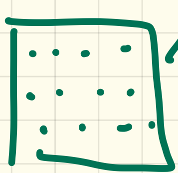

foo.java
Java M.
Java A.
Java J.
foo1.cpp
foo2.cpp
foo3.cpp
2. should be semantically same
1. may be syntactically different.

Example Compiler One: Infrastructure



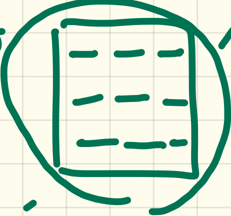
Example Compiler One: Scanner, Parser, Optimizer





characters

words



words

sentences

1. valid tokens
2. valid composition of tokens

subject verb noun

I play exam.

I eat exam.

class

Example

Java compile

```
class @234Foo {
    int a =
}
```

invalid token for identifier

Token_CLASS → class

ID_Token

LBRAC → {

Body of Class → RNon Terminal

RBRAC → }

not matched.

parser error.

pass scanner

class	Foo	{
int	a	=
}		

```
class A {
  a : int
}
```

→ composition of tokens for var. declaration in wrong order

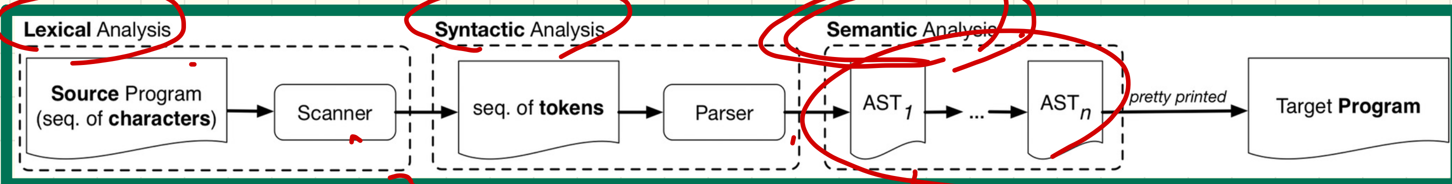
```
class A {
  B b;
}
```

→ semantic error unknown class

LECTURE 2

WEDNESDAY JANUARY 8

Example Compiler One: Scanner, Parser, Optimizer



class A {
B b;
C c;
}

Python
Haskell

IR → IR

semantics-preserving

class A { B b; C c; }

[a-zA-Z]*

a_b

[a-zA-Z_]*

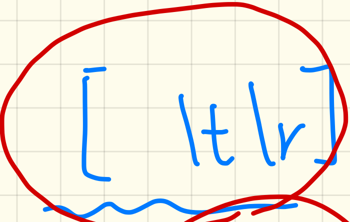
_ab

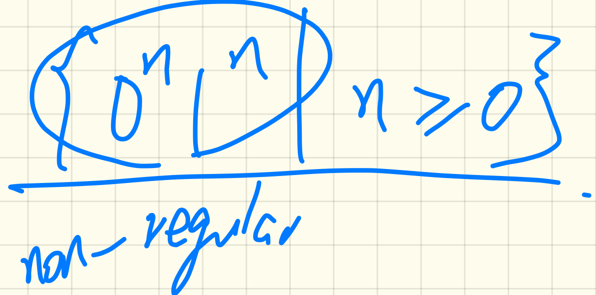
[a-zA-Z] [a-zA-Z_]*

[_a-zA-Z]

CFG?

Scanner





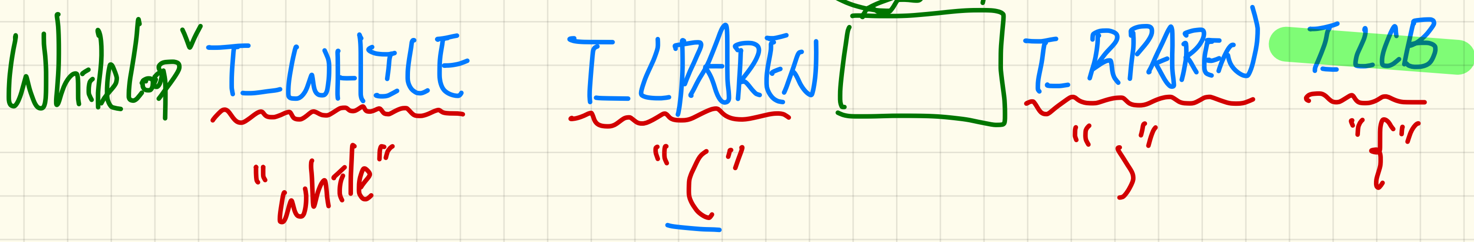
compiler

Scanner

CFG for while-loop.

BoolExp — Non-Terminals (recursive case)

::=



Instruction

T_RCB

"}"

BoolExp ::= . -

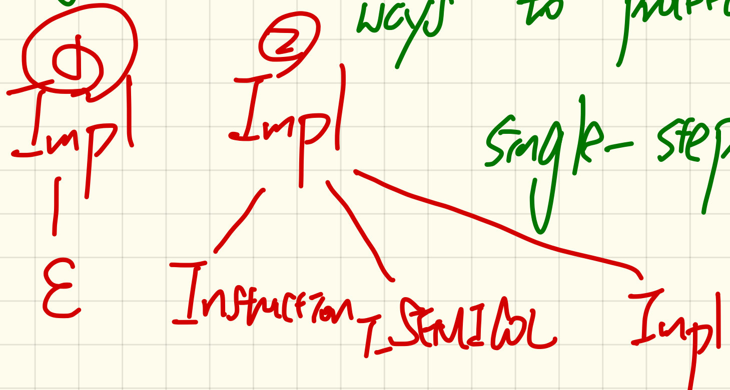
Instruction ::= . -

while-Loop: Context-Free Grammar (CFG)

WhileLoop	::=	WHILE LPAREN *BoolExpr* RPAREN LCBRAC *Impl* RCBRAC
Impl	::=	
	|	*Instruction* SEMICOL *Impl*

alternative (or)

starting from Impl, how many possible ways to proceed with a single-step of derivation?



① while (. . . .) {
x = y;
}

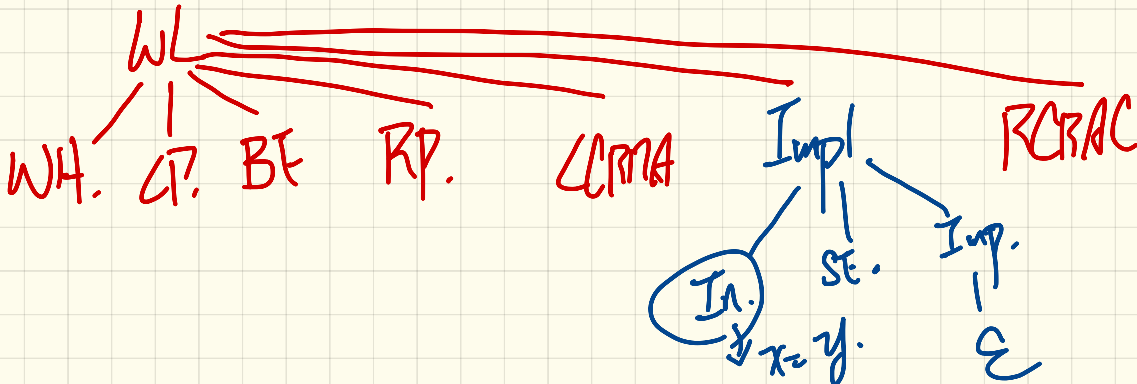
② while (. . .) { x = y;
y = z ; }

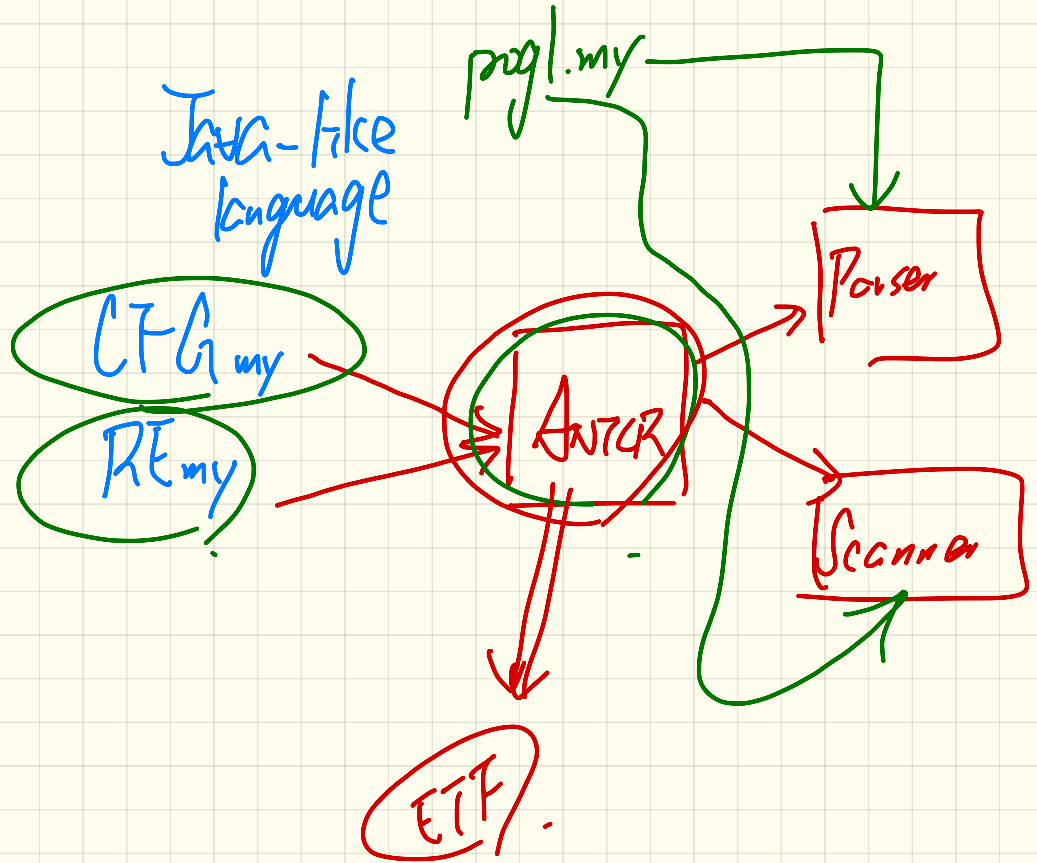
Q: ARRAY[INT] a[0].

```
WhileLoop ::= WHILE LPAREN BoolExpr RPAREN LCBRAC Impl RCBRAC
Impl      ::= ①
            | ② Instruction SEMICOL Impl
```

① while (. . .) {
x = y;
}

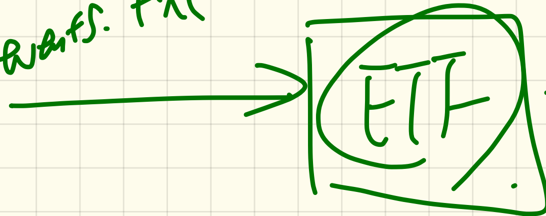
② while (. . .) { x = y;
y = z; }

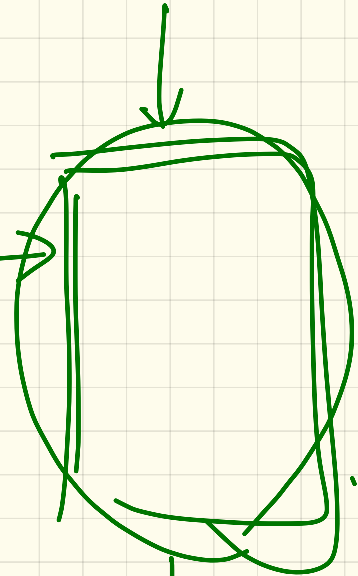




Java-like language
prog1.my
CFG my
RE my
ANTLR
Parser
Scanner
ETF

atol.txt



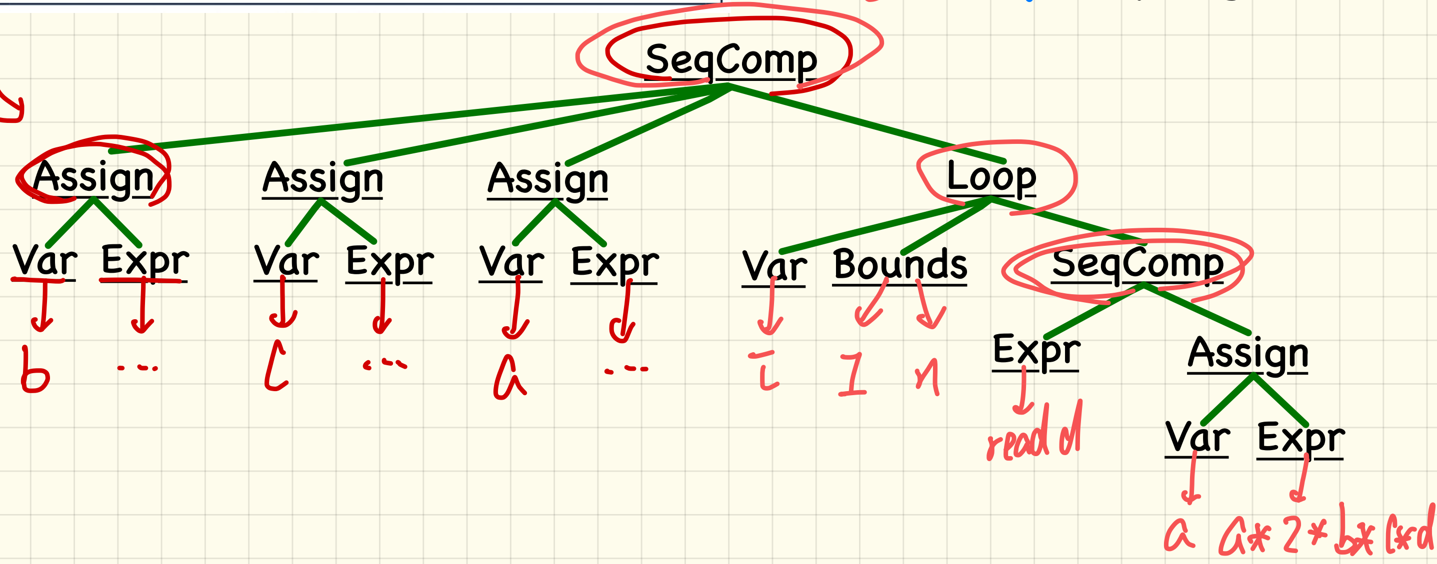


atol.out.txt

Example Compiler One: AST-to-AST Optimizer (1)

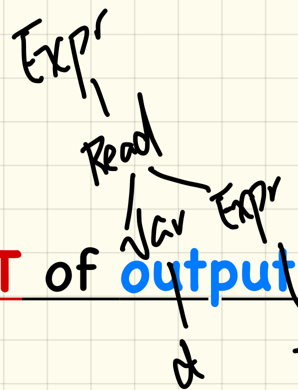
```
b := ... ; c := ... ; a := ... ;
across 1 |..| n is i
  loop
    read d
    a := a * 2 * b * c * d
  end
```

AST of input program:



Example Compiler One: AST-to-AST Optimizer (2)

```
b := ... ; c := ... ; a := ...
temp := 2 * b * c
across 1 |..| n is i
  loop
    read d
    a := a * d
  end
```



AST of output program:

SeqComp

Assign — Var, Expr

Assign — Var, Expr

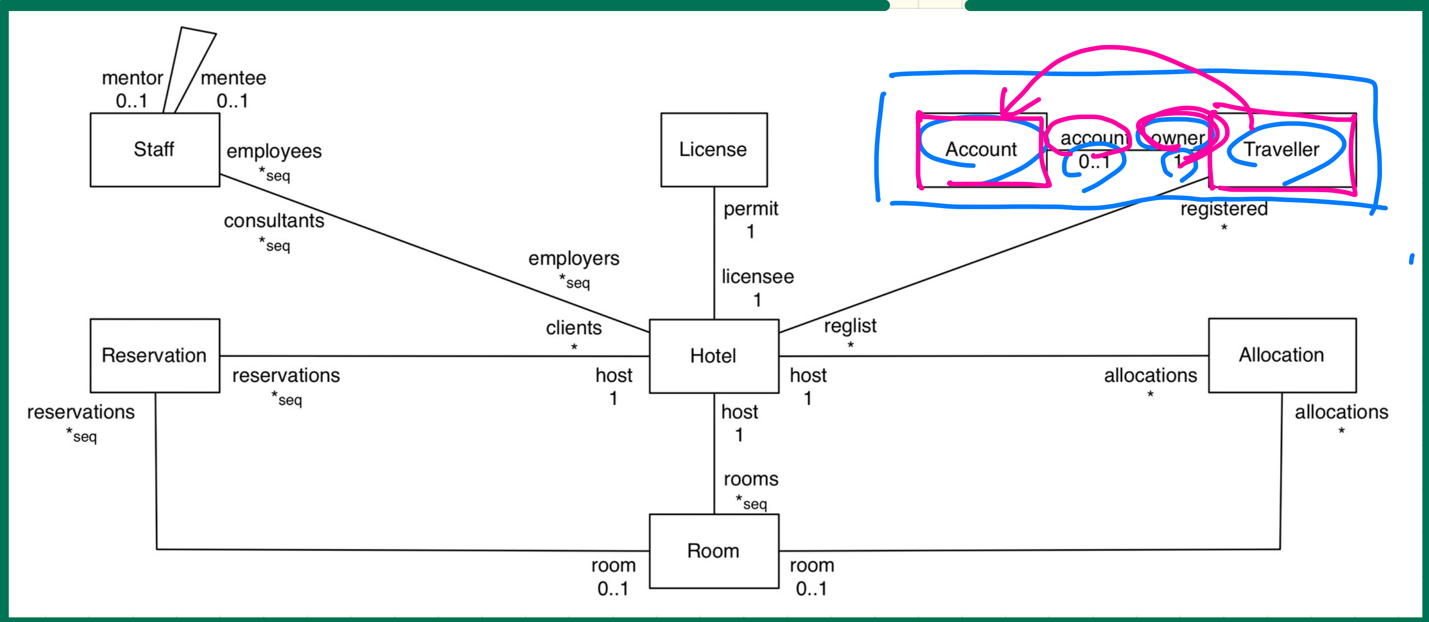
Assign — Var, Expr

Loop — Var, Bounds, SeqComp

SeqComp — Expr, Assign

Assign — Var, Expr

Example Compiler Two: Data Model




```
class Account{
  owner : Traveller.account [1]
}

class Traveller{
  account : Account.owner [0..1]
}
```

Example Compiler Two: Mapping Data

Attribute-to-Table Mapping

	SINGLE-VALUED	MULTI-VALUED
PRIMITIVE-TYPED	column in *class table*	*collection table*
REFERENCE-TYPED	*association table*	

Example Transformation

```
class A {
  attributes
    s: string
    as: set(A . b) [*] }
```

```
class B {
  attributes
    is: set (int)
    b: B . as }
```

Example Compiler Two: Source Program

```
class Account {
  attributes
    owner: Traveller . account
    balance: int
}
```

```
class Traveller {
  attributes
    name: string
    reglist: set(Hotel . registered)[*]
}
```

```
class Hotel {
  attributes
    name: string
    registered: set(Traveller . reglist)[*]
  methods
    register {
        t? : extent(Traveller)
      & t? /: registered
      ==>
         registered := registered \/ {t?}
      || t?.reglist := t?.reglist \/ {this}
    }
}
```

Example Compiler Two: Target Program

Table Schemas

```
CREATE TABLE `Account`(
  `oid` INTEGER AUTO_INCREMENT, `balance` INTEGER,
  PRIMARY KEY (`oid`));
CREATE TABLE `Traveller`(
  `oid` INTEGER AUTO_INCREMENT, `name` CHAR(30),
  PRIMARY KEY (`oid`));
CREATE TABLE `Hotel`(
  `oid` INTEGER AUTO_INCREMENT, `name` CHAR(30),
  PRIMARY KEY (`oid`));
CREATE TABLE `Account_owner_Traveller_account`(
  `oid` INTEGER AUTO_INCREMENT, `owner` INTEGER, `account` INTEGER,
  PRIMARY KEY (`oid`));
CREATE TABLE `Traveller_reglist_Hotel_registered`(
  `oid` INTEGER AUTO_INCREMENT, `reglist` INTEGER, `registered` INTEGER,
  PRIMARY KEY (`oid`));
```

Stored Procedures

```
CREATE PROCEDURE `Hotel_register`(IN `this?` INTEGER, IN `t?` INTEGER)
  BEGIN
    ...
  END
```

Example Compiler Two: Path Transformation

Object Path

```
this.owner.reglist
```

Table Queries

```
SELECT (VAR 'reglist')
       (TABLE 'Hotel_registered_Traveller_reglist')
       (VAR 'registered' = (SELECT (VAR 'owner')
                                   (TABLE 'Account_owner_Traveller_account')
                                   (VAR 'owner' = VAR 'this')))
```

Account	
oid	balance
1	100

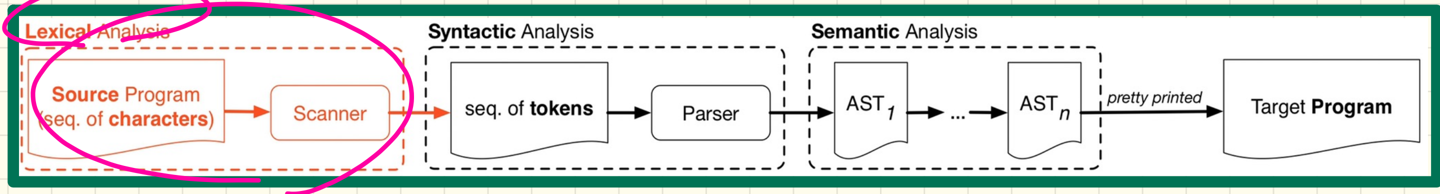
Traveller	
oid	name
2	alan
3	mark

Hotel	
oid	name
4	GLAD

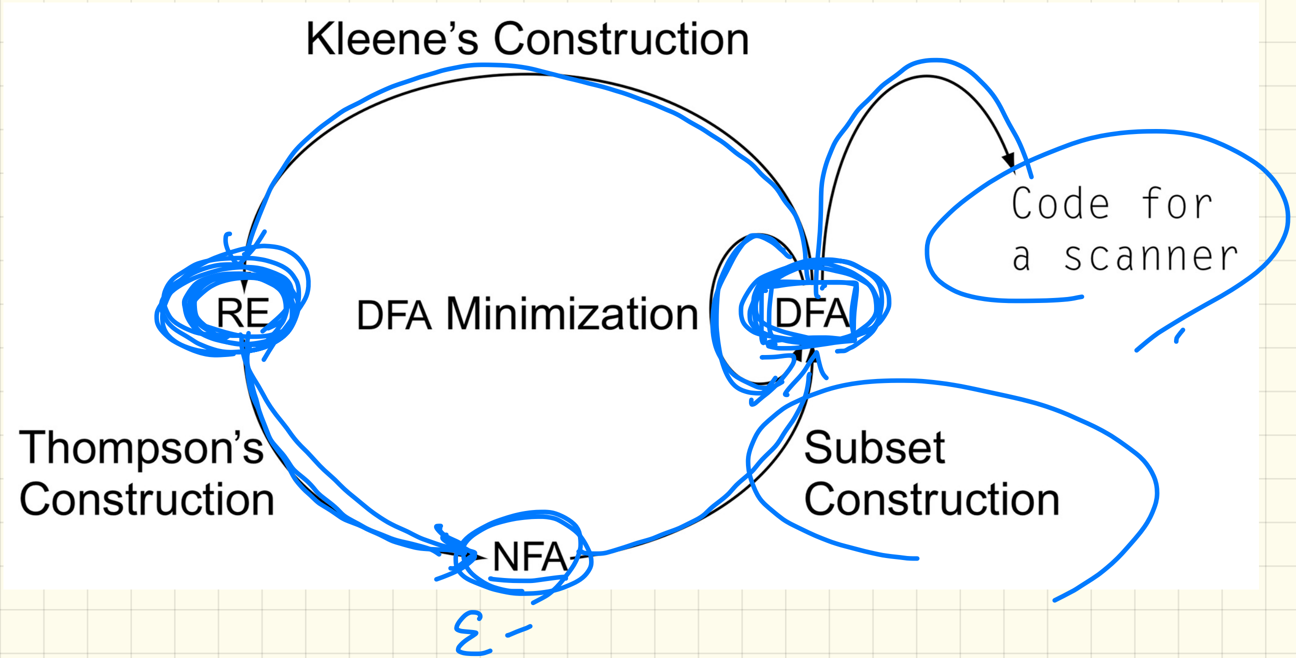
Account_owner_Traveller_account		
oid	owner	account
5	3	1

Hotel_registered_Traveller_reglist		
oid	registered	reglist
6	2	4
7	3	4

Scanner in Context



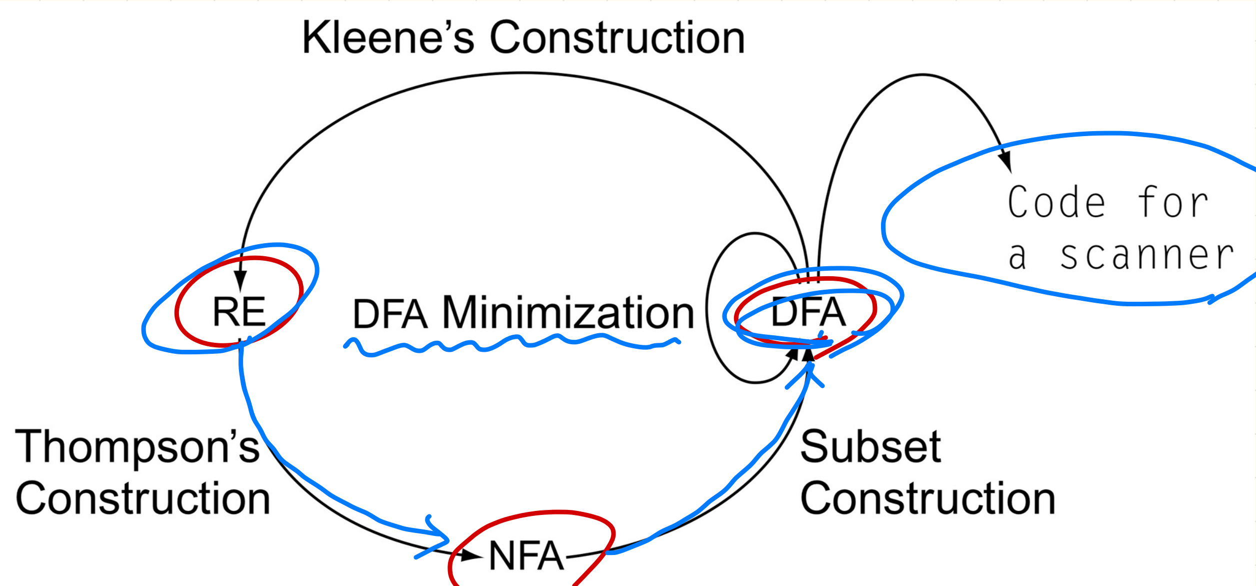
Scanner: Formulation & Implementation



Lecture 3

Monday January 13

Scanner: Formulation & Implementation



Set Comprehension

$$\{ \underbrace{2x}_{\text{term}} \mid \underbrace{1 \le x \le 10}_{\text{condition}} \}$$

$$= \{2, 4, 6, \ldots, 20\}$$

$$\Sigma_{bin}^{*} = \Sigma_{bin}^{0} \cup \Sigma_{bin}^{1} \cup \cdots$$

$$\Sigma_{bin}^{0} = \{\varepsilon\}$$

card. 1

$\varepsilon \approx$ ""

✗ $0|0|0 \in \Sigma_{bin}$

$00 \notin \Sigma_{bin}^{1}$

✓ $0|0|0 \in \Sigma_{bin}^{*}$

≠ strings of all possible lengths

$00 \in \Sigma_{bin}$ — {0, 1}

$$\Sigma_{bin} = \{0, 1\}$$

1, 0

$$\Sigma_{bin}^{1} = \{0, 1\}$$

$$\Sigma_{bin}^{2} = \{00, 01, 10, 11\}$$

alphabet

$$\Sigma_{my} = \{00, 11\}$$

single single

$00 11 \in \Sigma_{my}$ ✗

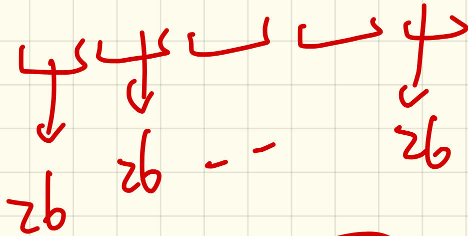
$00 \in \Sigma_{my}$ ✓

$$\Sigma_{my}^2 = \{0000, 0011, 1100, 1111\}$$

$$\underbrace{\{a \ldots z\}}_{26}{}^{5}$$

$$\{a \ldots z\}^{0}$$

$$\{\varepsilon\}$$



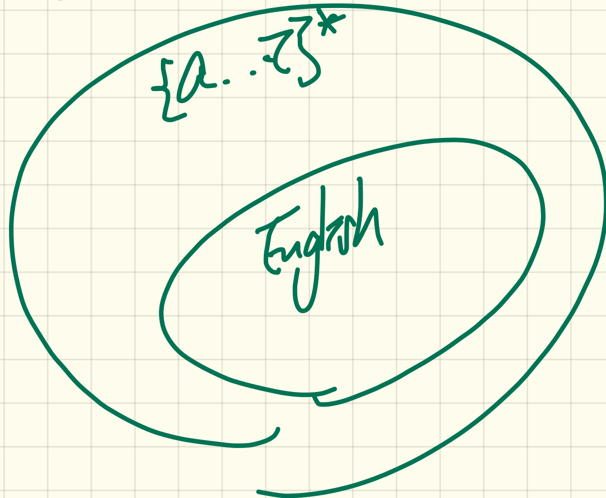
$$26^{5}.$$

$\{a..z\}^* = \text{English}$?

$aa \notin$ $\quad$ $w \leftarrow$ (L).

RL.

$aa \in$



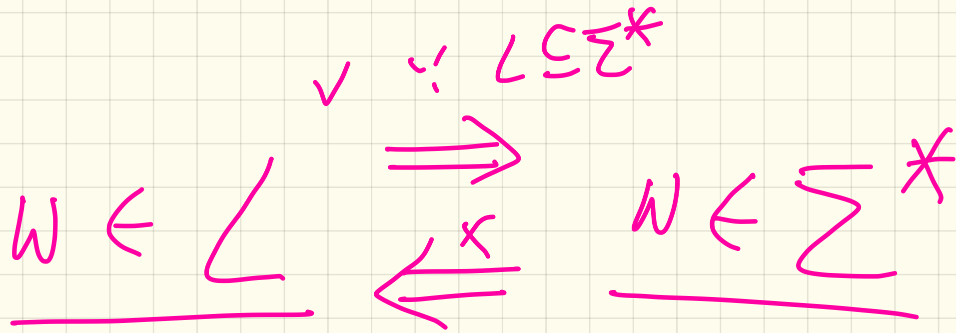
input program

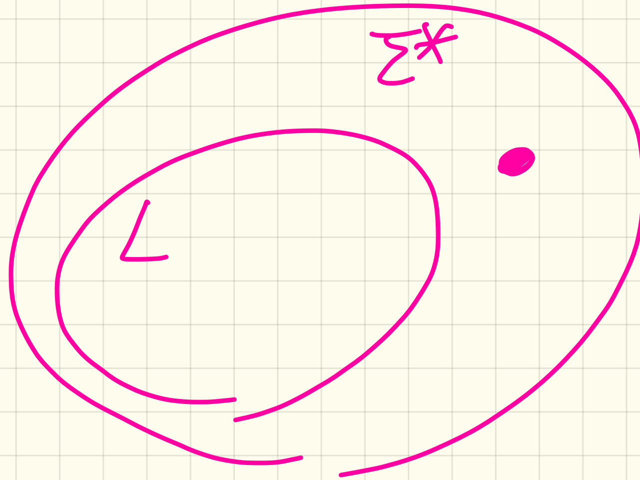
w

class A {

⋮

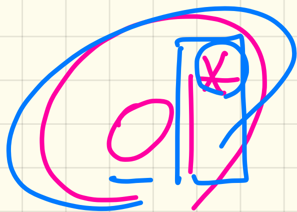
}

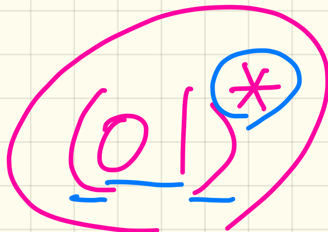

✓ ∵ L ⊆ Σ*
w ∈ L ⟹ w ∈ Σ*
⟸


Σ*
L

+

E | F





A	B
011	0101
0	ε

Regular Language Operations

Cardinalities?

1. Union

$$L \cup M = \{w \mid w \in L \vee w \in M\}$$

LM

$L = \{a, b\}$

$M = \{1, 2, 3\}$

2. Concatenation

$$LM = \{xy \mid x \in L \wedge y \in M\}$$

$|LM| = 6$

$\{a1, a2, a3, b1, b2, b3\}$

3. Kleene Closure (or Kleene Star)

$$L^* = \bigcup_{i \geq 0} L^i$$

$L^0 \cup L^1 \cup L^2 \cup L^3 \cdots$

$= L^*$

Constructions of REs

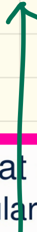
Base Case:

- Constants ϵ and $\varnothing$ are regular expressions.

$$L(\epsilon) = \{\epsilon\}$$
$$L(\varnothing) = \varnothing$$

- An input symbol $a \in \Sigma$ is a regular expression.

$$L(a) = \{a\}$$



RE op.

Recursive Case Given that E and F are regular expressions:

- The union $E + F$ is a regular expression.

$$L(E + F) = L(E) \cup L(F)$$

E | F

- The concatenation EF is a regular expression.

$$L(EF) = L(E)L(F) = \{xy \mid x \in L(E) \wedge y \in L(F)\}$$

- Kleene closure of E is a regular expression.

$$L(E^*) = (L(E))^*$$

RE operator — RL operator

- A parenthesized E is a regular expression.

$$L((E)) = L(E)$$

$$\emptyset^{*} = \emptyset^{0} \cup \emptyset^{1} \cup \emptyset^{2} \cdots$$

regular expression

$$\| $$

$$L(\emptyset) = \emptyset$$

$$= \{\varepsilon\} \cup \{x \mid x \in \emptyset\} \cup \{xy \mid x \in \emptyset \wedge y \in \emptyset\}$$

$$\{x \mid x \in \emptyset\} = \emptyset, \quad \{xy \mid x \in \emptyset \wedge y \in \emptyset\} = \emptyset$$

$$= \{\varepsilon\}.$$

$\{\emptyset\}$ vs. $\emptyset$

RE: Exercise

Write a regular expression for the following language

$$\{ w \mid w \text{ has alternating } 0\text{'s and } 1\text{'s} \}$$

$(E + F)^*$ $\quad \epsilon, \{x \mid x \in E\}, \{y \mid x \in F\}, \{xy \mid x \in E \cup F \wedge y \in E \cup F\}^*$

$(a+b)^* = \{\epsilon, a, b, aa, bb, ab, ba, \ldots\}$

$(1+\epsilon)(01)^* + (0+\epsilon)(10)^*$

101

101

101

$((1+\epsilon)(01)^* + (0+\epsilon)(10)^*)$

$1^*(01)^*$ → accepts 1101 not in language.

E1 $\neq$ E2

$$0 \mid 1^* + 1$$

$$0(1^* + 1)$$

1

Justify if $L(E1) = L(E2)$

RE: Operator Precedence

$10^* \text{ vs. } (10)^*$

$01^* + 1 \text{ vs. } 0(1^* + 1)$

$0 + 1^* \text{ vs. } (0 + 1)^*$

DFA: Exercise

0 1 0 1

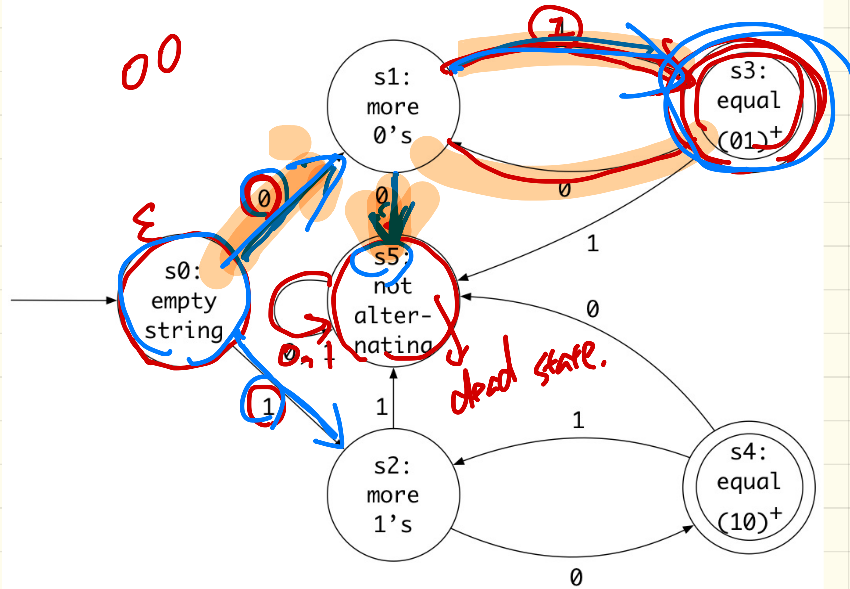
The *transition diagram* below defines a DFA which *accepts* exactly the language

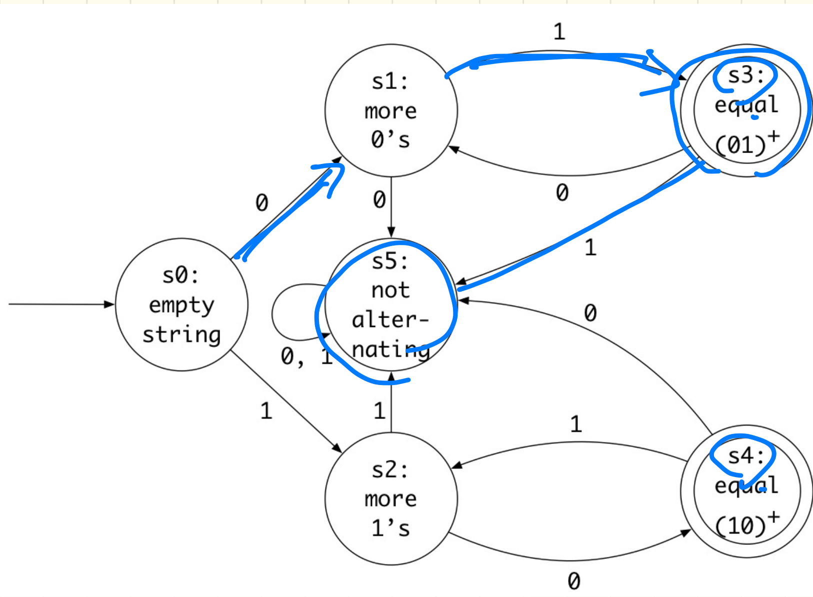
$$\left\{ w \mid \begin{array}{l} w \neq \epsilon \\ \wedge \; w \texttt{ has equal \# of alternating 0's and 1's} \end{array} \right\}$$

0 1 0 1 ✓

0 1 0 1 0 X

transitions $= |\Sigma_{bin}| = |\{0, 1\}|$





01 ✓

011 ✗

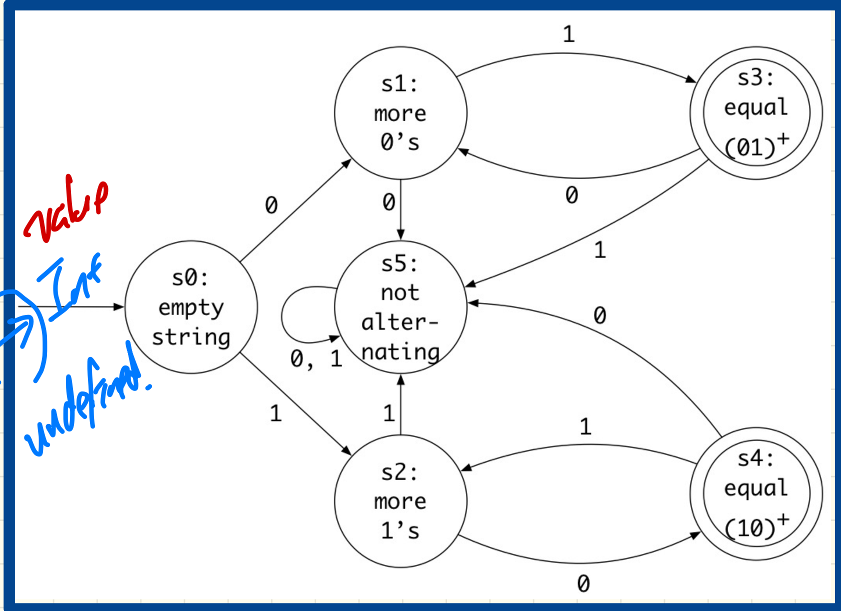
DFA: Formulation (1)

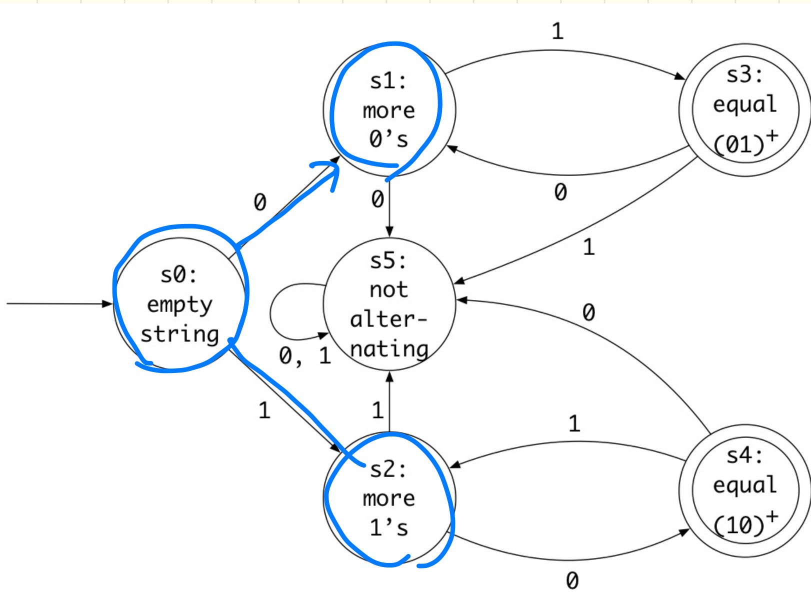
A *deterministic finite automata (DFA)* is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

Language of a DFA

$$L(M) = \left\{ \begin{array}{l} a_1 a_2 \ldots a_n \mid \\ \quad 1 \leq i \leq n \ \wedge \ a_i \in \Sigma \ \wedge \ \delta(q_{i-1}, a_i) = q_i \ \wedge \ q_n \in F \end{array} \right\}$$





δ

C.S. \ Input	0	1
s_0	s_1	s_2
s_1		
s_2		
s_3		
s_4		
s_5		

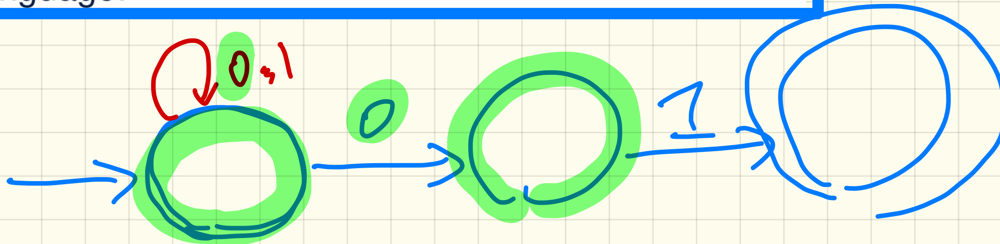
DFA vs. NFA

Problem: Design a DFA that accepts the following language:

$$L = \{\, x01 \mid x \in \{0,1\}^* \,\}$$

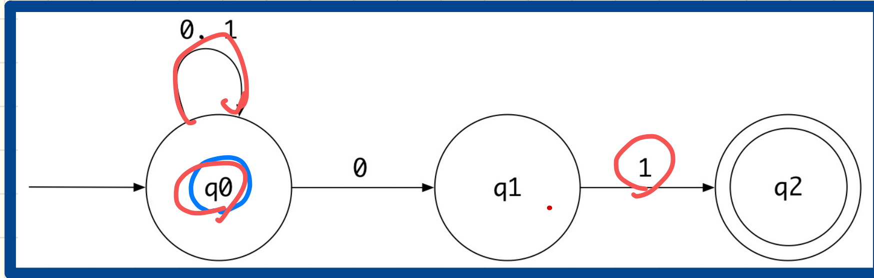
That is, L is the set of strings of 0s and 1s ending with 01.

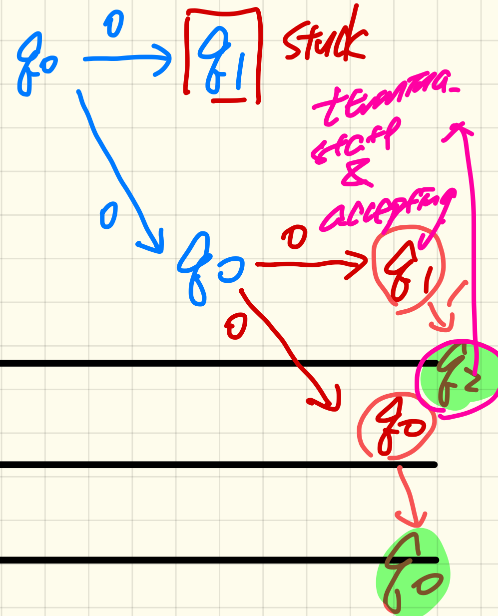
A *non-deterministic finite automata (NFA)* that accepts the same language:



NFA: Processing Strings

How an NFA determines if an input *00101* should be processed:





- **Read 0:**
- **Read 0:**
- **Read 1:**
- **Read 0:**
- **Read 1:**

LECTURE 4

WEDNESDAY JANUARY 15

Quiz 1: Wednesday
Jan. 21

Office Hours: Friday
2:30pm ~ 3:30pm

DFA: Formulation (1)

A *deterministic finite automata (DFA)* is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

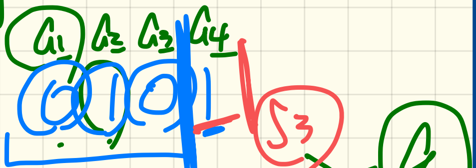
Language of a DFA

$$L(M) = \left\{ a_1 a_2 \dots a_n \;\middle|\; 1 \le i \le n \wedge a_i \in \Sigma \wedge \delta(q_{i-1}, a_i) = q_i \wedge q_n \in F \right\}$$

DFA languages

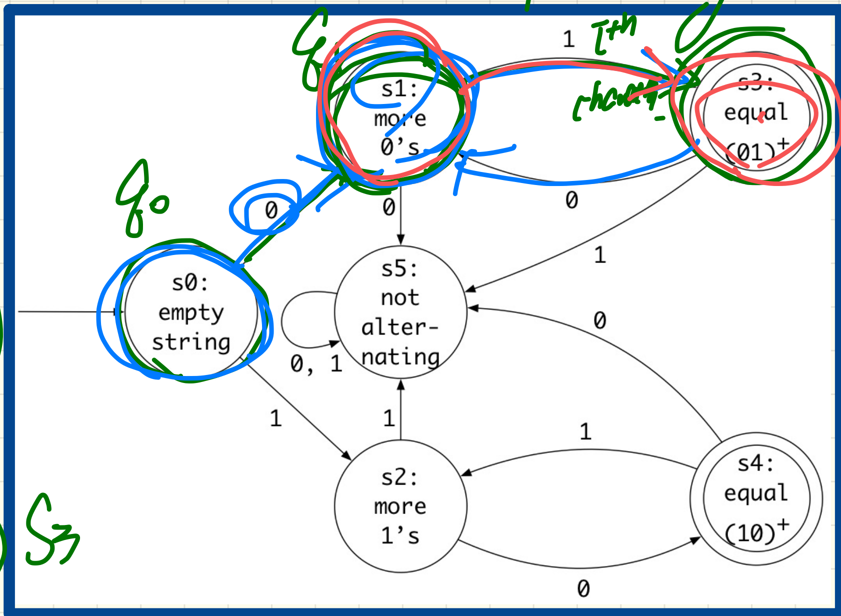
length

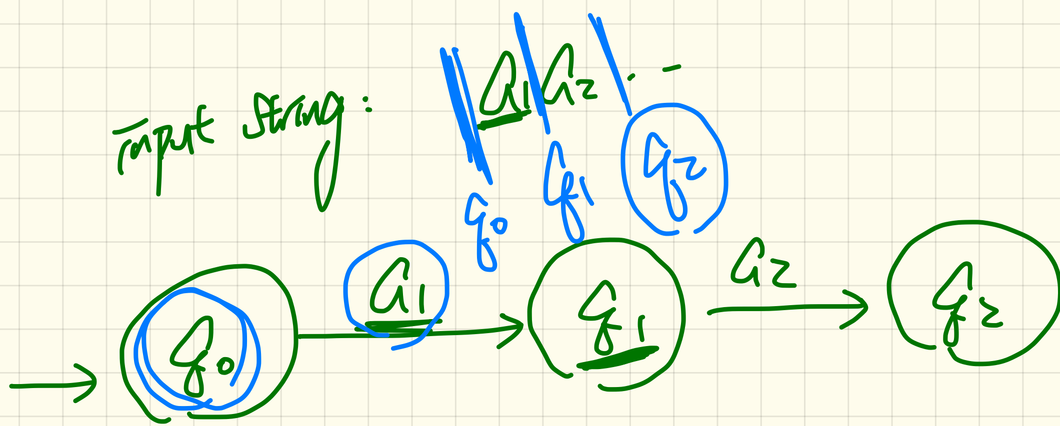
q_i the resulting state after reading the ith char



$i=1$ $\delta(q_0, a_1) = q_1$ (s1)

$i=2$ $\delta(q_1, a_2) = q_2$ (s3)





$$\delta(q_0, a_1) = q_1$$

DFA: Formulation (2)

A *deterministic finite automata (DFA)* is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

Language of a DFA

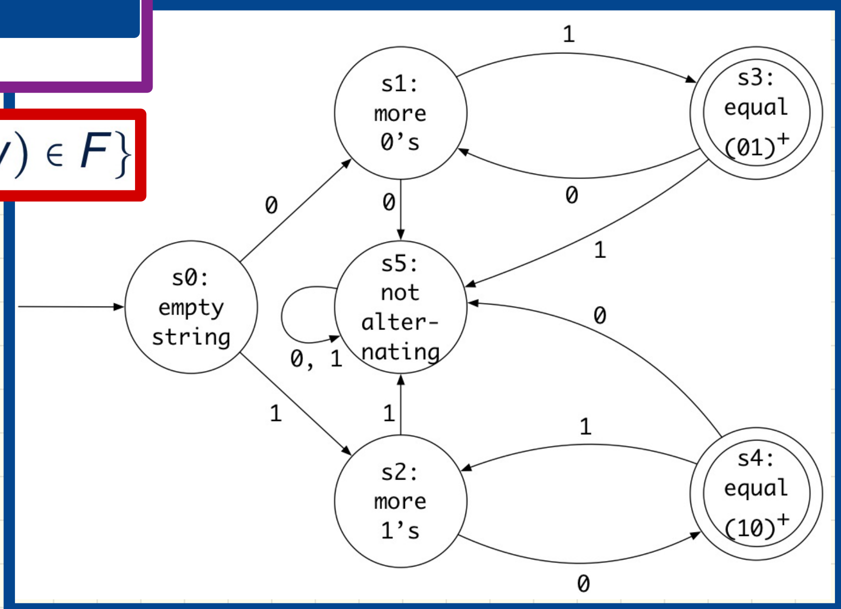
$$\hat{\delta} : (Q \times \Sigma^*) \rightarrow Q$$

We may define $\hat{\delta}$ recursively, using δ!

$$\hat{\delta}(q, \epsilon) =$$
$$\hat{\delta}(q, xa) =$$

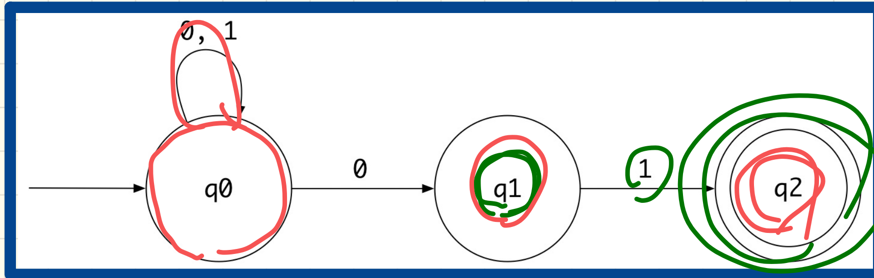
where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

$$L(M) = \{w \mid w \in \Sigma^* \land \hat{\delta}(q_0, w) \in F\}$$



NFA: Processing Strings

How an NFA determines if an input *00101* should be processed:



- **Read 0:** $\delta(q_0, 0) = \{q_0, q_1\}$
- **Read 0:** $\delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\}$
- **Read 1:** $\delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_2\}$
- **Read 0:** $\delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\}$
- **Read 1:** $\delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_2\}$ ✓

NFA: Formulation

A *nondeterministic finite automata (NFA)* is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

Language of a NFA

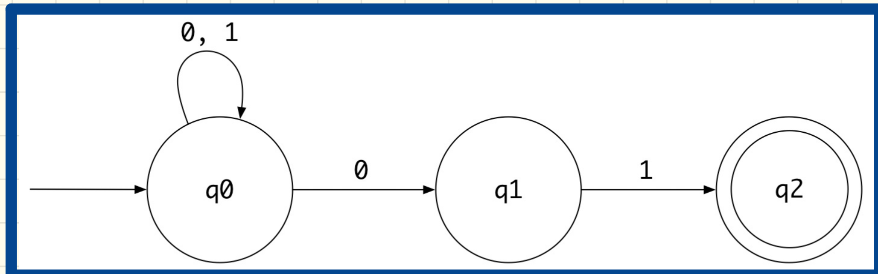
$$\hat{\delta} : (Q \times \Sigma^*) \rightarrow \mathbb{P}(Q)$$

We may define $\hat{\delta}$ recursively, using δ!

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= \{q\} \\ \hat{\delta}(q, xa) &= \bigcup\{\delta(q', a) \mid q' \in \hat{\delta}(q, x)\} \end{aligned}$$

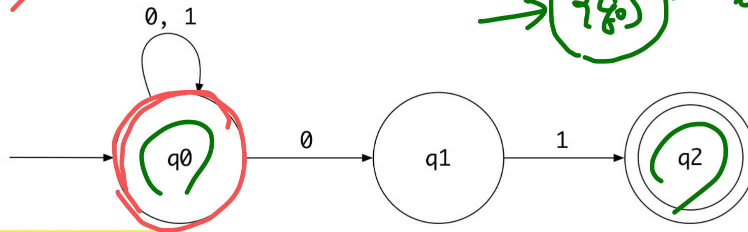
where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

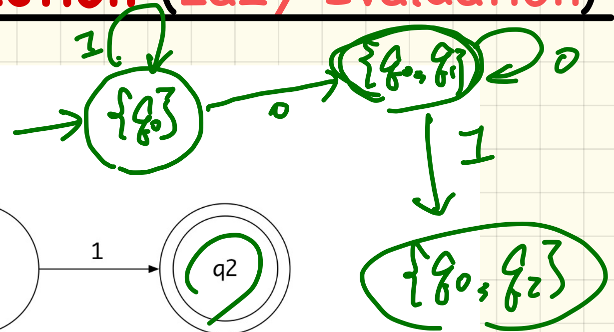
$$L(M) = \{w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$



NFA to DFA: Subset Construction (Lazy Evaluation)

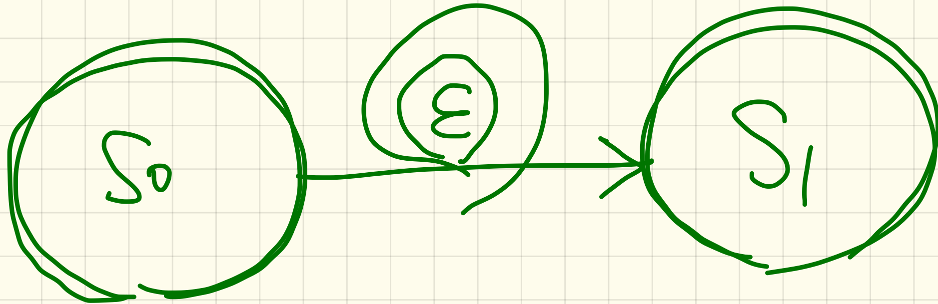
Given an *NFA*:





Subset construction (with *lazy evaluation*) produces a *DFA* transition table:

state \ input	0	1
$\{q_0\}$	$\delta(q_0, 0) = \{q_0, q_1\}$	$\delta(q_0, 1) = \{q_0\}$
$\{q_0, q_1\}$	$\delta(q_0, 0) \cup \delta(q_1, 0)$ ($\delta(q_1, 0) = \emptyset$) $= \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_2\}$

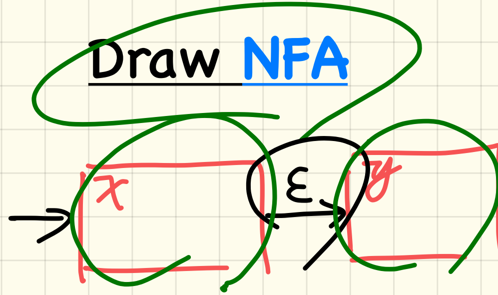


$\{S_0, S_1\}$.

epsilon-NFA: Motivation

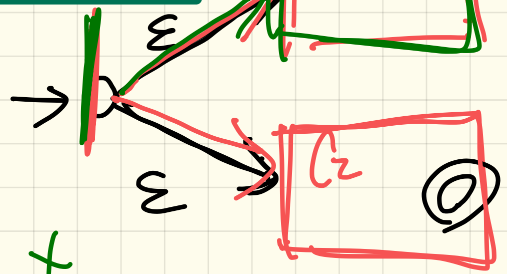
Draw NFA

$$\left\{ xy \;\middle|\; \begin{array}{l} x \in \{0,1\}^* \\ \wedge\; y \in \{0,1\}^* \\ \wedge\; x \text{ has alternating 0's and 1's} \\ \wedge\; y \text{ has an odd \# 0's and an odd \# 1's} \end{array} \right\}$$

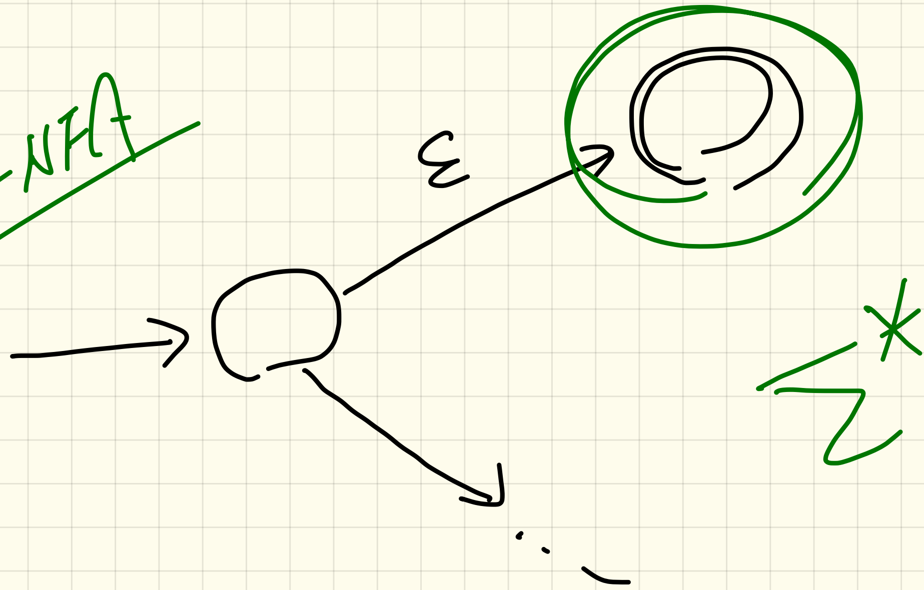


$$\left\{ w : \{0,1\}^* \;\middle|\; \begin{array}{l} w \text{ has alternating 0's and 1's} \\ \vee\; w \text{ has an odd \# 0's and an odd \# 1's} \end{array} \right\}$$

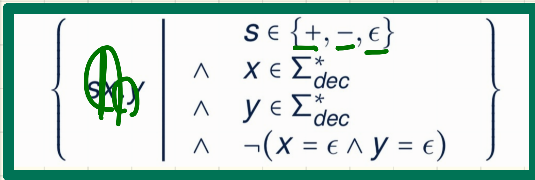
$$\left\{ sx.y \;\middle|\; \begin{array}{l} s \in \{+,-,\epsilon\} \\ \wedge\; x \in \Sigma_{dec}^* \\ \wedge\; y \in \Sigma_{dec}^* \\ \wedge\; \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$



ε-NFA



epsilon-NFA: Example



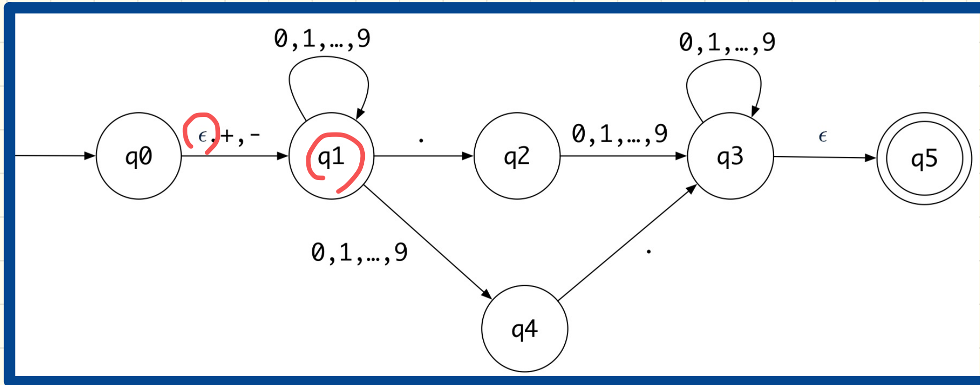
$$\left\{ sxy \;\middle|\; \begin{array}{ll} & s \in \{+, -, \epsilon\} \\ \wedge & x \in \Sigma_{dec}^{*} \\ \wedge & y \in \Sigma_{dec}^{*} \\ \wedge & \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$

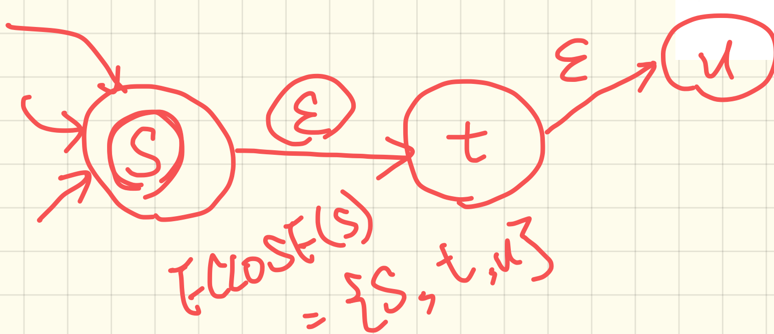
epsilon-NFA: Formulation (1)

An ϵ-*NFA* is a 5-tuple

$$M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

Example epsilon-NFA





Draw a transition table for the above NFA's δ function:

	ϵ	+, -	.	0 .. 9
q_0	$\{q_1\}$			
q_1	$\emptyset$			
q_2	$\emptyset$			
⋮				
q_5				

epsilon-NFA: Formulation (2)

An *ϵ-NFA* is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

we define the *epsilon closure* (or *ϵ-closure*) as a function

$$\text{ECLOSE} : Q \to \mathbb{P}(Q)$$

set of states

For any state $q \in Q$

$$\text{ECLOSE}(q) = \{q\} \cup \bigcup_{p \in \delta(q, \epsilon)} \text{ECLOSE}(p)$$

union the ECLOSE of all states reachable using ϵ from q

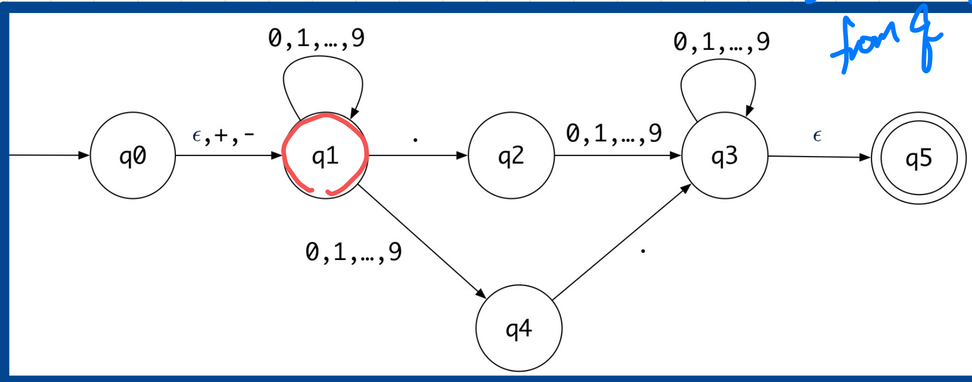
ECLOSE(q0) ?

$\text{ECLOSE}(q_0) = \{q_0\} \cup \text{ECLOSE}(q_1)$

$\{q_1\} \cup \emptyset$

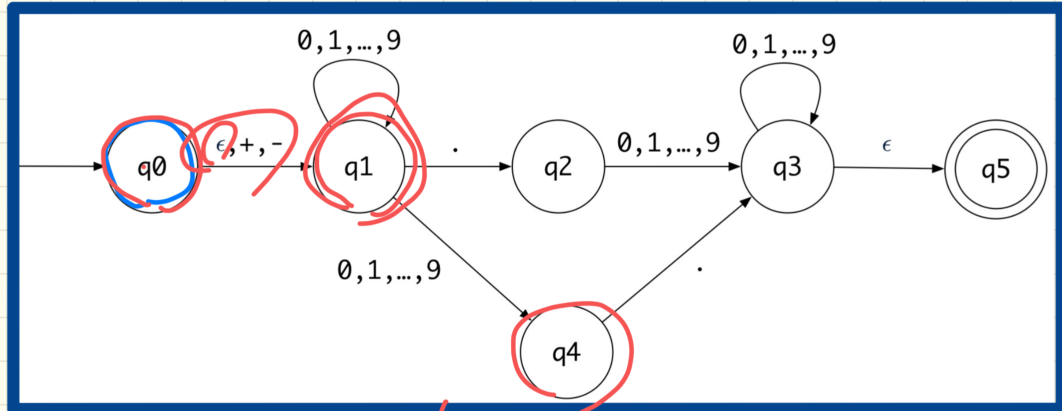
$= \{q_0, q_1\}$

Example epsilon-NFA



epsilon-NFA: Processing Strings

How an epsilon-NFA determines if input 5.6 should be processed



5.6

Starting state: ECLOSE(q_0) = {q_0, q_1}

Read 5	$\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$ → ECLOSE(q_1) ∪ ECLOSE(q_4)
Read .	
Read 6	

epsilon-NFA: Formulation (3)

An ϵ-*NFA* is a 5-tuple

$$M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

Language of a epsilon-NFA

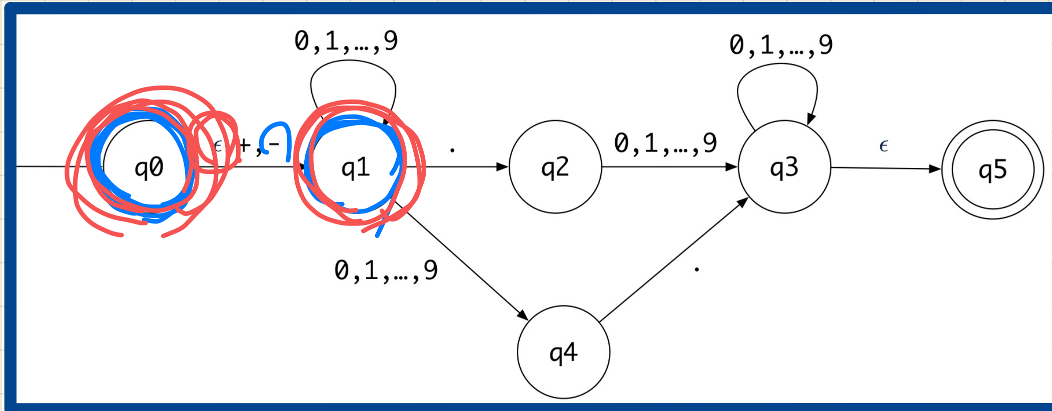
$$\hat{\delta} : (Q \times \Sigma^*) \to \mathbb{P}(Q)$$

We may define $\hat{\delta}$ recursively, using δ!

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= \text{ECLOSE}(q) \\ \hat{\delta}(q, xa) &= \bigcup \{ \quad \mid q'' \in \delta(q', a) \wedge q' \in \hat{\delta}(q, x) \ \} \end{aligned}$$

$$L(M) = \{ w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \cap F \neq \varnothing \}$$

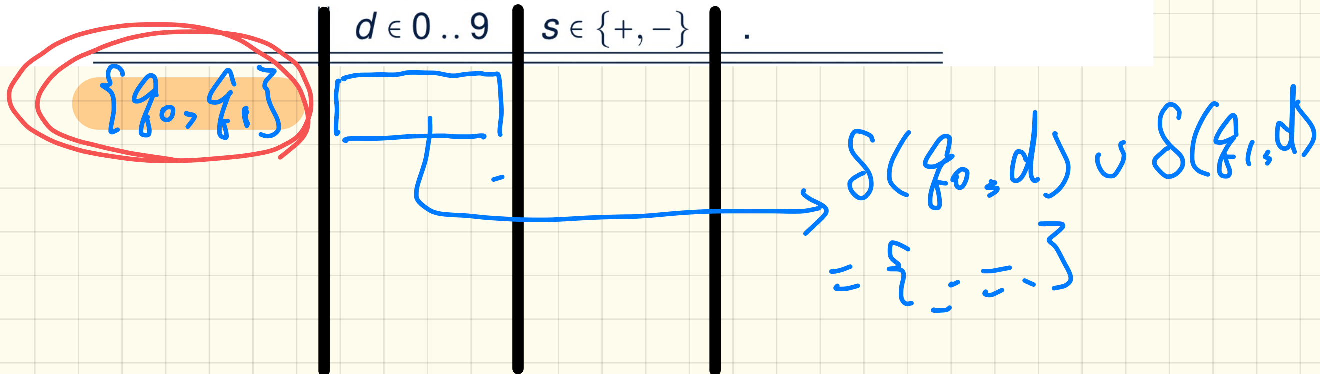
epsilon-NFA to DFA: Subset Construction



ECLOSE(q0) ?

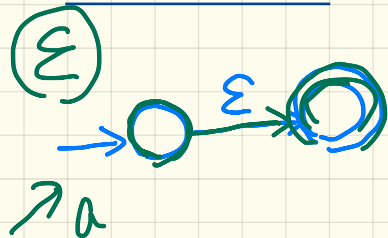
0 1

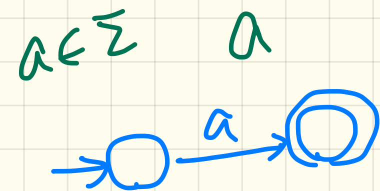
Subset construction (with *lazy evaluation* and *epsilon closures*) produces a *DFA* transition table.

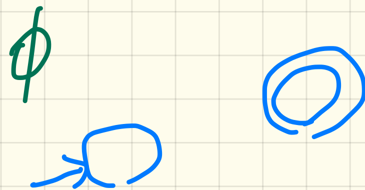


Regular Expression to epsilon-NFA

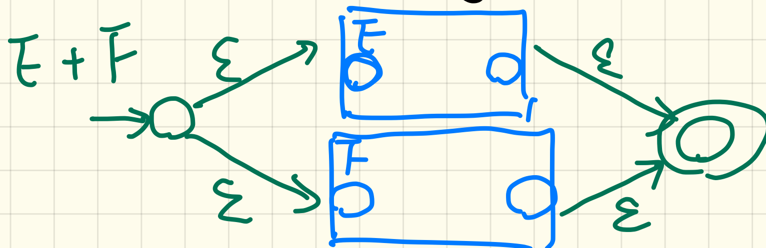
Base Cases

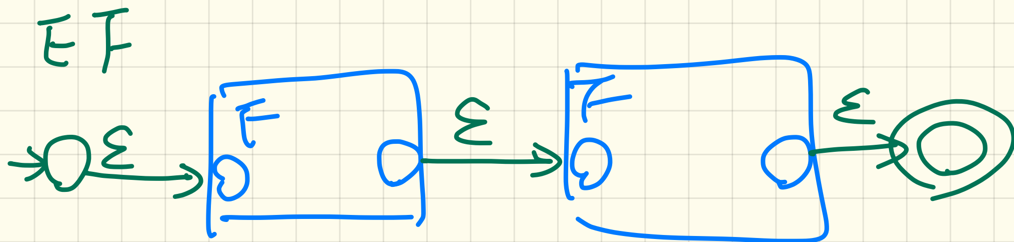


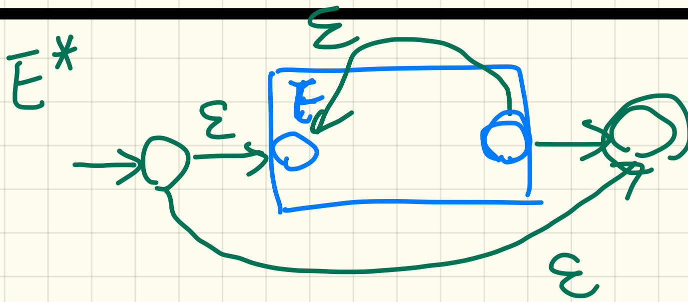


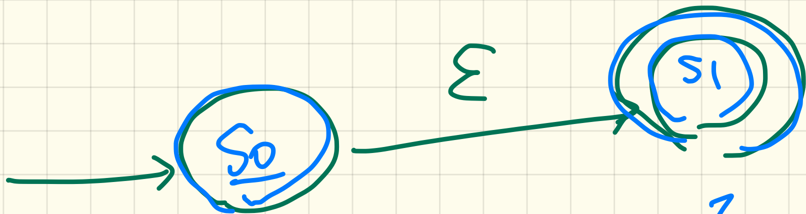


Recursive Cases (given REs E and F)







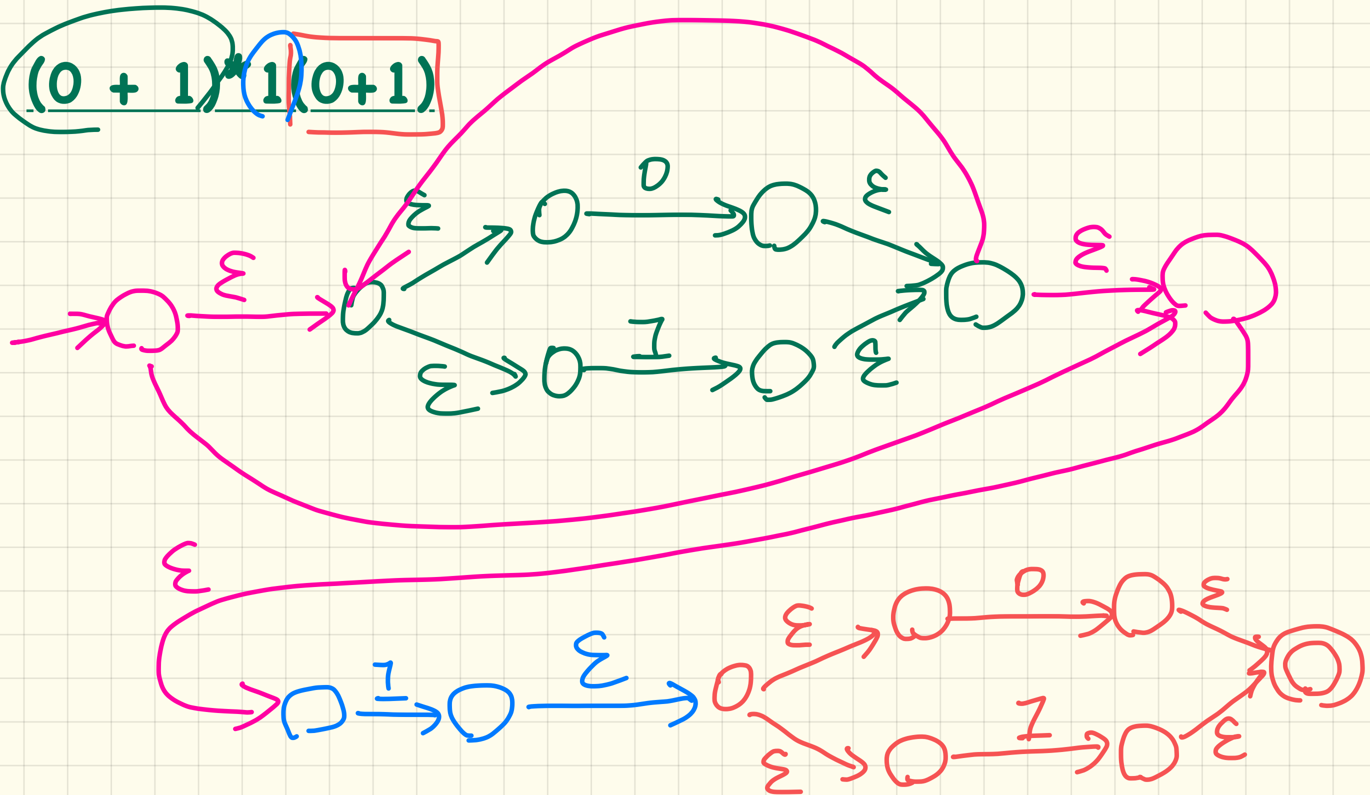


a

$ECLOSE(s_0) = \{s_0, s_1\}$

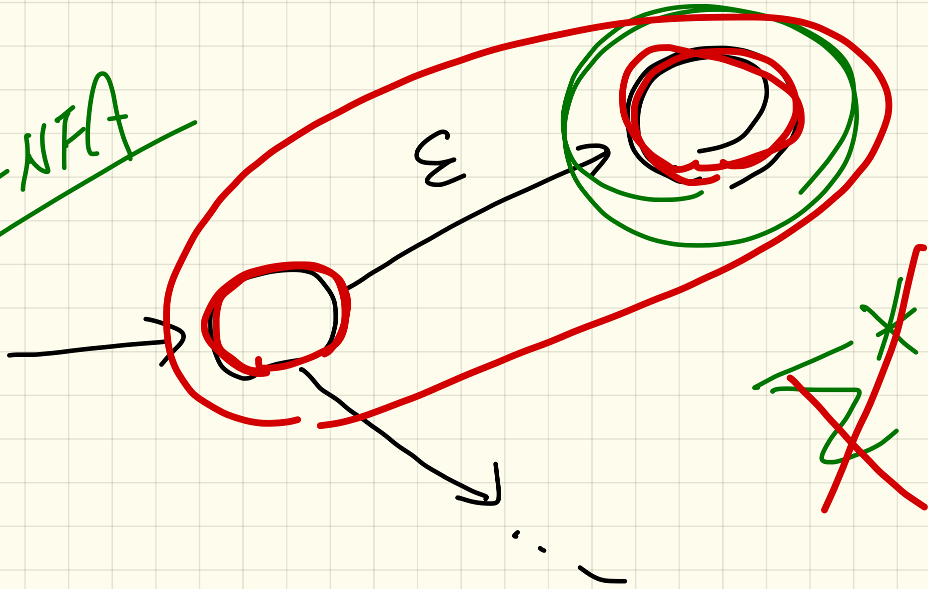
$\underbrace{\delta(s_0, a)}_{\phi} \cup \underbrace{\delta(s_1, a)}_{\phi} = \phi$

Regular Expression to epsilon-NFA: Example



LECTURE 5

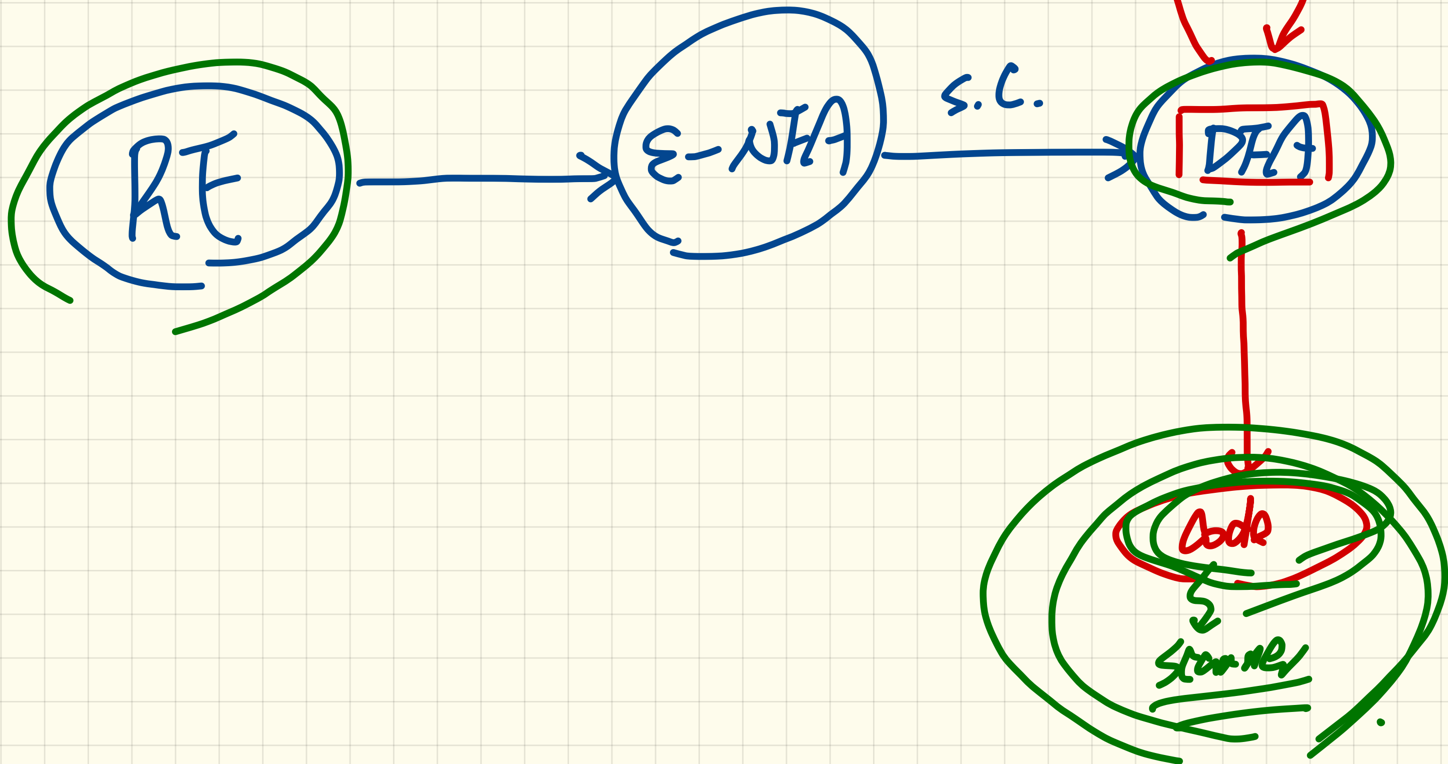
MONDAY JANUARY 20



$\{\varepsilon\}$

ab

$\emptyset$ $\emptyset$

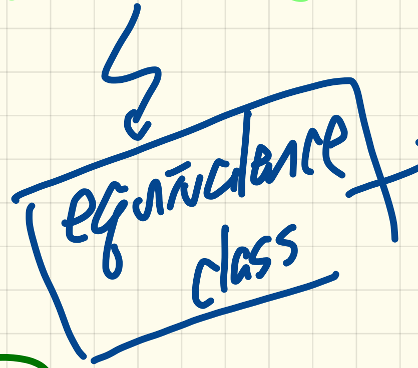


RE
ε-NFA
s.c.
DFA
minimize
process
code
scanner

Input

$Q = \{ s_0, s_1, s_2, s_3 \}$

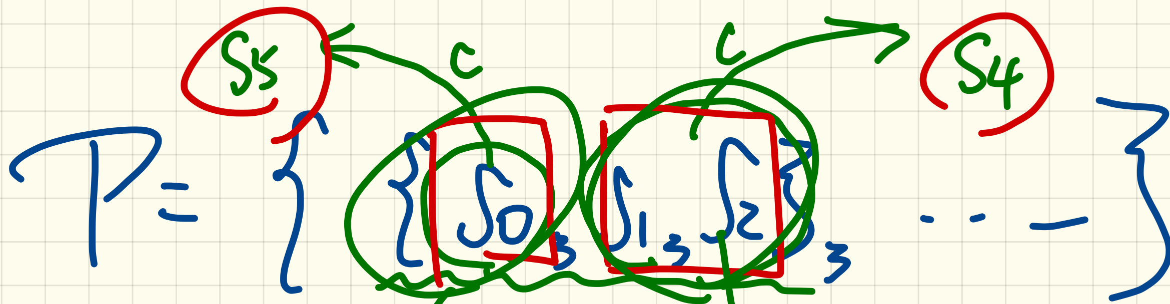
$P = \{ \{ s_0, s_1 \}, \{ s_2, s_3 \} \}$



→ each state in this set is considered as equivalent

$Q = \{ \{ s_0 \}, \{ s_1 \}, \{ s_2 \}, \{ s_3 \} \}$

the input is already init.



$p \in P$.

$c \in \Sigma$

S

$P - S$

maximal set to split out

Minimizing DFA: Algorithm

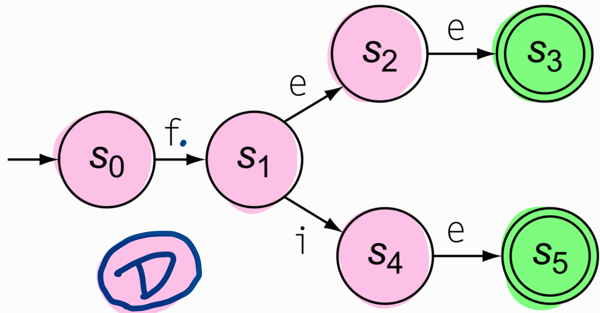
```
ALGORITHM: MinimizeDFAStates
  INPUT: DFA M = (Q, Σ, δ, q0, F)
  OUTPUT: M' s.t. minimum |Q| and equivalent behaviour as M
PROCEDURE:
  P := ∅ /* refined partition so far */
  T := { F, Q - F } /* last refined partition */
  while (P ≠ T):
    P := T
    T := ∅
    for(p ∈ P s.t. |p| > 1):
      find the maximal S ⊆ p s.t. splittable(p, S)
      if S ≠ ∅ then
        T := T ∪ {S, p - S}
      else
        T := T ∪ {p}
      end
```

splittable(p, S) holds iff there is $c \in \Sigma$ s.t.

- Transition c leads all $s \in S$ to states in the ***same partition*** $p1$.
- Transition c leads some $s \in p - S$ to a ***different partition*** $p2$ ($p2 \neq p1$).

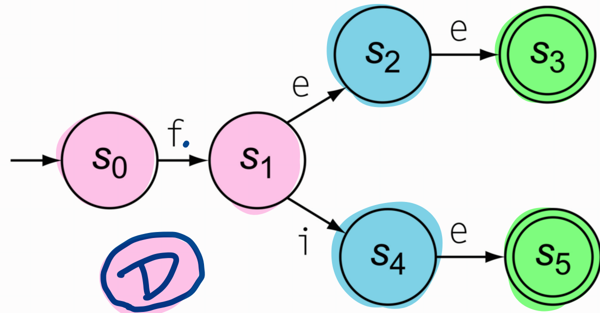
Minimizing DFA: Example (1)

fee | fie

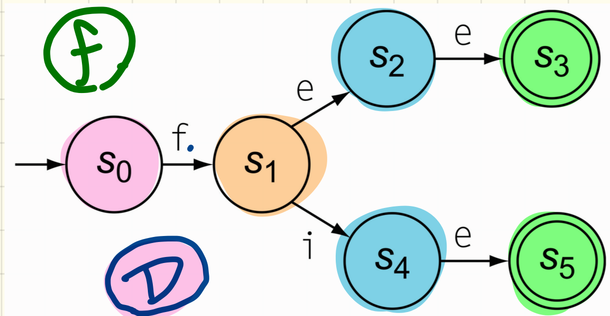


f ✓] $\forall s \in$ [ ] $\cdot\ s \xrightarrow{c}$ [ ]

e × $\{ \{s_0, s_1\}, \{s_2, s_4\}, \{s_3, s_5\} \}$

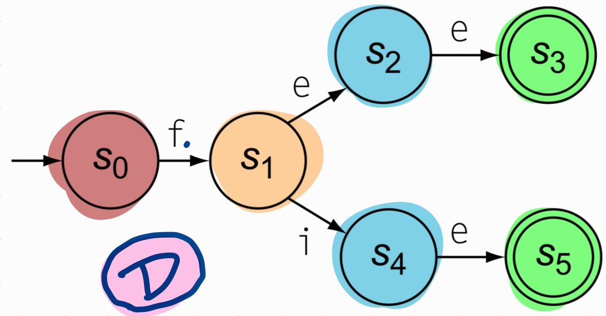


e × $\{ \{s_0\}, \{s_1\}, \{s_2, s_4\}, \{s_3, s_5\} \}$

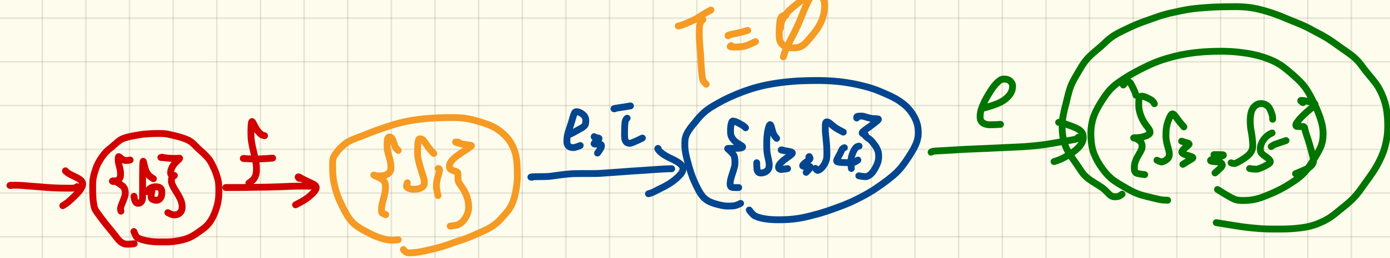


T

$$\{ \{s_0\}, \{s_1\}, \{s_2, s_4\}, \{s_3, s_5\} \}$$

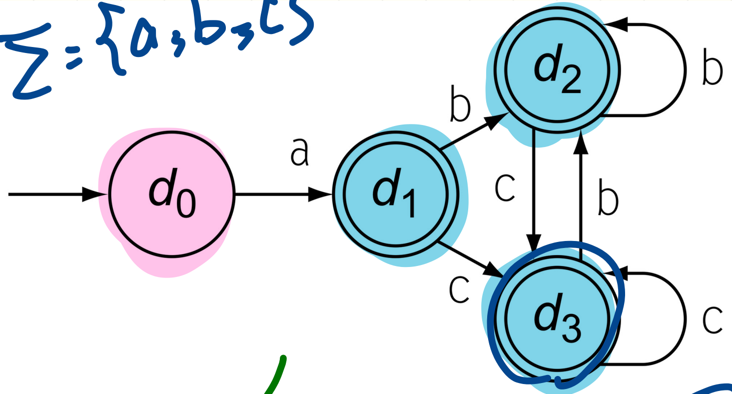


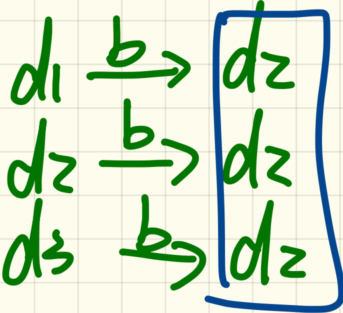
$P = T$

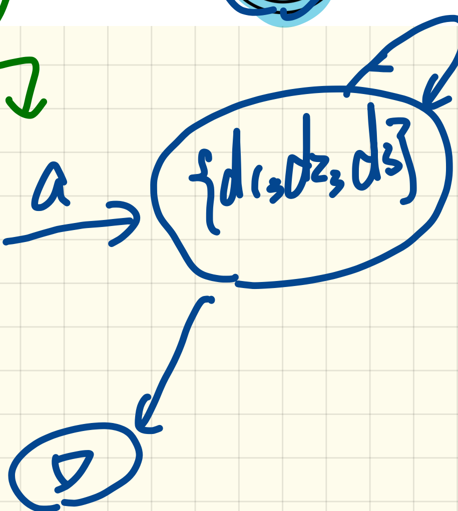


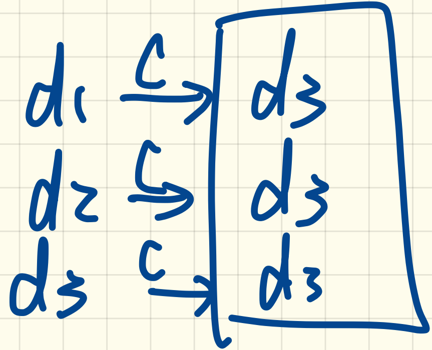
D

Minimizing DFA: Example (2)

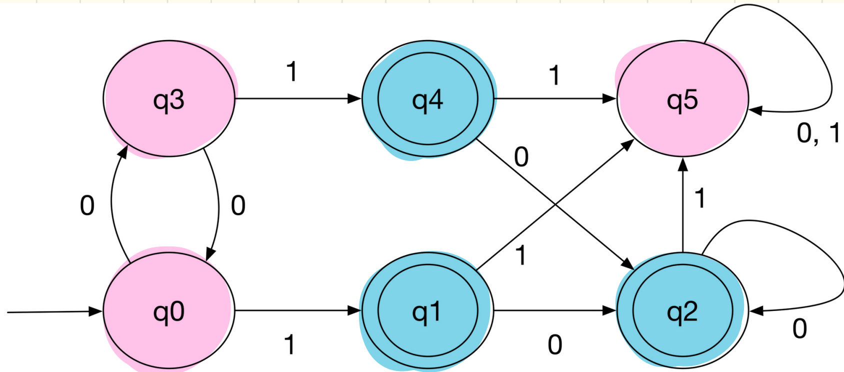


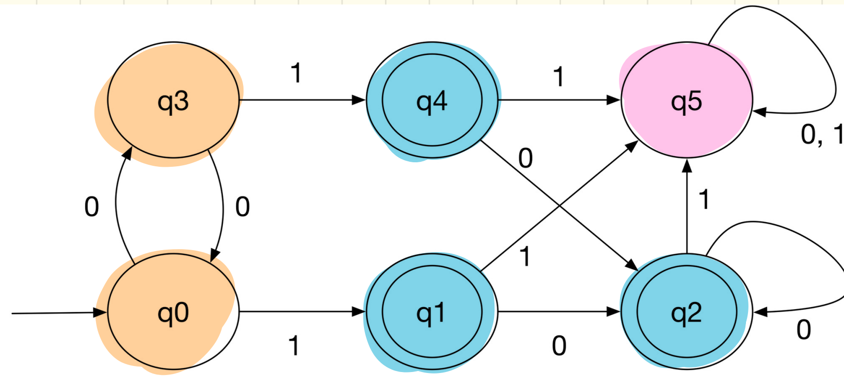


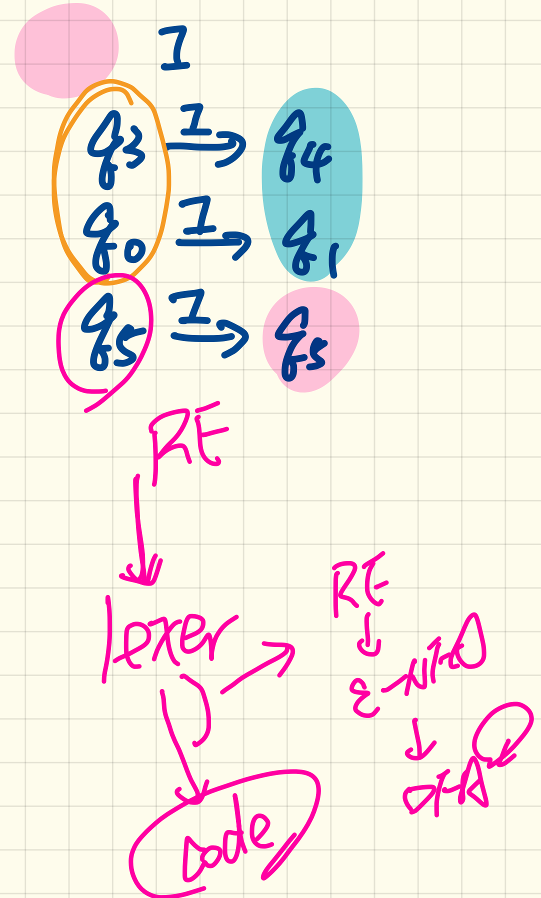




Minimizing DFA: Example (3)







while

identifier

while

Specification

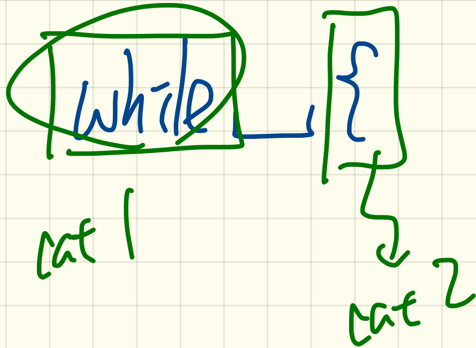
while

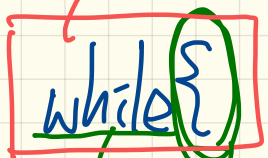
re. for identifier

while␣

while12

"␣␣while␣␣"





does not belong to any cat.

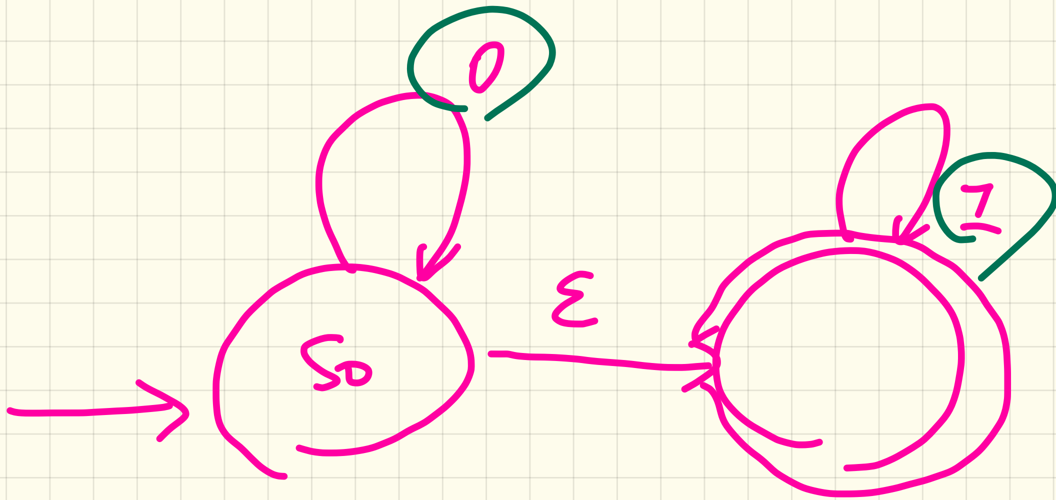
cat 1

while{

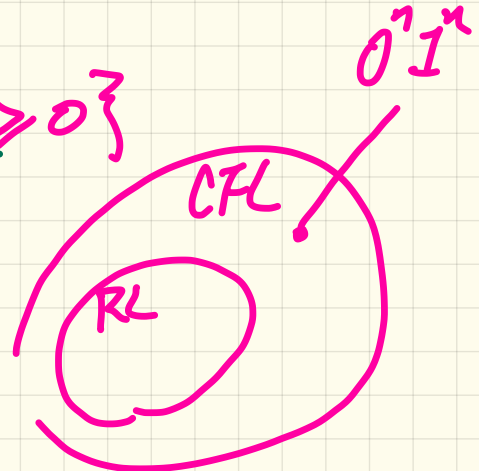
already recognizes "while" as a valid token, if we go on to include {

"while{" → no category → roll back.

Lecture 6
Wednesday January 22



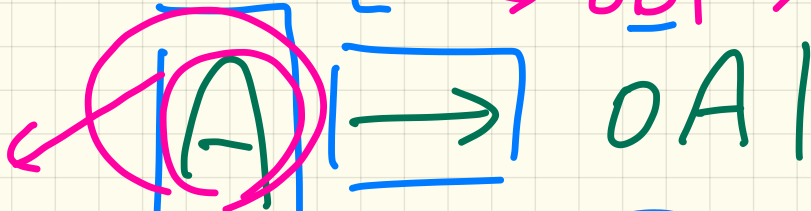
$$\{0^n 1^n \mid n \geq 0\}$$

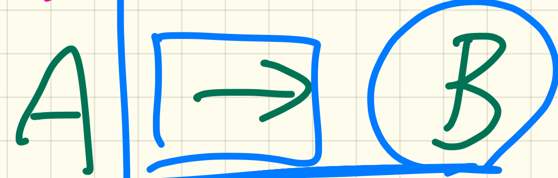


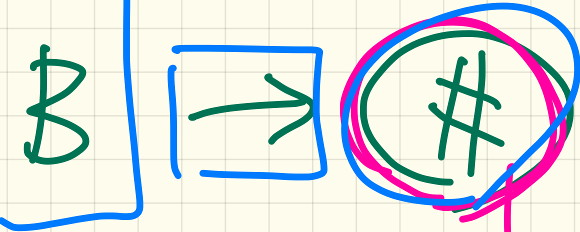
derivation

$$A \Rightarrow 0A1 \Rightarrow 0B1 \Rightarrow 0\#1$$

Start



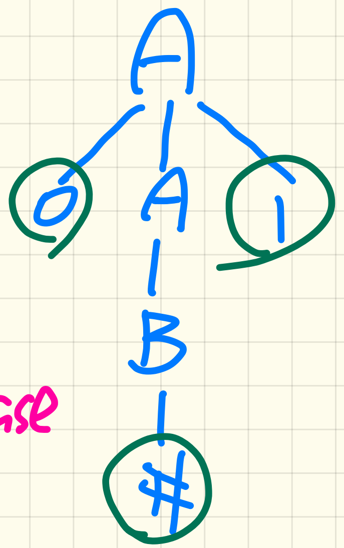


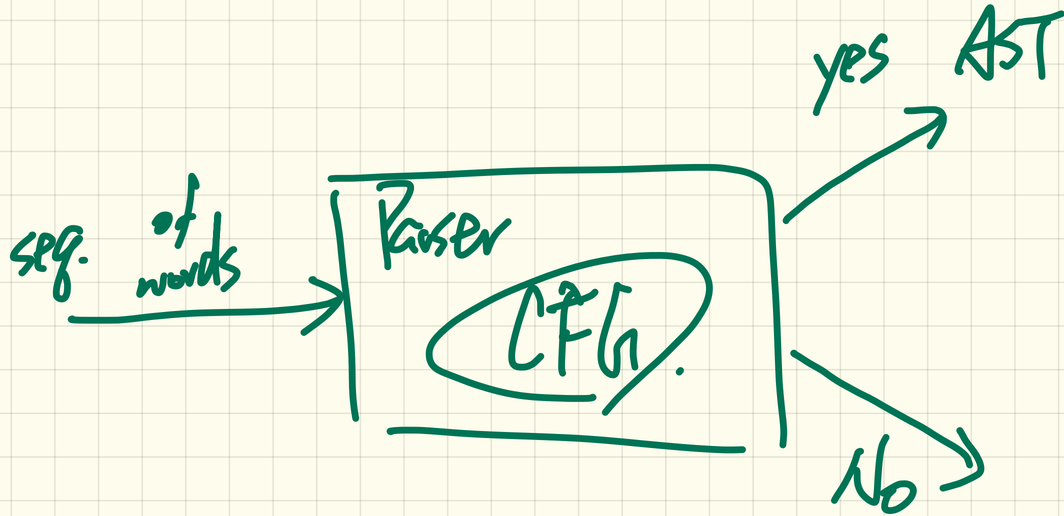


Variables (non-terminals)

↳ recursive cases

base case





if (true {

L_1

$$\{w \mid w \text{ is palindrome}\} =$$

$$\{ww^R \mid w \in \Sigma^*\} \quad L_2$$

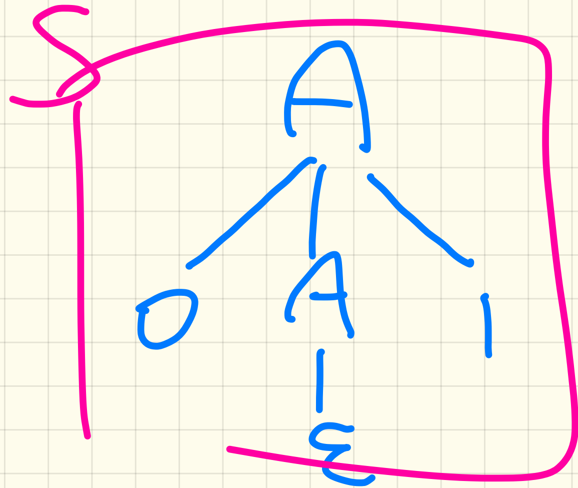
$1 \in L_1$

$1 \notin L_2$

A A A

$A \rightarrow 0 \mid$

unnecessary.

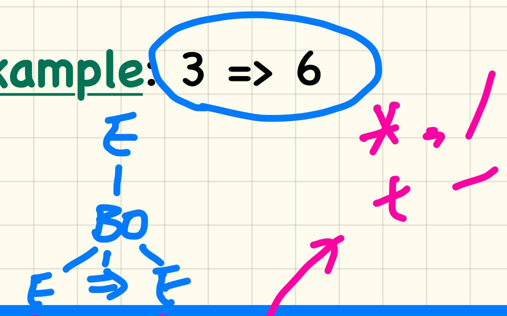


true + false

$3 \Rightarrow 4$

Context-Free Grammar (CFG): Example Version 1

Expression	→	*IntegerConstant*
	|	*BooleanConstant*
	|	*BinaryOp*
	|	*UnaryOp*
	|	(*Expression*)
IntegerConstant	→	*Digit*
	|	*Digit IntegerConstant*
	|	−*IntegerConstant*
Digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
BooleanConstant	→	TRUE
	|	FALSE



Example: 3 => 6

BinaryOp	→	*Expression* + *Expression*
	|	*Expression* − *Expression*
	|	*Expression* * *Expression*
	|	*Expression* / *Expression*
	|	*Expression* && *Expression*
	|	*Expression* || *Expression*
	|	*Expression* => *Expression*
	|	*Expression* == *Expression*
	|	*Expression* /= *Expression*
	|	*Expression* > *Expression*
	|	*Expression* < *Expression*
UnaryOp	→	! *Expression*

Example: 3 * 5 + 4

ambiguous

exercise: draw all possible parse trees for it.

Context-Free Grammar (CFG): Example Version 1

$$
\begin{array}{lcl}
\textit{Expression} & \rightarrow & \textit{IntegerConstant} \\
 & | & \textit{BooleanConstant} \\
 & | & \textit{BinaryOp} \\
 & | & \textit{UnaryOp} \\
 & | & (\ \textit{Expression}\ ) \\
\\
\textit{IntegerConstant} & \rightarrow & \textit{Digit} \\
 & | & \textit{Digit IntegerConstant} \\
 & | & -\textit{IntegerConstant} \\
\\
\textit{Digit} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\\
\textit{BooleanConstant} & \rightarrow & \texttt{TRUE} \\
 & | & \texttt{FALSE}
\end{array}
$$

Example: 3 * 5 + 4

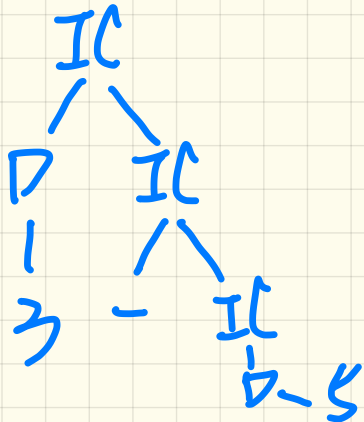
$$
\begin{array}{lcl}
\textit{BinaryOp} & \rightarrow & \textit{Expression} + \textit{Expression} \\
 & | & \textit{Expression} - \textit{Expression} \\
 & | & \textit{Expression} * \textit{Expression} \\
 & | & \textit{Expression} \ / \ \textit{Expression} \\
 & | & \textit{Expression} \ \&\& \ \textit{Expression} \\
 & | & \textit{Expression} \ || \ \textit{Expression} \\
 & | & \textit{Expression} => \textit{Expression} \\
 & | & \textit{Expression} == \textit{Expression} \\
 & | & \textit{Expression} \ /= \ \textit{Expression} \\
 & | & \textit{Expression} > \textit{Expression} \\
 & | & \textit{Expression} < \textit{Expression} \\
\\
\textit{UnaryOp} & \rightarrow & !\ \textit{Expression}
\end{array}
$$

Context-Free Grammar (CFG): Example Version 2

```
Expression        →  ArithmeticOp
                  |  RelationalOp
                  |  LogicalOp
                  |  ( Expression )

IntegerConstant   →  Digit
                  |  Digit IntegerConstant
                  |  -IntegerConstant

Digit             →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

BooleanConstant   →  TRUE
                  |  FALSE
```

Example: (1 + 2) => (5 / 4)

```
ArithmeticOp  →  ArithmeticOp + ArithmeticOp
              |  ArithmeticOp − ArithmeticOp
              |  ArithmeticOp * ArithmeticOp
              |  ArithmeticOp / ArithmeticOp
              |  (ArithmeticOp)
              |  IntegerConstant
RelationalOp  →  ArithmeticOp == ArithmeticOp
              |  ArithmeticOp /= ArithmeticOp
              |  ArithmeticOp > ArithmeticOp
              |  ArithmeticOp < ArithmeticOp
LogicalOp     →  LogicalOp && LogicalOp
              |  LogicalOp || LogicalOp
              |  LogicalOp => LogicalOp
              |  ! LogicalOp
              |  (LogicalOp)
              |  RelationalOp
              |  BooleanConstant
```



Context-Free Grammar (CFG): Example Version 2

Expression	→	*ArithmeticOp*
	|	*RelationalOp*
	|	*LogicalOp*
	|	(*Expression*)
IntegerConstant	→	*Digit*
	|	*Digit IntegerConstant*
	|	−*IntegerConstant*
Digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
BooleanConstant	→	`TRUE`
	|	`FALSE`

Example: (1 + 2) / (5 - (2 + 3))

↳ a valid string.

ArithmeticOp	→	*ArithmeticOp* + *ArithmeticOp*
	|	*ArithmeticOp* − *ArithmeticOp*
	|	*ArithmeticOp* ∗ *ArithmeticOp*
	|	*ArithmeticOp* / *ArithmeticOp*
	|	(*ArithmeticOp*)
	|	*IntegerConstant*
RelationalOp	→	*ArithmeticOp* == *ArithmeticOp*
	|	*ArithmeticOp* /= *ArithmeticOp*
	|	*ArithmeticOp* > *ArithmeticOp*
	|	*ArithmeticOp* < *ArithmeticOp*
LogicalOp	→	*LogicalOp* && *LogicalOp*
	|	*LogicalOp* || *LogicalOp*
	|	*LogicalOp* => *LogicalOp*
	|	! *LogicalOp*
	|	(*LogicalOp*)
	|	*RelationalOp*
	|	*BooleanConstant*

choice: report this as compilation error.

Context-Free Grammar (CFG): Example Version 2

Expression	→	*ArithmeticOp*
	|	*RelationalOp*
	|	*LogicalOp*
	|	(*Expression*)
IntegerConstant	→	*Digit*
	|	*Digit IntegerConstant*
	|	−*IntegerConstant*
Digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
BooleanConstant	→	TRUE
	|	FALSE

Example: 3 * 5 + 4

↳ exercise: parse trees.

ArithmeticOp	→	*ArithmeticOp* + *ArithmeticOp*
	|	*ArithmeticOp* − *ArithmeticOp*
	|	*ArithmeticOp* * *ArithmeticOp*
	|	*ArithmeticOp* / *ArithmeticOp*
	|	(*ArithmeticOp*)
	|	*IntegerConstant*
RelationalOp	→	*ArithmeticOp* == *ArithmeticOp*
	|	*ArithmeticOp* /= *ArithmeticOp*
	|	*ArithmeticOp* > *ArithmeticOp*
	|	*ArithmeticOp* < *ArithmeticOp*
LogicalOp	→	*LogicalOp* && *LogicalOp*
	|	*LogicalOp* || *LogicalOp*
	|	*LogicalOp* => *LogicalOp*
	|	! *LogicalOp*
	|	(*LogicalOp*)
	|	*RelationalOp*
	|	*BooleanConstant*

$A \to w$

$uAv \Rightarrow uwv$

$S \to (S) \mid SS \mid \varepsilon$

$S()$

LECTURE 7
MONDAY JANUARY 27

transitions

$q_0 : \$ > q_1$

$\{q_0, q_1\} : \$ > q_1$ X

(compile time)

Concrete syntax 1

website
states
"

Runtime

abstract syntax (intermediate format)

fa

Epsilon NFA

DFA

subset cons.

fa

DFA

Concrete syntax 2

machine M

states =
{q0, q1, q2}

transitions =
{(q0, e, {q1, q2})}
:

From RE to Scanner (1)

Regular Expression: r[0..9]+

Token Type (CharCat)

r	0, 1, 2, ..., 9	EOF	**Other**
Register	*Digit*	*Other*	*Other*

Transition

	Register	*Digit*	*Other*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Token Type (Type)

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

```
NextWord()
  -- Stage 1:  Initialization
  state := s0 ; word := ϵ
  initialize an empty stack s ; s.push(bad)
  -- Stage 2:  Scanning Loop
  while (state ≠ se)
    NextChar(char) ; word := word + char
    if state ∈ F then reset stack s end
    s.push(state)
    cat := CharCat[char]
    state := δ[state, cat]
  -- Stage 3:  Rollback Loop
  while (state ∉ F ∧ state ≠ bad)
    state := s.pop()
    truncate word
  -- Stage 4:  Interpret and Report
  if state ∈ F then return Type[state]
  else return invalid
  end
```

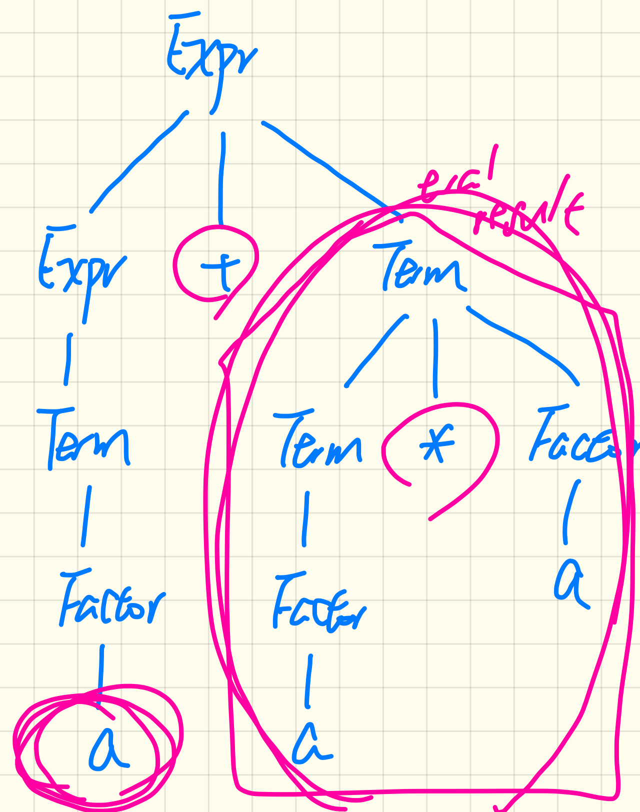
Example input: r241

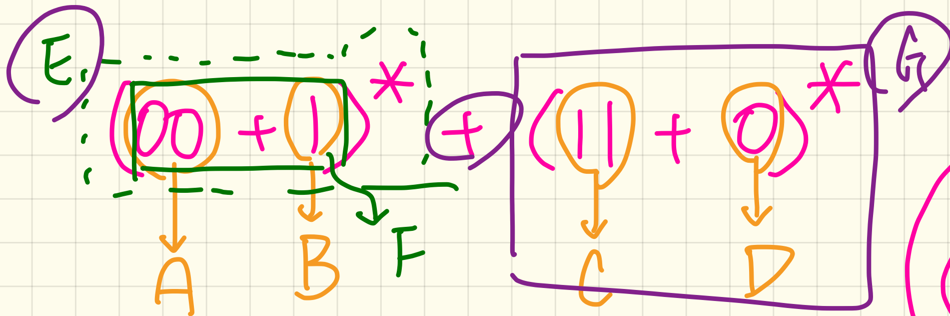
$$
\begin{aligned}
Expr &\rightarrow Expr + Term \\
&\mid Term \\
Term &\rightarrow Term * Factor \\
&\mid Factor \\
Factor &\rightarrow (Expr) \\
&\mid \texttt{a}
\end{aligned}
$$

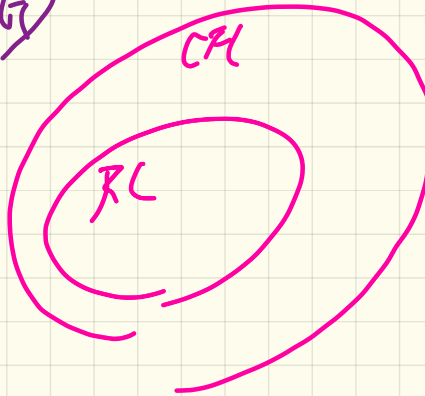
$a + a * a$

evaluation

post-order







S → E | G

E → ε | FE

F → A | B

A → 00

B → 1

C → 11

D → 0

Context-Free Grammar (CFG): from RE

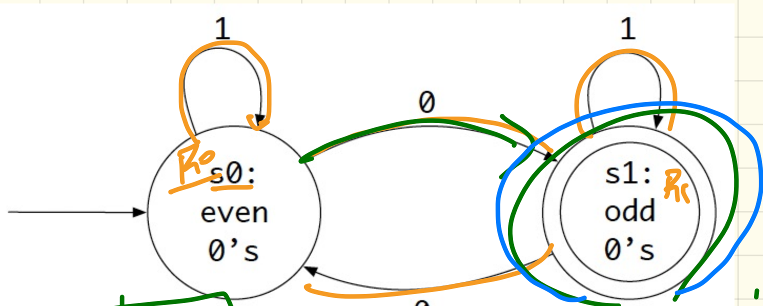
$(0+1)^*1(0+1)$

$S \to CBA$

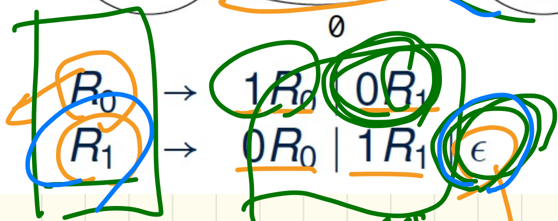
$C \to \varepsilon \mid CA$

$A \to 0 \mid 1$

$B \to 1$



start variable



states

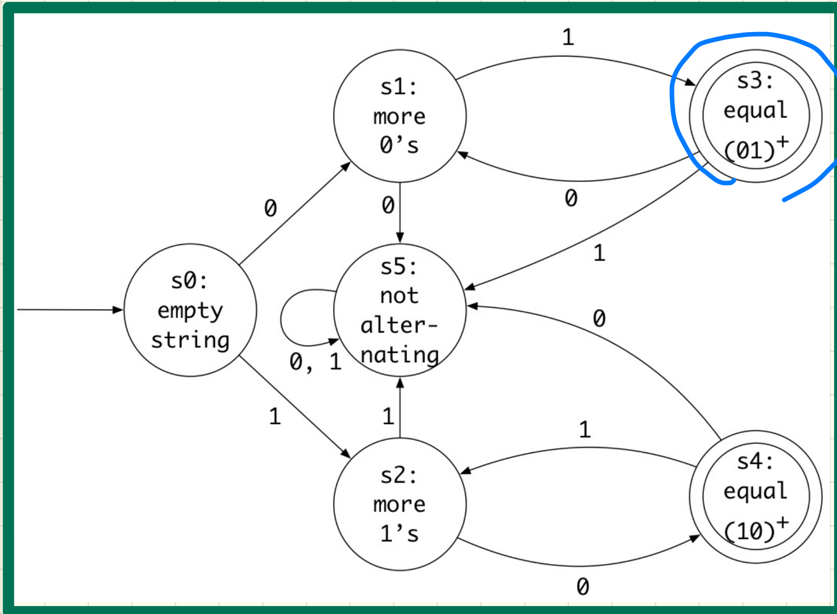
transitions

R_1 represents an accept state

$$R_0 \Rightarrow 0R_1 \Rightarrow 0$$

(R_1 → ϵ)

Context-Free Grammar (CFG): from DFA



start variab

$S_0 \rightarrow$

$S_1 \rightarrow$

$S_2 \rightarrow$

$S_3 \rightarrow \varepsilon \mid 0S_1 \mid 1S_5$

$S_4 \rightarrow \varepsilon$

$S_5 \rightarrow 0S_5 \mid 1S_5$

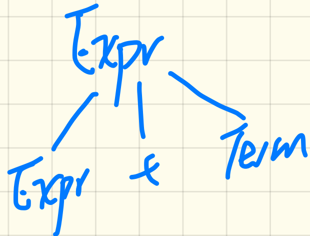
Context-Free Grammar (CFG): Leftmost Derivation

Expr	→	*Expr*	+	*Term*
	|	*Term*		
Term	→	*Term*	*	*Factor*
	|	*Factor*		
Factor	→	(*Expr*)		
	|	a		

Derivation: a + a * a

Expr ⇒ Expr + Term
⇒ Term + Term
⇒ Factor + Term
⇒ a + Term
⇒ a + Term * Factor
⇒ a + Factor * Factor
⇒ a + a * Factor
⇒ a + a * a

Parse Tree: a + a * a



Context-Free Grammar (CFG): Rightmost Derivation

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr + Term \\
 & | & Term \\
Term & \rightarrow & Term * Factor \\
 & | & Factor \\
Factor & \rightarrow & (\, Expr \,) \\
 & | & \texttt{a}
\end{array}
$$

Derivation: a + a * a

Parse Tree: a + a * a

Context-Free Grammar (CFG): Leftmost Derivation

Expr	$\rightarrow$	*Expr*	+	*Term*
	|	*Term*		
Term	$\rightarrow$	*Term*	*	*Factor*
	|	*Factor*		
Factor	$\rightarrow$	(*Expr*)		
	|	a		

Derivation: (a + a) * a

Parse Tree: a + a * a

Context-Free Grammar (CFG): Rightmost Derivation

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr + Term \\
 & | & Term \\
Term & \rightarrow & Term * Factor \\
 & | & Factor \\
Factor & \rightarrow & (\ Expr\ ) \\
 & | & \texttt{a}
\end{array}
$$

Derivation: (a + a) * a

Parse Tree: a + a * a

given input string $w \in \Sigma^*$

derivation 1

$\neq$ w

derivation 2

$\Downarrow$

G is ambiguous

$$(\boxed{a} + \boxed{a}) * \boxed{a}$$

Exercise

ex1. $a + a * a$

ex2. $a + (a * a)$

ex3. $(a + a) * a$

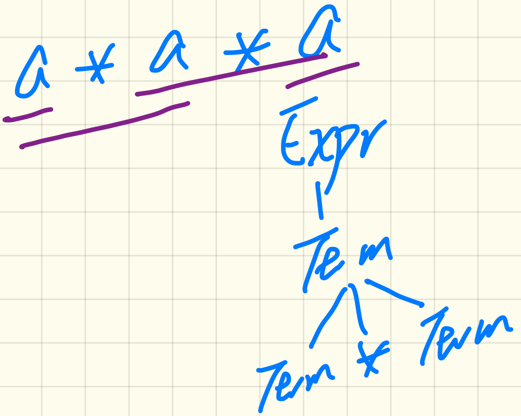
→ meaning1 (ex1, ex2)

→ meaning2 (ex3)

do they share the same meaning according G.

$$
\begin{aligned}
Expr &\rightarrow Expr + Term \\
&\mid Term \\
Term &\rightarrow Term * \text{~~Factor~~ } Term \\
&\mid Factor \\
Factor &\rightarrow (\, Expr \,) \\
&\mid \texttt{a}
\end{aligned}
$$

ambiguous

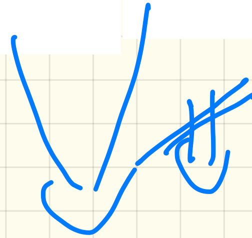


Unique leftmost derivation for the string `(a + a) * a`:

$$
\begin{aligned}
\textit{Expr} &\Rightarrow \textit{Term} \\
&\Rightarrow \textit{Term} \star \textit{Factor} \\
&\Rightarrow \textit{Factor} \star \textit{Factor} \\
&\Rightarrow (\ \textit{Expr}\ ) \star \textit{Factor} \\
&\Rightarrow (\ \textit{Expr} + \textit{Term}\ ) \star \textit{Factor} \\
&\Rightarrow (\ \textit{Term} + \textit{Term}\ ) \star \textit{Factor} \\
&\Rightarrow (\ \textit{Factor} + \textit{Term}\ ) \star \textit{Factor} \\
&\Rightarrow (\ a + \textit{Term}\ ) \star \textit{Factor} \\
&\Rightarrow (\ a + \textit{Factor}\ ) \star \textit{Factor} \\
&\Rightarrow (\ a + a\ ) \star \textit{Factor} \\
&\Rightarrow (\ a + a\ ) \star a
\end{aligned}
$$

Unique rightmost derivation for the string `(a + a) * a`:

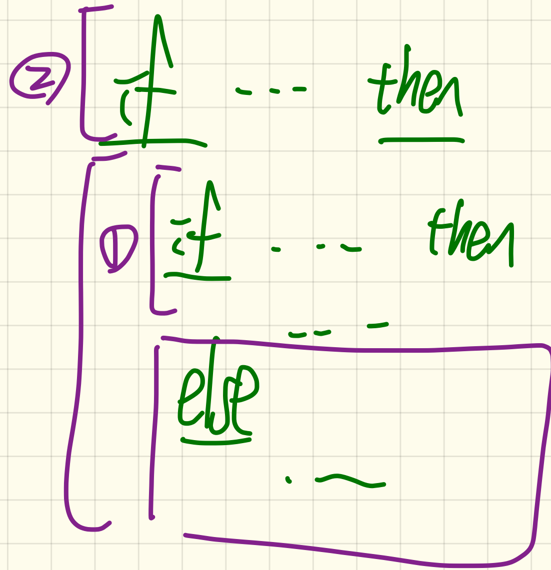
$$
\begin{aligned}
\textit{Expr} &\Rightarrow \textit{Term} \\
&\Rightarrow \textit{Term} \star \textit{Factor} \\
&\Rightarrow \textit{Term} \star a \\
&\Rightarrow \textit{Factor} \star a \\
&\Rightarrow (\ \textit{Expr}\ ) \star a \\
&\Rightarrow (\ \textit{Expr} + \textit{Term}\ ) \star a \\
&\Rightarrow (\ \textit{Expr} + \textit{Factor}\ ) \star a \\
&\Rightarrow (\ \textit{Expr} + a\ ) \star a \\
&\Rightarrow (\ \textit{Term} + a\ ) \star a \\
&\Rightarrow (\ \textit{Factor} + a\ ) \star a \\
&\Rightarrow (\ a + a\ ) \star a
\end{aligned}
$$



G is ambiguous

- i the two derivations not done in the same order.

dangling else



Lecture 8
Wednesday January 29

- Quizzes will be on Wednesdays
- Office Hours this Friday: 2:30pm

Quiz 1

Context-Free Grammar (CFG): Exercise (1)

Is the following CFG ambiguous?

$$Expr \rightarrow Expr + Expr \mid Expr * Expr \mid (Expr) \mid a$$

Context-Free Grammar (CFG): Exercise (2.1)

Is the following CFG ambiguous?

Statement	→	`if` *Expr* `then` *Statement*
	|	`if` *Expr* `then` *Statement* `else` *Statement*
	|	*Assignment*
	...	

if Expr1 then if Expr2 then Assignment1 else Assignment2

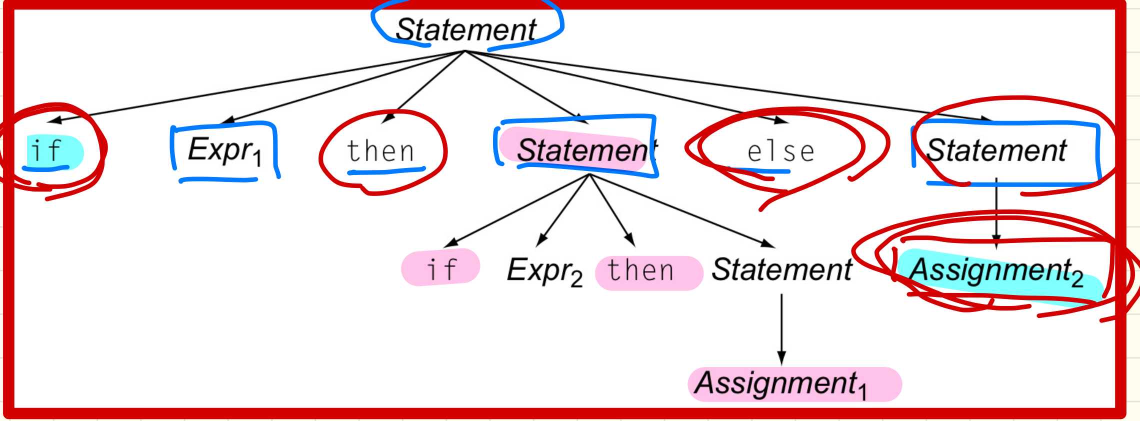
Context-Free Grammar (CFG): Exercise (2.2.1)

Is the following CFG ambiguous?

$$\begin{array}{lcl} Statement & \rightarrow & \texttt{if}\ Expr\ \texttt{then}\ Statement \\ & | & \texttt{if}\ Expr\ \texttt{then}\ Statement\ \texttt{else}\ Statement \\ & | & Assignment \\ & \dots & \end{array}$$

Example:

if *Expr1* **then if** *Expr2* **then** *Assignment1* **else** *Assignment2*



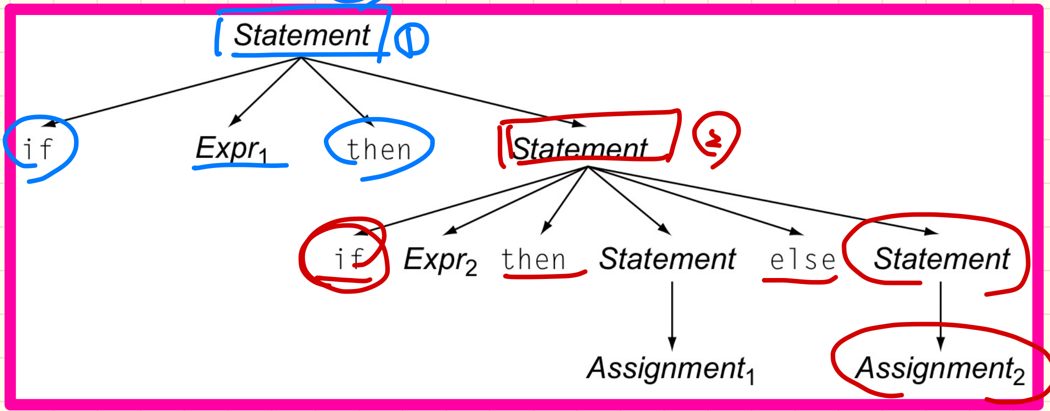
Context-Free Grammar (CFG): Exercise (2.2.2)

Is the following CFG ambiguous?

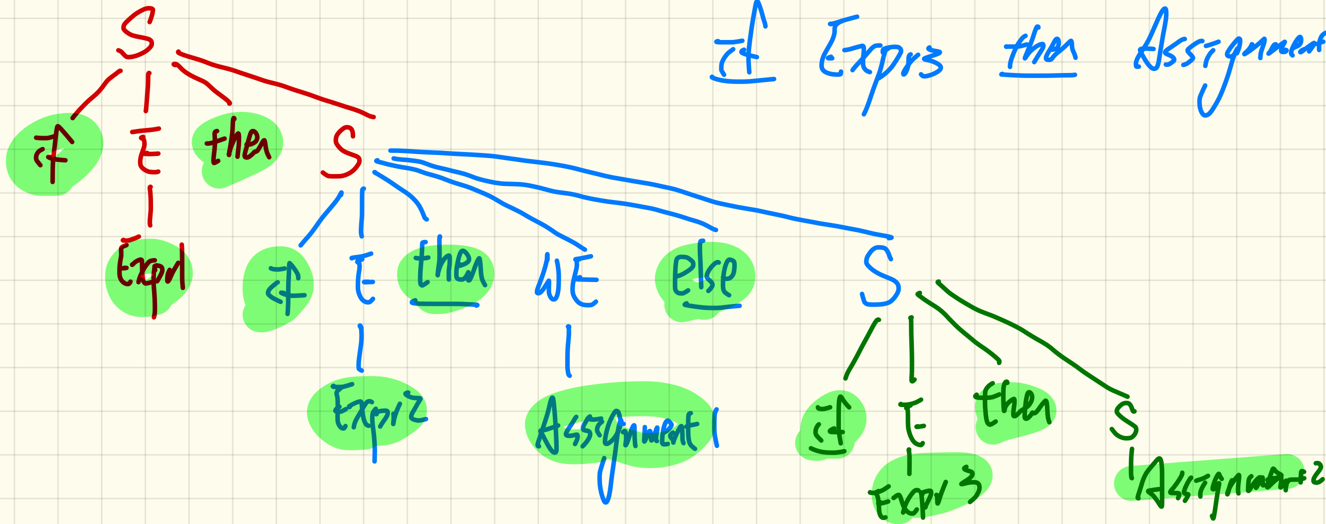
$$
\begin{array}{lll}
\textit{Statement} & \rightarrow & \texttt{if}\ \textit{Expr}\ \texttt{then}\ \textit{Statement} \\
& | & \texttt{if}\ \textit{Expr}\ \texttt{then}\ \textit{Statement}\ \texttt{else}\ \textit{Statement} \\
& | & \textit{Assignment} \\
& \ldots &
\end{array}
$$

Example:

if *Expr1* **then if** *Expr2* **then** *Assignment1* **else** *Assignment2*



Statement	→	`if` *Expr* `then` *Statement*
	|	`if` *Expr* `then` *WithElse* `else` *Statement*
	|	*Assignment*
WithElse	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
	|	*Assignment*



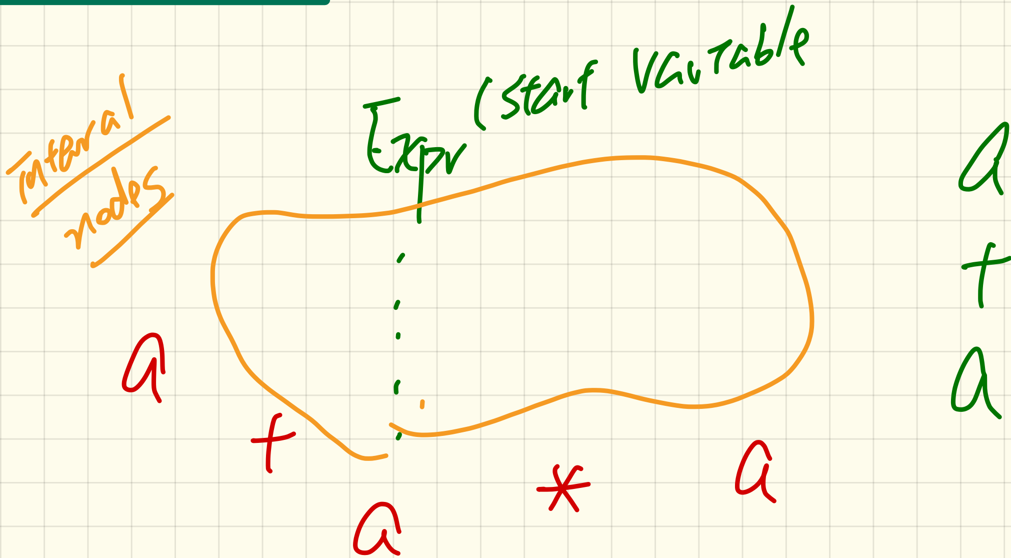
if Expr1 then
    if Expr2 then
        Assignment1
    else
        Assignment2

if Expr1 then
    if Expr2 then
        Assignment1
    else
        Assignment2

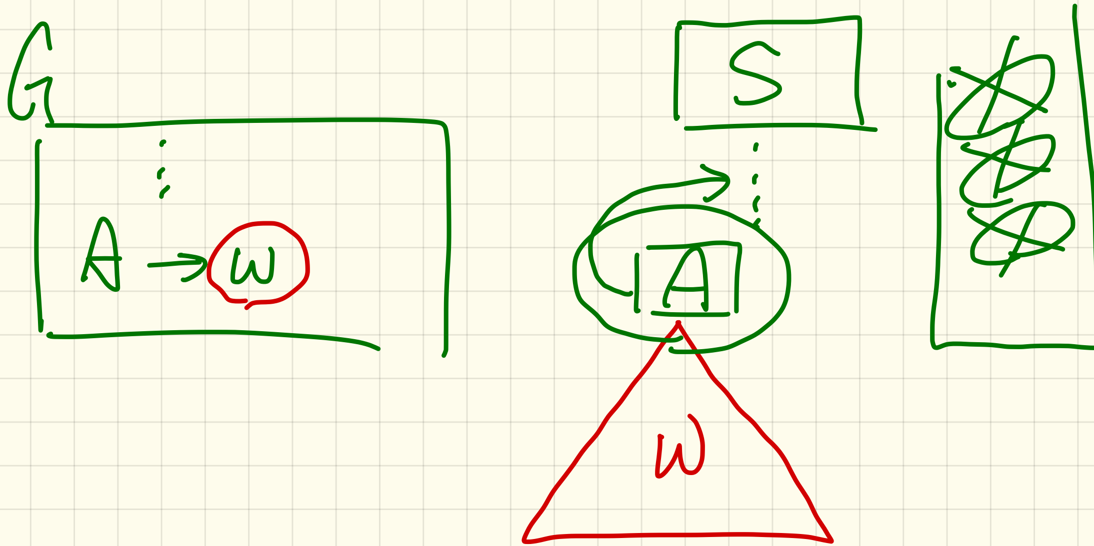
```
if (x == 0)
    if (y == 0)
        print("2")
    else
        print("3")
```

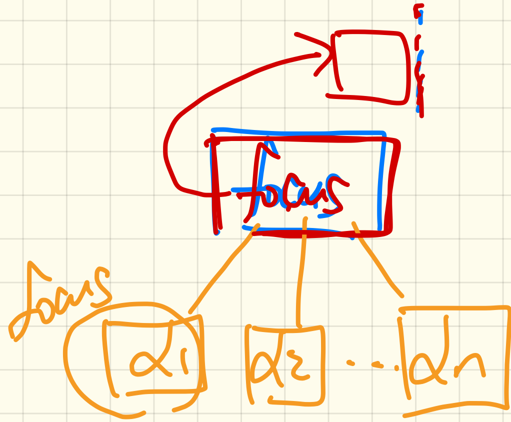
Expr	→	*Expr* + *Term*
	|	*Term*
Term	→	*Term* * *Factor*
	|	*Factor*
Factor	→	(*Expr*)
	|	a

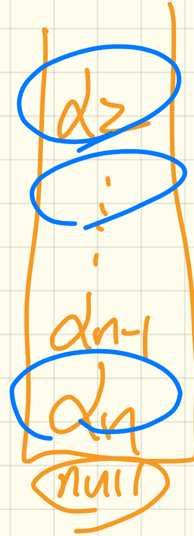
a + a * a



$w \in (V \cup \Sigma)^* \quad A \in V \quad A \to w \in R$







Expr → ✗

| —

focus → $\alpha_1 \alpha_2 \alpha_3 \cdots \alpha_n$

S → |

focus

Expr

focus

Top-Down Parsing: Algorithm

backtrack $\triangleq$ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

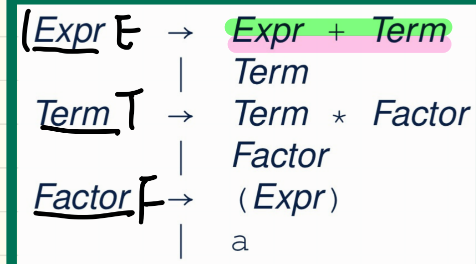
```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then  return root
    else backtrack
    end
```

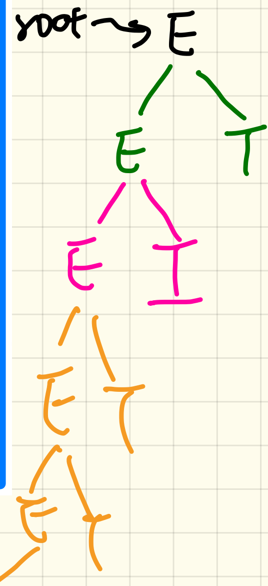
Top-Down Parsing: Discovering Leftmost Derivations (1)

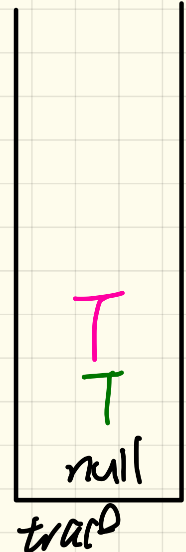
backtrack ≜ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

Parse: a + a * a

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
    end
```







word : a

focus : E E E

Lecture 9

Monday February 3

Top-Down Parsing: Algorithm

backtrack $\triangleq$ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
    end
```

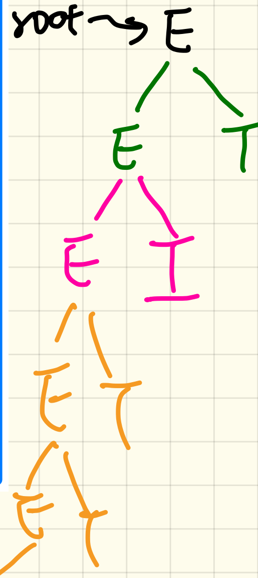
Top-Down Parsing: Discovering Leftmost Derivations (1)

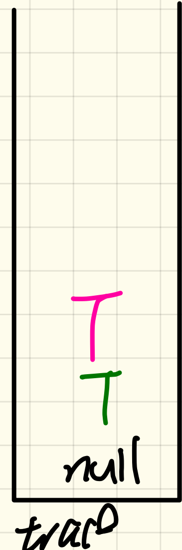
backtrack ≜ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

Parse: a + a * a

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1β2...βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βnβn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
    end
```

Expr E	→	*Expr* + *Term*
	|	*Term*
Term T	→	*Term* ⋆ *Factor*
	|	*Factor*
Factor F	→	(*Expr*)
	|	a





word : a

focus : E E E

Left-Recursions (LRs): Direct vs. Indirect

Direct Left-Recursions:

Expr	→	Expr	+	Term
	|	Term		
Term	→	Term	*	Factor
	|	Factor		
Factor	→	(Expr)		
	|	a		

Expr	→	Expr	+	Term
	|	Expr	−	Term
	|	Term		
Term	→	Term	*	Factor
	|	Term	/	Factor
	|	Factor		

Indirect Left-Recursions:

② $A \rightarrow Ba$, $B \stackrel{*}{\Rightarrow} Aafd$

A	→	Br
B	→	Cd
C	→	At

① $A \rightarrow Br$ ② $B \stackrel{*}{\Rightarrow} Atd$

A	→	Ba	|	b		
B	→	Cd	|	e		
C	→	Df	|	g		
D	→	f	|	Aa	|	Cg

CFGs: Left-Recursive vs. Right-Recursive

CFG with Left Recursions

$$
\begin{array}{lcl}
\textit{Expr} & \rightarrow & \textit{Expr} + \textit{Term} \\
& | & \textit{Term} \\
\textit{Term} & \rightarrow & \textit{Term} \ast \textit{Factor} \\
& | & \textit{Factor} \\
\textit{Factor} & \rightarrow & (\ \textit{Expr}\ ) \\
& | & \mathrm{a}
\end{array}
$$

CFG with Right Recursions

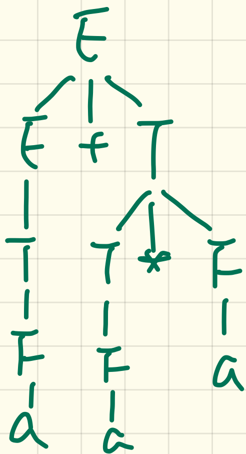
$$
\begin{array}{lcl}
\textit{Expr} & \rightarrow & \textit{Term}\ \textit{Expr}' \\
\textit{Expr}' & \rightarrow & +\ \textit{Term}\ \textit{Expr}' \\
& | & \epsilon \\
\textit{Term} & \rightarrow & \textit{Factor}\ \textit{Term}' \\
\textit{Term}' & \rightarrow & \ast\ \textit{Factor}\ \textit{Term}' \\
& | & \epsilon \\
\textit{Factor} & \rightarrow & (\ \textit{Expr}\ ) \\
& | & \mathrm{a}
\end{array}
$$

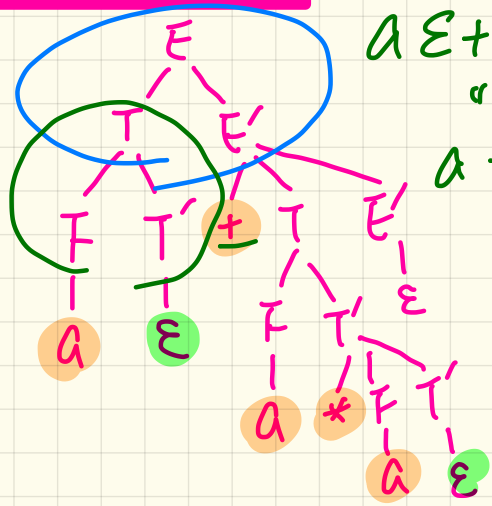
× ε-prod.

Example: a + a * a

algo.

semantically preserving.





a ε + a * a ε

"

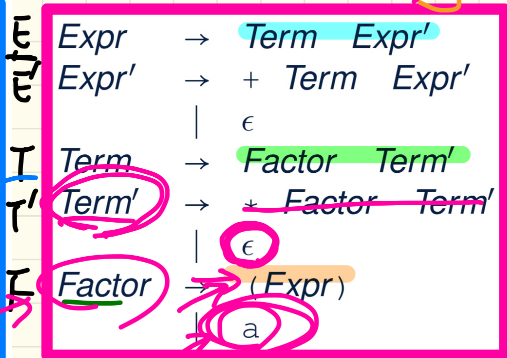
a + a * a

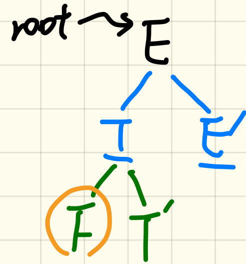
Top-Down Parsing: Discovering Leftmost Derivations (2)

backtrack ≜ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

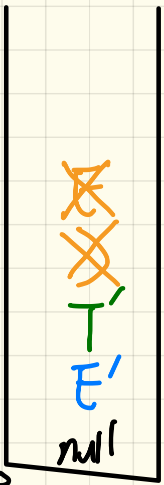
Parse: a + a * a

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
  end
```





look ahead



word: a

focus: E T E F

Top-Down Parsing: Discovering Leftmost Derivations (3)

backtrack ≜ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

Parse: (a + a) * a

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
    end
```

$Expr$	$\rightarrow$	$Term\ \ Expr'$
$Expr'$	$\rightarrow$	$+\ \ Term\ \ Expr'$
	$\mid$	ϵ
$Term$	$\rightarrow$	$Factor\ \ Term'$
$Term'$	$\rightarrow$	$\star\ \ Factor\ \ Term'$
	$\mid$	ϵ
$Factor$	$\rightarrow$	$(\ Expr\ )$
	$\mid$	a

Top-Down Parsing

→ automate

left-most derivation

preconditon: CFG G cannot contain any left-recursions direct or indirect

look ahead to avoid back tracking.

CFG G is right-recursive

$A \rightarrow B$

$B \rightarrow C$

$C \rightarrow A$

Removing Left-Recursions: Algorithm

```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G acyclic ∧ with no ε-productions
4     OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5               indirect & direct left-recursions
6   PROCEDURE:
7     impose an order on V: ⟨⟨A1, A2, ..., An⟩⟩
8     for i: 1 .. n:
9       for j: 1 .. i − 1:
10        if ∃ Ai → Ajγ ∈ R ∧ Aj → δ1 | δ2 | ... | δm ∈ R then
11          replace Ai → Ajγ with Ai → δ1γ | δ2γ | ... | δmγ
12        end
13      for Ai → Aiα | β ∈ R:
14        replace it with: Ai → βA′, A′ → αA′ | ε
```

$A_i \rightarrow A_j \gamma$

$\delta_1 \mid \delta_2 \ldots \delta_m$

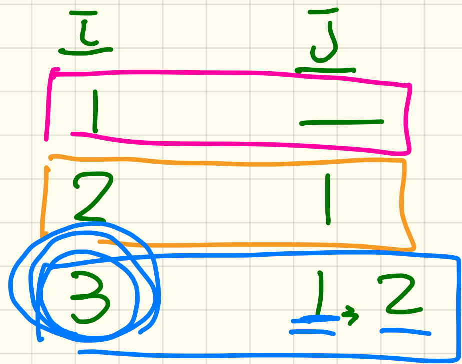
$A_i \rightarrow \delta_1 \gamma$
$\mid \delta_2 \gamma$
$\mid \vdots$
$\mid \delta_m \gamma$

remove direct LRs.

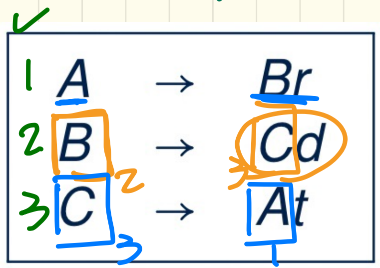
remove indirect LRs

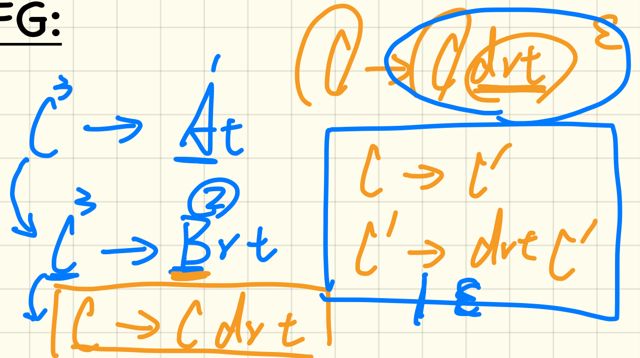
Removing Left-Recursions (1)

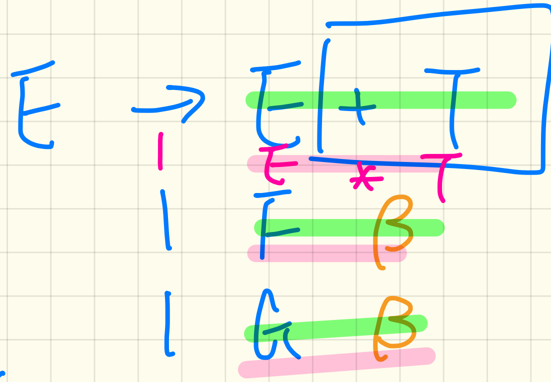
```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G acyclic ∧ with no ϵ-productions
4    OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5              indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11         replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12       end
13     for A_i → A_i α | β ∈ R:
14       replace it with: A_i → β A′, A′ → α A′ | ϵ
```

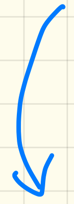


Indirectly Left-Recursive CFG:









$E \rightarrow F\ E'$

$\quad |\ G\ E'$

$E' \rightarrow +T\ E'$

$\quad |\ \varepsilon$

Removing Left-Recursions (2)

```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G acyclic ∧ with no ϵ-productions
4     OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5               indirect & direct left-recursions
6   PROCEDURE:
7     impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8     for i: 1 .. n:
9       for j: 1 .. i−1:
10        if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11          replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12        end
13      for A_i → A_i α | β ∈ R:
14        replace it with: A_i → β A′, A′ → α A′ | ϵ
```

Indirectly Left-Recursive CFG:

$$
\begin{array}{lllllll}
A & \rightarrow & Ba & | & b & & \\
B & \rightarrow & Cd & | & e & & \\
C & \rightarrow & Df & | & g & & \\
D & \rightarrow & f & | & Aa & | & Cg
\end{array}
$$

Removing Left-Recursions (3)

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G acyclic ∧ with no ϵ-productions
4    OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11         replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12       end
13     for A_i → A_i α | β ∈ R:
14       replace it with: A_i → β A′, A′ → α A′ | ϵ
```

Directly Left-Recursive CFG:

Expr	→	*Expr* + *Term*
	|	*Term*
Term	→	*Term* * *Factor*
	|	*Factor*
Factor	→	(*Expr*)
	|	a

Removing Left-Recursions (4)

```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G acyclic ∧ with no ϵ-productions
4     OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5               indirect & direct left-recursions
6   PROCEDURE:
7     impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8     for i: 1 .. n:
9       for j: 1 .. i − 1:
10        if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11          replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12        end
13      for A_i → A_i α | β ∈ R:
14        replace it with: A_i → βA′, A′ → αA′ | ϵ
```

Directly Left-Recursive CFG:

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr + Term \\
 & | & Expr - Term \\
 & | & Term \\
Term & \rightarrow & Term \star Factor \\
 & | & Term \,/\, Factor \\
 & | & Factor
\end{array}
$$

a is not **nullable**

$A \to \varepsilon$ **nullable**

$B \to C$

$C \to D$

$D \to \varepsilon$

B, C, D **nullable**

$B \to \alpha_1 \alpha_2 \ldots \alpha_n$ (a)

B is nullable if ?

$\alpha_1 \alpha_2 \alpha_3 \ldots \alpha_n$ all nullable

known: α_1

α_2

α_3 nullable

$$A \to \alpha_1 \alpha_2 \alpha_3$$

α_1	α_2	α_3
1	1	1
1	1	0
1	0	1
⋮		
0	0	0

$$A \to \alpha_1 \alpha_2 \alpha_3$$
$$\mid \alpha_1 \alpha_2$$
$$\mid \alpha_1 \alpha_3$$
$$\vdots$$
$$\mid \varepsilon$$ (crossed out)

$\alpha_1 \to \varepsilon$

$\alpha_2 \to \varepsilon$

$\alpha_3 \to \varepsilon$

Eliminating epsilon-Productions

$$S \rightarrow AB$$
$$A \rightarrow aAA \mid \epsilon$$
$$B \rightarrow bBB \mid \epsilon$$

→ G1

$G1 \not\equiv G2$

$\varepsilon \in L(G1)$

Q: **Nullable** variables?

A B S

G2

$$S \rightarrow A \mid B \mid AB$$
$$A \rightarrow aA \mid aAA \mid a$$
$$B \rightarrow bB \mid bBB \mid b$$

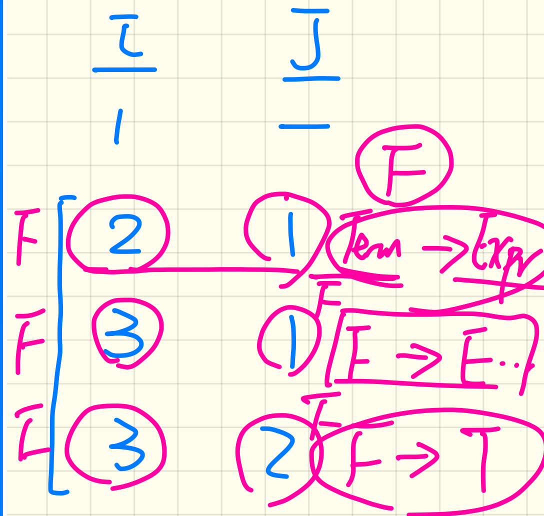
→ $\varepsilon \notin L(G2)$

Lecture 10
Wednesday February 5

Removing Left-Recursions (1)

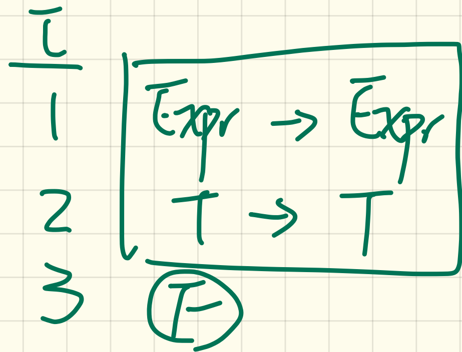
```
 1  ALGORITHM: RemoveLR
 2    INPUT: CFG G = (V, Σ, R, S)
 3    ASSUME: G acyclic ∧ with no ϵ-productions
 4    OUTPUT: G' s.t. G' ≡ G, G' has no
 5             indirect & direct left-recursions
 6  PROCEDURE:
 7    impose an order on V: ⟨⟨A1, A2, ..., An⟩⟩
 8    for i: 1 .. n:
 9      for j: 1 .. i − 1:
10        if ∃ Ai → Ajγ ∈ R ∧ Aj → δ1 | δ2 | ... | δm ∈ R then
11          replace Ai → Ajγ with Ai → δ1γ | δ2γ | ... | δmγ
12        end
13      for Ai → Aiα | β ∈ R:
14        replace it with: Ai → βA', A' → αA' | ϵ
```

not executed at all



Directly Left-Recursive CFG:

Expr	→	Expr + Term
	|	Term
Term	→	Term ∗ Factor
	|	Factor
Factor	→	(Expr)
	|	a



```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G acyclic ∧ with no ϵ-productions
4     OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5               indirect & direct left-recursions
6   PROCEDURE:
7     impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8     for i: 1 .. n:
9       for j: 1 .. i − 1:
10        if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11          replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12      end
13      for A_i → A_i α | β ∈ R:
14        replace it with: A_i → β A′, A′ → α A′ | ϵ
```

i j

D. LR

Expr	→	Expr + Term
	|	Term
Term	→	Term ⋆ Factor
	|	Factor
Factor	→	(Expr)
	|	a

α β

Expr → Term Expr′

Expr′ → + Term Expr′ | ε

Exercise.

Removing Left-Recursions (2)

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G acyclic ∧ with no ε-productions
4    OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11         replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12     end
13     for A_i → A_i α | β ∈ R:
14       replace it with: A_i → β A′, A′ → α A′ | ε
```

Directly Left-Recursive CFG:

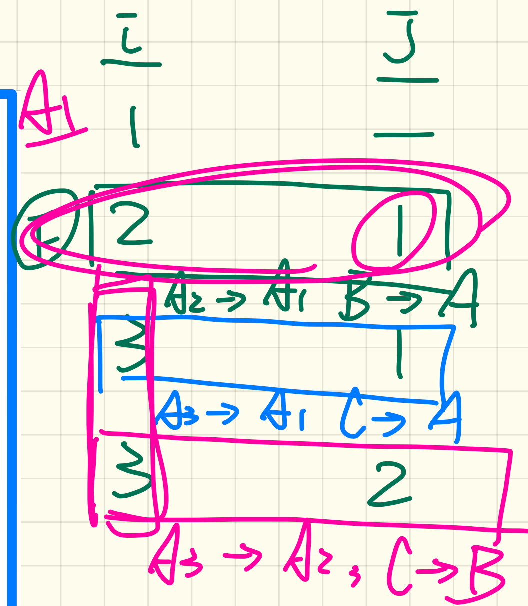
Expr	→	*Expr*	+	*Term*
	|	*Expr*	−	*Term*
	|	*Term*		
Term	→	*Term*	⋆	*Factor*
	|	*Term*	/	*Factor*
	|	*Factor*		

Exercise.

Removing Left-Recursions (3)

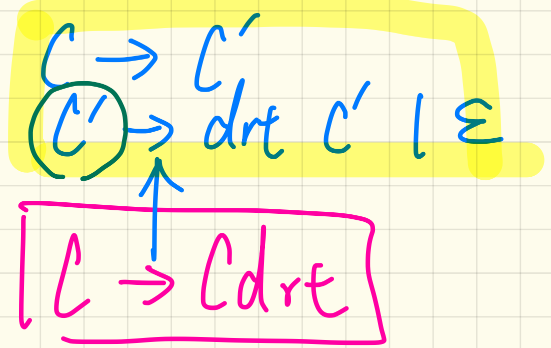
```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G acyclic ∧ with no ε-productions
4     OUTPUT: G' s.t. G' ≡ G, G' has no
5               indirect & direct left-recursions
6   PROCEDURE:
7     impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8     for i: 1 .. n:
9       for j: 1 .. i−1:
10        if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11          replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12        end
13      for A_i → A_i α | β ∈ R:
14        replace it with: A_i → β A', A' → α A' | ε
```

expect: direct LR resulted



Indirectly Left-Recursive CFG:

1	A	$\rightarrow$	Br
2	B	$\rightarrow$	Cd
3	C	$\rightarrow$	At



Removing Left-Recursions (4)

```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G acyclic ∧ with no ϵ-productions
4     OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5               indirect & direct left-recursions
6   PROCEDURE:
7     impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8     for i: 1 .. n:
9       for j: 1 .. i − 1:
10        if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11          replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12        end
13      for A_i → A_i α | β ∈ R:
14        replace it with: A_i → β A′, A′ → α A′ | ϵ
```

assert L.I. i=3

r.i.r.

r.d.r.

Indirectly Left-Recursive CFG:

A	$\rightarrow$	Ba	$\mid$	b		
B	$\rightarrow$	Cd	$\mid$	e		
C	$\rightarrow$	Df	$\mid$	g		
D	$\rightarrow$	f	$\mid$	Aa	$\mid$	Cg

Top-Down Parsing: Algorithm

backtrack $\triangleq$ pop *focus*.children; *focus* := *focus*.parent; *focus*.resetChildren

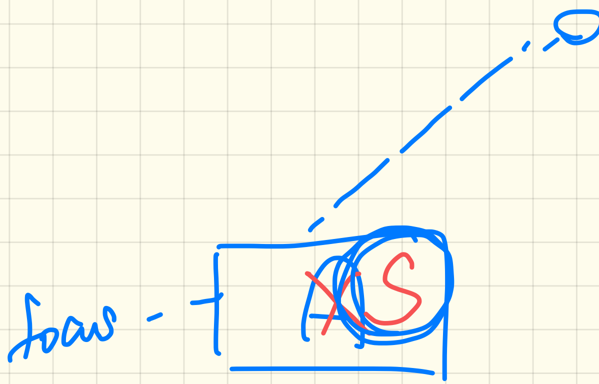
```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
    end
```

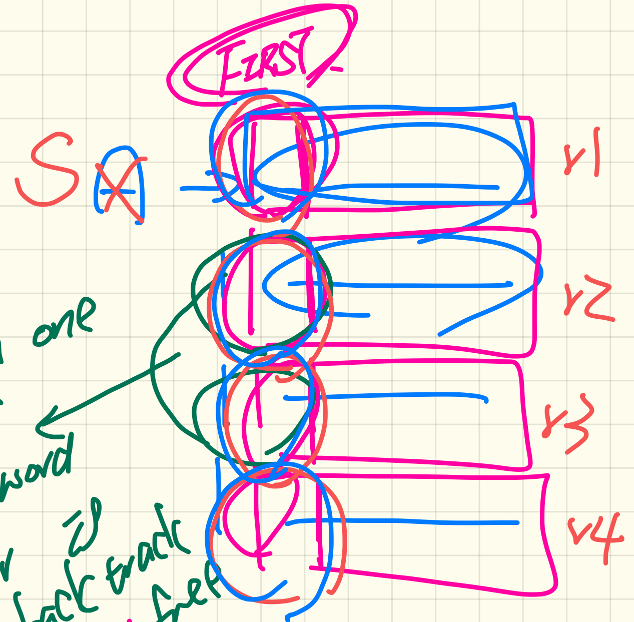
0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	× *Factor Term'*
7		|	÷ *Factor Term'*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

Term'

FIRST

word: *





more than one FIRST symbols match the word

⇒ Grammar is not backtrack free

word : ↵

FIRST of r1, r2, r3, r4 all mismatch word.

↳ input has syntax error.

exactly one FIRST symbol that matches word

⇒ Grammar is backtrack free

FIRST Set

$$\textsc{First}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \overset{*}{\Rightarrow} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

first word

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

1, 2, 3 a/an

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

focus → A

word: x

$x \in F1$
$x \in F2$
$x \in F3$

⇒ G is not backtrack free.

$F1 \cap F2 \cap F3 = \phi$.

Lecture 11

Monday February 10

```java
public ASTNode(String label, ASTNode... children) {
    /* Your Task */
}
```

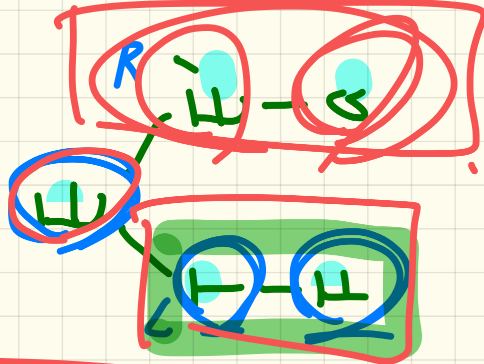
varargs
variable
argument

ASTNode[] children

children.length

children[i]

new ASTNode("E",
    new ASTNode("T",
        new ASTNode("F")
    ),
new ASTNode("E", new ASTNode("a"))).



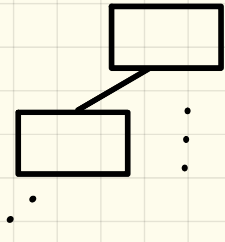
Top-Down Parsing: Backtrack

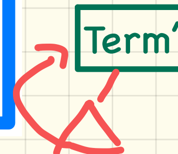
backtrack ≜ pop *focus*.siblings; *focus* := *focus*.parent; *focus*.resetChildren

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
        end
      end
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
    end
```

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ϵ
5	*Term*	→	*Factor Term'*
6	*Term'*	→	× *Factor Term'*
7		|	÷ *Factor Term'*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

FIRST





FIRST Set

$$\text{FIRST}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \overset{*}{\Rightarrow} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

single word

w/β

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

FIRST Set: Algorithm

$$\text{FIRST}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \overset{*}{\Rightarrow} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

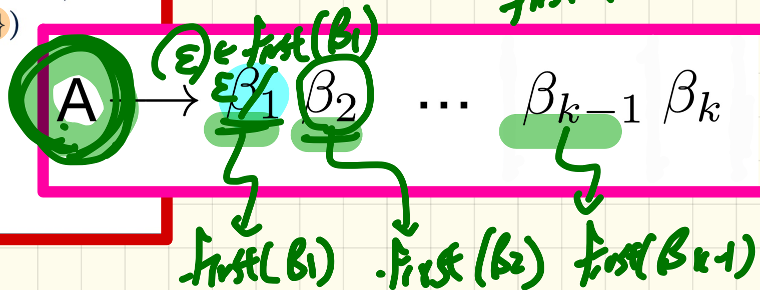
ALGORITHM: *GetFirst*
  INPUT: *CFG* $G = (V, \Sigma, R, S)$
  $T \subset \Sigma^*$ *denotes valid terminals*
  OUTPUT: $\text{FIRST} : V \cup T \cup \{\epsilon, eof\} \longrightarrow \mathbb{P}(T \cup \{\epsilon, eof\})$
PROCEDURE:
  for $\alpha \in (T \cup \{eof, \epsilon\})$: $\text{FIRST}(\alpha) := \{\alpha\}$
  for $A \in V$: $\text{FIRST}(A) := \varnothing$
  $lastFirst := \text{FIRST}$
  while ($lastFirst \neq \text{FIRST}$):
    for $A \to \beta_1\beta_2\ldots\beta_k \in R$ s.t. $\forall \beta_j : \beta_j \in (T \cup V)$:
      $rhs := \text{FIRST}(\beta_1) - \{\epsilon\}$
      for ($i := 1$; $\epsilon \in \text{FIRST}(\beta_i) \wedge i < k$; i++):
        $rhs := rhs \cup (\text{FIRST}(\beta_{i+1}) - \{\epsilon\})$
      if $i = k \wedge \epsilon \in \text{FIRST}(\beta_k)$ **then**
        $rhs := rhs \cup \{\epsilon\}$
      end
      $\text{FIRST}(A) := \text{FIRST}(A) \cup rhs$
    $lastFirst := \text{FIRST}$

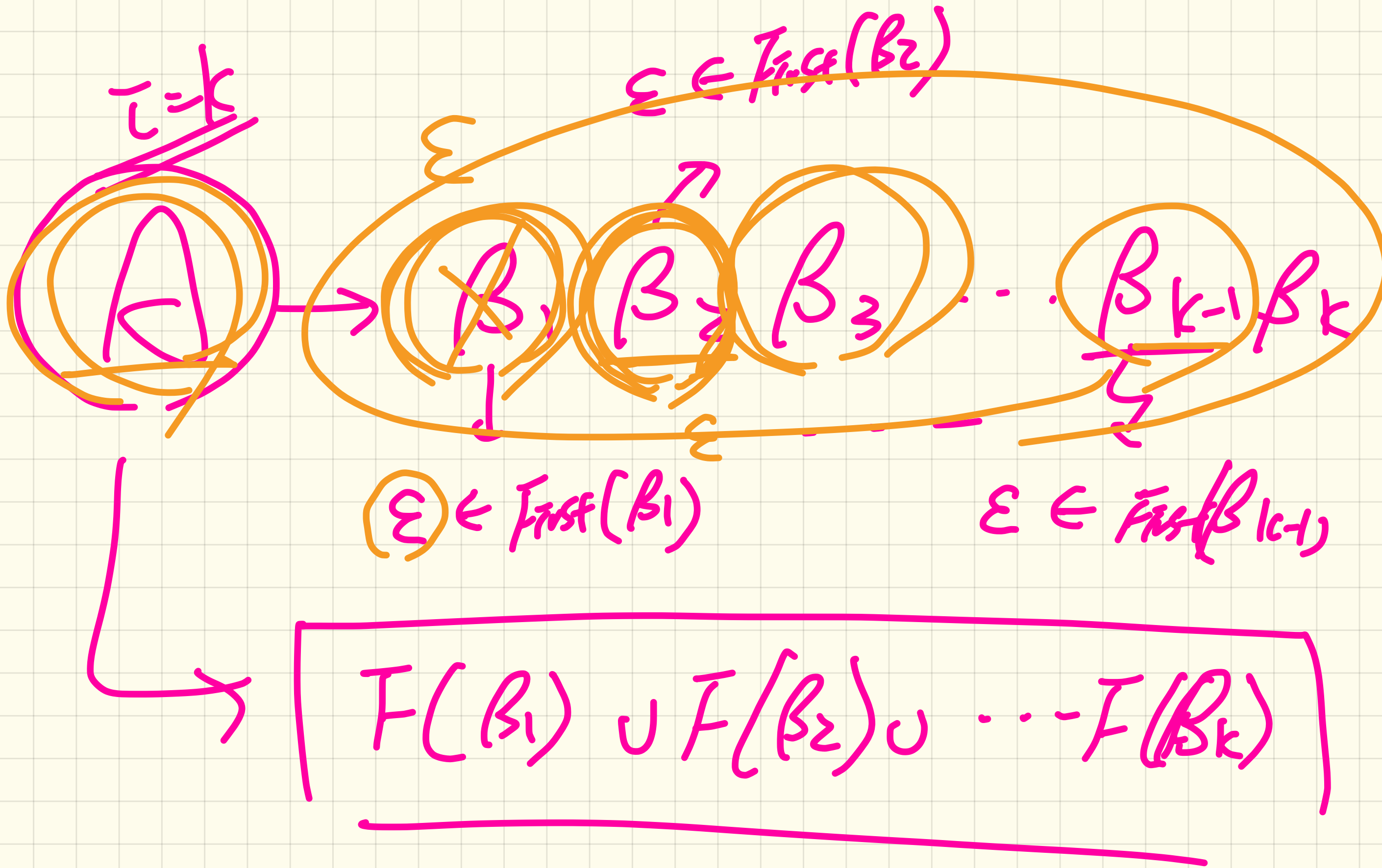
$\beta_1 \to$; ϵ

β_i is **nullable**

$\beta_1 \ldots \beta_{k-1}$ are **nullable**

first (A) = first (β_1) $\cup$ first (β_2)





i=k
A
ε
ε ∈ First(β2)
β1 β2 β3 ... βk-1 βk
ε
ε ∈ First(β1)
ε ∈ First(βk-1)
F(β1) ∪ F(β2) ∪ ... F(βk)

Right-Recursive CFG:

0	*Goal*	→	*Expr*	
1	*Expr*	→	*Term Expr'*	
2	*Expr'*	→	+	*Term Expr'*
3		|	-	*Term Expr'*
4		|	ϵ	
5	*Term*	→	*Factor Term'*	
6	*Term'*	→	×	*Factor Term'*
7		|	÷	*Factor Term'*
8		|	ϵ	
9	*Factor*	→	(*Expr*)	
10		|	num	
11		|	name	

F, E', T', T, E

FIRST Set: Tracing

First choose rules whose RHS starts with a terminal

ALGORITHM: *GetFirst*

INPUT: *CFG* $G = (V, \Sigma, R, S)$

$T \subset \Sigma^*$ *denotes valid terminals*

OUTPUT: $\textsc{First} : V \cup T \cup \{\epsilon, eof\} \longrightarrow \mathbb{P}(T \cup \{\epsilon, eof\})$

PROCEDURE:

```
for α ∈ (T ∪ {eof, ϵ}): FIRST := {α}
for A ∈ V: FIRST := ∅
lastFirst := FIRST
while (lastFirst ≠ FIRST):
  for A → β1 β2 ... βk ∈ R s.t. ∀βj : βj ∈ (T ∪ V):
    rhs := FIRST(β1) − {ϵ}
    for (i := 1; ϵ ∈ FIRST(βi) ∧ i < k; i++):
      rhs := rhs ∪ (FIRST(βi+1) − {ϵ})
    if i = k ∧ ϵ ∈ FIRST(βk) then
      rhs := rhs ∪ {ϵ}
    end
    FIRST(A) := FIRST(A) ∪ rhs
  lastFirst := FIRST
```

$\epsilon \in F(\beta_i)$

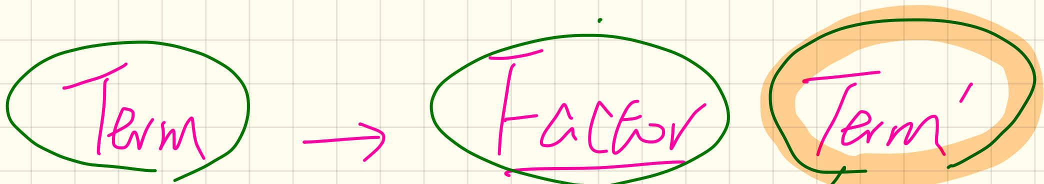
$A \to \cdots \beta_i \beta_{i+1}$

ϵ $F(\beta_{i+1})$

num	name	+	-	×	÷	(	)	eof	ϵ
num	name	+	−	×	÷	(	)	eof	ϵ

Expr	*Expr'*	*Term*	*Term'*	*Factor*
(, num, name	+, −, ϵ	(, num, name	×, ÷, ϵ	(, num, name

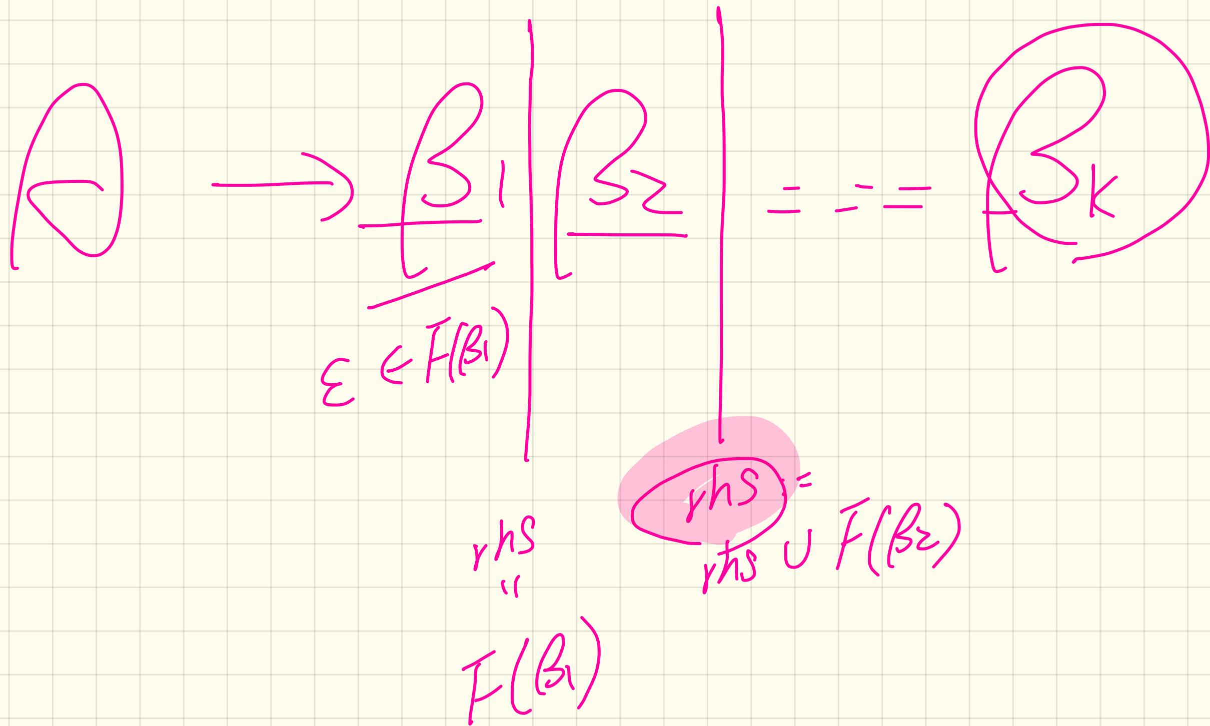
0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name



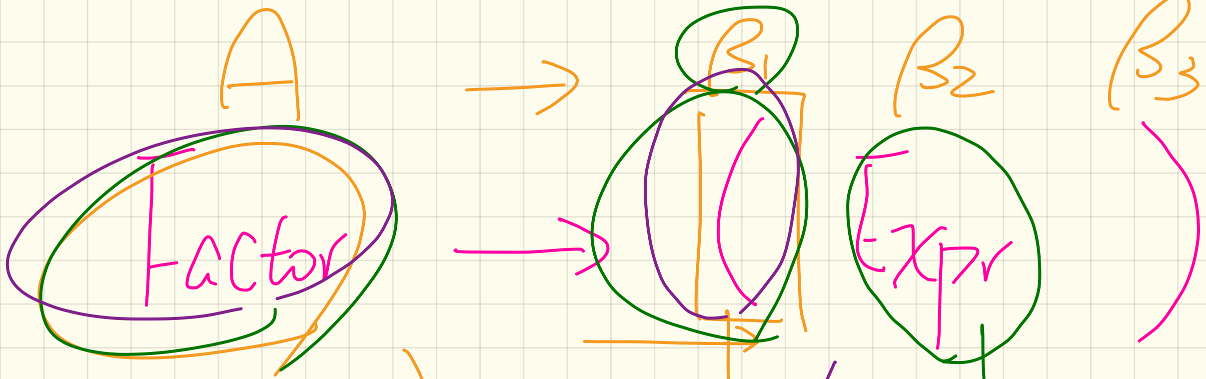
First(Term) := First(Factor)

Factor is not nullable

don't include First(Term')



A → β1 | β2 | ... βk
ε ∈ F(β1)
rhs
F(β1)
rhs :=
rhs ∪ F(β2)



$$First(Factor) = First(( ) = \{ (\}$$

$$First(\beta_1) = \{ (\}$$

no need to include First(E) in First(F)

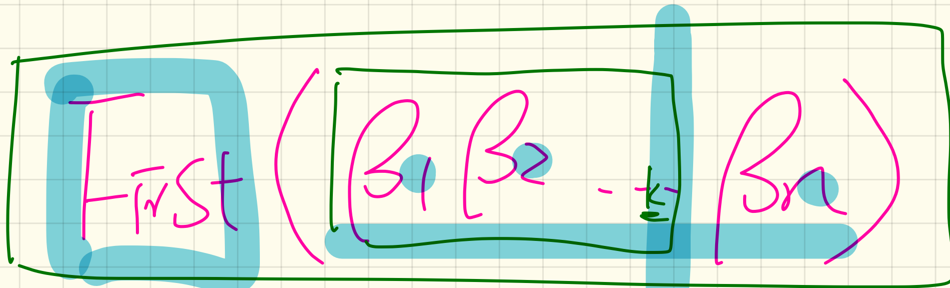
∵ (is not nullable

0	*Goal*	$\rightarrow$	*Expr*	6	*Term'*	$\rightarrow$	x *Factor Term'*
1	*Expr*	$\rightarrow$	*Term Expr'*	7		|	÷ *Factor Term'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*	8		|	ϵ
3		|	- *Term Expr'*	9	*Factor*	$\rightarrow$	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	$\rightarrow$	*Factor Term'*	11		|	name

V → Term Term' Expr' Factor

First(V) := First(Term) ⋃ E

First(V) = First(Term') - {ε} ∪ First(Expr') - {ε} ∪ First(Factor)



$\beta_{k+1} \cdots \beta_n$

$$= \text{First}(\beta_1) \cup \text{First}(\beta_2) \cup \cdots \cup \text{First}(\beta_k)$$

β_1 null, β_2 null, $\cdots$, β_{k-1} null, β_k not null

Extended First Set

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

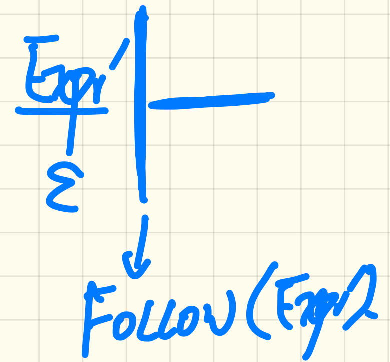
$$\text{FIRST}(\beta_1 \beta_2 \ldots \beta_n) = \left\{ \text{FIRST}(\beta_1) \cup \text{FIRST}(\beta_2) \cup \ldots \beta_k \;\middle|\; \begin{array}{l} \forall i : 1 \le i < k \bullet \epsilon \in \text{FIRST}(\beta_i) \\ \wedge \\ \epsilon \notin \text{FIRST}(\beta_k) \end{array} \right\}$$

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term'*	→	x *Factor Term'*
1	*Expr*	→	*Term Expr'*	7		|	÷ *Factor Term'*
2	*Expr'*	→	+ *Term Expr'*	8		|	ϵ
3		|	- *Term Expr'*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term'*	11		|	name

Is the FIRST Set Sufficient?

Expr′	→	+	*Term* *Term′*	(1)
	|	−	*Term* *Term′*	(2)
	|	ϵ		(3)



FIRST(+ Term Term′) =

FIRST(- Term Term′) =

FIRST(epsilon) =

a ∉ ε ⇒ syntax error

word: a

Expr′

Term

ε

focus

a

FOLLOW Set

variable

$V \Rightarrow \cdots \Rightarrow x$

$$\text{FOLLOW}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwv\}$$

derivable from v

Right-Recursive CFG:

0	*Goal*	$\rightarrow$	*Expr*	6	*Term'*	$\rightarrow$	x *Factor Term'*
1	*Expr*	$\rightarrow$	*Term Expr'*	7		|	÷ *Factor Term'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*	8		|	ϵ
3		|	- *Term Expr'*	9	*Factor*	$\rightarrow$	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	$\rightarrow$	*Factor Term'*	11		|	name

	Expr	Expr'	Term	Term'	Factor
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

	Expr	Expr'	Term	Term'	Factor
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	× *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

G, F, E, T, T′

FOLLOW Set: Tracing

First choose rules whose LHS is processed.

Then rules whose RHS ends with a terminal.

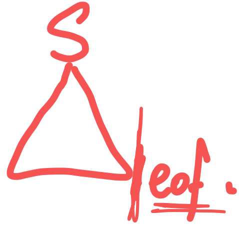
ALGORITHM: *GetFollow*

INPUT: *CFG* $G = (V, \Sigma, R, S)$

OUTPUT: FOLLOW : $V \longrightarrow \mathbb{P}(T \cup \{eof\})$

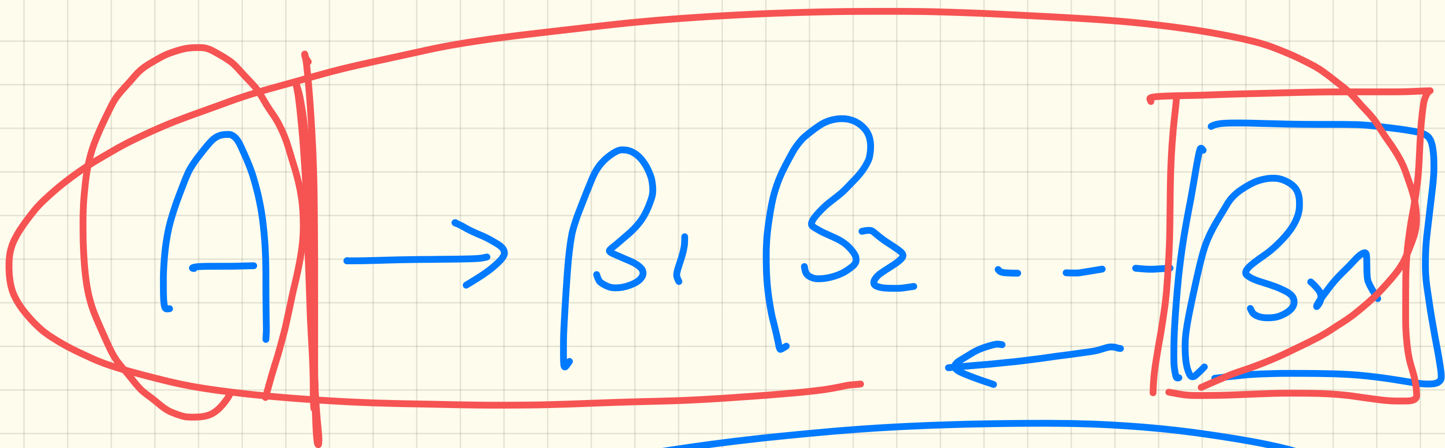
PROCEDURE:

```
for A ∈ V: FOLLOW := ∅
FOLLOW(S) := {eof}
lastFollow := FOLLOW
while (lastFirst ≠ FIRST):
  for A → β1 β2 ... βk ∈ R:
    trailer := FOLLOW(A)
    for i: k .. 1:
      if βi ∈ V then
        FOLLOW(βi) := FOLLOW(βi) ∪ trailer
        if ϵ ∈ FIRST(βi)
          then trailer := trailer ∪ (FIRST(βi) − ϵ)
          else trailer := FIRST(βi)
      else
        trailer := FIRST(βi)
  lastFollow := FOLLOW
```



	Expr	Expr′	Term	Term′	Factor
FIRST	(, name, num	+, -, ϵ	(, name, num	×, ÷, ϵ	(, name, num

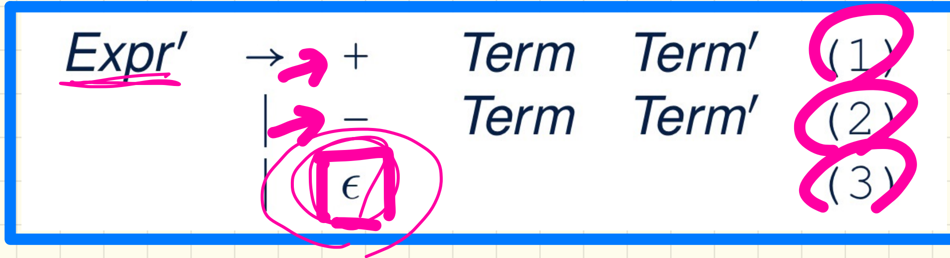
Expr	Expr′	Term	Term′	Factor
eof				



$$\text{Follow}(\beta_n) := \text{Follow}(A)$$

LECTURE 12
WEDNESDAY FEBRUARY 12

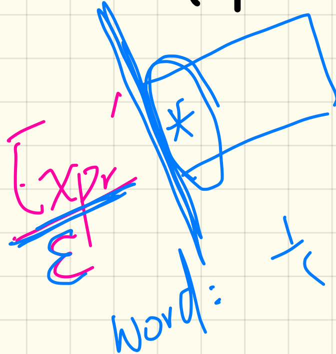
Is the FIRST Set Sufficient?

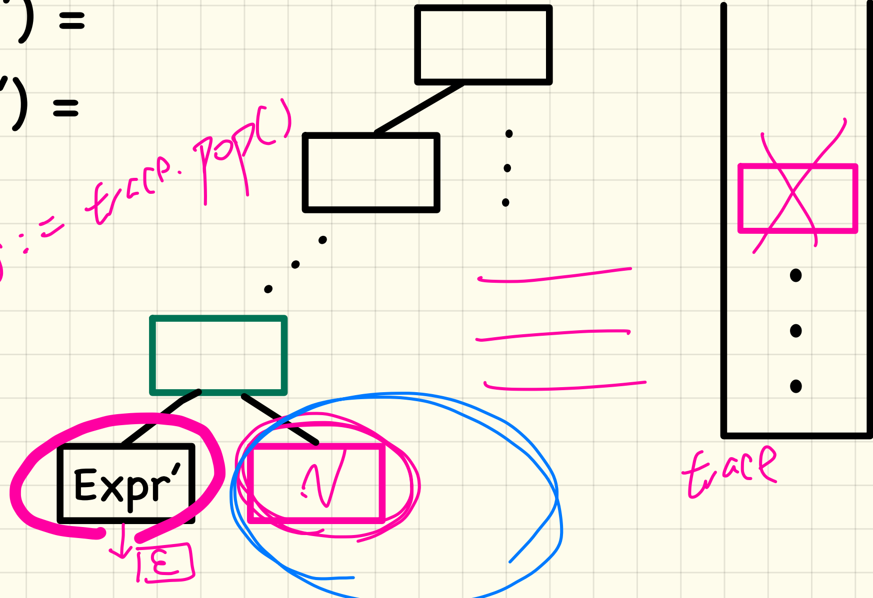


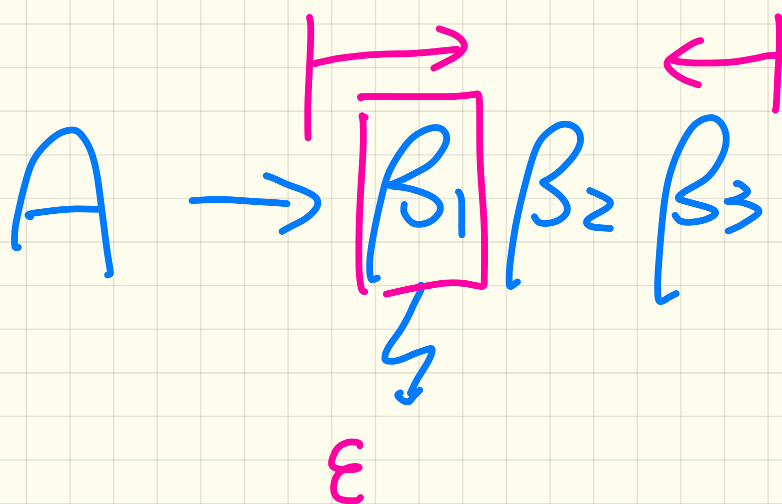
FIRST(+ Term Term') =

FIRST(- Term Term') =

FIRST(epsilon) =







$$\text{First}(A) = \text{First}(\beta_1) \cup \text{First}(\beta_2)$$

First

rhs

rhs →

Follow

trailer

← trail

FOLLOW Set

$$\text{FOLLOW}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy\}$$

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

	Expr	Expr′	Term	Term′	Factor
FIRST	(, name , num	+, - , ϵ	(, name , num	x , ÷ , ϵ	(, name , num

	Expr	Expr′	Term	Term′	Factor
FOLLOW	eof ,)	eof ,)	eof , + , - ,)	eof , + , - ,)	eof , + , - , x , ÷ ,)

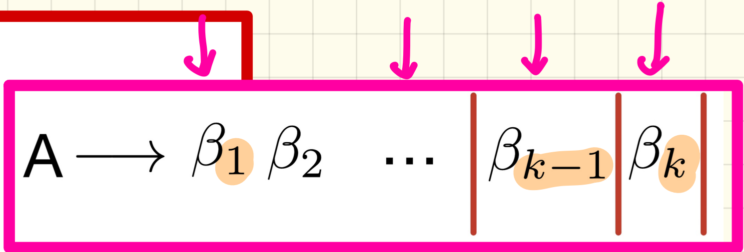
FOLLOW Set: Algorithm

$$\text{FOLLOW}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy\}$$

```
ALGORITHM: GetFollow
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: FOLLOW : V ⟶ ℙ(T ∪ {eof})
PROCEDURE:
  for A ∈ V: FOLLOW(A) := ∅
  FOLLOW(S) := {eof}
  lastFollow := ∅
  while (lastFollow ≠ FOLLOW):
    lastFollow := FOLLOW
    for A → β1 β2 ... βk ∈ R:
      trailer := FOLLOW(A)
      for i: k .. 1:
        if βi ∈ V then
          FOLLOW(βi) := FOLLOW(βi) ∪ trailer
          if ε ∈ FIRST(βi)
            then trailer := trailer ∪ (FIRST(βi) − ε)
            else trailer := FIRST(βi)
        else
          trailer := FIRST(βi)
```

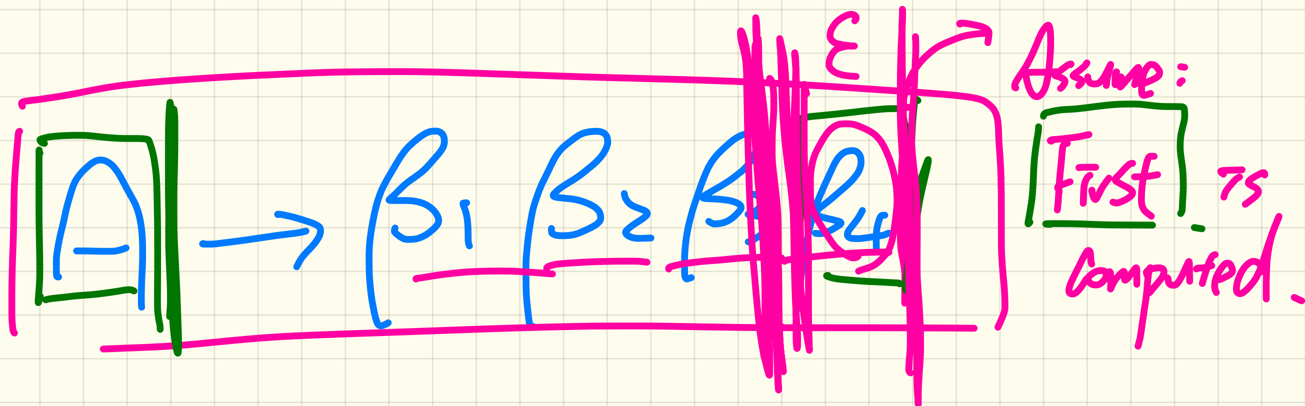
k down to 1

→ β_i is nullable



When i = k

When i = k − 1



Calculate Follow of β1, β2, β3, and β4

Term → Factor Term'

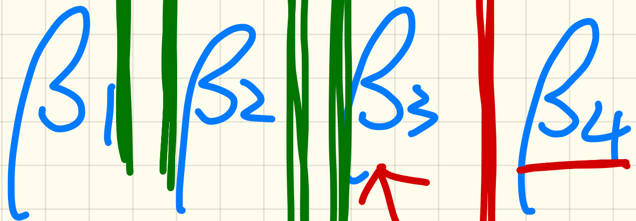
F(Term') = F(Term)

Follow(β4) = Follow(A)

Follow(β3) = { First(β4) if β4 is not nullable.
First(β4) ∪ Follow(β4) if β4 is nullable }

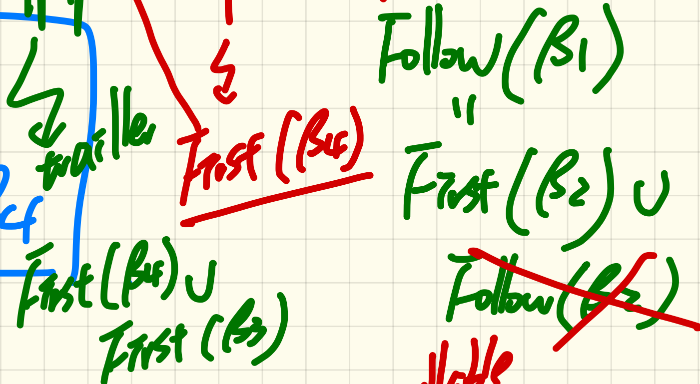
Follow(A)

$A \rightarrow$



nullable: β_3

not nullable: $\beta_1, \beta_2, \beta_4$



$Follow(\beta_4) = Follow(A)$

$Follow(\beta_3) = First(\beta_4) \cup$ ~~$Follow(\beta_4)$~~ $\because \beta_4$ not nullable

$Follow(\beta_2) = First(\beta_3) \cup Follow(\beta_3)$ $\leftarrow \because \beta_3$ nullable.

Right-Recursive CFG:

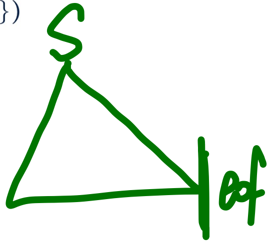
0	*Goal*	→	*Expr*	6	*Term′*	→	× *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

G, E, E′, T, T′

FOLLOW Set: Tracing

First choose rules whose LHS is processed.
Then rules whose RHS ends with a terminal.

```
ALGORITHM: GetFollow
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: FOLLOW : V ⟶ ℙ(T ∪ {eof})
PROCEDURE:
  for A ∈ V: FOLLOW(A) := ∅
  FOLLOW(S) := {eof}
  lastFollow := ∅
  while(lastFollow ≠ FOLLOW):
    lastFollow := FOLLOW
    for A → β1β2...βk ∈ R:
      trailer := FOLLOW(A)
      for i: k .. 1:
        if βi ∈ V then
          FOLLOW(βi) := FOLLOW(βi) ∪ trailer
          if ϵ ∈ FIRST(βi)
            then trailer := trailer ∪ (FIRST(βi) − ϵ)
            else trailer := FIRST(βi)
        else
          trailer := FIRST(βi)
```

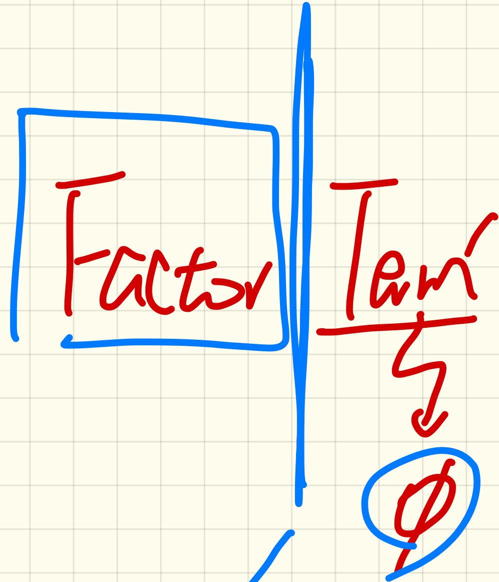


	Expr	Expr′	Term	Term′	Factor
FIRST	(, name, num	+, -, ϵ	(, name, num	×, ÷, ϵ	(, name, num

Goal: {eof}

Expr	Expr′	Term	Term′	Factor
eof	eof	+		*
)	)	−		÷
		ε		ε
		eof		
		)		

Term' → *

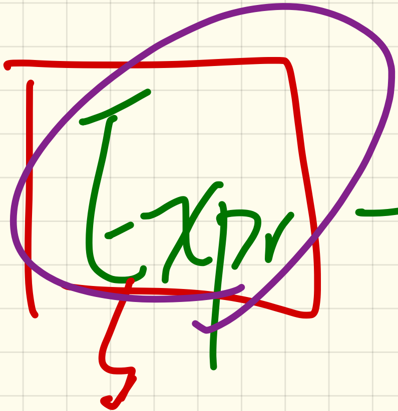


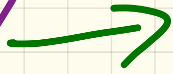
Follow(T') = ∅

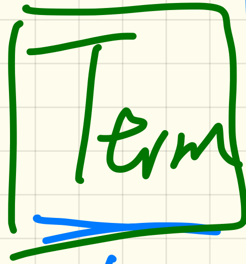
Follow(Factor) = First(Term')

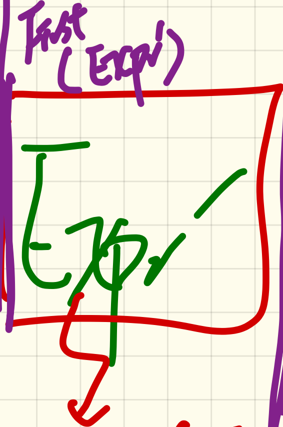
= * & ε

trailer := trailer ∪ t = Follow(E')









Follow(Expr) = {eof,)}

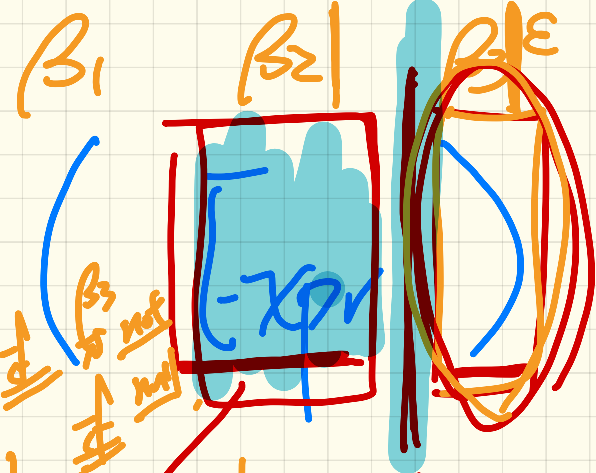
Follow(Term) = First(Expr') ∪ Follow(Expr')

{+, -, ε}

¬(ε ∈ First(Expr'))

Follow(Expr') = {eof,)}

Follow(β3) = Follow(Factor)

Factor → 

trailer

β3 ≠ not nullable

β3 ≠ nullable

Follow(β2) = ① First(β3)
② First(β3) ∪ Follow(β3)
β3 is nullable

Follow(Expr) = First( )) ∪ ~~Follow(Factor)~~ ")" is not nullable.

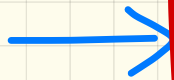
= {) }

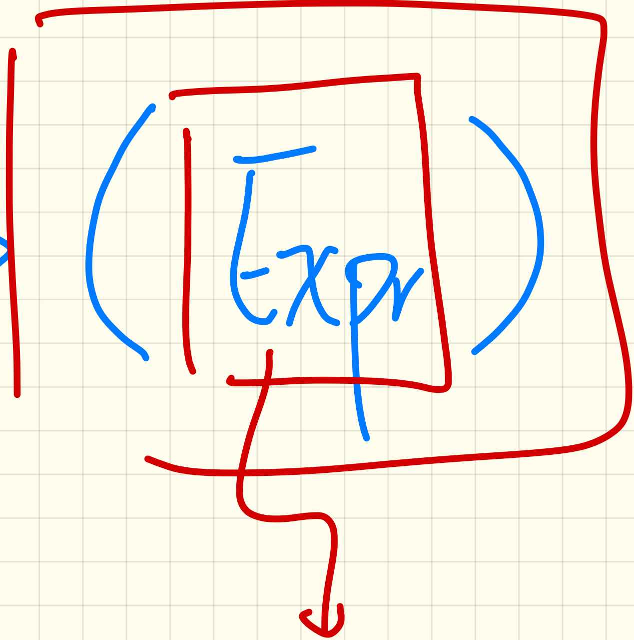
First( )) = {) }

"("

Follow(Expr) = {) }

Factor





assume:
Follow(Factor) already computed previously.

Compute now:
Follow(Expr)

First(___) → terminal, non-terminal

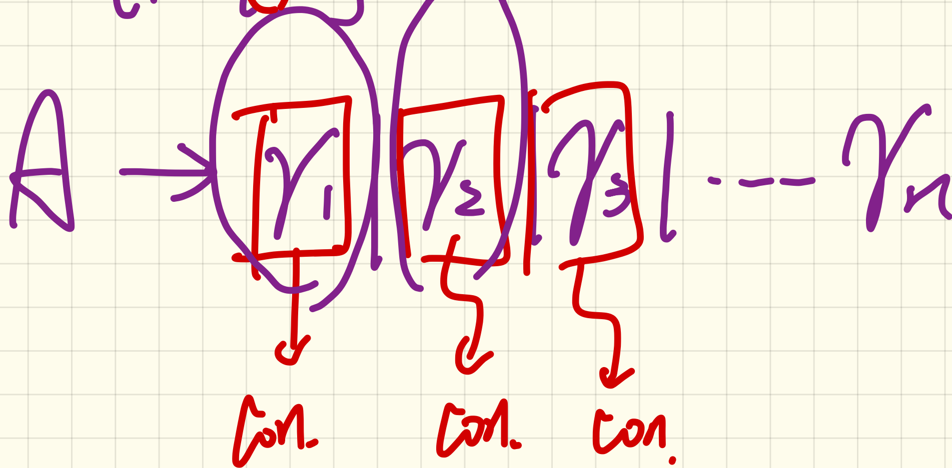
$First^{+}$( )

$\not{A}$(___)

First(___) → RHS of some production

Follow(___) → variable

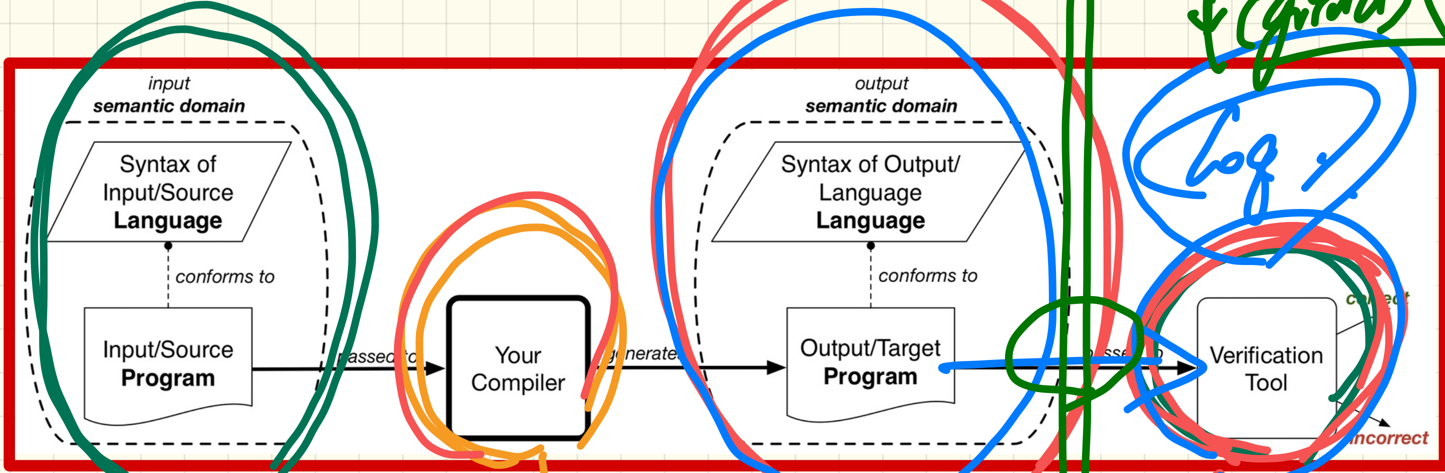
Backtrack-free Grammar



LECTURE 13

MONDAY, FEBRUARY 24

Project: Problem



Milestones of the Project

1. Confirm Team Member(s) and the Target Verification Tool

By the end of **Tuesday, March 3**, submit a plain text file `team.txt` for your team via the Prism account of a memer:

```
submit 4302 Project team.txt
```

2. Demonstrate Proficiency with the Chosen Target Verification Tool [**3%**]

– On **Thursday March 5** or **Friday March 6** (about 2 weeks after the project is released), your team is required to meet with Jackie and demonstrate that you are familiar with the verification tool (and its specification language) you choose.

– During the meeting, you must demonstrate (using your computer) **5 non-trivial examples** (ones that show the various target language features that are relevant to your compilation) of verification. For each example, you will demonstrate how to verify it using the tool.

– **In this meeting, Jackie may suggest specific tasks that your team should complete and will be included in the evaluation of the second milstone.**

3. Demonstrate Satisfactory Progress on the Compiler [**5%**]

– On **Thursday March 19** or **Friday March 20** (about 1 month after the project is released), your team is required to meet with Jackie and demonstrate a working version of your compiler on the following basic features of the source language features (of syntax of your own design), including:

- variable declarations
- variable assignments
- variable references (i.e., referring to declared variables in expressions)
- arithmetic, relational, and logical expressions
- conditionals
- specification (e.g., preconditions, postconditions, invariants, property assertions) in input programs that guide the target verification

– During the meeting, you must demonstrate (using your computer) **5 non-trivial examples** (ones that show the above source language features) of verification. For each example, you will demonstrate how to verify it using your compiler (e.g., given an input file, your tool will compile it into another file which can be taken as input by the target verification tool).

– **In this meeting, Jackie may suggest specific tasks that your team should complete and will be included in the evaluation of the final project in April.**

These two milestones are meant to make sure that you are on the right track. Based on your demo, Jackie will give you feedback.

Project: Milestones

April 6.

Project: Verification Tool

You must use the ANTLR4 Parser Generator (for Java) to build your compiler.

For the target verification tool, you must choose from one of the following (and confirm by the due date; see Section 7), where a suggested starting point is provided for each tool:

- PVS https://pvs.csl.sri.com/
  - This tool is available in Prism and used by *EECS4312 Software Engineering Requirements.*
  - More info here: `https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:pvs:start`
- Coq https://coq.inria.fr/
- Isabelle https://isabelle.in.tum.de/
- Alloy https://alloytools.org/
- PAT https://pat.comp.nus.edu.sg/
- Spin http://spinroot.com/spin/whatispin.html
- Z3 https://rise4fun.com/z3/tutorial

Nonetheless, if there is a particular verification tool which you prefer to working with but it is not in the above list, speak to Jackie by the due date of submitting the `team.txt` file (See Section 7 for the due date).

NASA

most u.f.c.

Cambridge

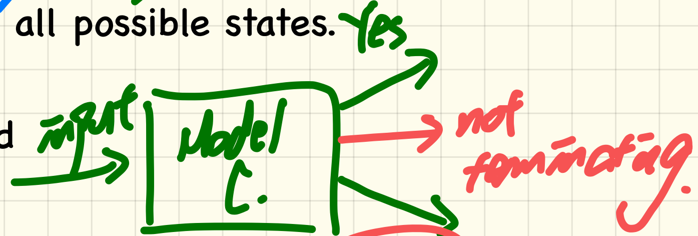
Radin

Paradigms of Verification (I)

Model Checking

- A transition graph is built based on all possible states.
- An algorithm is run to ensure that certain properties are satisfied
- Automated
- **State Explosion**:

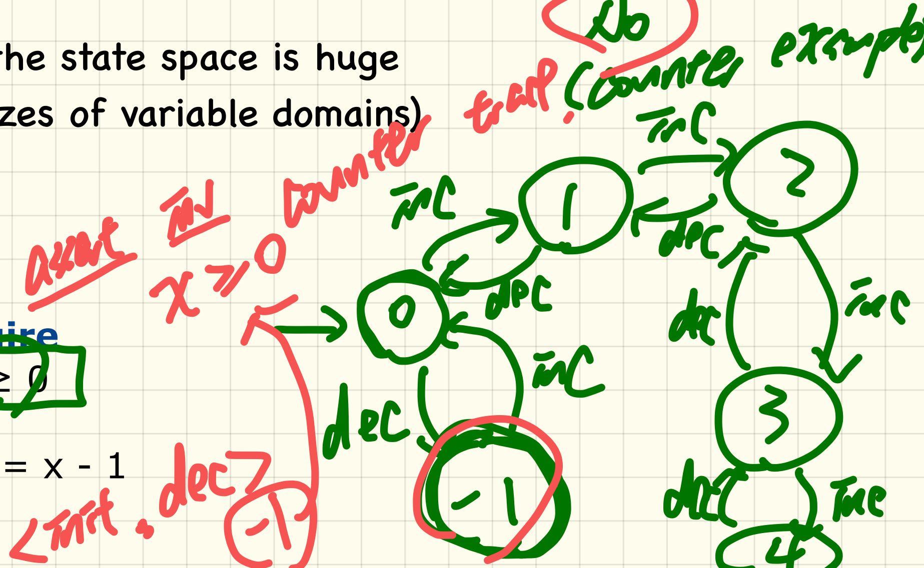
  not terminating if the state space is huge
  (combinatorial on sizes of variable domains)



Example:

```
inc
  require
    x < 5
  do
    x := x + 1
  end
```

```
dec
  require
    x ≥ 0
  do
    x := x - 1
  end
```



Paradigms of Verification (2)

Theorem Proving

- System is encoded as predicates (e.g., Hoare Triples in EECS3311)
- A proof system of deductions (axioms, lemmas) is used to prove that the system entails certain properties.
- Manual (EECS1090 proofs on a computer)
- Complete input domains of variables can be encoded.

Example:

```
swap
  require
    x > y
  do
    temp := x; x := y; y := temp
  ensure
    y > x
  end
```

Example:

Given $\neg(\neg p) \equiv p$

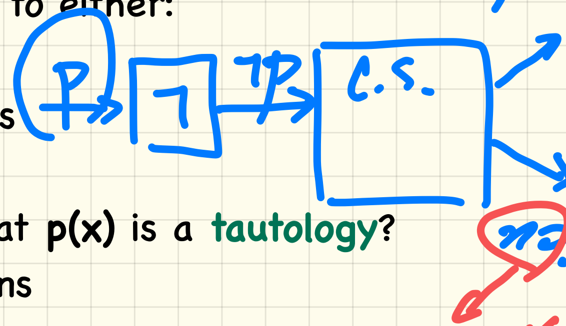
$p \Rightarrow q \equiv \neg p \vee q$

Prove: $\neg p \Rightarrow q \equiv p \vee q$

Paradigms of Verification (3)

Constraint Solving

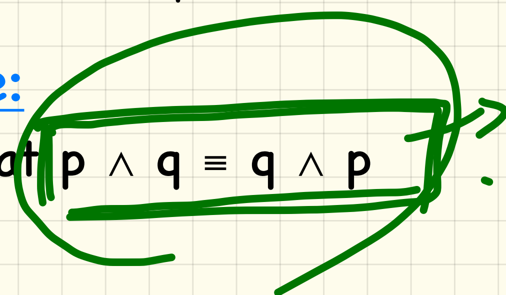
- System is encoded as predicates (e.g., Hoare Triples in EECS3311)
- Given predicate p(x), an algorithm is used to either:
  - (1) find a witness x s.t. p(x) is true.
  - (2) report that no such witness exists

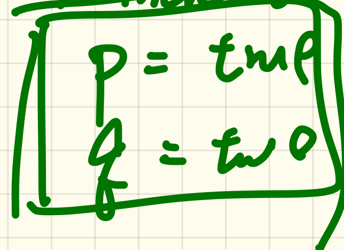


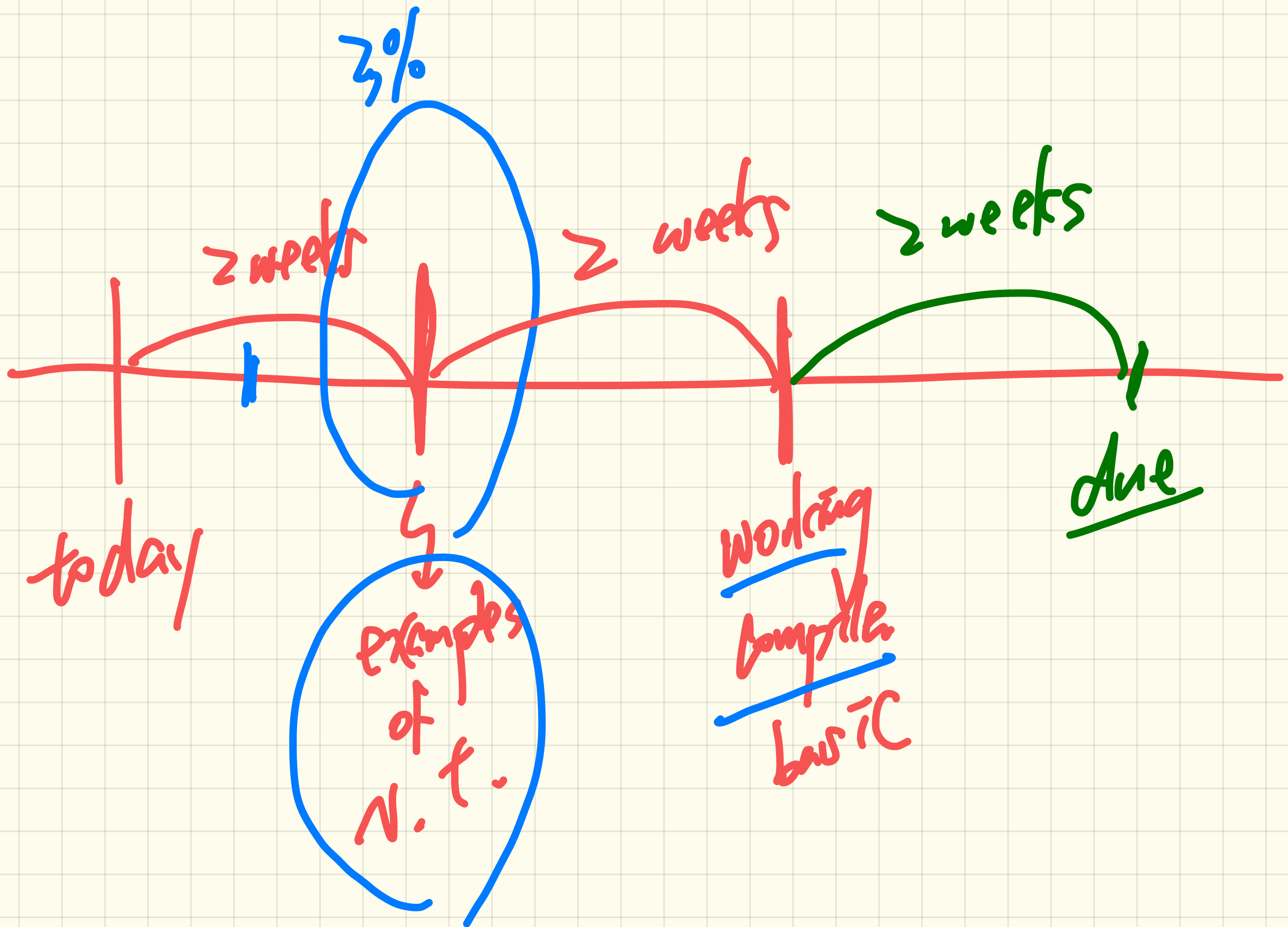
- Automated
- How do we then use a solver to prove that p(x) is a tautology?
- Combinatorial Explosion on Variable Domains

Example:

Prove that $p \wedge q \equiv q \wedge p$

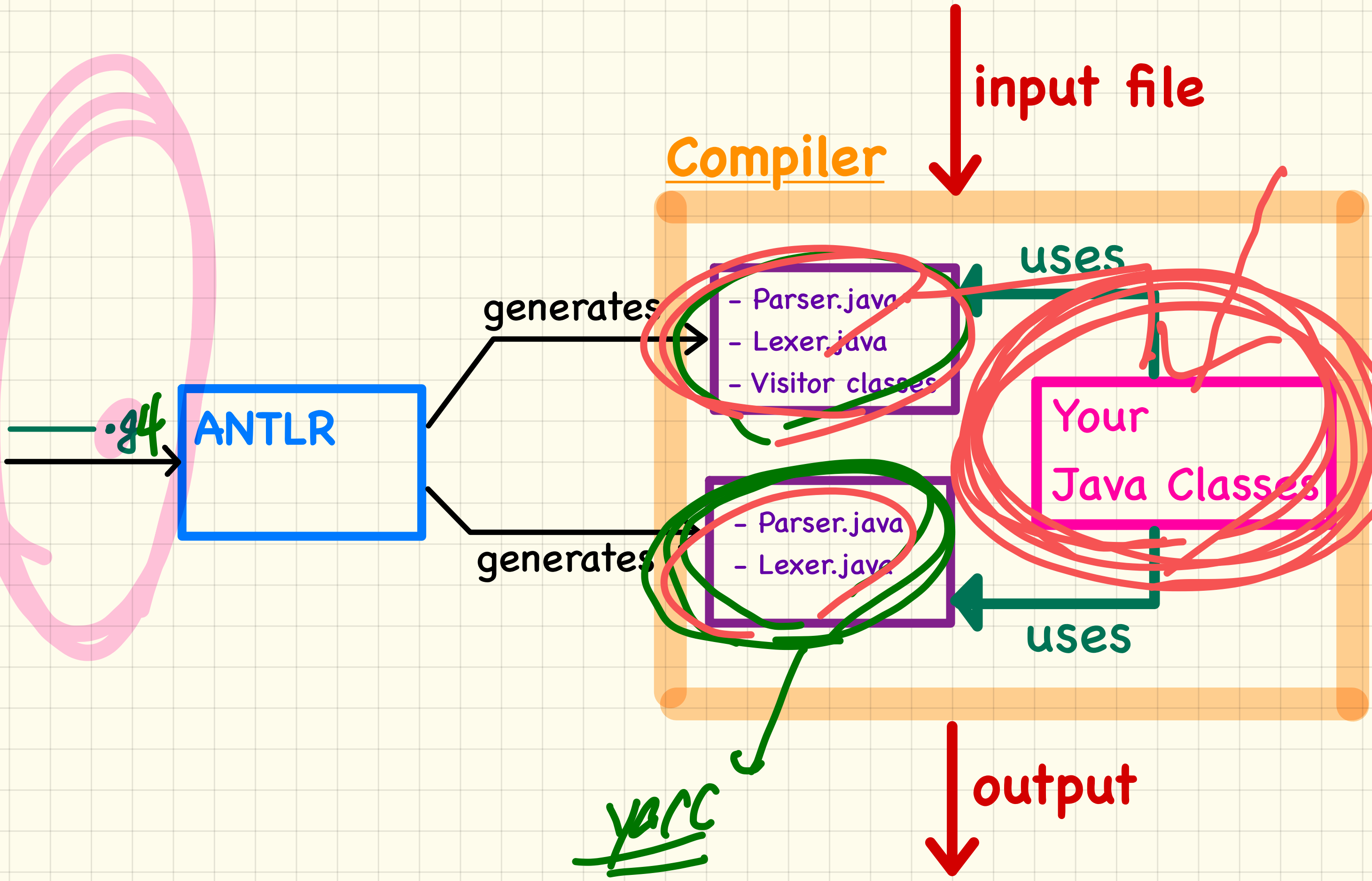


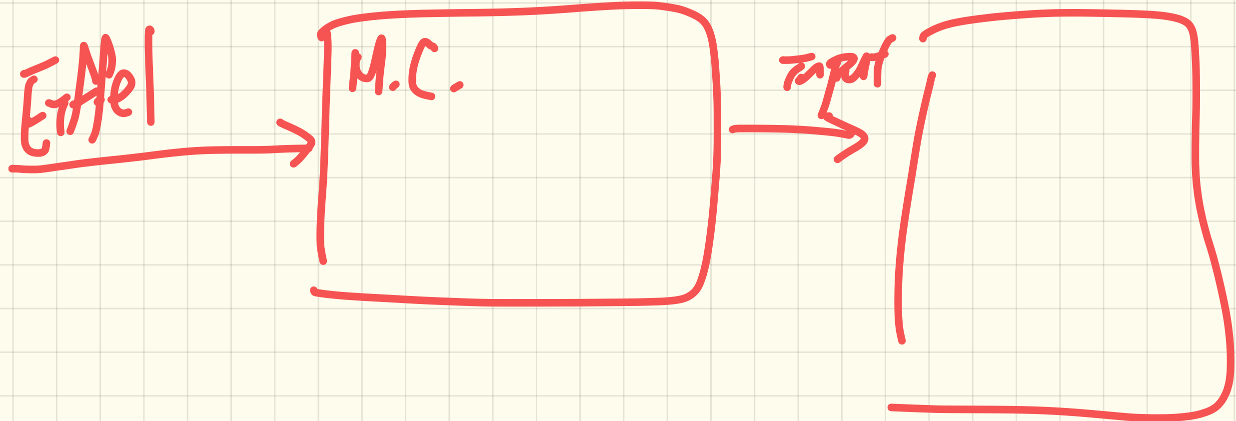




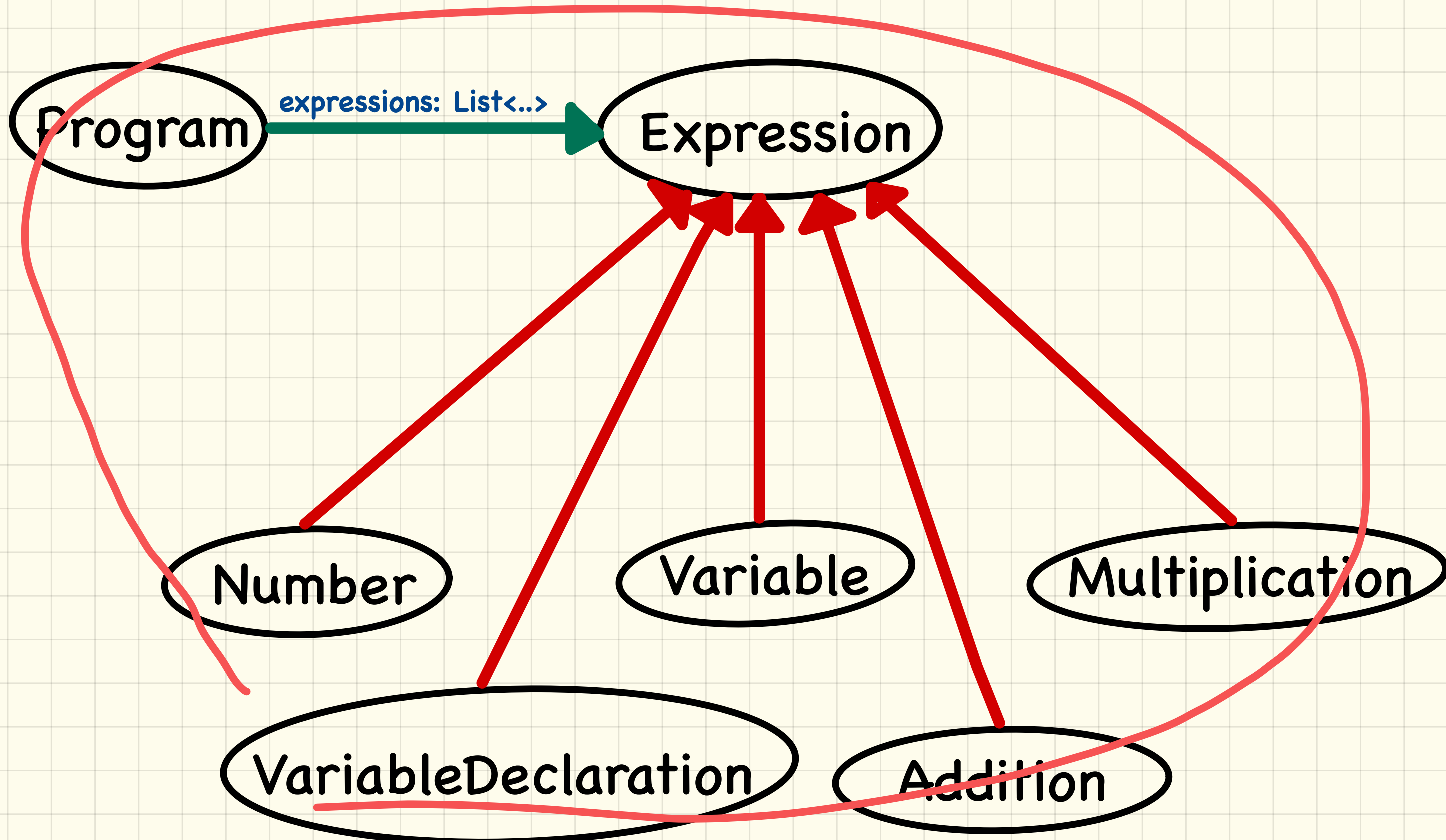
30%
2 weeks
2 weeks
2 weeks
today
examples of N. F.
working compiler basic
due

ANTLR (ANother Tool for Language Recognition)



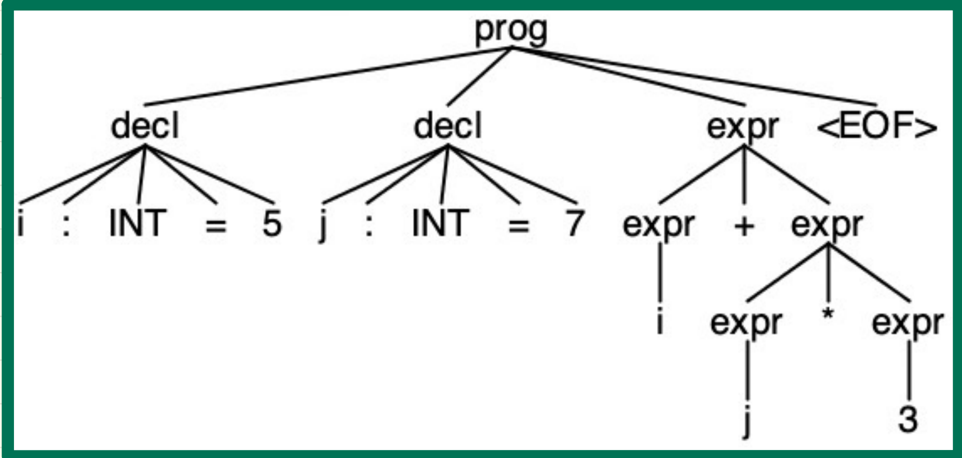

ExApel
M.C.
Input

Composite Pattern of Model Classes



Building Model Objects from Parse Trees

```
i : INT = 5
j : INT = 7
i + j * 3
```



prog

Program
expressions

0 1 2

VariableDeclaration
id
type
value

VariableDeclaration
id
type
value

Addition
left
right

Variable
id

Multiplication
left
right

Variable
id

Number
num

Backtrack-Free Grammar

$$\text{FIRST}^+(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \text{otherwise} \end{cases}$$

FIRST(β) is the extended version where β may be $\beta_1 \beta_2 \ldots \beta_n$

$A \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ satisfying:

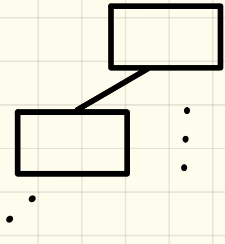
$$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \text{FIRST}^+(\gamma_i) \cap \text{FIRST}^+(\gamma_j) = \varnothing$$

Top-Down Parsing: Algorithm with lookahead

backtrack ≜ pop *focus*.siblings; *focus* := *focus*.parent; *focus*.resetChildren

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then % use FOLLOW set as well?
      if ∃ unvisited rule focus → β1β2...βn ∈ R ∧ word ∈ FIRST+(β) then
        create β1, β2 ... βn as children of focus
        trace.push(βnβn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	x *Factor Term'*
7		|	÷ *Factor Term'*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name



Term'

Backtrack-Free Grammar: Exercise

$$\text{FIRST}^+(A \to \beta) = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \text{otherwise} \end{cases}$$

FIRST(β) is the extended version where β may be $\beta_1 \beta_2 \ldots \beta_n$

$A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ satisfying:

$$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \text{FIRST}^+(\gamma_i) \cap \text{FIRST}^+(\gamma_j) = \varnothing$$

Is the following CFG ***backtrack free***?

11	*Factor*	$\to$	`name`
12		|	`name` [*ArgList*]
13		|	`name` (*ArgList*)
15	*ArgList*	$\to$	*Expr MoreArgs*
16	*MoreArgs*	$\to$	, *Expr MoreArgs*
17		|	ϵ

Left-Factoring: Removing Common Prefixes

Identify a common prefix α:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_j$$

[each of $\gamma_1, \gamma_2, \dots, \gamma_j$ does not begin with α]

Rewrite that production rule as:

$$\begin{aligned} A &\rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_j \\ B &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

11	*Factor*	$\rightarrow$	`name`
12		|	`name` [*ArgList*]
13		|	`name` (*ArgList*)
15	*ArgList*	$\rightarrow$	*Expr MoreArgs*
16	*MoreArgs*	$\rightarrow$	, *Expr MoreArgs*
17		|	ϵ

$F \rightarrow$ name F'

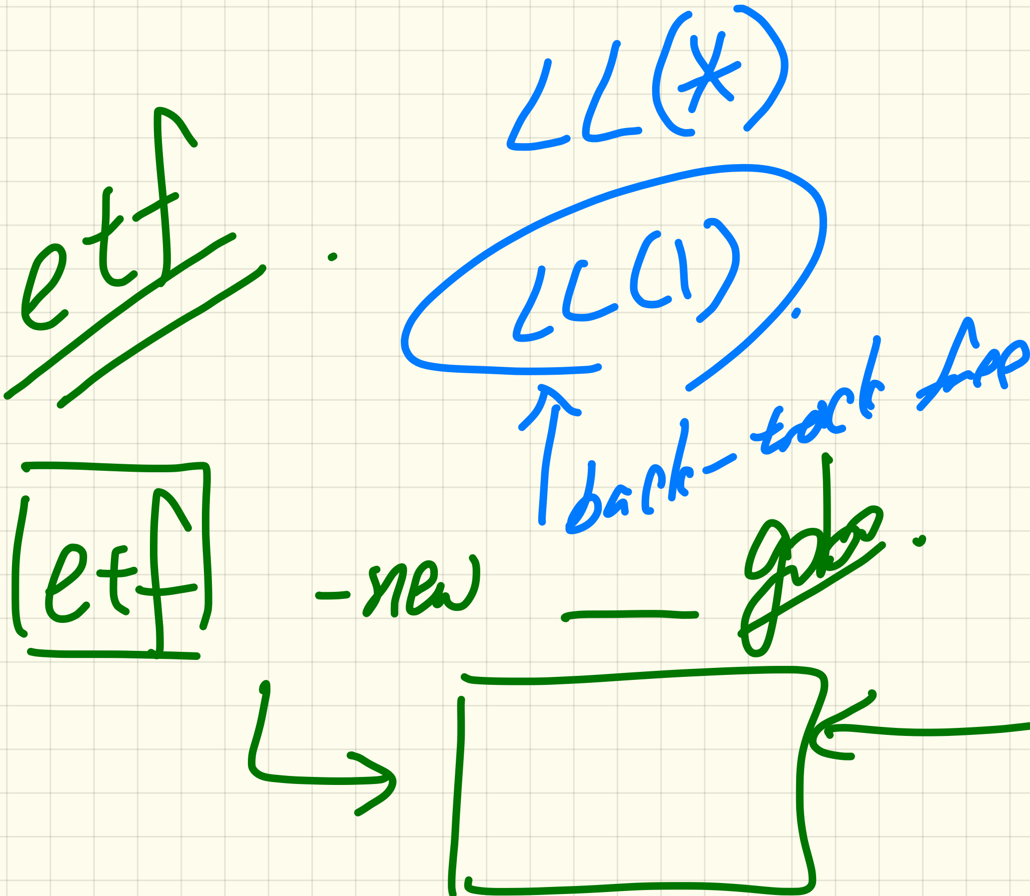
$F' \rightarrow \epsilon$
$\mid$ [Arglist]
$\mid$ (Arglist)

Implementing a Recursive-Descent Parser

generated by ANTLR4!

	Production	FIRST$^+$
2	$Expr' \rightarrow + \; Term \; Expr'$	{ + }
3	$\mid \; - \; Term \; Expr'$	{ - }
4	$\mid \; \epsilon$	{ ϵ, eof,) }

```
ExprPrim()
  if word = + ∨ word = - then /* Rules 2, 3 */
     word := NextWord()
     if(Term())
       then return ExprPrim()
       else return false
  elseif word = ) ∨ word = eof then /* Rule 4 */
     return true
  else
     report a syntax error
     return false
  end

Term()
  ...
```



etf
LL(*)
LL(1)
back-track free
etf
-new
goto

$$L \to Ra \mid Qba$$

$$R \to aba \mid caba \mid Rbc$$

$$Q \to bbc \mid bc$$

	alternative	First	intersection
L	Ra Qba		∅
R			
R'			
Q			
Q'			

left recursion

$$R \to abaR' \mid cabaR'$$

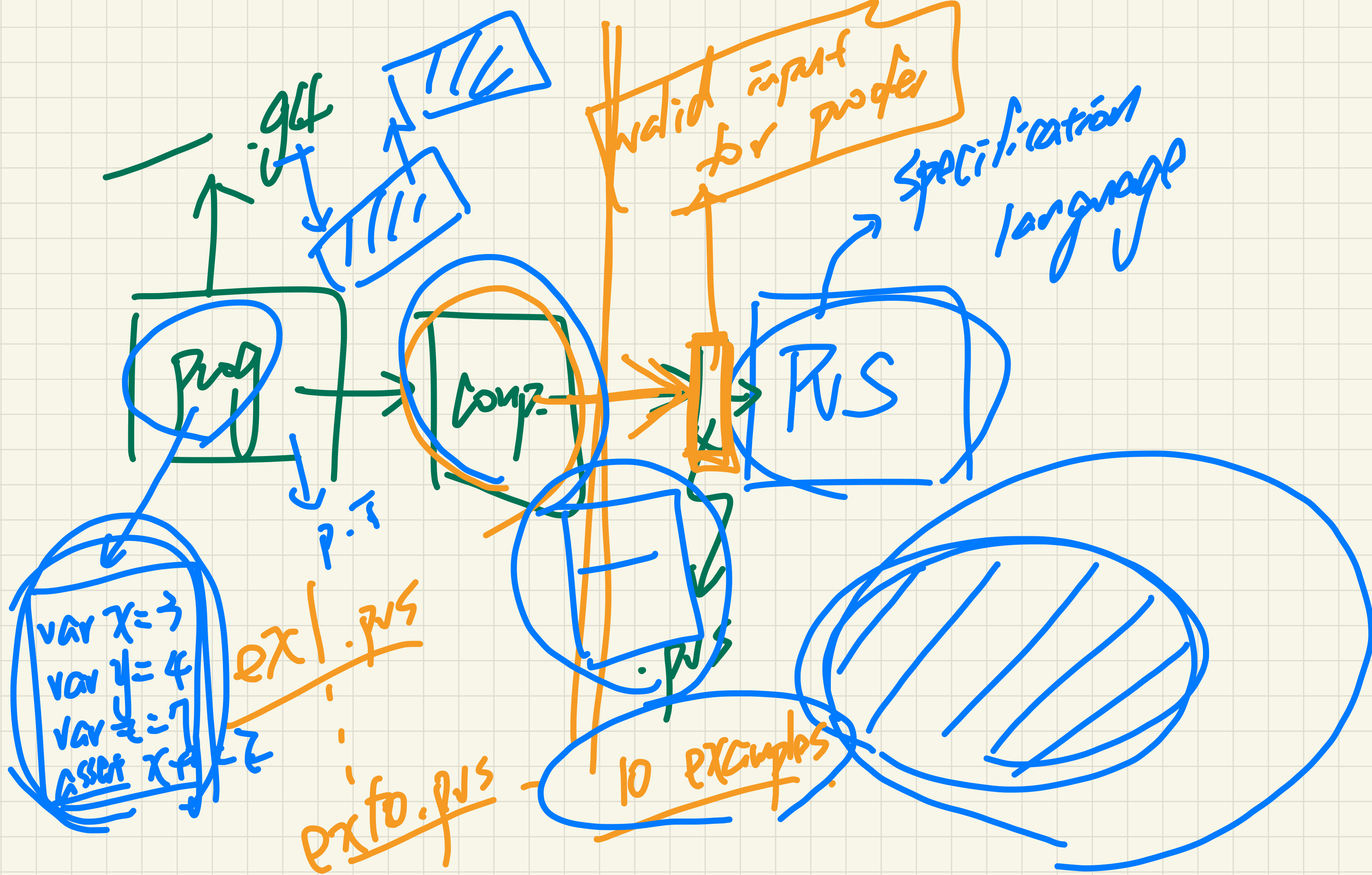
$$R' \to bcR' \mid \varepsilon$$

common prefix

$$Q \to bQ'$$

$$Q' \to bc \mid c$$

Lecture 14
Wednesday February 26



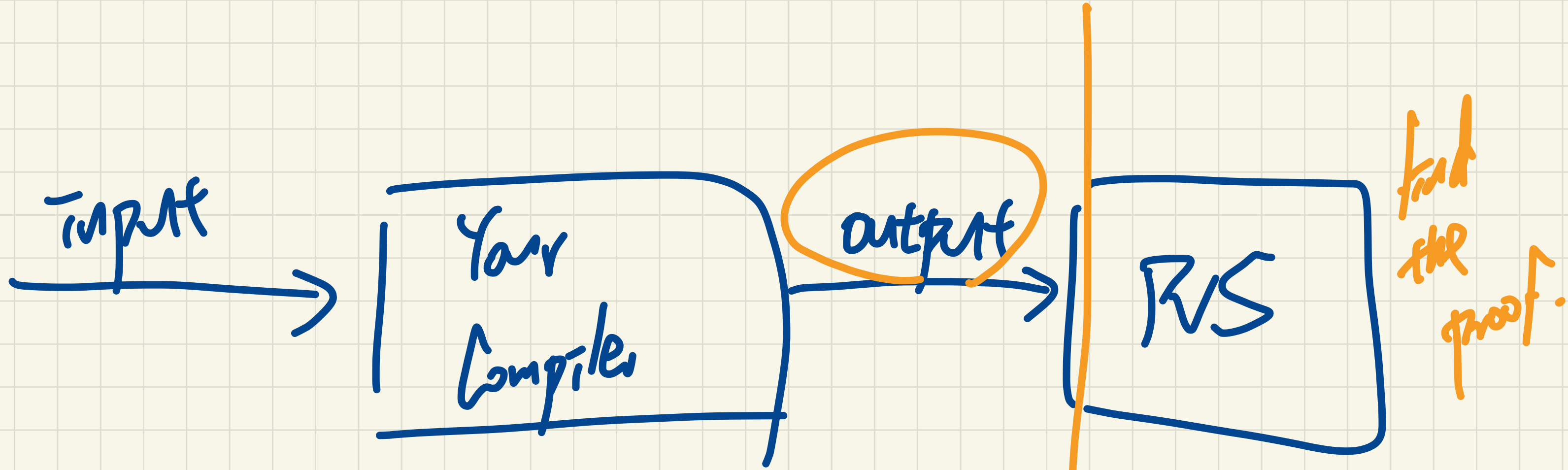
git
Prog
Comp
RuS
valid input for prover
specification language
var x=3
var y=4
var z=7
assert
ex1.rus
ex10.rus
10 examples

. raymond

fun nat square(. -)
do
return
od

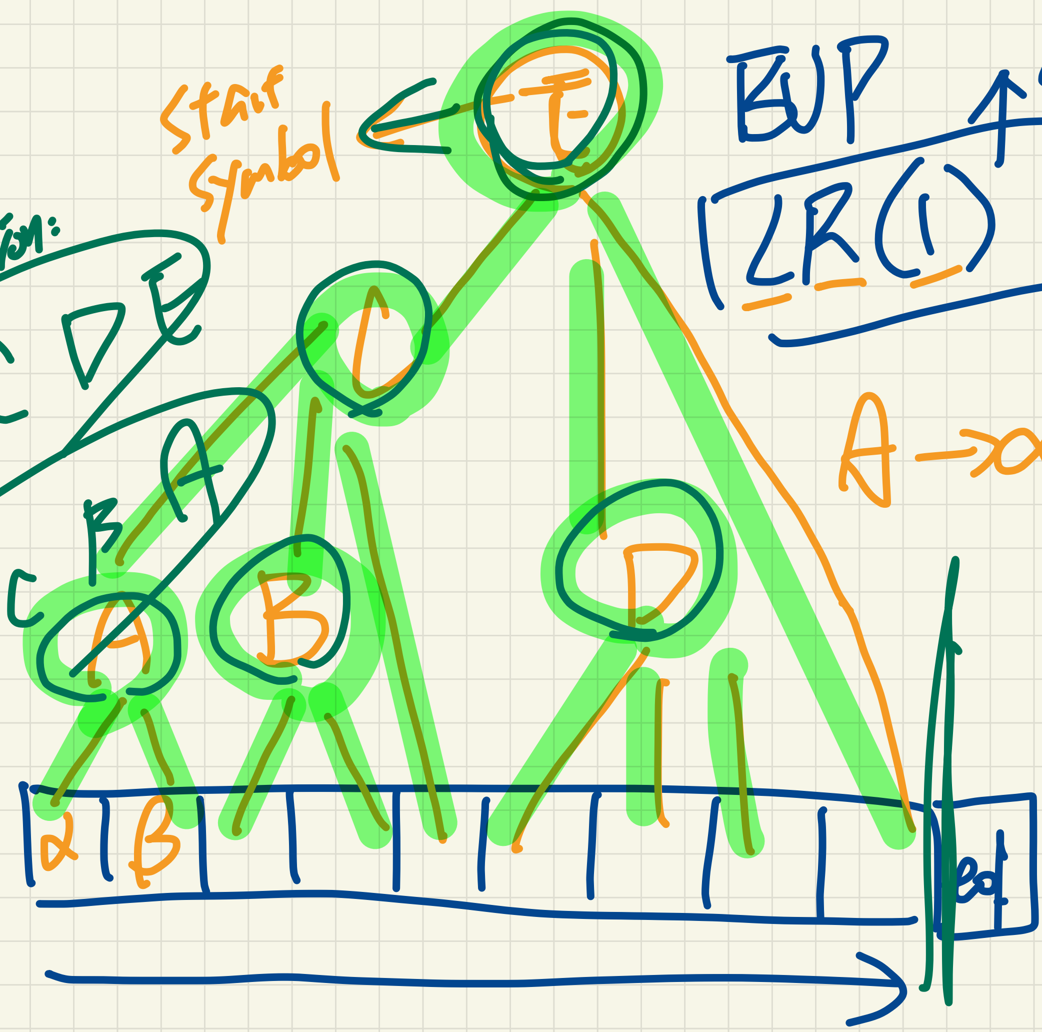
PVS

square(n: nat): nat = n * n

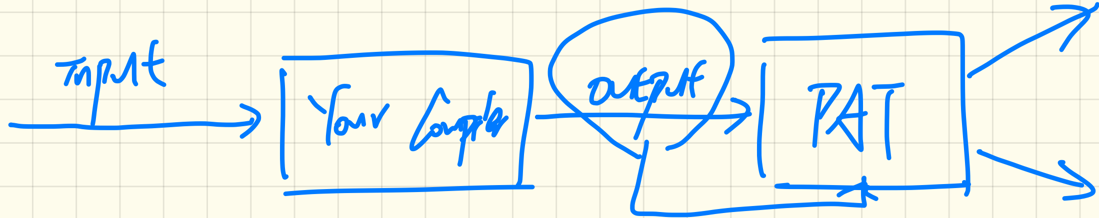


can be loaded ✓

not required to prove that the output is valid if your [illegible]

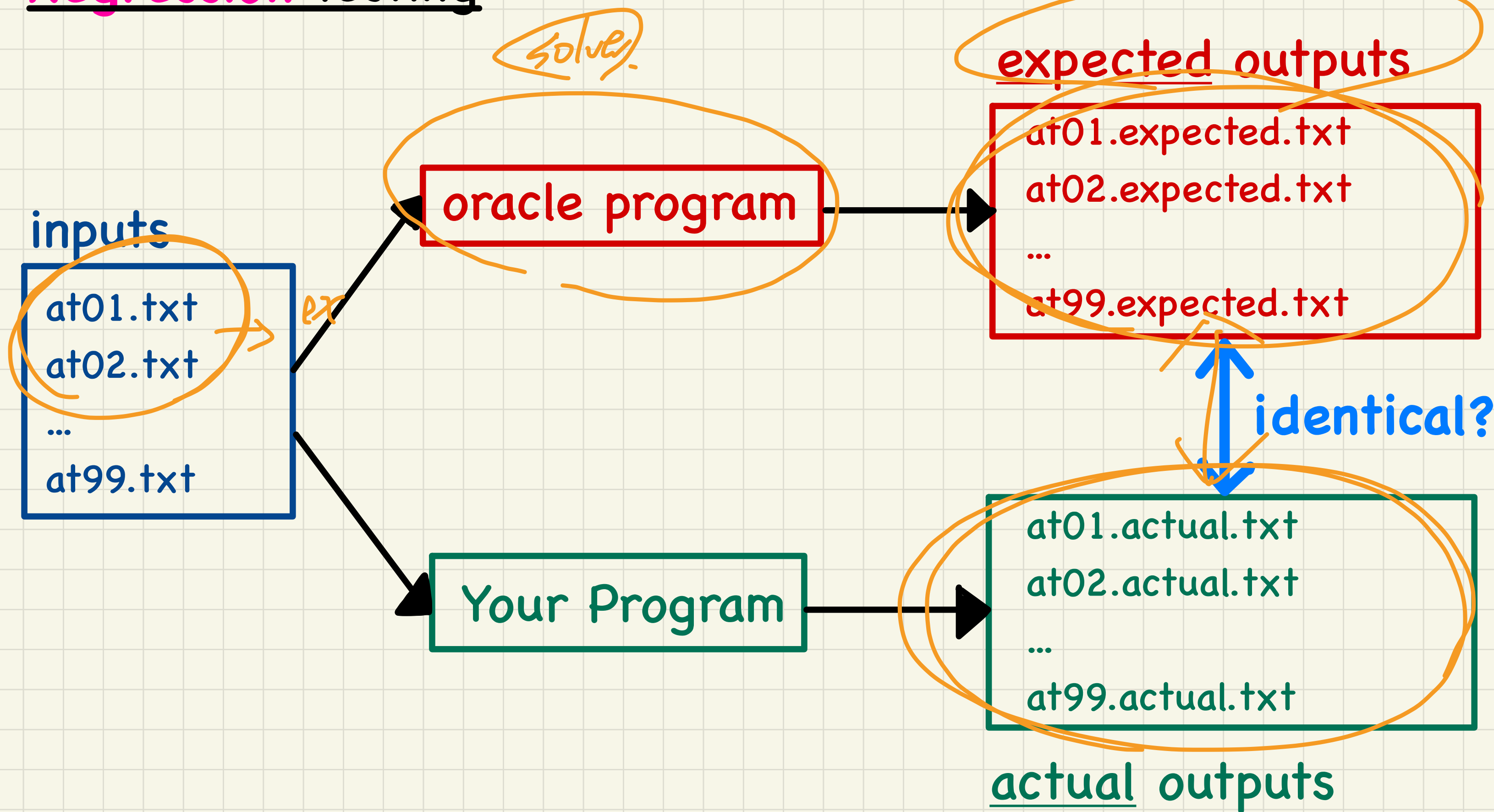

parse/reduction:
A B C D E
RMD.
E D C B A
scanner
start symbol
E
C
A
B
D
α
β
eof
BUP
LR(1)
LL(1)
A → αB

Lecture 15
Monday March 2

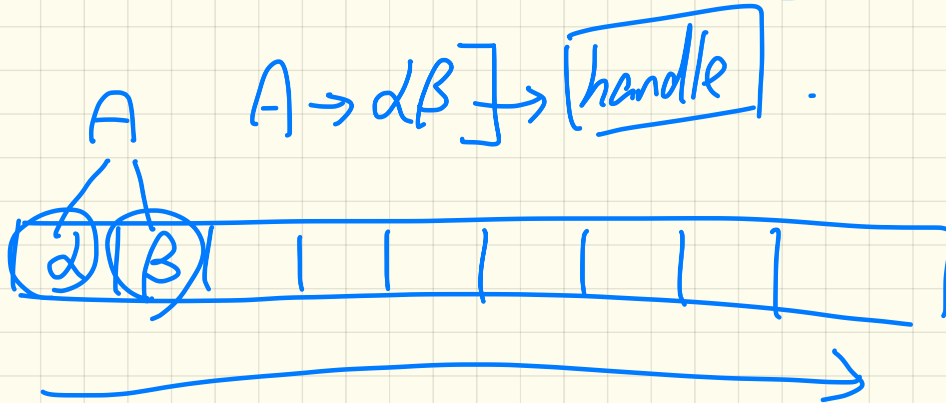


- Assignment 3: Automated Regression Testing

Regression Testing



$A \to \alpha\beta$



Bottom-Up Parsing: Algorithm

Actions: s3 → shift to state 3
r2 → reduce to LHS of rule 2

```
ALGORITHM: BUParse
  INPUT: CFG G = (V, Σ, R, S), Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(S) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce A → β'' then
      trace.pop() 2 × |β| times /* word + state */
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift s_i'' then
      trace.push(word)
      trace.push(s_i)
      word := NextWord()
    else
      fail()
```

|ab|=2

A → ab

trace.pop() * 4

	Rule
1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$\quad\mid Pair$
4	$Pair \rightarrow (\ Pair\ )$
5	$\quad\mid (\ )$

State	Action Table: eof	Action Table: (	Action Table:)	Goto Table: List	Goto Table: Pair
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

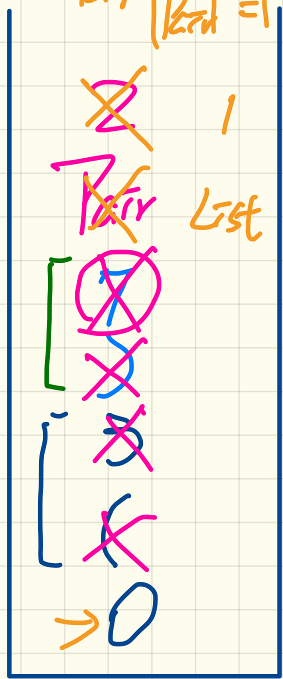
Bottom-Up Parsing: Discovering Rightmost Derivations (1)

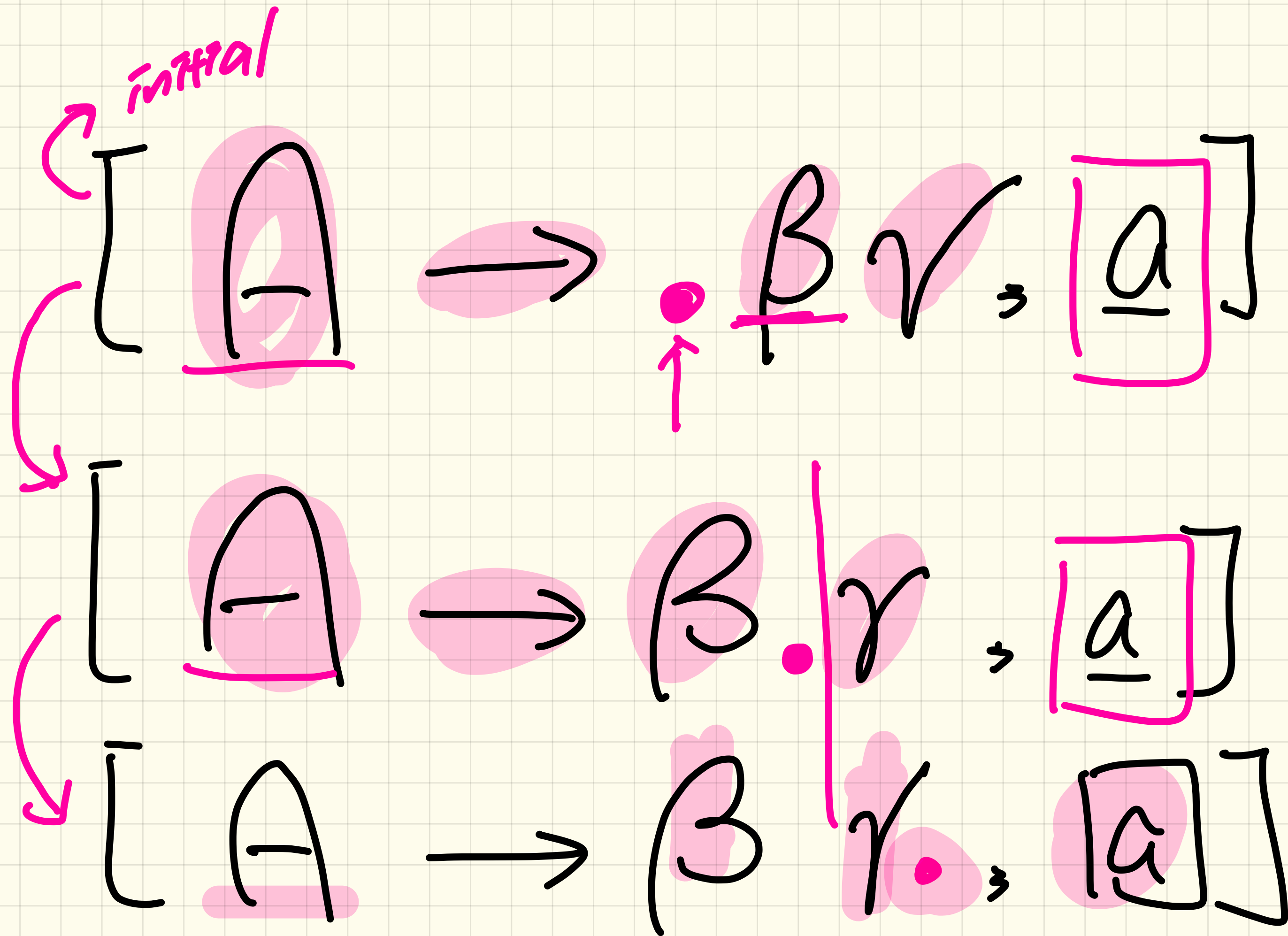
```
ALGORITHM: BUParse
  INPUT: CFG G = (V, Σ, R, S), Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(S) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce A → β'' then
      trace.pop() 2 × |β| times /* word + state */
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift s_i'' then
      trace.push(word)
      trace.push(s_i)
      word := NextWord()
    else
      fail()
```

1	Goal	→	List
2	List	→	List Pair
3		|	Pair
4	Pair	→	(Pair)
5		|	( )

State	Action Table			Goto Table	
	eof	(	)	List	Pair
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

Parse: ( ) eof




initial
[A → • β γ, a]
[A → β • γ, a]
[A → β γ •, a]

Bottom-Up Parsing: Discovering Rightmost Derivations (2)

Parse: (( )) ( )

```
ALGORITHM: BUParse
  INPUT: CFG G = (V, Σ, R, S), Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(S) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce A → β'' then
      trace.pop() 2 × |β| times /* word + state */
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift s_i'' then
      trace.push(word)
      trace.push(s_i)
      word := NextWord()
    else
      fail()
```

1	*Goal*	→	*List*	
2	*List*	→	*List Pair*	
3		|	*Pair*	
4	*Pair*	→	(*Pair*)	
5		|	( )	

State	Action Table: eof	Action Table: (	Action Table:)	Goto Table: List	Goto Table: Pair
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

Bottom-Up Parsing: Discovering Rightmost Derivations (3)

Parse: ( ))

```
ALGORITHM: BUParse
  INPUT: CFG G = (V, Σ, R, S), Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(S) /* start state */
  word := NextWord()
  while(true)
    state := trace.pop()
    act := Action[state, word]
    if act = ''accept'' then
      succeed()
    elseif act = ''reduce A → β'' then
      trace.pop() 2 × |β| times /* word + state */
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ''shift s_i'' then
      trace.push(word)
      trace.push(s_i)
      word := NextWord()
    else
      fail()
```

1	*Goal*	$\rightarrow$ *List*
2	*List*	$\rightarrow$ *List Pair*
3		| *Pair*
4	*Pair*	$\rightarrow$ (*Pair*)
5		| ( )

State	Action Table eof	(	)	Goto Table List	Pair
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

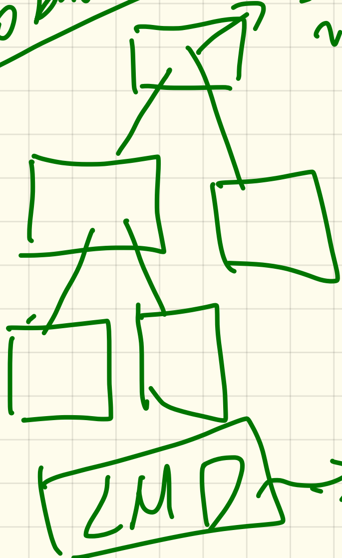
LECTURE 16

MONDAY MARCH 9

LL(1) vs. LR(1): Performance

TDP
no backtrack

start variab

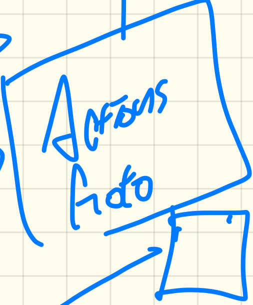


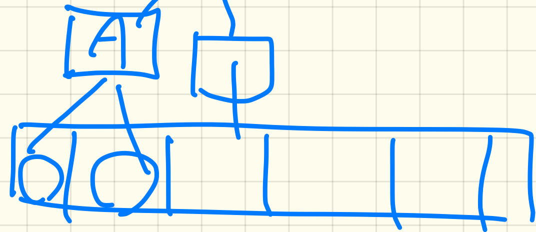
efforts to perform LMD

LR(1)

BUP

reverse order of RMD





LR(1) item.

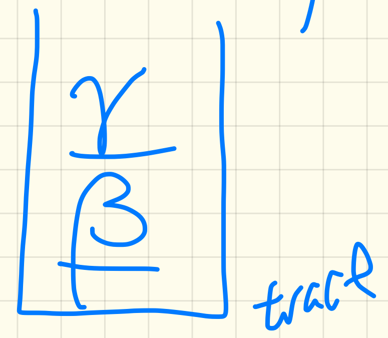
$[A \to \beta \bullet \gamma, a]$

top of stack of parser

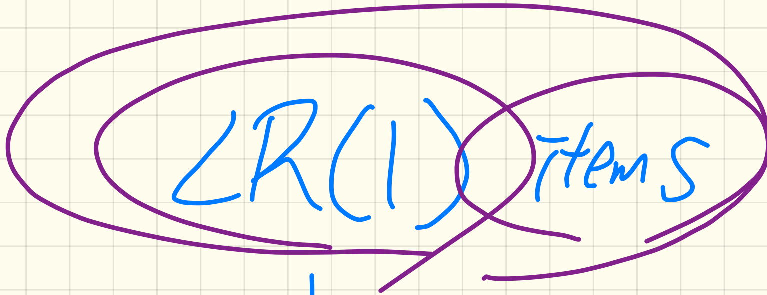
production rule

word (a)

lookahead symbol

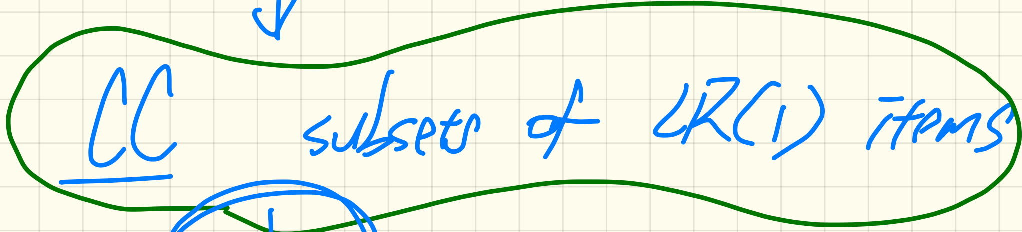


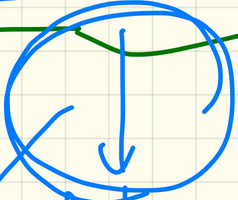
NFA



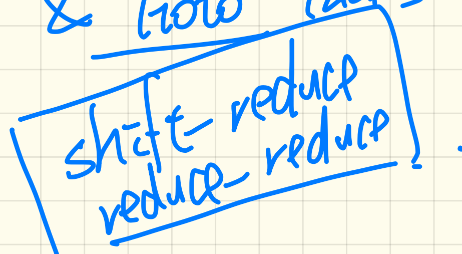
states of parser

DFA

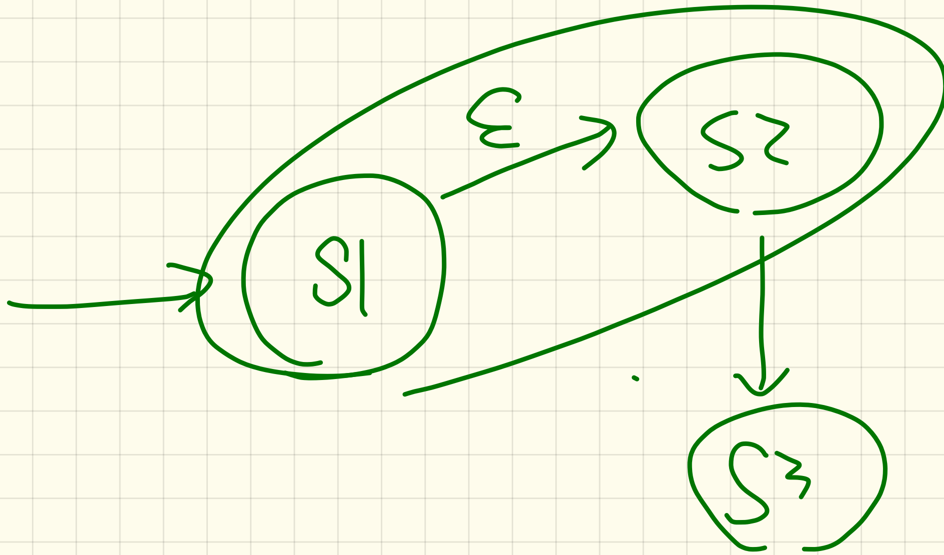




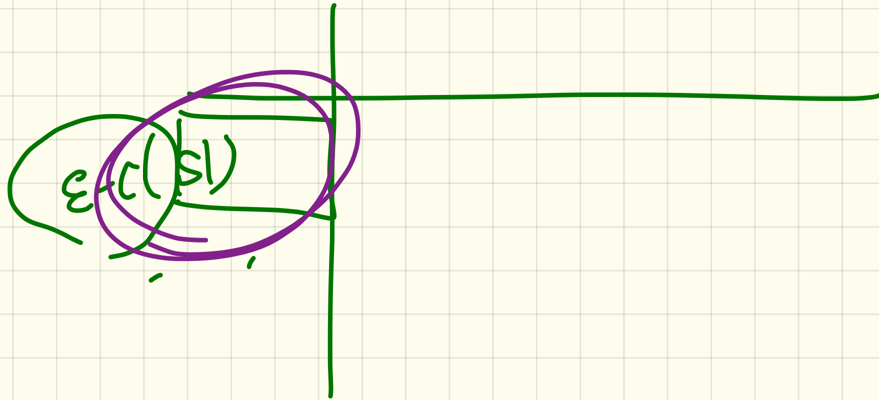
Fill in Actions & Goto tables



ambiguities



S1
ε
S2
S3



ε(S1)

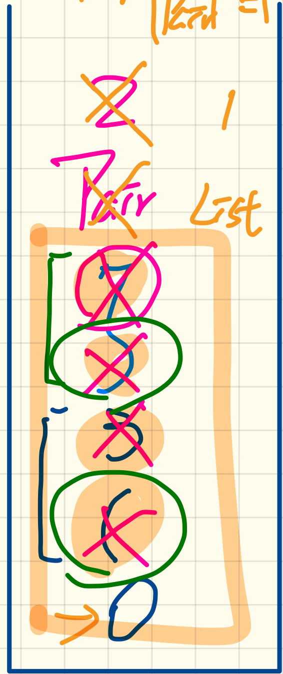
Bottom-Up Parsing: Discovering Rightmost Derivations (1)

```
ALGORITHM: BUParse
  INPUT: CFG G = (V, Σ, R, S), Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(S) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce A → β'' then
      trace.pop() 2 × |β| times /* word + state */
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift s_i'' then
      trace.push(word)
      trace.push(s_i)
      word := NextWord()
    else
      fail()
```

1	Goal	→	List	
2	List	→	List Pair	
3		|	Pair	
4	Pair	→	(Pair)	
5		|	( )	

State	Action Table: eof	(	)	Goto Table: List	Pair
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

Parse: ( ) eof



LR(1) Items: Exercise (1.1)

1	*Goal*	$\rightarrow$	*List*
2	*List*	$\rightarrow$	*List Pair*
3		|	*Pair*
4	*Pair*	$\rightarrow$	(*Pair*)
5		|	( )

Initial State: [*Goal* → •*List*, eof]

Desired Final State: [*Goal* → *List*•, eof]

Intermediate States: Subset Construction

Q. Derive all *LR(1) items* for the above grammar.

- FOLLOW(*List*) = {eof, (} FOLLOW(*Pair*) = {eof, (,)}

[• (Pair), eof]

[• (Pair), (]

[• (Pair),)]

LR(1) Items: Exercise (1.2)

1	*Goal*	$\rightarrow$	*List*
2	*List*	$\rightarrow$	*List Pair*
3		|	*Pair*
4	*Pair*	$\rightarrow$	(*Pair*)
5		|	( )

FOLLOW(*List*) = {eof, (}   FOLLOW(*Pair*) = {eof, (,)}

[*Goal* → • *List*,eof]

[*Goal* → *List* •,eof]

[*List* → • *List Pair*,eof]   [*List* → • *List Pair*,( ]
[*List* → *List* • *Pair*,eof]   [*List* → *List* • *Pair*,( ]
[*List* → *List Pair* •,eof]   [*List* → *List Pair* •,( ]

[*List* → • *Pair*,eof]   [*List* → • *Pair*,( ]
[*List* → *Pair* •,eof]   [*List* → *Pair* •,( ]

[*Pair* → • (*Pair*),eof]   [*Pair* → • (*Pair*),)]   [*Pair* → • (*Pair*),(]
[*Pair* → (• *Pair*),eof]   [*Pair* → (• *Pair*),)]   [*Pair* → (• *Pair*),(]
[*Pair* → (*Pair* •),eof]   [*Pair* → (*Pair* •),)]   [*Pair* → (*Pair* •),(]
[*Pair* → (*Pair*) •,eof]   [*Pair* → (*Pair*) •,)]   [*Pair* → (*Pair*) •,(]

[*Pair* → • ( ),eof]   [*Pair* → • ( ),(]   [*Pair* → • ( ),)]
[*Pair* → (•),eof]   [*Pair* → (•),(]   [*Pair* → (•),)]
[*Pair* → ( ) •,eof]   [*Pair* → ( ) •,(]   [*Pair* → ( ) •,)]

LR(1) Items: Exercise (2)

0	*Goal*	$\rightarrow$	*Expr*	6	*Term'*	$\rightarrow$	x *Factor Term'*
1	*Expr*	$\rightarrow$	*Term Expr'*	7		|	÷ *Factor Term'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*	8		|	ϵ
3		|	- *Term Expr'*	9	*Factor*	$\rightarrow$	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	$\rightarrow$	*Factor Term'*	11		|	name

FOLLOW Set

	Expr	Expr'	Term	Term'	Factor
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

|LR(1) items of R2| = 4 x 2 = 8

CC Construction: closure

```
1   ALGORITHM: closure
2     INPUT: CFG G = (V, Σ, R, S), a set s of LR(1) items
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     lastS := ∅
6     while (lastS ≠ s):
7       lastS := s
8       for [A → ⋯•Cδ, a] ∈ s:
9         for C → γ ∈ R:
10          for b ∈ FIRST(δa):
11            s := s ∪ { [C → •γ, b] }
12    return s
```

the next to match is C

γ can reduce to C

what can follow C:

$FIRST(\delta a) = \begin{cases} a & \text{if } \varepsilon \in FIRST(\delta) \\ FIRST(\delta) & \text{otherwise} \end{cases}$

make a new item from this production rule

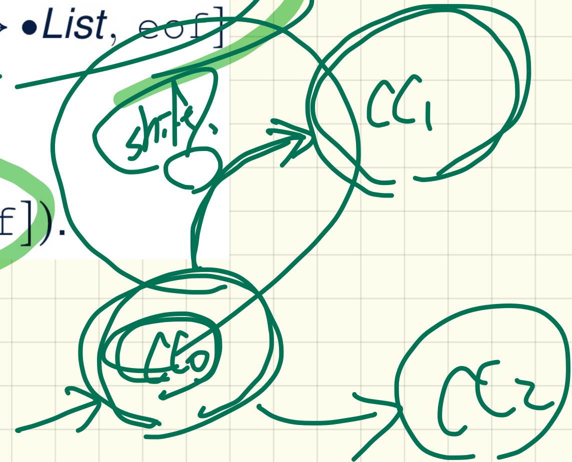
CC Construction: closure

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$\quad\mid Pair$
4	$Pair \rightarrow \underline{(}\ Pair\ \underline{)}$
5	$\quad\mid \underline{(}\ \underline{)}$

Initial State: $[Goal \rightarrow \bullet List,\ \texttt{eof}]$

Calculate $cc_0 = closure([Goal \rightarrow \bullet List,\ \texttt{eof}])$.

```
1   ALGORITHM: closure
2     INPUT: CFG G = (V, Σ, R, S), a set s of LR(1) items
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     lastS := ∅
6     while (lastS ≠ s) :
7       lastS := s
8       for [A → ··· • C δ, a] ∈ s:
9         for C → γ ∈ R:
10          for b ∈ FIRST(δa):
11            s := s ∪ { [ C → •γ, b] }
12    return s
```



1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$\mid\ Pair$
4	$Pair \rightarrow (\ Pair\ )$
5	$\mid\ (\ )$

Goal = A, ε = β, List = C, (ε) = δ, eof = a

$[A \rightarrow \beta \bullet C \delta, a]$

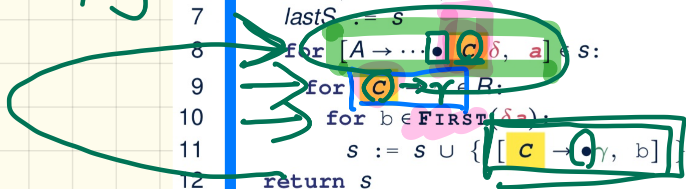
⓪ $[Goal \rightarrow \bullet List, eof]$ initial

$\text{FIRST}(\delta a) = \text{FIRST}(\varepsilon\ eof) = \{eof\}$

Step I

① $[List \rightarrow \bullet List\ Pair, eof]$

② $[List \rightarrow \bullet Pair, eof]$



```
1   ALGORITHM: closure
2     INPUT: CFG G = (V, Σ, R, S), a set s of LR(1) items
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     lastS := ∅
6     while (lastS ≠ s):
7       lastS := s
8       for [A → ··· • C δ, a] ∈ s:
9         for C → γ ∈ R:
10          for b ∈ FIRST(δa):
11            s := s ∪ {[C → • γ, b]}
12    return s
```

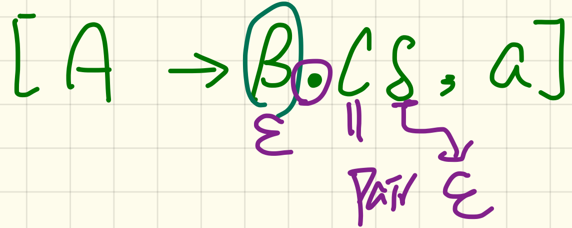
1	*Goal* → *List*
2	*List* → *List Pair*
3	| *Pair*
4	*Pair* → (*Pair*)
5	| ( )

Step 2

⓪ [Goal → • List , eof]

① [List → • List Pair , eof]

② [List → • Pair , eof]



FIRST(δa) = FIRST(ε eof) = {eof}

③ [Pair → • (Pair) , eof]

④ [Pair → • ( ) , eof]

$[A \to \beta \bullet C \delta, a]$

⓪ [Goal → • List , eof]

① [List → • List Pair , eof]

② [List → • Pair , eof]

③ [Pair → • (Pair) , eof]

④ [Pair → • ( ) , eof]

1	*Goal* → *List*
2	*List* → *List Pair*
3	| *Pair*
4	*Pair* → (*Pair*)
5	| ( )

Step 3

FIRST (δa) =

$[A \rightarrow \beta \cdot C\delta, a]$

⓪ $[Goal \rightarrow \cdot List, eof]$

① $[List \rightarrow \cdot List\ Pair, eof]$

② $[List \rightarrow \cdot Pair, eof]$

③ $[Pair \rightarrow \cdot (Pair), eof]$

④ $[Pair \rightarrow \cdot (\ ), eof]$

⑤ $[List \rightarrow \cdot List\ Pair, (]$

⑥ $[List \rightarrow \cdot Pair, (]$

1	*Goal*	$\rightarrow$ *List*
2	*List*	$\rightarrow$ *List Pair*
3		| *Pair*
4	*Pair*	$\rightarrow$ (*Pair*)
5		| ( )

Step 4

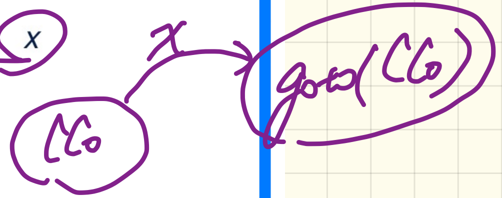
$|CC_0| = 9$

FIRST(δa) =

CC Construction: goto

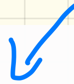
```
1  ALGORITHM: goto
2    INPUT: a set s of LR(1) items, a symbol x
3    OUTPUT: a set of LR(1) items
4  PROCEDURE:
5    moved := ∅
6    for item ∈ s:
7      if item = [α → β•xδ, a] then
8        moved := moved ∪ {[α → βx•δ, a]}
9      end
10   return closure(moved)
```

e.g. CC_0



state: - currently we are trying to reduce α (LHS)
- have already matched β
- next: x

CC Construction: goto



1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$\mid\ Pair$
4	$Pair \rightarrow \underline{(}\ Pair\ \underline{)}$
5	$\mid\ \underline{(}\ \underline{)}$

$$
cc_0 = \left\{ \begin{array}{lll} [Goal \rightarrow \bullet List, \texttt{eof}] & [List \rightarrow \bullet List\ Pair, \texttt{eof}] & [List \rightarrow \bullet List\ Pair, \underline{(}] \\ [List \rightarrow \bullet Pair, \texttt{eof}] & [List \rightarrow \bullet Pair, \underline{(}] & [Pair \rightarrow \bullet \underline{(}\ Pair\ \underline{)}, \texttt{eof}] \\ [Pair \rightarrow \bullet \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet \underline{(}\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \bullet \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}
$$

Calculate *goto*(cc_0, ( ). ["next state" from cc_0 taking ( ]

```
1   ALGORITHM: goto
2     INPUT: a set s of LR(1) items, a symbol x
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     moved := ∅
6     for item ∈ s:
7       if item = [α → β • xδ, a] then
8         moved := moved ∪ { [α → βx • δ, a] }
9       end
10    return closure(moved)
```

Lecture 17

Wednesday March 11

CC Construction: goto

```
1   ALGORITHM: goto
2     INPUT: a set s of LR(1) items, a symbol x
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     moved := ∅
6     for item ∈ s:
7       if item = [α → β • x δ, a] then
8         moved := moved ∪ { [α → β x • δ, a] }
9       end
10    return closure(moved)
```

CC Construction: goto

R4: [P → •(Pair),)]
R5: [(exercise)]

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$\mid\ Pair$
4	$Pair \rightarrow (\ Pair\ )$
5	$\mid\ (\ )$

$$cc_0 = \left\{ \begin{array}{lll} [Goal \rightarrow \bullet List, \texttt{eof}] & [List \rightarrow \bullet List\ Pair, \texttt{eof}] & [List \rightarrow \bullet List\ Pair, (] \\ [List \rightarrow \bullet Pair, \texttt{eof}] & [List \rightarrow \bullet Pair, (] & [Pair \rightarrow \bullet (Pair), \texttt{eof}] \\ [Pair \rightarrow \bullet (Pair), (] & [Pair \rightarrow \bullet ( ), \texttt{eof}] & [Pair \rightarrow \bullet ( ), (] \end{array} \right\}$$

FIRST( ) eof) = {) }

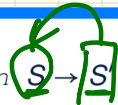
Calculate $goto(cc_0, ()$. ["next state" from cc_0 taking (]

```
1   ALGORITHM: goto
2     INPUT: a set s of LR(1) items, a symbol x
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     moved := ∅
6     for item ∈ s:
7       if item = [α → β • x δ, a] then
8         moved := moved ∪ { [α → β x • δ, a] }
9       end
10    return closure(moved)
```

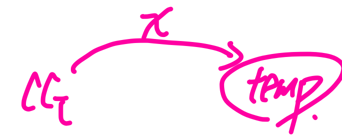
6: [P → (• P), eof]
7: [P → (• P), (]
8: [P → (•), eof]
9: [P → (•), (]

CC Construction: Algorithm

```
1   ALGORITHM: BuildCC
2     INPUT: a grammar G = (V, Σ, R, S),  goal production S → S'
3     OUTPUT:
4     (1) a set CC = {cc0, cc1, ..., ccn} where cci ⊆ G's LR(1) items
5     (2) a transition function
6   PROCEDURE:
7     cc0 := closure({[S' → •S, eof]})
8     CC := {cc0}
9     processed := {cc0}
10    lastCC := ∅
11    while (lastCC ≠ CC) :
12      lastCC := CC
13      for cci s.t. cci ∈ CC ∧ cci ∉ processed :
14        processed := processed ∪ {cci}
15        for x s.t. [··· → ··· • x ...] ∈ cci
16          temp := goto(cci, x)
17          if temp ∉ CC then
18            CC := CC ∪ {temp}
19          end
20          δ := δ ∪ (cci, x, temp)
```



initial state (cc_0) to start with.



transition

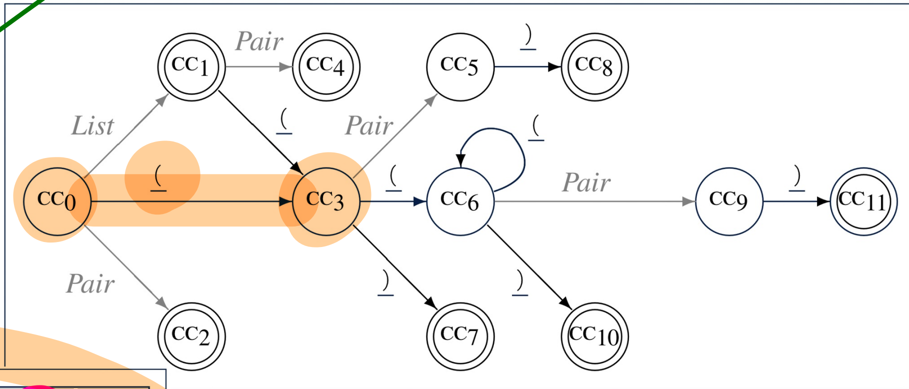
resulting set

CC Construction: Algorithm

1	*Goal*	$\rightarrow$	*List*
2	*List*	$\rightarrow$	*List Pair*
3		$\mid$	*Pair*
4	*Pair*	$\rightarrow$	$\underline{(}$ *Pair* $\underline{)}$
5		$\mid$	$\underline{(}$ $\underline{)}$

- Calculate $\mathcal{CC} = \{cc_0, cc_1, \ldots, cc_{11}\}$
- Calculate the transition function $\delta : \mathcal{CC} \times \Sigma \rightarrow \mathcal{CC}$

CC Construction: Algorithm



Iteration	Item	Goal	List	Pair	(	)	eof
0	cc_0	∅	cc_1	cc_2	cc_3	∅	∅
1	cc_1	∅	∅	cc_4	cc_3	∅	∅
	cc_2	∅	∅	∅	∅	∅	∅
	cc_3	∅	∅	cc_5	cc_6	cc_7	∅
2	cc_4	∅	∅	∅	∅	∅	∅
	cc_5	∅	∅	∅	∅	cc_8	∅
	cc_6	∅	∅	cc_9	cc_6	cc_{10}	∅
	cc_7	∅	∅	∅	∅	∅	∅
3	cc_8	∅	∅	∅	∅	∅	∅
	cc_9	∅	∅	∅	∅	cc_{11}	∅
	cc_{10}	∅	∅	∅	∅	∅	∅
4	cc_{11}	∅	∅	∅	∅	∅	∅

CC Construction: Algorithm

$$CC_0 = \left\{ \begin{array}{ccc} [Goal \to \bullet\, List,\ \texttt{eof}] & [List \to \bullet\, List\ Pair,\ \texttt{eof}] & [List \to \bullet\, List\ Pair,\ \underline{(}] \\ [List \to \bullet\, Pair,\ \texttt{eof}] & [List \to \bullet\, Pair,\ \underline{(}] & [Pair \to \bullet\, \underline{(}\ Pair\ \underline{)},\ \texttt{eof}] \\ [Pair \to \bullet\, \underline{(}\ Pair\ \underline{)},\underline{(}] & [Pair \to \bullet\, \underline{(}\ \underline{)},\ \texttt{eof}] & [Pair \to \bullet\, \underline{(}\ \underline{)},\underline{(}] \end{array} \right\}$$

$$CC_1 = \left\{ \begin{array}{ccc} [Goal \to List\, \bullet,\ \texttt{eof}] & [List \to List\, \bullet\, Pair,\ \texttt{eof}] & [List \to List\, \bullet\, Pair,\ \underline{(}] \\ [Pair \to \bullet\, \underline{(}\ Pair\ \underline{)},\ \texttt{eof}] & [Pair \to \bullet\, \underline{(}\ Pair\ \underline{)},\ \underline{(}] & [Pair \to \bullet\, \underline{(}\ \underline{)},\ \texttt{eof}] \\ & [Pair \to \bullet\, \underline{(}\ \underline{)},\ \underline{(}] & \end{array} \right\}$$

$$CC_2 = \left\{ [List \to Pair\, \bullet,\ \texttt{eof}] \quad [List \to Pair\, \bullet,\ \underline{(}] \right\}$$

$$CC_3 = \left\{ \begin{array}{ccc} [Pair \to \bullet\, \underline{(}\ Pair\ \underline{)},\ \underline{)}] & [Pair \to \underline{(}\, \bullet\, Pair\ \underline{)},\ \texttt{eof}] & [Pair \to \underline{(}\, \bullet\, Pair\ \underline{)},\ \underline{(}] \\ [Pair \to \bullet\, \underline{(}\ \underline{)},\ \underline{)}] & [Pair \to \underline{(}\, \bullet\, \underline{)},\ \texttt{eof}] & [Pair \to \underline{(}\, \bullet\, \underline{)},\ \underline{(}] \end{array} \right\}$$

$$CC_4 = \left\{ [List \to List\ Pair\, \bullet,\ \texttt{eof}] \quad [List \to List\ Pair\, \bullet,\ \underline{(}] \right\}$$

$$CC_5 = \left\{ [Pair \to \underline{(}\ Pair\, \bullet\, \underline{)},\ \texttt{eof}] \quad [Pair \to \underline{(}\ Pair\, \bullet\, \underline{)},\ \underline{(}] \right\}$$

$$CC_6 = \left\{ \begin{array}{cc} [Pair \to \bullet\, \underline{(}\ Pair\ \underline{)},\ \underline{)}] & [Pair \to \underline{(}\, \bullet\, Pair\ \underline{)},\ \underline{)}] \\ [Pair \to \bullet\, \underline{(}\ \underline{)},\ \underline{)}] & [Pair \to \underline{(}\, \bullet\, \underline{)},\ \underline{)}] \end{array} \right\}$$

$$CC_7 = \left\{ [Pair \to \underline{(}\ \underline{)}\, \bullet,\ \texttt{eof}] \quad [Pair \to \underline{(}\ \underline{)}\, \bullet,\ \underline{(}] \right\}$$

CC8

$$CC_8 = \left\{ [Pair \to \underline{(}\ Pair\ \underline{)}\, \bullet,\ \texttt{eof}] \quad [Pair \to \underline{(}\ Pair\ \underline{)}\, \bullet,\ \underline{(}] \right\}$$

$$CC_9 = \left\{ [Pair \to \underline{(}\ Pair\, \bullet\, \underline{)},\ \underline{)}] \right\}$$

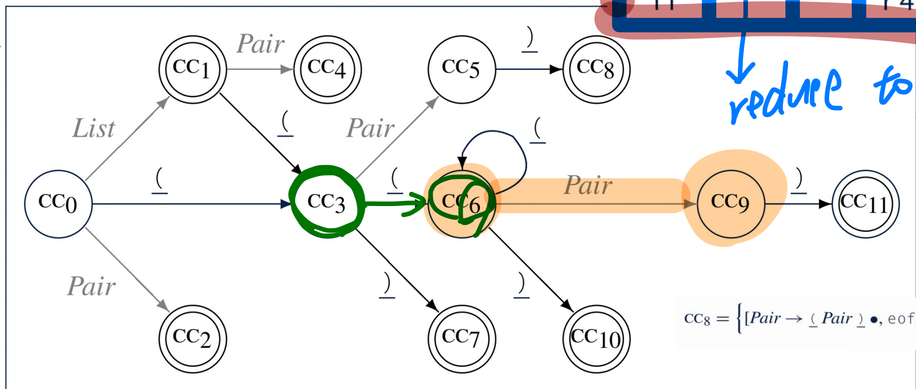
$$CC_{10} = \left\{ [Pair \to \underline{(}\ \underline{)}\, \bullet,\ \underline{)}] \right\}$$

$$CC_{11} = \left\{ [Pair \to \underline{(}\ Pair\ \underline{)}\, \bullet,\ \underline{)}] \right\}$$

Table Construction: Algorithm

```
1   ALGORITHM: BuildActionGotoTables
2     INPUT:
3       (1) a grammar G = (V, Σ, R, S)
4       (2) goal production S → S'
5       (3) a canonical collection CC = {cc_0, cc_1, ..., cc_n}
6       (4) a transition function δ : CC × Σ → CC
7     OUTPUT: Action Table & Goto Table
8   PROCEDURE:
9     for cc_i ∈ CC:
10      for item ∈ cc_i:
11        if item = [A → β • x γ, a] ∧ δ(cc_i, x) = cc_j then
12          Action[i, x] := shift j
13        elseif item = [A → β •, a] then
14          Action[i, a] := reduce A → β
15        elseif item = [S → S' •, eof] then
16          Action[i, eof] := accept
17        end
18      for v ∈ V:
19        if δ(cc_i, v) = cc_j then
20          Goto[i, v] = j
21        end
```

State	Action Table			Goto Table	
	eof	(	)	List	Pair
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		



$cc_8 = \{[Pair \rightarrow \underline{(}\ Pair\ \underline{)}\ \bullet, \texttt{eof}]\quad [Pair \rightarrow \underline{(}\ Pair\ \underline{)}\ \bullet, \underline{(}]\}$

TDP

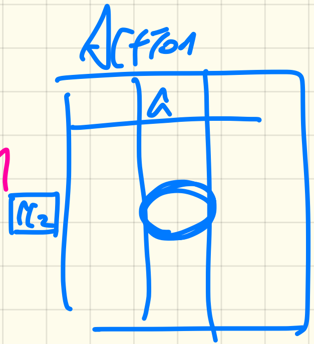
↓ g

no left rec.

LR(1)

↓ back track free

BUP



↓ skeleton grammar

LR(1) items

CC (≈ possible states of NFA)

↓

CC_0 - - -

transition

↓ Actions, Goto → deterministic

$$CC_{13} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt \bullet \texttt{ ,\{eof,else\}}], \\ [Stmt \rightarrow \texttt{if expr then } Stmt \bullet \texttt{ else } Stmt, \texttt{\{eof,else\}}] \end{array} \right\}$$

Action

13

① [Stmt → if expr then Stmt •, eof]

② [Stmt → if expr then Stmt •, else]

③ [Stmt → if expr then Stmt • else Stmt, eof]

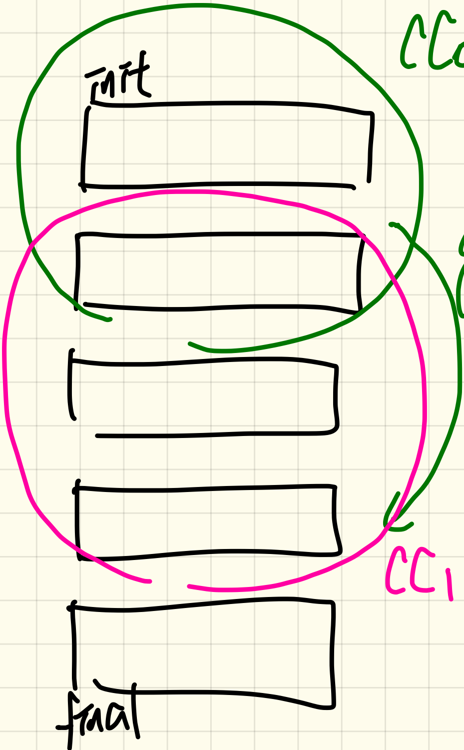
④ [Stmt → if expr then Stmt • else Stmt, else]

word

else

r.

LR(1) items



$goto(CC_0, x)$

CC

each member is a subset of LR(1) items.