EECS4302
Compilers and Interpreters

Fall 2022

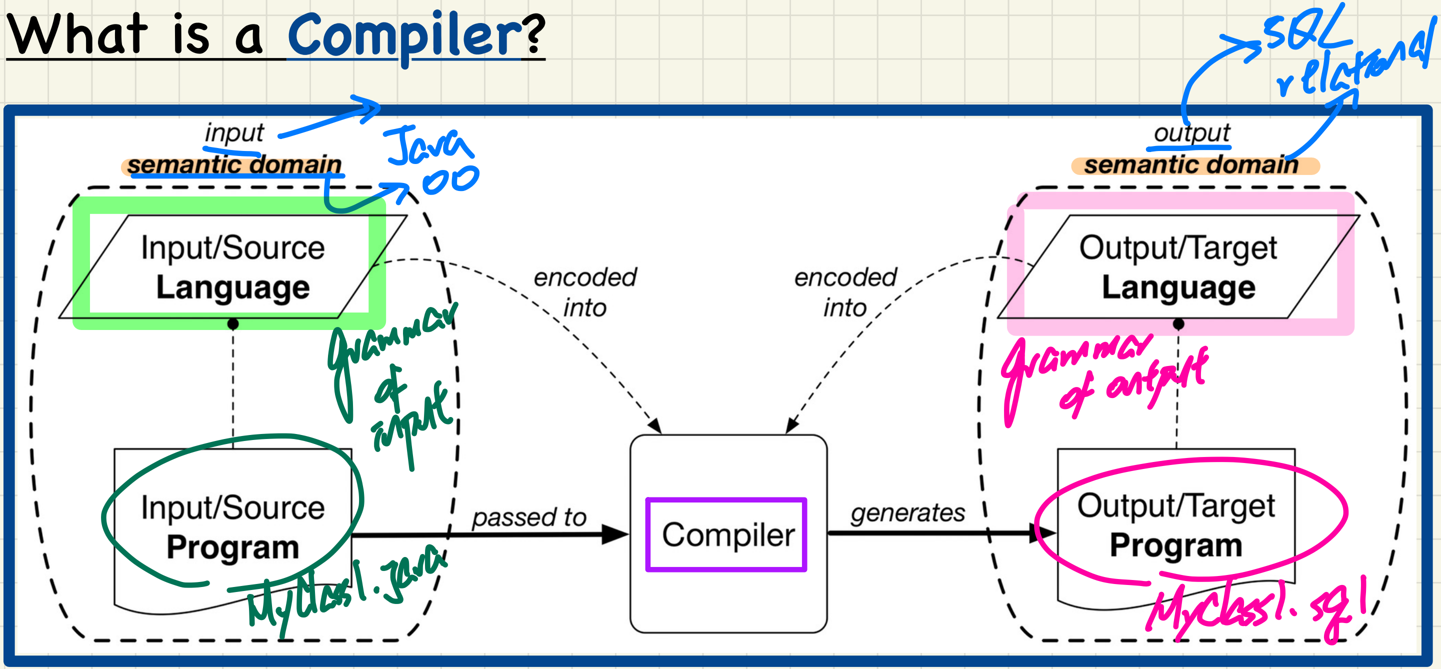
Instructor: Jackie Wang

Lecture 1 - Sep. 8

Syllabus & Overview of Compilation

Stages of a Compiler:
Lexical, Syntactic, Semantic

What is a Compiler?



input semantic domain

Java
OO

Input/Source **Language**

grammar of input

Input/Source **Program**

MyClass1.java

encoded into

passed to

Compiler

encoded into

generates

output semantic domain

SQL
relational

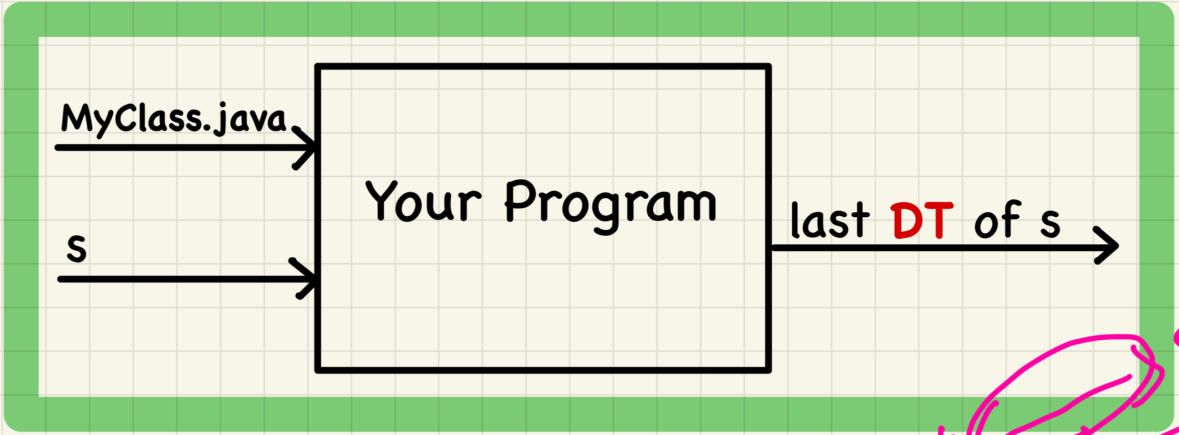
Output/Target **Language**

grammar of output

Output/Target **Program**

MyClass1.sql

An A+ Challenge: Inferring the DT of a Variable

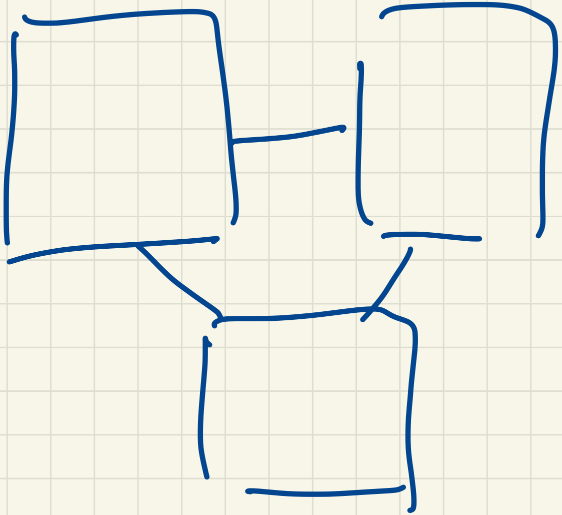


```
class MyClass {
  main (...)
    Student s = ...;
    ...
    s = new ResidentStudent(...);
  }
}
```

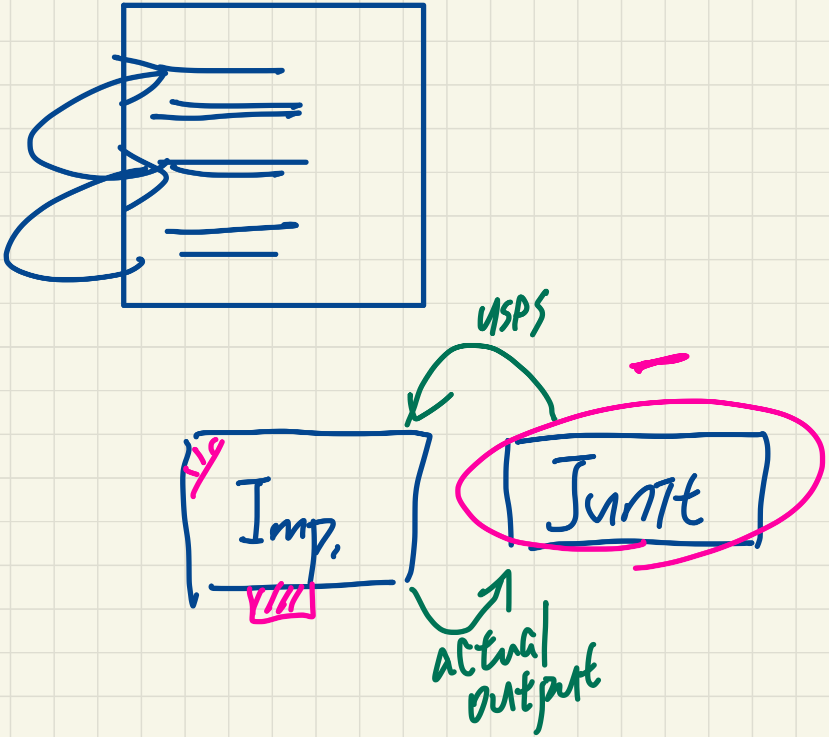
while { }

best: simulate the prog. and see/check the last DT

Modularity



Regression



Lecture 2 - Sep. 13

Overview of Compilation

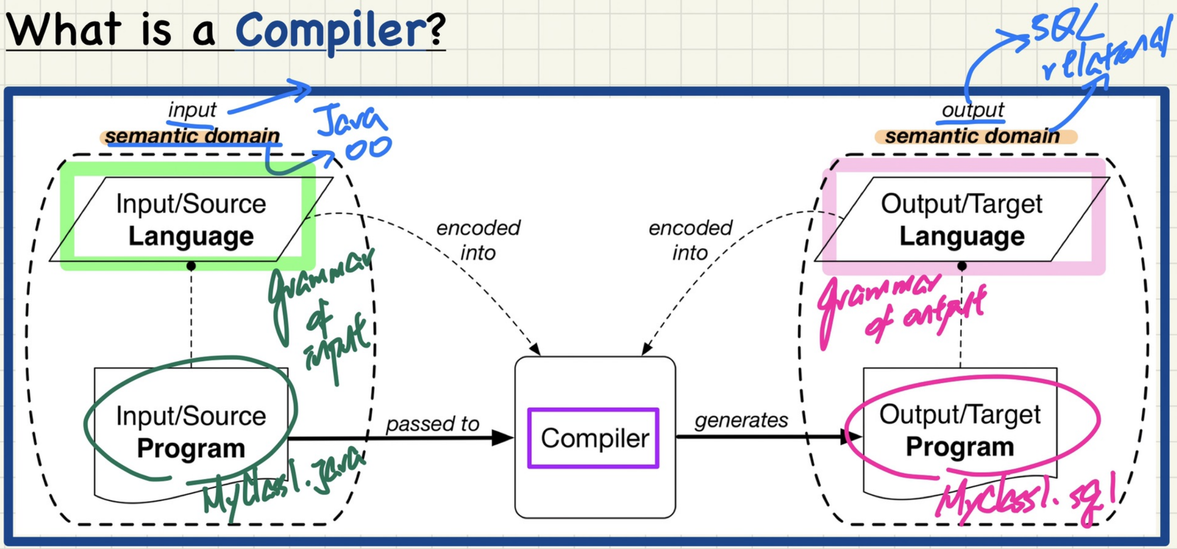
Components of a Compiler:

Frontend, Optimizer, Backend

Introducing Scanner

- Survey on Programming Test Time
- Office Hours

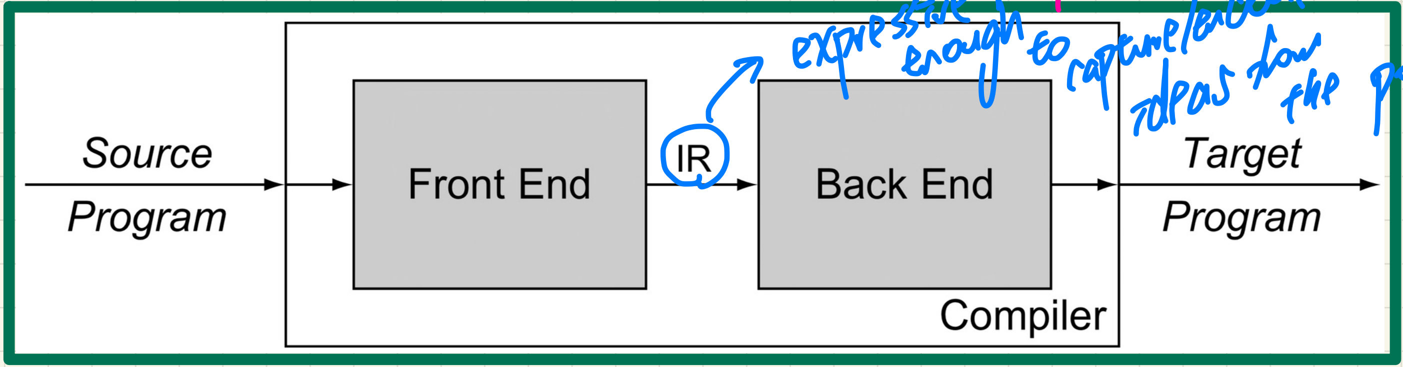
What is a Compiler?



Compiler: Typical Infrastructure (1)

Java C#

proced. OO



expressive enough to capture/encode ideas from the possible input

Concrete Syntax vs. Abstract Syntax

Java syntax

parse tree ✗

class A {
  int i;
}

→ class
name A
attributes
int i

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

- Java-to-C
- C#-to-C
- Java-to-Python
- C#-to-Python

IR1: Java-to-machine

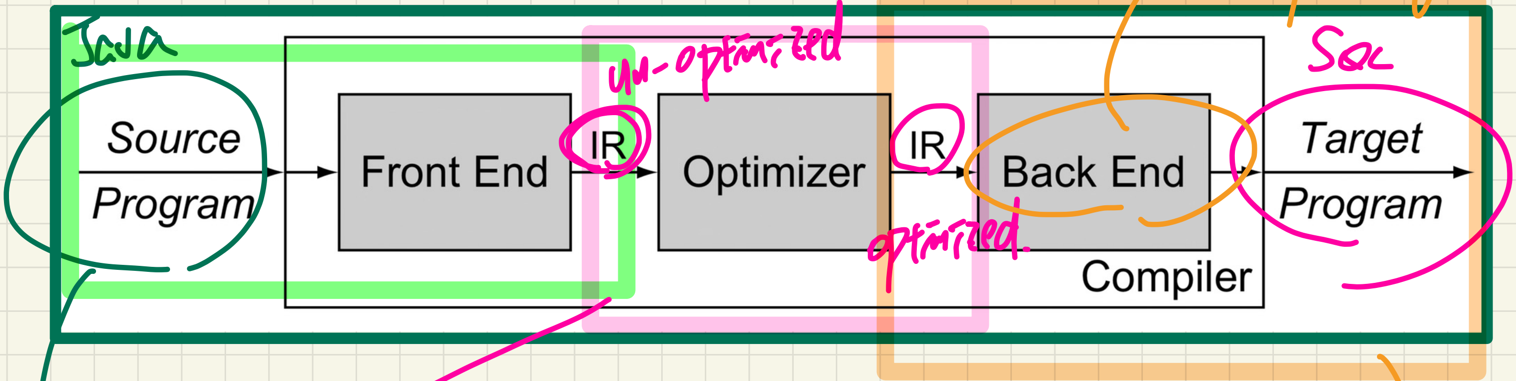
IR2: machine-to-Java

Java → IR

IR

Compiler: Typical Infrastructure (2)

pretty printing



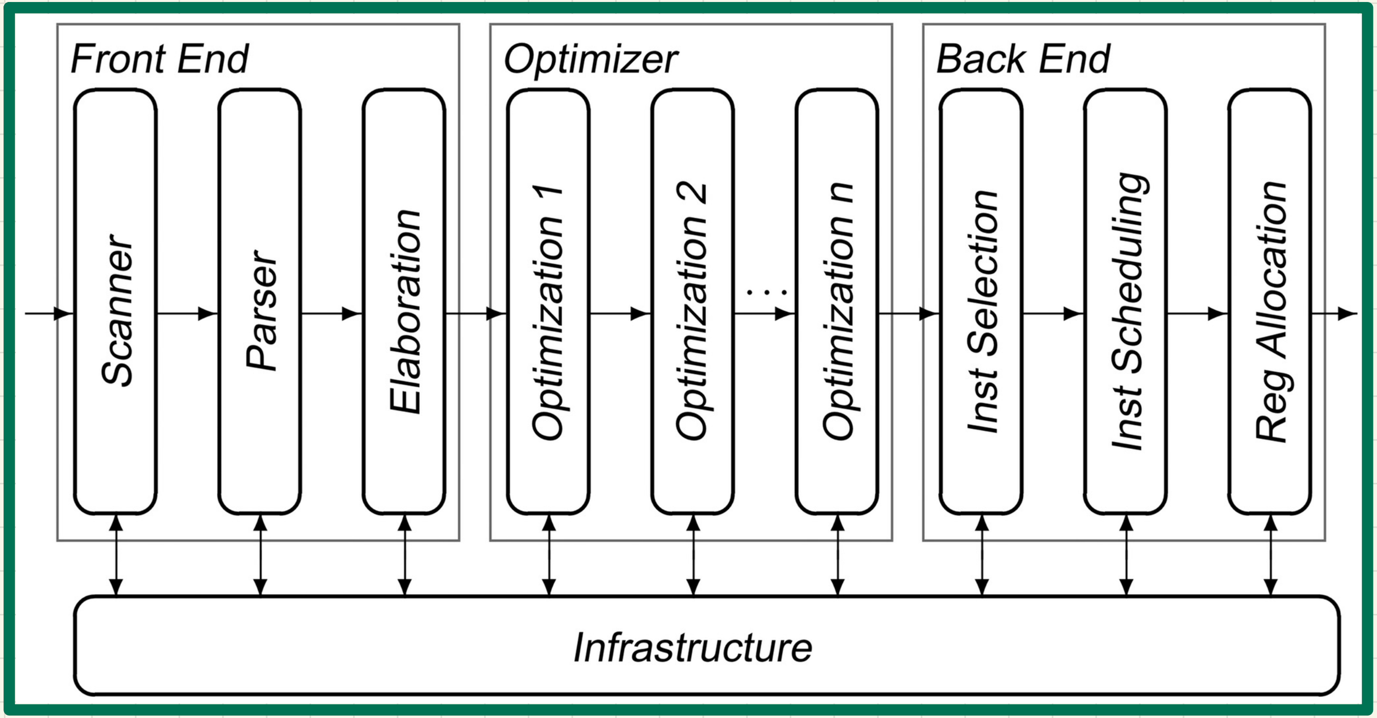
Q. What does the behaviour of the target program depend upon?

1. Input accurately encoded in IR

2. un-optimized IR accurately encoded in optimized IR

3. optimized IR accurately encoded in output

Example Compiler 1: Infrastructure



delimeter

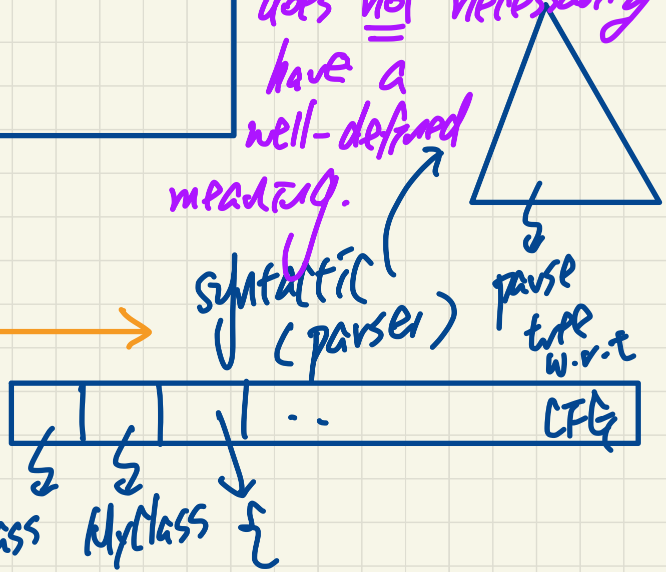
```
class MyClass {
  --- main() {
        println("Hello World");
  }
}
```

- A parse tree means the input is syntactically correct.
- A parse tree does not necessarily have a well-defined meaning.

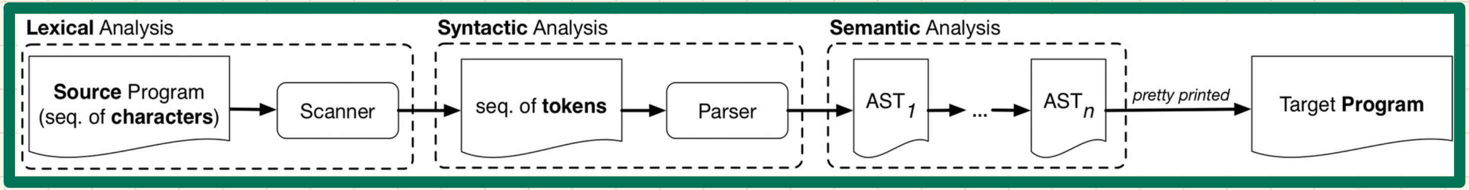
lexical (scanner)
↓
output: token

"class"
↳ token
  ↳ keywords
  ↳ identifier
  ⋮

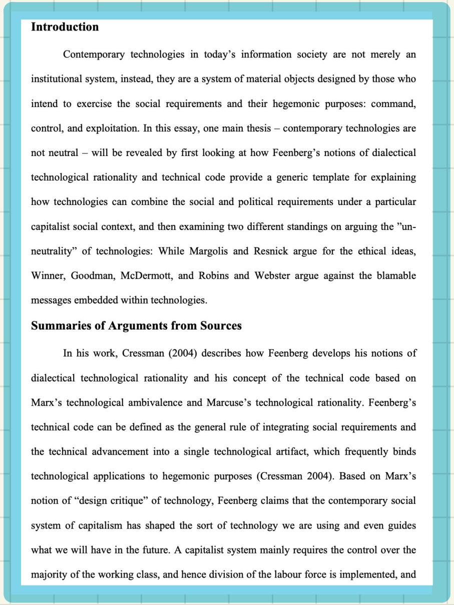
→ syntactic (parser)



Compiler Infrastructure: Scanner, Parser, Optimizer



Analogy: Compare Compilation to Essay Writing



Introduction

Contemporary technologies in today's information society are not merely an institutional system, instead, they are a system of material objects designed by those who intend to exercise the social requirements and their hegemonic purposes: command, control, and exploitation. In this essay, one main thesis – contemporary technologies are not neutral – will be revealed by first looking at how Feenberg's notions of dialectical technological rationality and technical code provide a generic template for explaining how technologies can combine the social and political requirements under a particular capitalist social context, and then examining two different standings on arguing the "un-neutrality" of technologies: While Margolis and Resnick argue for the ethical ideas, Winner, Goodman, McDermott, and Robins and Webster argue against the blamable messages embedded within technologies.

Summaries of Arguments from Sources

In his work, Cressman (2004) describes how Feenberg develops his notions of dialectical technological rationality and his concept of the technical code based on Marx's technological ambivalence and Marcuse's technological rationality. Feenberg's technical code can be defined as the general rule of integrating social requirements and the technical advancement into a single technological artifact, which frequently binds technological applications to hegemonic purposes (Cressman 2004). Based on Marx's notion of "design critique" of technology, Feenberg claims that the contemporary social system of capitalism has shaped the sort of technology we are using and even guides what we will have in the future. A capitalist system mainly requires the control over the majority of the working class, and hence division of the labour force is implemented, and

- words
- sentences
- meaning

lexical (spellings)

syntactic (grammar)

I tennis.

fails parser.

while-Loop: Context-Free Grammar (CFG)

$123 ? a234

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

Input: ① while true { print(...); }

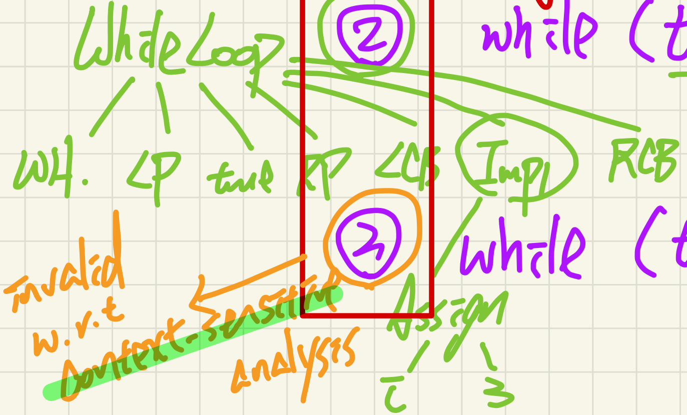
⇒ parse error (no parse tree)

valid PTs w.r.t context-free analysis ?

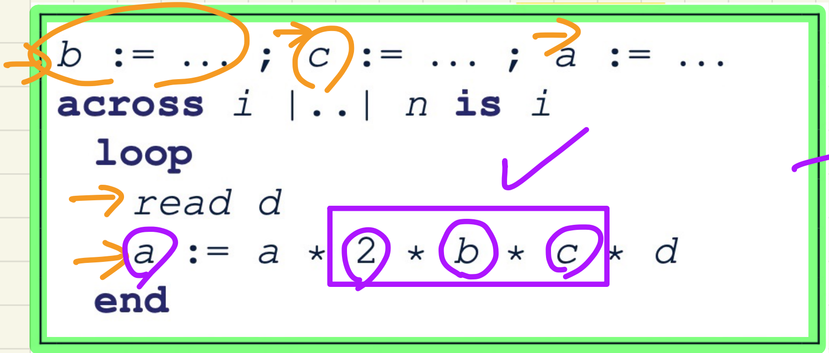
② while (true) { int i = 3; }

③ while (true) { int i = 3; int i = 4; }

invalid w.r.t context-sensitive analysis

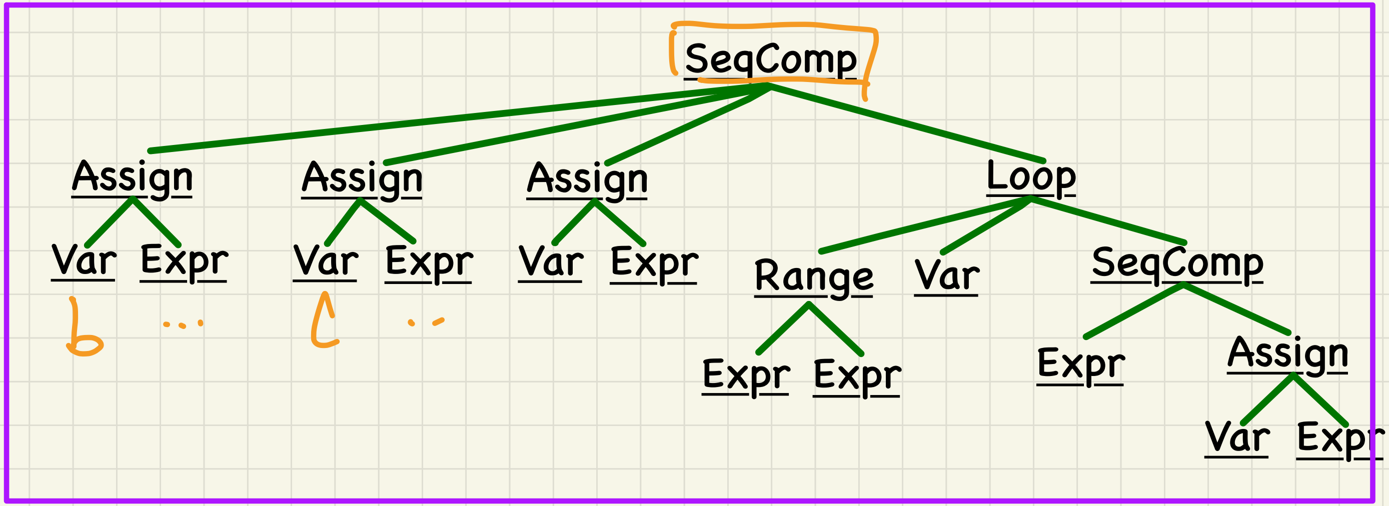


Compiler Infrastructure: AST-to-AST Optimizer (1)

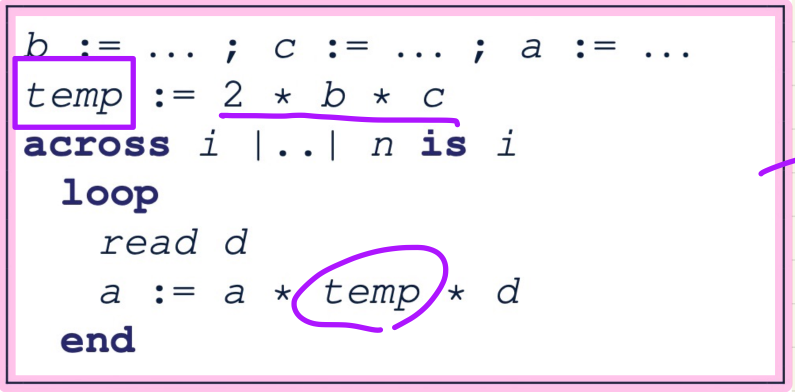


```
b := ... ; c := ... ; a := ...
across i |..| n is i
  loop
    read d
    a := a * 2 * b * c * d
  end
```

AST of input program:



Compiler Infrastructure: AST-to-AST Optimizer (2)



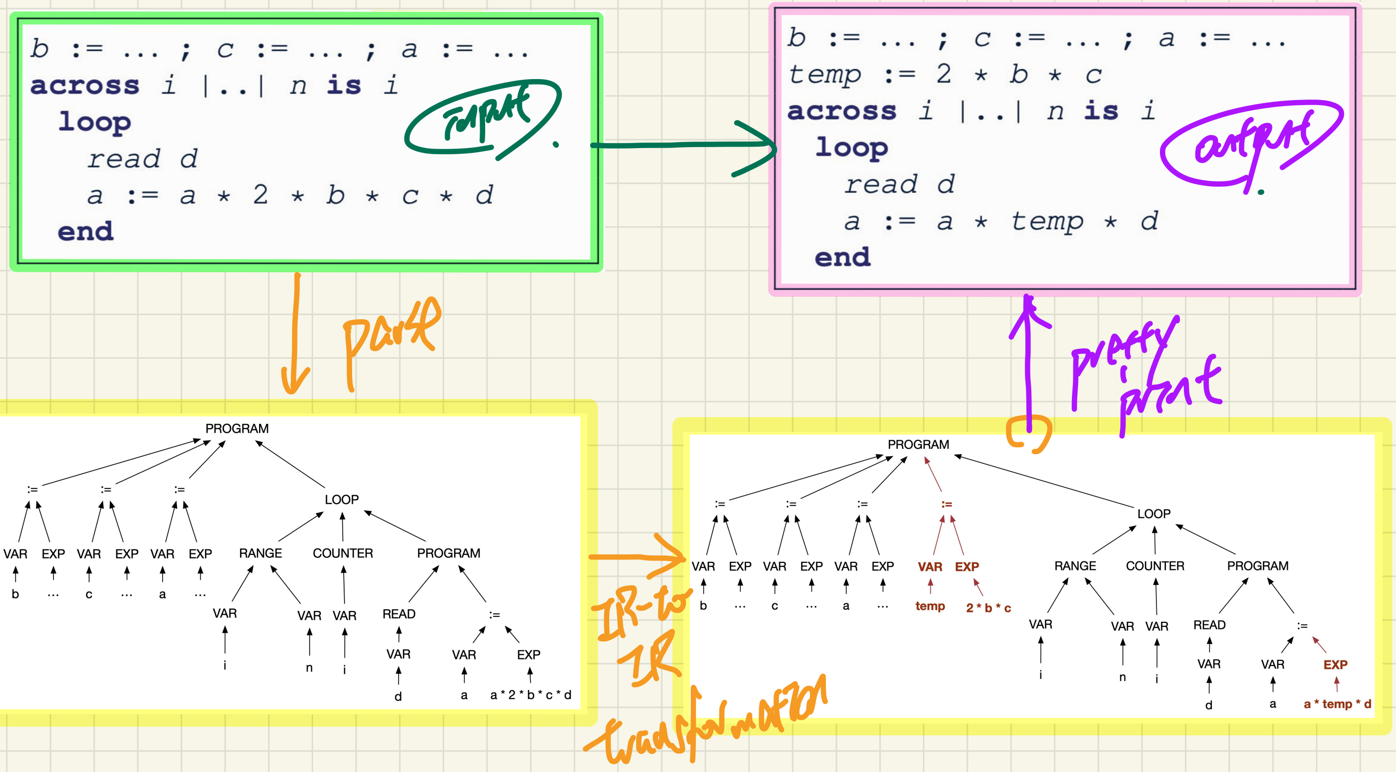
```
b := ... ; c := ... ; a := ...
temp := 2 * b * c
across i |..| n is i
  loop
    read d
    a := a * temp * d
  end
```

optimized version

AST of output program:

Compiler Infrastructure: AST-to-AST Optimizer (3)

Q. How should the various artifacts be connected?



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

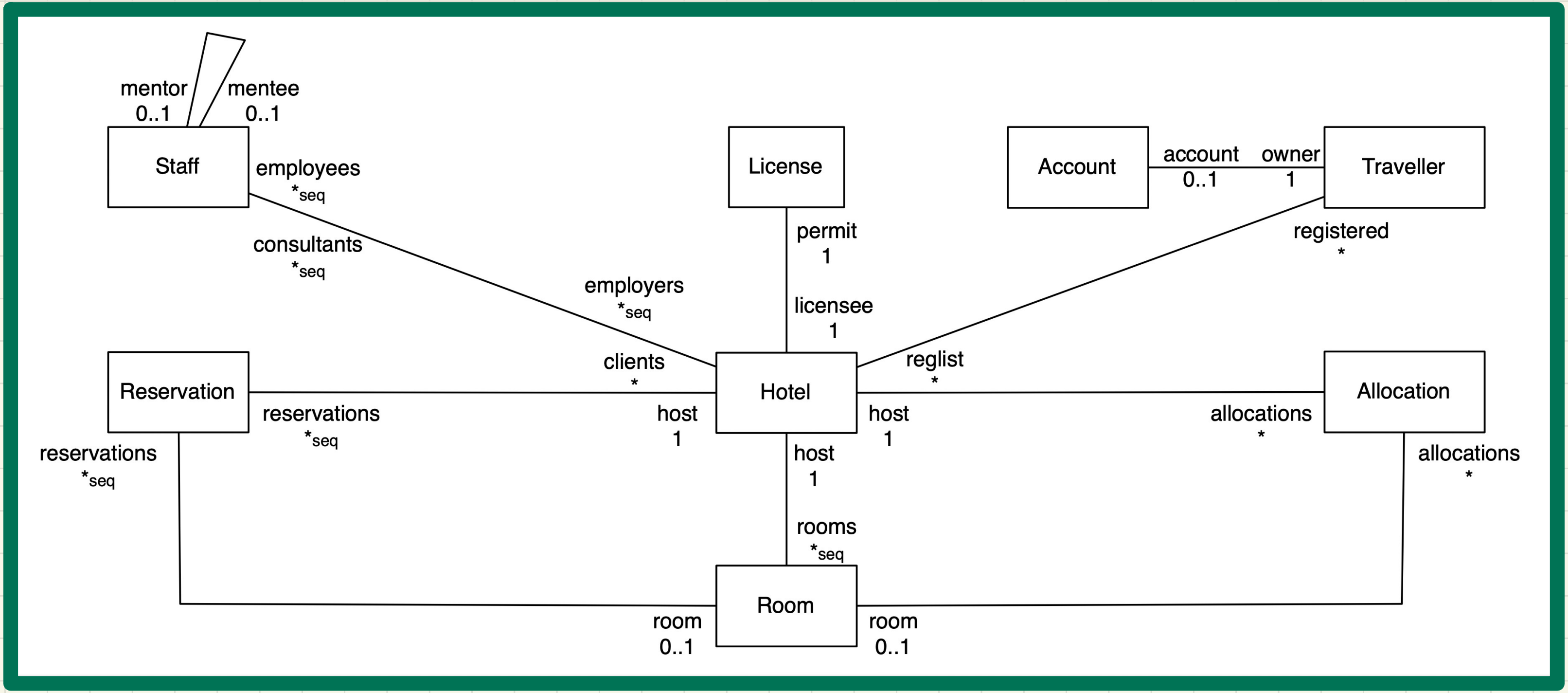
```
b := ... ; c := ... ; a := ...
temp := 2 * b * c
across i |..| n is i
  loop
    read d
    a := a * temp * d
  end
```

Lecture 3 - Sep. 15

Overview of Compilation

Example Compiler: Object-to-Relational Introducing Scanner

Example Compiler 2: Data Model



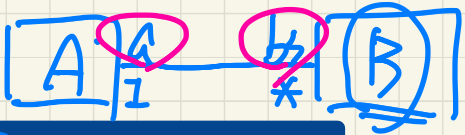
Example Compiler 2: Mapping Data

Attribute-to-Table Mapping

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

this

Example Transformation



```
class A {
  attributes
    s: string
    bs: set(B . a) [*] }
```

```
class B {
  attributes
    is: set(int)
    a: A . bs }
```

A

oid	s

B

oid	

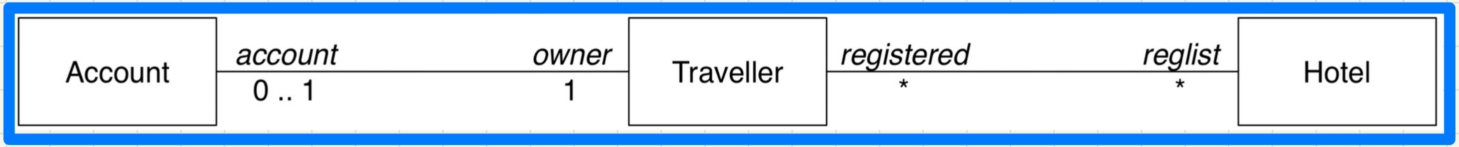
A_bs_B_a

oid	a	bs

is

oid	is

Example Compiler 2: Source Program



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

LCB

RCB

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

Scanner
- delimiter
- keywords

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

INPUT

⇒

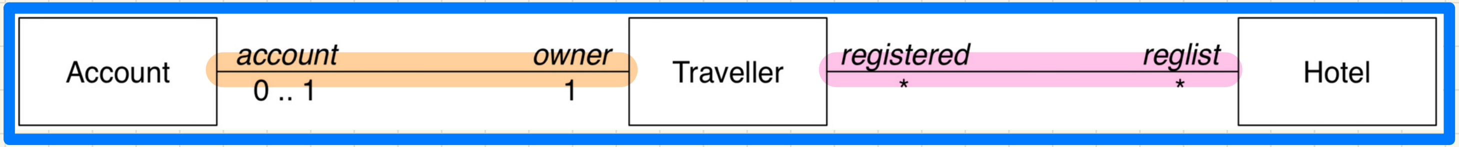
∉

∧

→ CFG for parser

Method ::=
Id LCB
Exp
IMP
Exp

Example Compiler 2: Target Program



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

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'));
```

MySQL

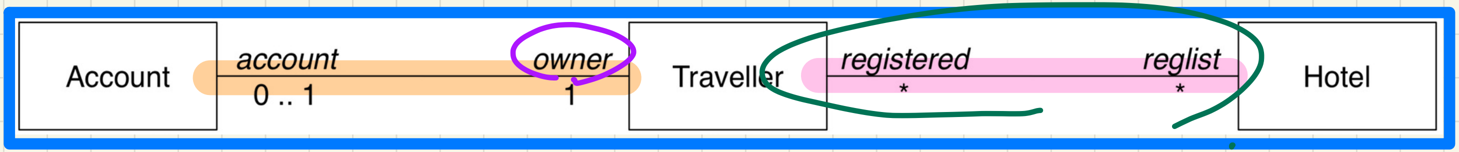
OO

```
m(..){
   :
   ;
   this.a.b.c
}
```

```
CREATE PROCEDURE 'Hotel_register'(IN 'this?' INTEGER, IN 't?' INTEGER)
  BEGIN
    ...
  END
```

Stored Procedures

Example Compiler 2: Path Transformation



Object Path

```
this.owner.reglist
```

oid::=4

oid::=3

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

this == 1

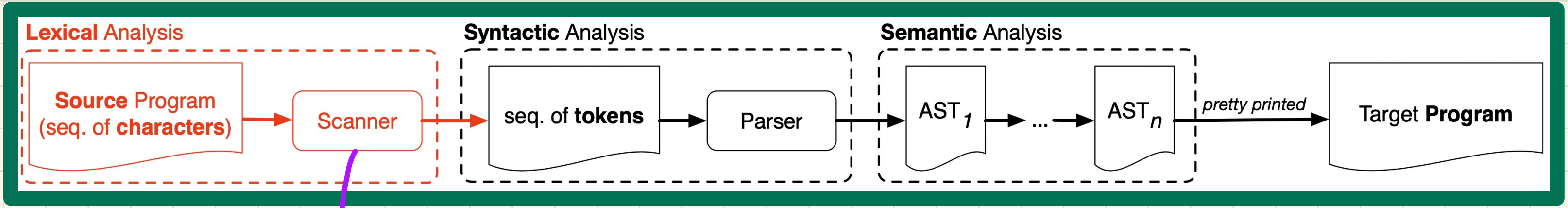
Traveller	
oid	name
2	alan
3	mark

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

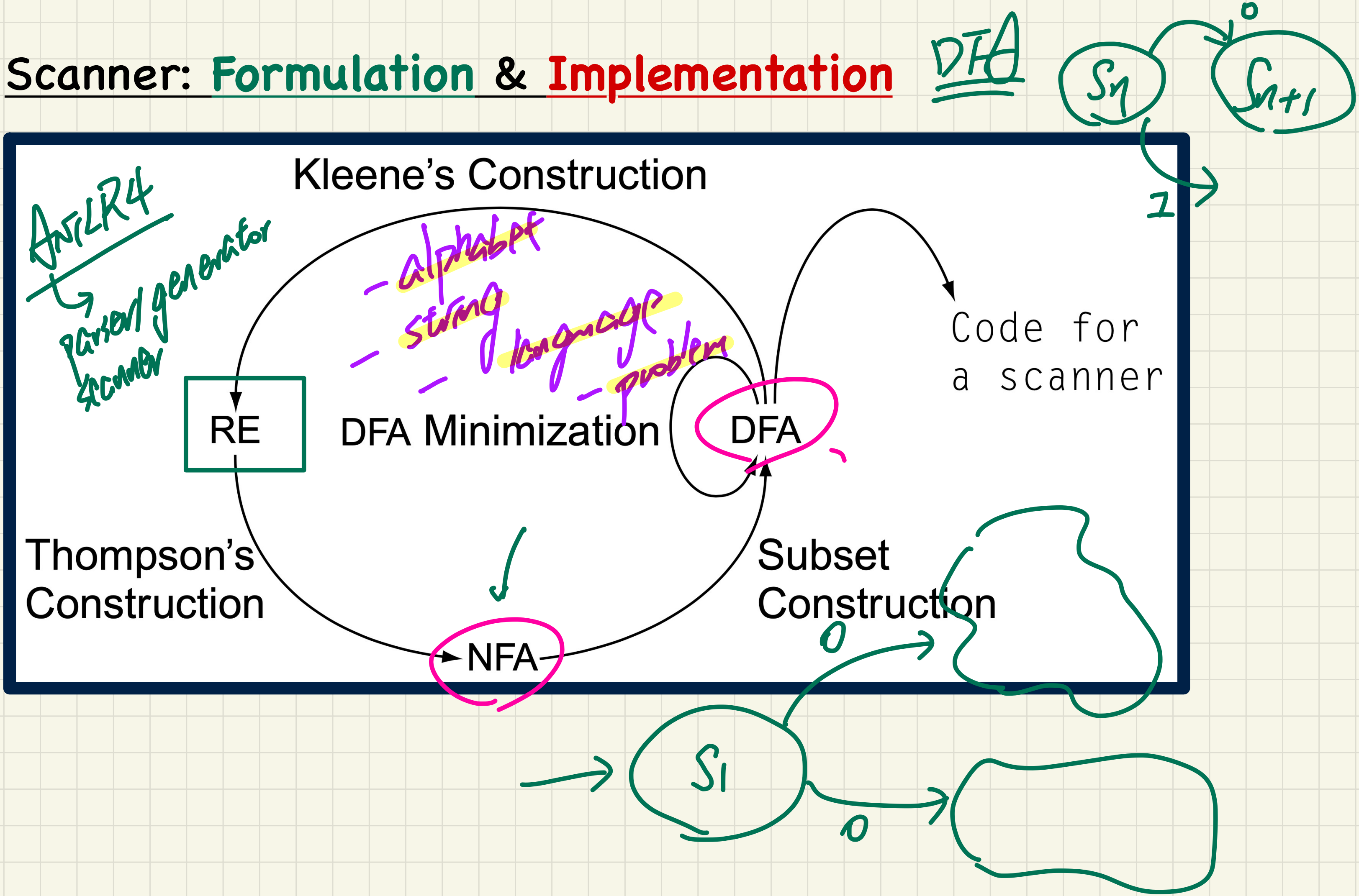
Scanner in Context



may also report error if there's any invalid char. seq.

Scanner: Formulation & Implementation

DFA



Set Comprehension

ε epsilon

↳ empty string.

$$\{ \underbrace{\qquad\qquad}_{\text{expression}} \mid \underbrace{\qquad\qquad\qquad\qquad}_{\text{predicates}} \}$$

↳ $\land$, $\lor$, $\neg$, $\Rightarrow$

↳ $\forall$, $\exists$

$$\Sigma_{dec} = \{ d \mid 0 \le d \le 9 \}$$

$\{0, 1\}$

$\boxed{01010} \in \Sigma_{bin}$

string

alphabet

$\mathbb{N} = \{0, 1, \ldots \infty\}$

natural #

$p \wedge \text{true} \equiv p$

$p \vee \text{false} \equiv p$

op	identity
$+$	0
$*$	1
concat	ε
$\wedge$	true
$\vee$	false

? $\Sigma^k = \{ xy \mid x \in \Sigma^1 \wedge y \in \Sigma^{k-1} \}$

$$\Sigma^k = \{ \underline{\underline{|w|}} \mid 0 \le |w| \le k \wedge \underset{=}{?} \}$$

#

not right ∵ the resulting set is a set of #'s

$$\Sigma^k = \{ \Sigma x \mid x \in \Sigma^{k-1} \}$$

not right ∵ concatenation only applies to two strings

w is a string of length k over Σ

$$\equiv \quad w = c_0 c_1 c_2 \cdots c_{k-1} \wedge (\forall i \cdot 0 \le i \le k-1 \Rightarrow c_i \in \Sigma).$$

$$\bigwedge_{0 \le i < k} c_i \in \Sigma$$

$$\Sigma^k = \{ w \mid \overset{\boxed{w \text{ is a string over } \Sigma}}{\vee} |w| = k \}$$

Lecture 4 - Sep. 20

Lexical Analysis

Strings, Languages
Regular Expressions

Announcements

- Assignment 1 Released
    + Required slides already made available
    + In-class discussion will catch up this or next week
- Programming Test date semi-confirmed:
    + 2:00pm to 3:20pm on Saturday, October 29
    + Venue to be confirmed (LAS)
- Quiz 1 next Tuesday

Is there any reason I need to wait to go through the ANTLR4 tutorial series on YouTube over reading week? Will I need the lecture right before to understand it?

- RE
- CFG
- OOP and Composite & visitor design patterns

Formulating Strings

✓ **Set of Strings of Length k**

alphabet → Σ^k language → L^k

$|\Sigma^k| = ?$ $|\Sigma|^k$

$\Sigma^0 =$ ① $\{\ \}$ ② ε ③ ✓ $\{\varepsilon\}$

$$\Sigma^k = \{ w \mid |w| = k \land w \in \Sigma^* \}$$

$\boxed{|\{0,1\}|^2}$ alphabet

all strings from $\{0,1\}$ with length 2

Set of Nonempty Strings

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots = \bigcup_{k>0} \Sigma^k$$

Set of Strings of All Possible Lengths

alphabet symbol

$$\Sigma = \{a, b\}$$

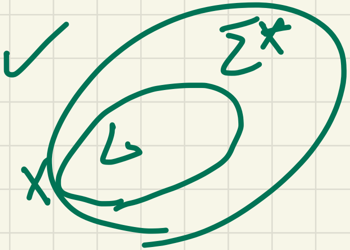
$$\Sigma^1 = \{a, b\}$$

string of length 1

$$L \subseteq \Sigma^*$$

① $w \in L \Rightarrow w \in \Sigma^*$ ✓

② $w \in \Sigma^* \Rightarrow w \in L$ ✗



$$\{xy \mid (x = 0 \wedge y = 1) \wedge$$

$|x|$

$$\{w_1 w_2 \mid w_1 \in \{0\}^* \wedge w_2 \in \{1\}^* \wedge |w_1| = |w_2|\}$$

union

$$\boxed{01^* + 10^*}$$

denotes some language (set of strings)

0^+

$$\{0x \mid x \in \{1\}^*\} \cup \{1x \mid x \in \{0\}^*\}$$

$$\{yx \mid (y=0 \wedge x \in \{1\}^*) \vee (y=1 \wedge x \in \{0\}^*)\}$$

$$\Sigma = \{0, 1\}$$

simplest "non-empty" RE: 0

1

Σ^k all strings with length k

L^k k concatenations of strings chosen from L.

Regular Language Operations

$L = \{ab, bc, ca\}$

$M = \{ba, cb\}$

$|L^i| = |L|^i$

1. *Union*

✓

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

$\{ab, bc, ca, ba, cb\}$

2. *Concatenation*

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

$\{wv \mid w \in L \wedge v \in M\}$

$\{abba, abcb, bcba, bccb, caba, cacb\}$

3. *Kleene Closure* (or *Kleene Star*)

$|L^*| =$ [redacted]

$L^0 = \{\varepsilon\}$

$L^1 = \{x \mid x \in L\} = L$

$L^2 = \{xy \mid x \in L \wedge y \in L\}$

$\vdots$

Cardinalities?

$$\underline{L} = \{0\}^*$$

$$L^* = \overset{\text{0 concatenations}}{\boxed{\underline{L}^0}} \cup \underline{L}^1 \cup \underline{L}^2 \cup \cdots$$

$$= \{\varepsilon\} \cup \{x \mid x \in \{0\}^*\}$$

$$\cup \; \{xy \mid x \in \{0\}^* \wedge y \in \{0\}^*\}$$

$$\cup$$

$$\vdots$$

Constructions of REs

Recursive Case: Given that E and F are regular expressions:

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

$$L(\underline{E + F}) = \text{[redacted]}$$

| (perl, J.S.)

$E \cup F$ ✗

$L(E) \cup L(F)$ ✓

- The concatenation EF is a regular expression.

$$L(\underline{EF}) = \text{[redacted]}$$

$L(E \cup F)$ ✗

$\boxed{L(E)L(F)}$

language concat-

given a written RE (e.g. E), L(E) denotes a language

- Kleene closure of E is a regular expression.

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

- A parenthesized E is a regular expression.

$$L(\,(E)\,) = L(E)$$

Base Case:

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

RE

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

$$L(\varnothing) = \varnothing$$

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

RE

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

Lecture 5 - Sep. 22

Lexical Analysis

RE: Exercises & Operator Precedence
DFA: Basics & Exercise

$$|\{a,b,\ldots,z\}^5| = |\{a,b,\ldots,z\}|^5$$

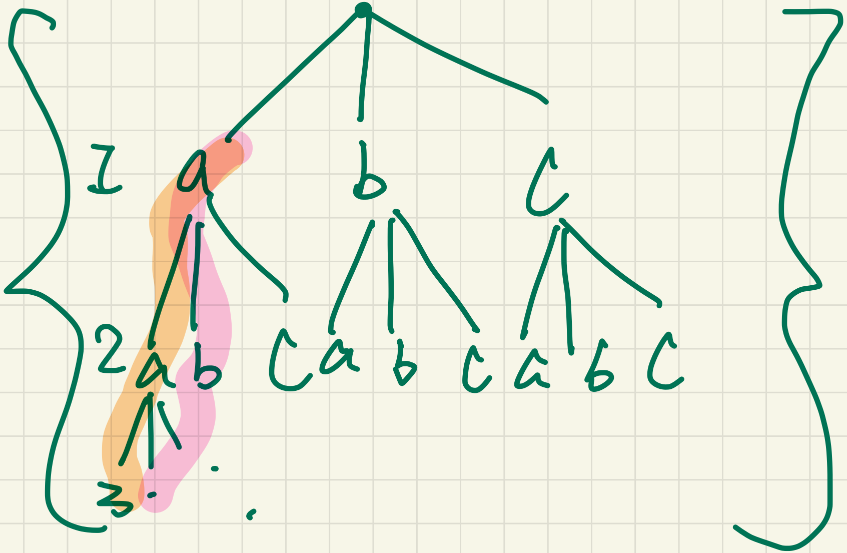
$$= 26^5$$

$$\boxed{\{\underline{a},\underline{b},c\}}^4 =$$

alphabet

↓

all strings of length 4



① $\Sigma_1 \subseteq \Sigma_2 \Rightarrow \Sigma_1^* \subseteq \Sigma_2^*$

Mathematical Induction

(Base Case) $\underset{\{\varepsilon\}}{\Sigma_1^0} \subseteq \underset{\{\varepsilon\}}{\Sigma_2^0}$ $\boxed{T}$.

(I.H.)

assume $\Sigma_1^n \subseteq \Sigma_2^n$ $(n>0)$.

(Prove) $\Sigma_1^{n+1} \subseteq \Sigma_2^{n+1}$

$c \in \Sigma_1$ ✓

to append to Σ_1^n,

it's guaranteed that $c \in \Sigma_2$

②

$$\Sigma_1 \subseteq \Sigma_2 \Rightarrow \Sigma_1^* \subseteq \underline{\underline{\Sigma_2^*}}$$

$$\Sigma_1^*$$

$$= \boxed{\Sigma_1^0} \cup \Sigma_1^1 \cup \Sigma_1^2 \cup \cdots$$

$$\subseteq \underline{\Sigma_2^0 \cup \Sigma_2^1 \cup \Sigma_2^2 \cup \cdots}$$

$$= \Sigma_2^*$$

$L1$ { start with 0s as many 1s as 0s }

$L2 = \{ xy \mid x \in \{0\}^* \wedge y \in \{1\}^+ \}$

$n \geq 0$ $m \geq 0$

$L3 = \{ 0^n 1^m \mid ✓ \; m \geq n \}$

0 1 0 ✗

$$\boxed{\begin{array}{l} 001 \in L2 \\ 001 \notin L1 \end{array}}$$

$L1 \subset L2$

↳ not a string s

$s \in L1$

$s \notin L2$

✓

$x \in \mathbb{N}$

$$L_2 = \{ a^x b^y c^z \mid x \geq 0 \wedge x \geq y + z \}$$

$y \geq 1$

$z \geq 1$

$L_1 =$ slide -

ab ∈

$a b^1 c^0 \in \frac{L_1}{\notin L_2}$

Exercise ✓

#s b's and c's at least as many as #a's

Σ^* is $\boxed{\text{a language over } \Sigma}$

Σ^*

R.E.

$$\boxed{\emptyset + L} = \emptyset \cup L = \underline{\underline{L}}.$$

$\Sigma^{\emptyset}$ vs L^{0}

$$\emptyset L = \{\underline{x}\underline{y} \mid x \in \emptyset \wedge y \in L\} = \emptyset.$$

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

$$= \{\varepsilon\} \cup \underbrace{\boxed{\{x \mid x \in \emptyset\}}}_{\emptyset} \cup \emptyset \cup \cdots$$

$$= \{\varepsilon\}$$

RE Specification: Exercise

Write a regular expression for the following language

$L_1 = \{ w \mid w \text{ has alternating } 0\text{'s and } 1\text{'s} \}$

0 ✗

1 ✗

0 1 ✓

1 0 ✓

0 1 0 ✓

1 0 1 ✓

$L_2 = (0(10)^+) + (1(01)^+)$

$01 \in L_1$

$01 \notin L_2$

RE: Operator Precedence

$\overset{L_1}{10^*}$ vs. $\overset{L_2}{(10)^*}$

$1(0^*)$

$1 \in L_1$
$1 \notin L_2$

$1010 \notin L_1$
$1010 \in L_2$

- Are RE_1 and RE_2 equivalent?
- A string in $L(RE_1)$ but not in $L(RE_2)$?
- A string in $L(RE_2)$ but not in $L(RE_1)$?

$01^* + 1$ vs. $0(1^* + 1)$

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

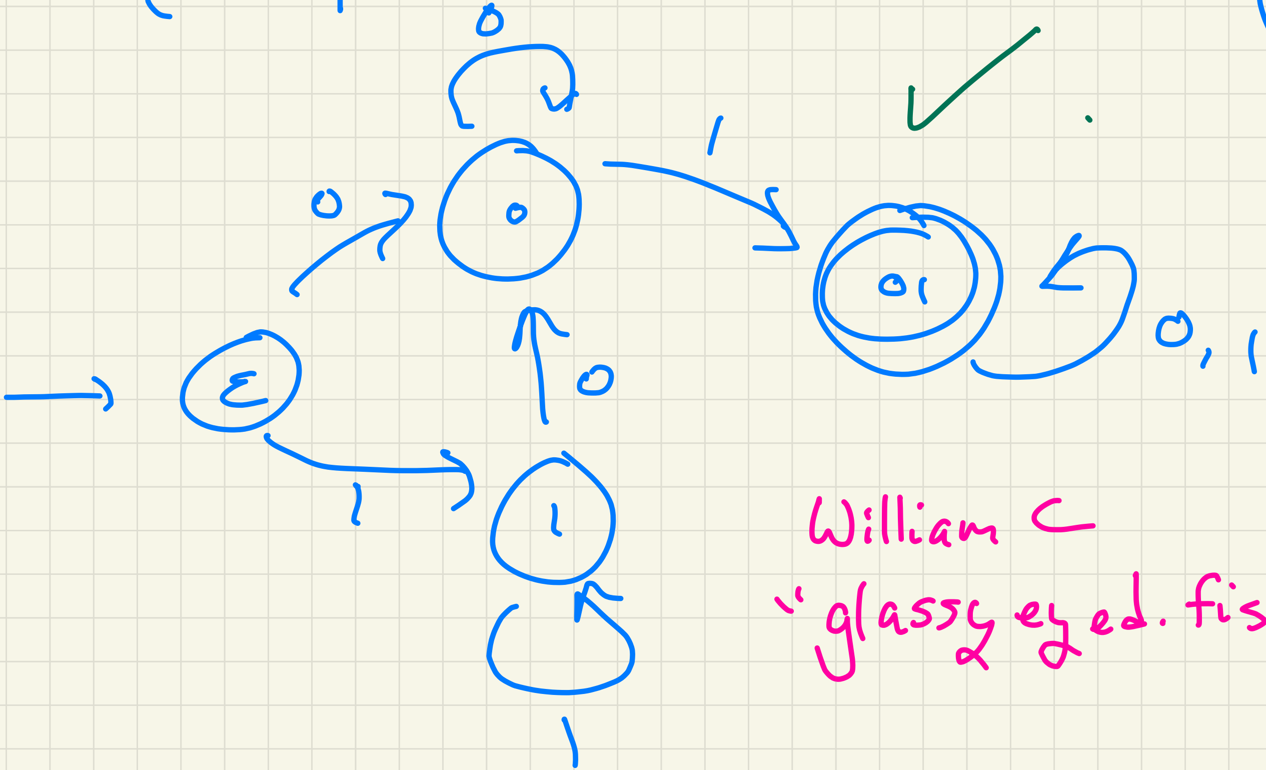
DFA: Exercise

Draw the transition diagram of a DFA which accepts/recognizes the following language:

{ w | w ≠ ε ∧ w has equal # of alternating 0's and 1's }

$\{ w \mid w \text{ contains } 01 \text{ as a substring} \}$

✓



William C

"glassyeyed.fish"

Lecture 6 - Sep. 27

Lexical Analysis

DFA: Formulations
NFA: Non-Deterministic Transitions

total function

$(\overset{\checkmark}{Q} \times \overset{\checkmark}{\Sigma}) \rightarrow Q$

for each combination of state and alphabets, there's always a corresponding state.

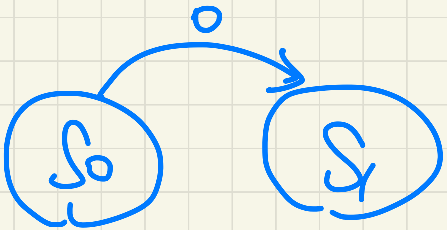
$(Q \times \Sigma) \nrightarrow Q$

partial function

add : $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

div : $\mathbb{Z} \times \mathbb{Z} \nrightarrow \mathbb{Z}$ e.g. div(3, 0) $\bot$

$$\delta = \{ ((s_0, 0), s_1), \\ \vdots \\ \}$$



DFA: Formulation (1)

Language of a DFA

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

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

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

e.g., 0101

$\boxed{1 \le i \le n} \wedge q_n \in F$

e.g. $0 \le i < n \wedge \delta(q_{\cancel{i}\, i-1}, a_{i\cancel{+1}}) = q_{i\cancel{+1}}$

$| a_1 | a_2 | a_3 |$

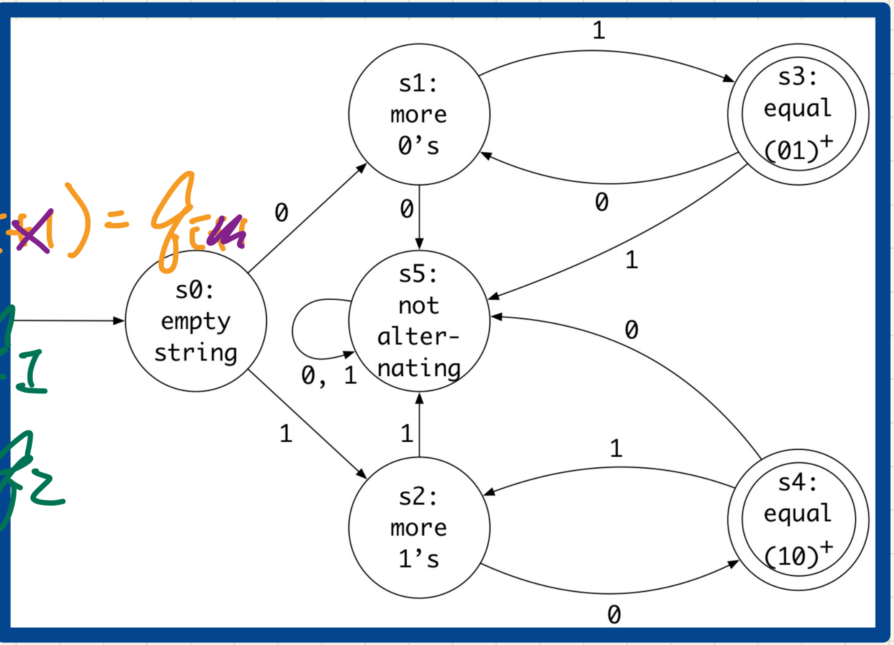
$q_0 \quad q_1 \quad q_2 \quad q_3$

$\delta(q_0, a_1) = q_1$

$\delta(q_1, a_2) = q_2$

$\vdots$

i



DFA: Formulation (2)

Language of a DFA

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

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

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

$$\hat{\delta}(q, xa) = \text{[redacted]}$$

last char

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

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

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

$$\delta(\underbrace{\hat{\delta}(q, x)}_{q'}, a)$$

e.g., 010

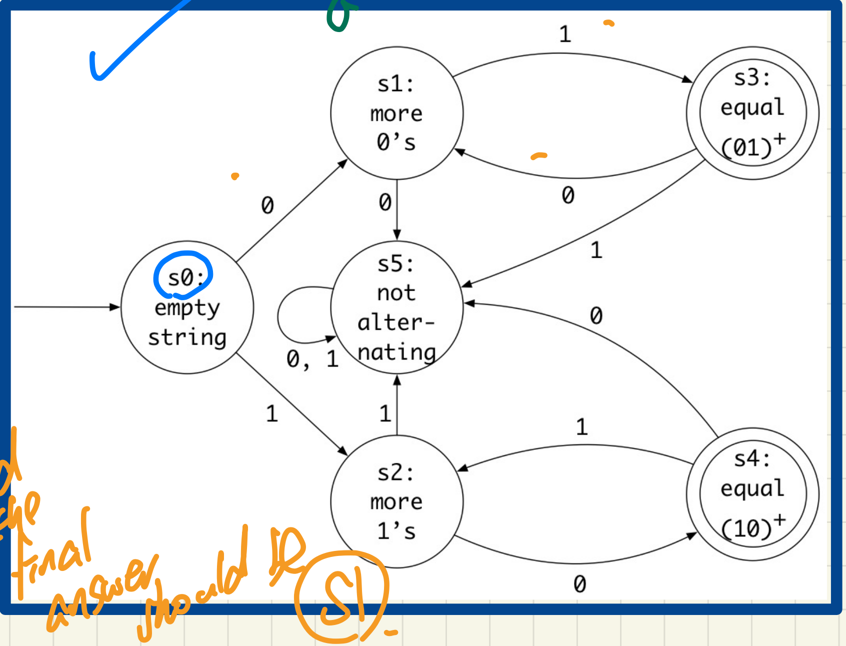
$$q \xrightarrow{x} q' \xrightarrow{a} \square$$

$$\hat{\delta}(S_0, 010)$$

$$= \delta(\hat{\delta}(S_0, 01), 0)$$

$$= \delta(\hat{\delta}(S_0, 0), 1)$$

Exercise: Finish unfolding this and the final answer should be S_1.



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

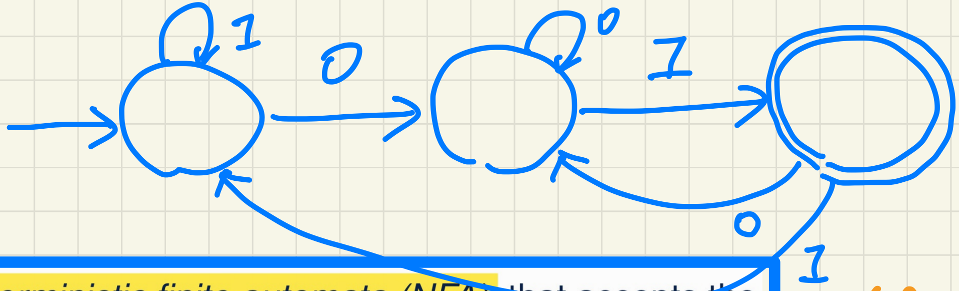
DFA vs. NFA

01111101

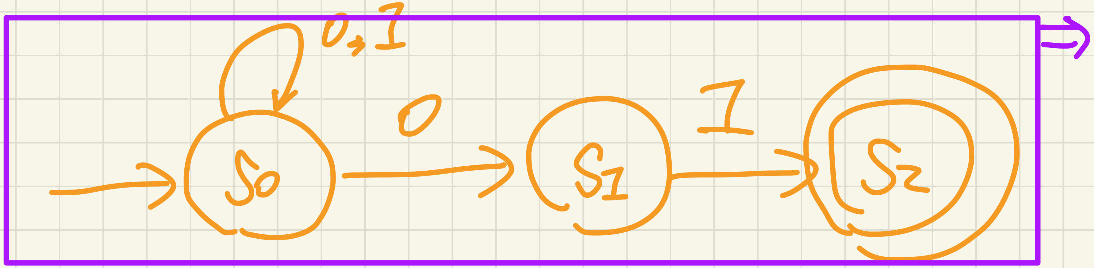
Problem: Design a DFA that accepts the following language:

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

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:



$(S_1, 0) \in \delta$?

⇒ not DFA

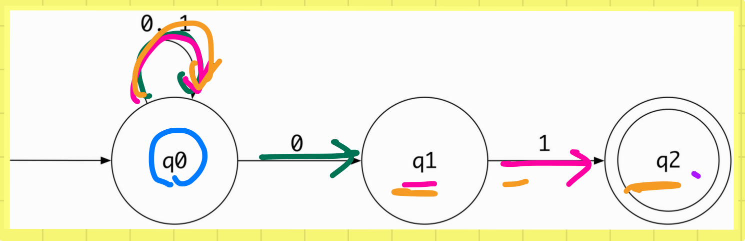
Lecture 7 - Sep. 29

Lexical Analysis

NFA: Tracing & Formulation
NFA to DFA Conversion
ε-NFA: Formulation and ε-Closure

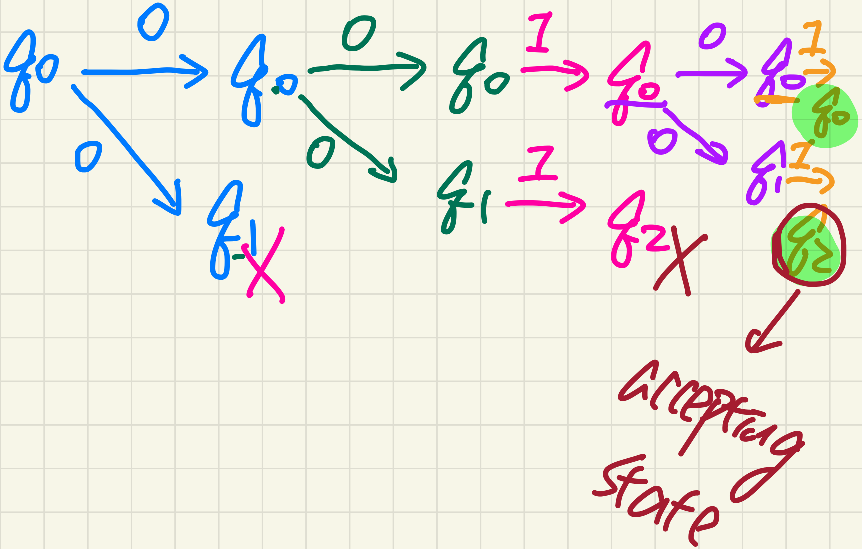
NFA Behaviour ≈ Alternative Universe

Obviously the **time continuum** has been disrupted, creating this new temporal event sequence resulting in this **alternate reality**. – Doc Brown



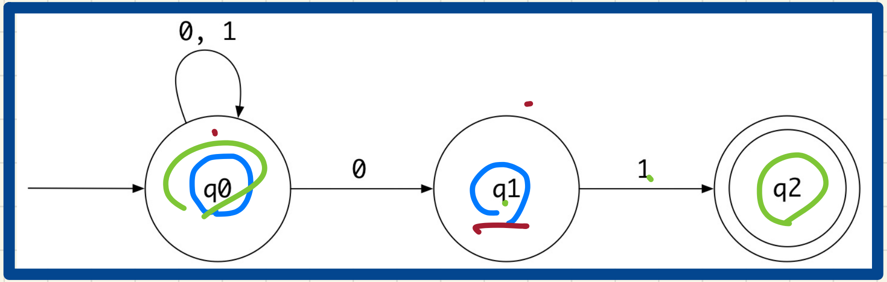
Trace: 00101

Try: 00111



NFA: Processing Strings

How an NFA determines if an input 00101 should be accepted:



Read 0: $\delta(q_0, 0) = \{\overset{\checkmark}{q_0}, \overset{\checkmark}{q_1}\}$

Read 0: $\delta(q_0, 0) \cup \delta(q_1, 0) = \overset{q_1}{\{q_0\}} \cup \{\} = \{q_0\}$

Read 0:

Read 0:

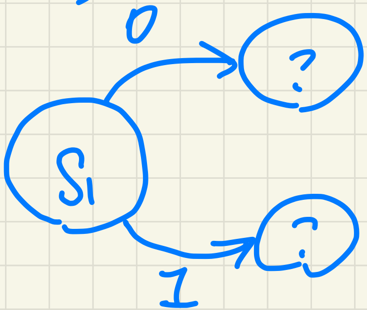
Read 0:

Exercise

$\Sigma = \{0, 1\}$

DFA

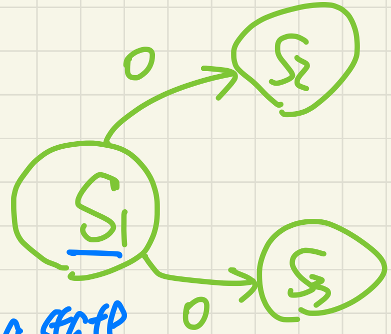
$$\delta : (Q, \Sigma) \rightarrow Q$$



NFA

$$\delta : (Q, \Sigma) \rightarrow \mathbb{P}(Q)$$

a set of states



$((S_1, 0), \{S_2, S_3\}) \in \delta$

not necessarily every (Q, Σ) has a defined resulting state

alt: $\delta : (Q, \Sigma) \nrightarrow Q$

$(S_1, 1)$ undefined $((S_1, 1), \phi) \in \delta$

NFA: Formulation

Language of a NFA

$q \xrightarrow{x} q' \xrightarrow{a} \bigcirc$

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

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

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

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

$$\hat{\delta}(q, \epsilon) = \{q\} \rightarrow \text{singleton set}$$

$$\hat{\delta}(q, \overset{\text{prefix}}{x}a) = \bigcup \{\delta(q', a) \mid q' \in \hat{\delta}(q, x)\}$$

resulting state after processing x

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

Given an input string 00101:

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

x

a

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

$\hat{\delta}(q_0, 00101)$

$= \delta(\underbrace{\hat{\delta}(q_0, 0010)}_{\text{set of states}}, \underbrace{1}_{})$

a set of states.

$\hat{\delta}(q_0, 0010\ 1)$

$q' \in \{q_0, q_1\}$

$= \boxed{\delta(q_0, 1)}_{\{q_0\}} \cup \boxed{\delta(q_1, 1)}^{\{q_2\}}$

Every DFA is an NFA.

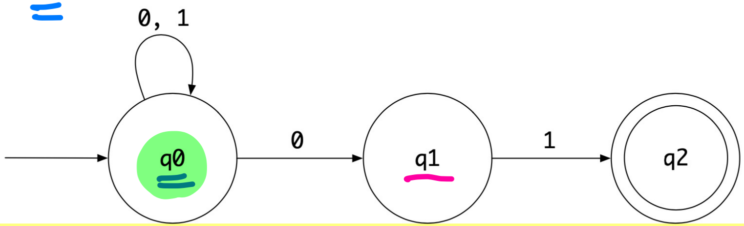
Not necessarily every NFA is a DFA.

has some missing transition

$\subset \Sigma$

NFA to DFA: Subset Construction (Lazy Evaluation)

Given an *NFA*: Q



Subset construction (with ***lazy evaluation***) produces a ***DFA*** with δ as:

subset state = each state in the DFA corresponds to a set of states in NFA $(\subseteq Q)$

state \ input	0	1
$\{q_0\}$	$\delta(q_0, 0) = \{q_0, q_1\}$	$\delta(q_0, 1) = \{q_0\}$ → discovered already.
$\{q_0, q_1\}$	$\delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_2\}$
$\{q_0, q_2\}$	(Exercise)	

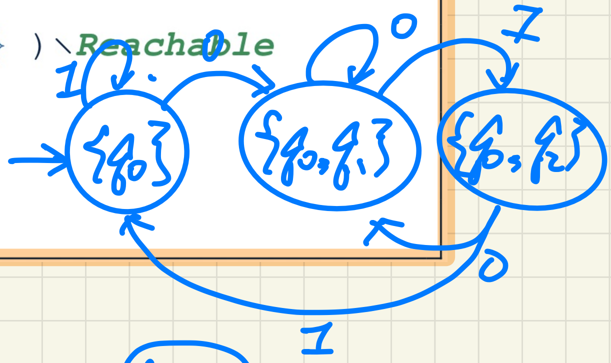
Subset Construction: Algorithmic Specification

Given an *NFA* $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$:

```
ALGORITHM: ReachableSubsetStates
  INPUT: $q_0 : Q_N$   ;   OUTPUT: $Reachable \subseteq \mathbb{P}(Q_N)$
PROCEDURE:
  $Reachable := \{ \{q_0\} \}$
  $ToDiscover := \{ \{q_0\} \}$
  while ( $ToDiscover \neq \emptyset$ ) {
    choose $S : \mathbb{P}(Q_N)$ such that $S \in ToDiscover$
    remove $S$ from $ToDiscover$
    $NotYetDiscovered :=$
        $( \{ \{\delta_N(s, 0) \mid s \in S\} \} \cup \{ \{\delta_N(s, 1) \mid s \in S\} \} ) \setminus Reachable$
    $Reachable := Reachable \cup NotYetDiscovered$
    $ToDiscover := ToDiscover \cup NotYetDiscovered$
  }
  return $Reachable$
```

to determine if the lazy eval. should continue.

state \ input	0	1
✓ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
✓ $\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
✓ $\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$



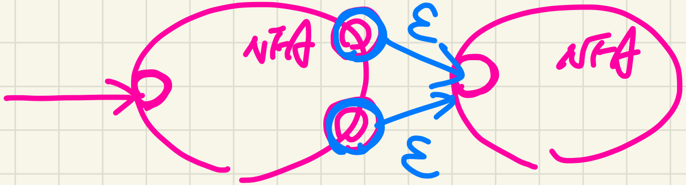
NFA: s_0, s_1, s_2 $\quad 2^{|Q|} = 8$

Worst case DFA: {} {s0} {s0,s1} {s1,s2} . . . {s0,s1,s2}

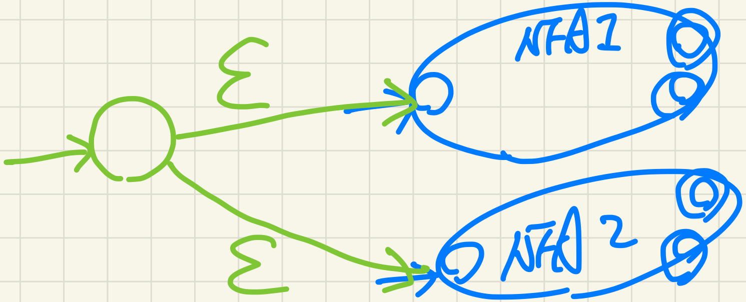
epsilon-NFA: Motivation

Draw NFA

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



$$\left\{ w : \{0,1\}^* \;\middle|\; \begin{array}{ll} 1 & w \texttt{ has alternating 0's and 1's} \\ \vee\ 2 & w \texttt{ has an odd \# 0's and an odd \# 1's} \end{array} \right\}$$



Lecture 8 - Oct. 4

Lexical Analysis

ε-NFA: ε-Closure & Conversion to DFA
From Regular Expressions to ε-NFA
Minimizing DFA

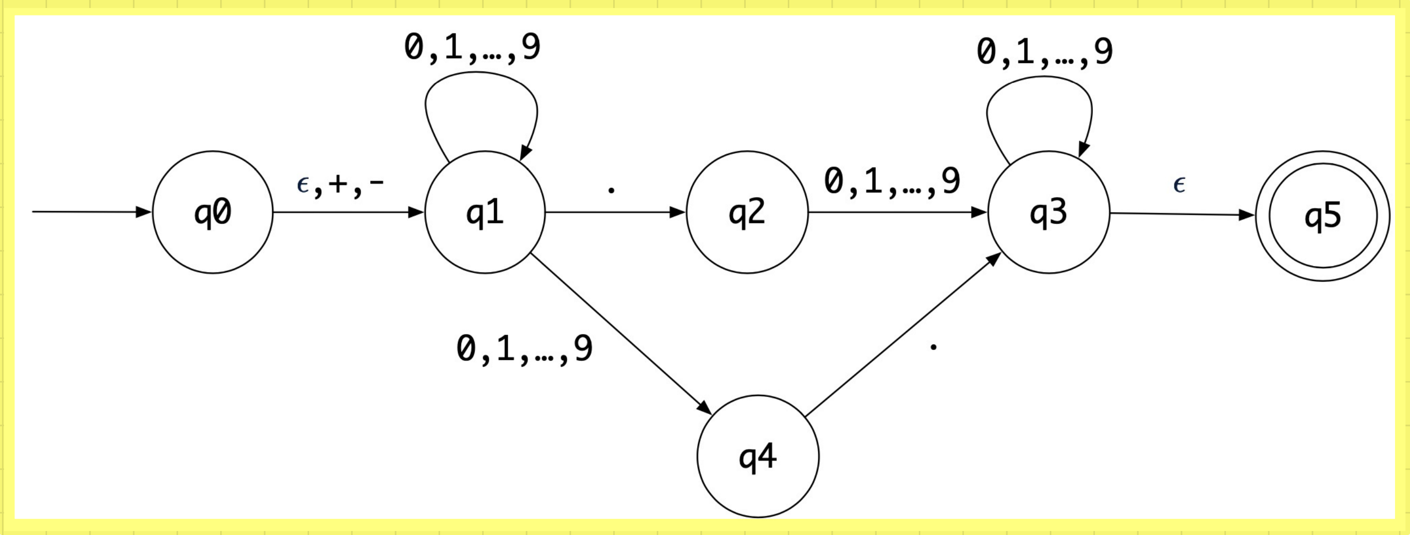
epsilon-NFA: Example

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

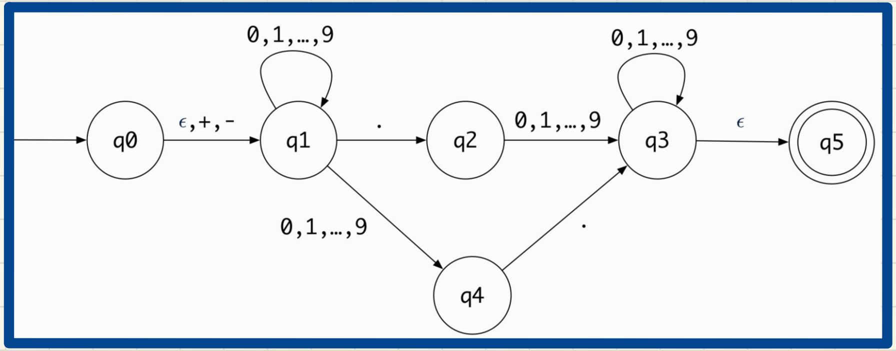
Is this a DFA? N.

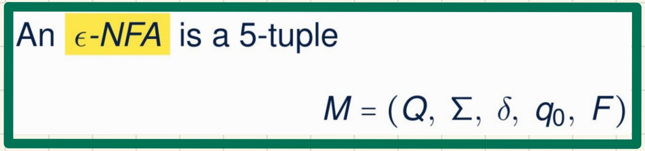
Is this an NFA? N.

Is this an ε-NFA? Y.



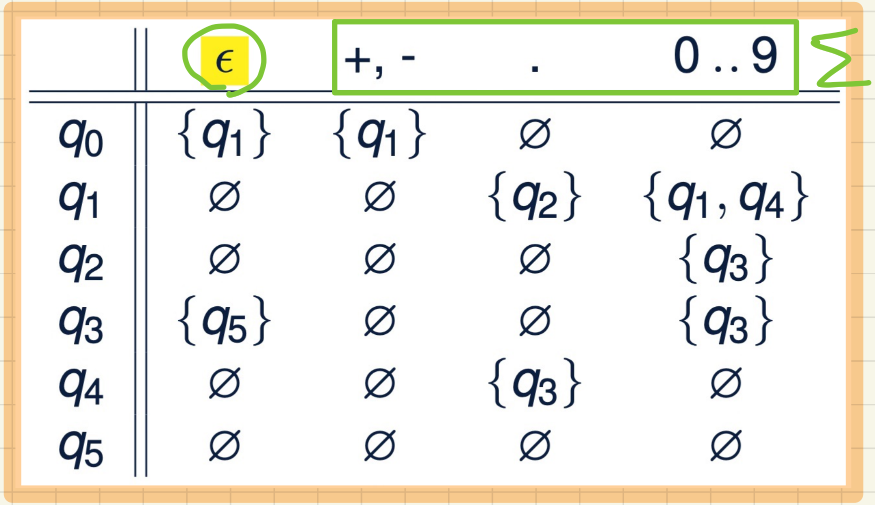
epsilon-NFA: Formulation (1)




An *ϵ-NFA* is a 5-tuple

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

Draw the transition table.



	ϵ	+, -	.	0 .. 9
q_0	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_4	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
q_5	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Σ

epsilon-NFA: Formulation (2)

An ϵ-NFA is a 5-tuple

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

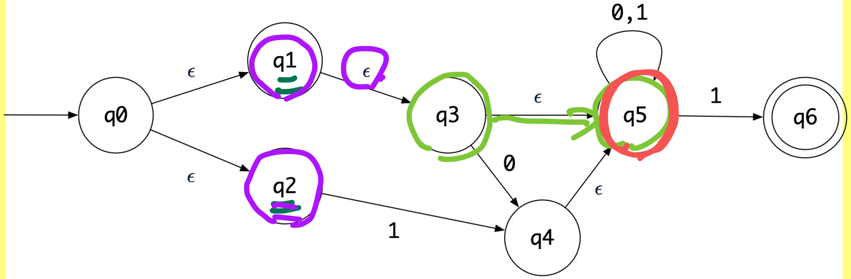
we define the *epsilon closure* (or *ϵ-closure*) as a function

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

For any state $q \in Q$

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

ECLOSE of all states reachable from p via ε.



Derive ECLOSE(q0).

$\text{ECLOSE}(q_0)$

$= \{q_0\} \cup \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_2)$

$\text{ECLOSE}(q_1) \rightarrow \{q_1\} \cup \boxed{\text{ECLOSE}(q_3)}$

$\text{ECLOSE}(q_2) \rightarrow \{q_2\}$

$\text{ECLOSE}(q_3) \rightarrow \{q_3\} \cup \text{ECLOSE}(q_5)$

$\text{ECLOSE}(q_5) \rightarrow \{q_5\}$

Answer:

$\{q_0, q_1, q_3, q_2, q_5\}$

epsilon-NFA: Formulation (3)

An ϵ-*NFA* is a 5-tuple

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

DFA: q

NFA: $\{q\}$

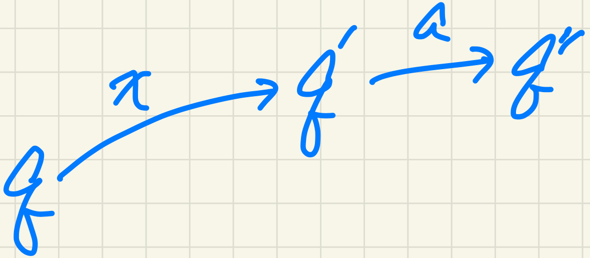
$$\hat{\delta} : (Q \times \Sigma^*) \to \mathbb{P}(Q)$$

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

$$\boxed{\hat{\delta}(q, \epsilon)} = \text{ECLOSE}(q)$$

$$\hat{\delta}(q, xa) = \bigcup\{\text{ECLOSE}(q'') \mid q'' \in \delta(q', a) \wedge q' \in \hat{\delta}(q, x)\}$$

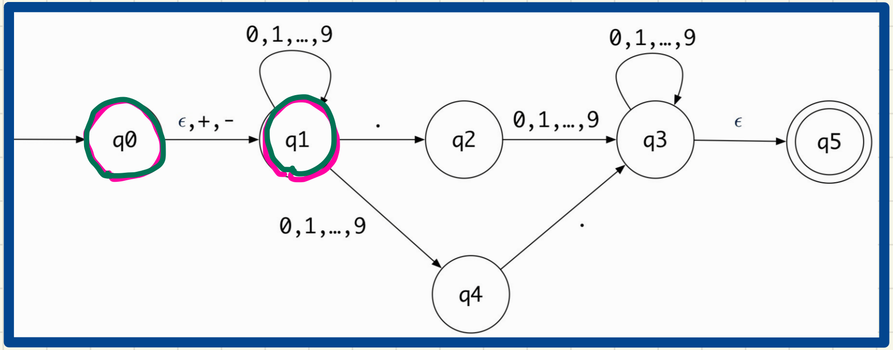
compare with $\hat{\delta}$ of NFA



Language of a epsilon-NFA

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

epsilon-NFA: Processing Strings



Exercises

① .6

② +23

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

$\hat{\delta}(q_0, \epsilon) =$ $\{q_0, q_1\}$

- **Read 5:** $\delta(q_0, 5) \cup \delta(q_1, 5) = \emptyset \cup \{q_1, q_4\} = \{q_1, q_4\}$

$\hat{\delta}(q_0, 5) =$ $\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$

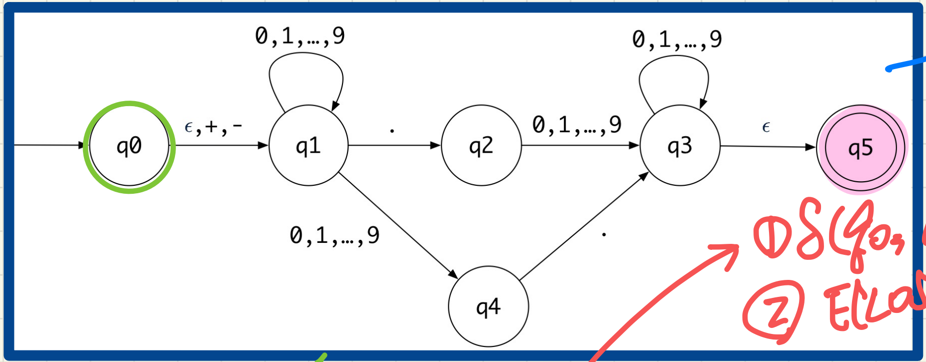
- **Read .:**

$\hat{\delta}(q_0, 5.) =$ Exercise

- **Read 6:**

$\hat{\delta}(q_0, 5.6) =$

epsilon-NFA to DFA: Extended Subset Construction

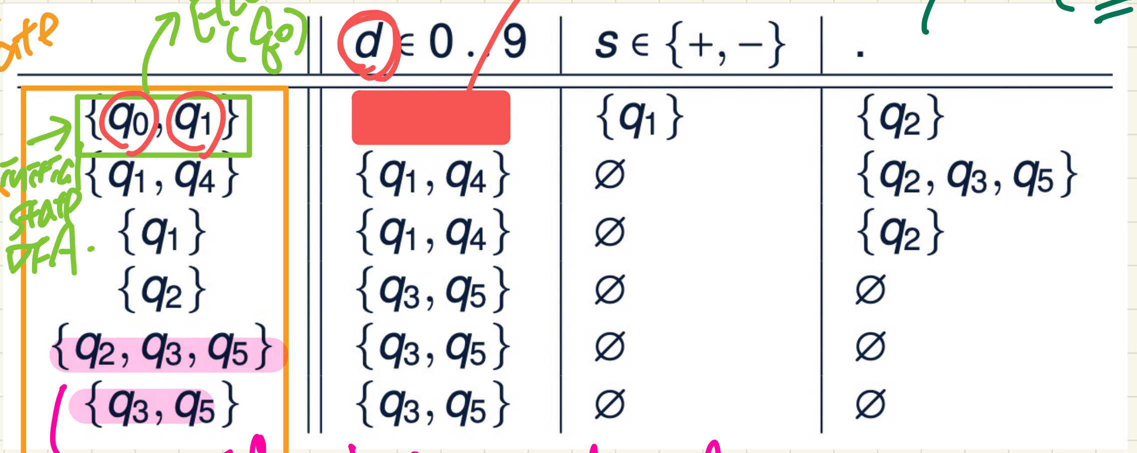


ε-NFA

① $\delta(q_0, d) \cup \delta(q_1, d) = \boxed{\ \ldots\ }$

② ECLOSE$(\ldots)$

δ of DFA. (no ε transition)



	$d \in 0..9$	$s \in \{+, -\}$	.
$\{q_0, q_1\}$	[redacted]	$\{q_1\}$	$\{q_2\}$
$\{q_1, q_4\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2, q_3, q_5\}$
$\{q_1\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2\}$
$\{q_2\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_2, q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$

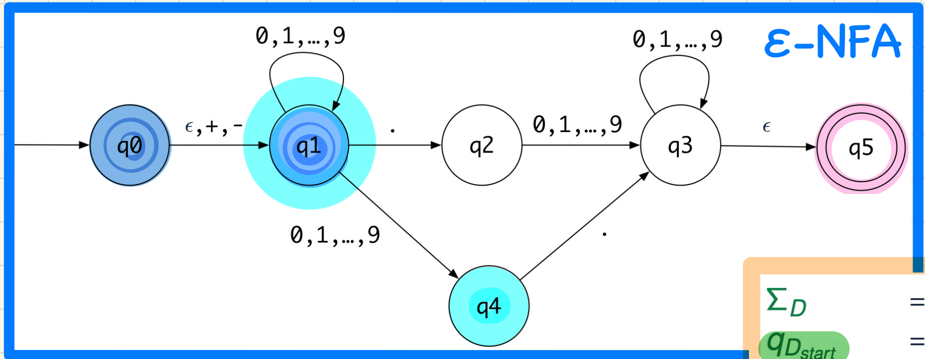
ECLOSE(q_0)

subset state

initial state of DFA.

accepting (subset) states of DFA.

epsilon-NFA to DFA: Extended Subset Construction



each DFA state is a subset of states in ε-NFA

w is a string

$\Sigma_D = \Sigma_N$

$q_{D_{start}} = \text{ECLOSE}(q_0)$

$F_D = \{ S \mid S \subseteq Q_N \wedge S \cap F_N \neq \varnothing \}$

$Q_D = \{ S \mid \boxed{S \subseteq Q_N} \wedge (\exists w \bullet w \in \Sigma^* \Rightarrow S = \hat{\delta}_N(q_0, w)) \}$

$\delta_D(S, a) = \bigcup \{ \text{ECLOSE}(s') \mid s \in S \wedge s' \in \delta_N(s, a) \}$

all subset states reachable from q_0

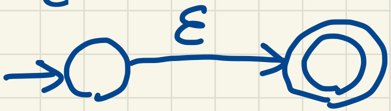
	$d \in 0..9$	$s \in \{+, -\}$	.
$\{q_0, q_1\}$	$\{q_1, q_4\}$	$\{q_1\}$	$\{q_2\}$
$\{q_1, q_4\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2, q_3, q_5\}$
$\{q_1\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2\}$
$\{q_2\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_2, q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$

DFA

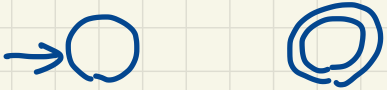
Regular Expression to epsilon-NFA

Base Cases

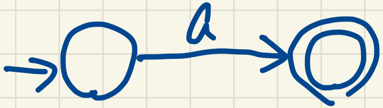
ε



∅

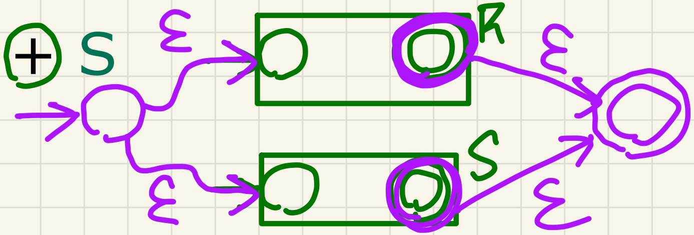


a (a ∈ Σ)

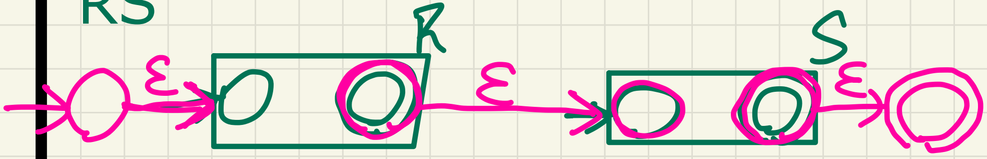


Recursive Cases (given REs E and F)

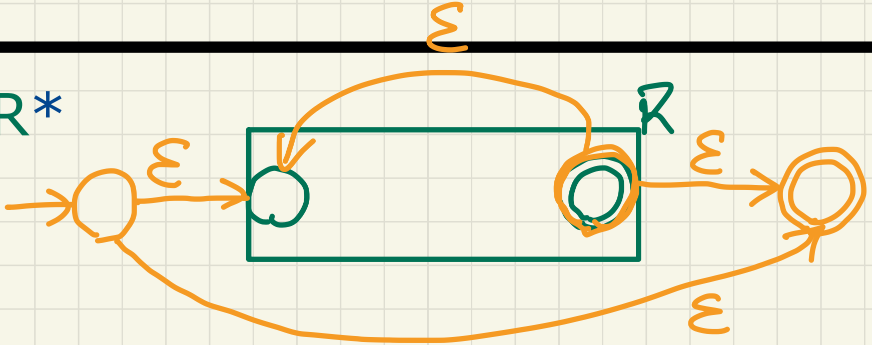
R ⊕ S



RS

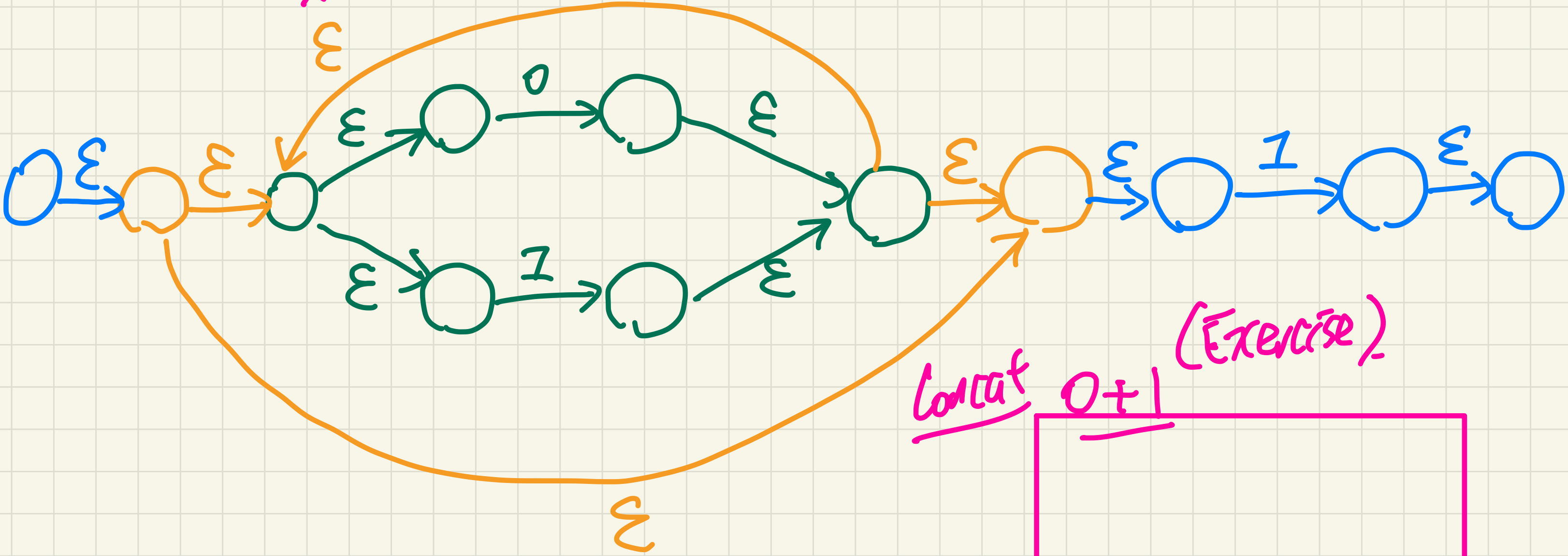


R*



Regular Expression to epsilon-NFA: Example

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



Concat 0+1 (Exercise)

Minimizing DFA: Algorithm

① What if $M' = M \Rightarrow$ no optimization can be done

② Is $|Q(M')| > |Q(M)|$ possible? ⇒ algo. not achieving what it's supposed to.

```
ALGORITHM: MinimizeDFAStates
  INPUT: DFA $M = (Q, \Sigma, \delta, q_0, F)$
  OUTPUT: $M'$ s.t. minimum |Q| and equivalent behaviour as M
PROCEDURE:
  $P := \varnothing$ /* refined partition so far */
  $T := \{ F, Q - F \}$ /* last refined partition */
  while ($P \neq T$):
    $P := T$
    $T := \varnothing$
    for($p \in P$):
      find the maximal $S \subset p$ s.t. splittable($p$, $S$)
      if $S \neq \varnothing$ then
        $T := T \cup \{S, p - S\}$
      else
        $T := T \cup \{p\}$
      end
```

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

1. $S \subset p$ (or equivalently: $p - S \neq \varnothing$)
2. Transitions via c lead all $s \in S$ to states in ***same partition*** $p1$ ($p1 \neq p$).

Partitions of States

input

e.g., $Q = \{s0, s1, s2, s3\}$

- Smallest number of partitions.
- Largest number of partitions.
- Partitions somewhere in-between
- Analogy from Software Testing: Equivalent Classes

single partition

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

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

no optimization

Lecture 9 - Oct. 6

Lexical Analysis, Syntactic Analysis

Minimizing DFA
Implementing a Scanner
Context-Free Grammar (CFG): Basics

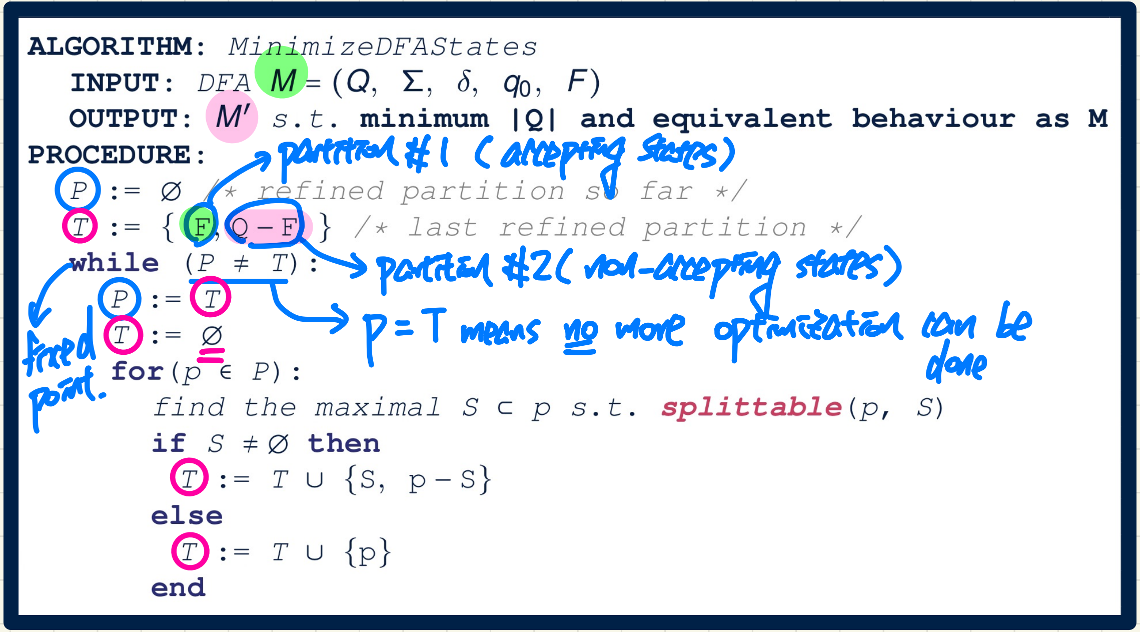
Announcements

- Reading week study item: ANTLR tutorial
  + RE
  + CFG
  + OOP and Composite & visitor design patterns
- Assignment 1 due tomorrow (Friday) at 2pm
- Programming Test date reminder:
  + 2:00pm to 3:20pm on Saturday, October 29
  + Venue to be confirmed
- Quiz 1 to be returned in class on October 17
- Quiz 2 postponed to Thursday, October 19

Minimizing DFA: Algorithm

① What if $M' = M \Rightarrow$ no optimization can be done

② Is $|Q(M')| > |Q(M)|$ possible? ⇒ algo. not achieving what it's supposed to.



```
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):
      find the maximal S ⊂ p s.t. splittable(p, S)
      if S ≠ ∅ then
        T := T ∪ {S, p − S}
      else
        T := T ∪ {p}
      end
```

partition #1 (accepting states)

partition #2 (non-accepting states)

$P = T$ means no more optimization can be done

fixed point.

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

1. $S \subset p$ (or equivalently: $p - S \neq \varnothing$)
2. Transitions via c lead all $s \in S$ to states in ***same partition*** $p1$ ($p1 \neq p$).

Partitions of States

input

e.g., $Q = \{s0, s1, s2, s3\}$

- Smallest number of partitions .
- Largest number of partitions .
- Partitions somewhere in-between
- Analogy from Software Testing: Equivalent Classes

single partition

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

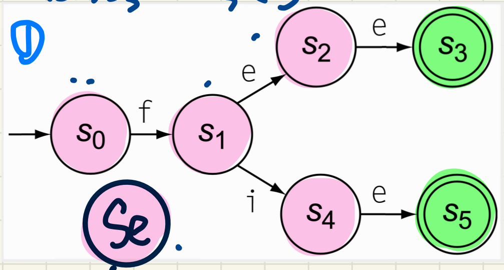
$$Q' = \{ \{s_0\}, \{s_1\}, \{s_2\}, \{s_3\} \}$$

no optimization

Minimizing DFA: Example (1)

$\Sigma = \{a, b, \ldots, z\}$

fee | fie



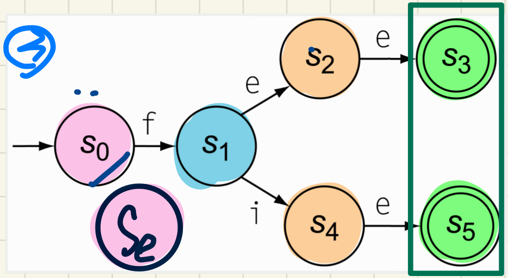
f:

$\{S_0, S_1, S_2, S_4, S_e\} \xrightarrow{f}$ 🟥

e:

$\{S_2, S_4\} \xrightarrow{e}$ 🟩

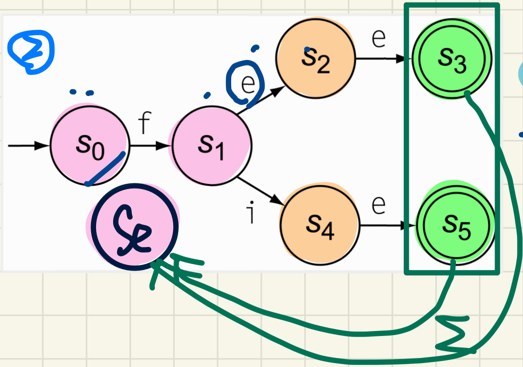
$\{S_e, S_0, S_1\} \xrightarrow{e}$ 🟥



f:

$\{S_0\} \xrightarrow{f}$ 🟦

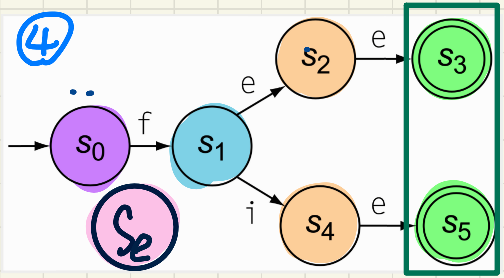
$\{S_e\} \xrightarrow{f}$ 🟥

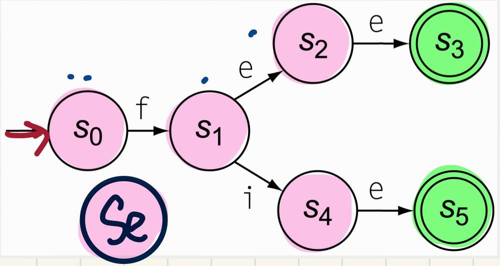


e:

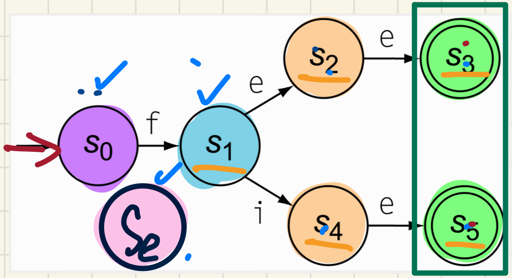
$\{S_1\} \rightarrow$ 🟧

$\{S_0, S_e\} \rightarrow$ 🟥



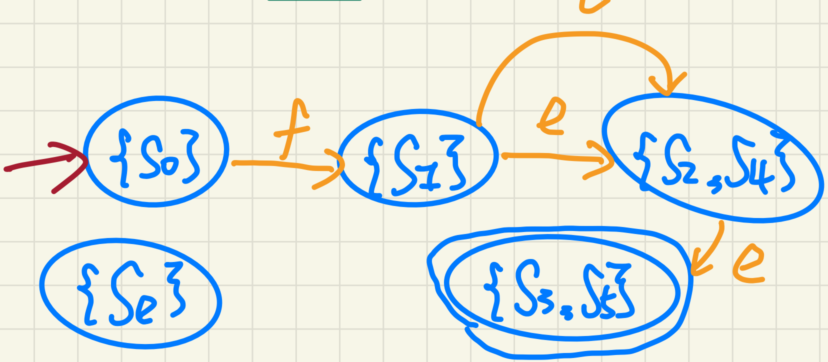


input: 7 states

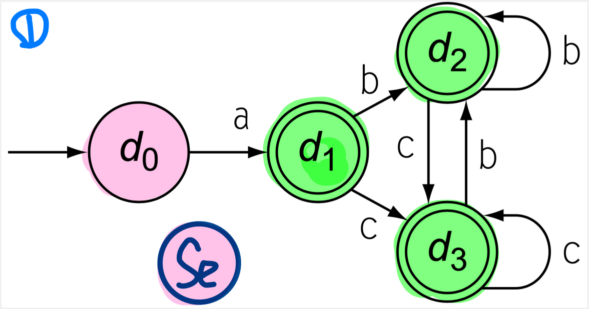


output: 5 partitions

7 states
↓
5 states.



Minimizing DFA: Example (2)



$\Sigma = \{a, b, c\}$

5 states

↓

3 partitions

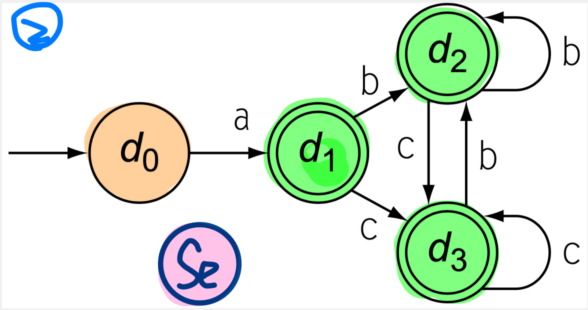
Exercise : draw DFA

partition

a:

$d_0 \xrightarrow{a} \{d_1, d_2, d_3\}$

$Se \xrightarrow{a} \{d_0, Se\}$



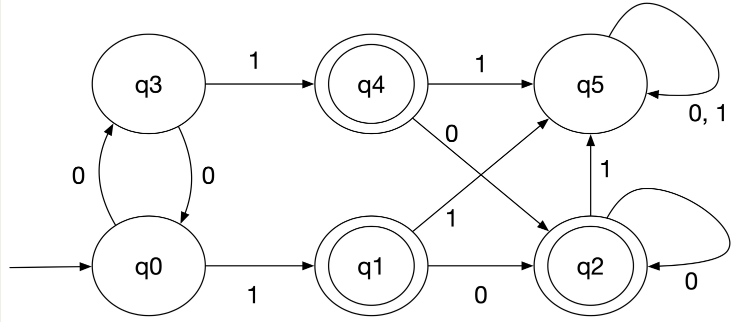
a:

$\{d_1, d_2, d_3\} \xrightarrow{a} \{Se\}$

$\{d_1, d_2, d_3\} \xrightarrow{b} \{d_1, d_2, d_3\}$

$\{d_1, d_2, d_3\} \xrightarrow{c} \{d_1, d_2, d_3\}$

Minimizing DFA: Example (3)



(Exercise).

From RE to Scanner (1)

Token Type (CharCat) ✓ ✓

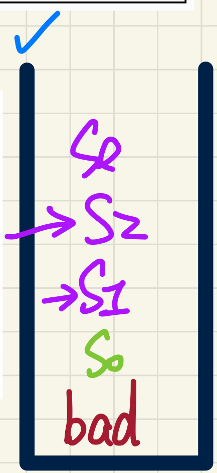
r	0, 1, 2, ..., 9	EOF	Other
Register	Digit	Other	Other

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



Regular Expression: r[0..9]+

```
NextWord()
  -- Stage 1:  Initialization
  state := $s_0$ ; word := $\epsilon$
  initialize an empty stack s ; s.push(bad)
  -- Stage 2:  Scanning Loop
  while (state ≠ $s_e$)
    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: r2~~4~~

EOF

word: ~~ϵ~~ r2

state: ~~s_0~~ ~~s_1~~ s_2

cat: ~~Register~~ Digit

From RE to Scanner (1)

Token Type (CharCat)

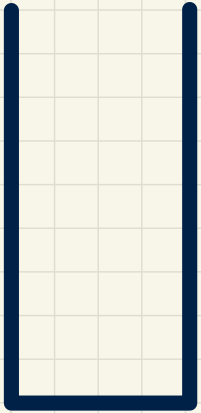
r	0, 1, 2, …, 9	EOF	Other
Register	*Digit*	*Other*	*Other*

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*



Regular Expression: r[0..9]+

```
NextWord()
  -- Stage 1:  Initialization
  state := $s_0$ ; word := $\epsilon$
  initialize an empty stack s ; s.push(bad)
  -- Stage 2:  Scanning Loop
  while (state ≠ $s_e$)
    NextChar(char) ; word := word + char
    if state ∈ F then reset stack s end
    s.push(state)
    cat := CharCat[char]
    state := $\delta$[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: r24*3

EOF.

(Exercise)

word:

state:

cat:

Context-Free Grammar (CFG): Terminology

The following language that is ***non-regular***

$$\{0^n\#1^n \mid n \overset{>}{\cancel{\geq}} 0\}$$

can be described using a ***context-free grammar (CFG)***:

$$\begin{array}{lcl} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \# \end{array}$$

Start variable

production rule

terminals

non-terminals (variables)

Visualization Derivations from CFG

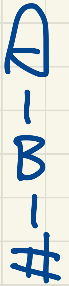
$$A \overset{\textcircled{1}}{\rightarrow} \underline{0A1}$$
$$A \overset{\textcircled{2}}{\rightarrow} B$$
$$B \overset{\textcircled{3}}{\rightarrow} \#$$

- Shortest Derivation? #
- (A) 000#111?
- (B) 010#101? No.

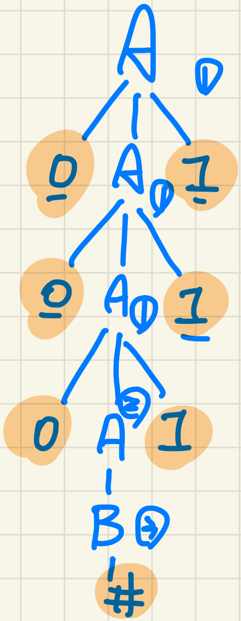
Exercise: Modify/extend the grammar to allow it.

$$A \overset{\textcircled{2}}{\Rightarrow} B \overset{\textcircled{3}}{\Rightarrow} \#$$

(derivation result)



(A)



Lecture 10 - Oct. 18

Syntactic Analysis

CFG: Case Studies

Semantic Analysis vs. Ambiguity

Announcements

- ANTLR tutorial
  - RE
  - CFG
  - OOP and Composite & visitor design patterns
- **Project** to be released by next Tuesday's class

Quiz 2:
1. no proofs
2. no essays.

to be finally confirmed on Thurs. class

- A possible alternative to **ProgTest**?
  14:30 to 16:00, Tuesday, November 1
- **Programming Test** date:
  - 2:00pm to 3:20pm on Saturday, October 29
  - Venue to be confirmed (LAS building)
  - Practice Test
- **Quiz 2** on Thursday, October 19

Discussion: Compare Two CFGs

```
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
```

1. V2 does semantic grouping of operators

? 1+2+3

2. V2 is less Ambiguous

∵ it does not accept 2 => 8

↳ accepted by v1

↳ rejected by v2

only 1 parse tree by v1 v2 CFG

v1

```
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
```

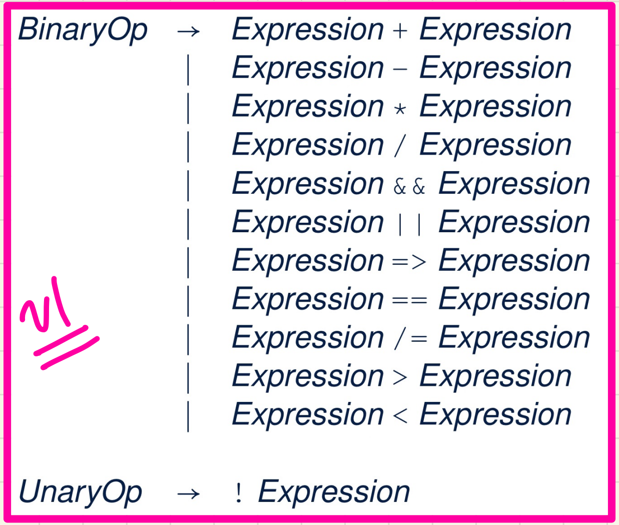
v2

expected numerical exp.

```
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
```

boolean exp.

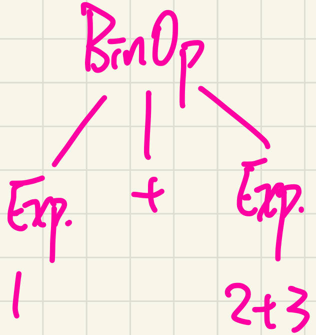
$\boxed{1+2+3}$

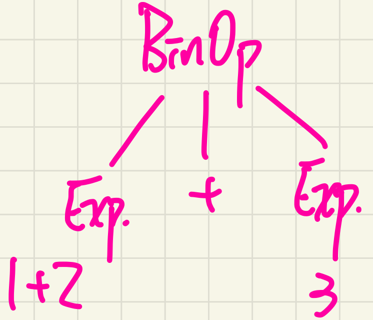


```
BinaryOp  →  Expression + Expression
          |  Expression − Expression
          |  Expression ⋆ Expression
          |  Expression / Expression
          |  Expression && Expression
          |  Expression || Expression
          |  Expression => Expression
          |  Expression == Expression
          |  Expression /= Expression
          |  Expression > Expression
          |  Expression < Expression

UnaryOp   →  ! Expression
```

Ambiguity?





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
```

```
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
```

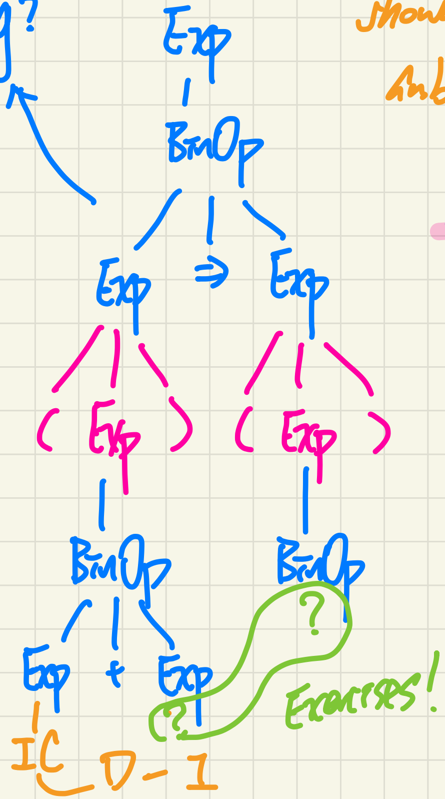
Example: (1 + 2) => (5 / 4)

contains semantic error to be discovered

not appropriate witness for showing the complete ambiguity.

semantic analysis

Is this an AST/PT with valid meaning?



5 - 6

↳ appropriate witness for proving ambiguity (exercise)

Context-Free Grammar (CFG): Example Version 1

1+2+3

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`

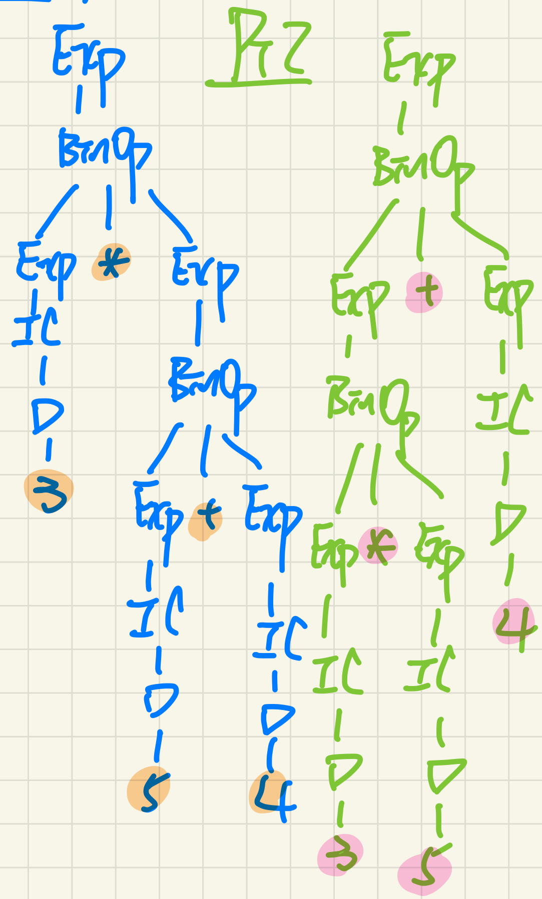
BinaryOp → *Expression* + *Expression*
| *Expression* − *Expression*
| *Expression* ⋆ *Expression*
| *Expression* / *Expression*
| *Expression* && *Expression*
| *Expression* || *Expression*
| *Expression* => *Expression*
| *Expression* == *Expression*
| *Expression* /= *Expression*
| *Expression* >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

UnaryOp → ! *Expression*

Example: 3 * 5 + 4

PT1

witness of ambiguity



Context-Free Grammar (CFG): Example Version 2

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

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

IntegerConstant   →  Digit
                  |  Digit IntegerConstant
                  |  -IntegerConstant

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

BooleanConstant   →  TRUE
                  |  FALSE
```

```
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
```

for v2, it's a parse error (no AST/PT can be built).

↳ not preferred as the user of compiler needs more feedback (e.g. Eclipse)

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
```

```
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
```

Q: No **semantic analysis** at all for Version 2 grammar?

Example: ~~(1 + 2) => (5 - (2 + 3))~~

((1+2)>0) ⇒

(4/(5-(2+3)) > 0)

division by zero.

for simple cases, it might be worth checking it's not 0.

Person P;

P. setName("Jim");

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

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*

Example: 3 * 5 + 4 .

Exercise: show the grammar is ambiguous.

Lecture 11 - Oct. 20

Syntactic Analysis

CFG: Formulation
From RE or DFA to CFG
Ambiguity, Dangling else

Announcements

- **Programming Test**
  - 2:00pm to 3:20pm on Saturday, October 29
  - Venue to be confirmed (LAS building)
- **Project** teammates (gather at the end of the class)

CFG: Formal Definition

Design the CFG for strings of properly-nested parentheses.

e.g., (), () (), ((( ) ( ))) ( ), *etc.*

Present your answer in a formal manner.

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

A ***context-free grammar (CFG)*** is a 4-tuple $(\overset{N.T.}{V}, \overset{T.}{\Sigma}, R, S)$:

- V is a finite set of **variables**/non-terminals
- Σ is a finite set of ***terminals***. $[V \cap \Sigma = \varnothing]$
- R is a finite set of ***rules*** s.t.

$$R \subseteq \{\overset{var}{v} \to \underset{string.}{s} \mid v \in V \land s \in (V \cup \Sigma)^*\}$$

$s \in V^*$ ✗ $s \in \Sigma^*$ ✗ $(V \cup \Sigma)^*$

$s \in V^* \lor s \in \Sigma^*$ ✗

- $S \in V$ is is the **start variable**.

Rules.

$S \to \underline{(S)}$ ↗ mix of t. and n.t.

$S \to \underline{SS}$ variables

Given strings $u, v, w \in \underline{(V \cup \Sigma)^*}$, variable $\underline{A} \in V$, a rule $\underline{A \to w}$:

- $\boxed{\underline{u}\,A\,\underline{v} \overset{A \to w}{\Rightarrow} \underline{u}\,w\,\underline{v}}$ menas that uAv ***yields*** uwv.
- $\boxed{\underline{u} \overset{*}{\Rightarrow} \underline{v}}$ means that u ***derives*** v, if:
  - $\underline{u} = \underline{v}$; or
  - $\underline{u} \Rightarrow \underline{u_1} \Rightarrow \underline{u_2} \Rightarrow \cdots \Rightarrow u_k \Rightarrow \underline{v}$ [a ***yield sequence***]

Given a CFG $G = (V, \Sigma, R, S)$, the language of G

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} \underline{w}\}$$

no non-terminals.

Context-Free Grammar (CFG): Example Version 3

$$\begin{array}{lcl} \textit{Expr} & \to & \textit{Expr} + \textit{Term} \\ & \mid & \textit{Term} \\ \textit{Term} & \to & \textit{Term} * \textit{Factor} \\ & \mid & \textit{Factor} \\ \textit{Factor} & \to & (\,\textit{Expr}\,) \\ & \mid & \mathrm{a} \end{array}$$

different precedences.

Example: a * a + a

↳ Exercise: draw PT.

$Expr \to \boxed{Expr} + Term$

↓

Term

↓

Term * Factor

higher precedence

Context-Free Grammar (CFG): from RE (1)

RE	CFG
$L(\epsilon)$	$S \to \varepsilon$
$L(a)$ $a \in \Sigma$	$S \to a$
$L(E + F)$	$S \to cfg(E) \mid cfg(F)$
$L(EF)$	$S \to cfg(E)\, cfg(F)$
$L(E^*)$	$S \to \varepsilon \mid S\, cfg(E)$
$L((E))$	$S \to (\, cfg(E)\,)$

Context-Free Grammar (CFG): from RE (2)

$$\overbrace{(0+1)^*}^{T}\,\overbrace{1}^{U}\,\overbrace{(0+1)}^{V}$$

$$S \to T\,U\,V$$

$$T \to \varepsilon \mid T\,T_2$$

$$T_2 \to 0 \mid 1$$

$$U \to 1$$

$$V \to 0 \mid 1$$

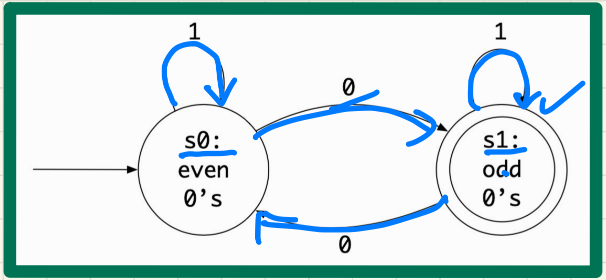
CNF

↳ Chomsky Normal Form

$$(00+1)^* + (11+0)^*$$

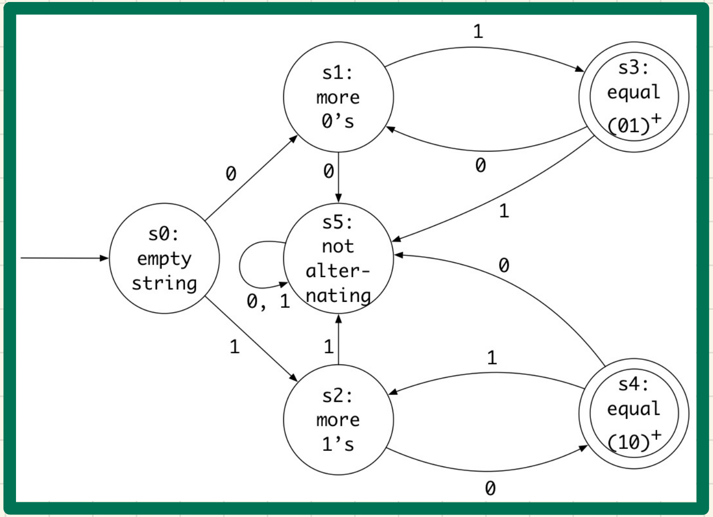
$\boxed{\text{Exercise}}$.

Context-Free Grammar (CFG): from DFA



$S_0 \rightarrow 0S_1 \mid 1S_0$

$S_1 \rightarrow \varepsilon \mid 1S_1 \mid 0S_0$



Exercise

Lecture 12 - Oct. 25

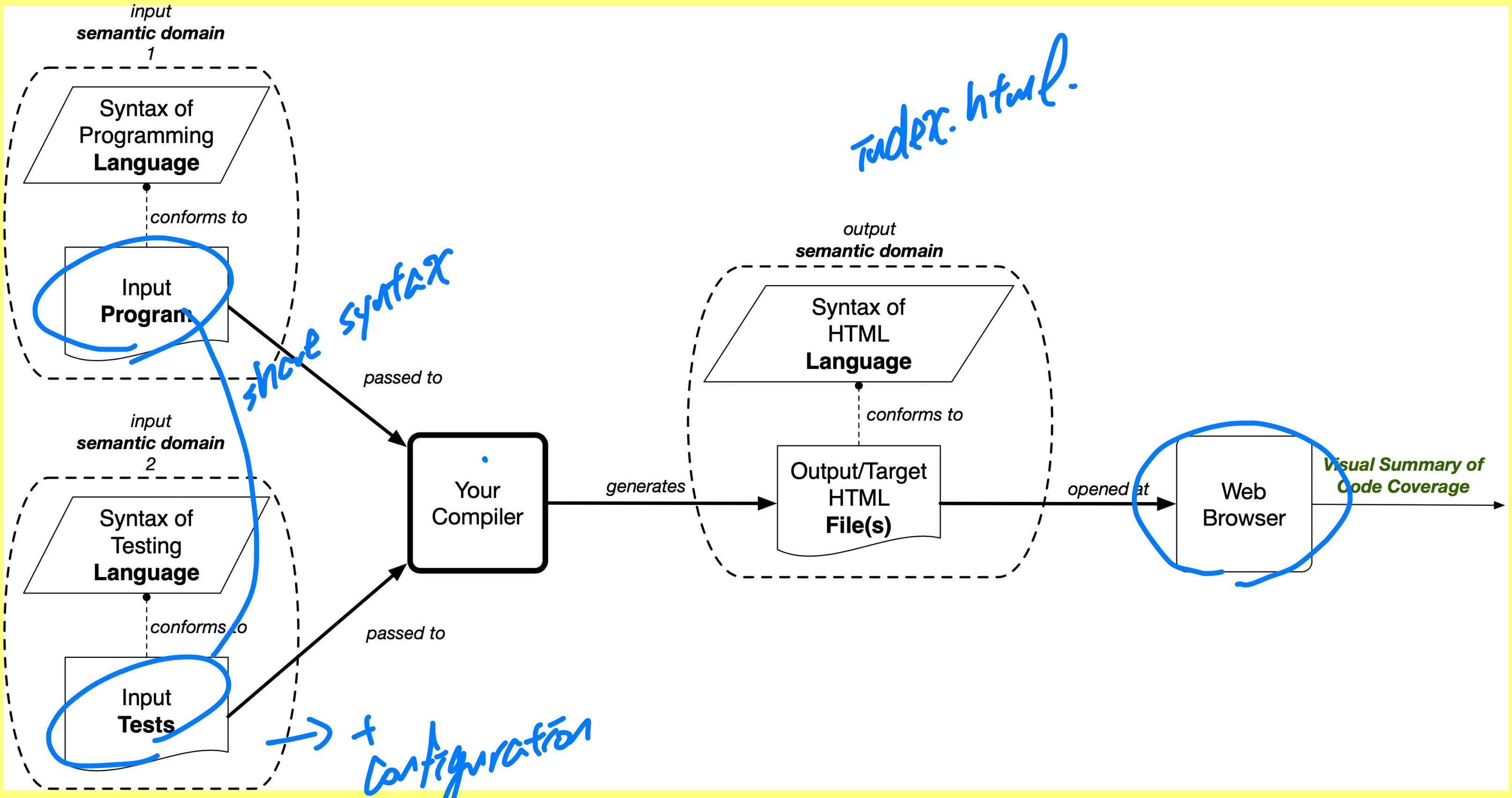
Syntactic Analysis

Derivations vs. Parse Trees
Ambiguity, Dangling else

Announcements

- Project teammate: Jovan
- Programming Test
  - 2:00pm to 3:20pm on Saturday, October 29
  - Venue: LAS1006 (the large lab)
- Exam confirmed by the registrar office:
  - 2pm to 5pm, Saturday, December 10
  - Last Class: Tuesday, December 6
  - Review session?
- Updated Calendar
- Quiz 3
- Project Specification

Project: Roadmap



input
semantic domain
1

Syntax of Programming **Language**

conforms to

Input **Program**

passed to

input
semantic domain
2

Syntax of Testing **Language**

conforms to

Input **Tests**

passed to

Your Compiler

generates

output
semantic domain

Syntax of HTML **Language**

conforms to

Output/Target HTML **File(s)**

opened at

Web Browser

Visual Summary of Code Coverage

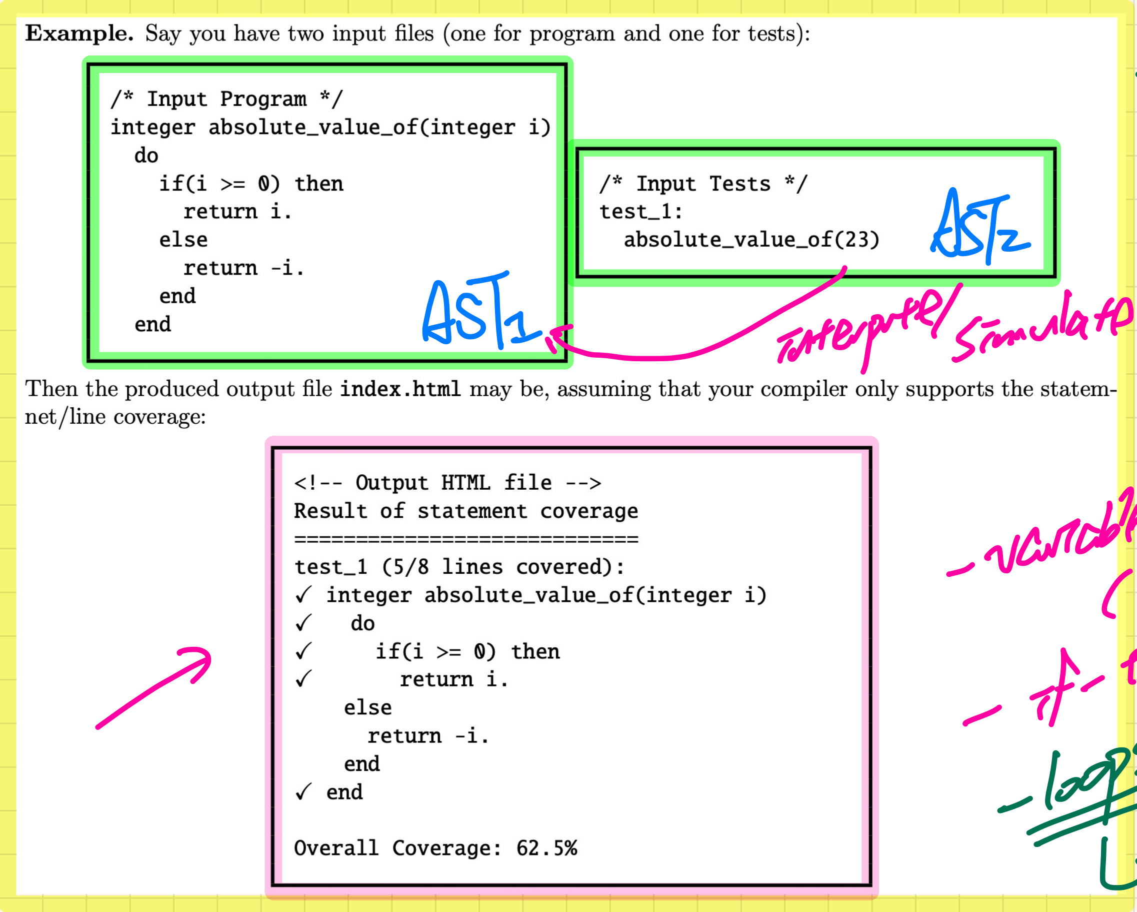
index.html.

share syntax

→ + configuration

Project: Example

for(int i from 10 to 70)



Example. Say you have two input files (one for program and one for tests):

```
/* Input Program */
integer absolute_value_of(integer i)
  do
    if(i >= 0) then
      return i.
    else
      return -i.
    end
  end
```

AST1

```
/* Input Tests */
test_1:
  absolute_value_of(23)
```

AST2

Interpreter/ Simulator

Then the produced output file **index.html** may be, assuming that your compiler only supports the statemnet/line coverage:

```
<!-- Output HTML file -->
Result of statement coverage
============================
test_1 (5/8 lines covered):
✓ integer absolute_value_of(integer i)
✓   do
✓     if(i >= 0) then
✓       return i.
      else
        return -i.
      end
✓ end

Overall Coverage: 62.5%
```

- variable assignments (expressions)
- if-then-else
- loops
  while( )
  ↳ bounded loop.

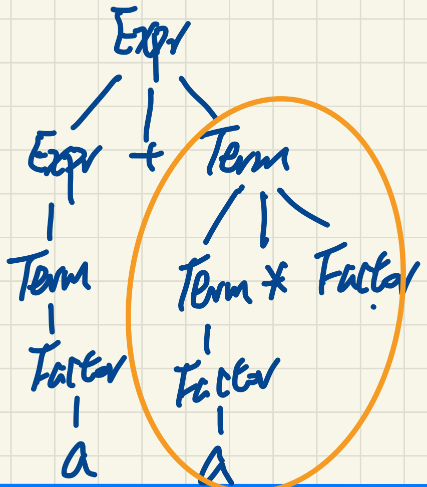
- Grading (part of) of compiler

① run examples supplied by you

② modify/create examples based on
   (a) your supplied examples
   (b) supported syntax

Context-Free Grammar (CFG): Leftmost Derivation

Parse Tree: a + a * a



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

LMD: a + a * a

before + can be applied, Term must be evaluated first

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

Order of evaluation?

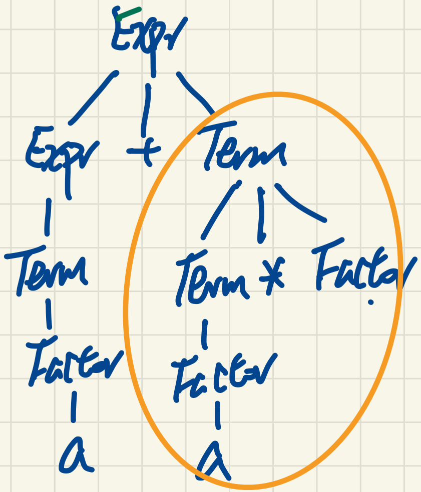
A **parse tree** may correspond to:

+ multiple derivations
+ a unique **LMD**

⇒ a + F * F
⇒ a + F * a
⇒ a + a * a

Context-Free Grammar (CFG): Rightmost Derivation

Parse Tree: a + a * a



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

RMD: a + a * a (Exercise).

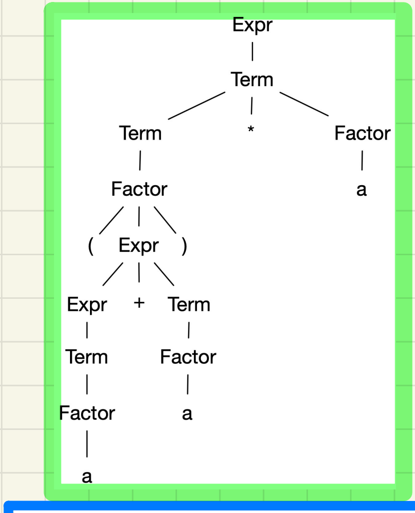
Order of evaluation?

A **parse tree** may correspond to:

+ multiple **derivations**
+ a unique **RMD**

Context-Free Grammar (CFG): Leftmost Derivation

Parse Tree: (a + a) * a



LMD: (a + a) * a

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

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \star \mathit{Factor} \\
 & | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & (\,\mathit{Expr}\,) \\
 & | & \mathrm{a}
\end{array}
$$

Order of evaluation?

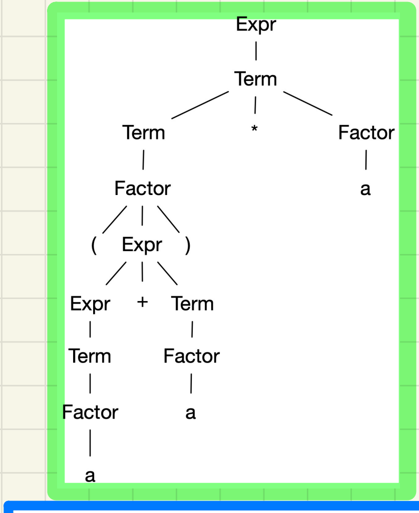
A parse tree may correspond to:

+ multiple derivations

+ a unique LMD

Context-Free Grammar (CFG): Rightmost Derivation

Parse Tree: (a + a) * a



RMD: (a + a) * a

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

$$
\begin{aligned}
Expr &\rightarrow Expr + Term \\
&\mid\ Term \\
Term &\rightarrow Term \ast Factor \\
&\mid\ Factor \\
Factor &\rightarrow (\,Expr\,) \\
&\mid\ \mathrm{a}
\end{aligned}
$$

Order of evaluation?

A **parse tree** may correspond to:

+ multiple **derivations**

+ a unique **RMD**

Q. A grammar is ambiguous if there's a string for which there are two or more derivations.

A. ?

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

Is the following CFG ambiguous?

$$\boxed{Expr \rightarrow Expr \; + \; Expr \mid Expr \; * \; Expr \mid (\; Expr \;) \mid a}$$

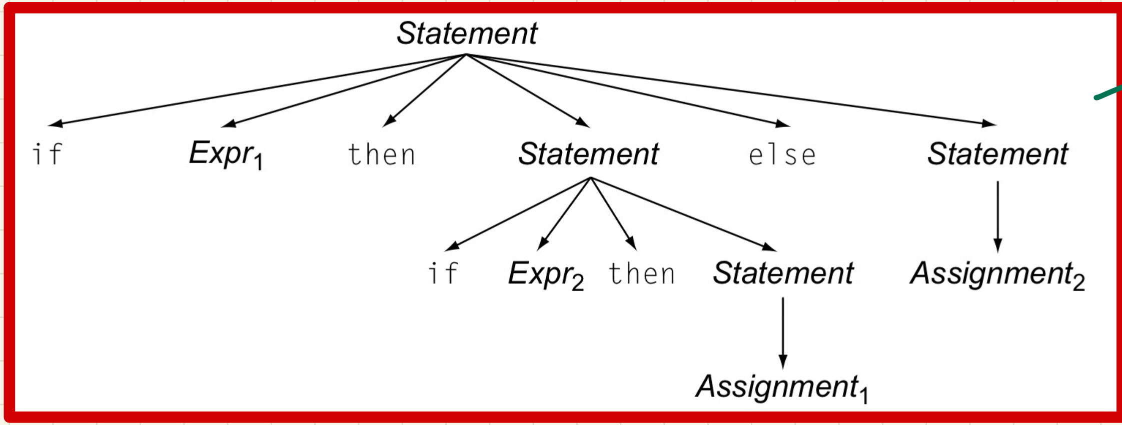
Context-Free Grammar (CFG): Exercise (2.1.1)

Is the following CFG ambiguous?

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

Example: A Possible **Semantic Interpretation**?

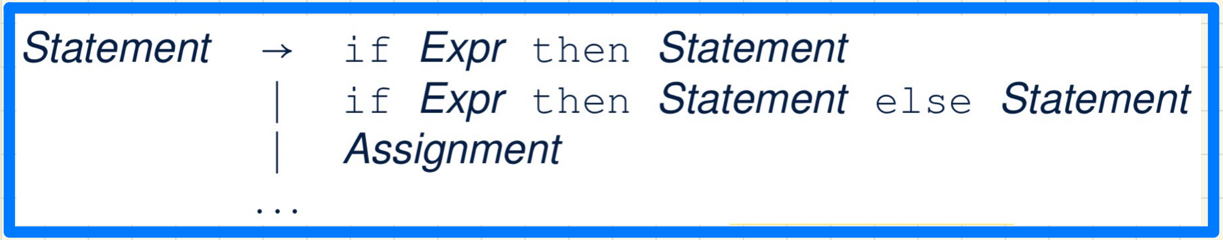
if *Expr1* **then** **if** *Expr2* **then** *Assignment1* **else** *Assignment2*



Exercise: Use two different LMDs to show.

Context-Free Grammar (CFG): Exercise (2.1.2)

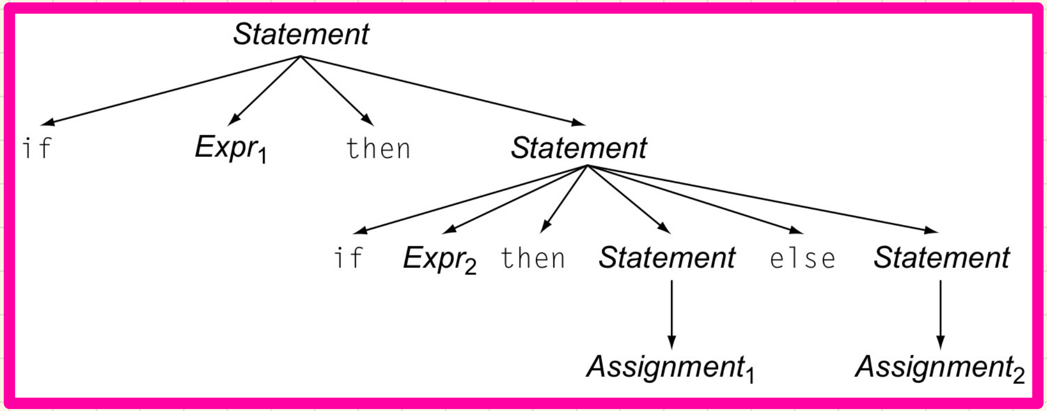
Is the following CFG ambiguous?



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

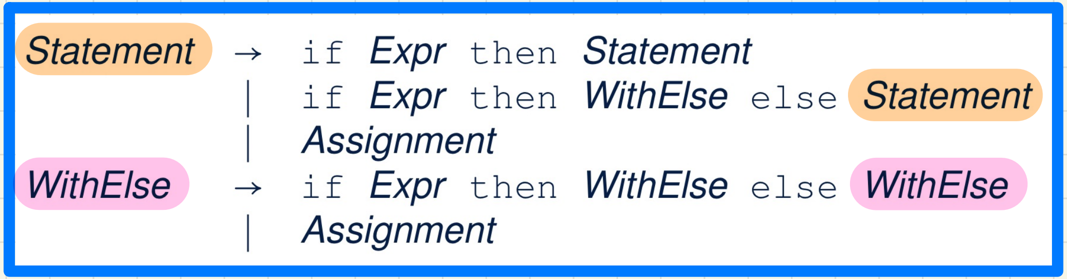
Example: A Possible **Semantic Interpretation**?

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



Context-Free Grammar (CFG): Exercise (2.2)

Is the following CFG ambiguous?



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

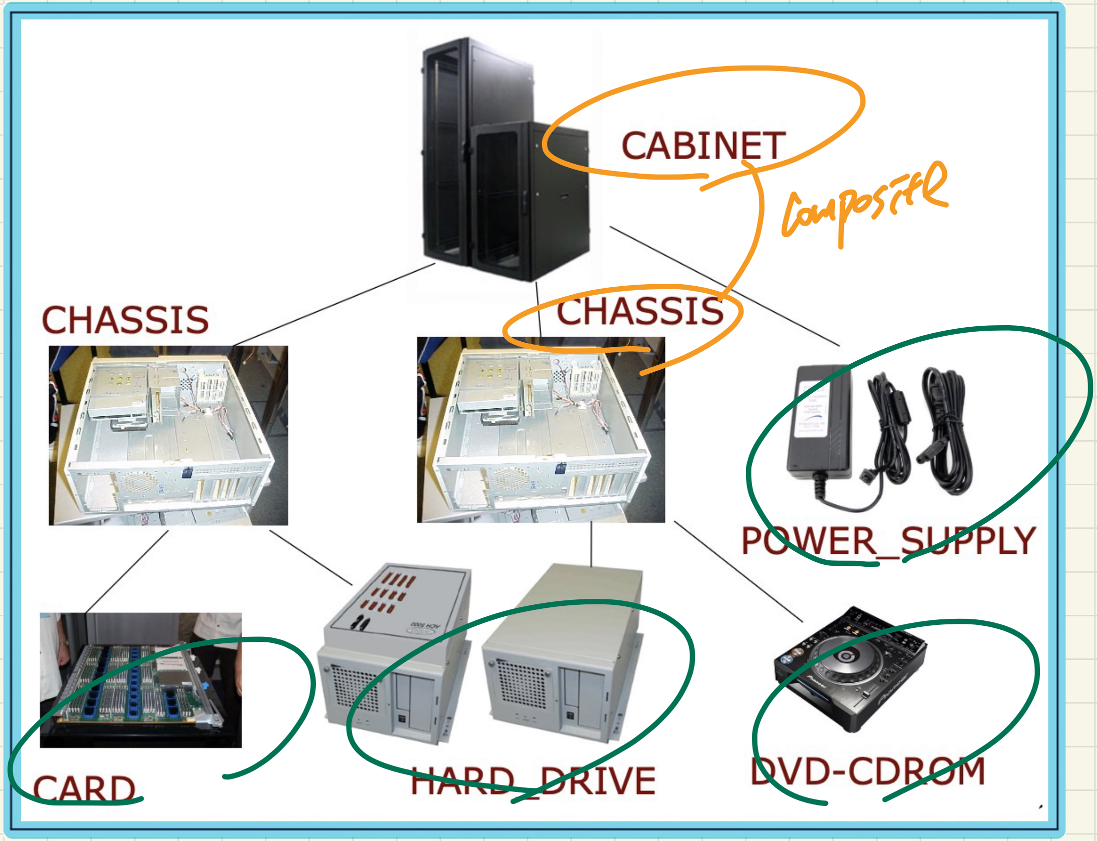
Example: How many possible **semantic interpretations**?

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

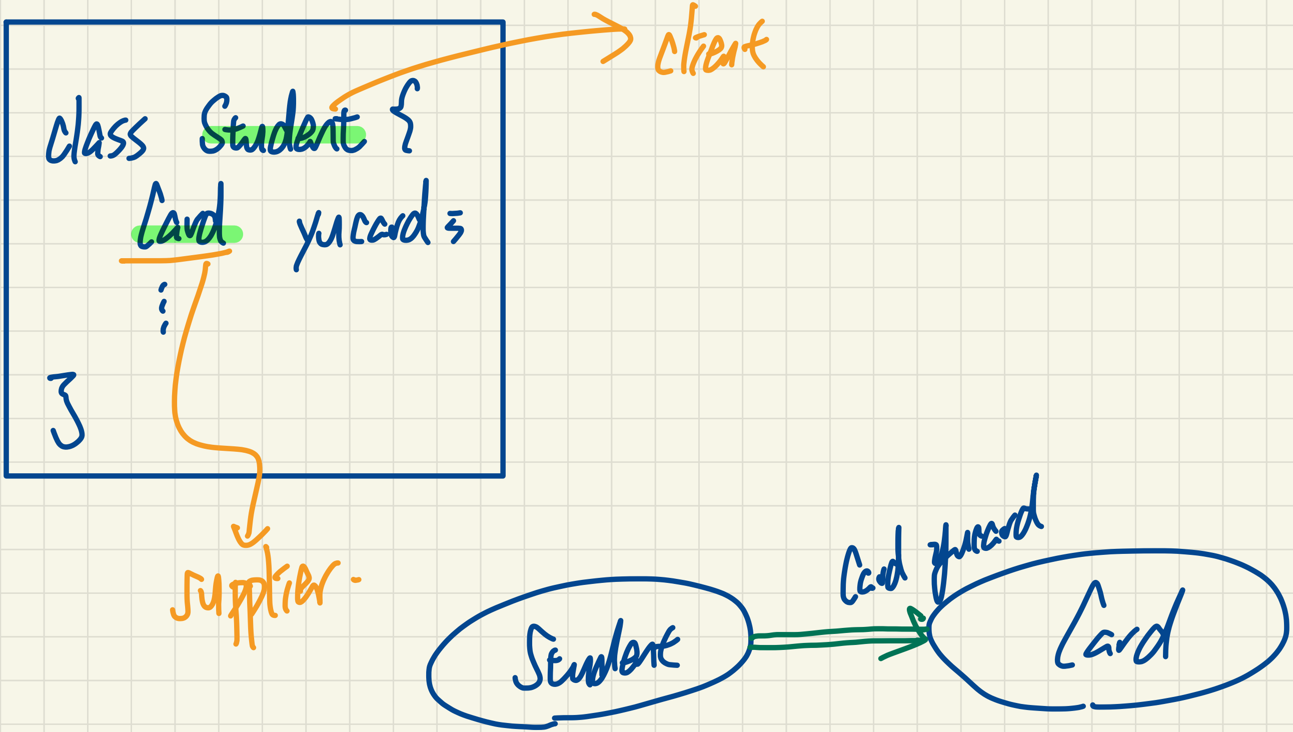
Can a derivation starting with **Statement** work?

Can a derivation starting with **WithElse** work?

Motivation Problem: Recursive Systems

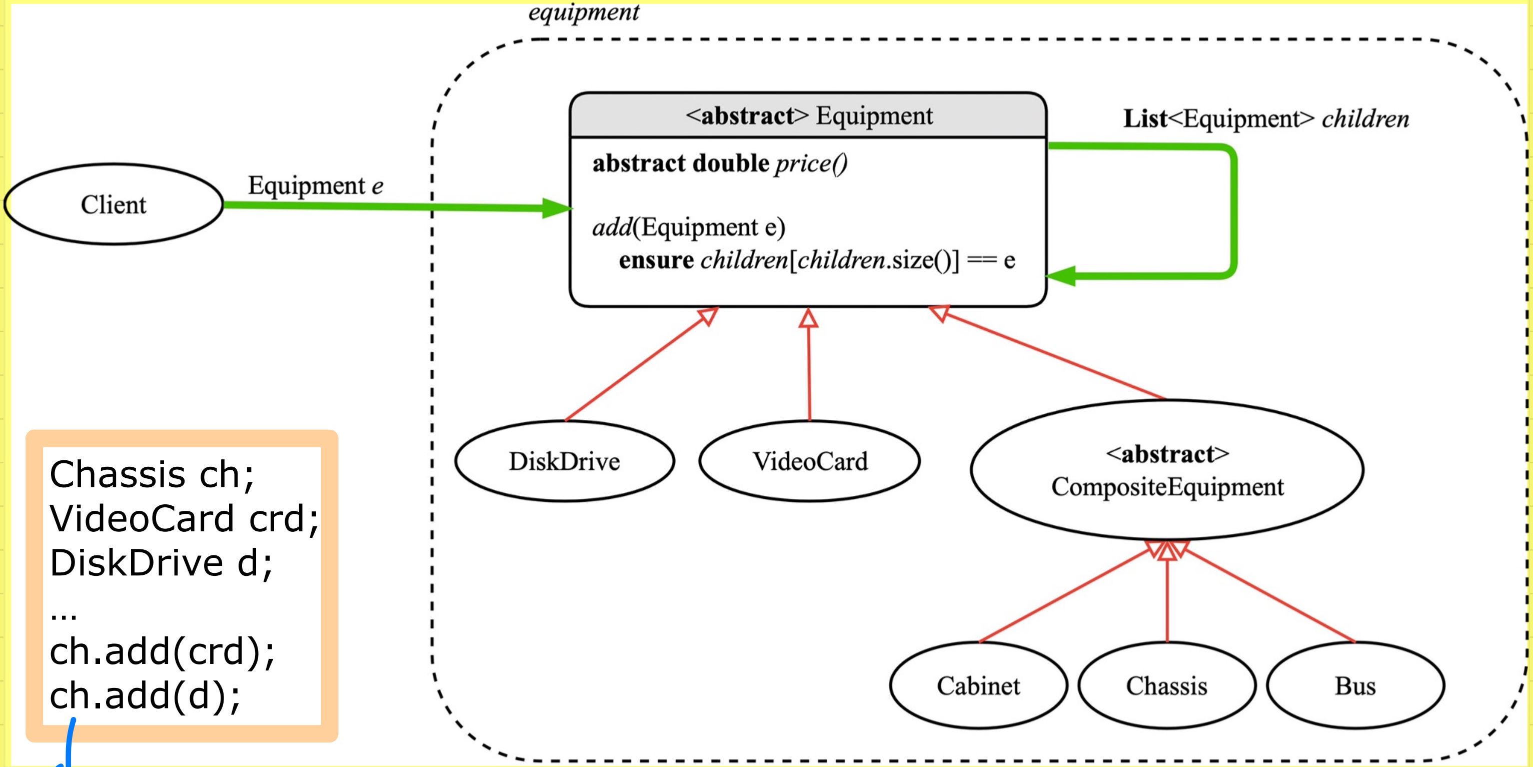


base components



```
class Student {
    Card yucard;
    ⋮
}
```

First Design Attempt



equipment

Client

Equipment *e*

<**abstract**> Equipment

abstract double *price()*

add(Equipment e)
ensure *children*[*children*.size()] == e

List<Equipment> *children*

DiskDrive

VideoCard

<**abstract**>
CompositeEquipment

Cabinet

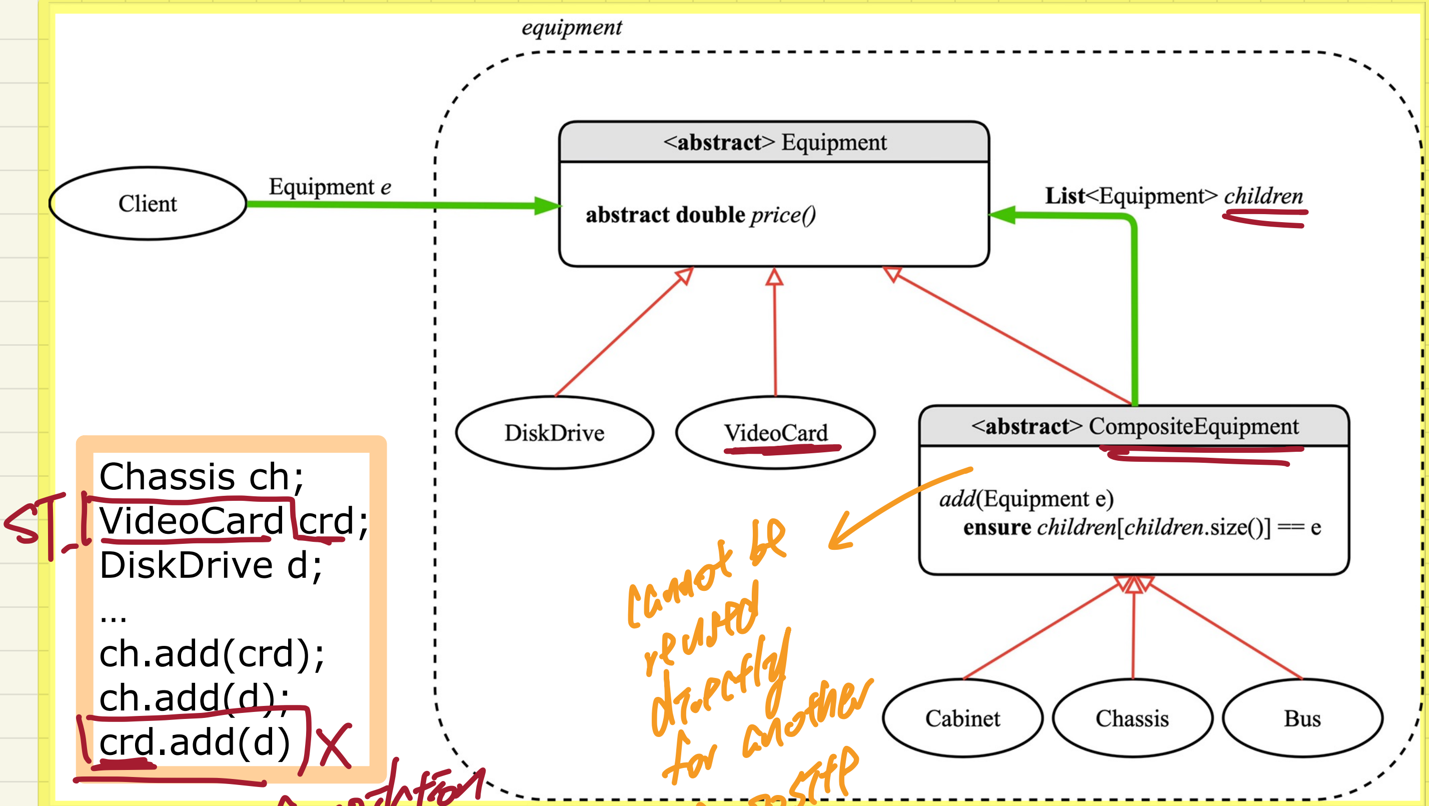
Chassis

Bus

```
Chassis ch;
VideoCard crd;
DiskDrive d;
…
ch.add(crd);
ch.add(d);
```

crd.add(d) : supported but doesn't make real sense

Second Design Attempt


equipment
Client
Equipment *e*
<**abstract**> Equipment
abstract double *price()*
List<Equipment> *children*
DiskDrive
VideoCard
<**abstract**> CompositeEquipment
add(Equipment e)
ensure *children*[*children*.size()] == e
Cabinet
Chassis
Bus

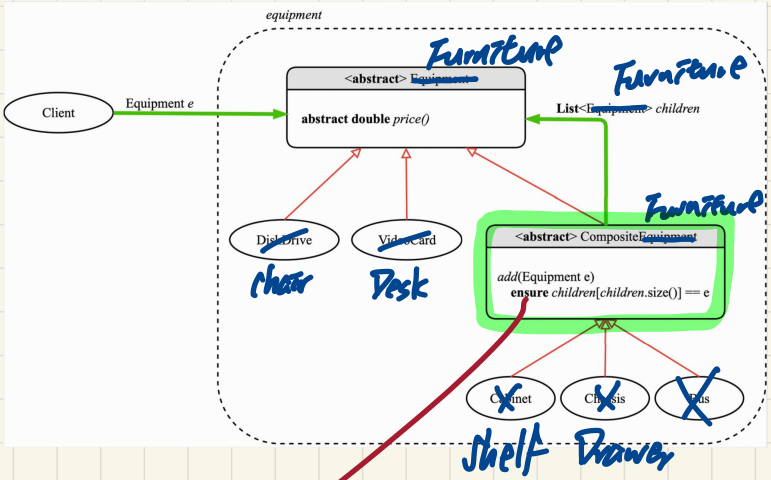
```
Chassis ch;
VideoCard crd;
DiskDrive d;
...
ch.add(crd);
ch.add(d);
crd.add(d)
```

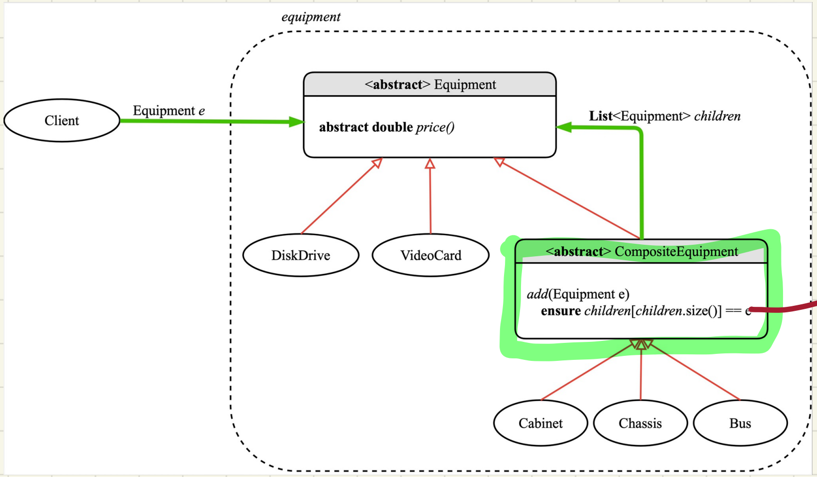
ST_

X

Compilation error

cannot be reused directly for another composite type e.g., furniture.


equipment
Client
Equipment e
<abstract> ~~Equipment~~ Furniture
abstract double price()
List<~~Equipment~~ Furniture> children
~~DiskDrive~~ Chair
~~VideoCard~~ Desk
<abstract> Composite~~Equipment~~ Furniture
add(Equipment e)
ensure children[children.size()] == e
~~Cabinet~~ Shelf
~~Chassis~~ Drawer
~~Bus~~



duplicated code

⇒ the design smells!

Lecture 13 - Oct. 27

Composite & Visitor

Composite:

Architecture, Implementation, Tests

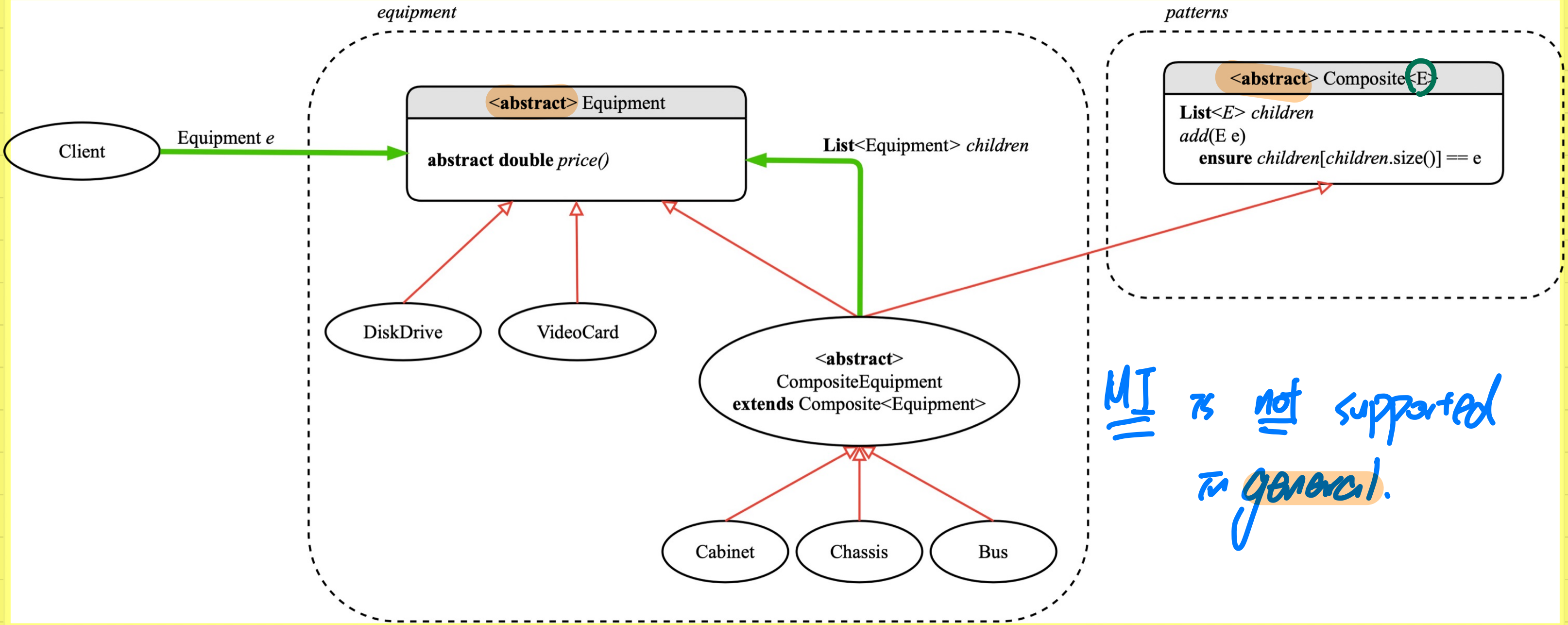
Visitor:

Architecture, Double Dispatch

Announcements

- Programming Test
  - 2:00pm to 3:20pm on Saturday, October 29
  - Venue: LAS1006 (the large lab)
- Quiz 3
- Project team.txt file due today
- Project Milestone 1

Third Design Attempt



equipment

patterns

Client

Equipment *e*

<**abstract**> Equipment

abstract double *price()*

List<Equipment> *children*

<**abstract**> Composite<E>

List<*E*> *children*

add(E e)

ensure *children*[*children*.size()] == e

DiskDrive

VideoCard

<**abstract**>
CompositeEquipment
extends Composite<Equipment>

Cabinet

Chassis

Bus

MI is not supported in general.

- abstract class → a class can extend at most one class (abstract or not)
  - ↳ method: abstract vs. non-abstract
  - ↳ non-static attribute

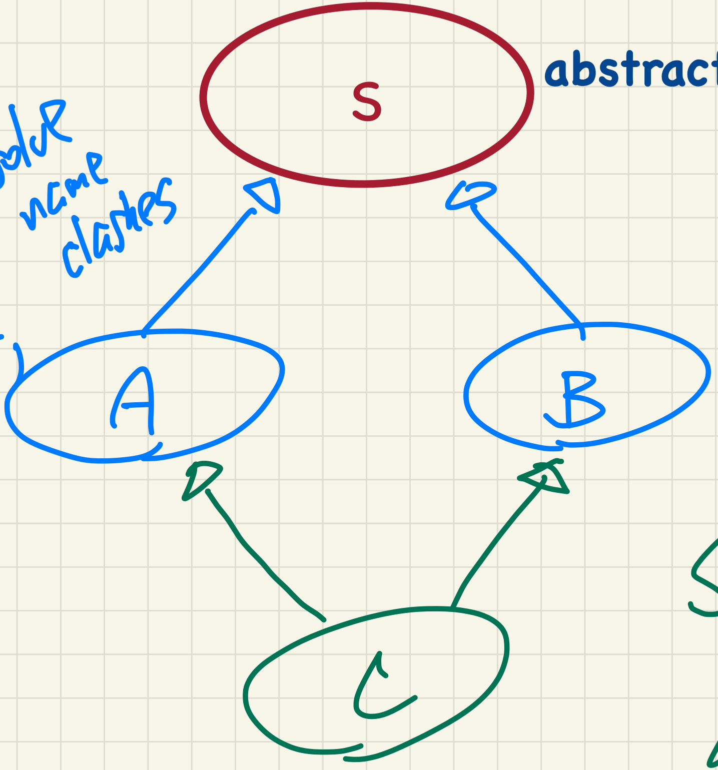
- interface → implement multiple interface
  - ↳ all methods are abstract
  - ↳ no non-static attributes
  - ↳ may declare static variable

Multiple Inheritance in Java: Diamond Problem

Eiffel

↳ MI

↳ resolve name clashes



abstract void doSomething();

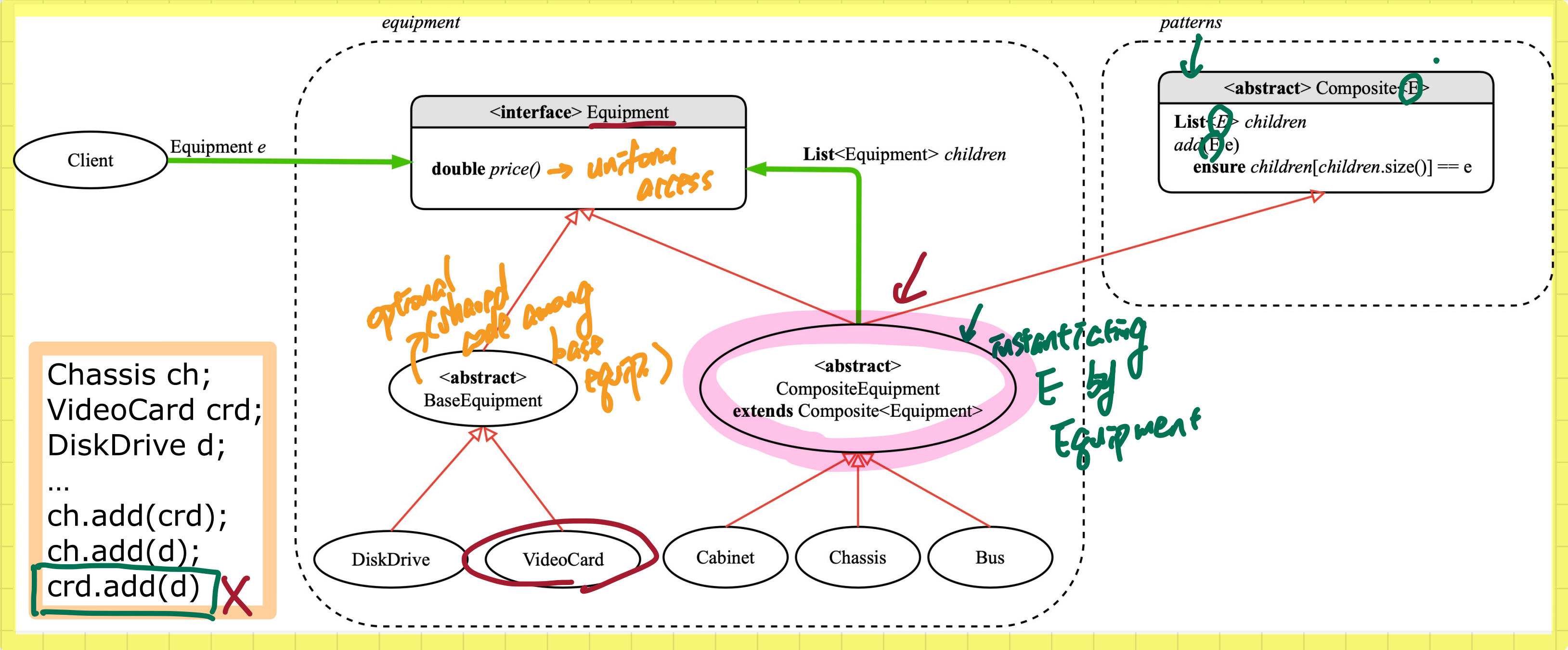
v1 doSomething()

v2 doSomething()

S obj = new C();

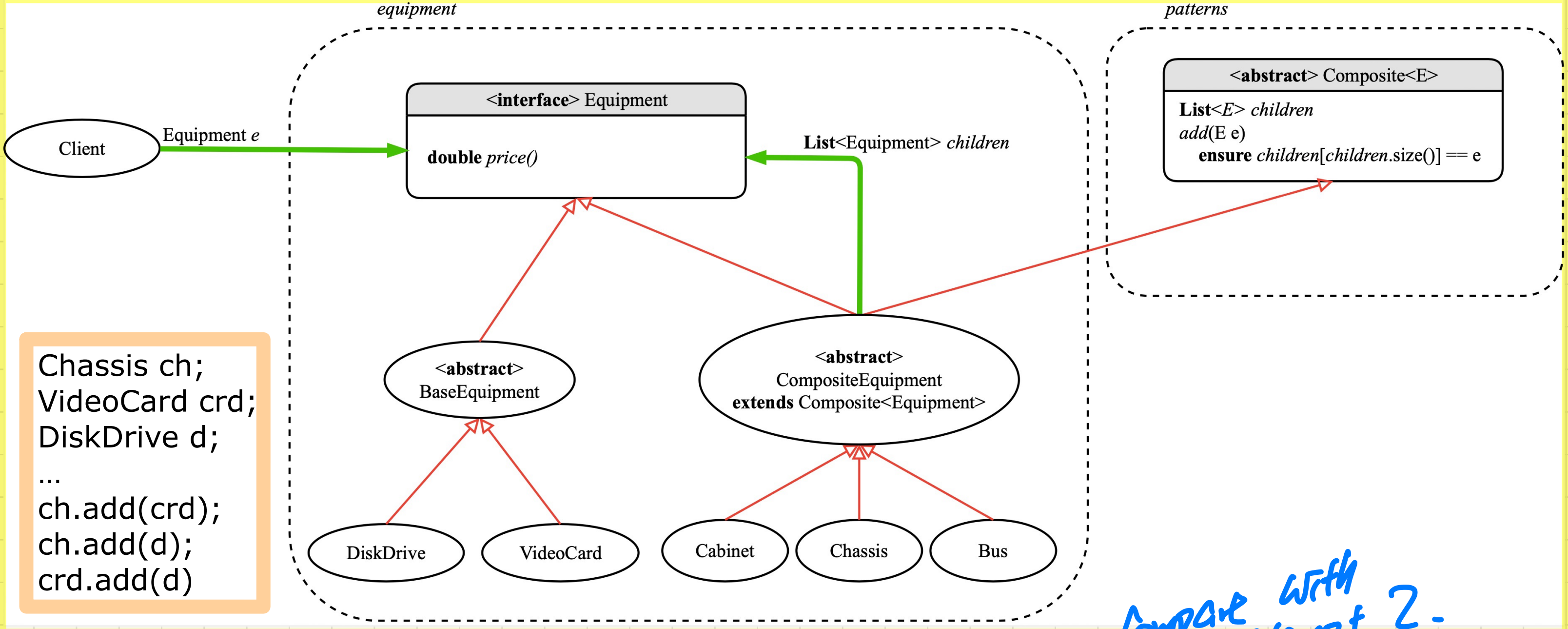
obj.doSomething();

Composite Pattern: Architecture



```
Chassis ch;
VideoCard crd;
DiskDrive d;
…
ch.add(crd);
ch.add(d);
crd.add(d)
```

Composite Pattern: Architecture



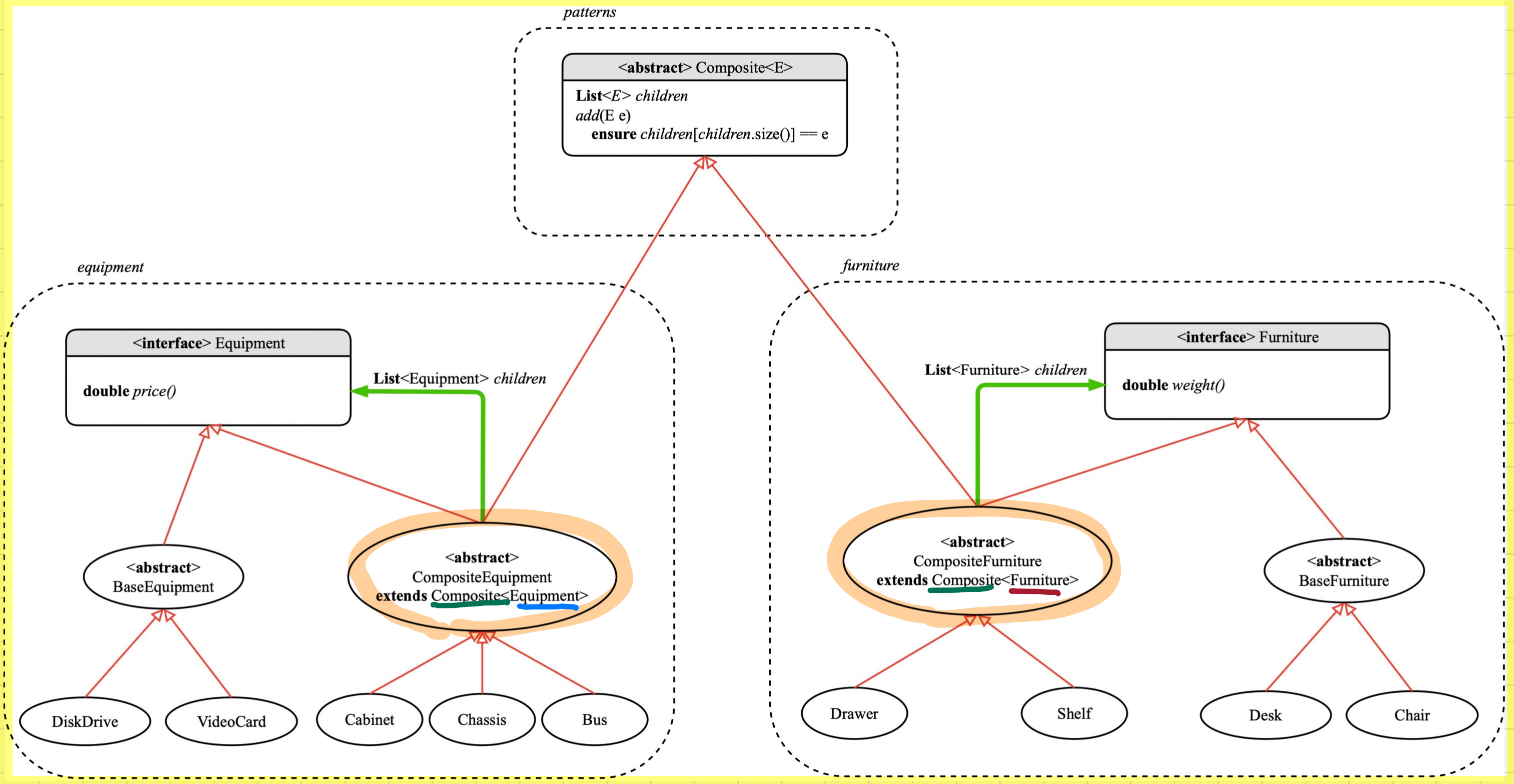
```
Chassis ch;
VideoCard crd;
DiskDrive d;
…
ch.add(crd);
ch.add(d);
crd.add(d)
```

Compare with attempt 2.

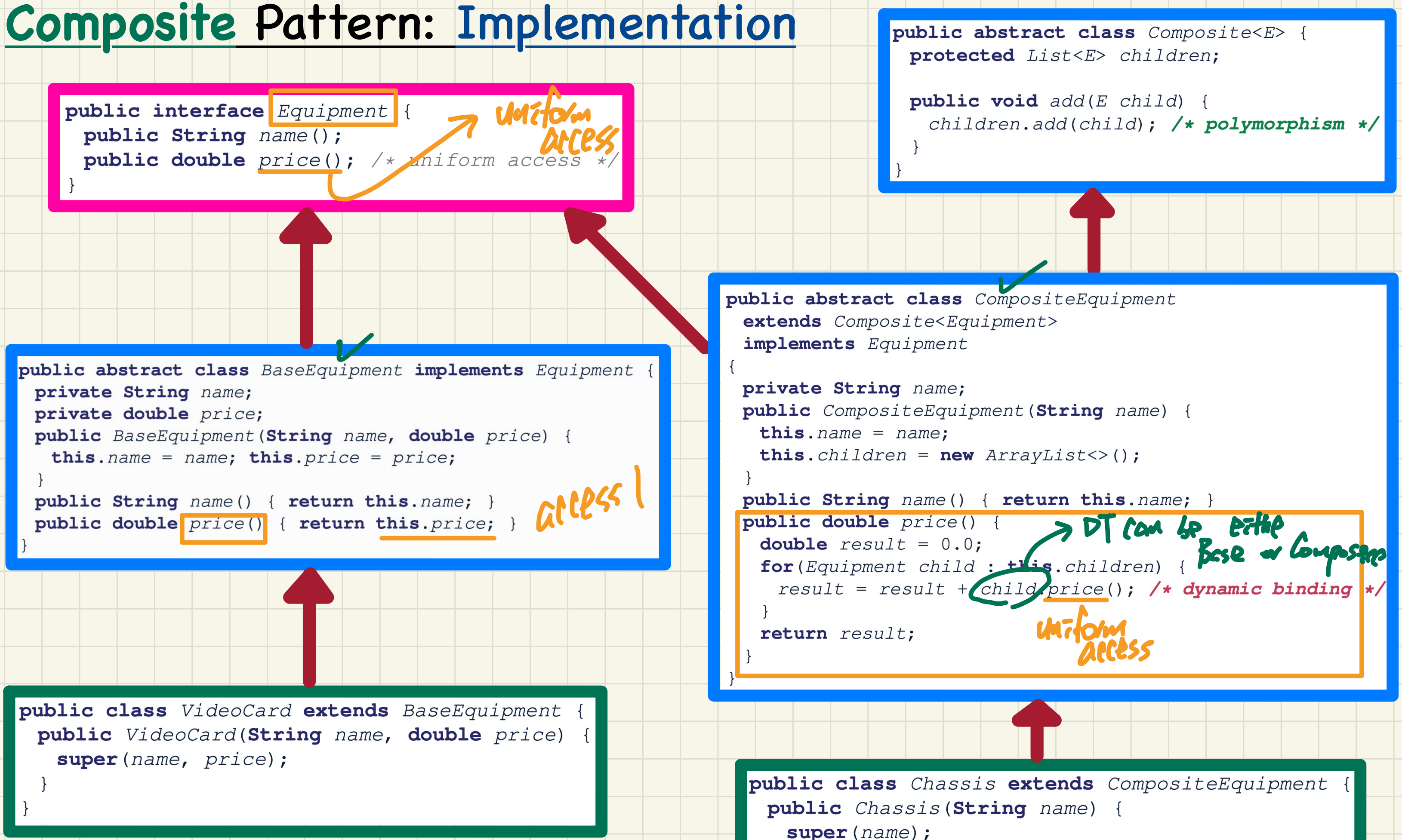
Why is **Composite** a separate, generic class?

Composite Pattern: Architecture

Composite class is **reusable** by instances of the **composite** pattern.



Composite Pattern: Implementation



```
public interface Equipment {
  public String name();
  public double price(); /* uniform access */
}
```

uniform access

```
public abstract class Composite<E> {
  protected List<E> children;

  public void add(E child) {
    children.add(child); /* polymorphism */
  }
}
```

```
public abstract class BaseEquipment implements Equipment {
  private String name;
  private double price;
  public BaseEquipment(String name, double price) {
    this.name = name; this.price = price;
  }
  public String name() { return this.name; }
  public double price() { return this.price; }
}
```

access

```
public abstract class CompositeEquipment
  extends Composite<Equipment>
  implements Equipment
{
  private String name;
  public CompositeEquipment(String name) {
    this.name = name;
    this.children = new ArrayList<>();
  }
  public String name() { return this.name; }
  public double price() {
    double result = 0.0;
    for(Equipment child : this.children) {
      result = result + child.price(); /* dynamic binding */
    }
    return result;
  }
}
```

DT can be either Base or Composite

uniform access

```
public class VideoCard extends BaseEquipment {
  public VideoCard(String name, double price) {
    super(name, price);
  }
}
```

```
public class Chassis extends CompositeEquipment {
  public Chassis(String name) {
    super(name);
  }
}
```

Composite Pattern: Testing

```java
@Test
public void test_equipment() {
  Equipment card, drive;
  Bus bus;
  Cabinet cabinet;
  Chassis chassis;

  card = new VideoCard("16Mbs Token Ring", 200);
  drive = new DiskDrive("500 GB harddrive", 500);
  bus = new Bus("MCA Bus");
  chassis = new Chassis("PC Chassis");
  cabinet = new Cabinet("PC Cabinet");
  bus.add(card);
  chassis.add(bus);
  chassis.add(drive);
  cabinet.add(chassis);

  assertEquals(700.00, cabinet.price(), 0.1);
}
```

500 ~~chassis.children[1].price()~~

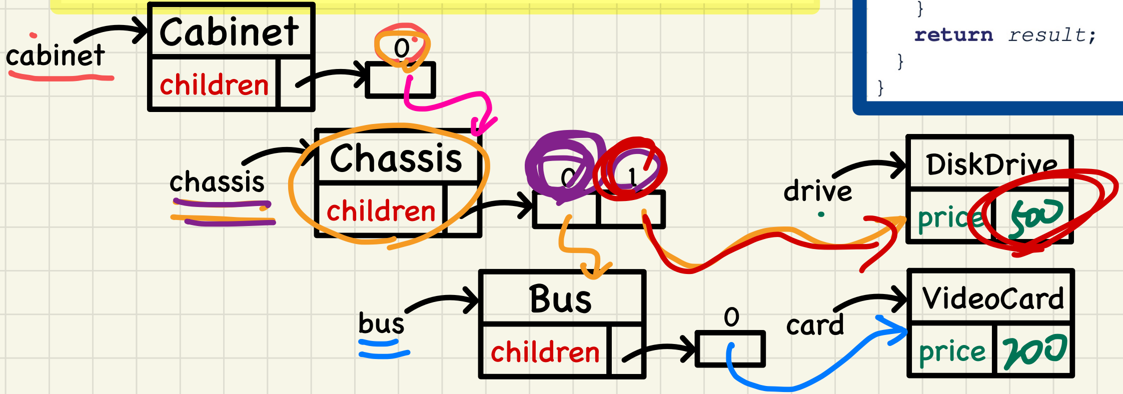
200 ~~chassis.children[0].price()~~

cabinet.c[0].price()

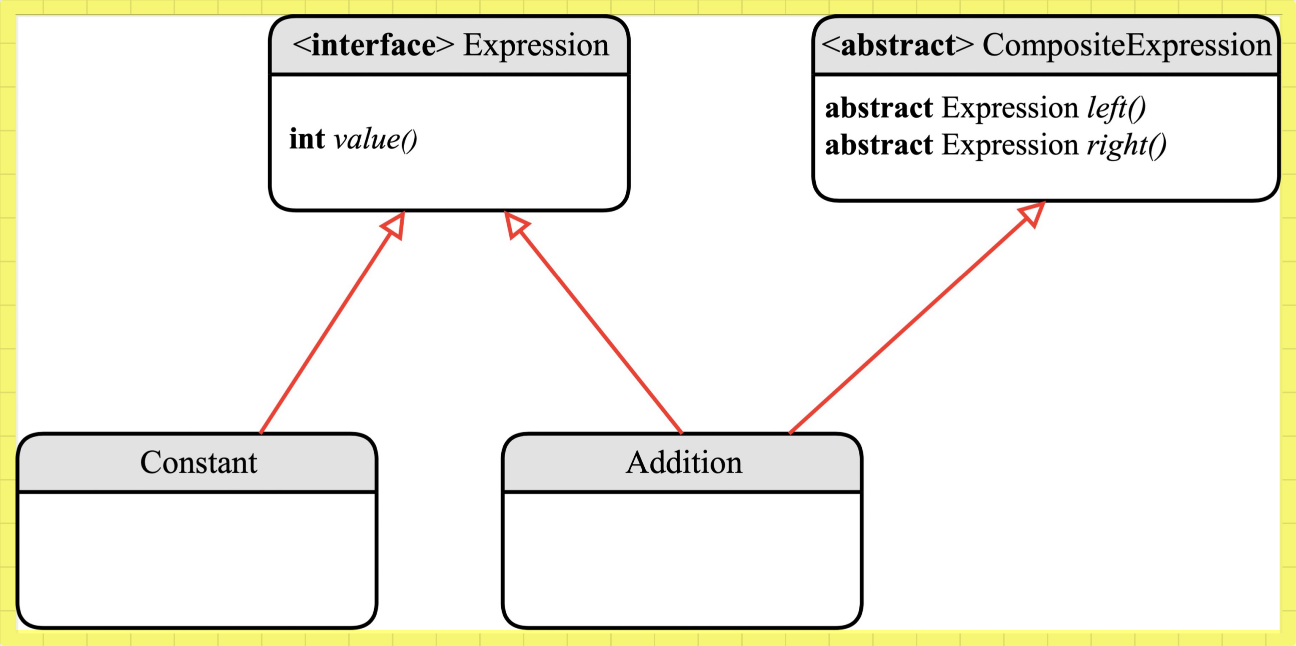
cabinet.price()

```java
public abstract class BaseEquipment implements Equipment {
  private String name;
  private double price;
  public BaseEquipment(String name, double price) {
    this.name = name; this.price = price;
  }
  public String name() { return this.name; }
  public double price() { return this.price; }
}
```

```java
public abstract class CompositeEquipment
  extends Composite<Equipment>
  implements Equipment
{
  private String name;
  public CompositeEquipment(String name) {
    this.name = name;
    this.children = new ArrayList<>();
  }
  public String name() { return this.name; }
  public double price() {
    double result = 0.0;
    for(Equipment child : this.children) {
      result = result + child.price(); /* dynamic binding */
    }
    return result;
  }
}
```



Design of Language Structure: Composite Pattern

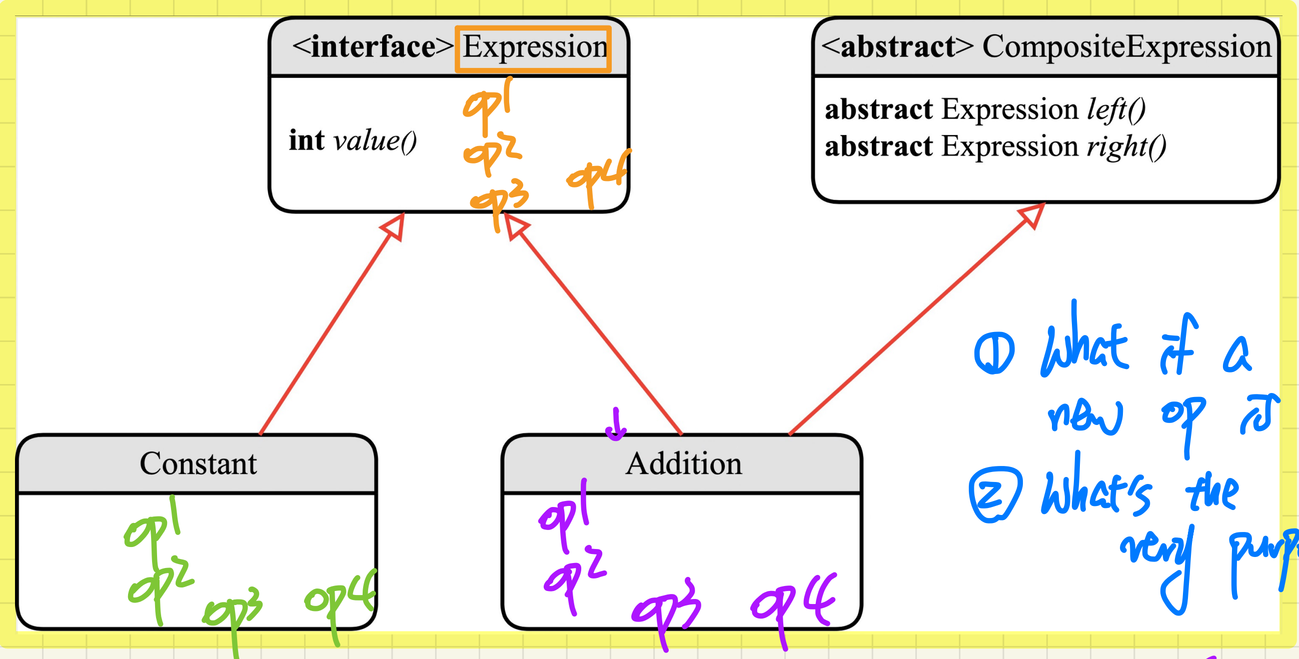


<interface> Expression
int value()
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
Addition

Q: How to construct a composite object representing "341 + 2"?

Q: How to extend the design to include variables and subtractions?

Design of Language Operation: How to Extend the Composite Pattern?



Structure

① What if a new op is needed?

② What's the very purpose of a class?

(superman class)

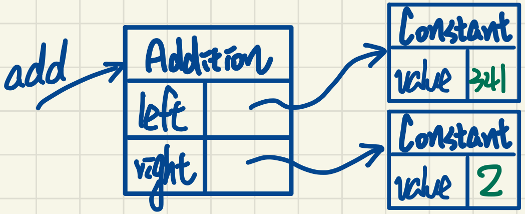
↕

modularity

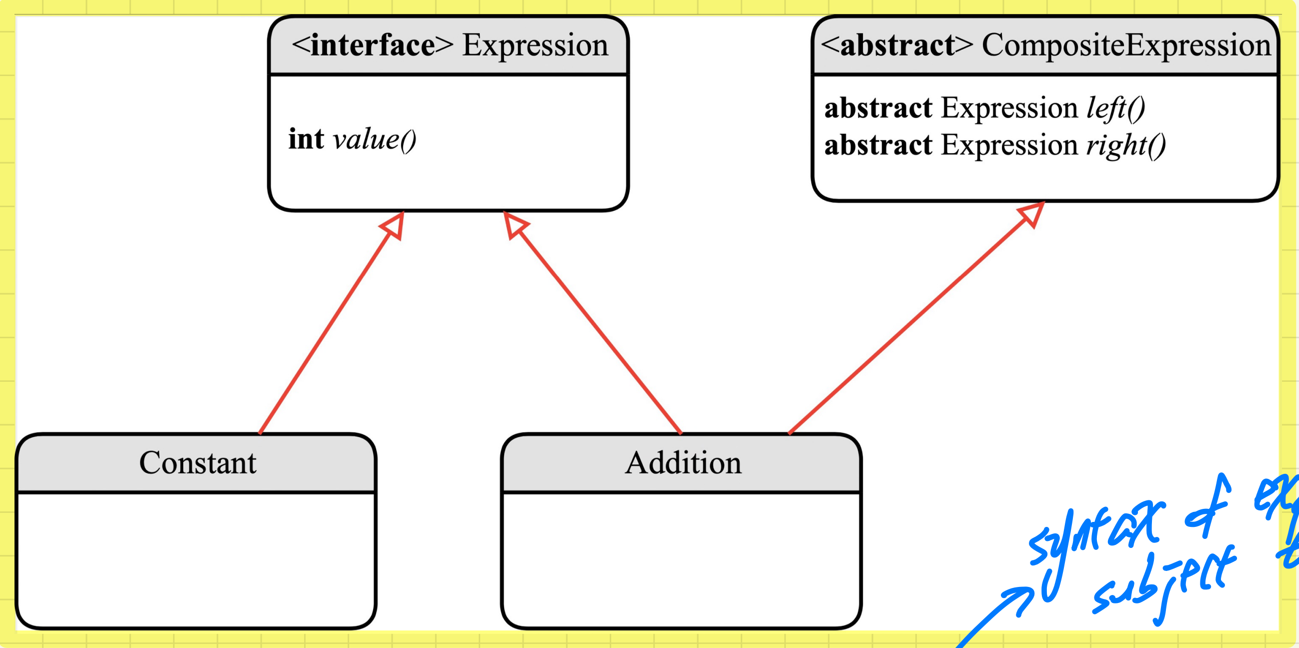
Operations

op1	evaluate
op2	print_prefix
op3	print_postfix
op4	type_check

op4



Design of a Language Application: Open-Closed Principle



<interface> Expression

int *value()*

<abstract> CompositeExpression

abstract Expression *left()*
abstract Expression *right()*

Constant

Addition

Structure

syntax of exp. subject to changes

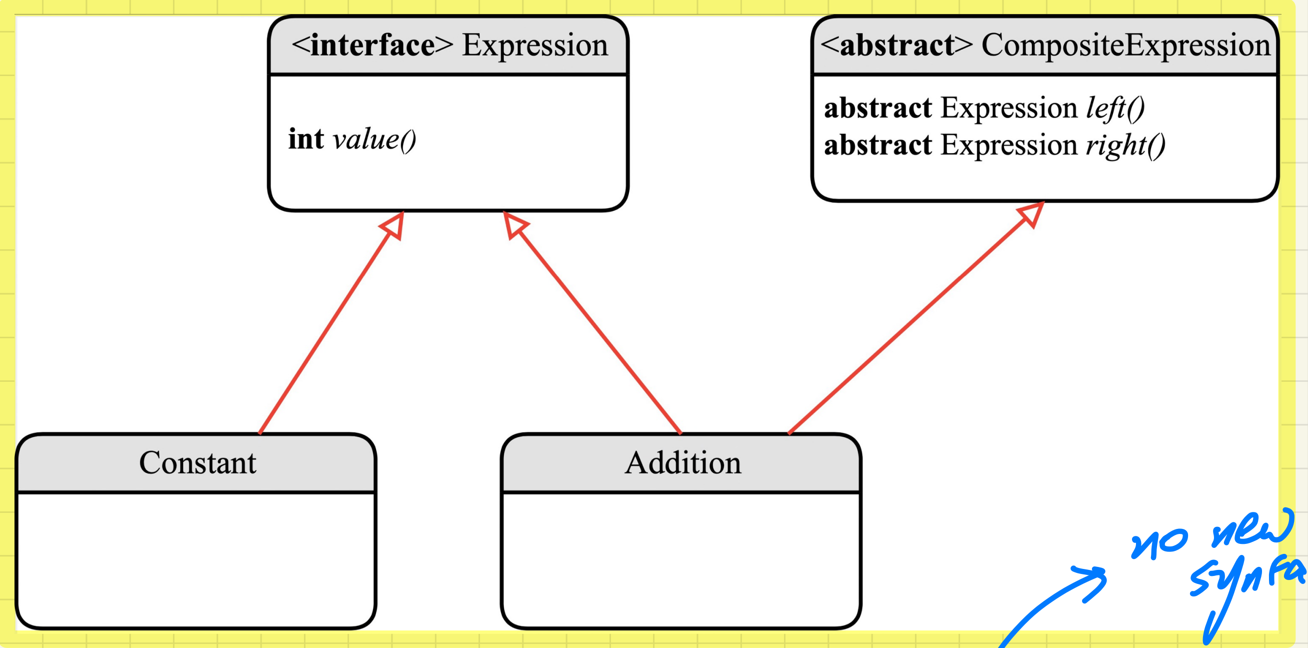
evaluate
print_prefix
print_postfix
type_check

Operations

	Structure	Operations
Alternative 1	Open	Closed
Alternative 2	Closed	Open

list of supported ops. is fixed

Design of a Language Application: Open-Closed Principle



<interface> Expression
int value()
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
Addition

Structure

evaluate
print_prefix
print_postfix
type_check

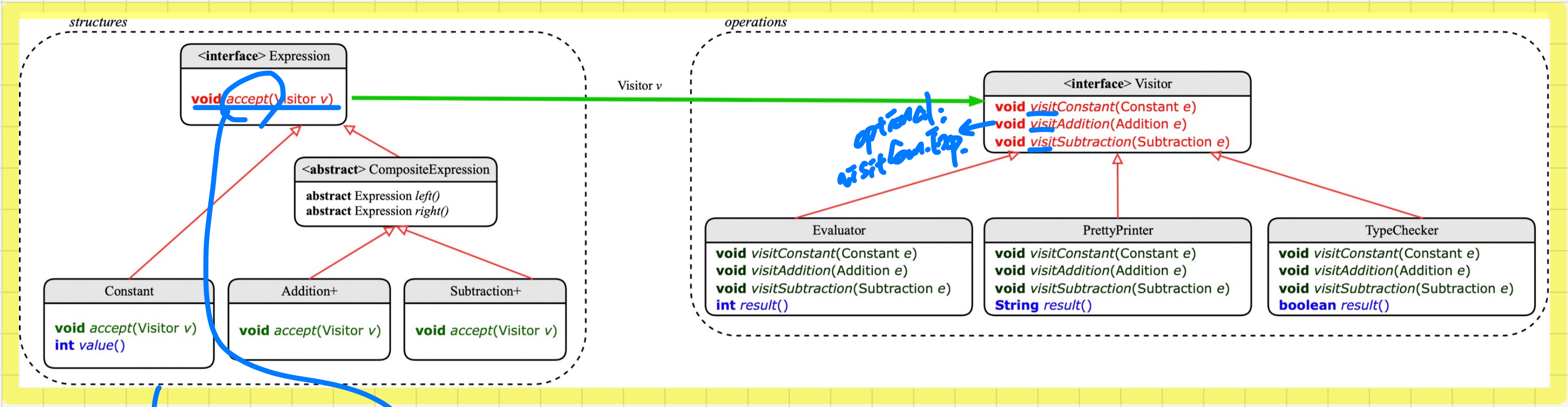
Operations

	Structure	Operations
Alternative 1	Open	Closed
Alternative 2	Closed	Open

no new syntax

keep adding new ops.

Visitor Design Pattern: Architecture



structures

<interface> Expression

void *accept*(Visitor *v*)

<abstract> CompositeExpression

abstract Expression *left()*
abstract Expression *right()*

Constant

void *accept*(Visitor *v*)
int *value*()

Addition+

void *accept*(Visitor *v*)

Subtraction+

void *accept*(Visitor *v*)

Visitor *v*

operations

<interface> Visitor

void *visitConstant*(Constant e)
void *visitAddition*(Addition e)
void *visitSubtraction*(Subtraction e)

optional: visitCom.Exp.

Evaluator

void *visitConstant*(Constant e)
void *visitAddition*(Addition e)
void *visitSubtraction*(Subtraction e)
int *result*()

PrettyPrinter

void *visitConstant*(Constant e)
void *visitAddition*(Addition e)
void *visitSubtraction*(Subtraction e)
String *result*()

TypeChecker

void *visitConstant*(Constant e)
void *visitAddition*(Addition e)
void *visitSubtraction*(Subtraction e)
boolean *result*()

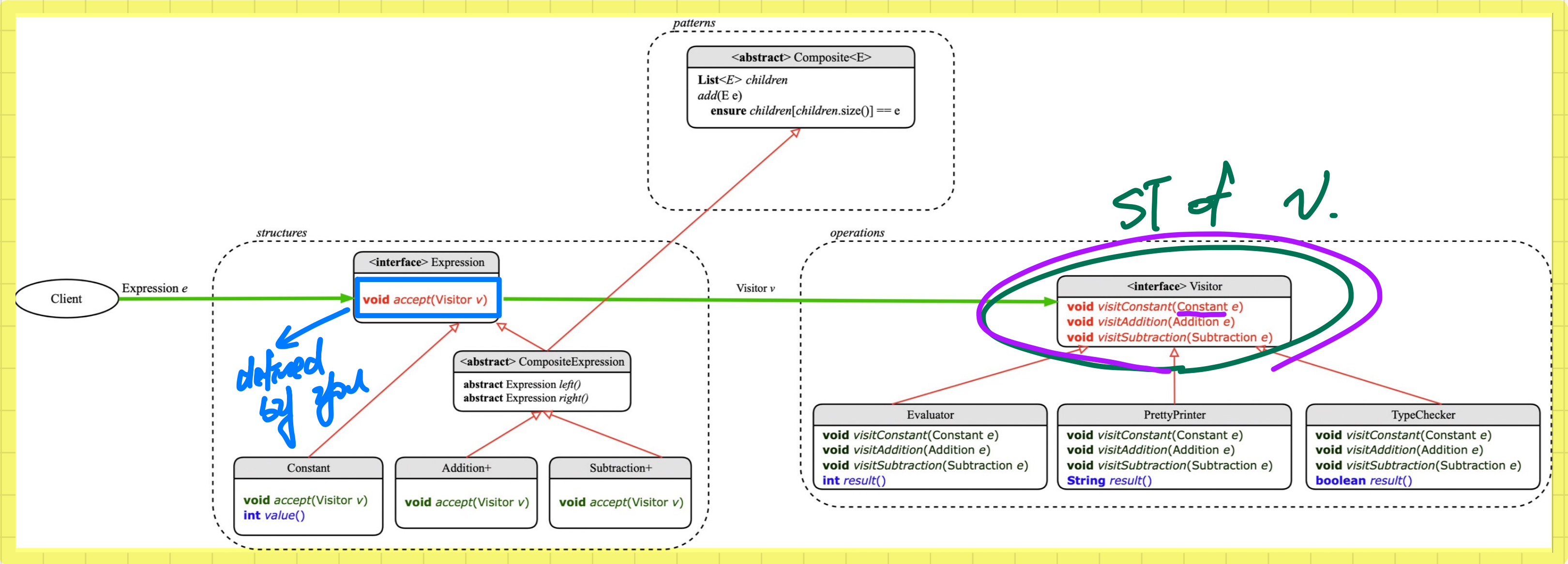
Composite.

DT.

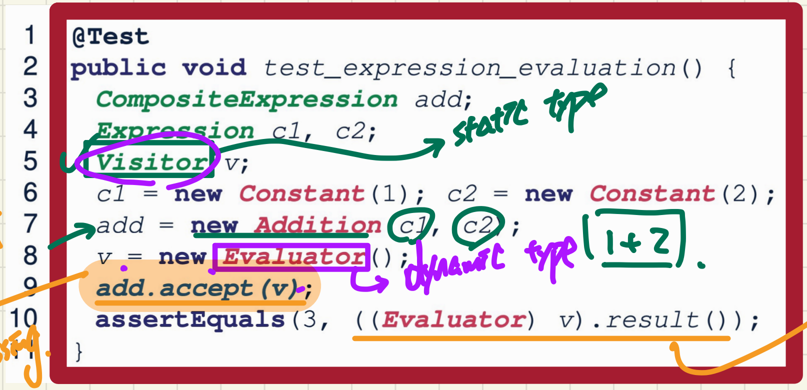
Expression e = ______

e.accept(visitor)

Visitor Design Pattern: Architecture


patterns
<abstract> Composite<E>
List<E> children
add(E e)
ensure children[children.size()] == e
structures
Client
Expression e
<interface> Expression
void accept(Visitor v)
defined by you
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
void accept(Visitor v)
int value()
Addition+
void accept(Visitor v)
Subtraction+
void accept(Visitor v)
Visitor v
operations
ST of v.
<interface> Visitor
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
Evaluator
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
int result()
PrettyPrinter
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
String result()
TypeChecker
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
boolean result()

How to Use Visitors

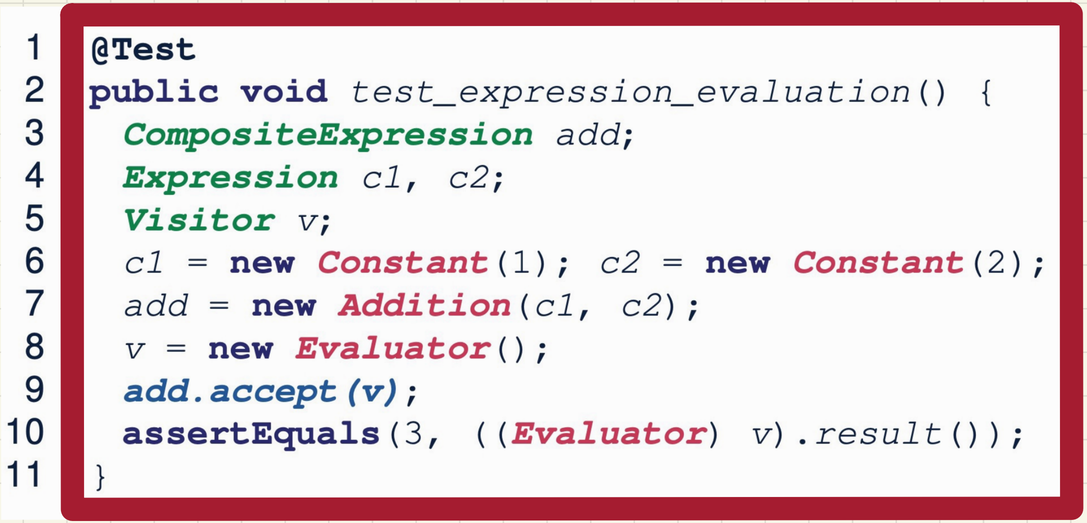


```
@Test
public void test_expression_evaluation() {
  CompositeExpression add;
  Expression c1, c2;
  Visitor v;
  c1 = new Constant(1); c2 = new Constant(2);
  add = new Addition(c1, c2);
  v = new Evaluator();
  add.accept(v);
  assertEquals(3, ((Evaluator) v).result());
}
```

Can I write v.result()?

↳ ST of v is Visitors which does not support result()

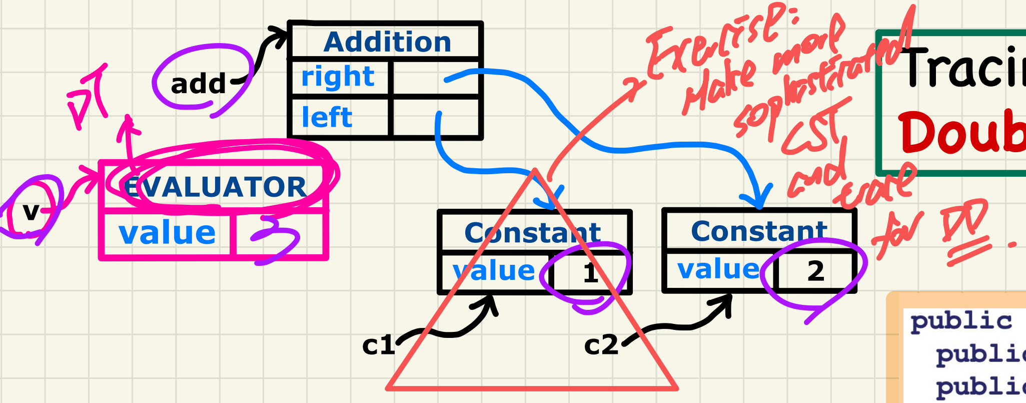
Visitor Design Pattern: Implementation



```
1  @Test
2  public void test_expression_evaluation() {
3    CompositeExpression add;
4    Expression c1, c2;
5    Visitor v;
6    c1 = new Constant(1); c2 = new Constant(2);
7    add = new Addition(c1, c2);
8    v = new Evaluator();
9    add.accept(v);
10   assertEquals(3, ((Evaluator) v).result());
11 }
```

Visualizing Line 3 to Line 7

Executing Composite and Visitor Patterns at Runtime



Exercise: Make more sophisticated CST and trace for DD.

Tracing add.accept(v)

Double Dispatch

DT

DT

1st dispatch: DT of add is Addition
↳ version of accept in Addition is invoked.

```
public interface Visitor {
  public void visitConstant(Constant e);
  public void visitAddition(Addition e);
  public void visitSubtraction(Subtraction e);
}
```

```
public class Constant implements Expression {
  ...
  public void accept(Visitor v) {
    v.visitConstant(this);
  }
}
```

↳ Evaluator version

```
public class Addition extends CompositeExpression {
  ...
  public void accept(Visitor v) {
    v.visitAddition(this);
  }
}
```

add

2nd dispatch:
DT of v is Evaluator
↳ version of visitAddition in Evaluator is invoked

```
public class Evaluator implements Visitor {
  private int result;
  ...
  public void visitConstant(Constant e) {
    this.result = e.value();
  }
  public void visitAddition(Addition e) {
    Evaluator evalL = new Evaluator();
    Evaluator evalR = new Evaluator();
    e.getLeft().accept(evalL);
    e.getRight().accept(evalR);
    this.result = evalL.result() + evalR.result();
  }
}
```

DT: Constant

add e → +
L R

→ double-dispatch

→ double-dispatch

③ ① ②

Lecture 14 - Nov. 1

Visitor, Syntactic Analysis

Visitor: Double Dispatch

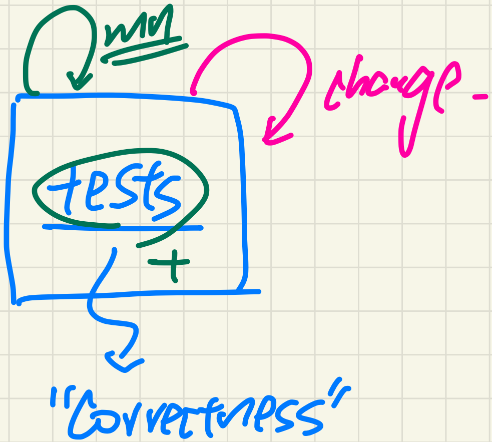
Visitor: Open-Closed Principle

Visitor: Single-Choice Principle

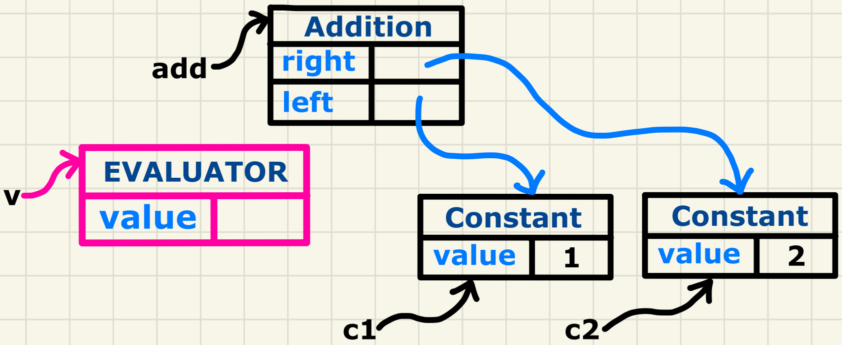
Announcements

- **Assignment 2** released
  + Python script for Regression Testing

- **Project Milestone 1** due next week
  + Sign-Up sheet activated tomorrow (Wednesday) at 6pm



Executing Composite and Visitor Patterns at Runtime



Tracing add.accept(v) → DT → DT

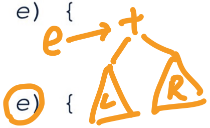
Double Dispatch

```
public interface Visitor {
  public void visitConstant(Constant e);
  public void visitAddition(Addition e);
  public void visitSubtraction(Subtraction e);
}
```

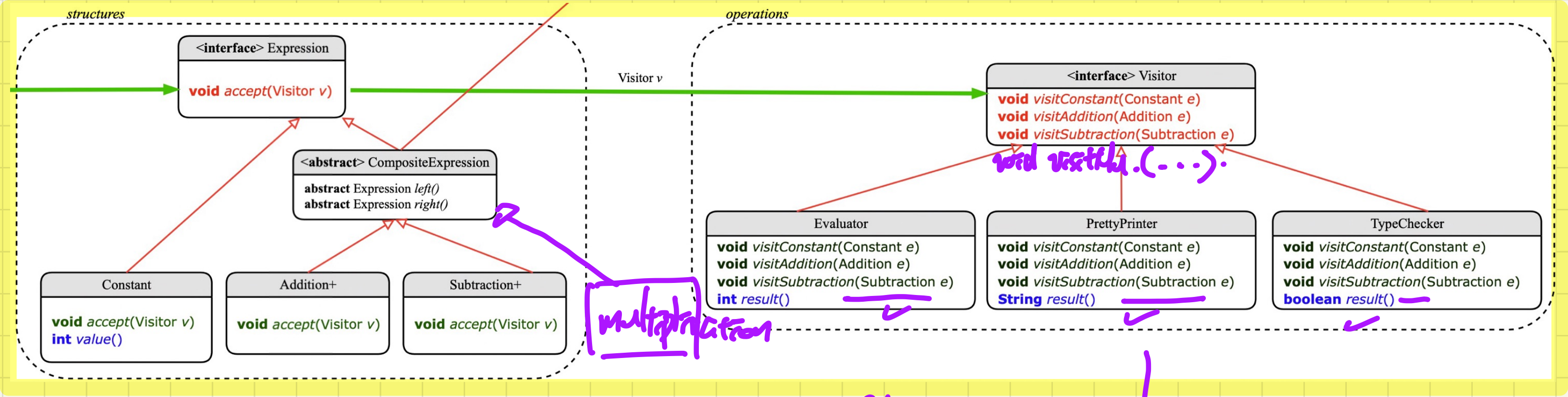
```
public class Constant implements Expression {
  ...
  public void accept(Visitor v) {
    v.visitConstant(this);
  }
}
```

```
public class Addition extends CompositeExpression {
  ...
  public void accept(Visitor v) {
    v.visitAddition(this);
  }
}
```

```
public class Evaluator implements Visitor {
  private int result;
  ...
  public void visitConstant(Constant e) {
    this.result = e.value();
  }
  public void visitAddition(Addition e) {
    Evaluator evalL = new Evaluator();
    Evaluator evalR = new Evaluator();
    e.getLeft().accept(evalL);
    e.getRight().accept(evalR);
    this.result = evalL.result() + evalR.result();
  }
}
```



Visitor Pattern: Open-Closed and Single-Choice Principles



structures
<interface> Expression
void accept(Visitor v)
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
void accept(Visitor v)
int value()
Addition+
void accept(Visitor v)
Subtraction+
void accept(Visitor v)
multiplication
Visitor v
operations
<interface> Visitor
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
void visitMul(...);
Evaluator
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
int result()
PrettyPrinter
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
String result()
TypeChecker
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
boolean result()

↳ violates SCP.

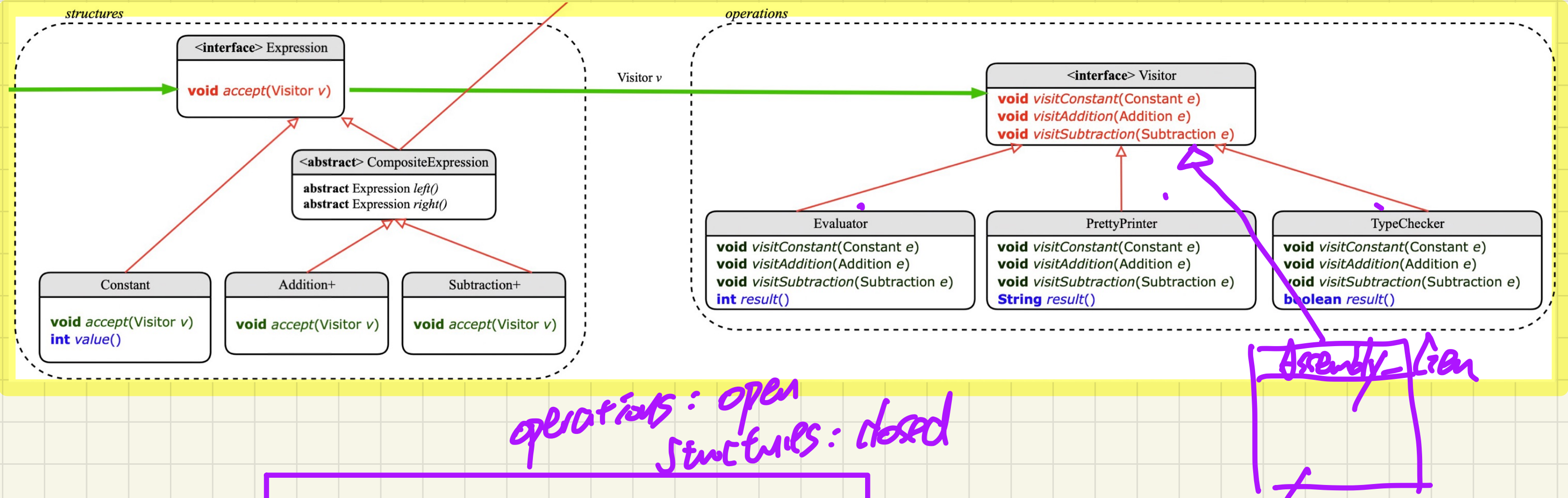
structure: open
operation: closed

What if a new language construct is added?

↳ unsuitable for visitor pattern

If the visitor pattern is adopted, what should be closed?

Visitor Pattern: Open-Closed and Single-Choice Principles


structures
<interface> Expression
void accept(Visitor v)
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
void accept(Visitor v)
int value()
Addition+
void accept(Visitor v)
Subtraction+
void accept(Visitor v)
Visitor v
operations
<interface> Visitor
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
Evaluator
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
int result()
PrettyPrinter
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
String result()
TypeChecker
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
boolean result()
Assembly_Gen

operations : open

structures : closed

What if a new language operation is added?

In this single place, implement all visit methods

↳ SCP satisfied

If the visitor pattern is adopted, what should be open?

Lecture 15 - Nov. 3

Syntactic Analysis

Identifying Derivations: TDP vs. BUP
Top-Down Parsing: Algorithm
Left-Recursive CFG

Announcements

- Assignment 2 released

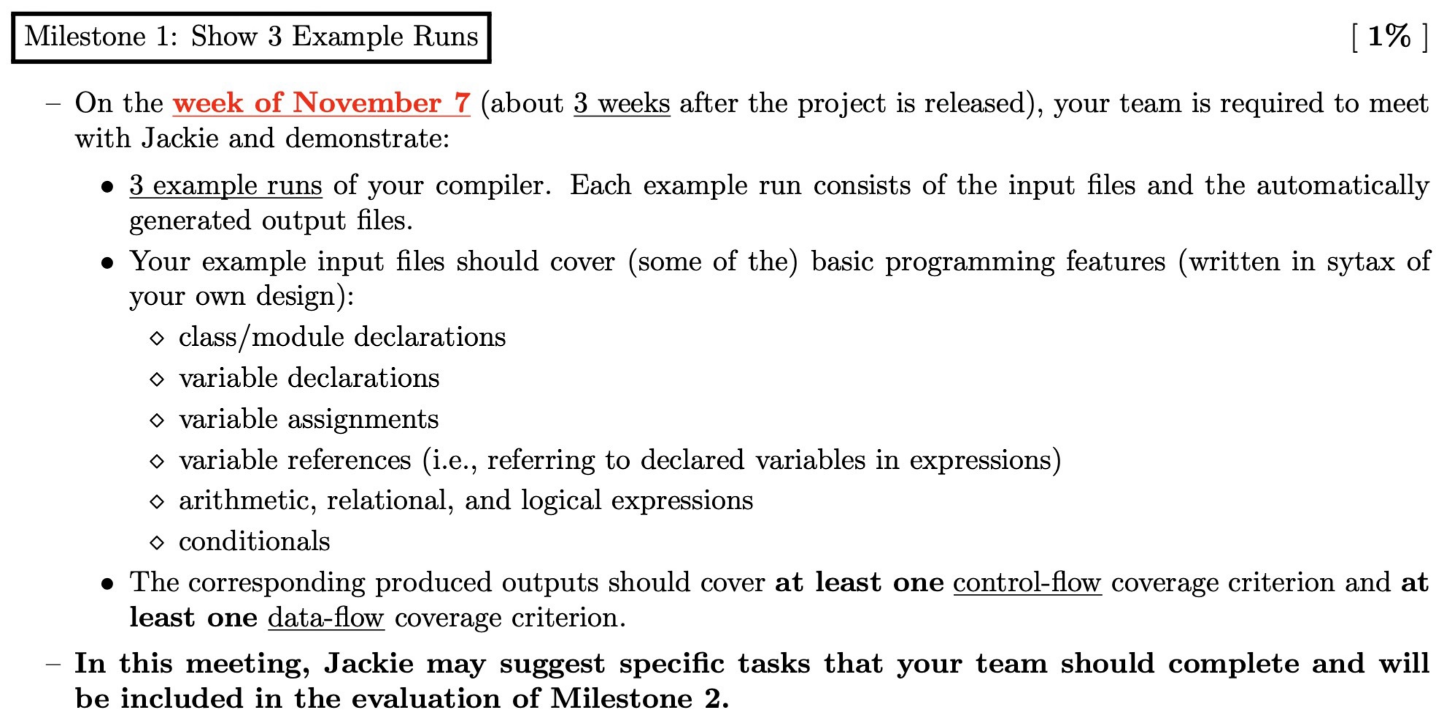
- Project Milestone 1 next week
  - Source project due at 11:59 PM on Tuesday
  - A simple readme.txt file explains how to run your tool: e.g.,

    ```
    java -jar compiler.jar prog.txt test.txt
    ```

    (and where to find the output HTML file)
  - Example files you supplied are supposed to work automatically
  - Jackie will share his screen to build, run, and explore your code.

- Visitor Pattern source code: Type Casting

Project: Milestone 1



Milestone 1: Show 3 Example Runs [**1%**]

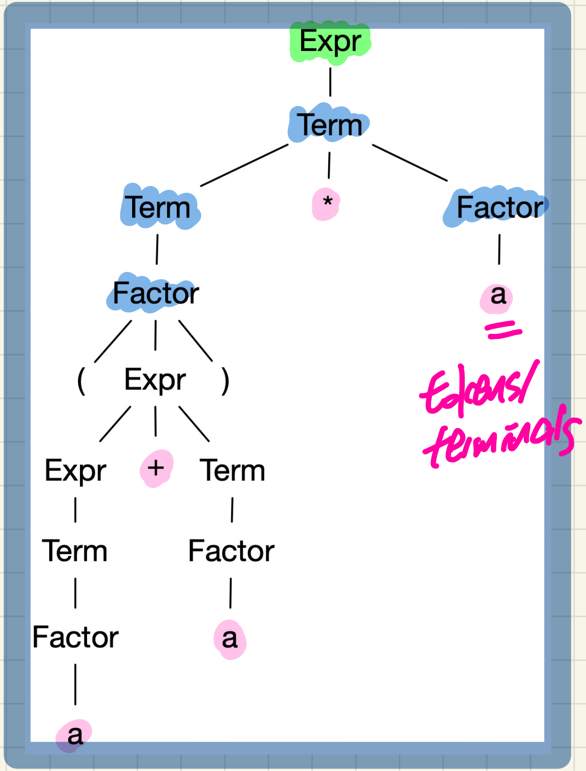
- On the **week of November 7** (about 3 weeks after the project is released), your team is required to meet with Jackie and demonstrate:
  - 3 example runs of your compiler. Each example run consists of the input files and the automatically generated output files.
  - Your example input files should cover (some of the) basic programming features (written in sytax of your own design):
    - ◇ class/module declarations
    - ◇ variable declarations
    - ◇ variable assignments
    - ◇ variable references (i.e., referring to declared variables in expressions)
    - ◇ arithmetic, relational, and logical expressions
    - ◇ conditionals
  - The corresponding produced outputs should cover **at least one** control-flow coverage criterion and **at least one** data-flow coverage criterion.
- **In this meeting, Jackie may suggest specific tasks that your team should complete and will be included in the evaluation of Milestone 2.**

Discovering Derivations

Input Grammar G

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

AST: (a + a) * a

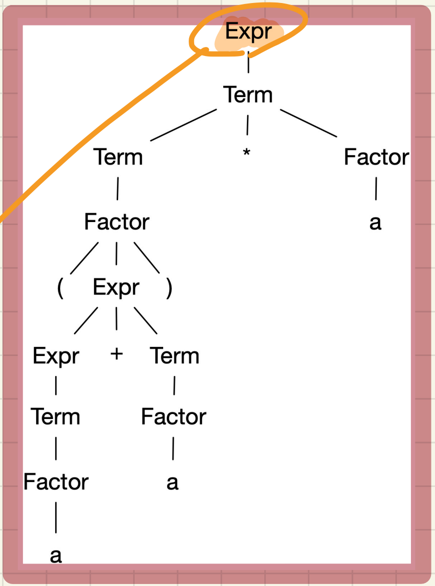


Discovering Derivations: Top-Down vs. Bottom-Up

Input Grammar G

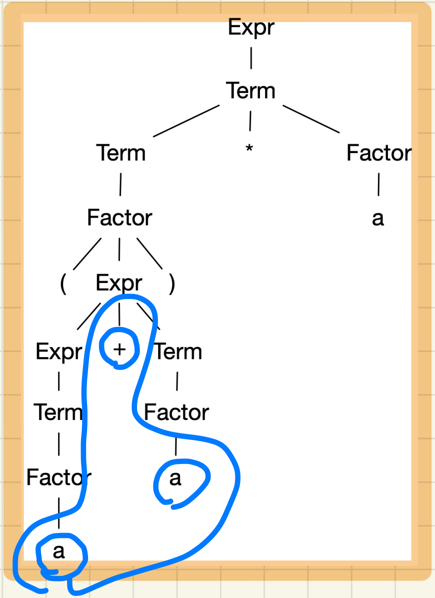
Expr	→	*Expr* + *Term*
	|	*Term*
Term	→	*Term* * *Factor*
	|	*Factor*
Factor	→	(*Expr*)
	|	a

TDP: (a + a) * a



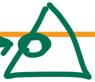
possibility of backtrack.

BUP: (a + a) * a



Top-Down Parsing: Algorithm

```
ALGORITHM: TDParse
  INPUT: CFG $G = (V, \Sigma, R, S)$ string.
  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 $focus \in V$ then
      if $\exists$ unvisited rule $focus \rightarrow \beta_1 \beta_2 \ldots \beta_n \in R$ then
        create $\beta_1, \beta_2 \ldots \beta_n$ as children of focus
        trace.push($\beta_n \beta_{n-1} \ldots \beta_2$)
        focus := $\beta_1$
      else
        if focus = $S$ then report syntax error
        else backtrack
✓   elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
✓   elseif word = EOF $\wedge$ focus = null then return root
    else backtrack
```



success

transforming a linear input token seq. Expr into a non-linear AST.

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

Input Grammar G

$Expr \rightarrow Expr + Term$

$\quad\mid Term$

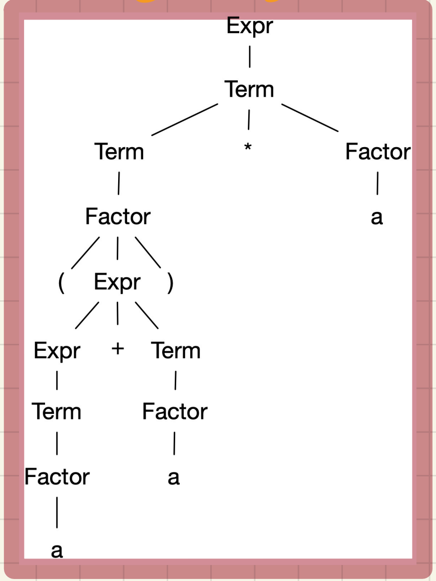
$Term \rightarrow Term * Factor$

$\quad\mid Factor$

$Factor \rightarrow (Expr)$

$\quad\mid a$

TDP: (a + a) * a



Top-Down Parsing: Discovering Leftmost Derivations (1)

$Expr \rightarrow \underbrace{Expr}_{\beta_1}\ \underbrace{+}_{\beta_2}\ \underbrace{Term}_{\beta_3}$

```
ALGORITHM: TDParse
  INPUT: CFG $G = (V, \Sigma, R, S)$
  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 $\in V$ then
      if $\exists$ unvisited rule focus $\rightarrow \beta_1 \beta_2 \ldots \beta_n \in R$ then
        create $\beta_1, \beta_2 \ldots \beta_n$ as children of focus
        trace.push($\beta_n \beta_{n-1} \ldots \beta_2$)
        focus := $\beta_1$
      else
        if focus = $S$ then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF $\wedge$ focus = null then return root
    else backtrack
```

β_1 missing

attempts to find LMD

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

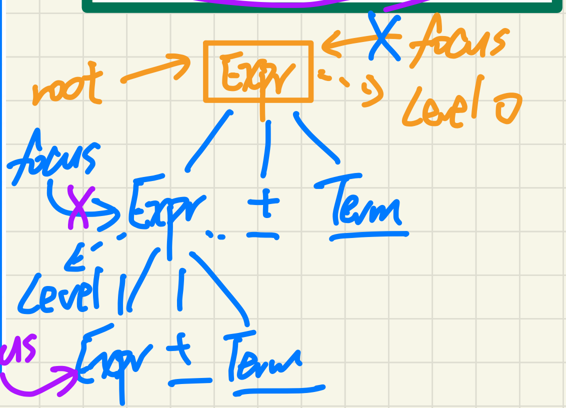
word: "a"

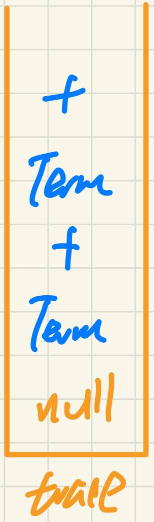
non-terminating.

Parse: a + a * a

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

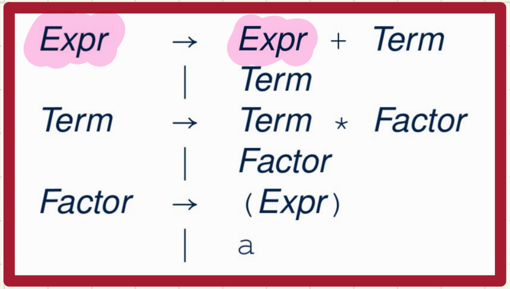
left-recursive



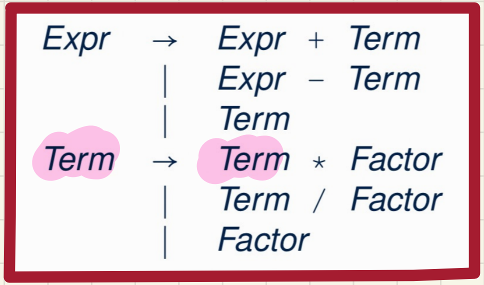


Left-Recursions (LRs): Direct vs. Indirect

Direct Left-Recursions:

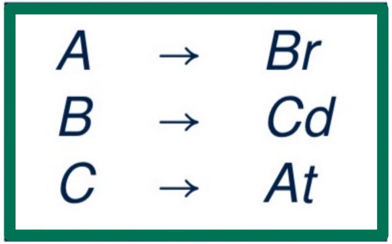


$$\begin{array}{lcll}
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \star \mathit{Factor} \\
 & | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & (\,\mathit{Expr}\,) \\
 & | & \mathtt{a}
\end{array}$$

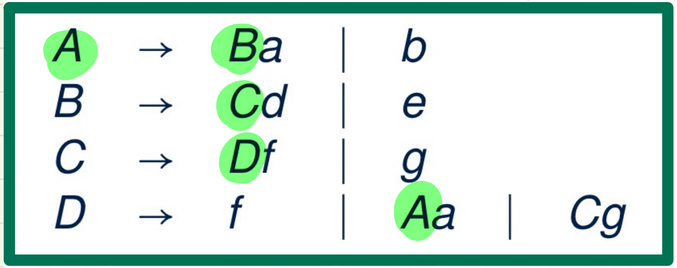


$$\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & | & \mathit{Expr} - \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \star \mathit{Factor} \\
 & | & \mathit{Term}\ /\ \mathit{Factor} \\
 & | & \mathit{Factor}
\end{array}$$

Indirect Left-Recursions:



$$\begin{array}{lcl}
A & \rightarrow & Br \\
B & \rightarrow & Cd \\
C & \rightarrow & At
\end{array}$$

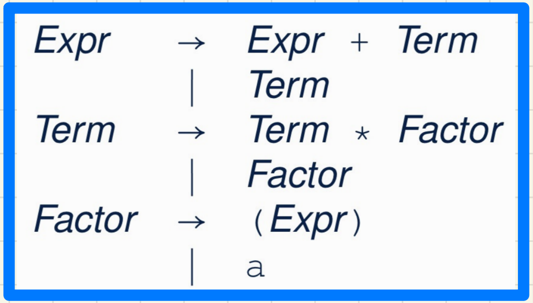


$$\begin{array}{lcllll}
A & \rightarrow & Ba & | & b & \\
B & \rightarrow & Cd & | & e & \\
C & \rightarrow & Df & | & g & \\
D & \rightarrow & f & | & Aa & | \quad Cg
\end{array}$$

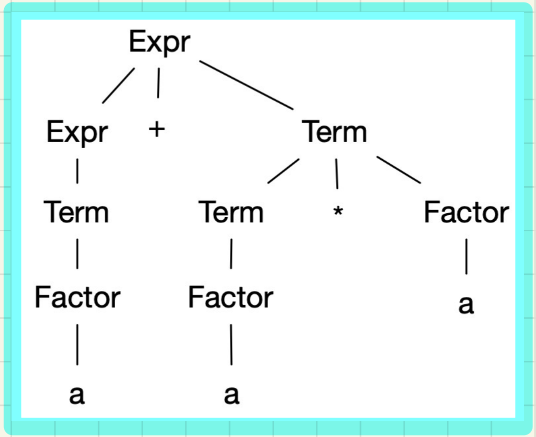
CFGs: Left-Recursive vs. Right-Recursive

Example: a + a * a

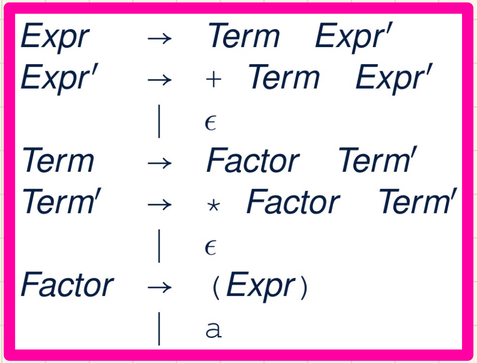
CFG with Left Recursions



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

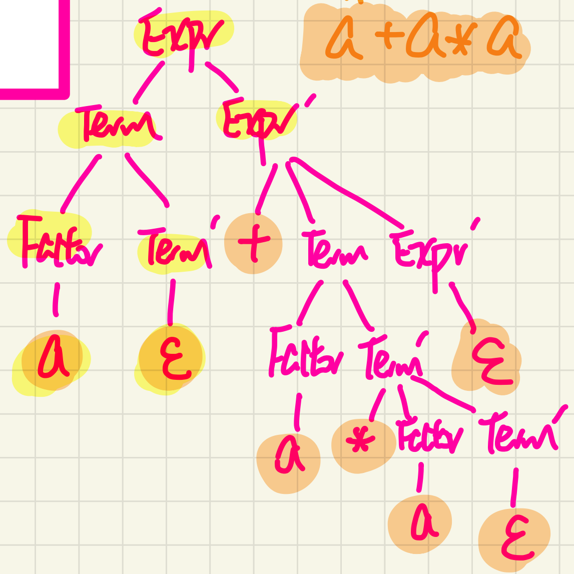


CFG with Right Recursions

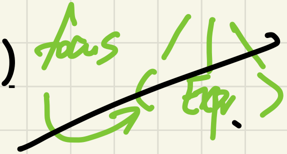


$$\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Term} \;\; \mathit{Expr}' \\
\mathit{Expr}' & \rightarrow & + \;\; \mathit{Term} \;\; \mathit{Expr}' \\
 & | & \epsilon \\
\mathit{Term} & \rightarrow & \mathit{Factor} \;\; \mathit{Term}' \\
\mathit{Term}' & \rightarrow & * \;\; \mathit{Factor}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

$a \, \varepsilon + a * a \, \varepsilon \, \varepsilon \equiv a + a * a$



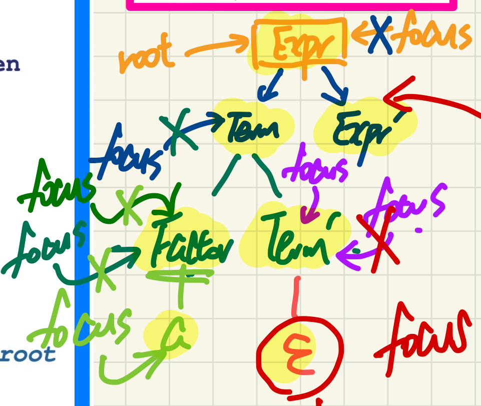
Top-Down Parsing: Discovering Leftmost Derivations (2)

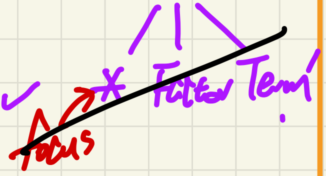


```
ALGORITHM: TDParse
  INPUT: CFG $G = (V, \Sigma, R, S)$
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → $\beta_1 \beta_2 \dots \beta_n$ ∈ R then
        create $\beta_1, \beta_2 \dots \beta_n$ as children of focus
        trace.push($\beta_n \beta_{n-1} \dots \beta_2$)
        focus := $\beta_1$
      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
```

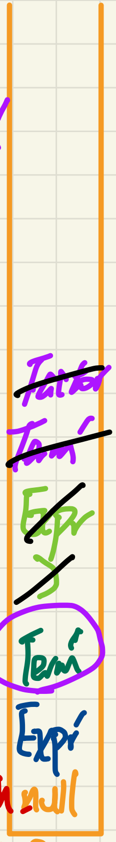
Parse: a + a * a

Expr	→	Term Expr' ✓
Expr'	→	+ Term Expr'
	|	ϵ
Term	→	Factor Term' ✓
Term'	→	⋆ Factor Term' ✓
	|	ϵ
Factor	→	(Expr) ✓
	|	a ✓





Rest: exercise



doesn't match "+" but ε has no affect on concat ⇒ the next symbol may match

trace

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

word: "~~a~~" "+"

Top-Down Parsing: Discovering Leftmost Derivations (3)

```
ALGORITHM: TDParse
  INPUT: CFG $G = (V, \Sigma, R, S)$
  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 $\in V$ then
      if $\exists$ unvisited rule focus $\rightarrow \beta_1 \beta_2 \dots \beta_n \in R$ then
        create $\beta_1, \beta_2 \dots \beta_n$ as children of focus
        trace.push($\beta_n \beta_{n-1} \dots \beta_2$)
        focus := $\beta_1$
      else
        if focus = $S$ then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF $\wedge$ focus = null then return root
    else backtrack
```

Parse: (a + a) * a

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

Exercise!

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

Expr

$\Rightarrow \text{Term} \ \underline{\text{Expr}'}$

$\Rightarrow \underline{\text{Term}} \ \varepsilon$

$\Rightarrow \text{Factor} \ \underline{\text{Term}'}$

$\Rightarrow \text{Factor} * \text{Factor} \ \underline{\text{Term}'}$

$\Rightarrow \text{Factor} * \underline{\text{Factor}} \ \varepsilon$

$\Rightarrow \underline{\text{Factor}} * a$

$\Rightarrow (\underline{\text{Expr}}) * a$

$\Rightarrow (\text{Term} \ \underline{\text{Expr}'}) * a$

$(\text{Term} + \text{Term} \ \underline{\text{Expr}'}) * a$

$\Rightarrow (\text{Term} + \text{Term} \ \varepsilon) * a$

$\Rightarrow (\text{Term} + \text{Factor} \ \underline{\text{Term}'} \ \varepsilon) * a$

$\Rightarrow (\text{Term} + \underline{\text{Factor}} \ \varepsilon \ \varepsilon) * a$

$\Rightarrow (\text{Term} + a \ \varepsilon \ \varepsilon) * a$

$\Rightarrow (\text{Factor} \ \underline{\text{Term}'} + a \ \varepsilon \ \varepsilon) * a$

$\Rightarrow (\text{Factor} \ \varepsilon + a \ \varepsilon \ \varepsilon) * a$

$\Rightarrow (a \ \varepsilon + a \ \varepsilon \ \varepsilon) * a$

Lecture 16 - Nov. 10

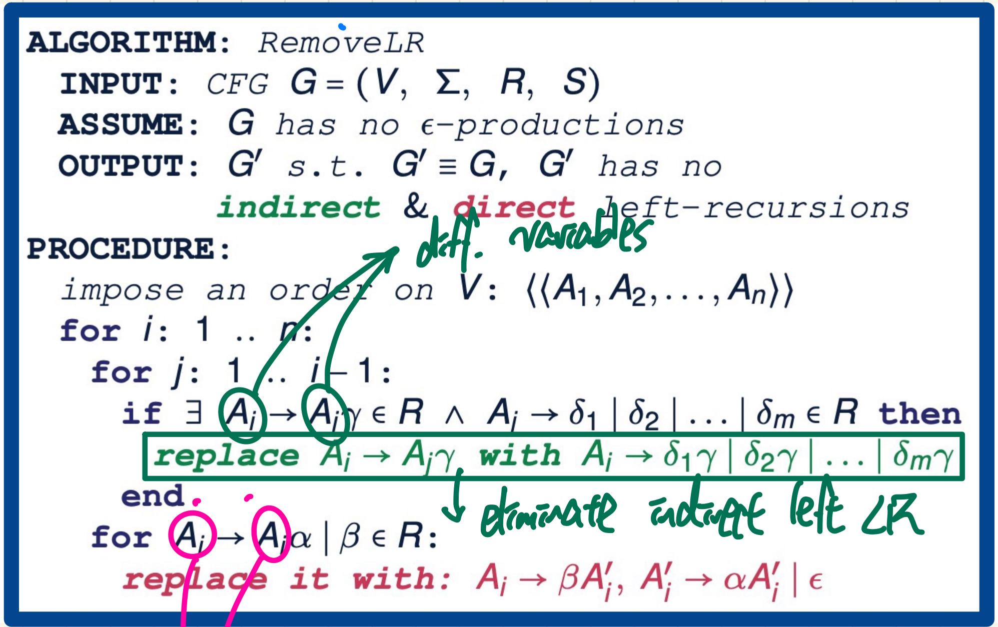
Syntactic Analysis

Removing Left-Recursion from CFG
Computing the FIRST Set

Announcements

- **ProgTest** marks and results released
- **Assignment 2** due next Monday
- **Quiz2** and **Quiz3** papers ready for pick-up on Monday

Removing Left-Recursions: Algorithm



```
ALGORITHM: RemoveLR
  INPUT: CFG $G = (V, \Sigma, R, S)$
  ASSUME: $G$ has no $\epsilon$-productions
  OUTPUT: $G'$ s.t. $G' \equiv G$, $G'$ has no
          indirect & direct left-recursions
PROCEDURE:
  impose an order on $V$: $\langle\langle A_1, A_2, \ldots, A_n \rangle\rangle$
  for $i$: 1 .. $n$:
    for $j$: 1 .. $i-1$:
      if $\exists\ A_i \to A_j\gamma \in R \wedge A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_m \in R$ then
        replace $A_i \to A_j\gamma$ with $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_m\gamma$
      end
    for $A_i \to A_i\alpha \mid \beta \in R$:
      replace it with: $A_i \to \beta A_i'$, $A_i' \to \alpha A_i' \mid \epsilon$
```

diff. variables

↓ eliminate indirect left LR

same variable ⇒ direct left recursion

$A \to \varepsilon$

$\boxed{\varepsilon\text{-production}}$

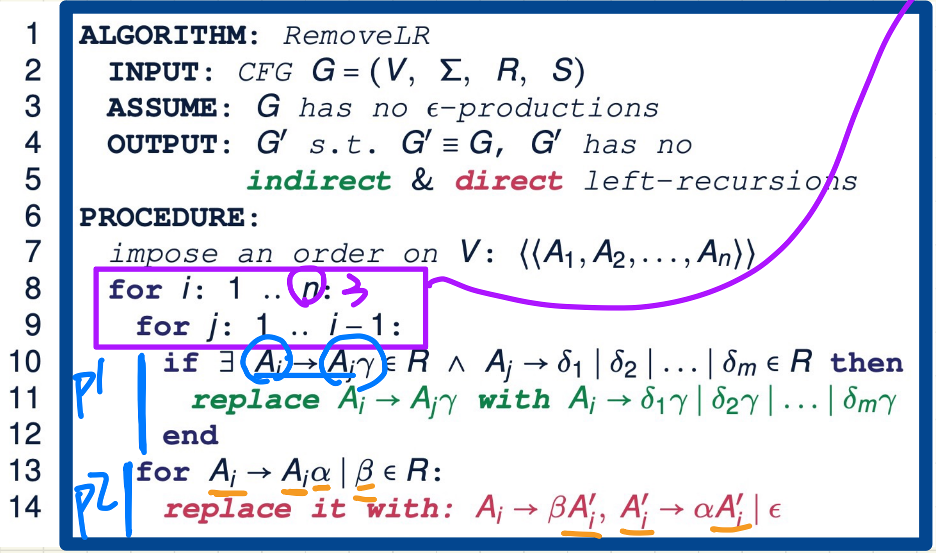
$A_i \to A_i\alpha \mid \beta$ left-recursion

① $A_i \Rightarrow \beta$

② $A_i \Rightarrow A_i\alpha \Rightarrow A_i\alpha\alpha \Rightarrow \beta\alpha\alpha$

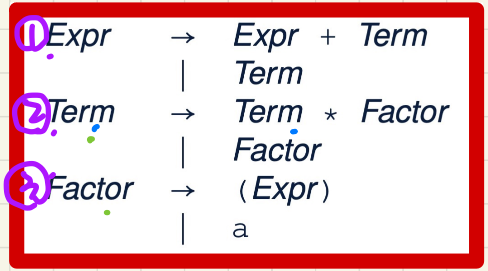
right-recursive
$A_i \to \beta A_i'$
$A_i' \to \alpha A_i' \mid \varepsilon$

Removing Left-Recursions (1a)



```
ALGORITHM: RemoveLR
  INPUT: CFG $G = (V, \Sigma, R, S)$
  ASSUME: $G$ has no $\epsilon$-productions
  OUTPUT: $G'$ s.t. $G' \equiv G$, $G'$ has no
          indirect & direct left-recursions
PROCEDURE:
  impose an order on $V$: $\langle\langle A_1, A_2, \dots, A_n \rangle\rangle$
  for $i$: 1 .. $n$: 3
    for $j$: 1 .. $i-1$:
P1    if $\exists\, A_i \rightarrow A_j\gamma \in R \;\wedge\; A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_m \in R$ then
        replace $A_i \rightarrow A_j\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_m\gamma$
      end
P2  for $A_i \rightarrow A_i\alpha \mid \beta \in R$:
      replace it with: $A_i \rightarrow \beta A'_i$, $A'_i \rightarrow \alpha A'_i \mid \epsilon$
```

Directly Left-Recursive CFG:



① Expr	→	Expr	+	Term
	|	Term		
② Term	→	Term	*	Factor
	|	Factor		
③ Factor	→	(Expr)		
	|	a		

Term → Factor Term'

Term' → * Factor Term'

| ε

i	j	
① Expr	—	✗
② Term	① Expr	✗
③ Factor	① → ②	
	Expr Term	

[i, j = 2, 1] $A_i \rightarrow A_j$

(P1) Term → Expr ? no

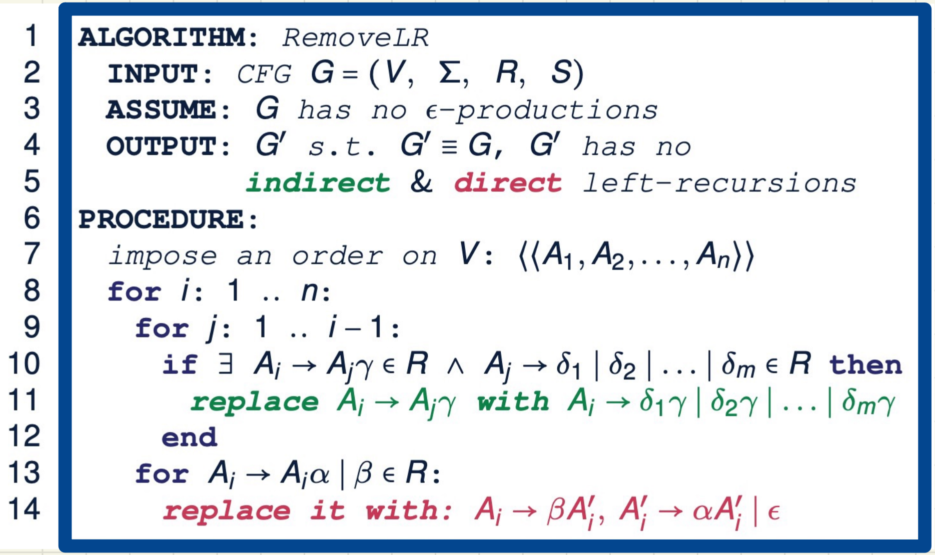
↳ do nothing.

(P2) A_i → A_i α ? Yes

Term → Term * Factor ?

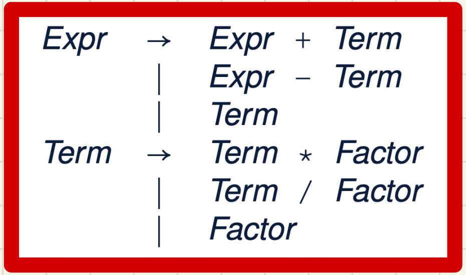
| Factor β

Removing Left-Recursions (1b)



```
ALGORITHM: RemoveLR
  INPUT: CFG $G = (V, \Sigma, R, S)$
  ASSUME: $G$ has no $\epsilon$-productions
  OUTPUT: $G'$ s.t. $G' \equiv G$, $G'$ has no
          indirect & direct left-recursions
PROCEDURE:
  impose an order on $V$: $\langle\langle A_1, A_2, \ldots, A_n \rangle\rangle$
  for $i$: 1 .. $n$:
    for $j$: 1 .. $i-1$:
      if $\exists\ A_i \rightarrow A_j\gamma \in R \ \wedge\ A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_m \in R$ then
        replace $A_i \rightarrow A_j\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_m\gamma$
      end
    for $A_i \rightarrow A_i\alpha \mid \beta \in R$:
      replace it with: $A_i \rightarrow \beta A_i',\ A_i' \rightarrow \alpha A_i' \mid \epsilon$
```

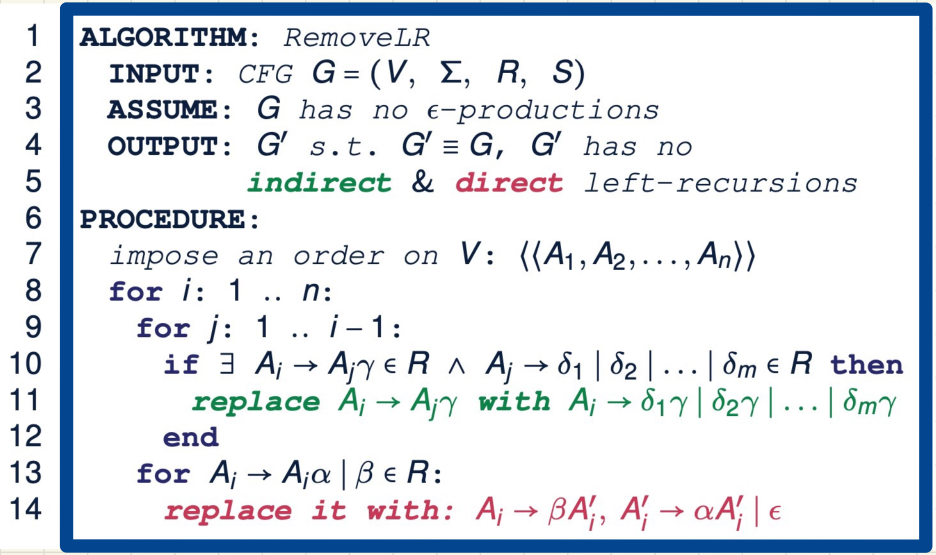
Directly Left-Recursive CFG:



$$\begin{array}{lclcl}
Expr & \rightarrow & Expr & + & Term \\
 & | & Expr & - & Term \\
 & | & Term & & \\
Term & \rightarrow & Term & * & Factor \\
 & | & Term & / & Factor \\
 & | & Factor & &
\end{array}$$

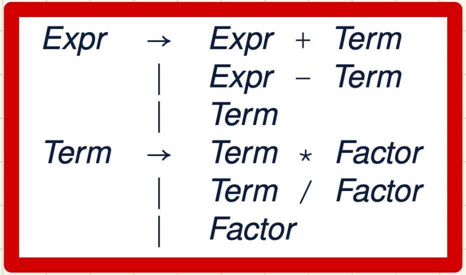
Exercise

Removing Left-Recursions (1b)



```
ALGORITHM: RemoveLR
  INPUT: CFG $G = (V, \Sigma, R, S)$
  ASSUME: $G$ has no $\epsilon$-productions
  OUTPUT: $G'$ s.t. $G' \equiv G$, $G'$ has no
          indirect & direct left-recursions
PROCEDURE:
  impose an order on $V$: $\langle\langle A_1, A_2, \ldots, A_n \rangle\rangle$
  for $i$: 1 .. $n$:
    for $j$: 1 .. $i-1$:
      if $\exists\ A_i \to A_j\gamma \in R\ \wedge\ A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_m \in R$ then
        replace $A_i \to A_j\gamma$ with $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_m\gamma$
      end
    for $A_i \to A_i\alpha \mid \beta \in R$:
      replace it with: $A_i \to \beta A_i',\ A_i' \to \alpha A_i' \mid \epsilon$
```

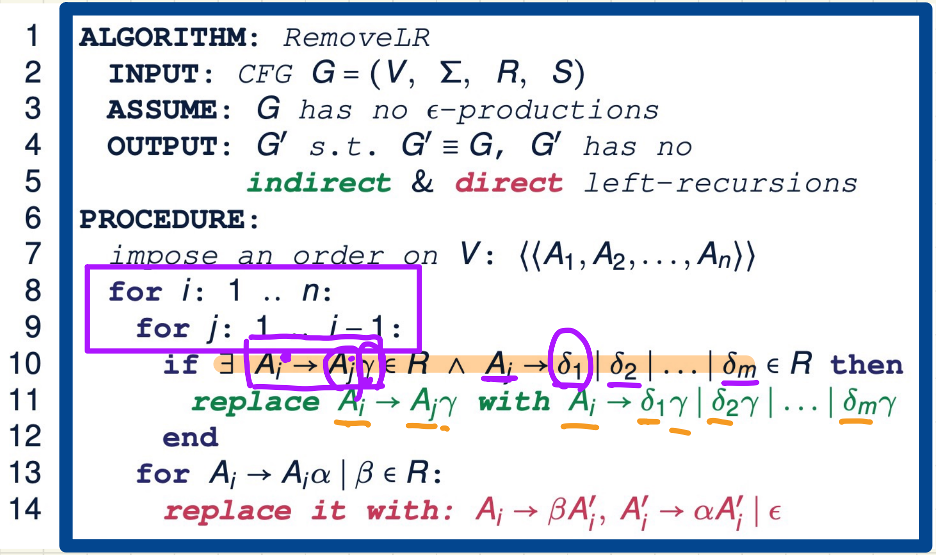
Directly Left-Recursive CFG:



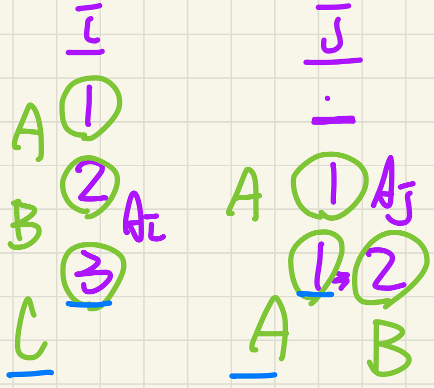
$$
\begin{array}{lcl}
Expr & \to & Expr + Term \\
 & \mid & Expr - Term \\
 & \mid & Term \\
Term & \to & Term * Factor \\
 & \mid & Term\ /\ Factor \\
 & \mid & Factor
\end{array}
$$

$$
\begin{array}{lcl}
Expr & \to & Term\ Expr' \\
Expr' & \to & +\ Term\ Expr' \\
 & \mid & -\ Term\ Expr' \\
 & \mid & \varepsilon \\
Term & \to & Factor\ Term' \\
Term' & \to & *\ Factor\ Term' \\
 & \mid & /\ Factor\ Term' \\
 & \mid & \varepsilon
\end{array}
$$

Removing Left-Recursions (2a)



```
ALGORITHM: RemoveLR
  INPUT: CFG $G = (V, \Sigma, R, S)$
  ASSUME: $G$ has no $\epsilon$-productions
  OUTPUT: $G'$ s.t. $G' \equiv G$, $G'$ has no
          indirect & direct left-recursions
PROCEDURE:
  impose an order on $V$: $\langle\langle A_1, A_2, \dots, A_n \rangle\rangle$
  for $i$: 1 .. $n$:
    for $j$: 1 .. $i-1$:
      if $\exists\ A_i \to A_j\gamma \in R \ \wedge\ A_j \to \delta_1 \mid \delta_2 \mid \dots \mid \delta_m \in R$ then
        replace $A_i \to A_j\gamma$ with $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_m\gamma$
      end
    for $A_i \to A_i\alpha \mid \beta \in R$:
      replace it with: $A_i \to \beta A_i'$, $A_i' \to \alpha A_i' \mid \epsilon$
```



$\boxed{I, J = 2, 1}$

$B \to A$? No

↳ do nothing.

$\boxed{I, J = 3, 1}$

$\underbrace{C}_{A_i} \to \underbrace{A}_{A_j}\underbrace{t}_{\gamma}$

$\underbrace{A}_{A_j} \to \underbrace{Br}_{\delta_1}$

$\to \underbrace{C}_{A_i} \to \underbrace{Br}_{\delta_1}\underbrace{t}_{\gamma}$

Indirectly Left-Recursive CFG:

① ~~$A \to Br$~~

② $B \to Cd$

③ $C \to At$

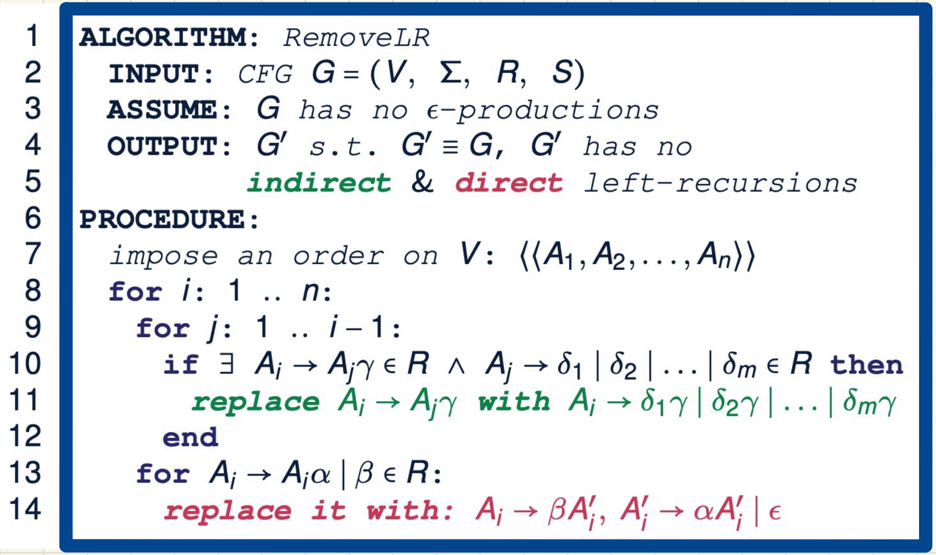
$C \to Brt$

$\boxed{I, J = 3, 2}$

↳ exercise.

Removing Left-Recursions (2b)

Does the **order** of variables matter?



```
ALGORITHM: RemoveLR
  INPUT: CFG $G = (V, \Sigma, R, S)$
  ASSUME: $G$ has no $\epsilon$-productions
  OUTPUT: $G'$ s.t. $G' \equiv G$, $G'$ has no
          indirect & direct left-recursions
PROCEDURE:
  impose an order on $V$: $\langle\langle A_1, A_2, \ldots, A_n \rangle\rangle$
  for $i$: 1 .. $n$:
    for $j$: 1 .. $i-1$:
      if $\exists\ A_i \to A_j\gamma \in R \ \wedge\ A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_m \in R$ then
        replace $A_i \to A_j\gamma$ with $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_m\gamma$
      end
    for $A_i \to A_i\alpha \mid \beta \in R$:
      replace it with: $A_i \to \beta A_i'$, $A_i' \to \alpha A_i' \mid \epsilon$
```

Indirectly Left-Recursive CFG:

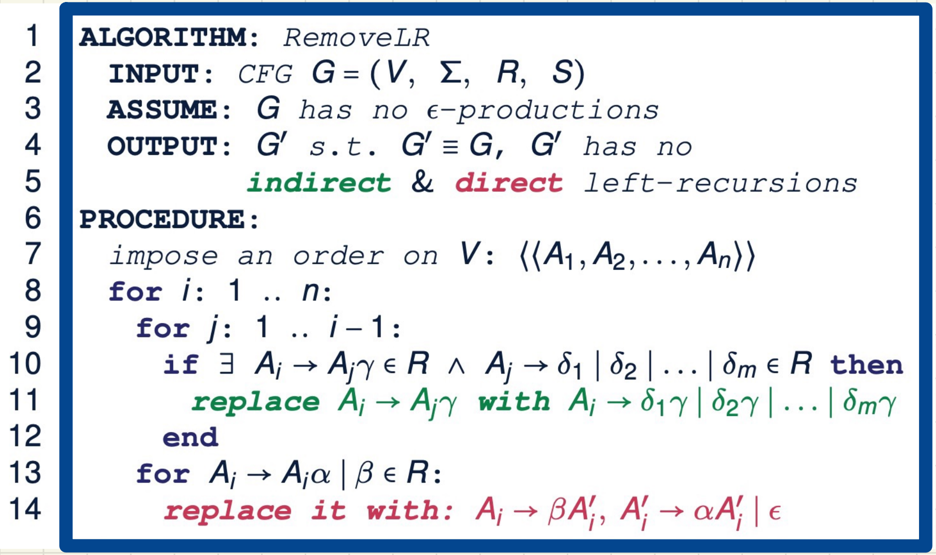
① $C \longrightarrow At$

② $B \longrightarrow Cd$

③ $A \longrightarrow Br$

Removing Left-Recursions (2b)

Does the order of variables matter?



```
ALGORITHM: RemoveLR
  INPUT: CFG $G = (V, \Sigma, R, S)$
  ASSUME: $G$ has no $\epsilon$-productions
  OUTPUT: $G'$ s.t. $G' \equiv G$, $G'$ has no
          indirect & direct left-recursions
PROCEDURE:
  impose an order on $V$: $\langle\langle A_1, A_2, \ldots, A_n \rangle\rangle$
  for $i$: 1 .. $n$:
    for $j$: 1 .. $i-1$:
      if $\exists\ A_i \to A_j\gamma \in R \wedge A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_m \in R$ then
        replace $A_i \to A_j\gamma$ with $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_m\gamma$
      end
    for $A_i \to A_i\alpha \mid \beta \in R$:
      replace it with: $A_i \to \beta A_i'$, $A_i' \to \alpha A_i' \mid \epsilon$
```

i	j
① C	–
② B	① C
③ A	①, ② C B

Indirectly Left-Recursive CFG:

① $C \longrightarrow At$

② ~~$B \longrightarrow Cd$~~ $B \to Atd$

③ ~~$A \longrightarrow Br$~~ $A \to Atdr$

(1) $\boxed{i, j = 2, 1}$

$B \to Cd$, $C \to At$

↳ $\boxed{B \to Atd}$

(2) $\boxed{i, j = 3, 2}$

$A \to Br$, $B \to Atd$

↳ $\boxed{A \to Atdr}$

Removing Left-Recursions (2c)

Exercise

```
ALGORITHM: RemoveLR
  INPUT: CFG $G = (V, \Sigma, R, S)$
  ASSUME: $G$ has no $\epsilon$-productions
  OUTPUT: $G'$ s.t. $G' \equiv G$, $G'$ has no
          indirect & direct left-recursions
PROCEDURE:
  impose an order on $V$: $\langle\langle A_1, A_2, \ldots, A_n \rangle\rangle$
  for $i$: 1 .. $n$:
    for $j$: 1 .. $i-1$:
      if $\exists\ A_i \rightarrow A_j\gamma \in R \ \wedge\ A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_m \in R$ then
        replace $A_i \rightarrow A_j\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_m\gamma$
    end
    for $A_i \rightarrow A_i\alpha \mid \beta \in R$:
      replace it with: $A_i \rightarrow \beta A_i',\ A_i' \rightarrow \alpha A_i' \mid \epsilon$
```

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

$A_i \quad A_i \quad \alpha \qquad\qquad \beta$

$D \rightarrow D\underline{fdaa} \mid \boxed{gdaa \mid eaa \mid ba \mid f \mid g}$

$D \rightarrow Cdaa \mid eaa \mid ba \mid f \mid Cg$

$D \rightarrow Baa \mid ba \mid f \mid Cg$

$D \to f \mid Dfdaa \mid gdaa \mid eaa \mid ba \mid Dfg$

$A \to B_1 \underline{B_2 B_3}$

<u>B_1 is nullable</u> $B_1 \to b \mid \varepsilon$

<u>B_2, B_3 are nullable</u> $B_2 \to c \mid \varepsilon$

$B_3 \to d \mid \varepsilon$

$B \to C\boxed{A}D$

nullable

$\underline{C \to c}$

$\underline{D \to d}$

$A \to a \mid \cancel{\varepsilon}$

↓

$B \to CD$

$\quad \mid CaD$

$A \to a$

$$A \to \underline{X_1}\ \underline{X_2} \cdots \underline{X_{10}}$$

↳ What if all 10 variables nullable

What if $\underline{X_2}$, $\underline{X_3}$, $\underline{X_4}$ are nullable. $2^{10} - \boxed{1}$

↓ when all variables produce ε

How many versions of A to produce?

$$\boxed{2^3}$$

$$A \to X_1\ \_\ \_\ \_\ X_5 \cdots X_{10}$$

$X_3\ X_4$

$X_2\ \ X_4$

$X_2\ X_3$

⋮

Eliminating epsilon-Productions

$$
\begin{array}{lcl}
S & \rightarrow & \underline{A}\,\underline{B} \\
A & \rightarrow & aAA \mid \epsilon \\
B & \rightarrow & bBB \mid \underline{\underline{\epsilon}}
\end{array}
$$

Q: **Nullable** variables?

↳ $S \rightarrow B \mid A \mid AB$

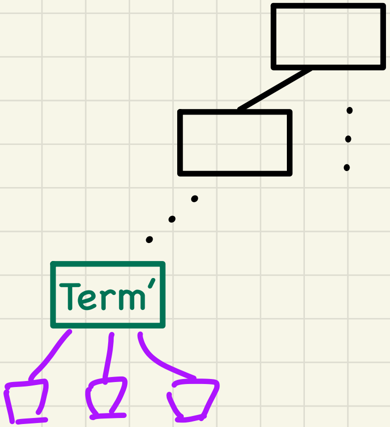
$A \rightarrow a\underline{AA} \mid aA \mid a$

$B \rightarrow bBB \mid bB \mid b$

Top-Down Parsing: Backtrack

```
ALGORITHM: TDParse
  INPUT: CFG $G = (V, \Sigma, R, S)$
  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 $\in V$ then
        if $\exists$ unvisited rule focus $\rightarrow \beta_1 \beta_2 \dots \beta_n \in R$ then
           create $\beta_1, \beta_2 \dots \beta_n$ as children of focus
           trace.push($\beta_n \beta_{n-1} \dots \beta_2$)
           focus := $\beta_1$
        else
           if focus = $S$ then report syntax error
           else backtrack
     elseif word matches focus then
        word := NextWord()
        focus := trace.pop()
     elseif word = EOF $\wedge$ focus = null then return root
     else backtrack
```

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



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

FIRST Set

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

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

→ what if:

Factor → ε

	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

Lecture 17 - Nov. 15

Syntactic Analysis

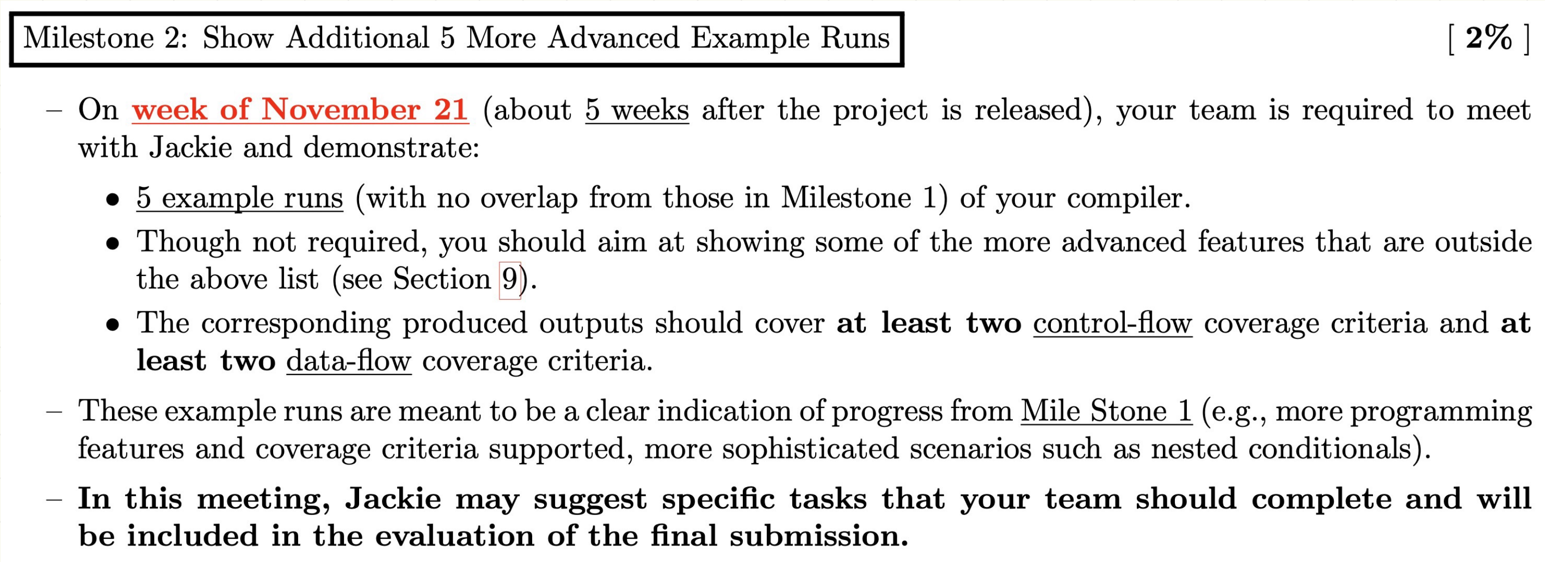
FIRST Set: Algorithm

Announcements

- Assignment 3 released

- Project Milestone 2 meeting signup starting 6pm on Wednesday

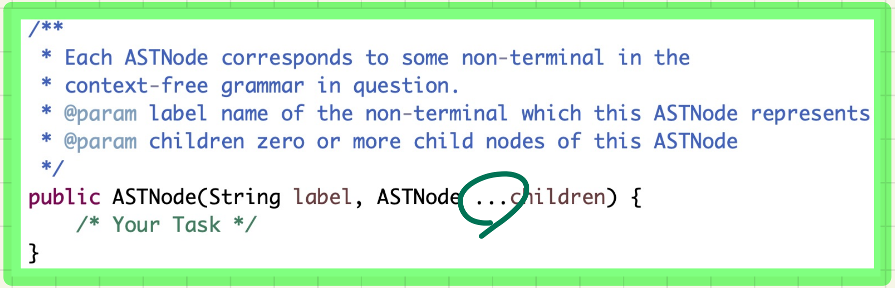
Project: Milestone 2



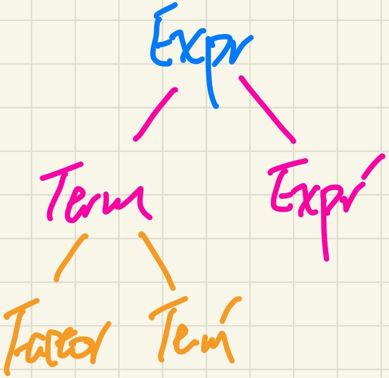
Milestone 2: Show Additional 5 More Advanced Example Runs [**2%**]

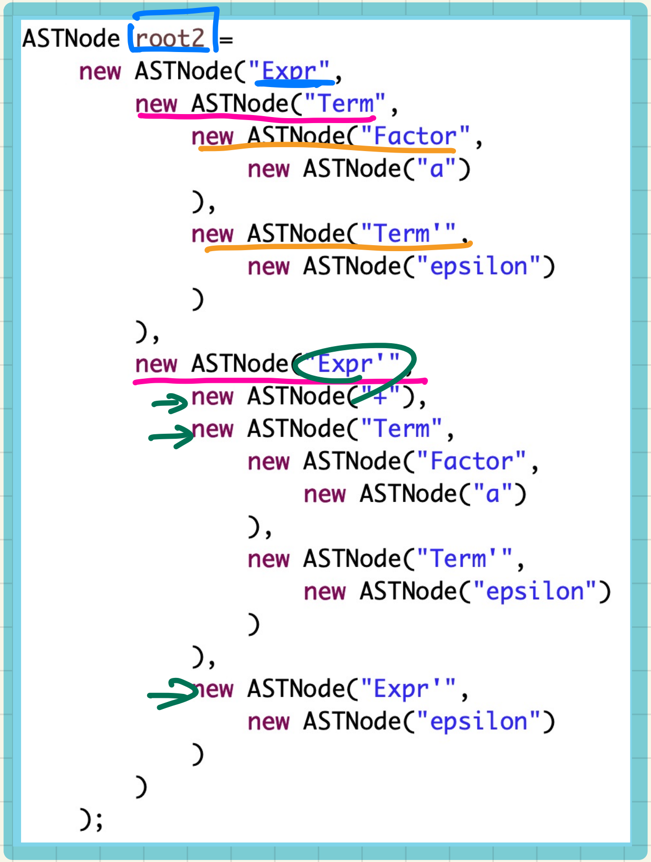
- On week of November 21 (about 5 weeks after the project is released), your team is required to meet with Jackie and demonstrate:
  - 5 example runs (with no overlap from those in Milestone 1) of your compiler.
  - Though not required, you should aim at showing some of the more advanced features that are outside the above list (see Section 9).
  - The corresponding produced outputs should cover **at least two** control-flow coverage criteria and **at least two** data-flow coverage criteria.
- These example runs are meant to be a clear indication of progress from Mile Stone 1 (e.g., more programming features and coverage criteria supported, more sophisticated scenarios such as nested conditionals).
- **In this meeting, Jackie may suggest specific tasks that your team should complete and will be included in the evaluation of the final submission.**

Assignment$\overset{3}{2}$: Variable Arguments



```
/**
 * Each ASTNode corresponds to some non-terminal in the
 * context-free grammar in question.
 * @param label name of the non-terminal which this ASTNode represents
 * @param children zero or more child nodes of this ASTNode
 */
public ASTNode(String label, ASTNode ...children) {
    /* Your Task */
}
```





```
ASTNode root2 =
    new ASTNode("Expr",
        new ASTNode("Term",
            new ASTNode("Factor",
                new ASTNode("a")
            ),
            new ASTNode("Term'",
                new ASTNode("epsilon")
            )
        ),
        new ASTNode("Expr'",
            new ASTNode("+"),
            new ASTNode("Term",
                new ASTNode("Factor",
                    new ASTNode("a")
                ),
                new ASTNode("Term'",
                    new ASTNode("epsilon")
                )
            ),
            new ASTNode("Expr'",
                new ASTNode("epsilon")
            )
        )
    );
```

FIRST Set: Algorithm

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

terminals/words

starting symbols

terminal

variable

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

function

given a valid (non-)terminal, return its FIRST symbols

FIRST for terminals

as soon as FIRST map cannot be grown exit.

exit when: $\epsilon \notin$ FIRST(β_i) $\vee$ $i \geq k$

keep collecting FIRST until its not nullable

every component of A is nullable ⇒ A is nullable

🟧 β_i is **nullable**

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

$$A \longrightarrow \beta_1\, \beta_2\ \cdots\ \beta_{k-1}\ \beta_k$$

need to move on to β2

stop right here no need to go to β2

(C1) $\epsilon \in$ FIRST(β_1)

(C2) $\epsilon \notin$ FIRST(β_1)

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

β1 β2 β3

FIRST Set: Tracing

First choose rules whose RHS starts with a terminal

F, E′, T′, T, E

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

β1 β2 β3

Factor → (Expr)

not executed

∵ FIRST("(") does not contain ε

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

Expr	Expr′	Term	Term′	Factor
∅	∅	∅	∅	∅
	{+}			{(}
	{+, -}			{(, num}
	{+, -, ε}			{(, num, name}

FIRST Set

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

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

nullable

| Term′ Factor

β_1 β_2

FIRST(Expr′) = {+, -, ε} ∪ FIRST(Term′) ε ∈ FIRST(Term′) ∪ FIRST(Factor)

Q. Will FIRST(Expr′) change if we add another rule?

Expr′ → Term′ Factor

Lecture 18 - Nov. 17

Syntactic Analysis

Extended FIRST Set Computation
FOLLOW Set, START Set, Left Factoring
TDP: Terminating & Min. Backtracking

Announcements

- **Assignment 3** released
- **Project Milestone 2** submission due at 11:59pm on Tuesday, Nov. 22
- **Project Report Template** to be walked over on Tuesday's class

Extended First Set

Q. How about FIRST(Expr′ → Term′ Factor)?

	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

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

variable or terminal

k-1 β1, β2, ... βk-1

nullable

first component that's not nullable
⇒ no need to collect FIRST further.

$A \rightarrow \beta_1 \beta_2 \ldots \beta_{k-1} \mid \beta_k \ldots \beta_n$

nullable   not nullable

$\epsilon \notin$ FIRST(Term)

FIRST(Term Expr′)
= FIRST(Term) = {(, n, n}

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?

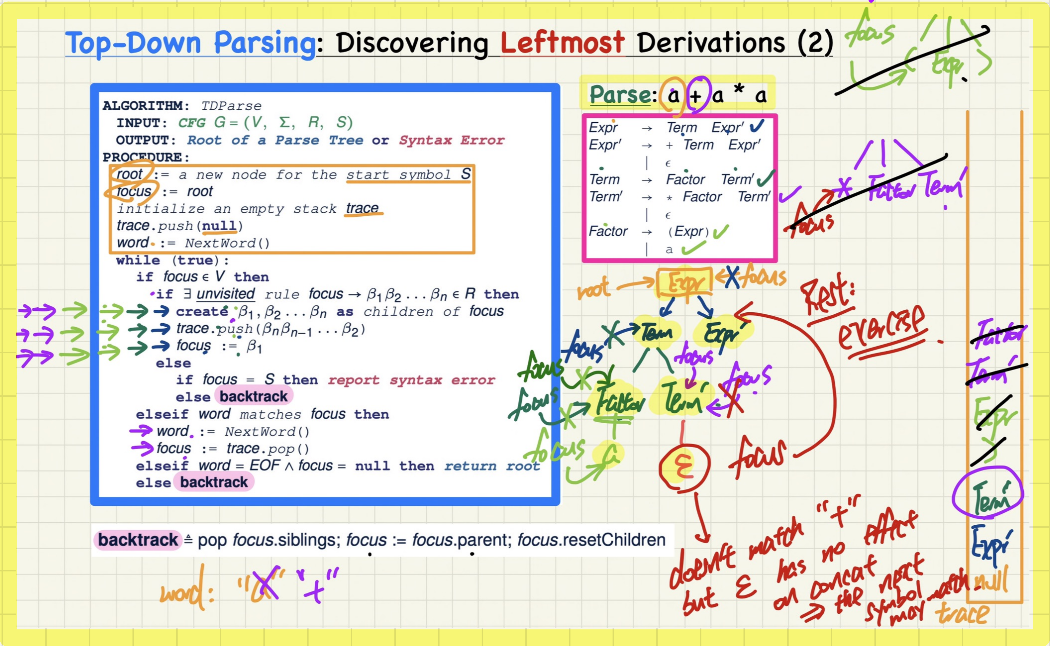
$$
\begin{array}{llllll}
Expr' & \rightarrow & + & Term & Term' & (1) \\
 & | & - & Term & Term' & (2) \\
 & | & \epsilon & & & (3)
\end{array}
$$

useful if we can know what symbols follows Expr'

FIRST(+ Term Term') = {+}

FIRST(- Term Term') = {-}

FIRST(epsilon) = {ε}



Top-Down Parsing: Discovering Leftmost Derivations (2)

```
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
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

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

Parse: a + a * a

Expr	→	Term Expr' ✓
Expr'	→	+ Term Expr'
	|	ϵ
Term	→	Factor Term' ✓
Term'	→	⋆ Factor Term'
	|	ϵ
Factor	→	(Expr) ✓
	|	a ✓

Rest: exercise

doesn't match "+" but ε has no effect on concat ⇒ the next symbol may match null trace

word: "a" "+"

FOLLOW Set

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

terminals only

Expr' ⇒ Term Expr'

start variable

derived from v

a string that follows the derivation of v

assumption: FIRST is already computed.

Right-Recursive CFG:

0	*Goal*	→	*Expr* eof	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

FOLLOW(Expr) ∵ FIRST(Expr') contains ε

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

FOLLOW(Expr)

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 $G = (V, \Sigma, R, S)$
  OUTPUT: FOLLOW : $V \longrightarrow \mathbb{P}(T \cup \{eof\})$
PROCEDURE:
  for $A \in V$: FOLLOW($A$) := $\varnothing$
  FOLLOW($S$) := $\{eof\}$
  $lastFollow$ := $\varnothing$
  while ($lastFollow \neq$ FOLLOW) :
    $lastFollow$ := FOLLOW
    for $A \rightarrow \beta_1\beta_2\ldots\beta_k \in R$:
      trailer := FOLLOW($A$)
      for i: k .. 1:
        if $\beta_i \in V$ then
          FOLLOW($\beta_i$) := FOLLOW($\beta_i$) $\cup$ trailer
          if $\epsilon \in$ FIRST($\beta_i$)
            then trailer := trailer $\cup$ (FIRST($\beta_i$) $- \epsilon$)
            else trailer := FIRST($\beta_i$)
        else
          trailer := FIRST($\beta_i$)
```

map

start variable

when considering this rule, Follow may need to be updated for k ≡

→ go from right to left iterates

is βi nullable.

$$A \longrightarrow \beta_1\ \beta_2\ \cdots\ \boxed{\beta_{k-1}}\ \beta_k$$

Follow?

if βi nullable, accumulate its FIRST for the Follow of βi-1

FOLLOW(βk) = ? Follow(A)

When ε ∈ FIRST(βk)

FOLLOW(βk-1) = ?

When ε ∉ FIRST(βk)

FOLLOW(βk-1) = ?

if βi is not nullable, then don't include its FIRST.

Computing the FOLLOW Sets: Trailers

$A \rightarrow \beta_1\beta_2\beta_3$

not nullable

follow of β_2

not nullable

$A \rightarrow \beta_1 \beta_2 | \beta_3$

Case 1: $\varepsilon \notin$ FIRST(β_3), $\varepsilon \notin$ FIRST(β_2)

- FOLLOW(β_3) = Follow(A)
- FOLLOW(β_2) = FIRST(β_3) ∪ ~~Follow(β_3)~~ ?
- FOLLOW(β_1) = FIRST(β_2).

nullable nullable

$A \rightarrow \beta_1 | \beta_2 | \beta_3$

Case 2: $\varepsilon \in$ FIRST(β_3), $\varepsilon \in$ FIRST(β_2)

- FOLLOW(β_3) = Follow(A)
- FOLLOW(β_2) = FIRST(β_3) ∪ Follow(β_3)
- FOLLOW(β_1) = FIRST(β_2) ∪ FIRST(β_3) ∪ Follow(β_3)

trailer

Right-Recursive CFG:

nullable

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

Follow = Follow (E)

β_1 β_2 β_3

G, F, E, T, T′

```
ALGORITHM: GetFollow
  INPUT: CFG $G = (V, \Sigma, R, S)$
  OUTPUT: FOLLOW : $V \longrightarrow \mathbb{P}(T \cup \{eof\})$
PROCEDURE:
  for $A \in V$: FOLLOW($A$) := $\varnothing$
  FOLLOW($S$) := $\{eof\}$
  lastFollow := $\varnothing$
  while (lastFollow ≠ FOLLOW):
    lastFollow := FOLLOW
    for $A \to \beta_1 \beta_2 \dots \beta_k \in R$:
      trailer := FOLLOW($A$)
      for $i$: $k$ .. 1:
        if $\beta_i \in V$ then
          FOLLOW($\beta_i$) := FOLLOW($\beta_i$) ∪ trailer
          if $\epsilon \in$ FIRST($\beta_i$)
            then trailer := trailer ∪ (FIRST($\beta_i$) − $\epsilon$)
            else trailer := FIRST($\beta_i$)
        else
          trailer := FIRST($\beta_i$)
```

FOLLOW Set: Tracing

First choose rules whose LHS is processed.

Then rules whose RHS ends with a terminal.

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

Goal	*Expr*	*Expr′*	*Term*	*Term′*	*Factor*
eof	eof,)	eof,)	+, −, eof,)		*, ÷,

→ FIRST(")")

↳ Follow(Expr′) ∵ RHS nullable

Backtrack-Free Grammar

A ***backtrack-free grammar*** has each of its productions $A \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ satisfying:

$$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \boxed{\textsf{START}(\gamma_i) \cap \textsf{START}(\gamma_j) = \varnothing}$$

between any two production rules, we can always unambiguously choose one.

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

→ β is not nullable

→ β is nullable

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

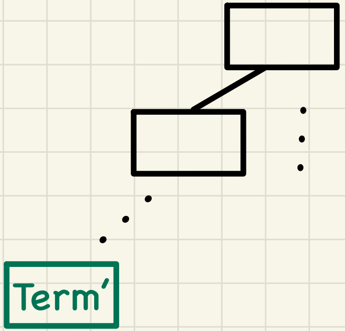
0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

Top-Down Parsing: Algorithm with lookahead

```
ALGORITHM: TDParse
  INPUT: CFG $G = (V, \Sigma, R, S)$
  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 → $\beta_1 \beta_2 \ldots \beta_n \in R$ ∧ word ∈ START(β) then
        create $\beta_1, \beta_2 \ldots \beta_n$ as children of focus
        trace.push($\beta_n \beta_{n-1} \ldots \beta_2$)
        focus := $\beta_1$
      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
```

always choose the prod. rule whose start symbol matches.

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



Lecture 19 - Nov. 22

Syntactic Analysis

Left Factoring
TDP: Terminating & Min. Backtracking
LL(1) vs. LR(1) Parser
Bottom-Up Parsing, RMDs

Announcements

- Project Milestone 2 due tonight
- Assignment 3 due soon
- Project Report Template

Backtrack-Free Grammar: Exercise

A ***backtrack-free grammar*** has each of its productions $A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ satisfying:

$$\forall i,j : 1 \le i,j \le n \land i \ne j \bullet \text{START}(\gamma_i) \cap \text{START}(\gamma_j) = \varnothing$$

no ambiguity in choosing a production rule using a lookahead symbol

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

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

Is the following CFG ***backtrack free***? No common prefix.

word: name.

11	*Factor*	→	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	, *Expr MoreArgs*
17		|	ϵ

START(Factor → name)

= ? {name} " FIRST(name) {name}

START(| name [ArgList])

= {name}

Left-Factoring: Removing Common Prefixes

Identify a common prefix α:

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

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

Rewrite that production rule as:

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

α β_1 β_2

11	*Factor*	→	name |ε.
12		|	name [*ArgList*]
13		|	name (*ArgList*) β_3
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	, *Expr MoreArgs*
17		|	ϵ

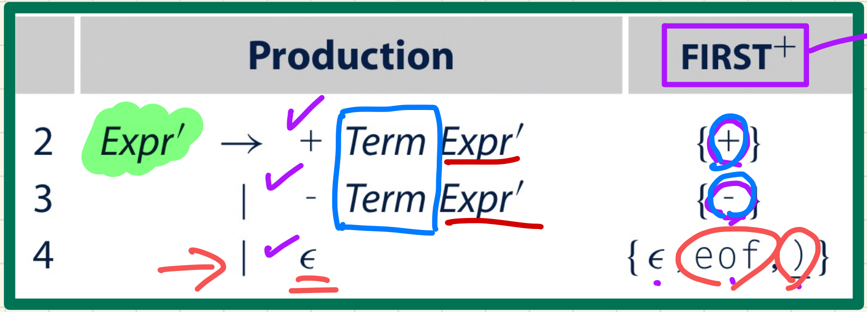
→

Factor → name Arguments

Arguments → ε
| [Arglist]
| (Arglist]

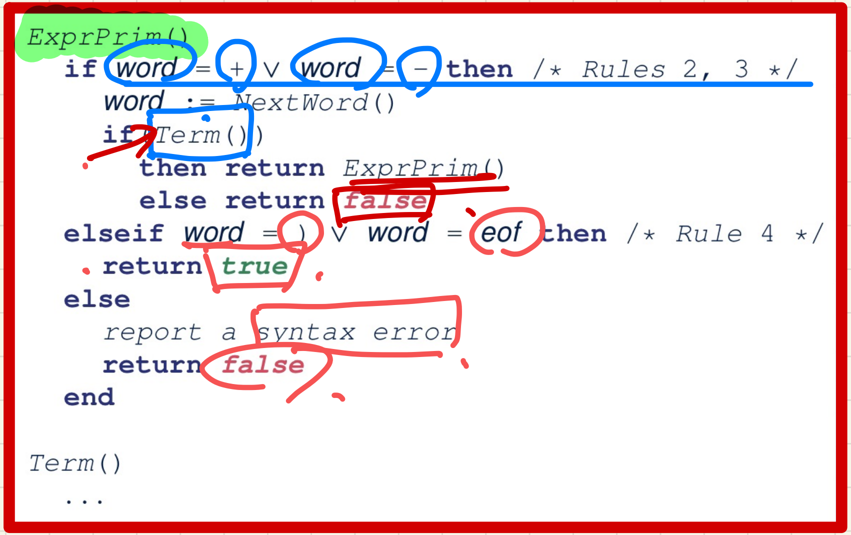
↓ satisfy the backtrack-free property.

Implementing a Recursive-Descent Parser



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

START



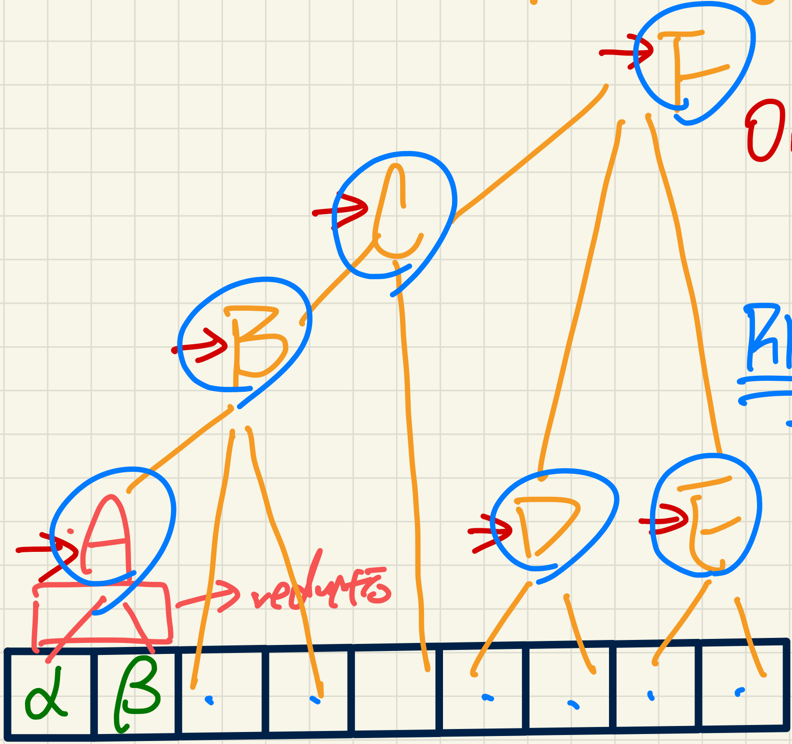
```
ExprPrim()
  if word = + ∨ word = - then /* Rules 2, 3 */
    word := NextWord()
    if (Term())
      then return ExprPrim()
      else return false
  elseif word = ) ∨ word = eof then /* Rule 4 */
    return true
  else
    report a syntax error
    return false
  end

Term()
  ...
```

Discovering Derivations: Bottom-Up Parsing

production rule

$$\boxed{A \rightarrow \alpha\beta}$$



scanner

Order of reductions:

A B C D E F

RMD:

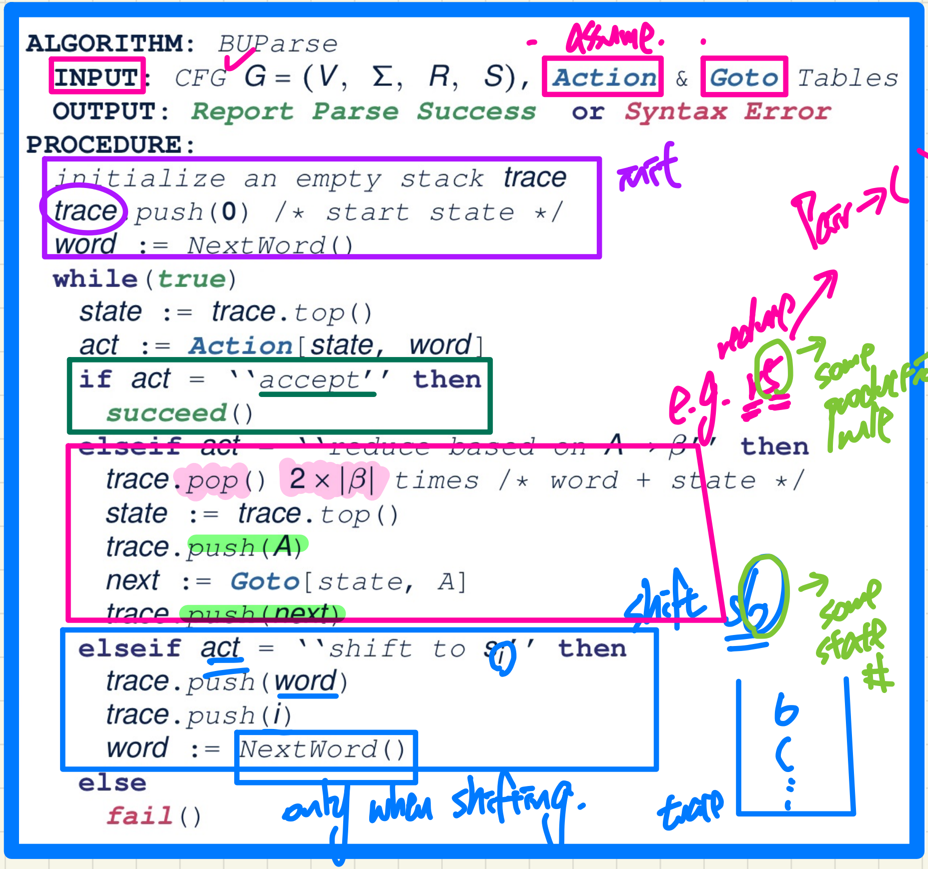
F E D C B A

Lecture 20 - Nov. 24

Syntactic Analysis

Bottom-Up Parsing: shift vs. reduce
Exercise: LL(1) Parser

Bottom-Up Parsing: Algorithm



assume.

```
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(0) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ''accept'' then
      succeed()
    elseif act = ''reduce based on 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 to si'' then
      trace.push(word)
      trace.push(i)
      word := NextWord()
    else
      fail()
```

start

Pair → ( )

reduce

e.g. r5 → some production rule

shift s6 → some state #

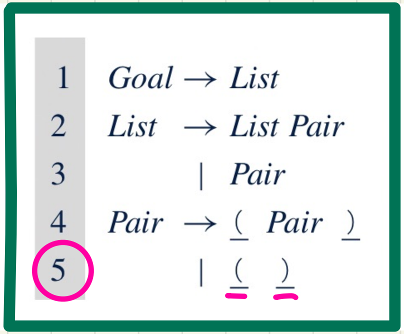
b

(

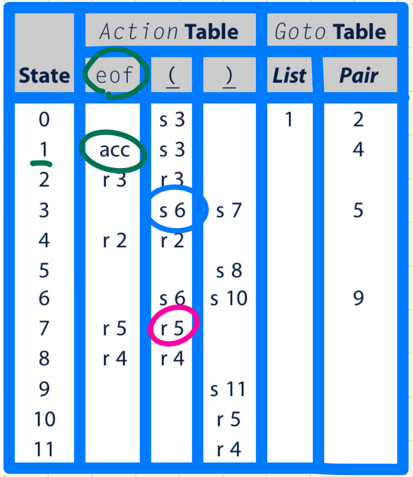
:

trace

only when shifting.



$$
\begin{array}{rlcl}
1 & Goal & \rightarrow & List \\
2 & List & \rightarrow & List\ Pair \\
3 & & | & Pair \\
4 & Pair & \rightarrow & \underline{(}\ \ Pair\ \ \underline{)} \\
5 & & | & \underline{(}\ \ \underline{)}
\end{array}
$$



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		

Bottom-Up Parsing: Discovering Rightmost Derivations (1)

```
ALGORITHM: BUParse
  INPUT: CFG $G = (V, \Sigma, R, S)$, Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(0) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ''accept'' then
      succeed()
    elseif act = ''reduce based on $A \rightarrow \beta$'' then
      trace.pop() $2 \times |\beta|$ times /* word +
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ''shift to $s_i$'' then
      trace.push(word) ✓ ✓
      trace.push(i) ✓ ✓
      word := NextWord()
    else
      fail()
```

2 1 0 3 "(" ")" eof eof

List Pair r3 r5 s3 s7

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: ( )

word: ~~"("~~ ~~")"~~ eof

state: ~~0~~ ~~3~~ ~~7~~ ~~0~~ ~~2~~ ~~0~~ 1

handle

1
List
~~2~~
~~Pair~~
~~7~~
~~)~~
~~3~~
~~(~~
0

trace

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(0) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce based on $A \rightarrow \beta$'' then
      trace.pop() $2 \times |\beta|$ times /* word +
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift to $s_i$'' then
      trace.push(word)
      trace.push(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		

Bottom-Up Parsing: Discovering Rightmost Derivations (3)

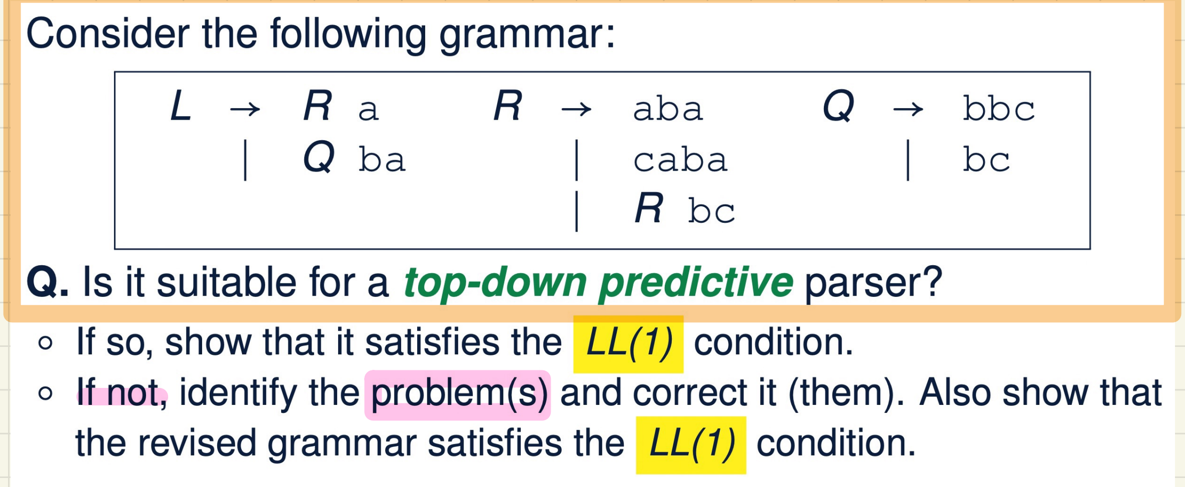
```
ALGORITHM: BUParse
  INPUT: CFG $G = (V, \Sigma, R, S)$, Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(0) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce based on $A \rightarrow \beta$'' then
      trace.pop() $2 \times |\beta|$ times /* word +
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift to $s_i$'' then
      trace.push(word)
      trace.push(i)
      word := NextWord()
    else
      fail()
```

Parse: ( ))

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

	Action Table			Goto Table	
State	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		

Exercise: LL(1) Parser

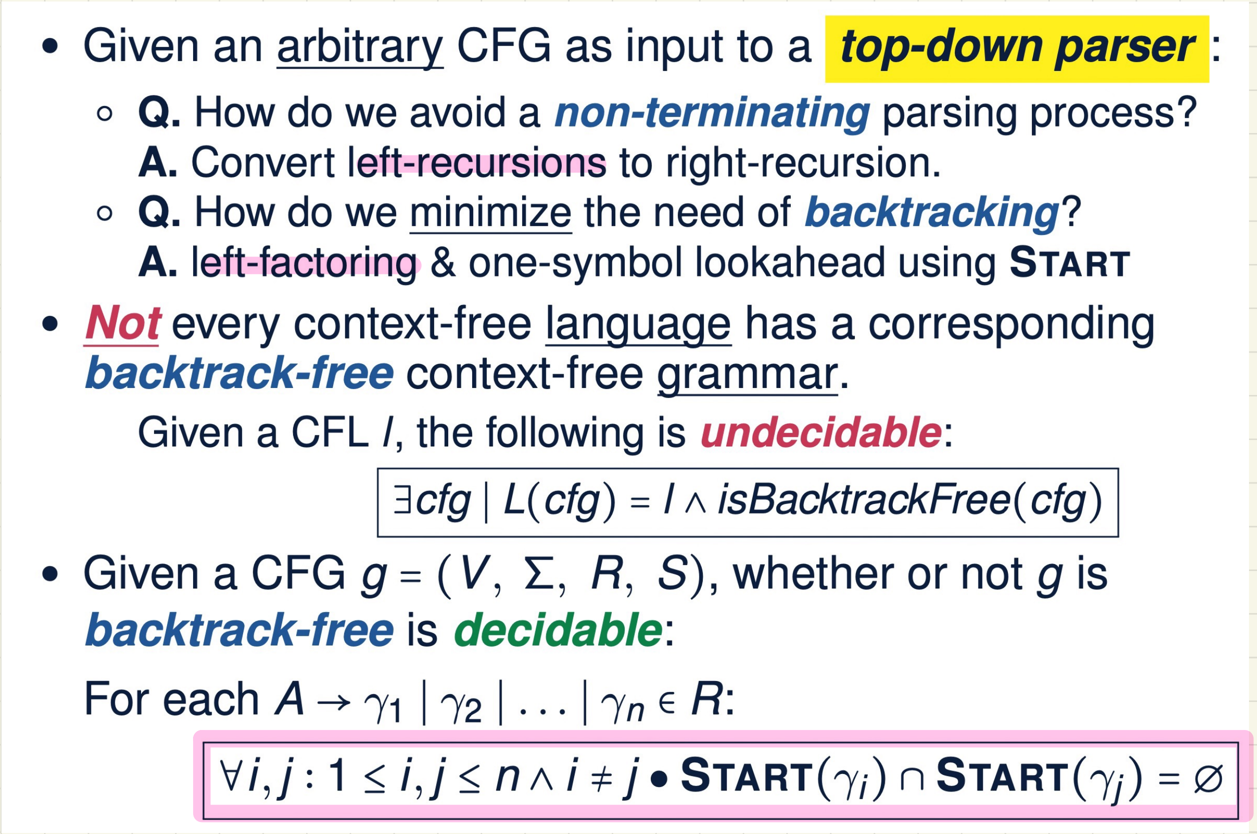


Consider the following grammar:

$$\begin{array}{llll}
L & \rightarrow & R\ \texttt{a} \\
 & \mid & Q\ \texttt{ba}
\end{array}
\qquad
\begin{array}{llll}
R & \rightarrow & \texttt{aba} \\
 & \mid & \texttt{caba} \\
 & \mid & R\ \texttt{bc}
\end{array}
\qquad
\begin{array}{llll}
Q & \rightarrow & \texttt{bbc} \\
 & \mid & \texttt{bc}
\end{array}$$

Q. Is it suitable for a ***top-down predictive*** parser?

- If so, show that it satisfies the *LL(1)* condition.
- If not, identify the problem(s) and correct it (them). Also show that the revised grammar satisfies the *LL(1)* condition.



- Given an arbitrary CFG as input to a ***top-down parser***:
  - **Q.** How do we avoid a ***non-terminating*** parsing process?
    A. Convert left-recursions to right-recursion.
  - **Q.** How do we minimize the need of ***backtracking***?
    A. left-factoring & one-symbol lookahead using **START**
- ***Not*** every context-free language has a corresponding ***backtrack-free*** context-free grammar.

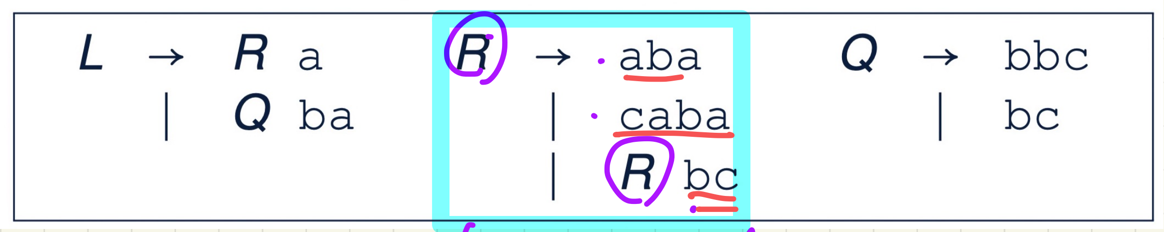
  Given a CFL l, the following is ***undecidable***:

  $$\boxed{\exists cfg \mid L(cfg) = l \wedge isBacktrackFree(cfg)}$$

- Given a CFG $g = (V, \Sigma, R, S)$, whether or not g is ***backtrack-free*** is ***decidable***:

  For each $A \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \in R$:

  $$\boxed{\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \textbf{START}(\gamma_i) \cap \textbf{START}(\gamma_j) = \varnothing}$$

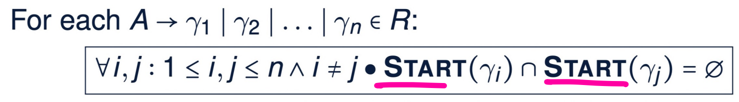


$L \to R\ \mathtt{a} \mid Q\ \mathtt{ba}$

$R \to \mathtt{aba} \mid \mathtt{caba} \mid R\ \mathtt{bc}$

$Q \to \mathtt{bbc} \mid \mathtt{bc}$

direct left recursion



For each $A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \in R$:

$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \textsc{Start}(\gamma_i) \cap \textsc{Start}(\gamma_j) = \varnothing$

aba

caba bc bc bc

Ex1: Remove left Recursion

$R \to abaR' \mid cabaR'$

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

$L \to Ra \mid Qba$

$R \to abaR' \mid cabaR'$

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

$Q \to bbc \mid bc$

problematic

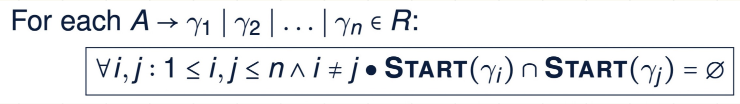
$$
\begin{aligned}
L &\to Ra \\
&\mid Qba \\
R &\to abaR' \\
&\mid cabaR' \\
R' &\to bcR' \\
&\mid \varepsilon
\end{aligned}
$$

$$
\boxed{\begin{aligned}
Q &\to \underline{b}\boxed{bc} \\
&\mid \underline{b}\boxed{c}
\end{aligned}}
\xrightarrow{\text{left factoring}}
\begin{aligned}
Q &\to bQ' \\
Q' &\to bc \\
&\mid c
\end{aligned}
$$

① left recursion

② common prefix

③



For each $A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \in R$:

$$\boxed{\forall i, j : 1 \le i, j \le n \land i \ne j \bullet \textbf{START}(\gamma_i) \cap \textbf{START}(\gamma_j) = \varnothing}$$

$L \to Ra$
$\quad | \; Qba$

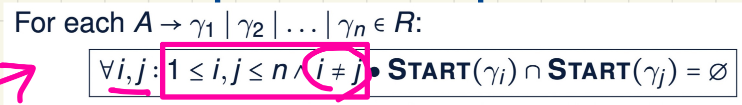
$R \to abaR'$
$\quad | \; cabaR'$

$R' \to bcR'$
$\quad | \; \varepsilon$

$Q \to bQ'$

$Q' \to bc$
$\quad | \; c$

Non-Terminal	Alternative	START Set	Intersection
Q'	bc	$\{b\}$	$\emptyset$
	c	$\{c\}$	
R	$abaR'$	$\{a\}$	$\emptyset$
	$cabaR'$	$\{c\}$	
L			
R'			
Q			



For each $A \to \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n \in R$:

$$\forall i, j : 1 \le i, j \le n \wedge i \ne j \bullet \textsf{START}(\gamma_i) \cap \textsf{START}(\gamma_j) = \varnothing$$

F

true

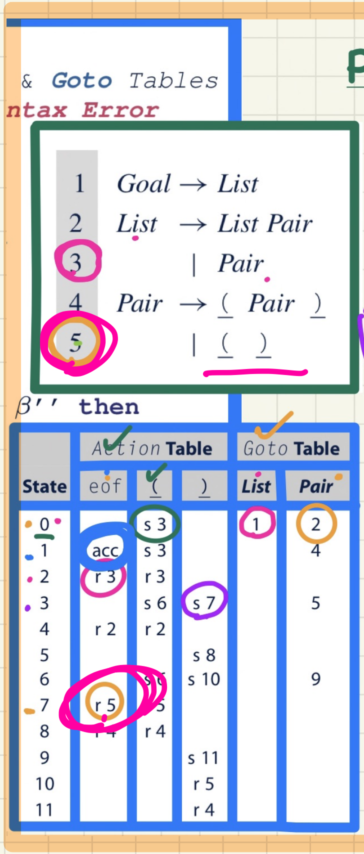
trivially.

Lecture 21 - Nov. 29

Syntactic Analysis

Bottom-Up Parsing: Handles
Bottom-Up Parsing: Reverse RMD
LR(1) Items: Definition & Exercises

Bottom-Up Parsing: Handles



& *Goto* Tables

ntax Error

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

β'' then

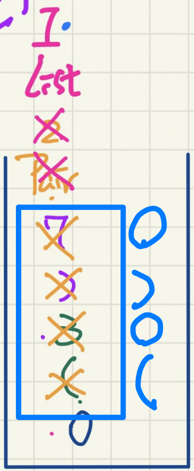
	Action Table			Goto Table	
State	eof	(	)	*List*	*Pair*
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

Parse: ( )

word: ~~"("~~ ~~")"~~ eof

state: ~~0~~ ~~3~~ ~~7~~ ~~0~~ ~~2~~ ~~0~~ 1

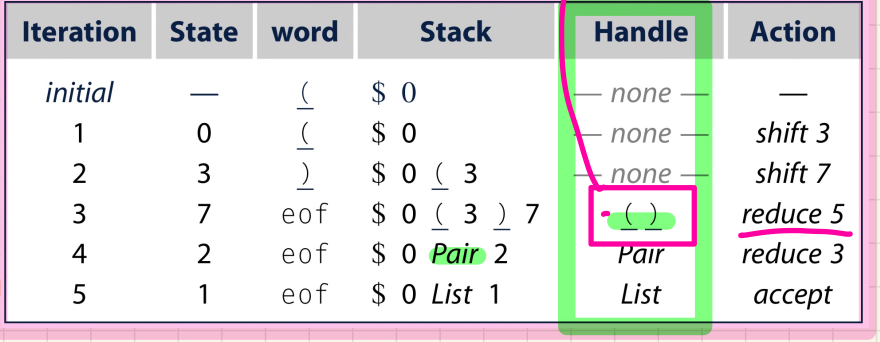
handle



trace

A **handle** denotes a parser's state that's ready for **reduction**.

state ready for reduction



Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— *none* —	—
1	0	(	$ 0	— *none* —	*shift 3*
2	3	)	$ 0 (3	— *none* —	*shift 7*
3	7	eof	$ 0 (3) 7	( )	*reduce 5*
4	2	eof	$ 0 *Pair* 2	*Pair*	*reduce 3*
5	1	eof	$ 0 *List* 1	*List*	*accept*

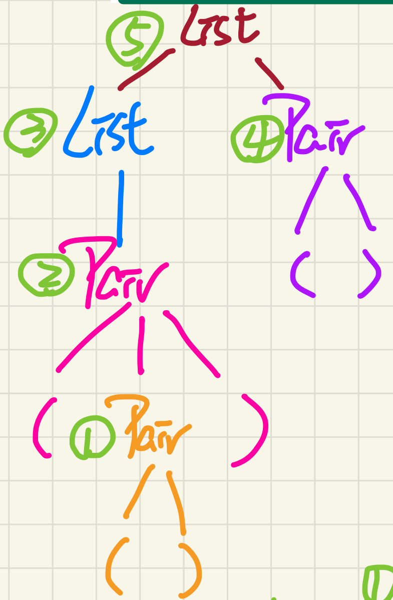
Bottom-Up Parsing: Right-Most Derivation

Parse: (( )) ( )

The **BUP** process corresponds to the revserse of a **RMD**.

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

Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— *none* —	—
1	0	(	$ 0	— *none* —	*shift 3*
2	3	(	$ 0 (3	— *none* —	*shift 6*
3	6	)	$ 0 (3 (6	— *none* —	*shift 10*
4	10	)	$ 0 (3 (6) 10	( )	*reduce 5*
5	5	)	$ 0 (3 *Pair* 5	— *none* —	*shift 8*
6	8	(	$ 0 (3 *Pair* 5) 8	(*Pair*)	*reduce 4*
7	2	(	$ 0 *Pair* 2	*Pair*	*reduce 3*
8	1	(	$ 0 *List* 1	— *none* —	*shift 3*
9	3	)	$ 0 *List* 1 (3	— *none* —	*shift 7*
10	7	eof	$ 0 *List* 1 (3) 7	( )	*reduce 5*
11	4	eof	$ 0 *List* 1 *Pair* 4	*List Pair*	*reduce 2*
12	1	eof	$ 0 *List* 1	*List*	*accept*



Order of reductions: ① – ⑤

Order of RMD: reverse!

LR(1) Items: Definition

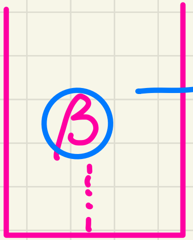
$$[A \to \beta \bullet \gamma, a]$$

production rule $A \to \beta\gamma$

possible states of parser

current top of stack

look-ahead $\in$ Follow(A)



- we have already recognize β
- once we recognize γ

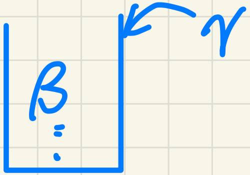
↳ in a handle ready for reducing into A

LR(1) Items: Scenarios

Possibility: $[\, A \rightarrow \bullet\, \beta\, \gamma,\ a \,]$

↳ initial state of parsing towards reduction to A

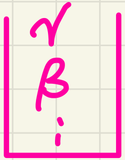
Partial Completion: $[\, A \rightarrow \beta \bullet \gamma,\ a \,]$



↳ already recognized β

still expecting to recognize γ

Completion: $[\, \underline{A} \rightarrow \underline{\beta}\, \underline{\gamma} \bullet,\ a \,]$



∈ Follow(A)

if word matches a, reduce to A

LR(1) Items: Exercise (1.1a)

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

Q. LR(1) item denoting the initial state of parsing?

$[Goal \rightarrow \bullet List, \boxed{eof}]$

← Follow(Goal)
{eof}

Q. LR(1) item denoting the desired final state of parsing?

not necessarily
≠
the final state
X

$[Pair \rightarrow (\ )\ \bullet,\ \ ]$

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

LR(1) Items: Exercise (1.1b)

Q. Derive **all** LR(1) items for the production rule $A \rightarrow \beta\gamma$

- union
- set comprehension
- floating "point"

D_1 : floating positions of •

→ $A \rightarrow \bullet\beta\gamma$

$A \rightarrow \beta\bullet\gamma$

$A \rightarrow \beta\gamma\bullet$

D_2 : Follow(A)

$$\{[A \rightarrow \bullet\beta\gamma,\ a] \mid a \in Follow(A)\}$$

$$\cup$$

$$\{[A \rightarrow \beta\bullet\gamma,\ a] \mid a \in Follow(A)\}$$

$$\cup$$

$$\{[A \rightarrow \beta\gamma\bullet,\ a]\ a \in Follow(A)\}$$

LR(1) Items: Exercise (1.2)

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	<u>(</u> *Pair* <u>)</u>
5		|	<u>(</u> <u>)</u>

How many LR(1) items?

- possible floating point positions: 4
- cardinality of Follow(Pair)

|Follow(Pair)|: 3

12

Q. Derive all LR(1) items for the production rule Pair → (Pair)

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

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

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

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

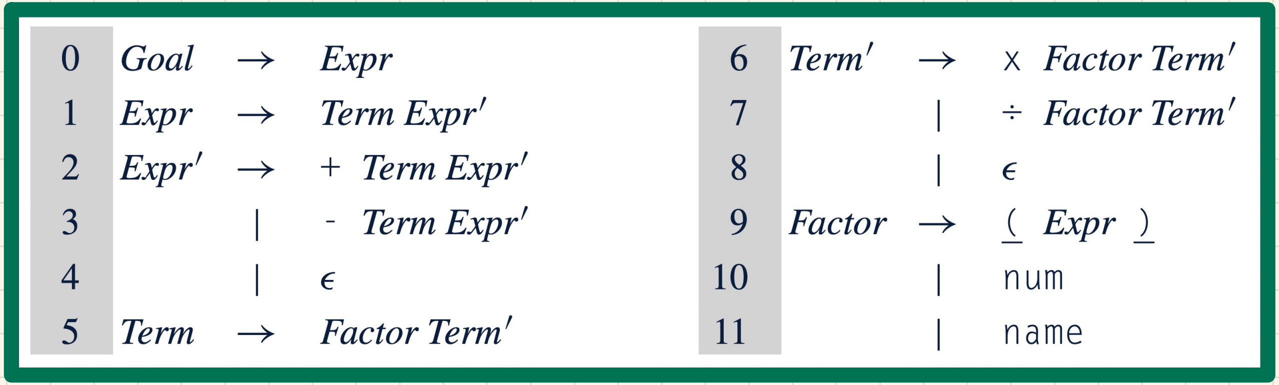
LR(1) Items: Exercise (1.3)

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

$\textsc{Follow}(List) = \{\texttt{eof}, (\}$ $\quad$ $\textsc{Follow}(Pair) = \{\texttt{eof}, (,)\}$

$[Goal \rightarrow \bullet\, List, \texttt{eof}]$		
$[Goal \rightarrow List\, \bullet, \texttt{eof}]$		
$[List \rightarrow \bullet\, List\ \ Pair, \texttt{eof}]$	$[List \rightarrow \bullet\, List\ \ Pair, \underline{(}\,]$	
$[List \rightarrow List \bullet Pair, \texttt{eof}]$	$[List \rightarrow List \bullet Pair, \underline{(}\,]$	
$[List \rightarrow List\ \ Pair\, \bullet, \texttt{eof}]$	$[List \rightarrow List\ \ Pair\, \bullet, \underline{(}\,]$	
$[List \rightarrow \bullet\, Pair, \texttt{eof}\,]$	$[List \rightarrow \bullet\, Pair, \underline{(}\,]$	
$[List \rightarrow Pair\, \bullet, \texttt{eof}\,]$	$[List \rightarrow Pair\, \bullet, \underline{(}\,]$	
$[Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \texttt{eof}\,]$	$[Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{)}]$	$[Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{(}]$
$[Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \texttt{eof}\,]$	$[Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \underline{)}]$	$[Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \underline{(}]$
$[Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \texttt{eof}\,]$	$[Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \underline{)}]$	$[Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \underline{(}]$
$[Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \texttt{eof}\,]$	$[Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \underline{)}]$	$[Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \underline{(}]$
$[Pair \rightarrow \bullet\, \underline{(}\ \ \underline{)}, \texttt{eof}]$	$[Pair \rightarrow \bullet\, \underline{(}\ \ \underline{)}, \underline{(}]$	$[Pair \rightarrow \bullet\, \underline{(}\ \ \underline{)}, \underline{)}]$
$[Pair \rightarrow \underline{(}\ \bullet\ \underline{)}, \texttt{eof}]$	$[Pair \rightarrow \underline{(}\ \bullet\ \underline{)}, \underline{(}]$	$[Pair \rightarrow \underline{(}\ \bullet\ \underline{)}, \underline{)}]$
$[Pair \rightarrow \underline{(}\ \ \underline{)}\, \bullet, \texttt{eof}]$	$[Pair \rightarrow \underline{(}\ \ \underline{)}\, \bullet, \underline{(}]$	$[Pair \rightarrow \underline{(}\ \ \underline{)}\, \bullet, \underline{)}]$

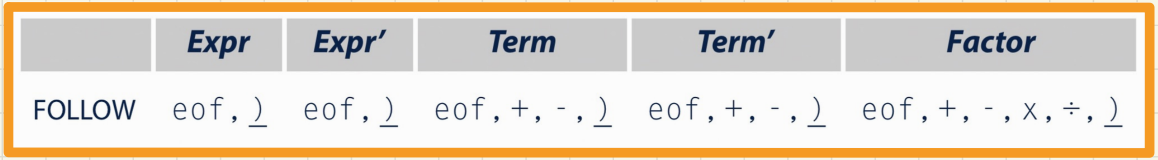
LR(1) Items: Exercise (2)



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

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

FOLLOW Set



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

Lecture 22 - Dec. 1

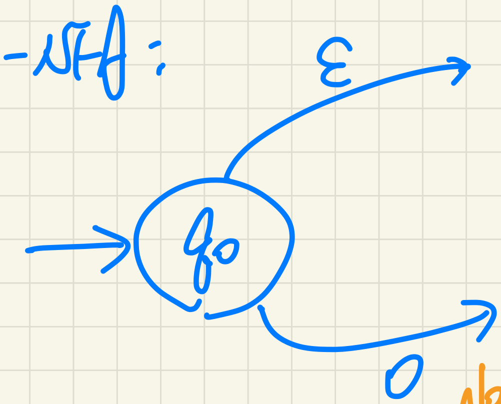
Syntactic Analysis

Canonical Collection vs. Subset States
Algorithms: closure, goto

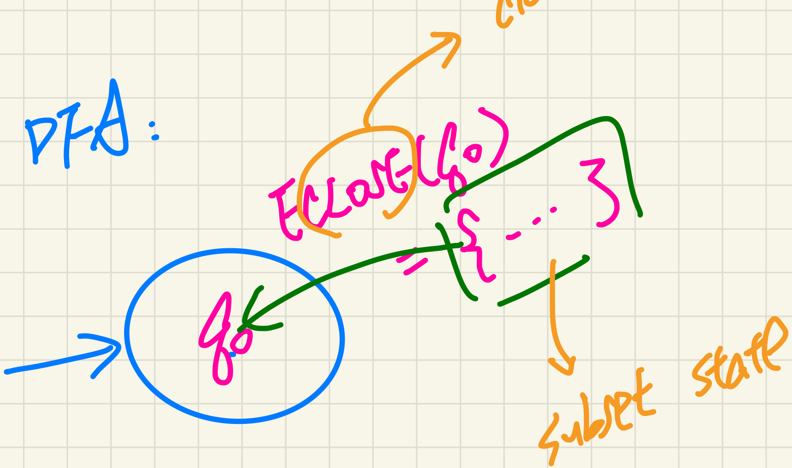
Announcements

- **Project** final submission guideline to be released on Friday
- **Review session** on Thursday, December 8?

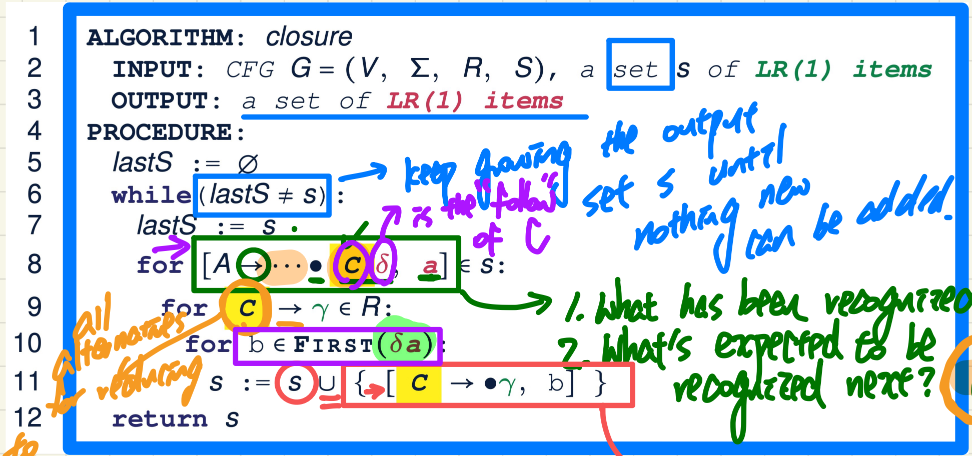
Input ε-NFA:



output DFA:



CC Construction: closure



```
ALGORITHM: closure
  INPUT: CFG $G = (V, \Sigma, R, S)$, a set s of LR(1) items
  OUTPUT: a set of LR(1) items
PROCEDURE:
  $lastS := \varnothing$
  while ($lastS \neq s$):
    $lastS := s$
    for $[A \rightarrow \cdots \bullet C\delta,\ a] \in s$:
      for $C \rightarrow \gamma \in R$:
        for $b \in \text{FIRST}(\delta a)$:
          $s := s \cup \{[C \rightarrow \bullet\gamma,\ b]\}$
  return s
```

keep growing the output set s until nothing new can be added.

is the "follow" of C

1. What has been recognized? ...
2. What's expected to be recognized next? C

all alternatives returning to C.

$b \in \text{FIRST}(\delta a)$

$\varepsilon \in \text{FIRST}(\delta)$

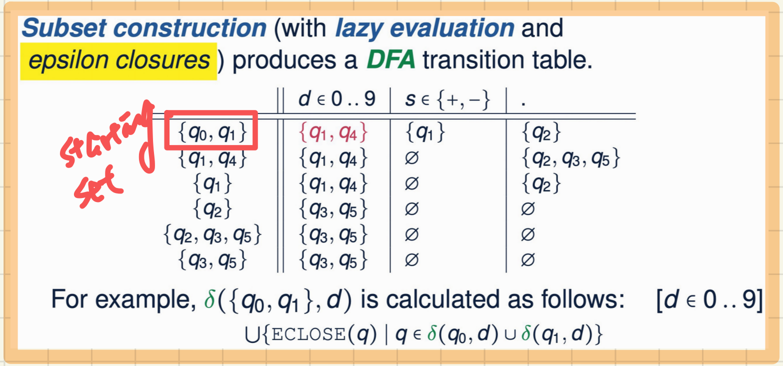
Q. Why not $b \in \text{FIRST}(\delta)$?

∵ δ might be nullable

$[C \rightarrow \bullet\gamma,\ b]$

new LR(1) item to be added to the closure.

Analogy: ε-NFA to DFA



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

	$d \in 0..9$	$s \in \{+,-\}$	.
$\{q_0, q_1\}$	$\{q_1, q_4\}$	$\{q_1\}$	$\{q_2\}$
$\{q_1, q_4\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2, q_3, q_5\}$
$\{q_1\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2\}$
$\{q_2\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_2, q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$

starting set

For example, $\delta(\{q_0, q_1\}, d)$ is calculated as follows: $[d \in 0..9]$

$$\bigcup\{\text{ECLOSE}(q) \mid q \in \delta(q_0, d) \cup \delta(q_1, d)\}$$

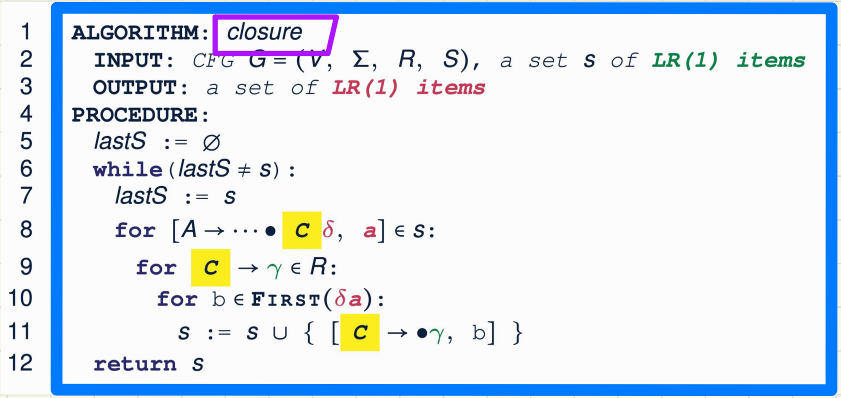
CC Construction: cc_0

set of subset states → a set of LR(1) items

Calculate cc_0 of the following grammar.

Hint: **Closure** of the **singleton set** containing the parser's **initial state**.

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



```
ALGORITHM: closure
  INPUT: CFG G = (V, Σ, R, S), a set s of LR(1) items
  OUTPUT: a set of LR(1) items
PROCEDURE:
  lastS := ∅
  while (lastS ≠ s):
    lastS := s
    for [A → ⋯ • C δ, a] ∈ s:
      for C → γ ∈ R:
        for b ∈ FIRST(δa):
          s := s ∪ { [C → •γ, b] }
  return s
```

initial subset state of the parser

ε-NFA q_0 → DFA → ECLOSE(q_0)

→ CC_0

parser's initial state:

$\{[Goal \to \bullet\, List,\ eof]\}$ = input to closure

( )   (( ))   (( ))( )

CC Construction: cc_0 Step 1

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

(0) [Goal $\xrightarrow{\checkmark}$ • List, eof]   initial parser state

Hint 1. How is $[\underset{Goal}{A} \rightarrow \underset{\varepsilon}{\beta} \bullet \underset{List}{C} \underset{\varepsilon}{\delta}, \underset{eof}{a}]$ instantiated?

Hint 2. What are $C \rightarrow \gamma \in R$?

→List → List Pair   List → Pair ✓

Hint 3. $\text{FIRST}(\delta a) = \text{FIRST}(\varepsilon\ eof) = \text{FIRST}(eof) = \{eof\}$

Two new LR(1) items:

1. [List → • List Pair, eof]
2. [List → • Pair, eof]

How should s be extended?

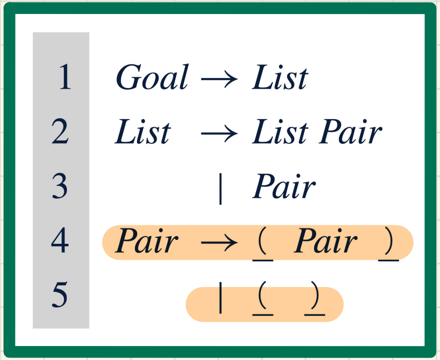
```
for [A → ⋯ • C δ, a] ∈ s:
  for C → γ ∈ R:
    for b ∈ FIRST(δa):
      s := s ∪ { [C → •γ, b] }
```

$cc_0 = \{$

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

$\}$

CC Construction: CC_0 Step 2



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

(0) [Goal → • List, eof]

(1) [List → • List Pair, eof]

(2) [List → • Pair, eof]

Hint 1. How is $[\overset{List}{A} \to \overset{\varepsilon}{\beta} \bullet \overset{Pair}{C}\ \overset{\varepsilon}{\delta}, \overset{eof}{a}]$ instantiated?

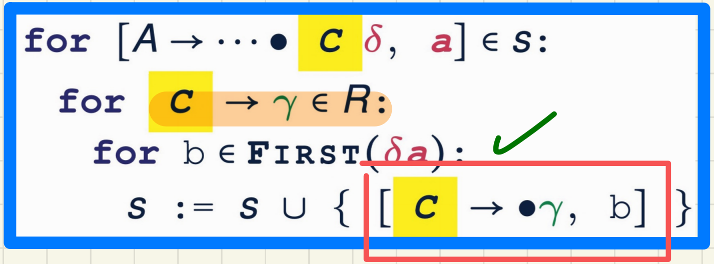
Hint 2. What are $C \to \gamma \in R$?   Pair → (Pair)   Pair → ( )

Hint 3. FIRST(δa) = FIRST(ε eof) = FIRST(eof) = { eof }

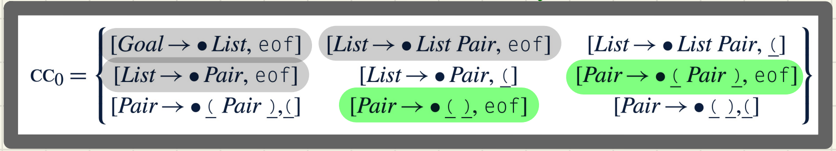
How should s be extended?

[Pair → • (Pair) , eof]

[Pair → • ( ) , eof]



```
for [A → ⋯ • C δ, a] ∈ s:
    for C → γ ∈ R:
        for b ∈ FIRST(δa):   ✓
            s := s ∪ { [C → •γ, b] }
```



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

CC Construction: CC_0 Step 3

(0) [Goal → • List, eof]

(1) [List → • List Pair, eof]

(2) [List → • Pair, eof]

(3) [Pair → • (Pair), eof]

(4) [Pair → • ( ), eof]

Hint 1. How is $[\overset{List}{A} \rightarrow \overset{\varepsilon}{\beta} \bullet \overset{List}{C}\, \overset{Pair}{\delta}, \underset{eof}{a}]$ instantiated?

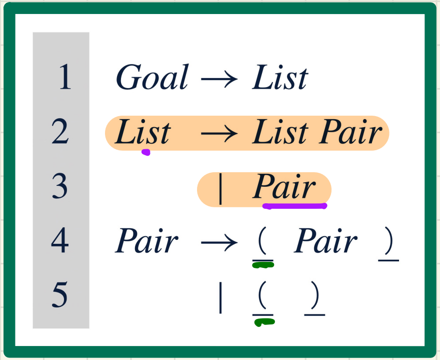
Hint 2. What are $C \rightarrow \gamma \in R$? $\varepsilon \notin FIRST(Pair)$ -

Hint 3. $\mathbf{FIRST}(\delta a)$ = $FIRST(Pair\ eof) = \{ (\}$

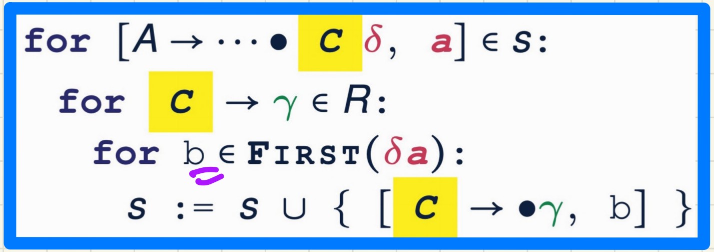
How should s be extended?

$[List \rightarrow \bullet\ List\ Pair, ( ]$

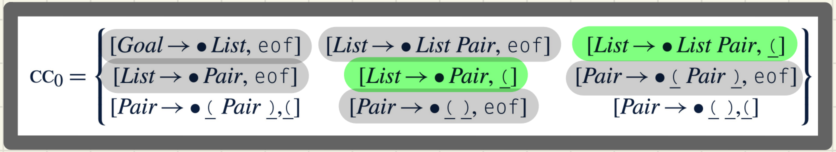
$[List \rightarrow \bullet\ Pair, ( ]$



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



```
for [A → ··· • C δ, a] ∈ s:
  for C → γ ∈ R:
    for b ∈ FIRST(δa):
      s := s ∪ { [C → •γ, b] }
```



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

CC Construction: CC_0 Step 4

(0) [Goal → • List, eof]

(1) [List → • List Pair, eof]

(2) [List → • Pair, eof]

(3) [Pair → • (Pair), eof]

(4) [Pair → • ( ), eof]

(5) [List → • List Pair, ( ]

(6) [List → • Pair, ( ]

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

- **Hint 1.** How is $[\overset{List}{A} \to \overset{\varepsilon}{\beta} \bullet \overset{Pair}{C}\ \overset{\varepsilon}{\delta}, \overset{(}{a}]$ instantiated?
- **Hint 2.** What are $C \to \gamma \in R$?
- **Hint 3.** $\text{FIRST}(\delta a)$ = $\text{FIRST}(\varepsilon\ () = \{ (\}$ =

How should s be extended?

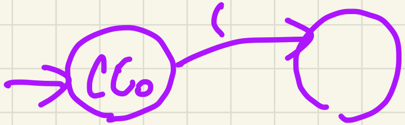
```
for $[A \to \cdots \bullet C\ \delta, a] \in s$:
  for $C \to \gamma \in R$:
    for $b \in \text{FIRST}(\delta a)$:
      $s := s \cup \{ [C \to \bullet\gamma, b] \}$
```

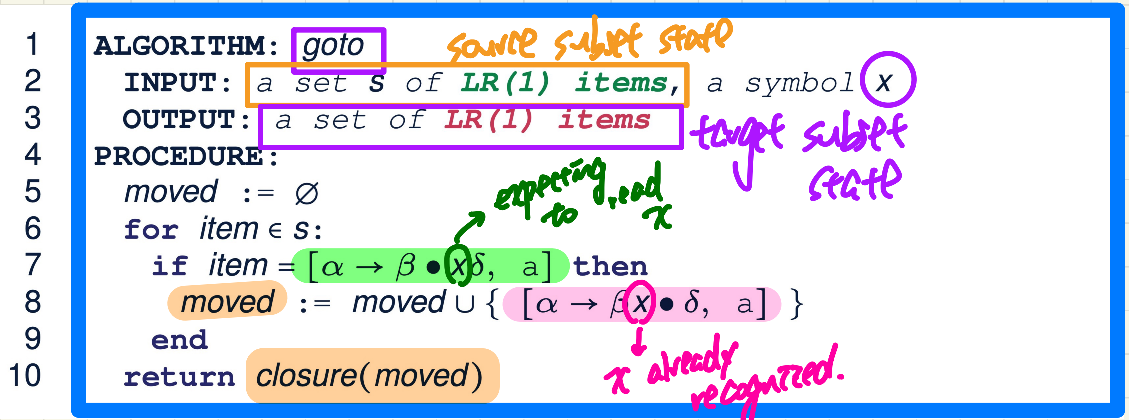
Two additional LR(1) items:

1. [Pair → • (Pair), (]
2. [Pair → • ( ), (]

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

CC Construction: goto





```
ALGORITHM: goto
  INPUT: a set s of LR(1) items, a symbol x
  OUTPUT: a set of LR(1) items
PROCEDURE:
  moved := ∅
  for item ∈ s:
    if item = [$\alpha \to \beta \bullet x \delta$, a] then
      moved := moved ∪ { [$\alpha \to \beta x \bullet \delta$, a] }
    end
  return closure(moved)
```

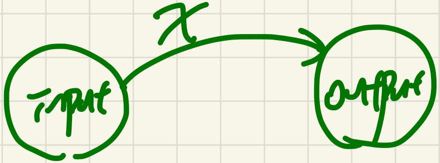
source subset state

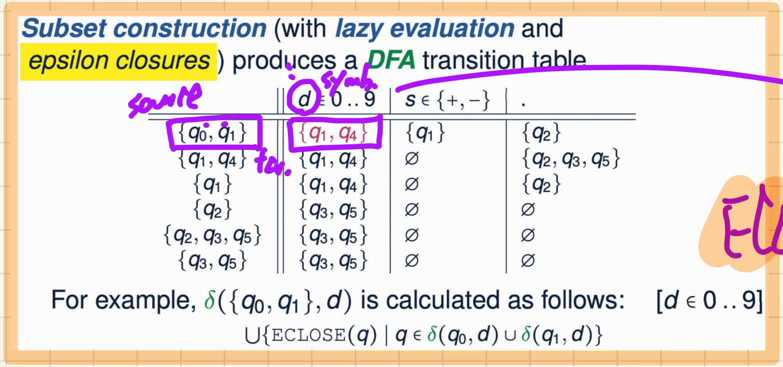
target subset state

expecting to read x

x already recognized.

Analogy: ε-NFA to DFA





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

source

symb

tar.

	$d \in 0..9$	$s \in \{+,-\}$	.
$\{q_0, q_1\}$	$\{q_1, q_4\}$	$\{q_1\}$	$\{q_2\}$
$\{q_1, q_4\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2, q_3, q_5\}$
$\{q_1\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2\}$
$\{q_2\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_2, q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$

For example, $\delta(\{q_0, q_1\}, d)$ is calculated as follows: $[d \in 0..9]$

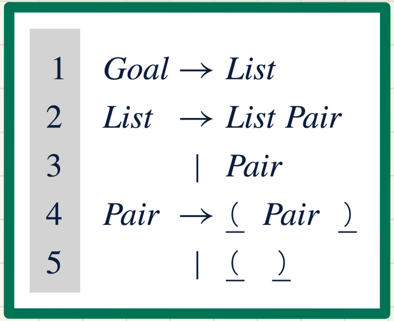
$$\bigcup\{\text{ECLOSE}(q) \mid q \in \delta(q_0, d) \cup \delta(q_1, d)\}$$

$$\text{ECLOSE}\left(\begin{array}{c}\delta(q_0, d)\\ \cup\\ \delta(q_1, d)\end{array}\right)$$

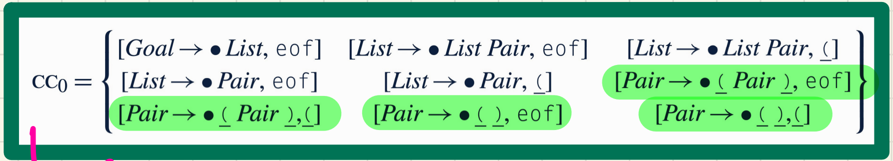
CC Construction: goto

Calculate goto(cc₀, ( )

i.e., “next subset state” from cc₀ taking (



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



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

[Pair → • (Pair), ( ]

[Pair → • ( ), eof]

[Pair → • (Pair), eof]

[Pair → • ( ), ( ]

will trigger additional items

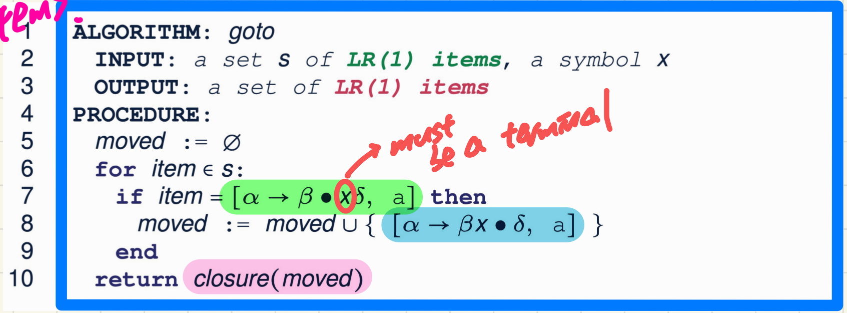
[Pair → (• Pair), ( ]

[Pair → (•), eof]

[Pair → (• Pair), eof]

[Pair → (•), ( ]

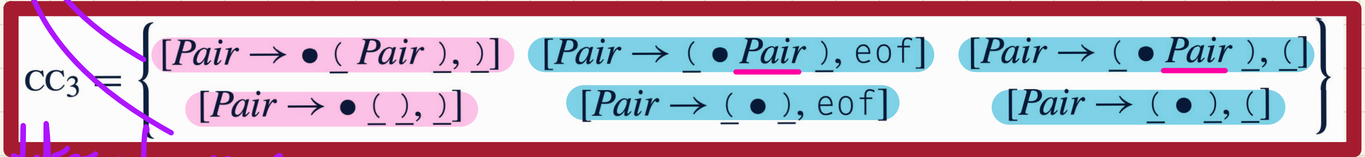
closure



```
ALGORITHM: goto
  INPUT: a set S of LR(1) items, a symbol x
  OUTPUT: a set of LR(1) items
PROCEDURE:
  moved := ∅
  for item ∈ S:
    if item = [$\alpha \rightarrow \beta \bullet x \delta$, a] then
      moved := moved ∪ { [$\alpha \rightarrow \beta x \bullet \delta$, a] }
    end
  return closure(moved)
```

must be a terminal

Exercise: why the highlighted items trigger the two additional items



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

Lecture 23 - Dec. 6

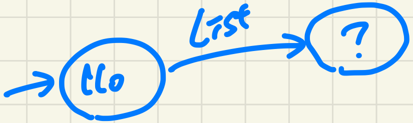
Syntactic Analysis

Algorithms: BuildCC, BuildTables
Conflicts: shift-reduce vs. reduce-reduce

Announcements

- **Project** final submission tonight!
- **Review session** at 1pm on Thursday, December 8

CC Construction: goto



Calculate goto(cc₀, List)

i.e., "next subset state" from cc₀ taking ~~x~~ List

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

closure({ [Goal → List • , eof], [List → List • Pair, eof], [List → List • Pair, ( ] })

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

Dimension 1: Two alt. for Pair

Pair → (Pair)

Pair → ( )

Dimension 2: FIRST(δa)

```
ALGORITHM: goto
  INPUT: a set S of LR(1) items, a symbol x
  OUTPUT: a set of LR(1) items
PROCEDURE:
  moved := ∅
  for item ∈ S:
    if item = [$\alpha \rightarrow \beta \bullet x\delta$, a] then
      moved := moved ∪ { [$\alpha \rightarrow \beta x \bullet \delta$, a] }
    end
  return closure(moved)
```

↳ 1. terminal
2. variable

List

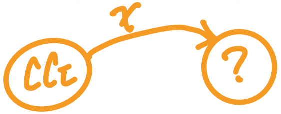
List

$CC_1 = \{ [Goal \rightarrow List \bullet, \texttt{eof}]\ [List \rightarrow List \bullet Pair, \texttt{eof}]\ [List \rightarrow List \bullet Pair, (]\ [Pair \rightarrow \bullet (Pair), \texttt{eof}]\ [Pair \rightarrow \bullet (Pair), (]\ [Pair \rightarrow \bullet ( ), \texttt{eof}]\ [Pair \rightarrow \bullet ( ), (] \}$

CC and δ Construction: Algorithm and Exercise

```
ALGORITHM: BuildCC
  INPUT: a grammar $G = (V, \Sigma, R, S)$, goal production $S \rightarrow S'$
  OUTPUT:
    (1) a set $CC = \{cc_0, cc_1, \ldots, cc_n\}$ where $cc_i \subseteq G'$s $LR(1)$ items
    (2) a transition function
PROCEDURE:
  $cc_0$ := closure({[$S' \rightarrow \bullet S$, eof]})
  $CC$ := $\{cc_0\}$
  processed := $\{cc_0\}$
  lastCC := $\varnothing$
  while (lastCC $\neq$ CC):
    lastCC := CC
    for $cc_i$ s.t. $cc_i \in CC \land cc_i \notin$ processed:
      processed := processed $\cup \{cc_i\}$
      for x s.t. $[\cdots \rightarrow \cdots \bullet x \ldots] \in cc_i$
        temp := goto($cc_i$, x)
        if temp $\notin$ CC then
          CC := CC $\cup$ {temp}
        end
        $\delta$ := $\delta \cup (cc_i, x, temp)$
```

start var.



make a transition from CC_i via recognizing x

ready to recognize a terminal or variable

src state

tgt state

transition

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

Ex1. Calculate CC (i.e., all reachable subset states).

Ex2. Calculate δ (i.e., relating members of CC by terminals and non-terminals).

CC and δ Construction: Output 1

$$\mathrm{CC}_0 = \left\{ \begin{array}{ccc} [\mathit{Goal} \rightarrow \bullet\, \mathit{List}, \texttt{eof}] & [\mathit{List} \rightarrow \bullet\, \mathit{List}\ \mathit{Pair}, \texttt{eof}] & [\mathit{List} \rightarrow \bullet\, \mathit{List}\ \mathit{Pair}, \underline{(}] \\ [\mathit{List} \rightarrow \bullet\, \mathit{Pair}, \texttt{eof}] & [\mathit{List} \rightarrow \bullet\, \mathit{Pair}, \underline{(}] & [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{Pair}\ \underline{)}, \texttt{eof}] \\ [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{Pair}\ \underline{)}, \underline{(}] & [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] & [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

List

$$\mathrm{CC}_1 = \left\{ \begin{array}{ccc} [\mathit{Goal} \rightarrow \mathit{List}\, \bullet, \texttt{eof}] & [\mathit{List} \rightarrow \mathit{List}\, \bullet\, \mathit{Pair}, \texttt{eof}] & [\mathit{List} \rightarrow \mathit{List}\, \bullet\, \mathit{Pair}, \underline{(}] \\ [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{Pair}\ \underline{)}, \texttt{eof}] & [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{Pair}\ \underline{)}, \underline{(}] & [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] \\ & [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] & \end{array} \right\}$$

$$\mathrm{CC}_2 = \left\{ [\mathit{List} \rightarrow \mathit{Pair}\, \bullet, \texttt{eof}] \quad [\mathit{List} \rightarrow \mathit{Pair}\, \bullet, \underline{(}] \right\}$$

(

$$\mathrm{CC}_3 = \left\{ \begin{array}{ccc} [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{Pair}\ \underline{)}, \underline{)}] & [\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \mathit{Pair}\ \underline{)}, \texttt{eof}] & [\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \mathit{Pair}\ \underline{)}, \underline{(}] \\ [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{)}] & [\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \underline{)}, \texttt{eof}] & [\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \underline{)}, \underline{(}] \end{array} \right\}$$

$$\mathrm{CC}_4 = \left\{ [\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\, \bullet, \texttt{eof}] \quad [\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\, \bullet, \underline{(}] \right\}$$

$$\mathrm{CC}_5 = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{Pair}\, \bullet\, \underline{)}, \texttt{eof}] \quad [\mathit{Pair} \rightarrow \underline{(}\ \mathit{Pair}\, \bullet\, \underline{)}, \underline{(}] \right\}$$

$$\mathrm{CC}_6 = \left\{ \begin{array}{cc} [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{Pair}\ \underline{)}, \underline{)}] & [\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \mathit{Pair}\ \underline{)}, \underline{)}] \\ [\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{)}] & [\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \underline{)}, \underline{)}] \end{array} \right\}$$

$$\mathrm{CC}_7 = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\, \bullet, \texttt{eof}] \quad [\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\, \bullet, \underline{(}] \right\}$$

$$\mathrm{CC}_8 = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{Pair}\ \underline{)}\, \bullet, \texttt{eof}] \quad [\mathit{Pair} \rightarrow \underline{(}\ \mathit{Pair}\ \underline{)}\, \bullet, \underline{(}] \right\}$$

$$\mathrm{CC}_9 = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{Pair}\, \bullet\, \underline{)}, \underline{)}] \right\}$$

$$\mathrm{CC}_{10} = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\, \bullet, \underline{)}] \right\}$$

$$\mathrm{CC}_{11} = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{Pair}\ \underline{)}\, \bullet, \underline{)}] \right\}$$

CC and δ Construction: Output 2

Transition Function

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}	∅	∅	∅	∅	∅	∅

DFA of the LR(1) Parser

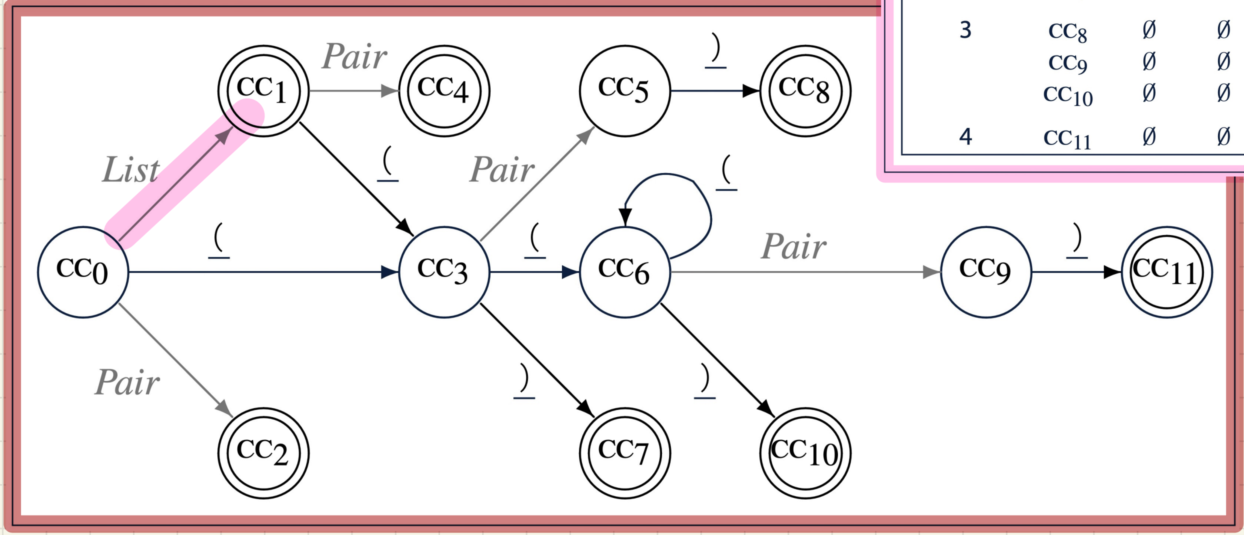


Table Construction: Algorithm

```
ALGORITHM: BuildActionGotoTables
  INPUT:
    (1) a grammar $G = (V, \Sigma, R, S)$
    (2) goal production $S \rightarrow S'$
    (3) a canonical collection $CC = \{cc_0, cc_1, \dots, cc_n\}$
    (4) a transition function $\delta : CC \times \Sigma \rightarrow CC$
  OUTPUT: Action Table & Goto Table
PROCEDURE:
  for $cc_i \in CC$:
    for item $\in cc_i$:
      if item $= [A \rightarrow \beta \bullet x\gamma,\ a] \wedge \delta(cc_i, x) = cc_j$ then
        Action[i, x] := shift j
      elseif item $= [A \rightarrow \beta \bullet,\ a]$ then
        Action[i, a] := reduce $A \rightarrow \beta$
      elseif item $= [S \rightarrow S' \bullet,\ eof]$ then
        Action[i, eof] := accept
      end
    for $v \in V$:
      if $\delta(cc_i, v) = cc_j$ then
        Goto[i, v] := j
      end
```

produced by BuildCC

$\delta(CC_3, () = CC_6$

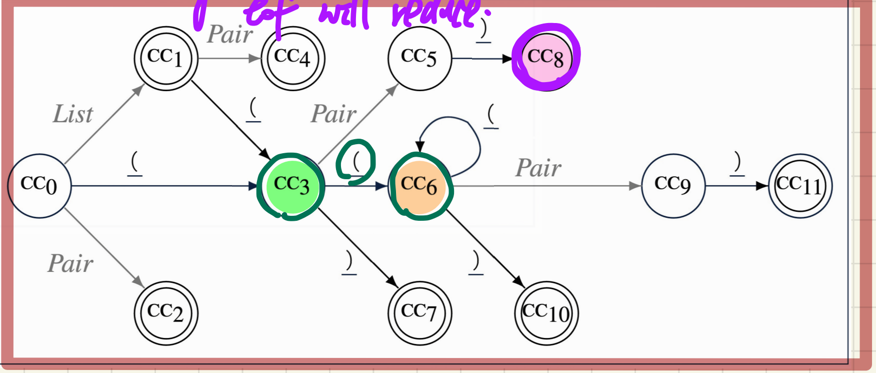
i j

3 r 6

CC8 is already an accepting state, meaning reading eof will reduce.

fill in Goto table

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		



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

Bottom-Up Parsing: Discovering Ambiguities

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

Certain state of paser

by reading eof or else, reduce to Stmt

by reading else, we shift to

What if the current word to match is else?

γδ already recognized

shift or reduce to Stmt

↳ shift-redure conflict

↳ in practice, shift will be done.

$$CC_i = \left\{ \begin{array}{l} [A \rightarrow \gamma\delta \bullet, \texttt{a}], \\ [B \rightarrow \gamma\delta \bullet, \texttt{a}] \end{array} \right\}$$

by readin a,

What if the current word to match is a?

some reduction

↳ redure-redure conflict → must fix the grammar.

Exam.

1. no multiple choice questions
2. no data sheets (algorithms included)
3. format similar to quizzes
4. cumulative.

That's all!

I hope you enjoyed the learning journey with me.

Best of luck with your future endeavours!

Jackie

Dec. 7, 2022