

# **CS 536 / Spring 2021**

Introduction to programming languages and compilers

Loris D'Antoni  
[loris-teach@cs.wisc.edu](mailto:loris-teach@cs.wisc.edu)

# About me

PhD at University of Pennsylvania

Joined University of Wisconsin in 2015

Research in

Program verification

Program synthesis

<http://pages.cs.wisc.edu/~loris/>

# About the course

We will study compilers

We will understand how they work

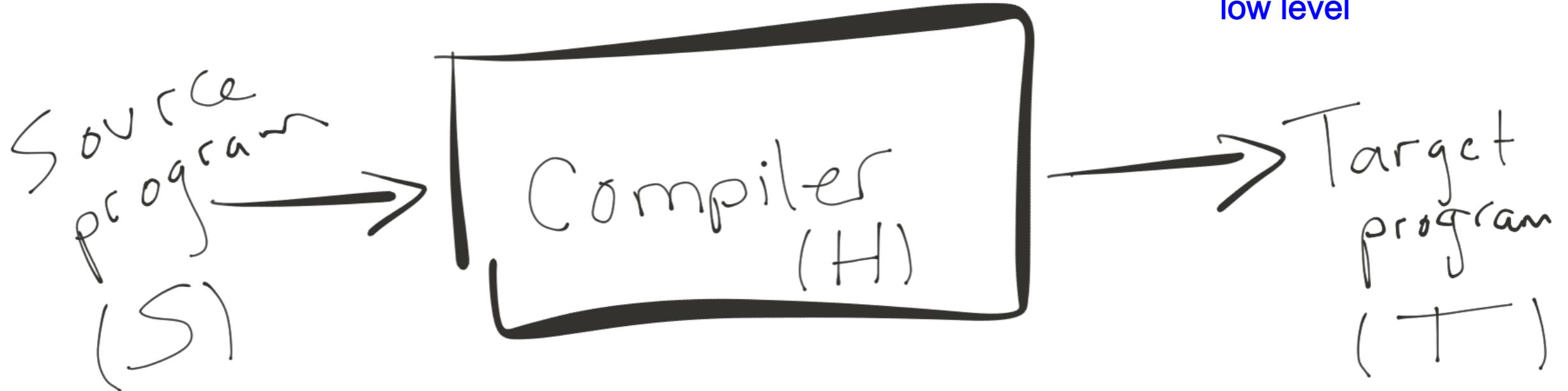
We will build a **full** compiler

We will have fun

# Course Mechanics

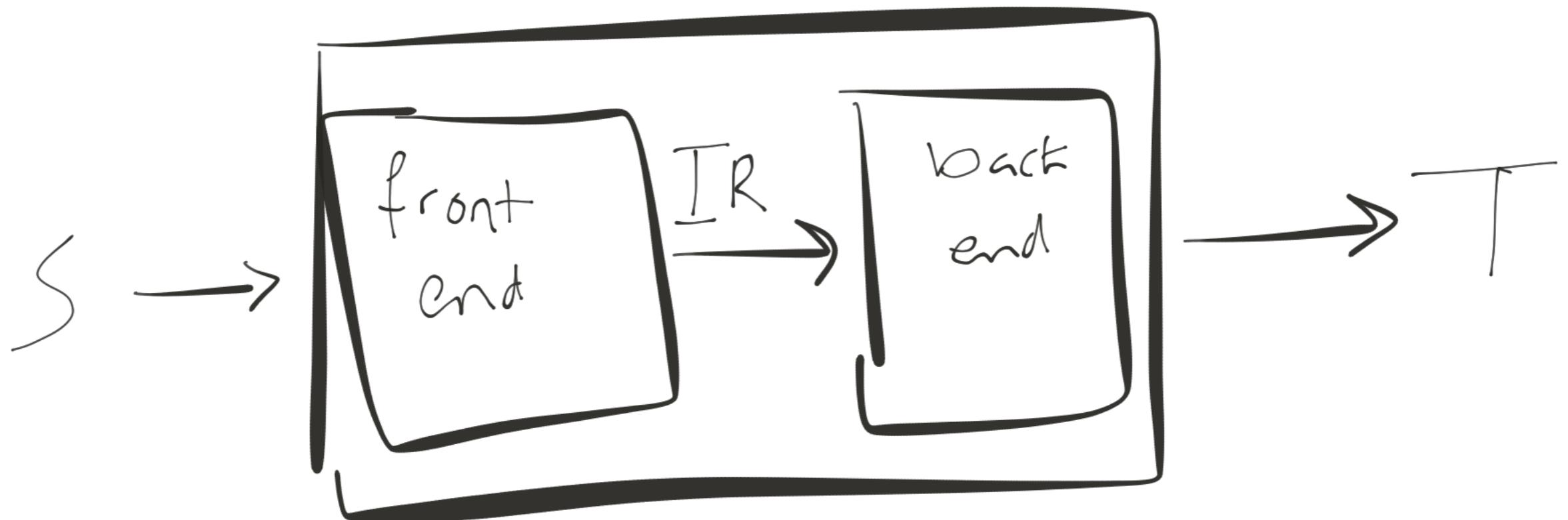
- Home page: <http://pages.cs.wisc.edu/~loris/cs536/>
- Piazza: <https://piazza.com/wisc/spring2021/cs536/>
- Workload:
  - 6 Programs ( $50\% = 5\% + 9\% + 9\% + 9\% + 9\% + 9\%$ )
  - 10 short homework problems (optional, not graded)
  - 2 online exams (midterm: 25% + final: 25%)
- For information about late policy, collaboration, etc., see <http://pages.cs.wisc.edu/~loris/cs536/info.html>

high level



A compiler is a  
recognizer of language S  
a translator from S to T  
a program in language H

What will we name S? **WUMBO**



**front end** = understand source code S

**IR** = intermediate representation

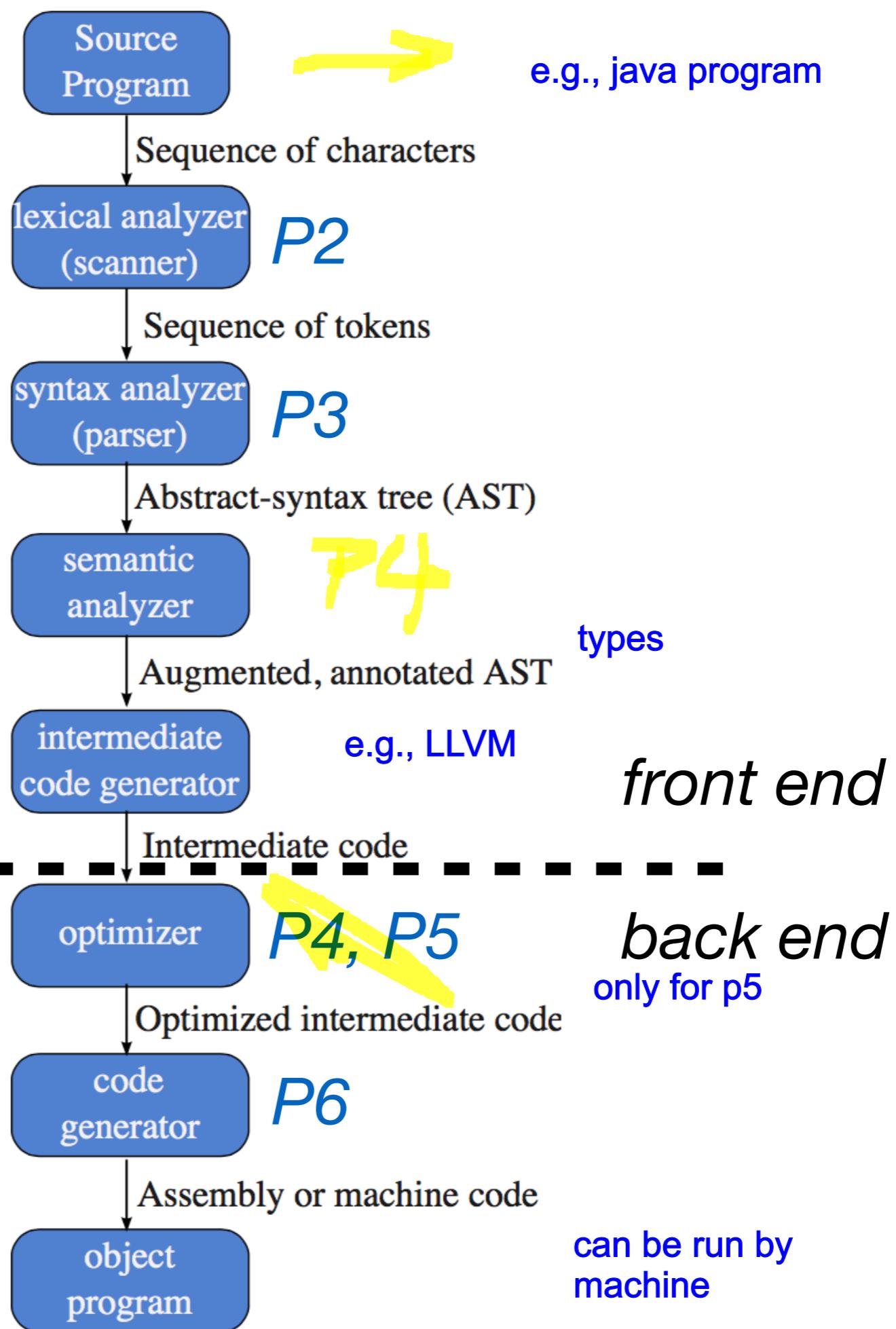
**back end** = map IR to T

# Phases of a compiler

P1

Symbol  
table

data struct that allow executor  
get the meaning of symbols



# Scanner

lexical analyzer

**Input:** characters from source program

**Output:** sequence of tokens

**Actions:**

group chars into lexemes (tokens)

Identify and ignore whitespace, comments, etc.

**What errors can it catch?**

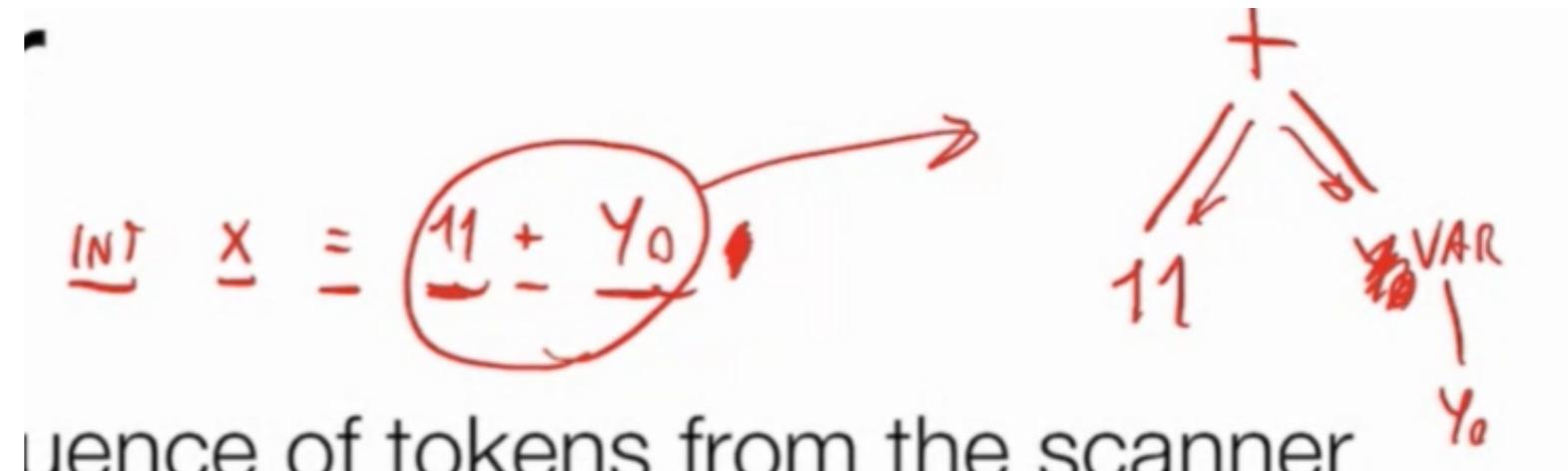
bad characters such as ^

unterminated strings, e.g., "Hello

int literals that are too large



# Parser



**Input:** sequence of tokens from the scanner

**Output:** AST (abstract syntax tree)

**Actions:**

groups tokens into sentences

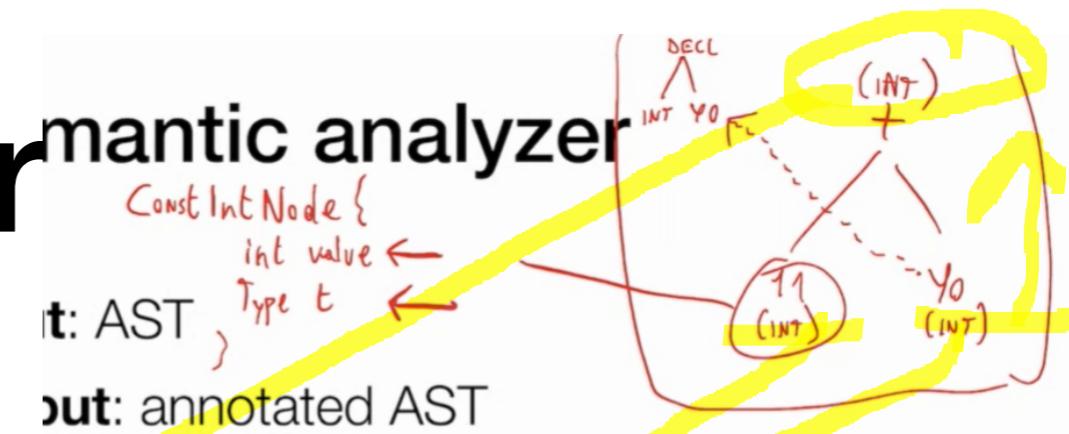
抓句式, 语法错误

**What errors can it catch?**

syntax errors, e.g., X = y \* = 5    ill-formed sentence

(possibly) static semantic errors, e.g., use of undeclared variables

# Semantic analyzer



**Input:** AST

**Output:** annotated AST

**Actions:** does more static semantic checks

Name analysis

*process declarations and uses of variables*

P4

enforces scope

Type checking

*checks types*

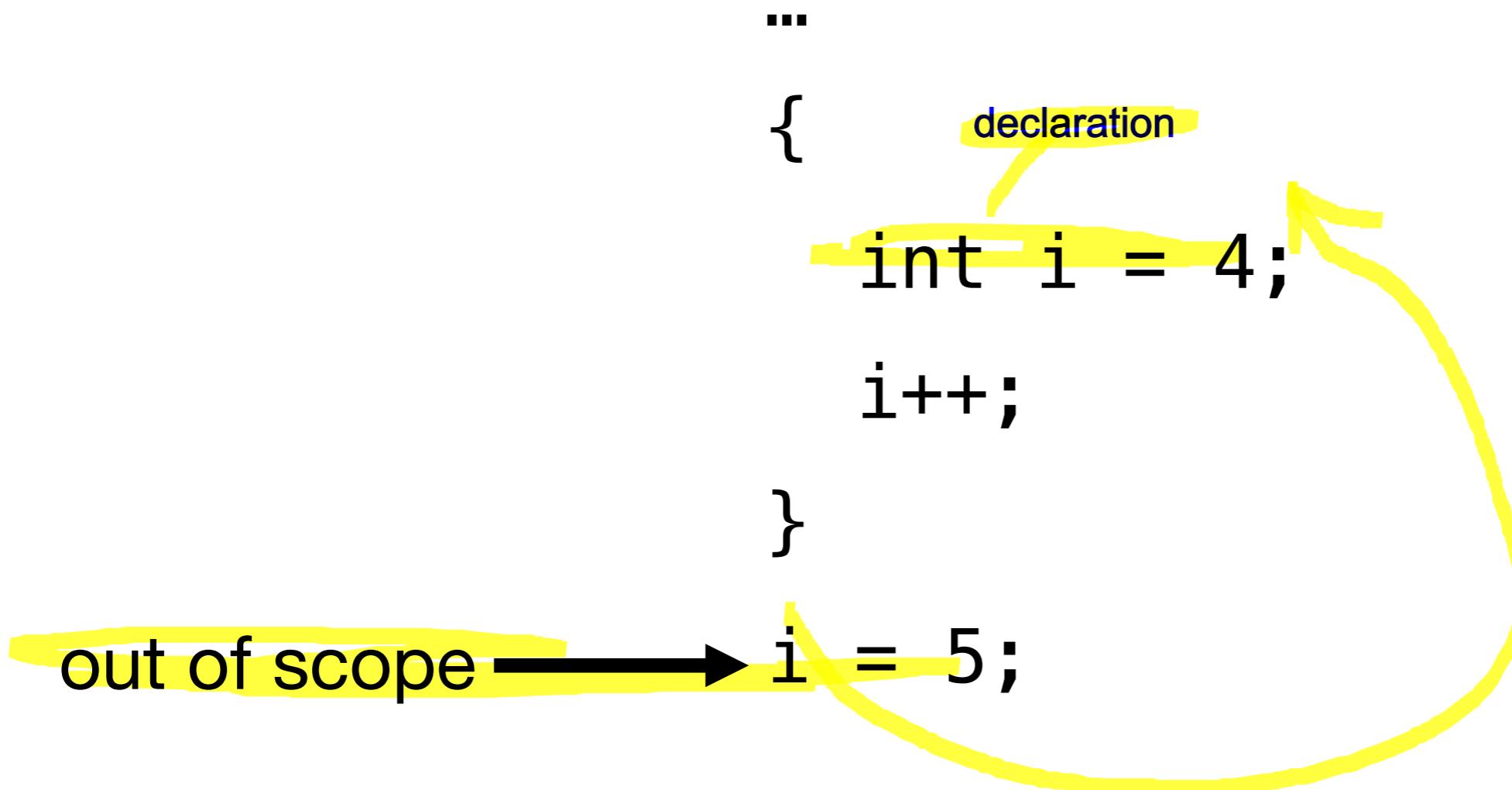
P5

*augments AST w/ types*

# Semantic analyzer

static-> compile time  
dynamic->run time  
e.g., divide by 0, nullptr exception,  
stackoverflow  
the whole dynamic is called  
"verification"

Scope example:



# Intermediate code generation

sth. in between ast and assembly, but still have space to optimize, not in final state

static errors

**Input:** annotated AST (assumes no errors)

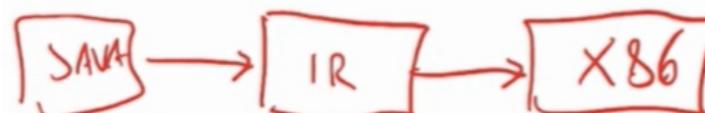
**Output:** intermediate representation (IR)

e.g., 3-address code

instructions have 3 operands at most

easy to generate from AST

1 instr per AST internal node      tr per AST internal node

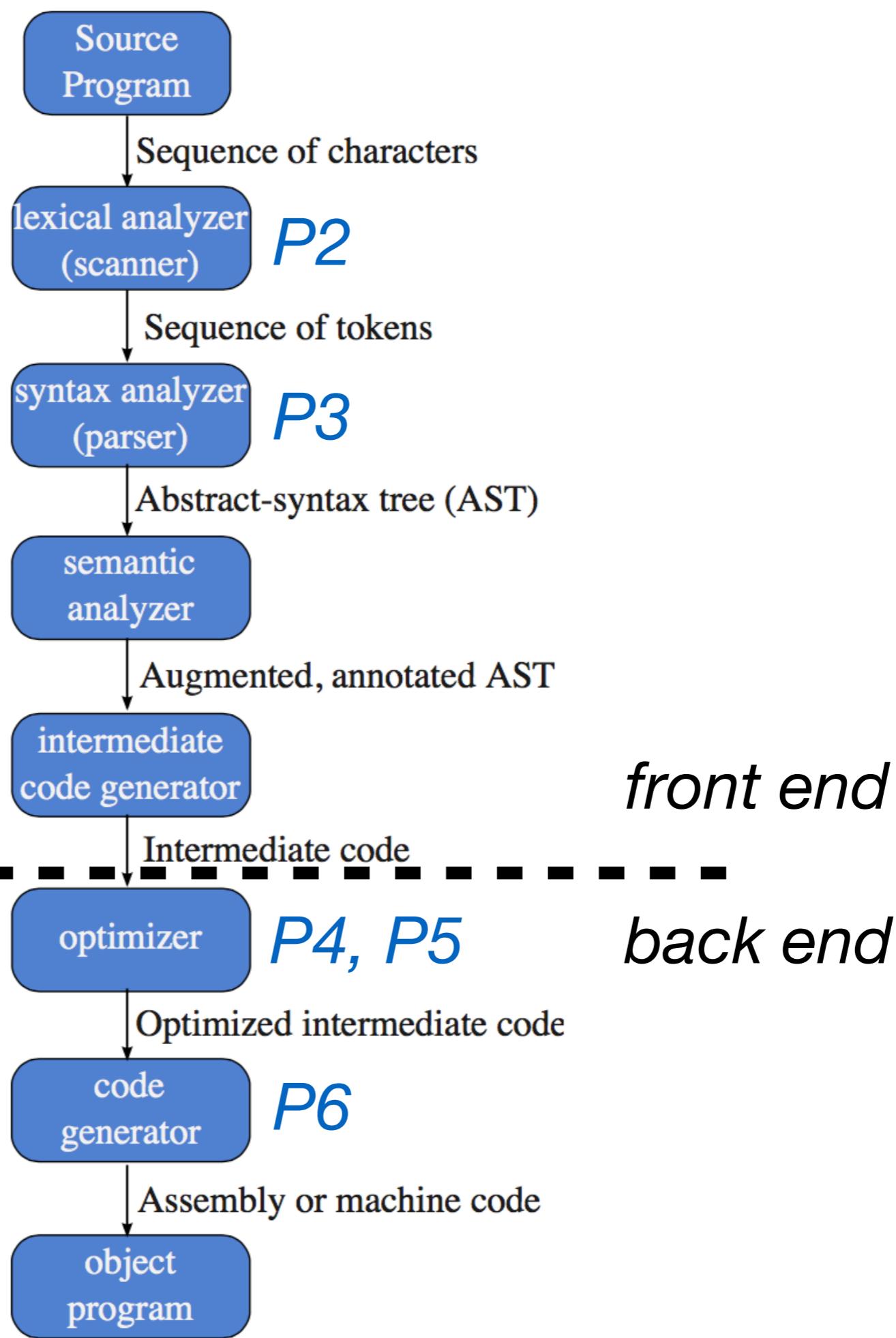


$$\begin{aligned} X &= 2 * Y + 6 * 8 \\ \textcircled{V}_1 &= 2 * Y \quad \leftarrow \text{IR} \\ \textcircled{T}_2 &= 6 * 8 \\ \textcircled{V}_3 &= \textcircled{V}_1 + \textcircled{T}_2 \end{aligned}$$

# Phases of a compiler

*P1*

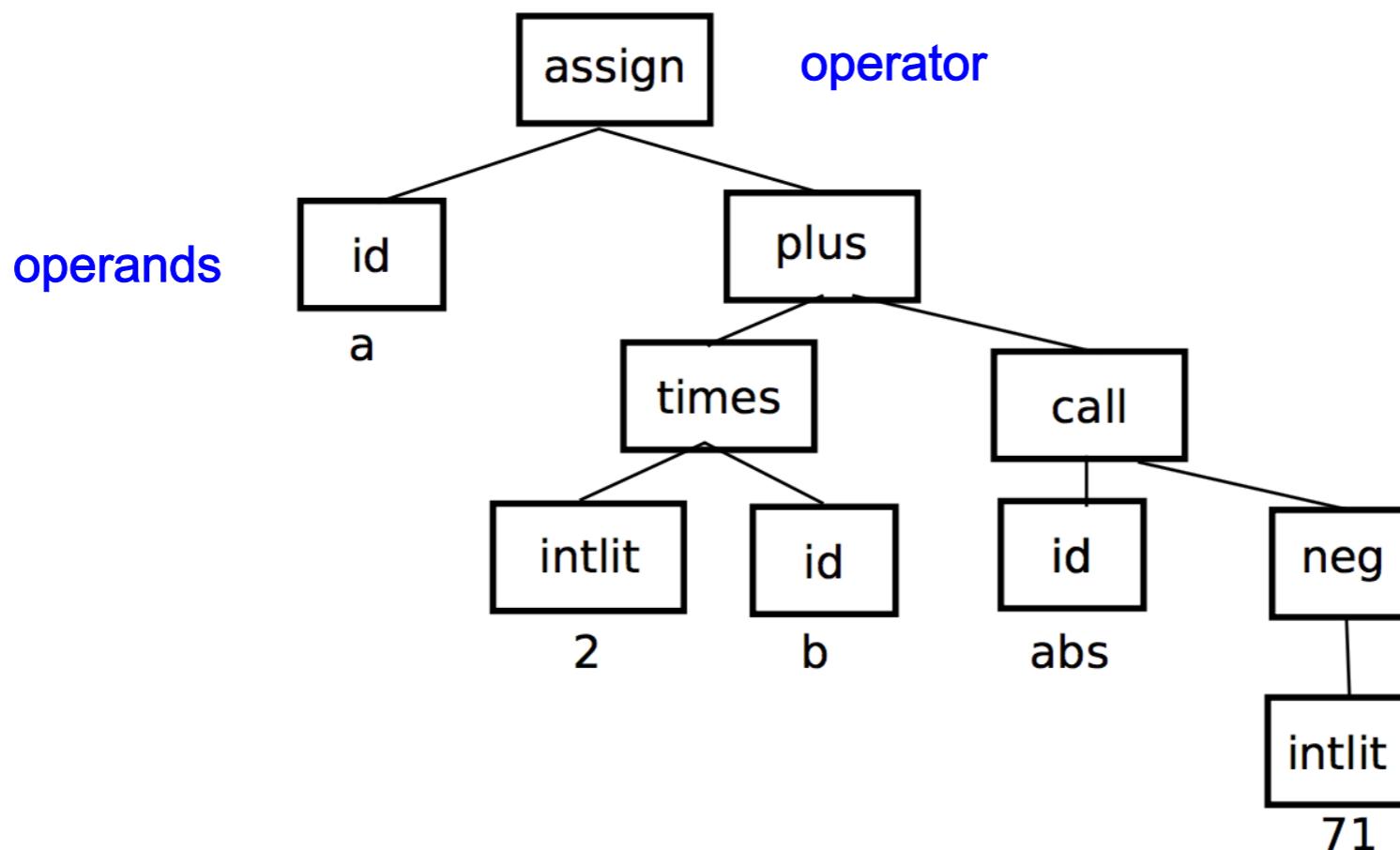
Symbol  
table



# Example

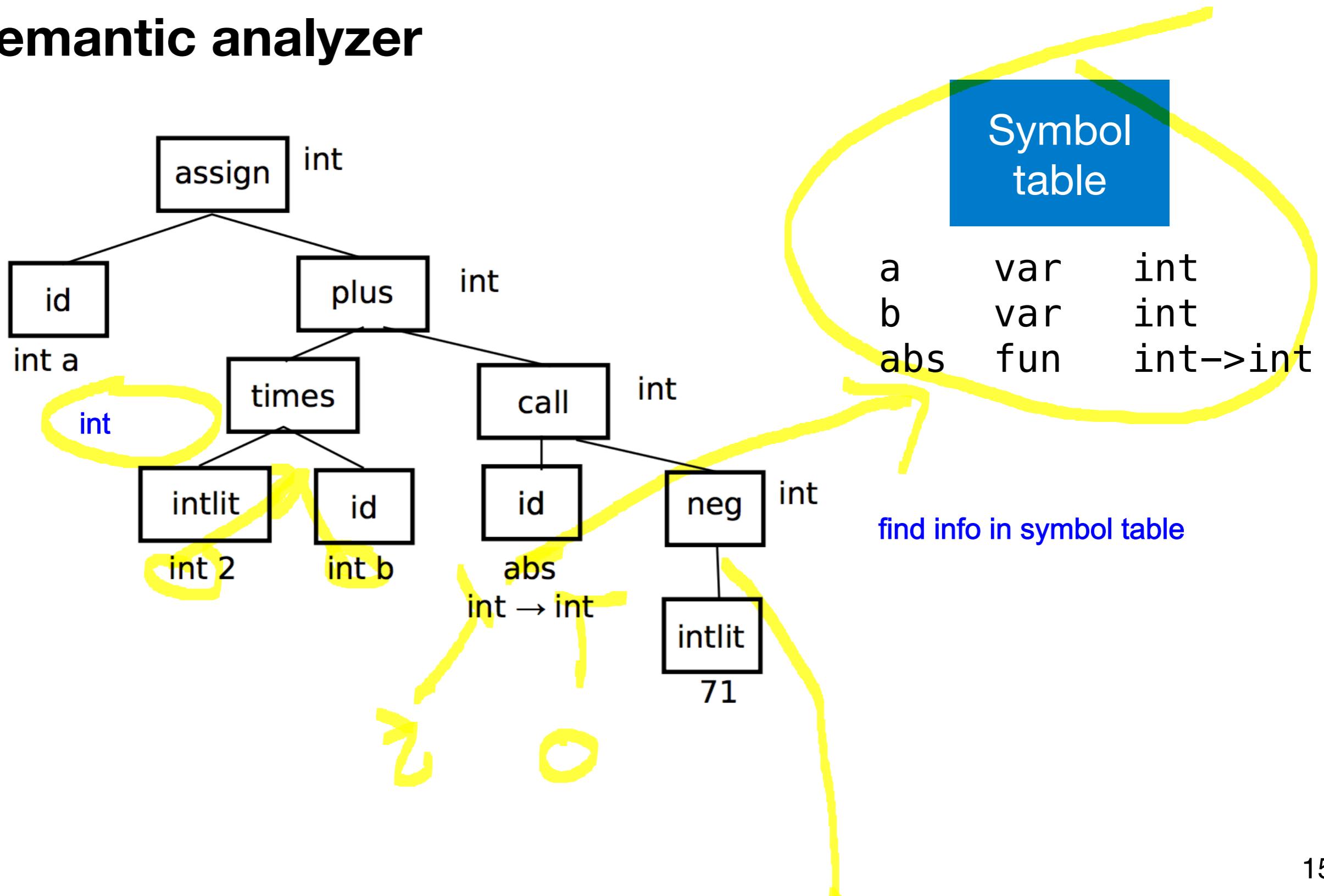
scanner *text*  
list<token> a = 2 \* b + abs( -71 )  
ident (a) asgn int lit (2) times ident (b) plus ident (abs) lparens minus int lit (71) rparens

## parser



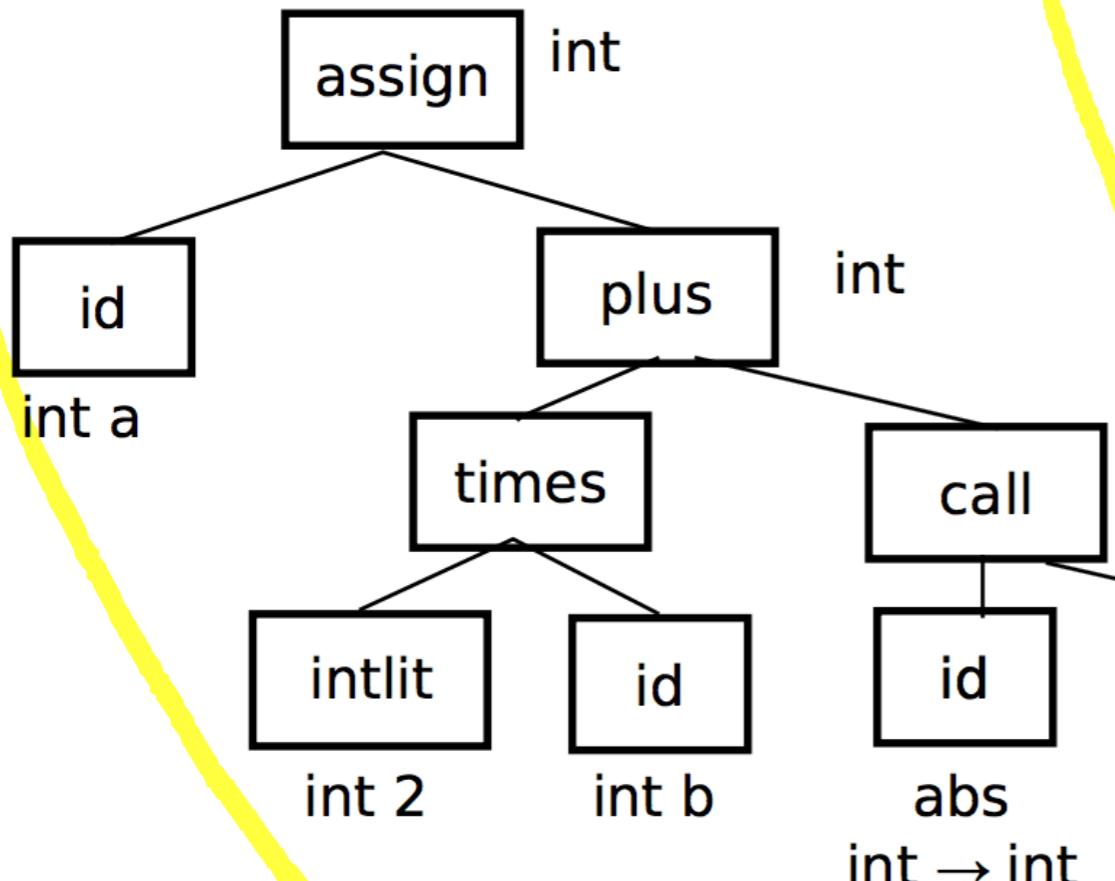
# Example (cont'd)

## semantic analyzer



# Example (cont'd)

## code generation



IR

still not assembly, cannot run by machine

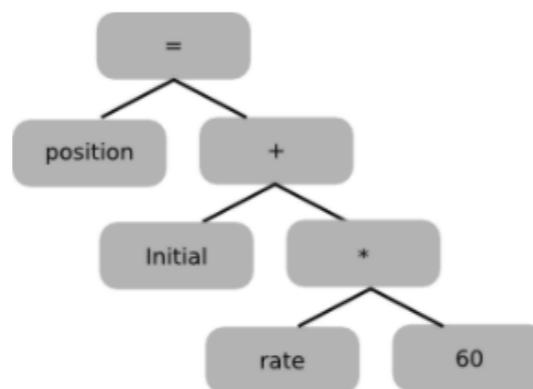
```
tmp1 = 0 - 71  
move tmp1 param1  
call abs  
move ret1 tmp2  
tmp3 = 2*b  
tmp4 = tmp3 + tmp2  
a = tmp4
```

x86

# Optimizer

source code: position = initial + rate \* 60 ;

Abstract syntax tree:



**Input:** IR

**Output:** optimized IR

**Actions:** Improve code

make it run faster; make it smaller      *make O(time) O(space) smaller*



several passes: local and global optimization

more time spent in compilation; less time in execution

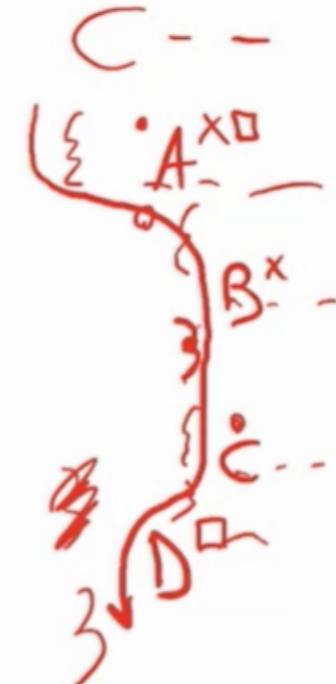
# Code generator

**Input:** IR from optimizer

**Output:** target code

# Symbol table

每个大括号对应一个scope



Compiler keeps track of names in  
semantic analyzer – both name analysis and type checking  
code generation – offsets into stack  
optimizer – def-use info

P1: implement symbol table

# Symbol table

## Block-structured language

java, c, c++

Ideas:

*nested visibility of names (no access to a variable out of scope)*

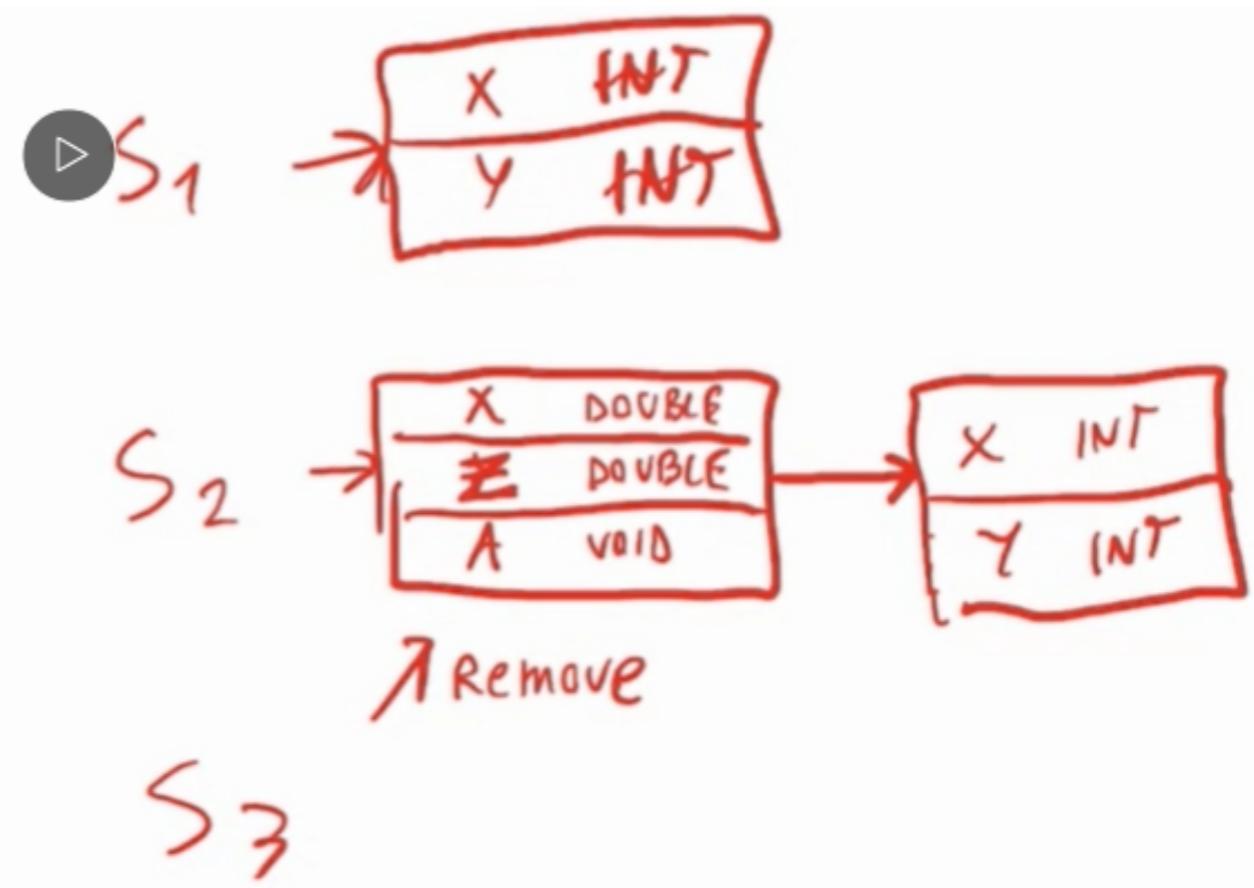
*easy to tell which def of a name applies (nearest definition)*

*lifetime of data is bound to scope*

# Symbol table

```
S1
int x, y;
void A() {
    double x, z;
    C(x, y, z)
}
void B() {
    C(x, y, z);
}
```

**block structure:** need  
symbol table with nesting  
*implement as list of hashtables*



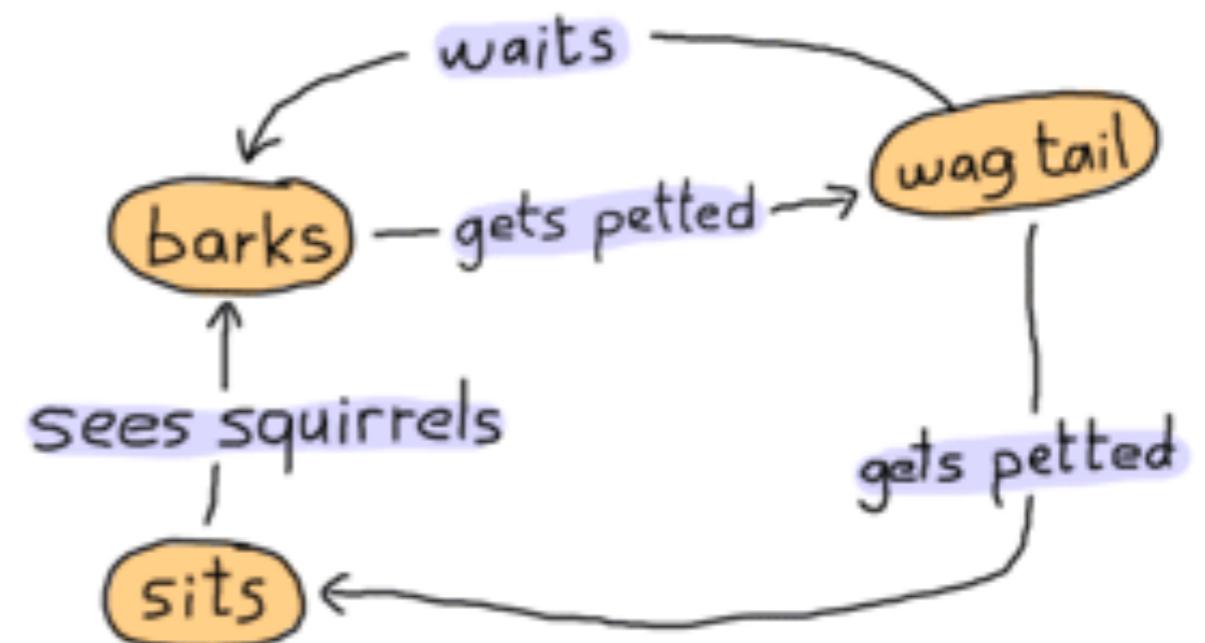
# **Announcements**

**P1 part 1 due next Tuesday**

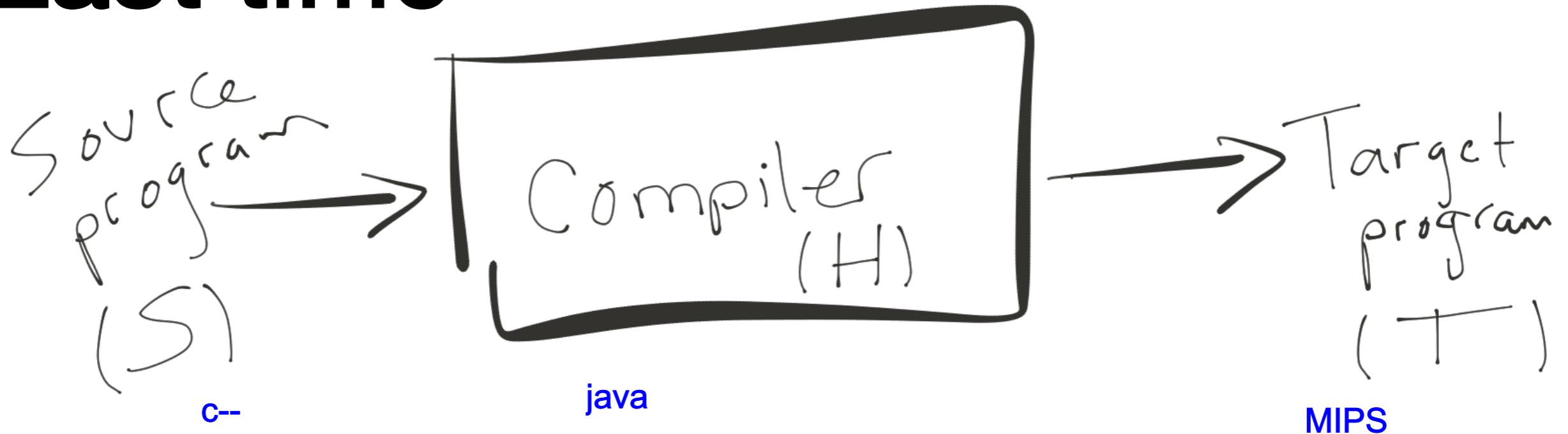
**P1 part 2 due next Friday**

# Finite-state machines

CS 536



# Last time



A compiler is a

recognizer of language  $S$  (Source)

a translator from  $S$  to  $T$  (Target)

a program in language  $H$  (Host)

For example, gcc:  $S$  is C,  $T$  is x86,  $H$  is C

# Last time

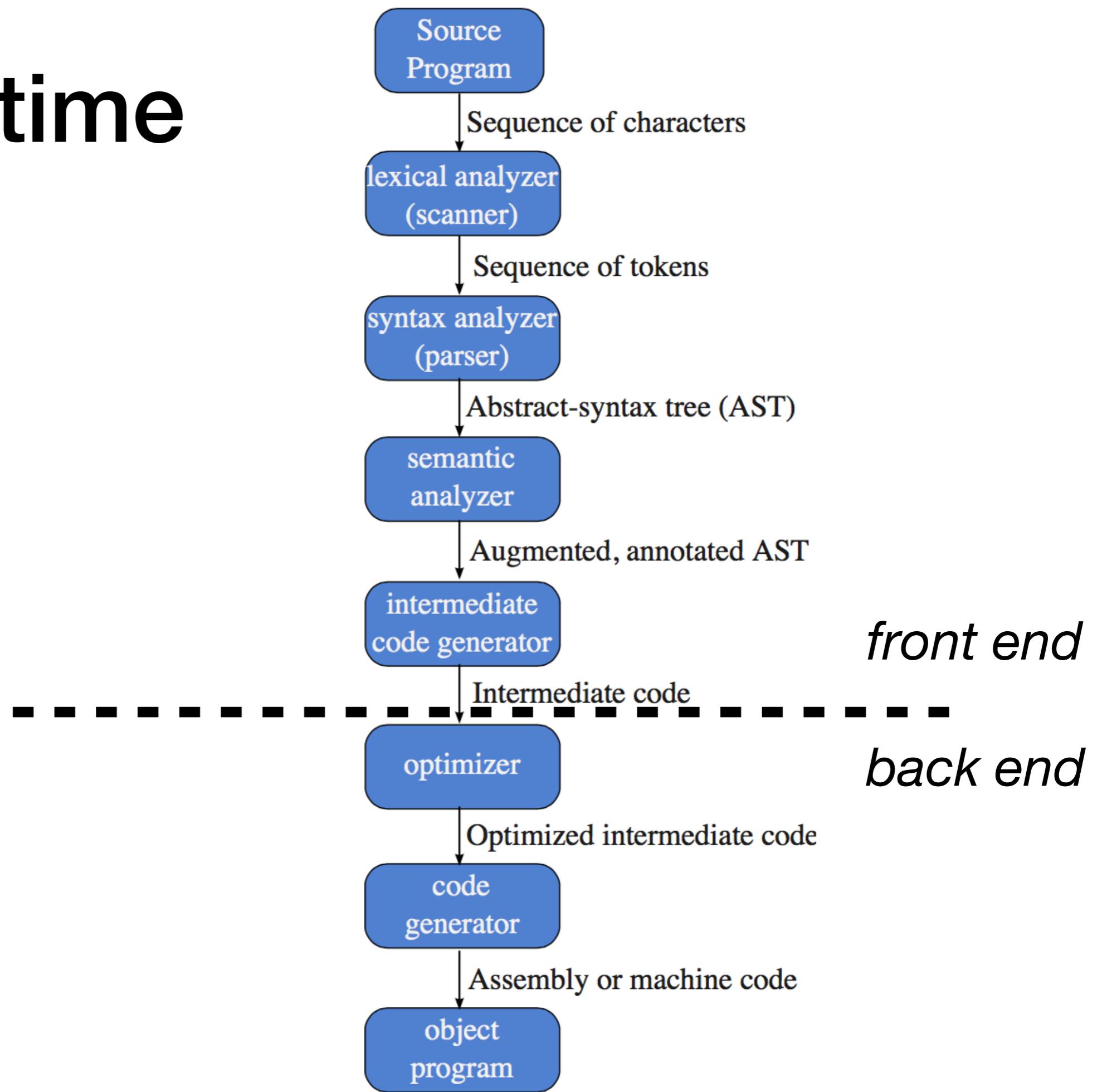
Why do we need a compiler?

- Processors can execute only binaries  
(machine-code/assembly programs)
- Writing assembly programs will make you want to reconsider your life choices
- Write programs in a nice(ish) high-level language like Java; compile to binaries

low-level:  
e.g., x86,  
MIPS  
RISC-v

high-level:  
java, C...  
enables  
loops, funcs

# Last time



# The scanner

Translates sequence of chars into sequence of tokens

Each time scanner is called it should:

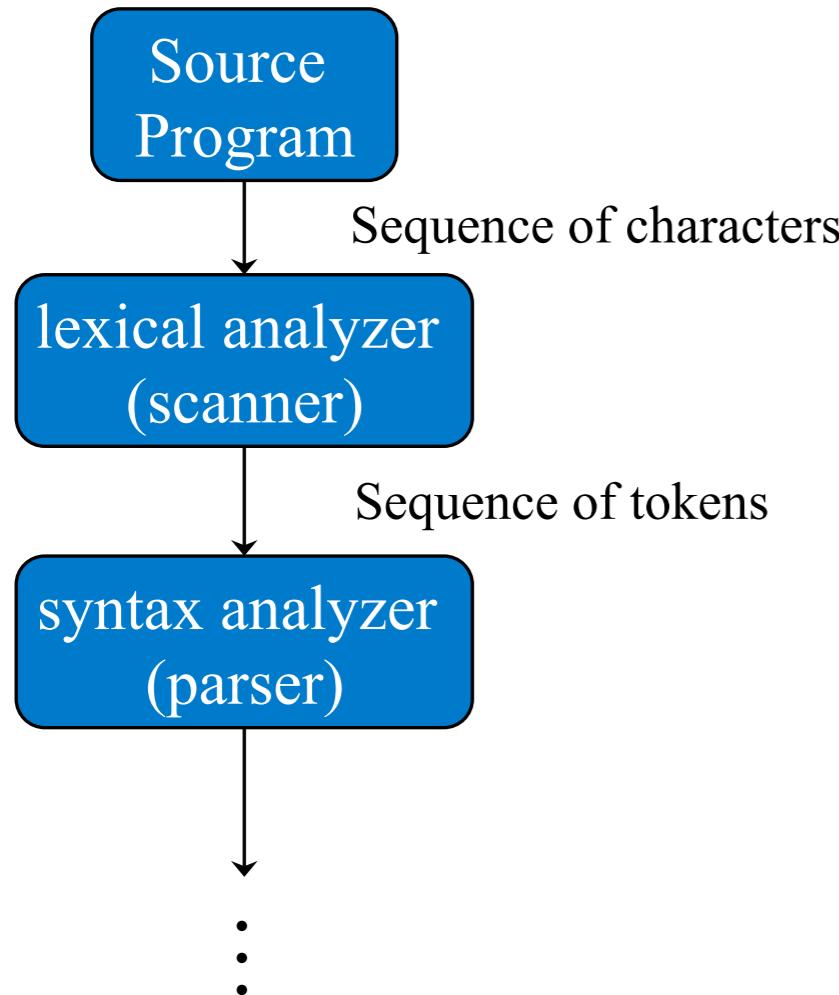
find longest sequence of chars corresponding to a token

e.g, we need ++, not +; again , we need 1000, not 1,0,0,0  
1,0,0,0 are 4 tokens, 1000 is 1 token

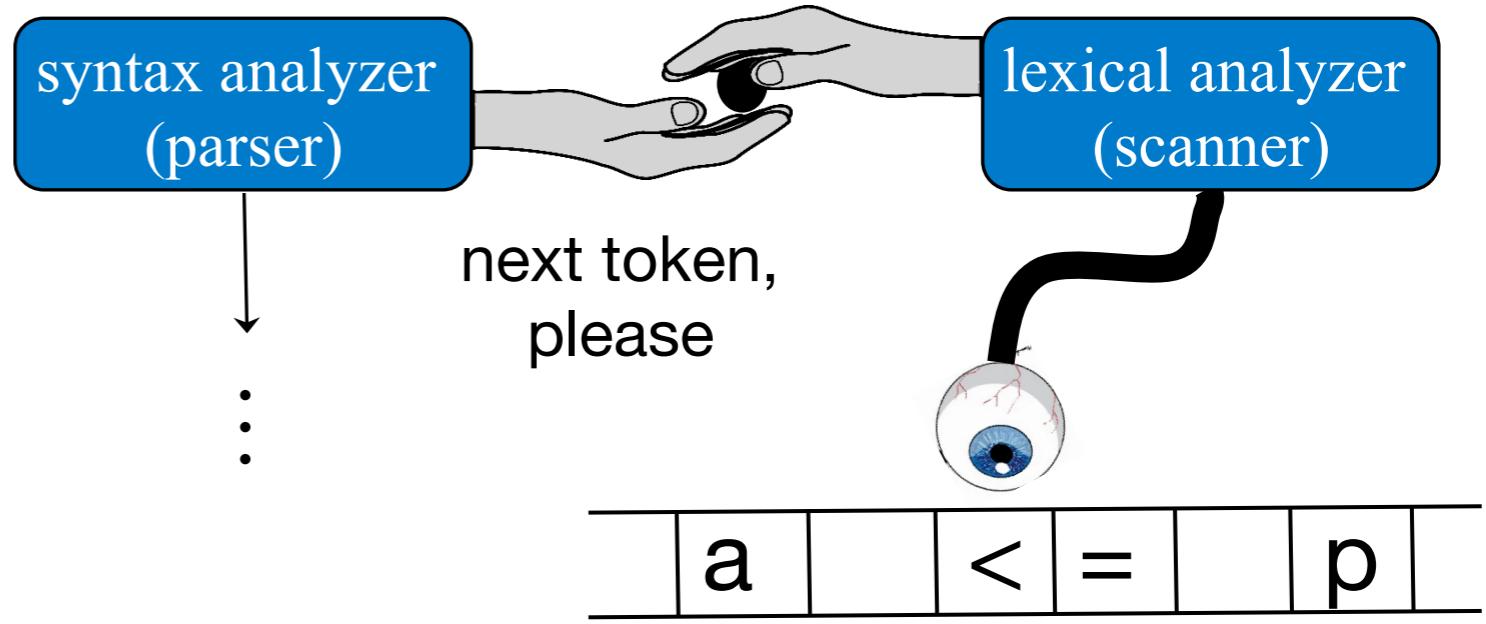
return that token



# Special linkage between scanner and parser in most compilers

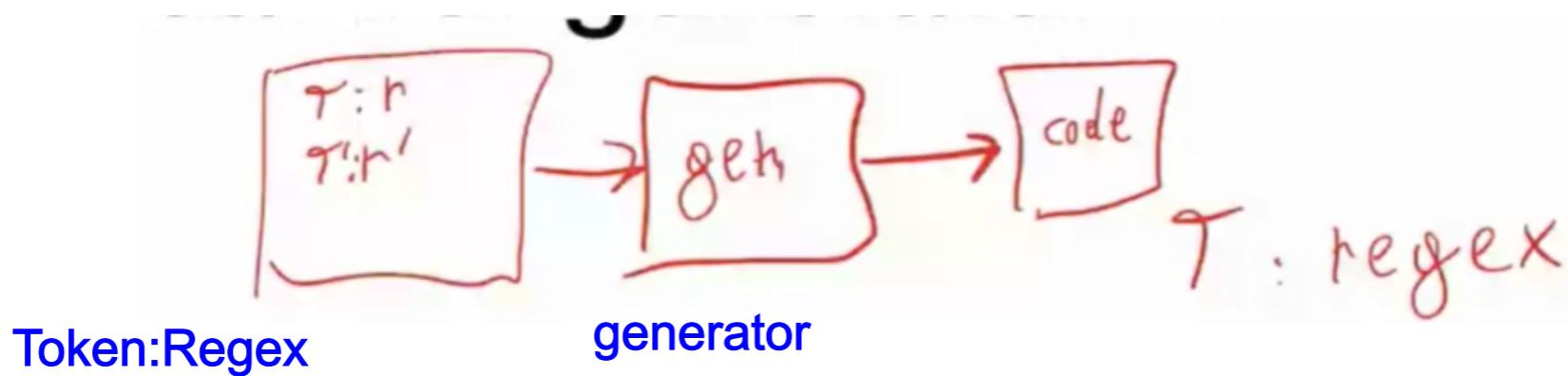


give token one by one at a time;  
reason : parser can stop executing when it finds an error



Conceptual organization

# Scanner generator



Generates a scanner!!!

Needs one regular expression for each token

Needs regular expressions for things to ignore  
comments, whitespace, etc.

To understand how it works, we need FSMs  
finite state machines

# FSMs: Finite State Machines

Aka finite automata

**Input:** string (seq of chars)

**Output:** accept / reject

i.e., input is legal in language

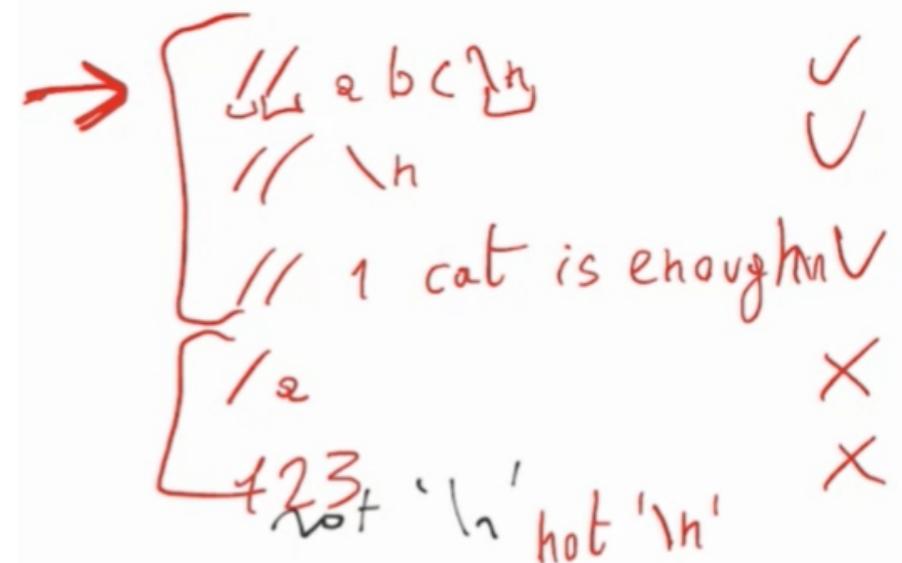
# FSMs

Represent regular languages

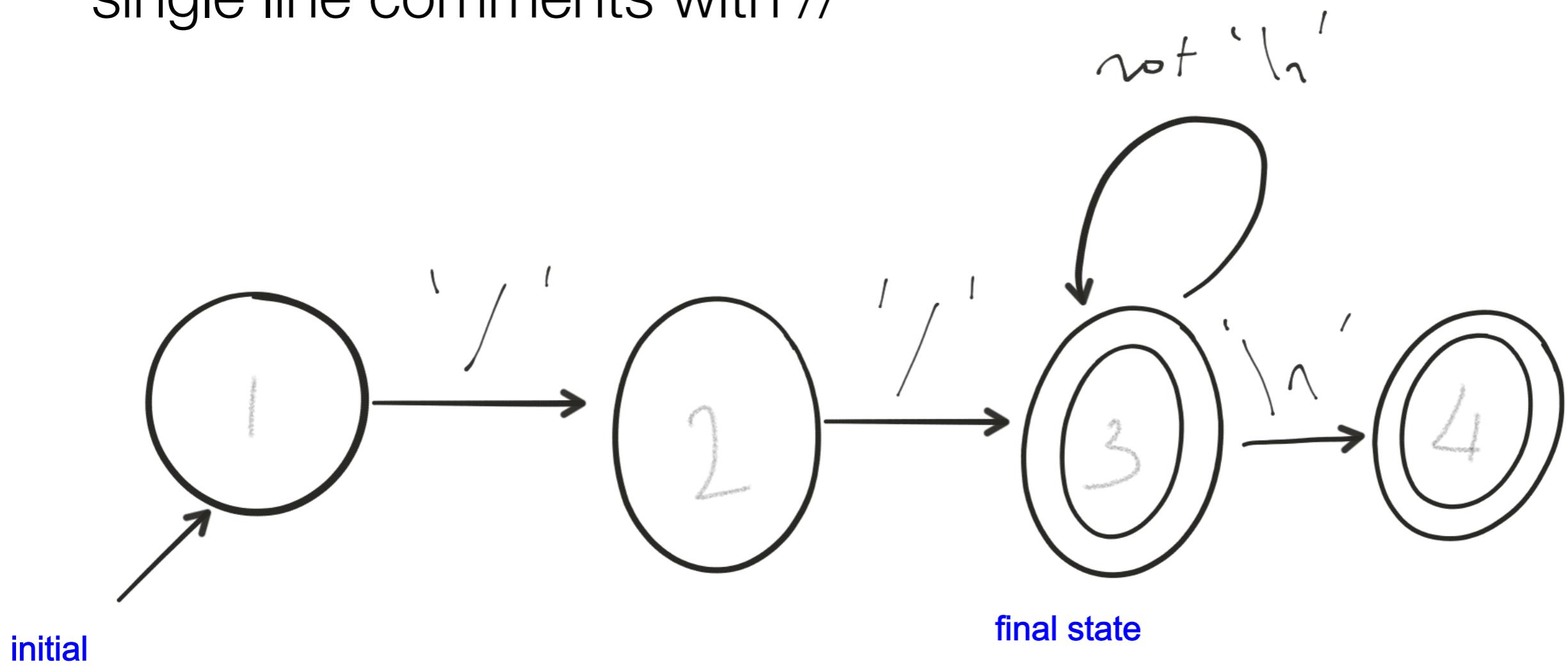
Good enough for tokens in PLs

# Example 1

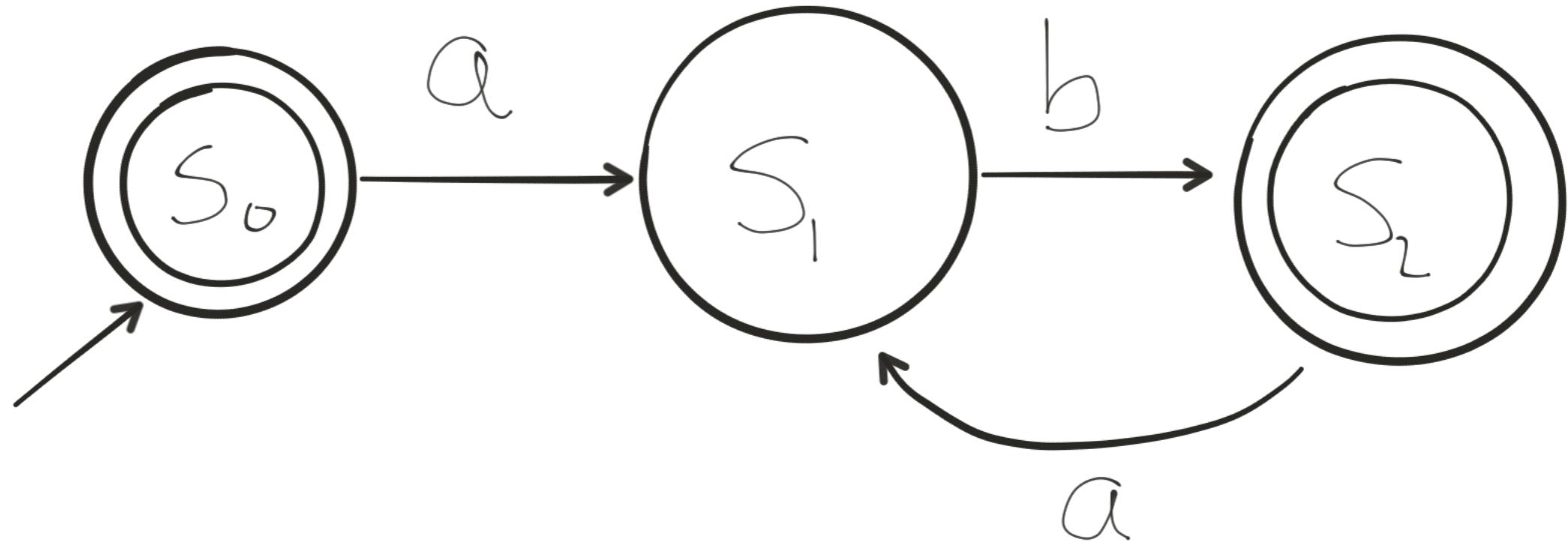
用regex&finite state machine去判断这句instruction是否以//开头



single line comments with //



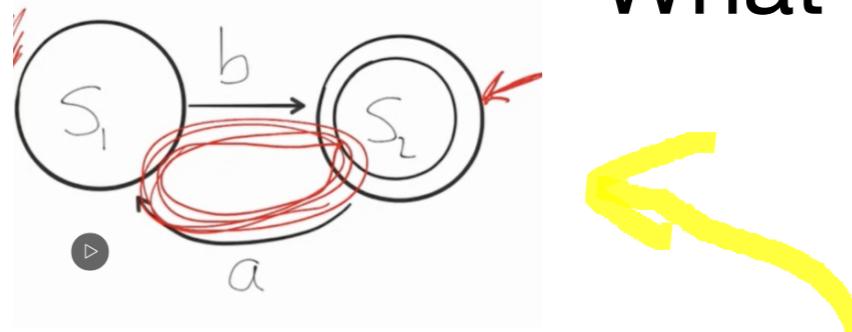
# Example 2



What language does this accept?

$(a \cup b)^n$

$n \geq 0$



Can you find an equivalent, but smaller, FSM?

# How an FSM works

curr\_state = start\_state

**let** in\_ch = current input char

**repeat**

**if** there is edge out of curr\_state with  
    label in\_ch into next\_state

        cur\_state = next\_state

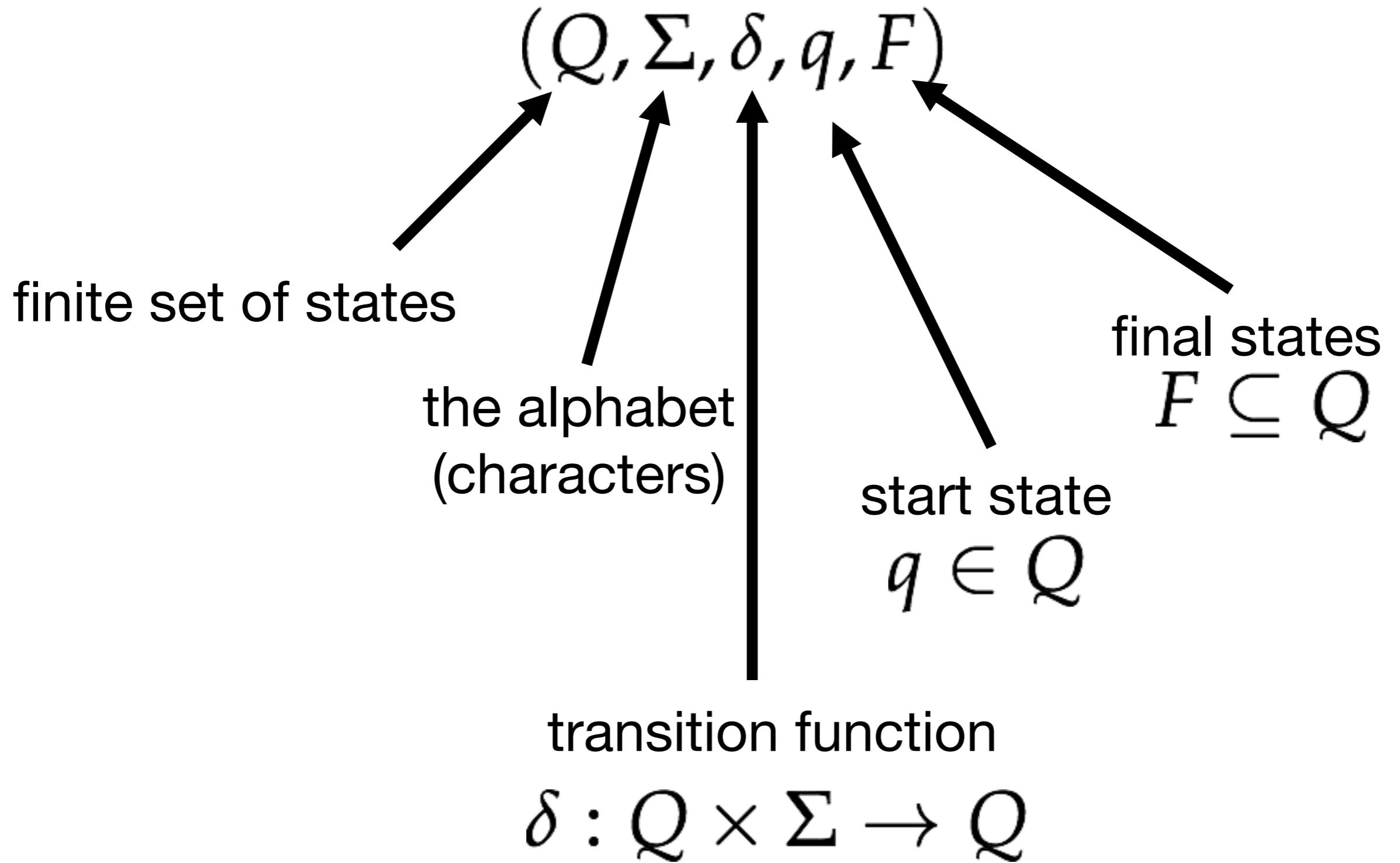
        in\_ch = next char of input

**o/w** stuck // error condition

**until** stuck or input string is consumed

string is accepted iff entire string is  
consumed and final\_states.contains(cur\_state)

# FSMs, formally



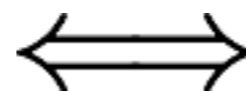
# FSMs, formally

semantics

$$(Q, \Sigma, \delta, q, F)$$

**FSM accepts string**

$$x_1 x_2 x_3 \dots x_n$$

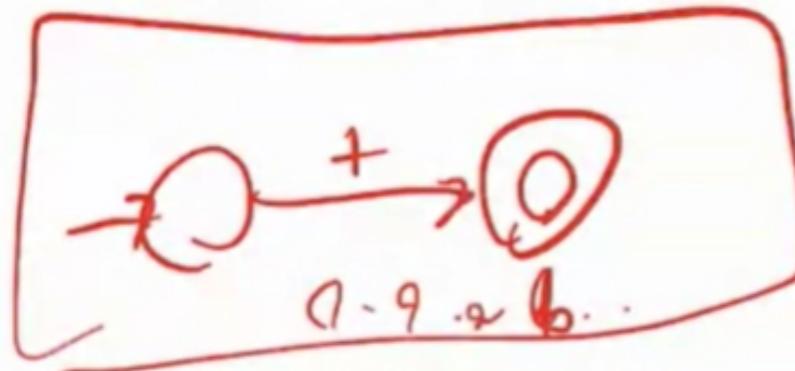


$$\delta(\dots \delta(\delta(\delta(q, x_1), x_2), x_3) \dots, x_n) \in F$$

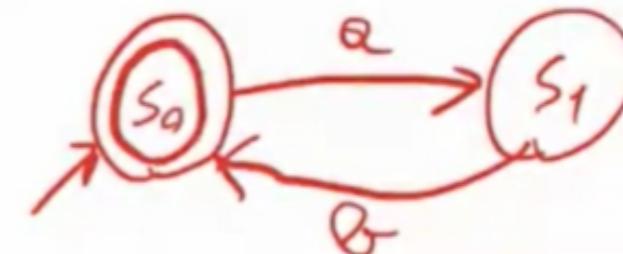
$$\delta(\dots \delta(\delta(\delta(q, x_1), x_2), x_3) \dots, x_n) \in F$$

The language of FSM  $M$  is the set of all words it accepts,  
denoted  $L(M)$  { $s|s$  is accepted by  $M$ }

# FSM example, formally



$(Q, \Sigma, \delta, q, F)$



$Q = \{s_0, s_1\}$  ✓

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

$q = s_0$  ✓

$F = \{s_0\}$  ✓ partial

$\delta = s_0, a \rightarrow s_1$   
 $s_1, b \rightarrow s_0$



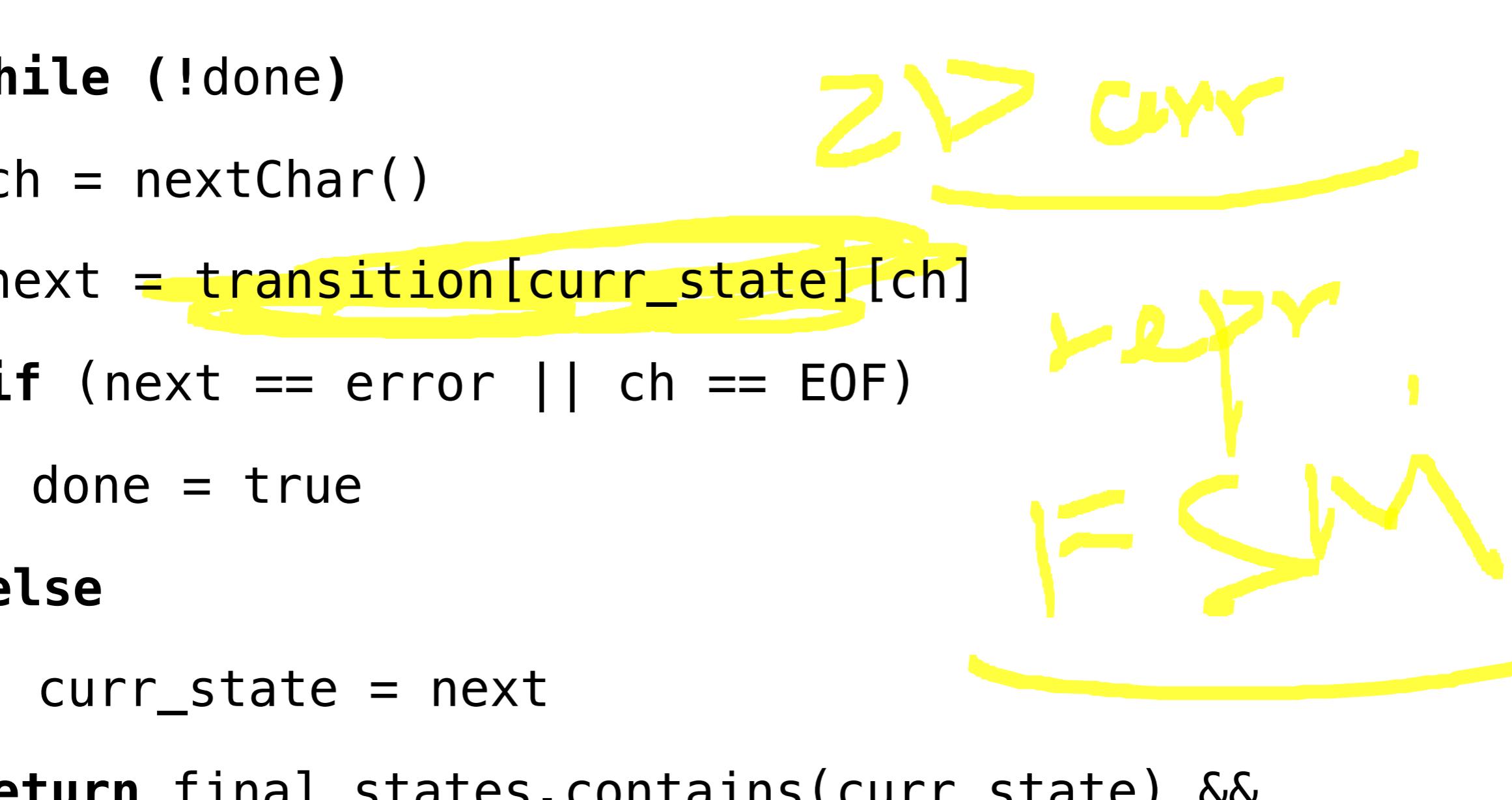
	a	b	c
s0	s1	stuck	X
s1	X	s0	X

anything else, machine is stuck

# Coding an FSM

implement the previous slide details in code

```
curr_state = start_state  
done = false  
while (!done)  
    ch = nextChar()  
    next = transition[curr_state][ch]  
    if (next == error || ch == EOF)  
        done = true  
    else  
        curr_state = next  
return final_states.contains(curr_state) &&  
        next!=error
```



The handwritten annotations are written in yellow ink and include:

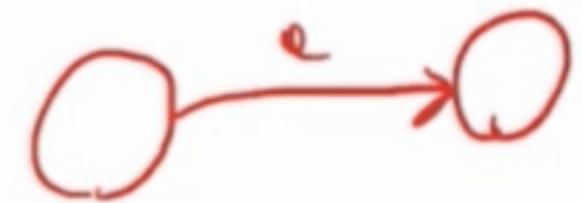
- A large yellow arrow pointing from the word "curr" in the first line of code to the variable "curr\_state" in the second line.
- A yellow bracket under the line "next = transition[curr\_state][ch]" with the handwritten text "ZV> curr" above it.
- A yellow bracket under the line "if (next == error || ch == EOF)" with the handwritten text "expr" above it.
- A yellow bracket under the line "curr\_state = next" with the handwritten text "IF SW" above it.

# FSM types: DFA & NFA

Deterministic

no state has  $> 1$  outgoing edge with same label

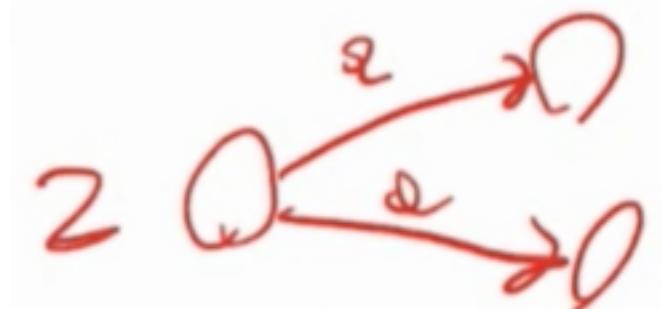
deterministic



Nondeterministic

states may have multiple outgoing edges with same label

nondeterministic



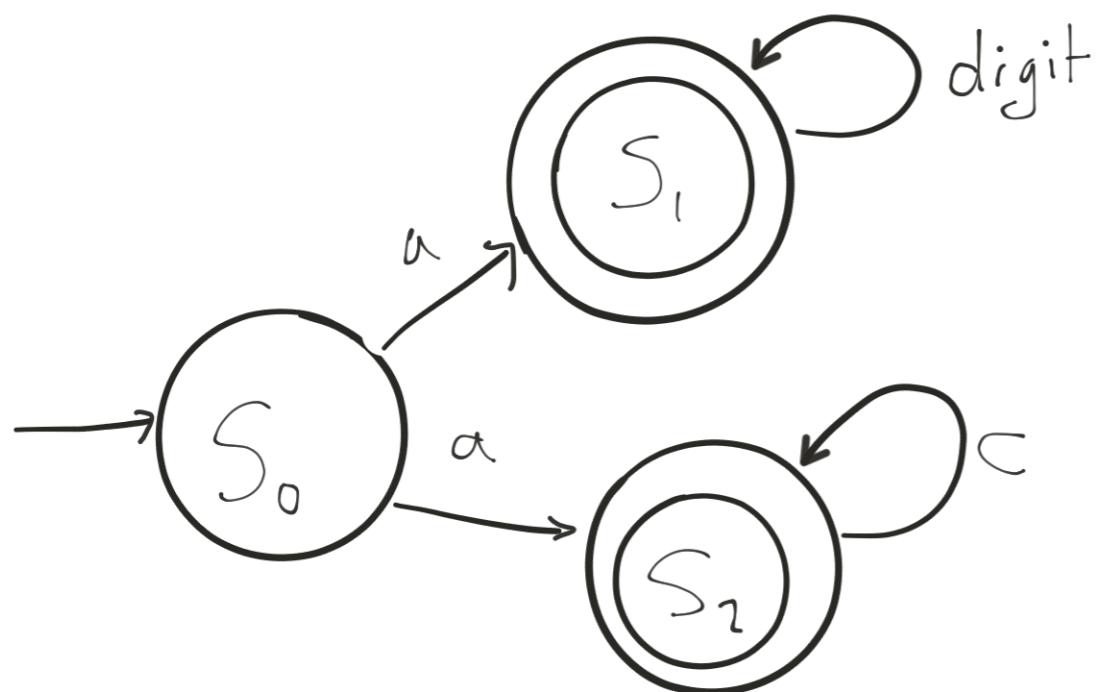
edges may be labelled with special symbol  $\epsilon$  (empty string)

$\epsilon$ -transitions can happen without reading input

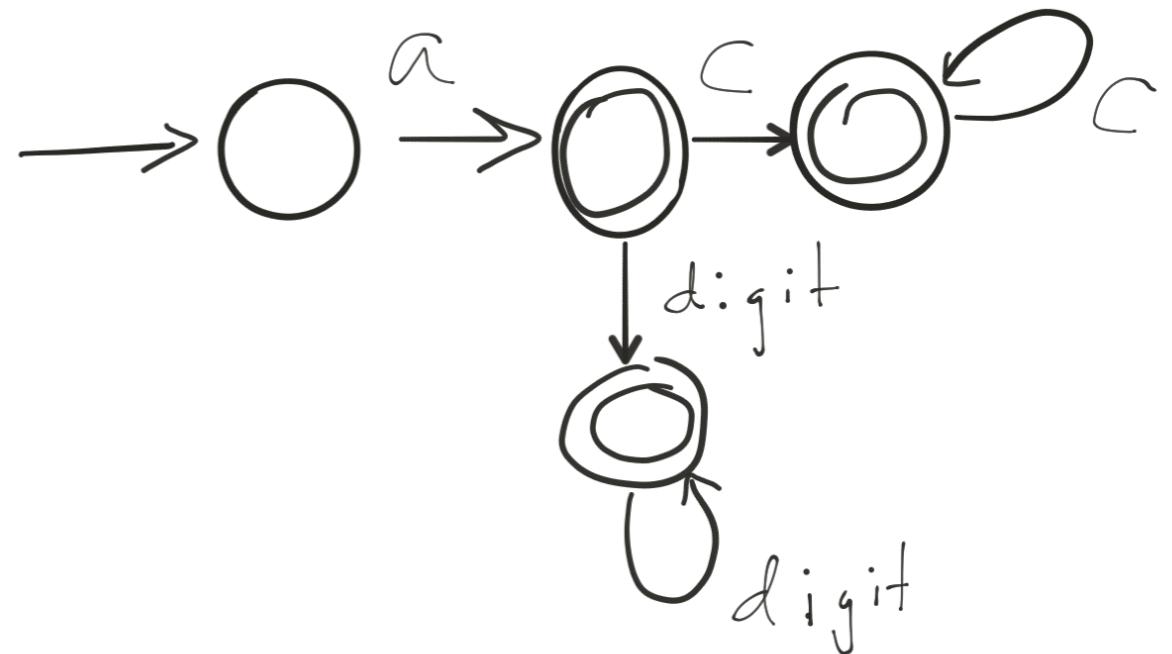
nfa两个限制条件：1.  $>1$ 路径 2. 在这些路径上，label是一样的  
故，如果有大于1的路径但是不同的label，这个不属于nfa



# NFA example



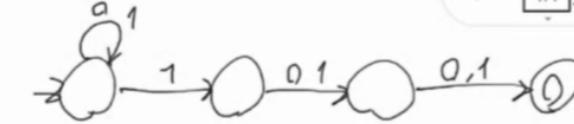
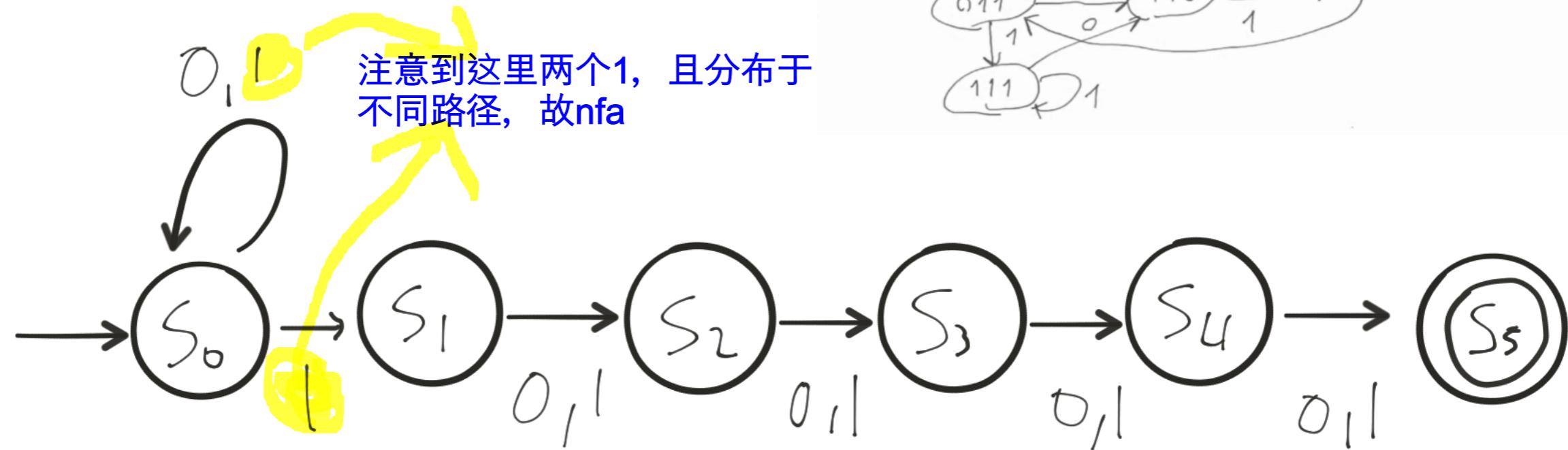
## Equivalent DFA



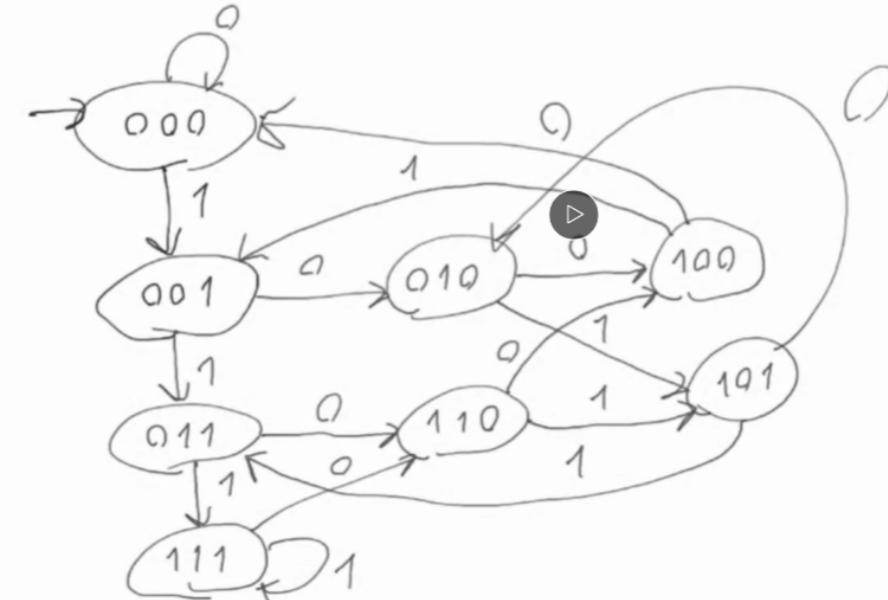
NFA DFA 之前可互相转换, i.e., 若我们能用其中一个定义一组 string, 也一定能用另外一个定义之

# Why NFA?

Much more compact



3<sup>rd</sup> LAST bit is 1



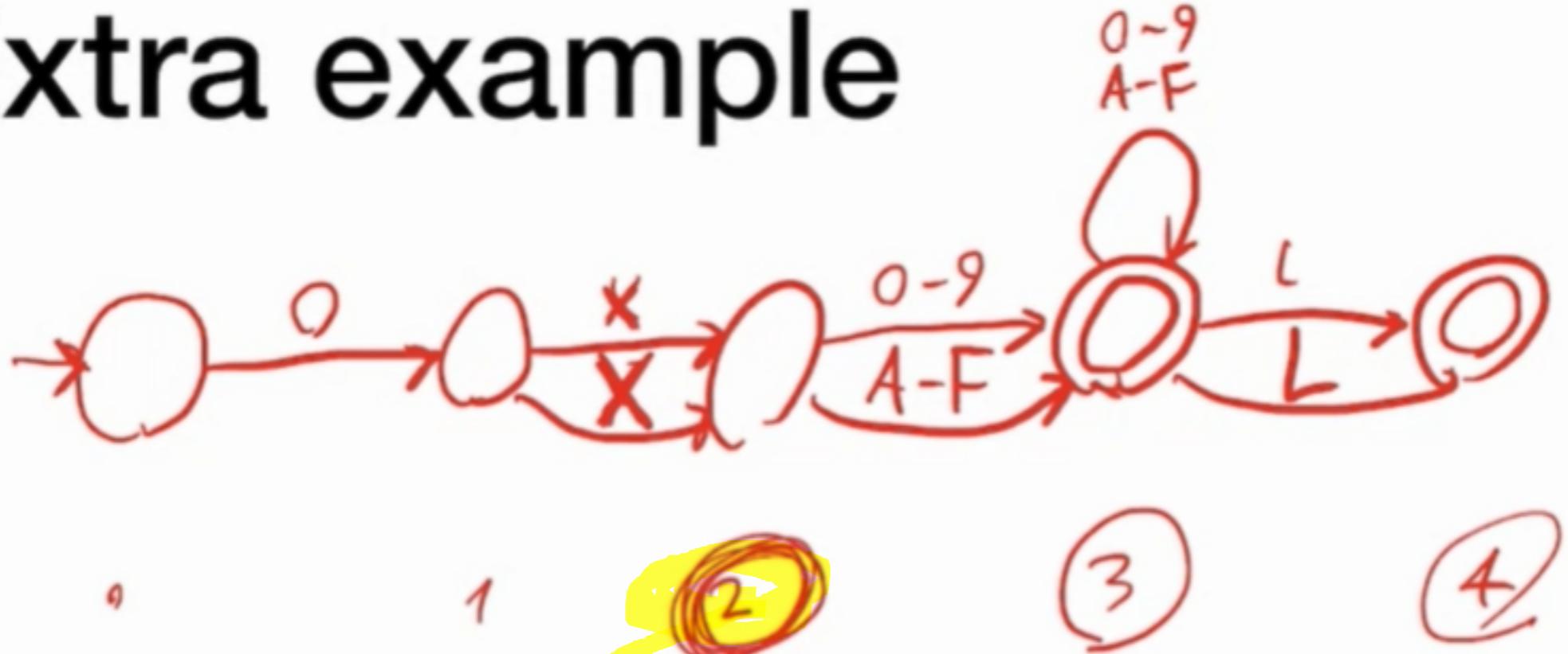
What does this accept?

每个state 0,1两个选择，共五个这样的路径，故 $2^5$

An equivalent DFA needs  $2^5$  states

so, nfa is more efficient

# Extra example



## Hex literals

must start with 0x or 0X

so state 2 is not final state

followed by at least one hex digit (0-9,a-f,A-F)

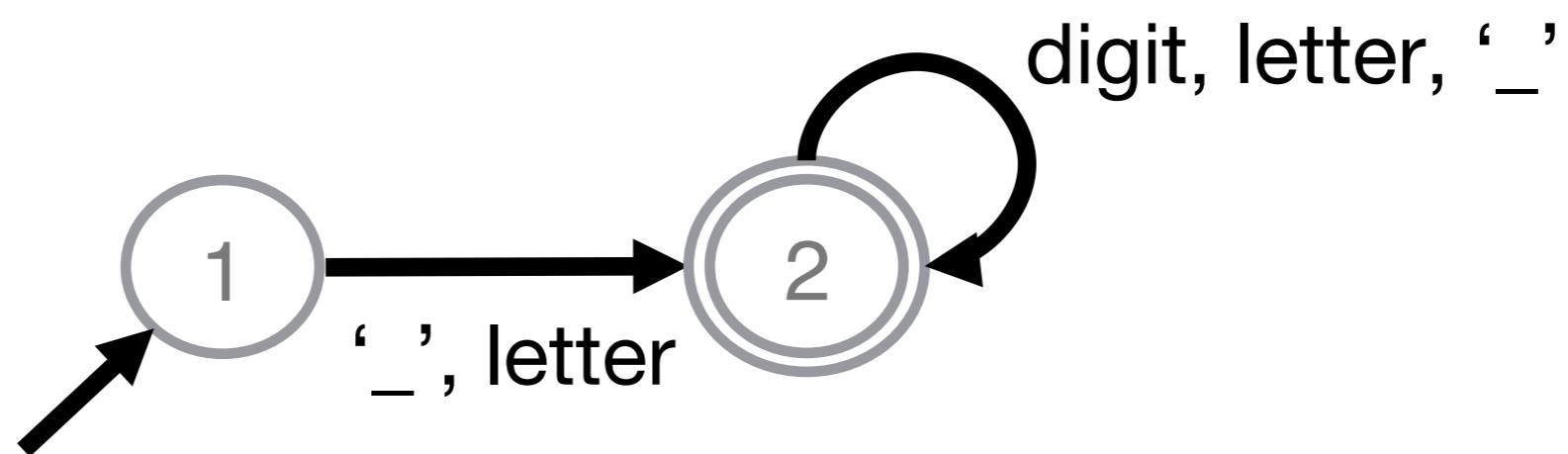
can optionally have long specifier (I,L) at the end

# Extra example

A C/C++ identifier is a sequence of one or more letters, digits, or underscores. It cannot start with a digit.

# Extra Example - Part 1

A C/C++ identifier is a sequence of one or more letters, digits, or underscores. It cannot start with a digit.



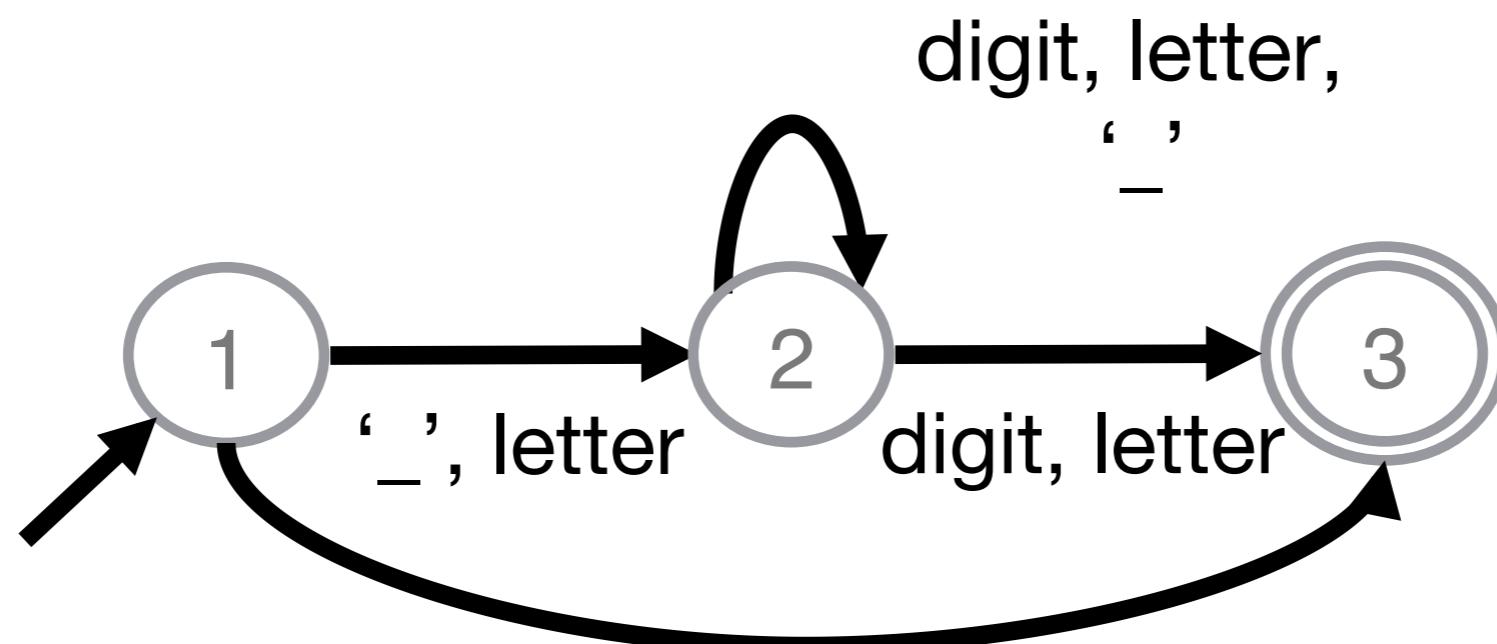
# Extra example

A C/C++ identifier is a sequence of one or more letters, digits, or underscores. It cannot start with a digit.

*What if you wanted to add the restriction that it can't end with an underscore?*

# Extra Example - Part 2

What if you wanted to add the restriction that it  
can't end with an underscore?



这个题核心在于：到final state的路径中不得包含underscore

# Recap

The scanner reads stream of characters and finds tokens

Tokens are defined using regular expressions, which are finite-state machines

Finite-state machines can be non-deterministic

Next time: understand connection between deterministic and non-deterministic FSMs

**automatatutor.com**

**Course ID: 219CS536S**

**Password: YDXXMP3E**

# Announcements

- H1 posted. Due Next Tuesday

# **Nondeterministic Finite Automata**

CS 536

# Previous Lecture

Scanner: converts a sequence of characters to a sequence of tokens

Scanner and parser: master-slave relationship

Scanner implemented using FSMs

FSM: DFA or NFA

# This Lecture

NFAs from a formal perspective

Theorem: NFAs and DFAs are equivalent

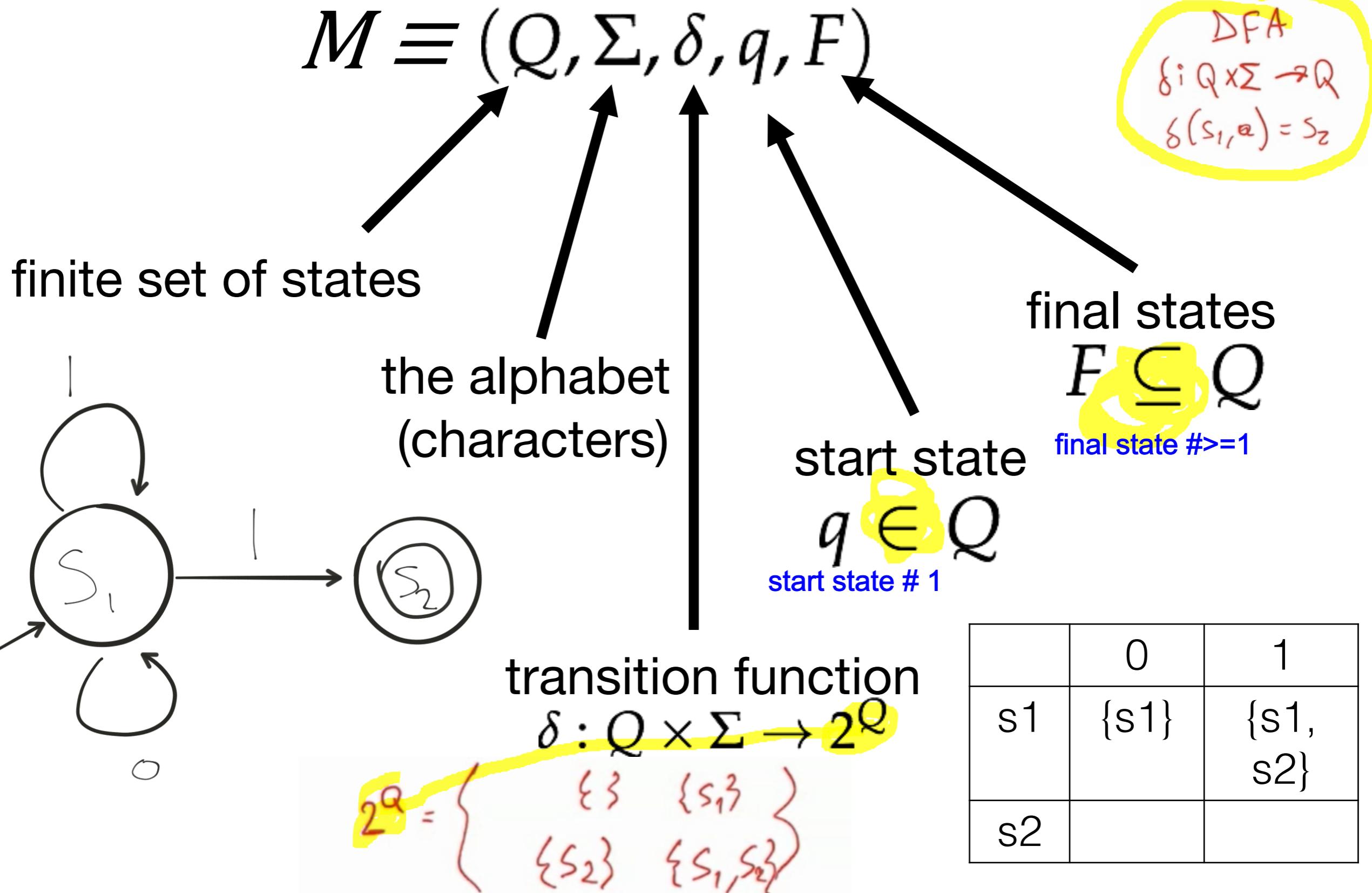
Regular languages and Regular expressions



# NFAs, formally

$$\delta(s_1, 1) = \{s_1, s_2\}$$

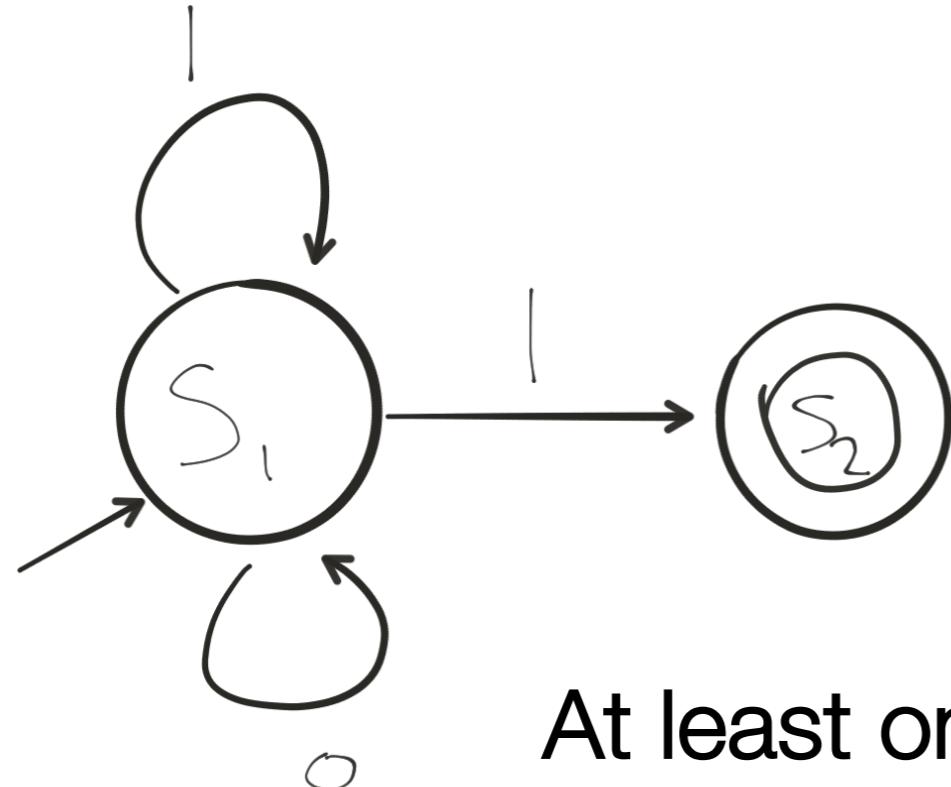
NFA transition func



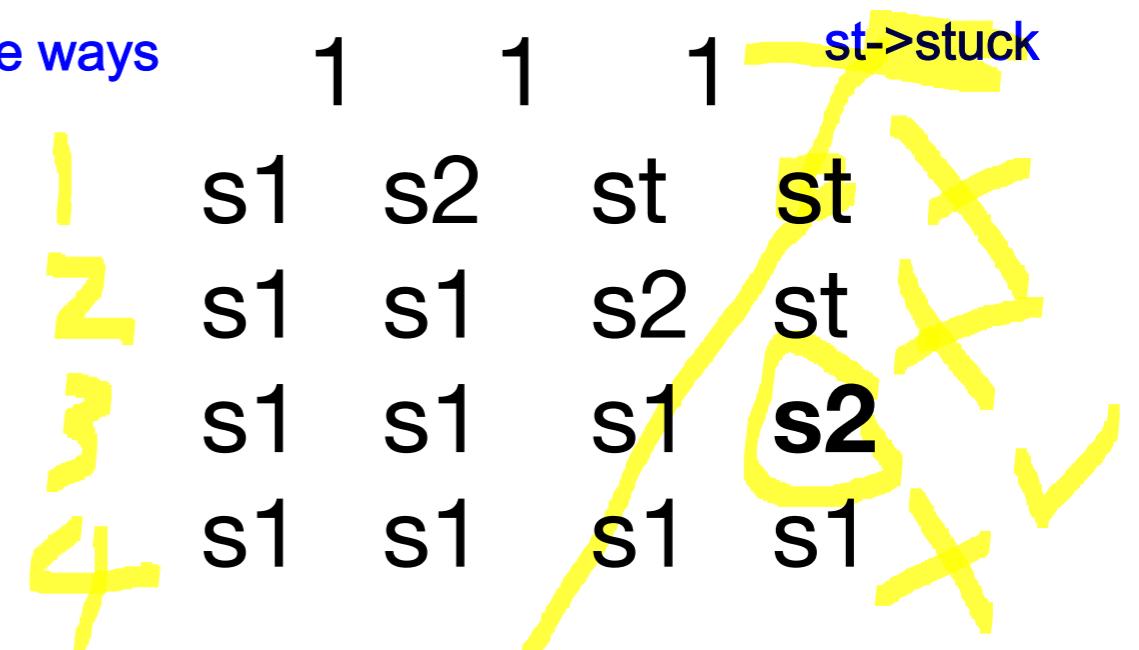
# NFA

if no seq. of transition satisfy these two conditions, then it fails

To check if string is in  $L(M)$  of NFA  $M$ , simulate **set of choices** it could make



4 possible ways



At least one sequence of transitions that:

must satisfy these 2 conditions



Consumes all input (without getting stuck)



Ends in one of the final states

# NFA and DFA are Equivalent

Two automata  $M$  and  $M'$  are equivalent iff  $L(M) = L(M')$

set of string  $M$  accepts

lemma 1 is very easy to prove

Lemmas to be proven

$$\delta(q_1, \alpha) = q_1$$

$$\delta(q_1, \alpha) = \{q_1\}$$

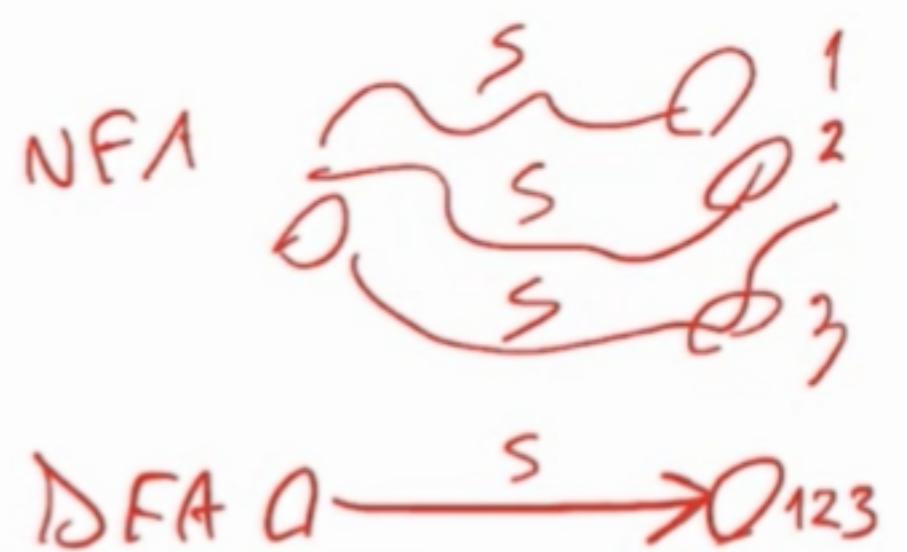
$$DFA \subseteq NFA$$

✓ Lemma 1: Given a DFA  $M$ , one can construct an NFA  $M'$  that recognizes the same language as  $M$ , i.e.,  $L(M') = L(M)$

$$NFA \subseteq DFA$$

Lemma 2: Given an NFA  $M$ , one can construct a DFA  $M'$  that recognizes the same language as  $M$ , i.e.,  $L(M') = L(M)$

# Proving lemma 2



**Lemma 2:** Given an NFA  $M$ , one can construct a DFA  $M'$  that recognizes the same language as  $M$ , i.e.,  $L(M') = L(M)$

**Idea:** we can only be in finitely many subsets of states at any one time

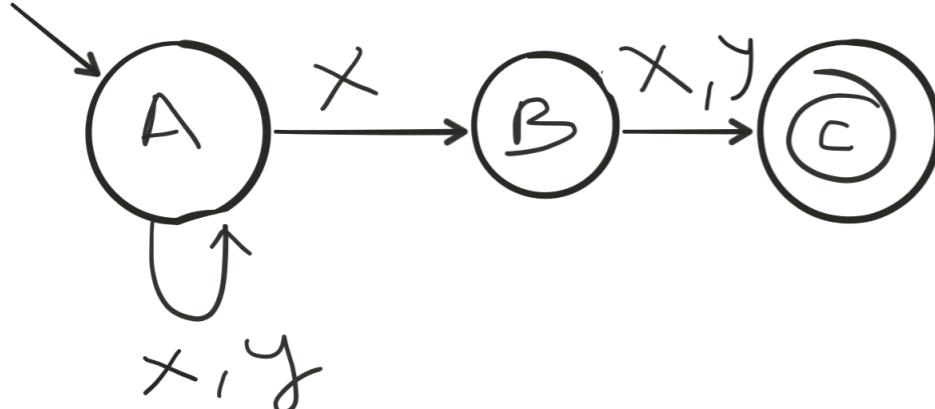
$$2^{|Q|}$$

possible combinations of states

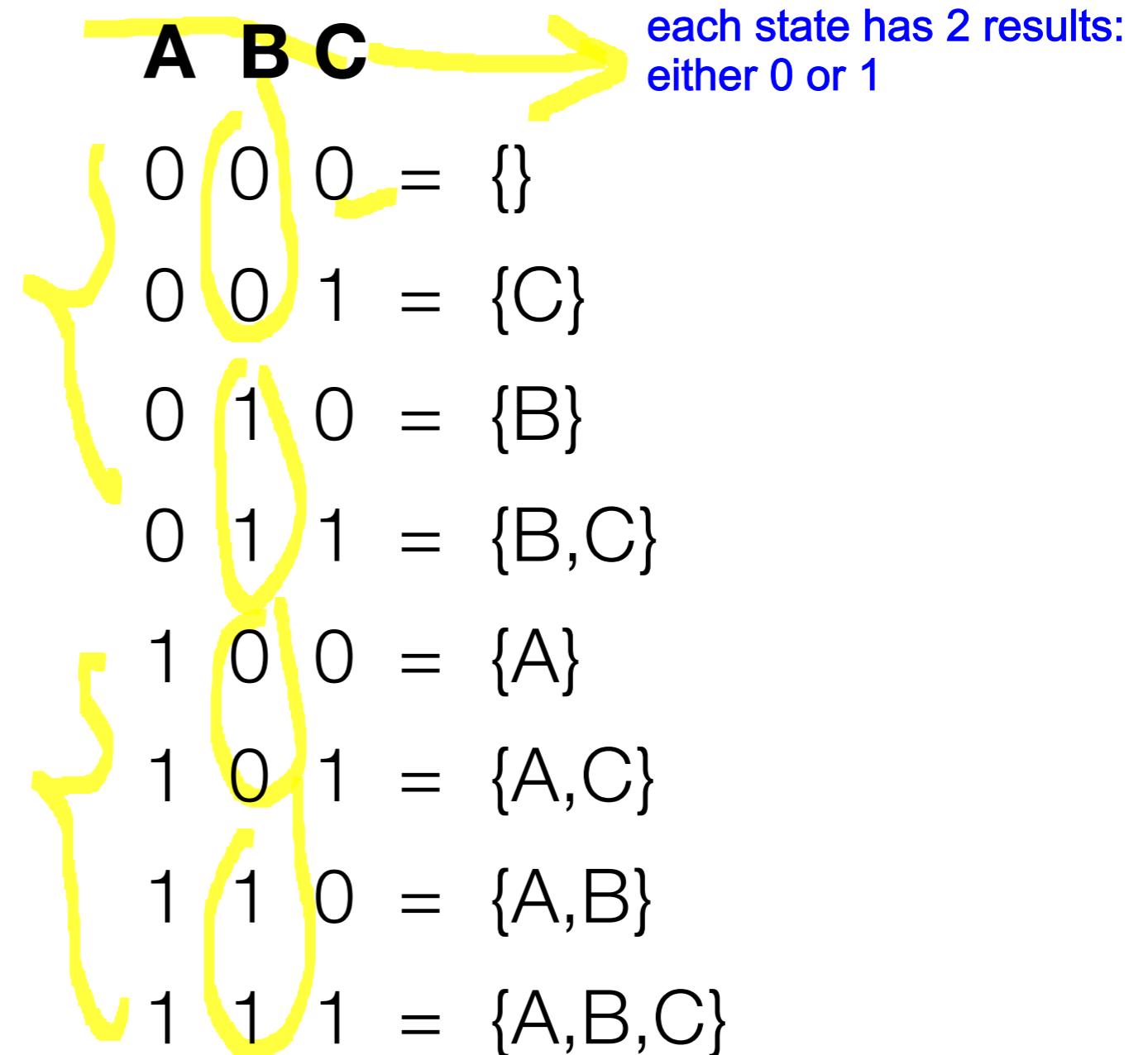
Why?

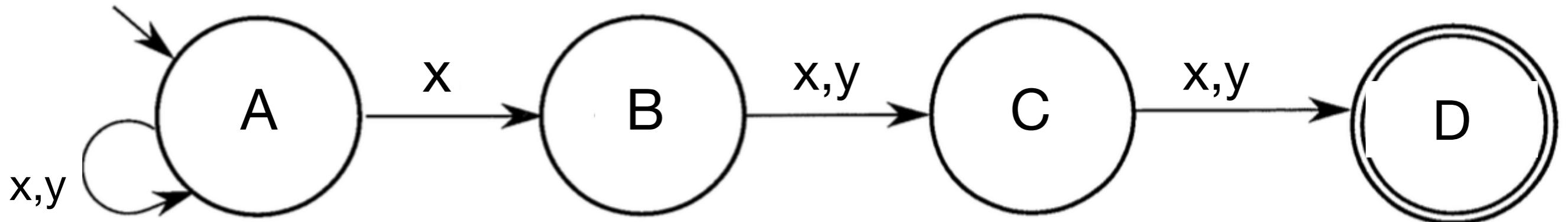
# Why $2^{|Q|}$ states?

3 STATES  
 $2^3$  SUBSETS



**Build DFA that tracks set of states the NFA is in!**

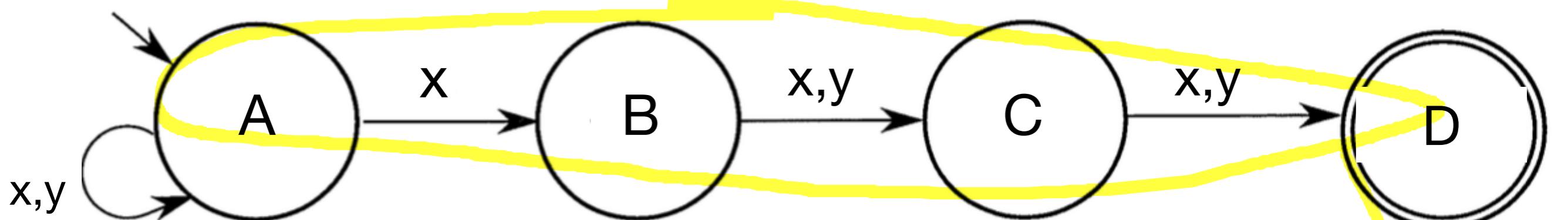




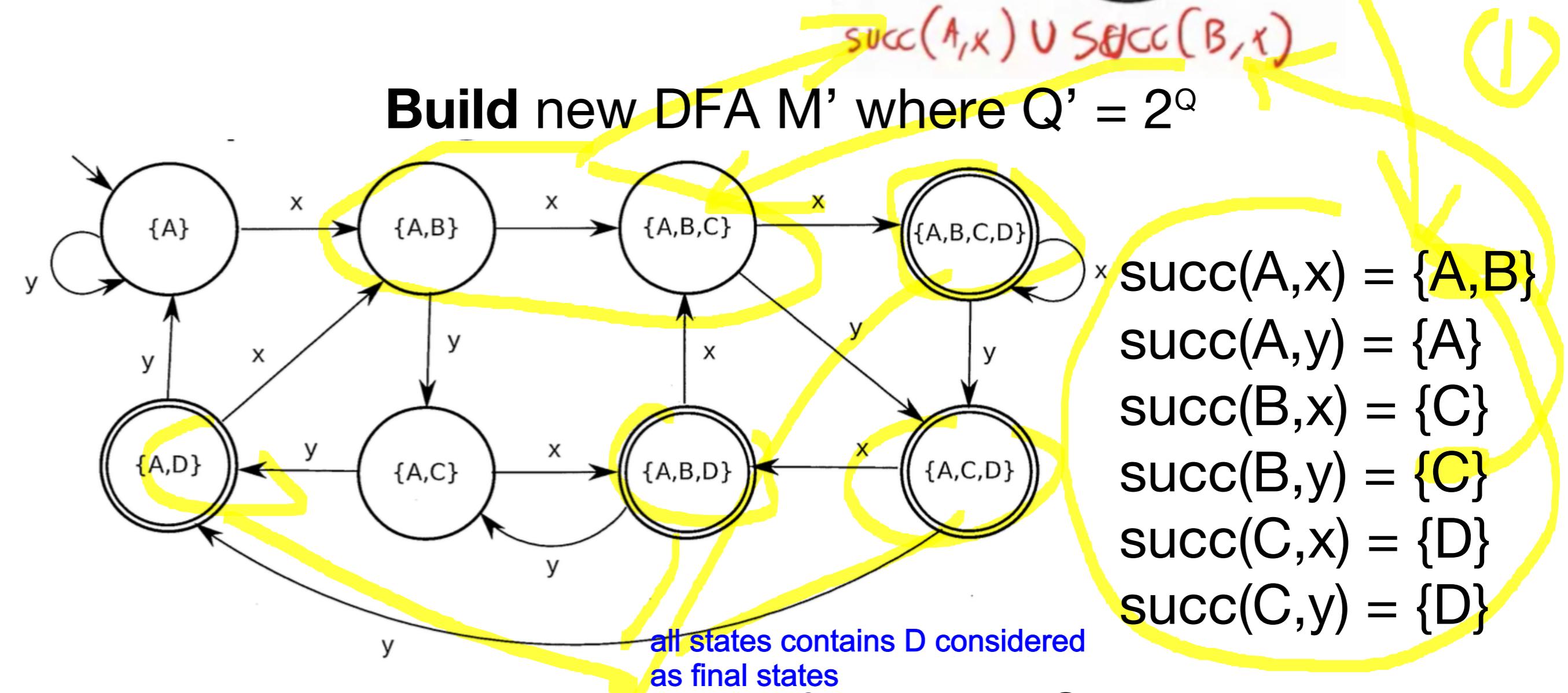
**Defn:** let  $\text{succ}(s,c)$  be the set of choices the NFA could make in state  $s$  with character  $c$

$$\begin{aligned}
 \text{succ}(A,x) &= \{A, B\} \\
 \text{succ}(A,y) &= \{A\} \\
 \text{succ}(B,x) &= \{C\} \\
 \text{succ}(B,y) &= \{C\} \\
 \text{succ}(C,x) &= \{D\} \\
 \text{succ}(C,y) &= \{D\}
 \end{aligned}$$

从nfa->dfa的方法



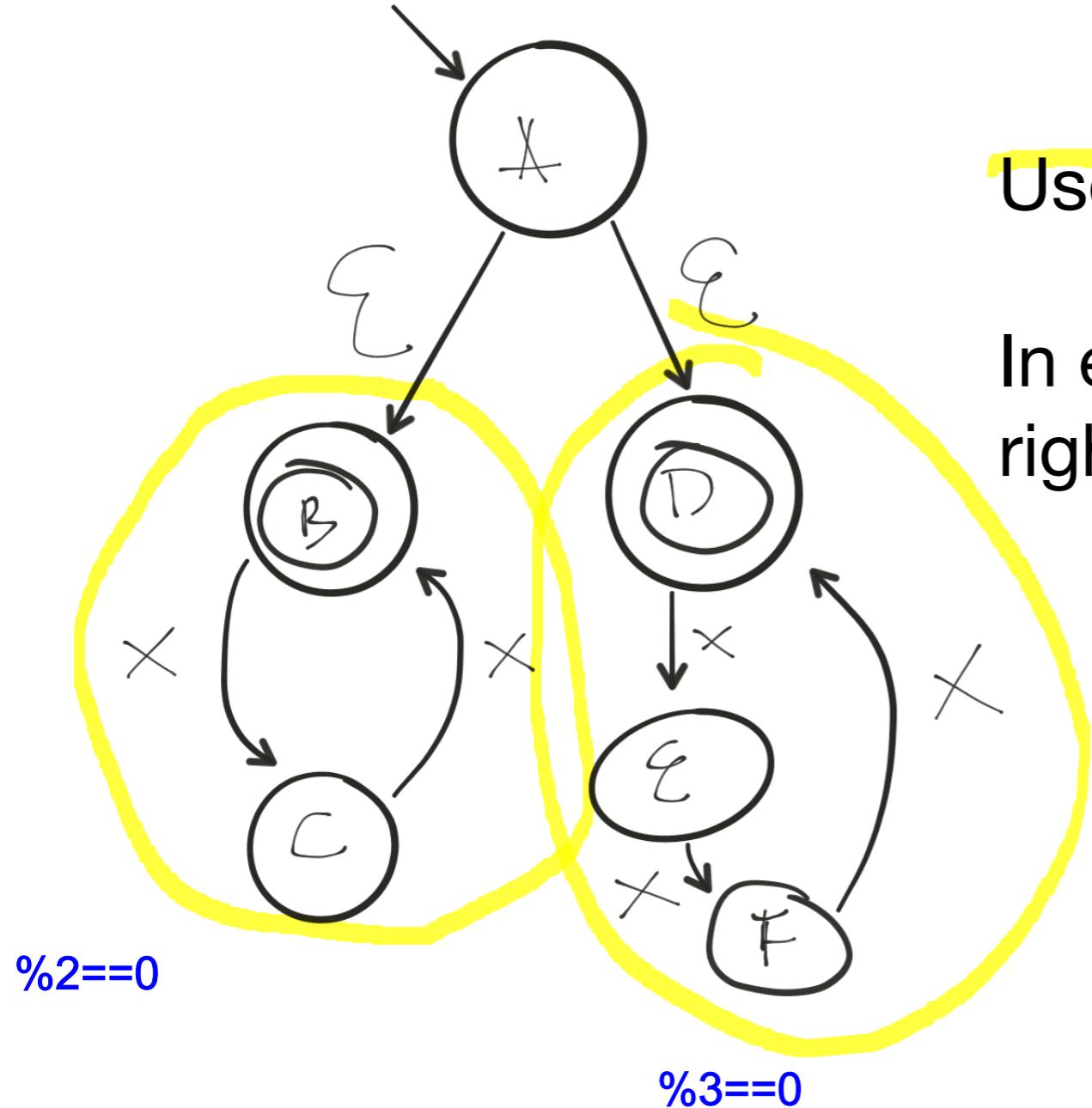
Build new DFA  $M'$  where  $Q' = 2^Q$



To build DFA: Add an edge from state S on character c to state S' if S' represents the union of states that all states in S could possibly transition to on input c

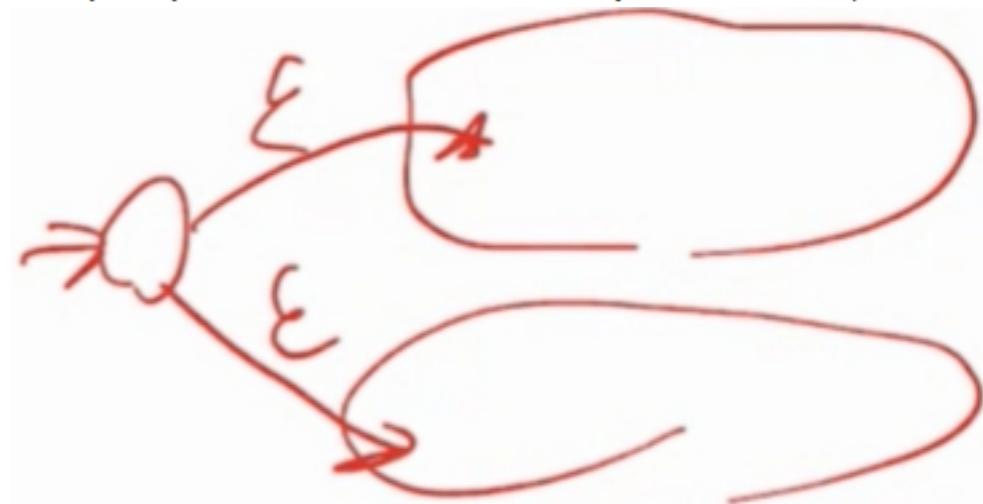
# $\epsilon$ -transitions

Eg:  $x^n$ , where n is even **or** divisible by 3



$\epsilon$  is an empty string just like  $" "$ . It is not a symbol of the alphabet. It simply means (in the NFA in your picture) that if you are on state  $q_1$ , without consuming any symbol you can move to state  $q_3$  but you are not obliged to do so. but if you are on  $q_1$  and you read  $b$ , then you must go to  $q_2$ .

Note that a word is accepted in a NFA if there **exists** a path from the initial state to an accepting state labelled by the symbols of this words. It should be rejected if no such a path exists.



Useful for taking union of two FSMs

In example, left side accepts even n;  
right side accepts n divisible by 3

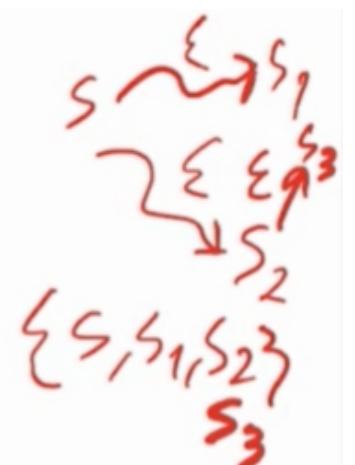
X	X
AB	C
AD	E
A	F

# Eliminating $\epsilon$ -transitions

We want to construct  $\epsilon$ -free FSM  $M'$  that is equivalent to  $M$

## Definition:

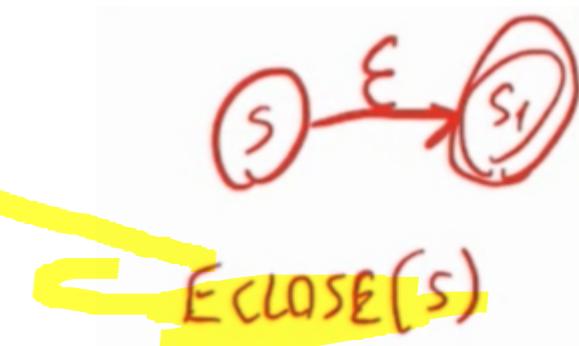
$\text{eclose}(s) = \text{set of all states reachable from } s \text{ in zero or more epsilon transitions}$



## $M'$ components

$s$  is an accepting state of  $M'$  iff  $\text{eclose}(s)$  contains an accepting state

$s - c -> t$  is a transition in  $M'$  iff  $q - c -> t$  for some  $q$  in  $\text{eclose}(s)$

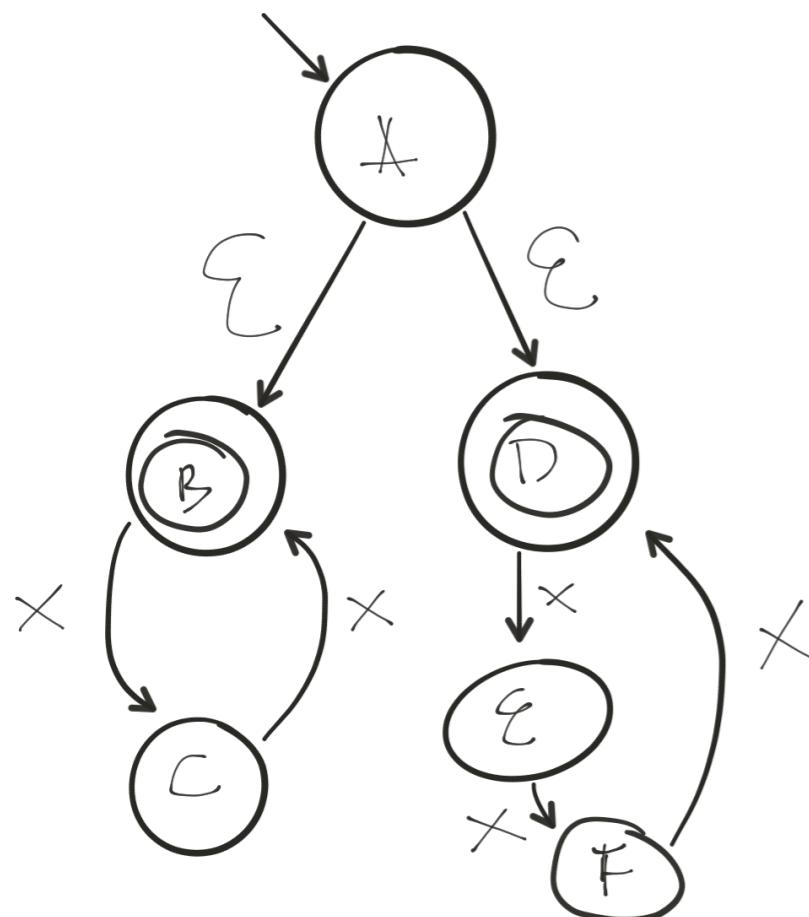


# Eliminating $\epsilon$ -transitions

We want to construct  $\epsilon$ -free NFA  $M'$  that is equivalent to  $M$

## Definition: Epsilon Closure

$\text{eclose}(s) = \text{set of all states reachable from } s \text{ using zero or more epsilon transitions}$



	eclose
A	{A, B, D}
B	{B}
C	{C}
D	{D}
E	{E}
F	{F}

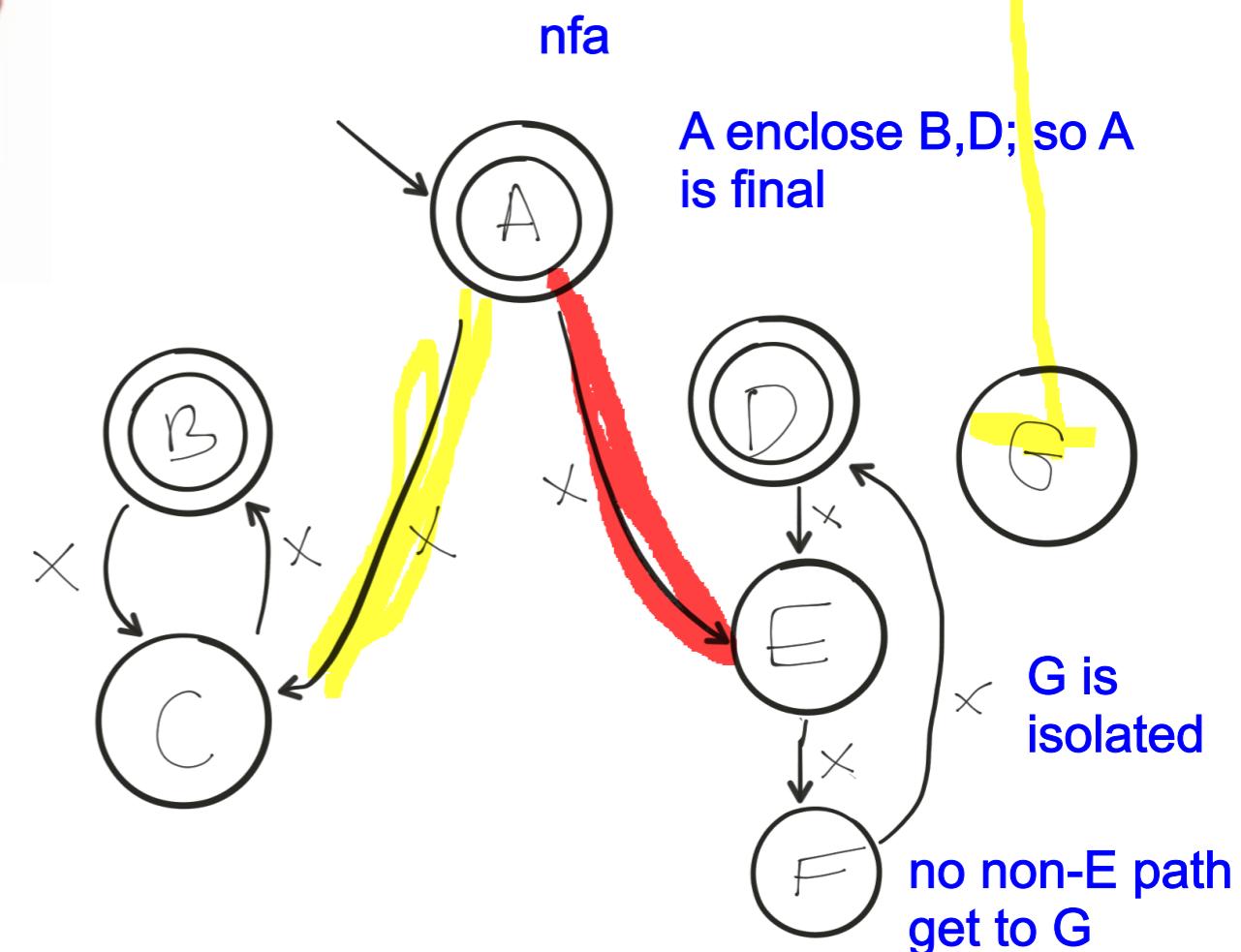
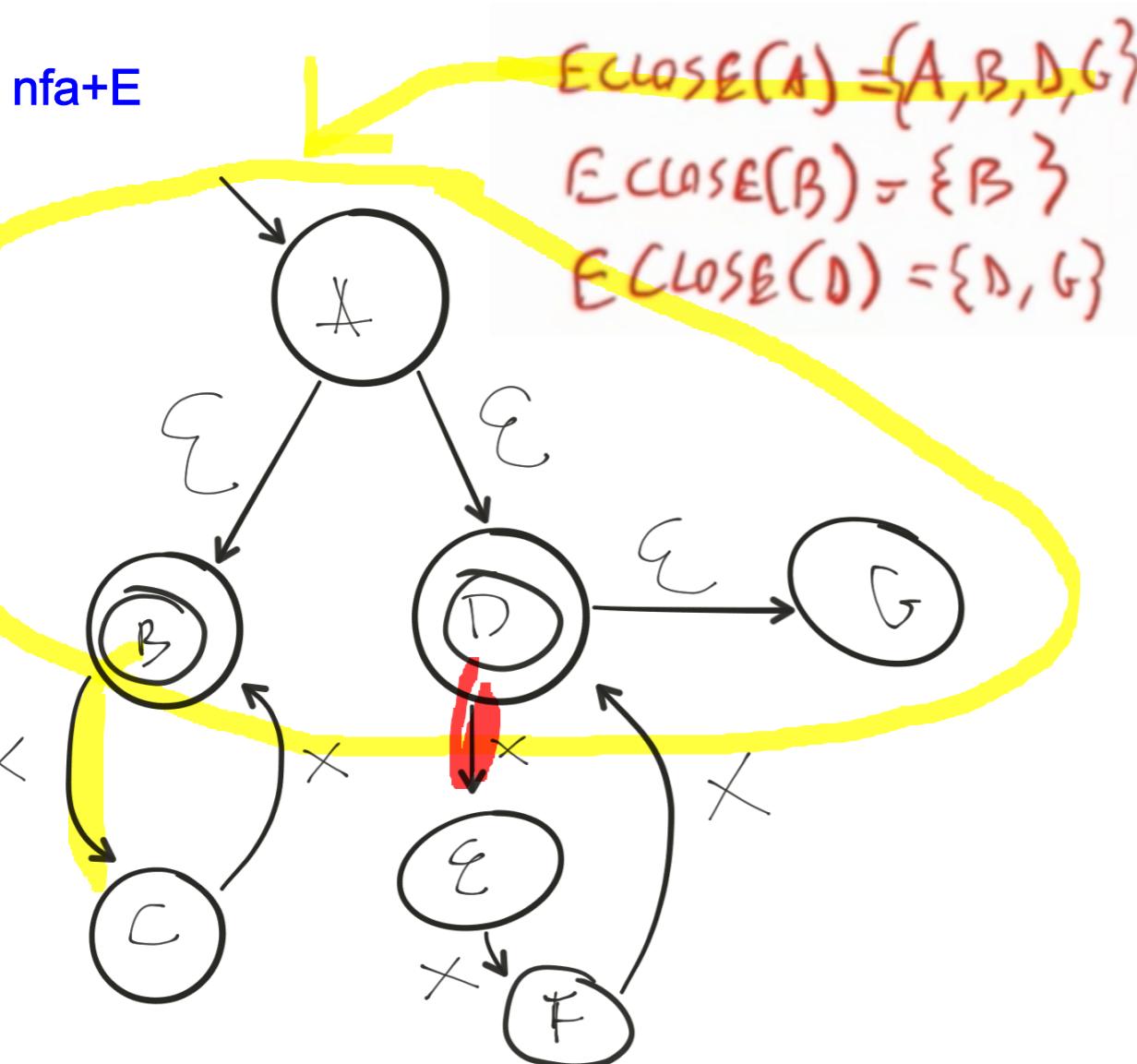
**Def:**  $\text{eclose}(s)$  = set of all states reachable from  $s$  in zero or more epsilon transitions

$s$  is an accepting state of  $M'$  iff  $\text{eclose}(s)$  contains an accepting state

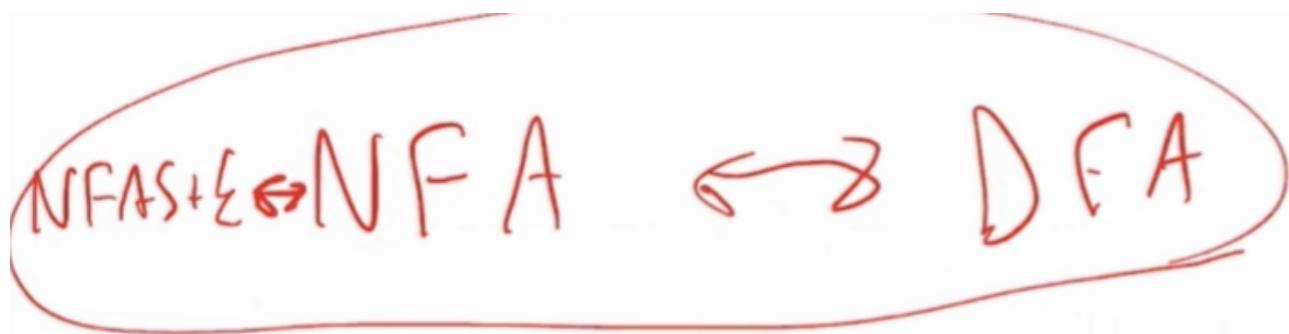
$s - c \rightarrow t$  is a transition in  $M'$  iff  
 $q - c \rightarrow t$  for some  $q$  in  $\text{eclose}(s)$

$$\text{ECLOSE}(G) = \{G\}$$

so  $G$  is not final



# Recap



these 3 are all equivalent

NFAs and DFAs are equally powerful

any language definable as an NFA is definable as a DFA

$\epsilon$ -transitions do not add expressiveness to NFAs

we showed a simple algorithm to remove epsilons

# **Regular Languages and Regular Expressions**

# Regular Language

Any language recognized by an FSM is a regular language

Examples:

- Single-line comments beginning with //
- Integer literals
- $\{\epsilon, ab, abab, ababab, abababab, \dots\}$
- C/C++ identifiers

# Regular expressions

Pattern describing a language

**operands:** single characters, epsilon

**operators:** from low to high precedence

alternation “or”:  $a \mid b$

catenation:  $a.b$ ,  $ab$ ,  $a^3$  (which is  $aaa$ )

iteration:  $a^*$  (0 or more a's) aka Kleene star

$\epsilon\alpha = \alpha$

$\alpha\epsilon = \alpha$

# Why do we need them?

Each token in a programming language can be defined by a regular language

Scanner-generator input: one regular expression for each token to be recognized by scanner

Regular expressions are inputs to a scanner generator

# Regexp, cont'd

正则出现的目的是写程序的时候去方便地  
implement FA

$\alpha^*$  0 or More  $\alpha$   
 $\alpha^+$  1 or more  $\alpha$

## Conventions:

$a^+$  is  $aa^*$

letter is  $a|b|c|d|\dots|y|z|A|B|\dots|Z$

digit is  $0|1|2|\dots|9$

not( $x$ ) all characters except  $x$

. is any character

parentheses for grouping, e.g.,  $(ab)^*$

$\epsilon, ab, abab, ababab$

$\Sigma = .$

e.g.,  $(ab)^*$

$\overline{ab}^*$   $\overline{ab}\overline{ab}$

# Regexp, example

这里表示出现0或1次

## regexp, example

$(0x|0X) \text{ DIGIT}^+ (L|E)$

$(0x|0X)$

Hex strings

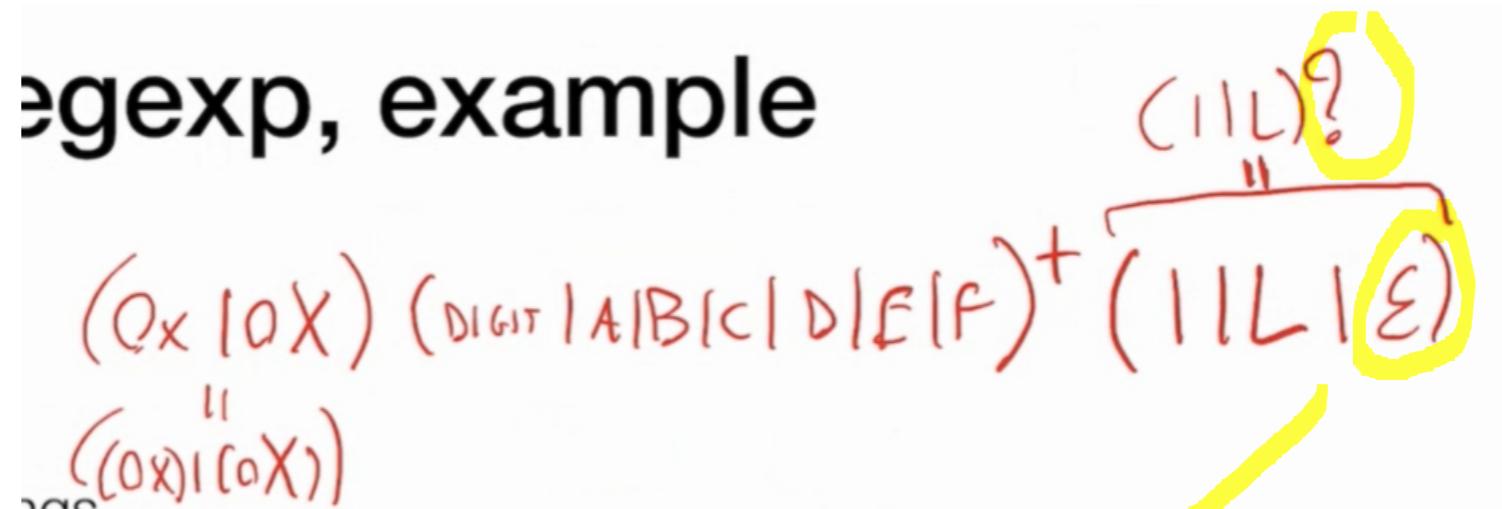
start with 0x or 0X

followed by one or more hexadecimal digits

optionally end with L or E

$0(x|X)\text{hexdigit}^+(L|E)$

where hexdigit = digit|a|b|c|d|e|f|A|...|F



# Regexp, example

Single-line comments in Java/C/C++

```
// this is a comment
```

```
//((not('\n'))*)('n|epsilon)
```

\n is optional

we don't want \n be included

# Regexp, example

C/C++ identifiers: sequence of letters/digits/underscores; cannot begin with a digit; cannot end with an underscore

Example: a, \_bbb7, cs\_536

Regular expression

letter | (letter|\_)(letter|digit|\_)\* (letter|digit)

# Recap

Regular Languages

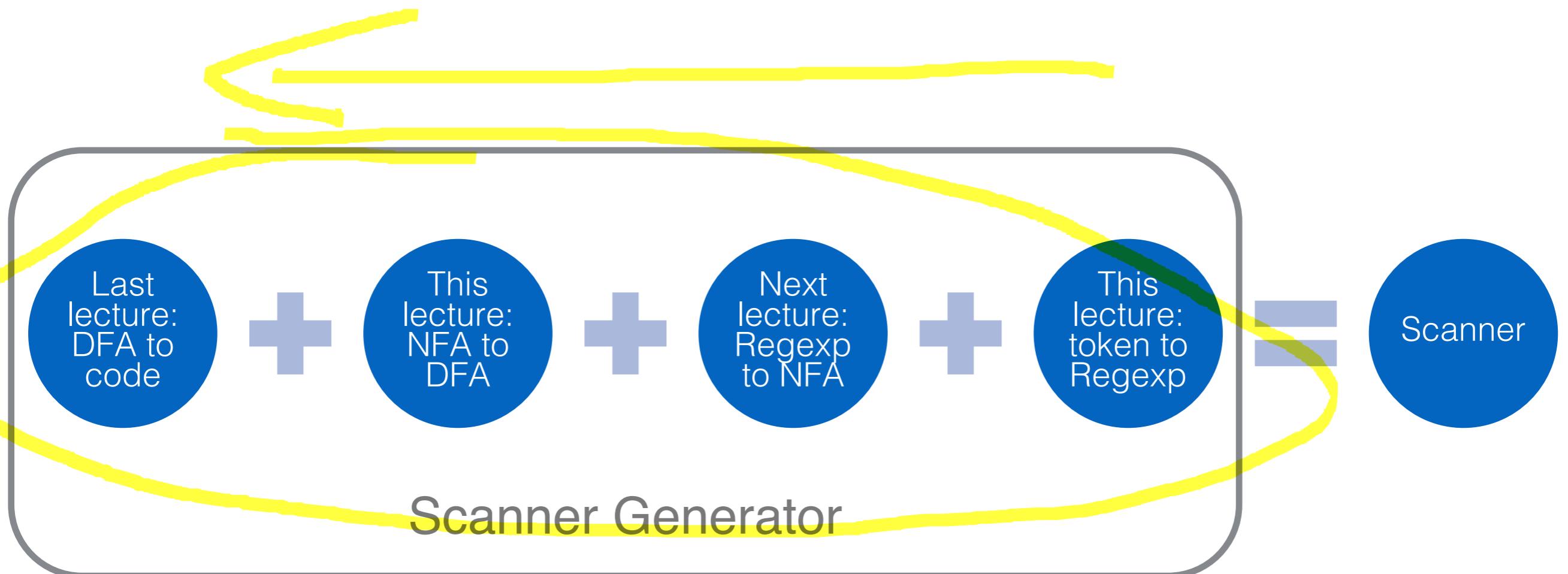
Languages recognized/defined by FSMs

Regular Expressions

Single-pattern representations of regular languages

Used for defining tokens in a scanner generator

# Creating a Scanner



# Announcements

P2 will be assigned today or tomorrow

# **RegExps & DFAs**

CS 536

# Pre-class warm up

Write the regexp for Fortran real literals

An optional sign ('+' or '-')

An integer or:

1 or more digits followed by a '.' followed by 0 or more digits

or: A '.' followed by one or more digits

$$((+ \mid - \mid \varepsilon)(\text{digit}+(\text{.}' \mid \varepsilon)) \mid (\text{digit}^* \text{.}' \text{digit}))$$

# Last time

Explored NFAs

for every NFA there is an equivalent DFA

epsilon edges add no expressive power

Introduce regular languages / expressions

# Today

Convert regexps to DFAs

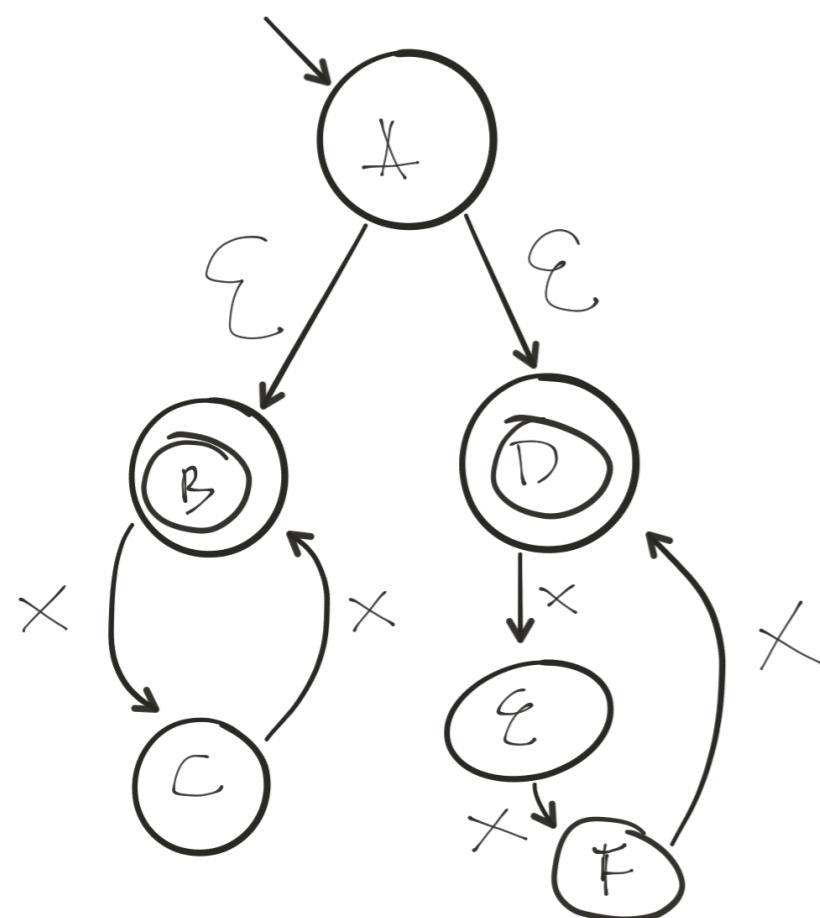
From language recognizers to tokenizers

# Regexp to NFAs

Literals/epsilon correspond to simple DFAs

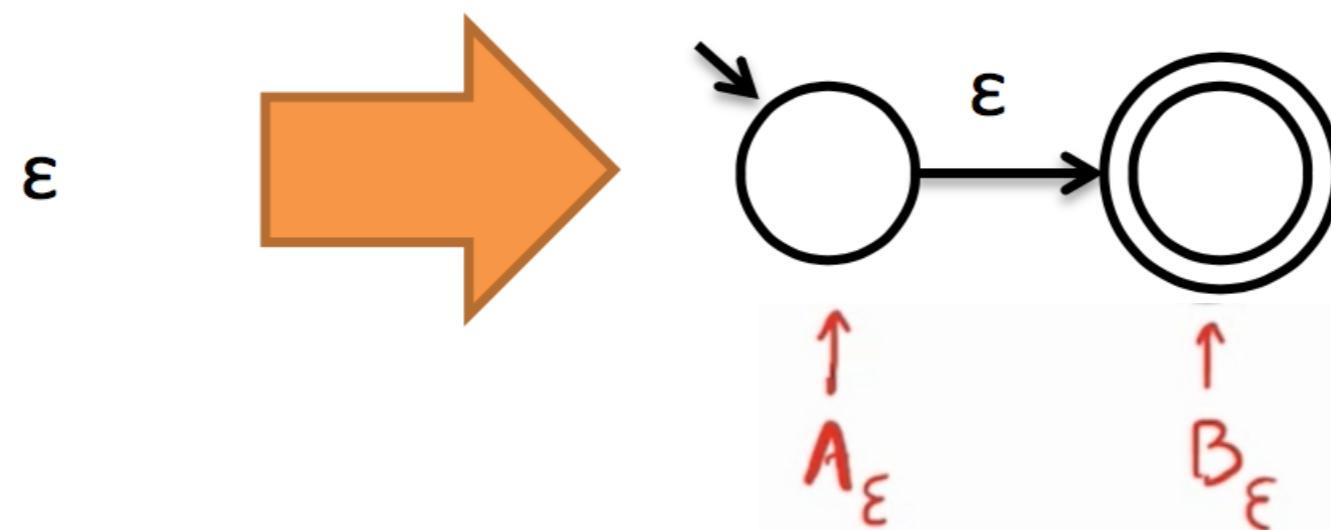
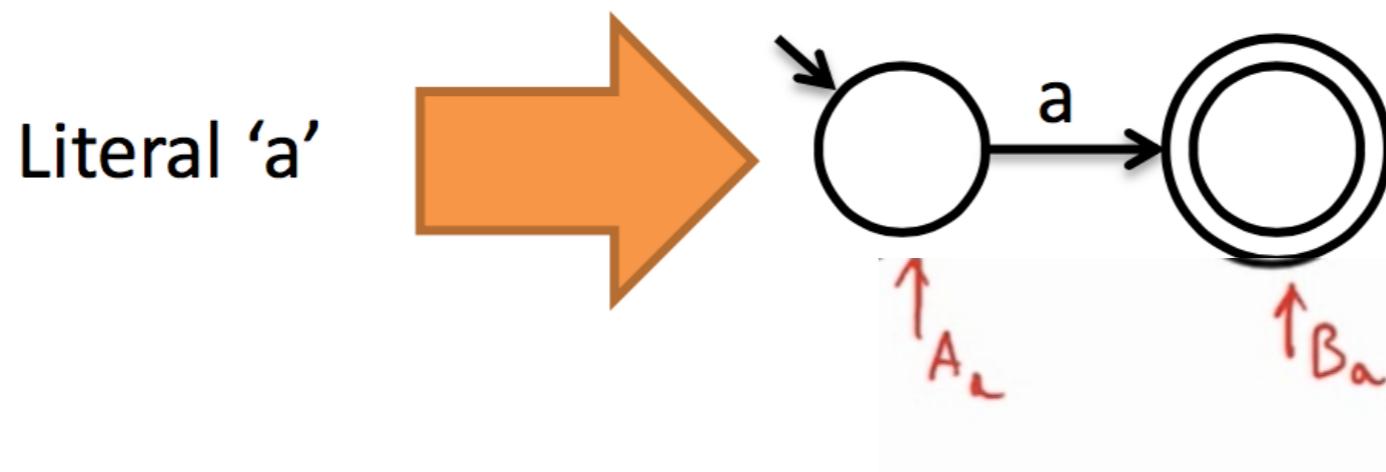
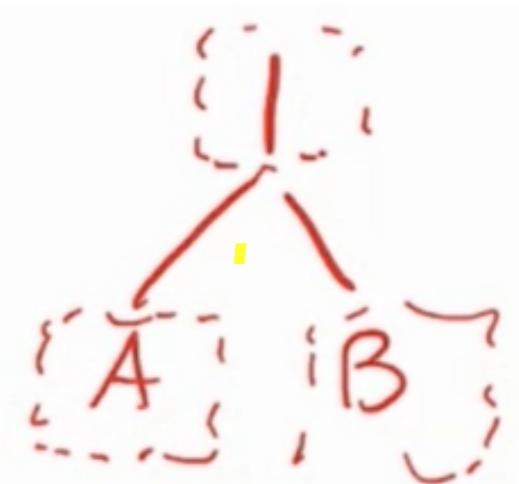
Operators correspond to methods of joining  
DFAs

$x^n$ , where n is even **or** divisible by 3



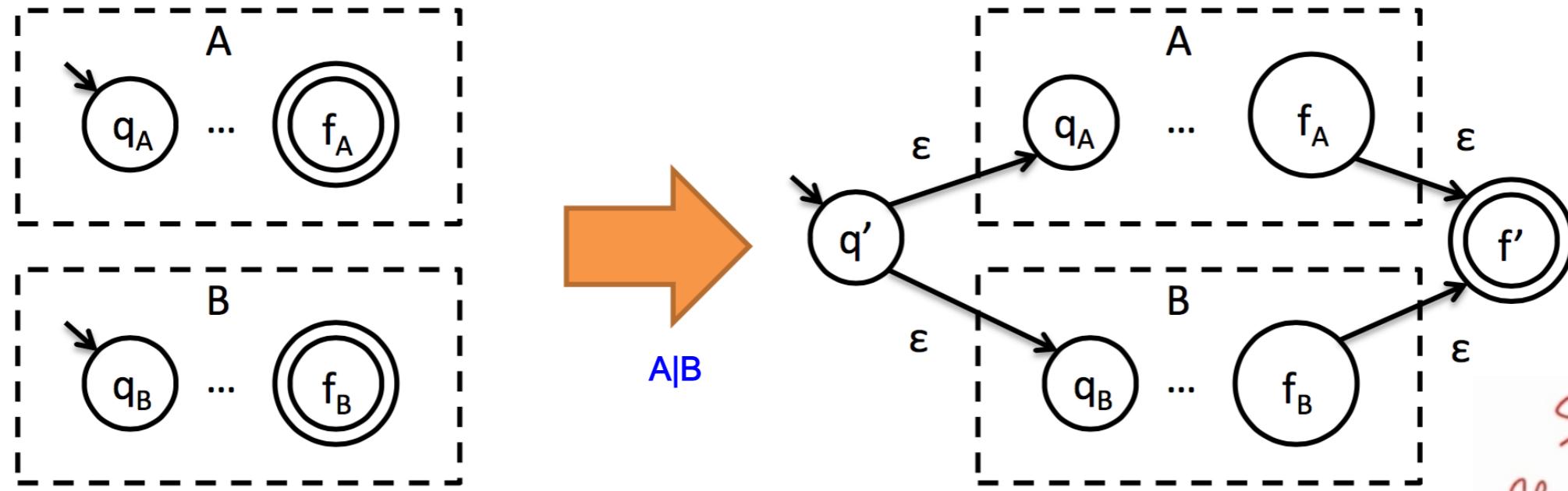
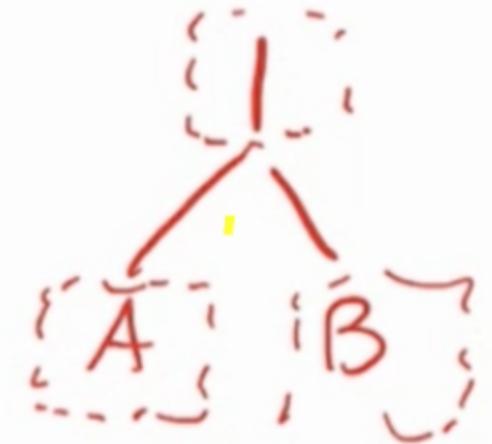
# Regexp to NFA rules

Rules for operands



# Regexp to NFA rules

Rules for alternation  $A|B$



$SEL(A)$   
or  $SEL(B)$

Make new start state  $q'$  and new final state  $f'$

Make original final states non-final

Add to  $\delta$ :

$$q', \epsilon \rightarrow q_A$$

$$q', \epsilon \rightarrow q_B$$

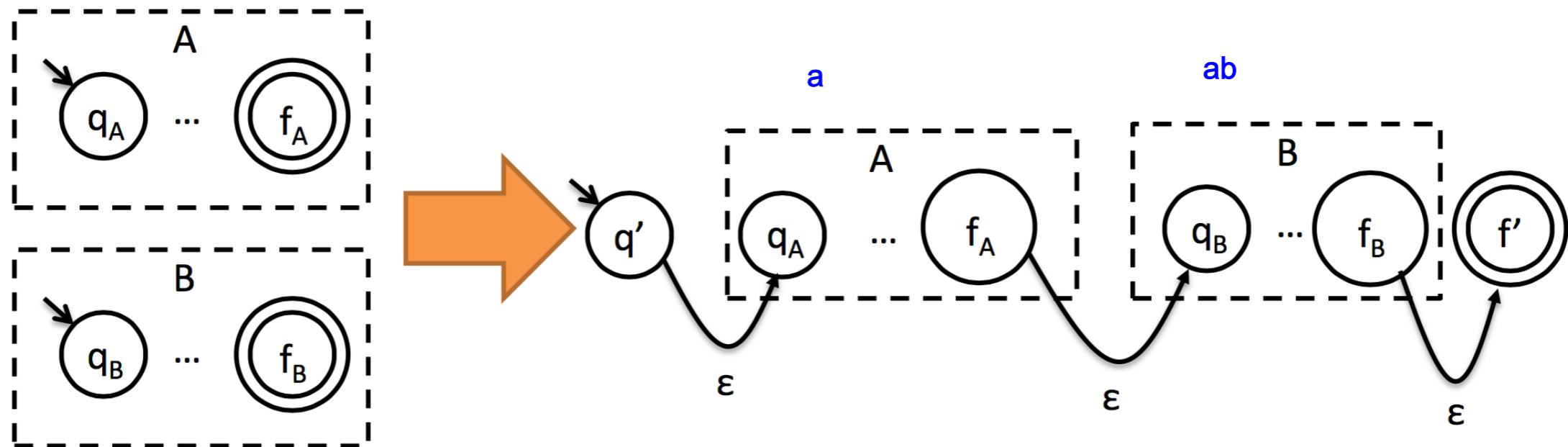
$$F_a, \epsilon \rightarrow f'$$

$$F_b, \epsilon \rightarrow f'$$

# Regexp to NFA rules

Rule for catenation A.B

Regex: aab



Make new start state  $q'$  and new final state  $f'$

Make original final states non-final

Add to  $\delta$ :

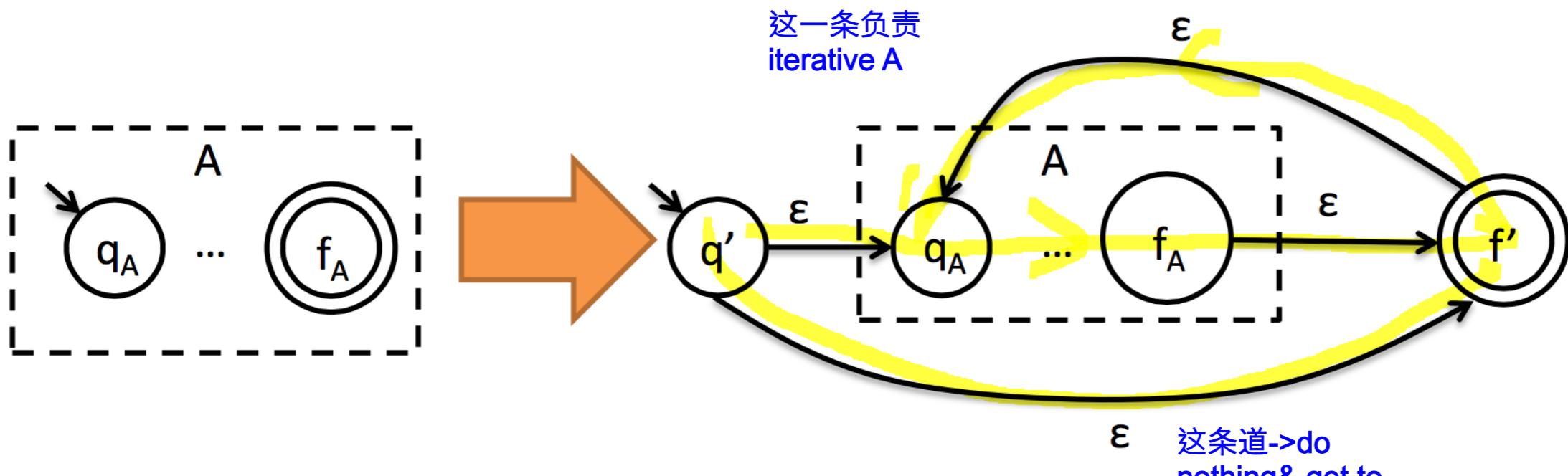
$$q', \epsilon \rightarrow q_A$$

$$f_A, \epsilon \rightarrow q_B$$

$$f_B, \epsilon \rightarrow f'$$

# Regexp to NFA rules

Rule for iteration  $A^*$



Make new start state  $q'$  and new final state  $f'$

Make original final states non-final

Add to  $\delta$ :

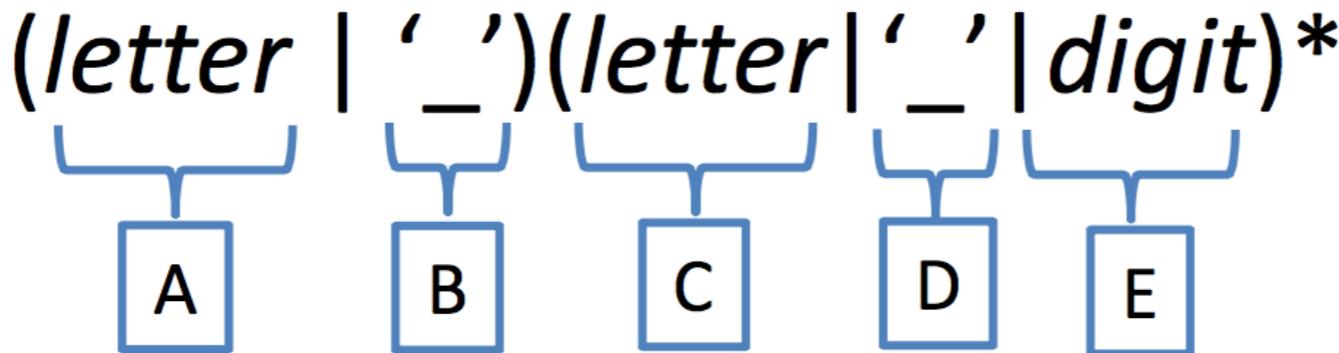
$$\begin{aligned} q', \epsilon &\rightarrow q_A \\ q', \epsilon &\rightarrow f' \\ f', \epsilon &\rightarrow q_A \end{aligned}$$

# Regexp operator precedence

$$s \mid ab \equiv (a) | (ab)$$
$$ab^* \equiv a(b^*)$$

Operator	Precedence	Analogous math operator
	low	addition
.	medium	multiplication
*	high	exponentiation

# Tree representation of a regexp



Operator	Precedence
----------	------------

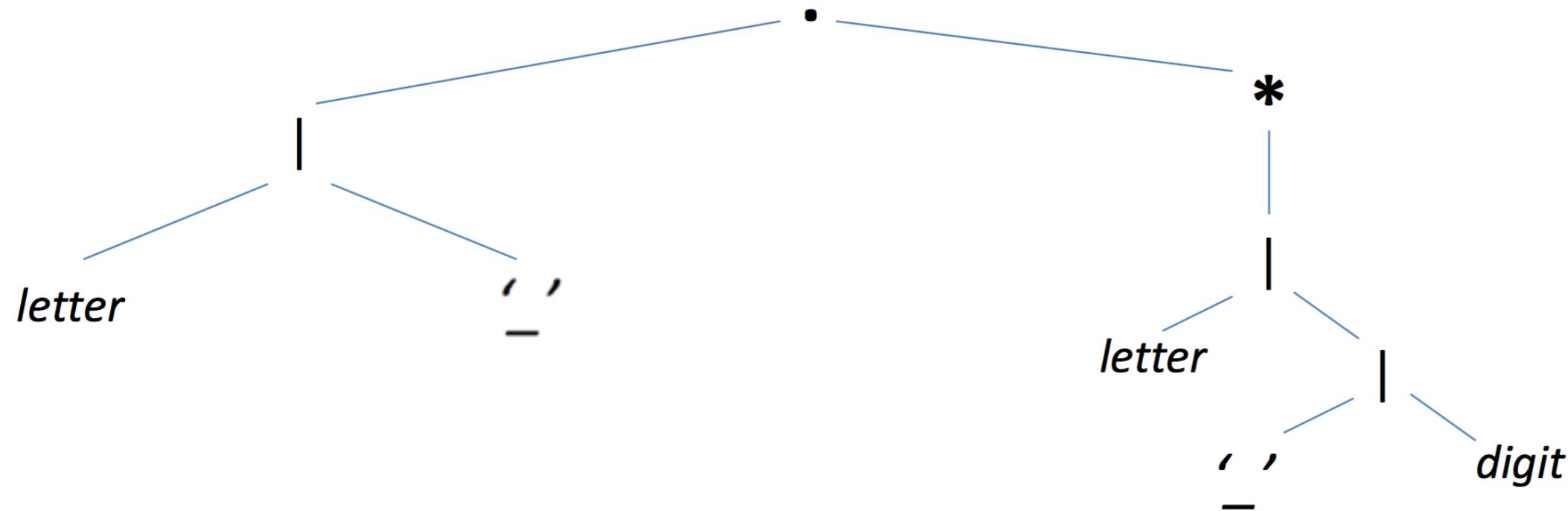
	low
.	medium

*	high
---	------

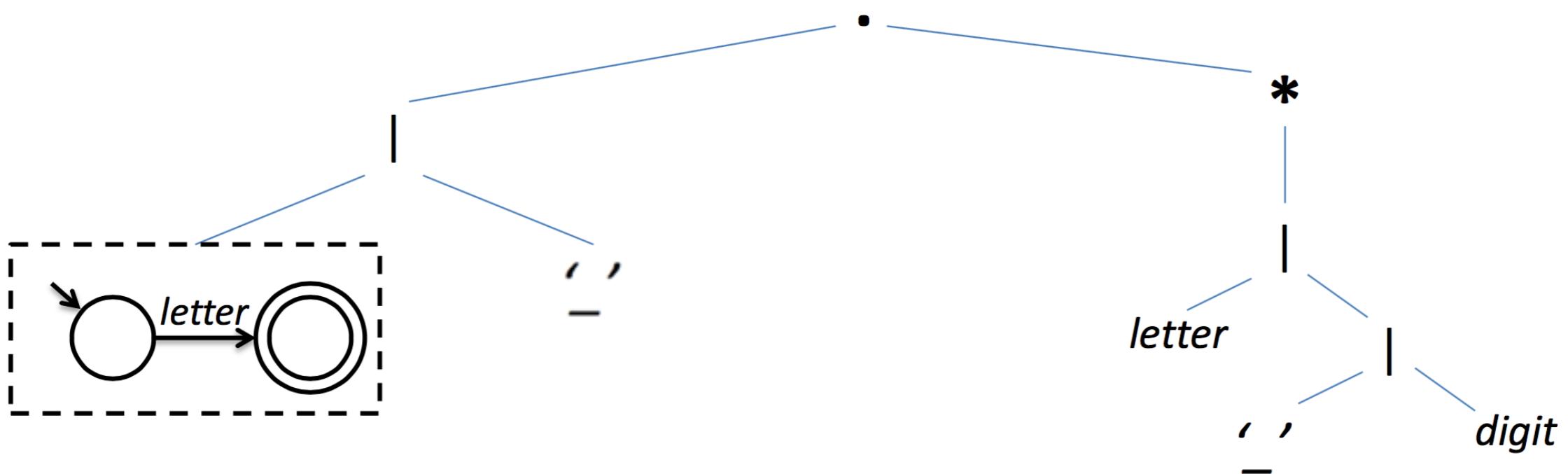
$R \rightarrow \text{TOKENS} \rightarrow$  Tree  $\rightarrow NFA$

convert Regex to NFA

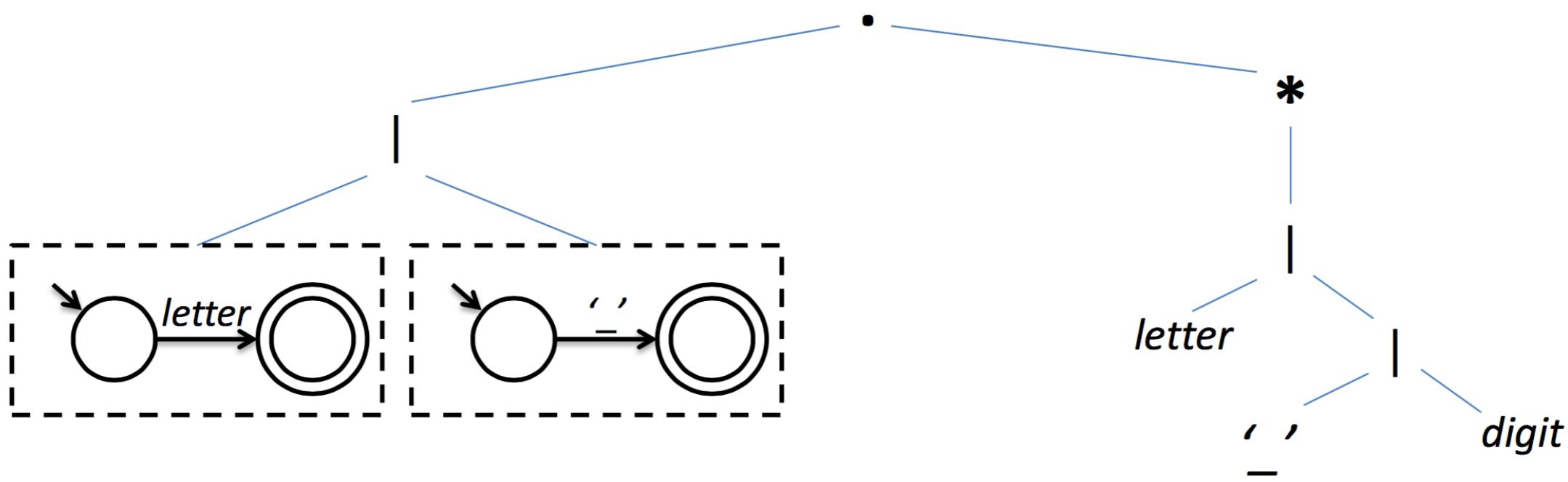
# Bottom-up conversion



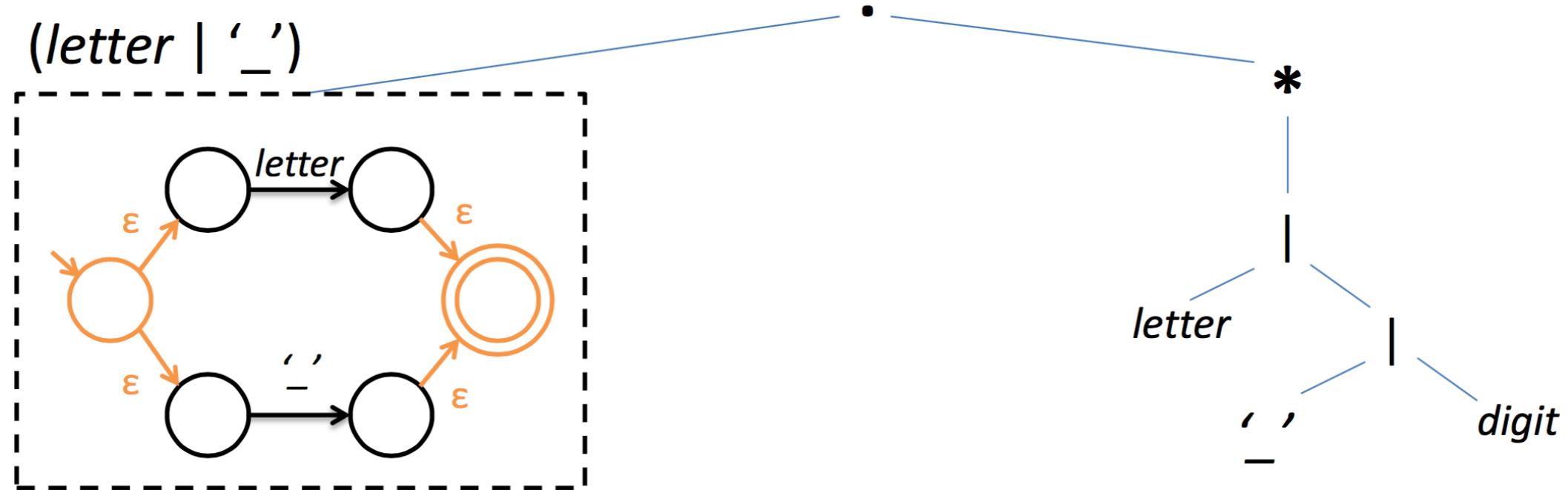
# Bottom-up conversion



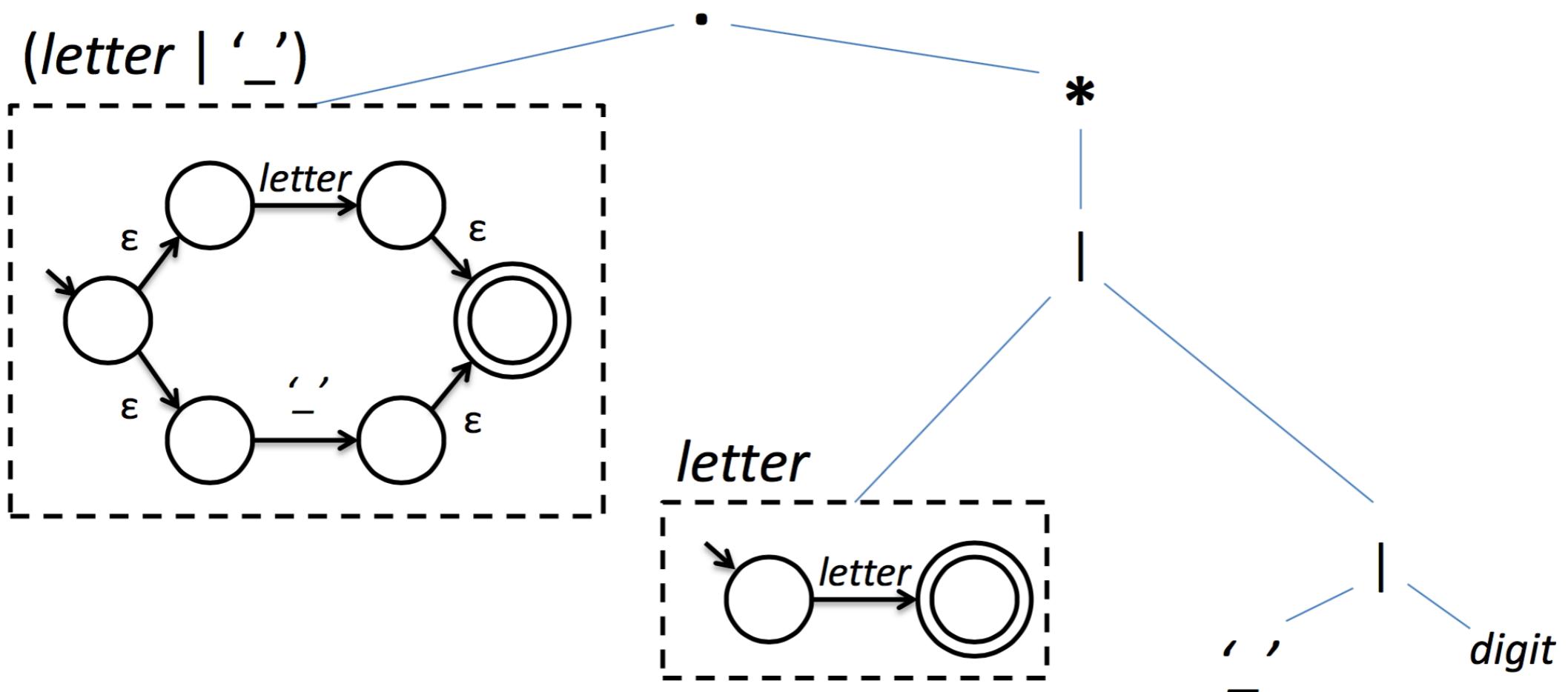
# Bottom-up conversion



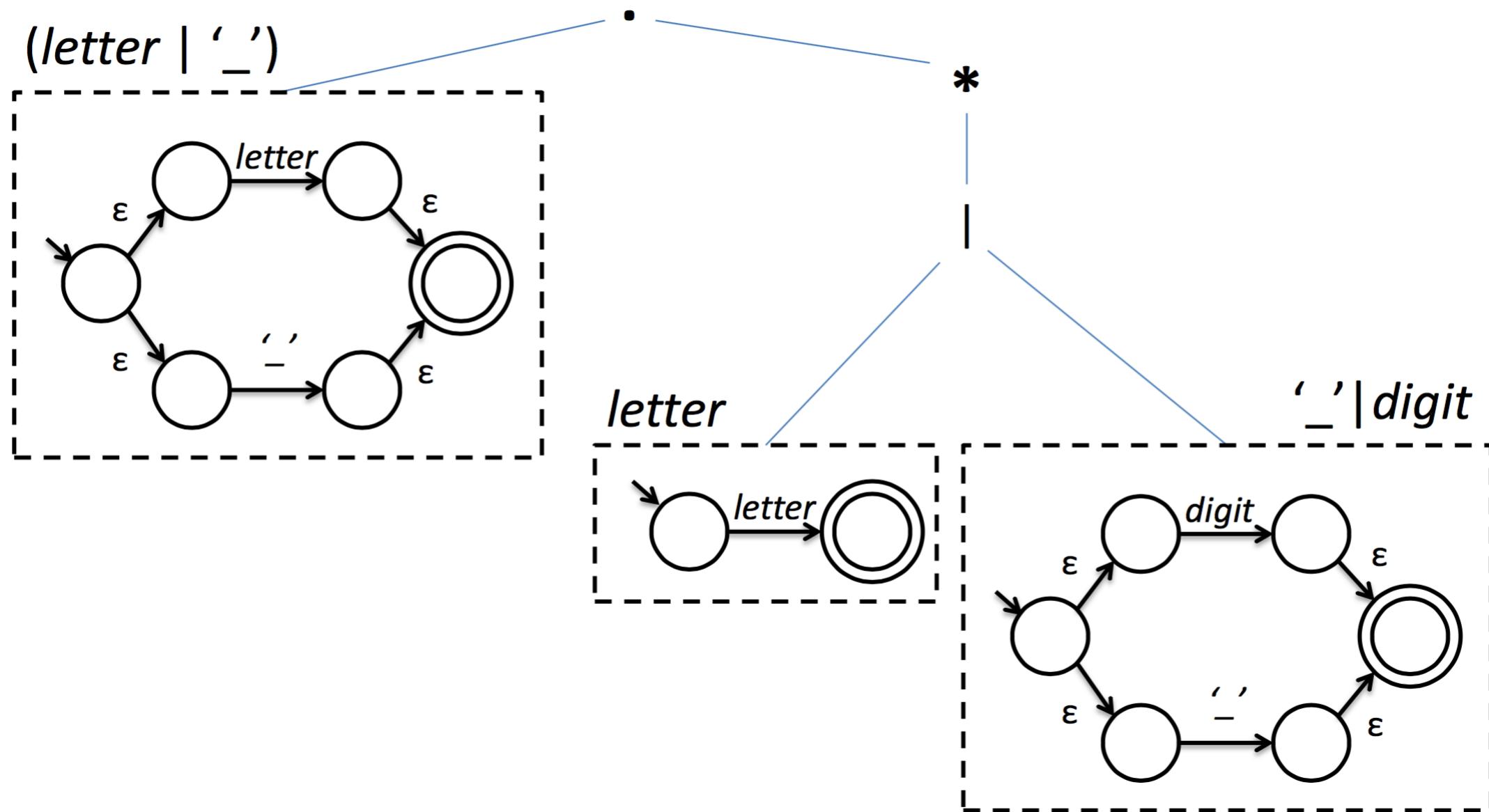
# Bottom-up conversion



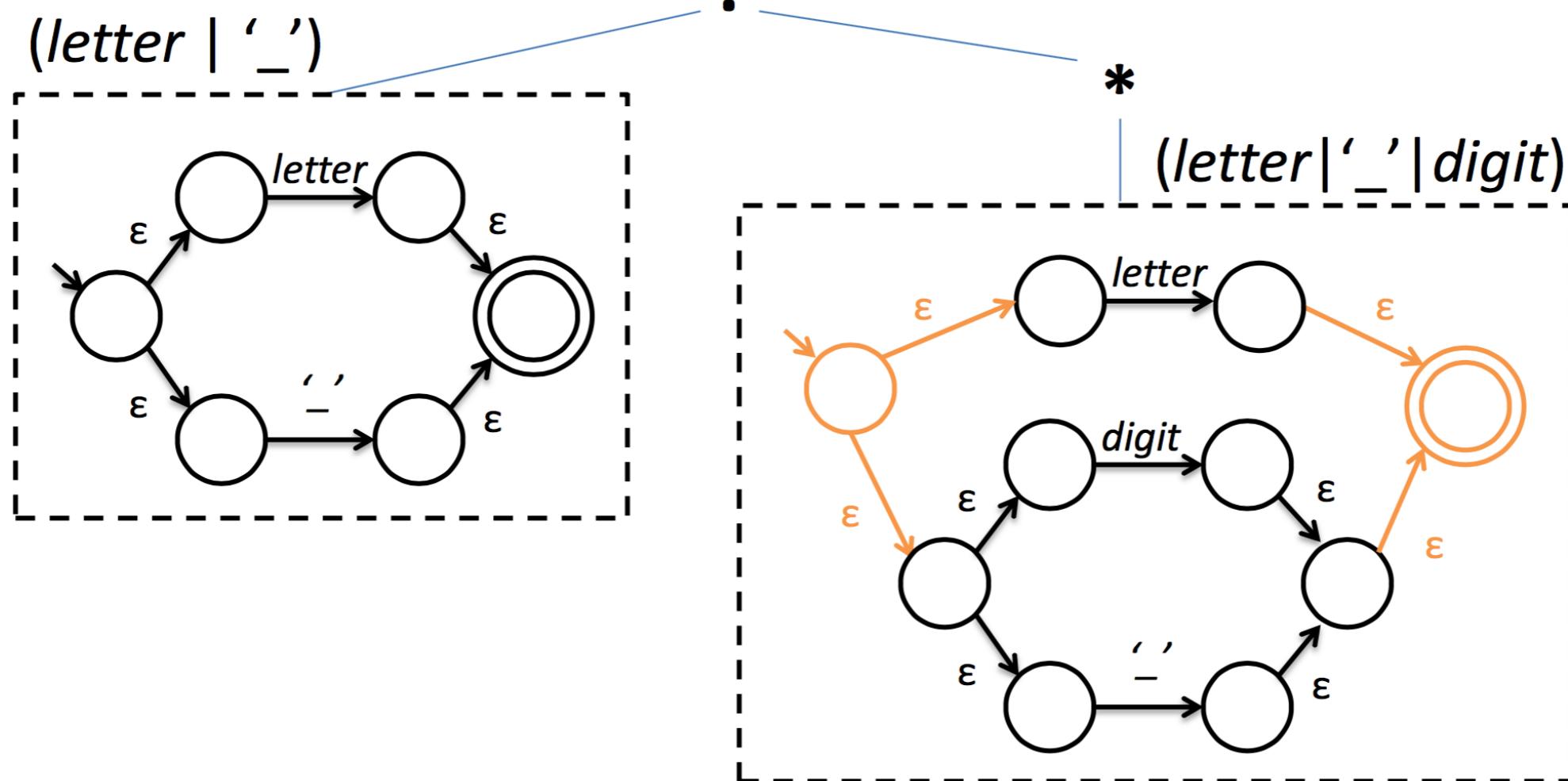
# Bottom-up conversion



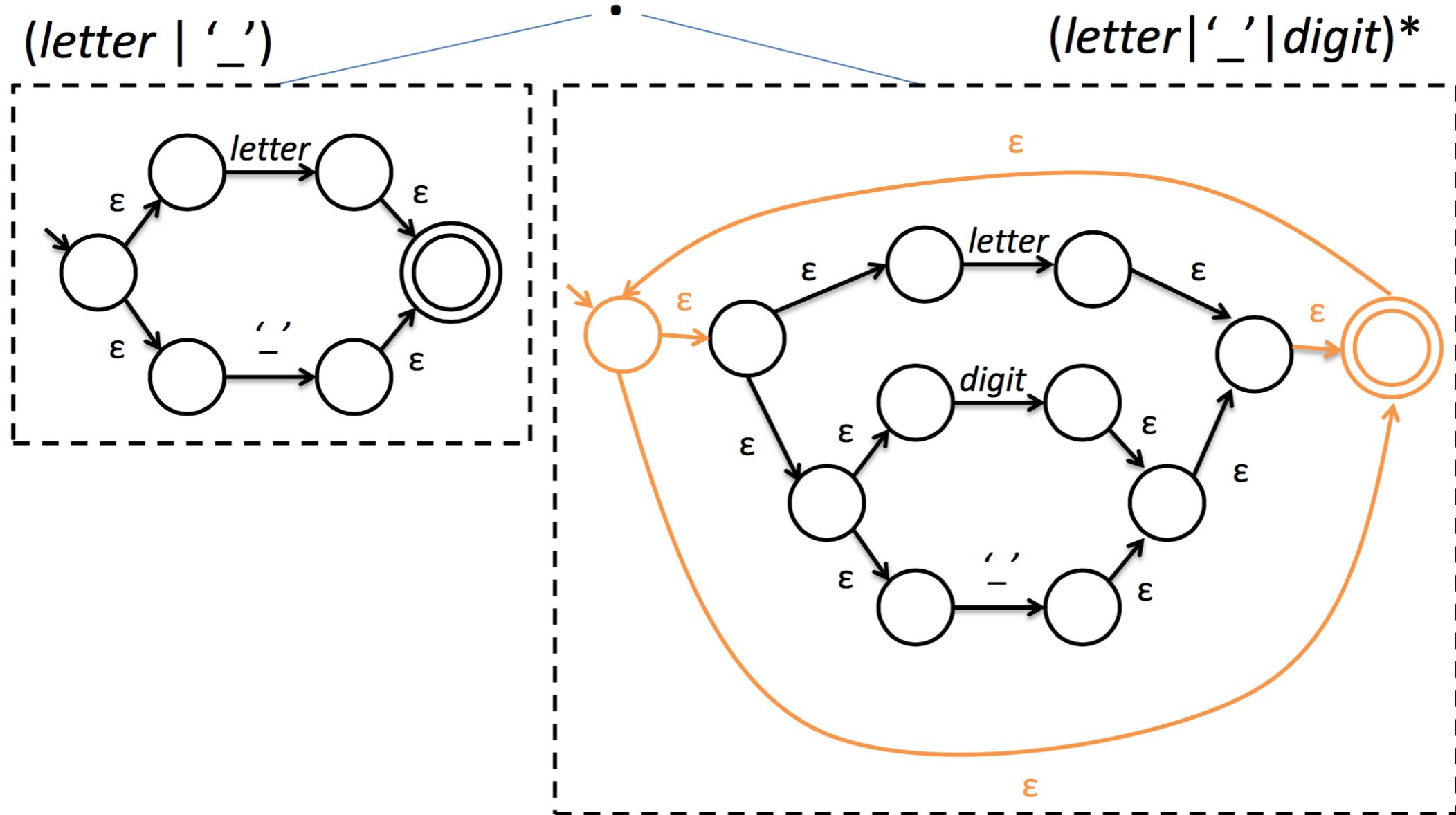
# Bottom-up conversion



# Bottom-up conversion

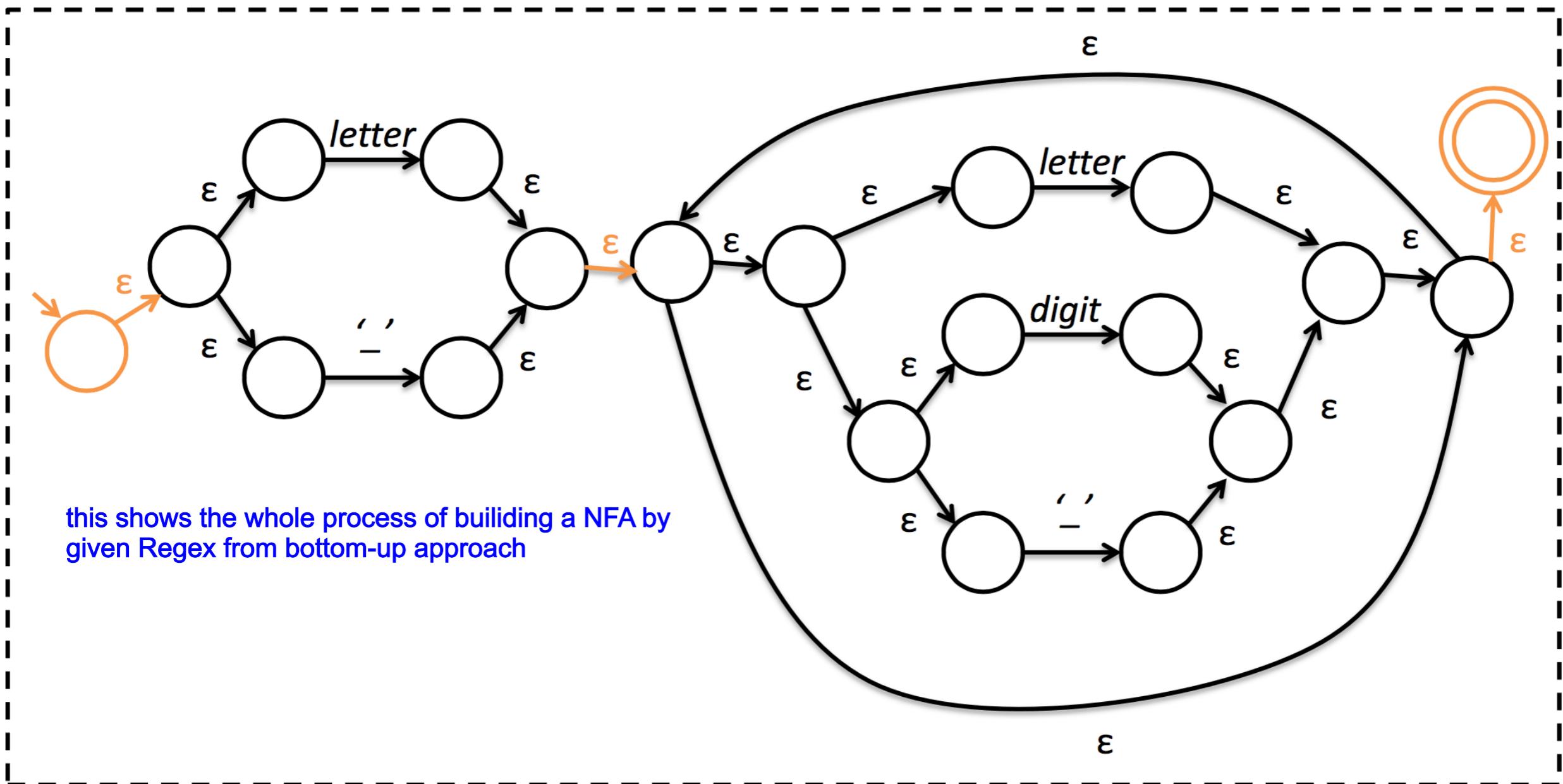


# Bottom-up conversion



# Bottom-up conversion

$(letter \mid '_')(letter \mid '_' \mid digit)^*$

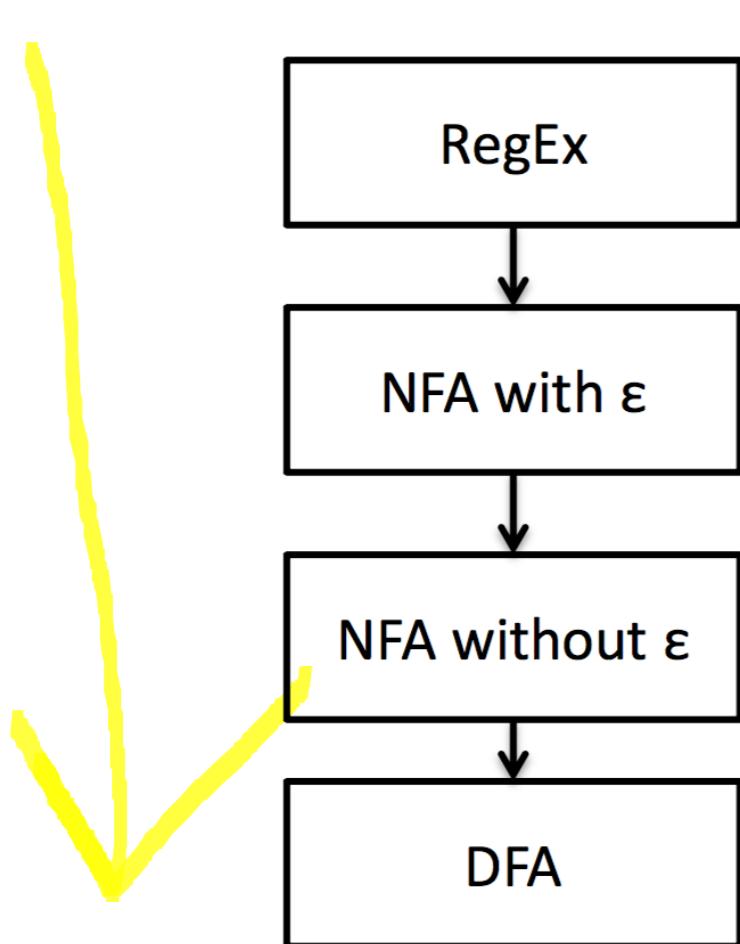


this shows the whole process of building a NFA by given Regex from bottom-up approach

# Regexp to DFAs

We now have an NFA

We need to go to DFA



to day  
last loc

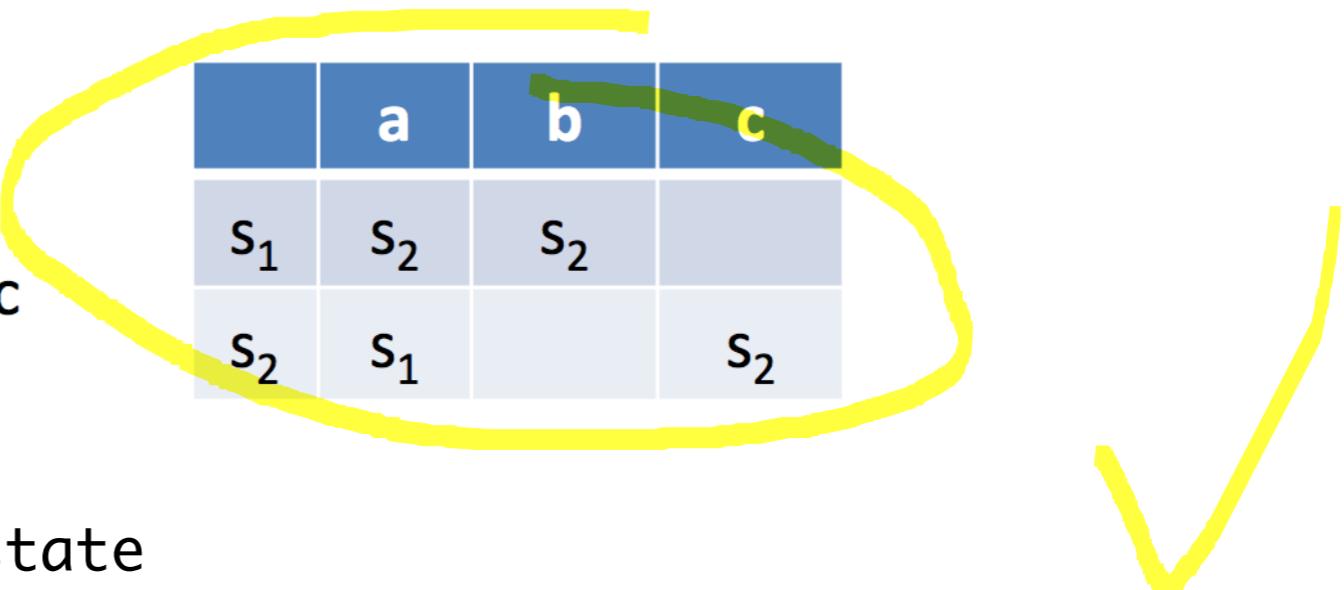
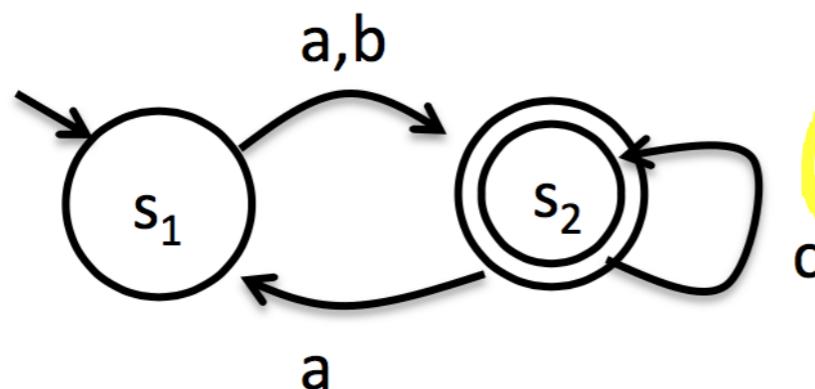
But what's so great about DFAs?

very easy to implement, just a look-up table

# Table-driven DFAs

Recall that  $\delta$  can be expressed as a table

This leads to a very efficient array representation



```
s = start state  
while (more input){  
    c = read char  
    s = table[s][c]  
}  
if s is final, accept
```

# FSMs for tokenization

FSMs only check for language membership of a string

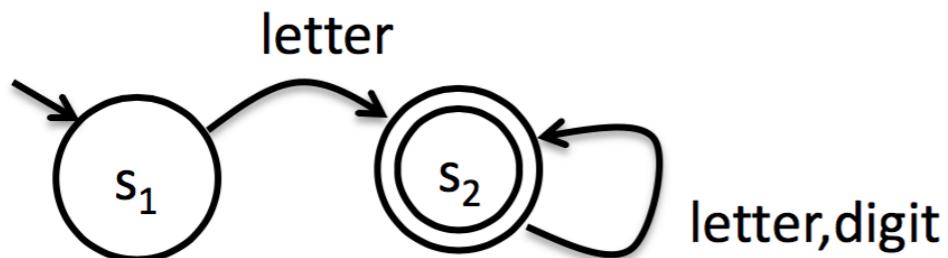
- the scanner needs to recognize a stream of many different tokens using the longest match

- the scanner needs to know what was matched

Idea: imbue states with actions that will fire when state is reached

# A first cut at actions

Consider the language of Pascal identifiers

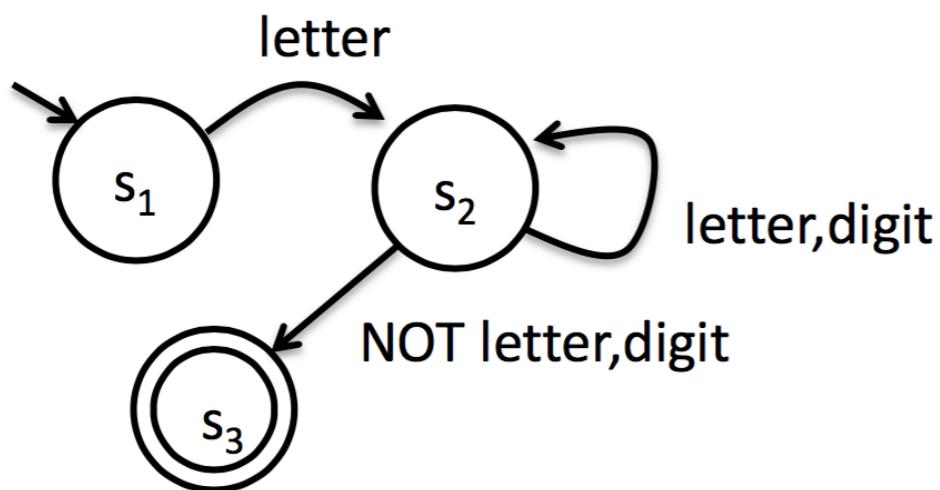


State	Actions
$s_2$	return ID

BAD: not longest match

just find the shortest match

Accounting for longest matches

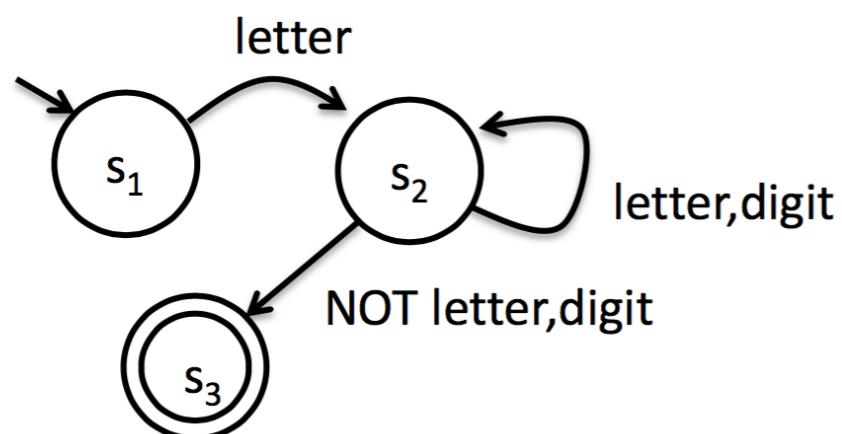


State	Actions
$s_3$	return ID

BAD: maybe we needed that character

# A second take at actions

Give our FSMs ability to put chars back



State	Actions
$s_3$	Put 1 char back, return ID

Since we're allowing our FSM to peek at characters past the end of a valid token, it's also convenient to add an EOF symbol

# Our first scanner

Consider a language with two statements

assignments: ID = expr

increments: ID += expr

where **expr** is of the form

ID + ID

ID ^ ID

ID < ID

ID <= ID

Token name	Regular Expression
ASSIGN	=
INC	+=
PLUS	+
EXP	^
LT	<
LEQ	<=
ID	(letter _)(letter digit _)*

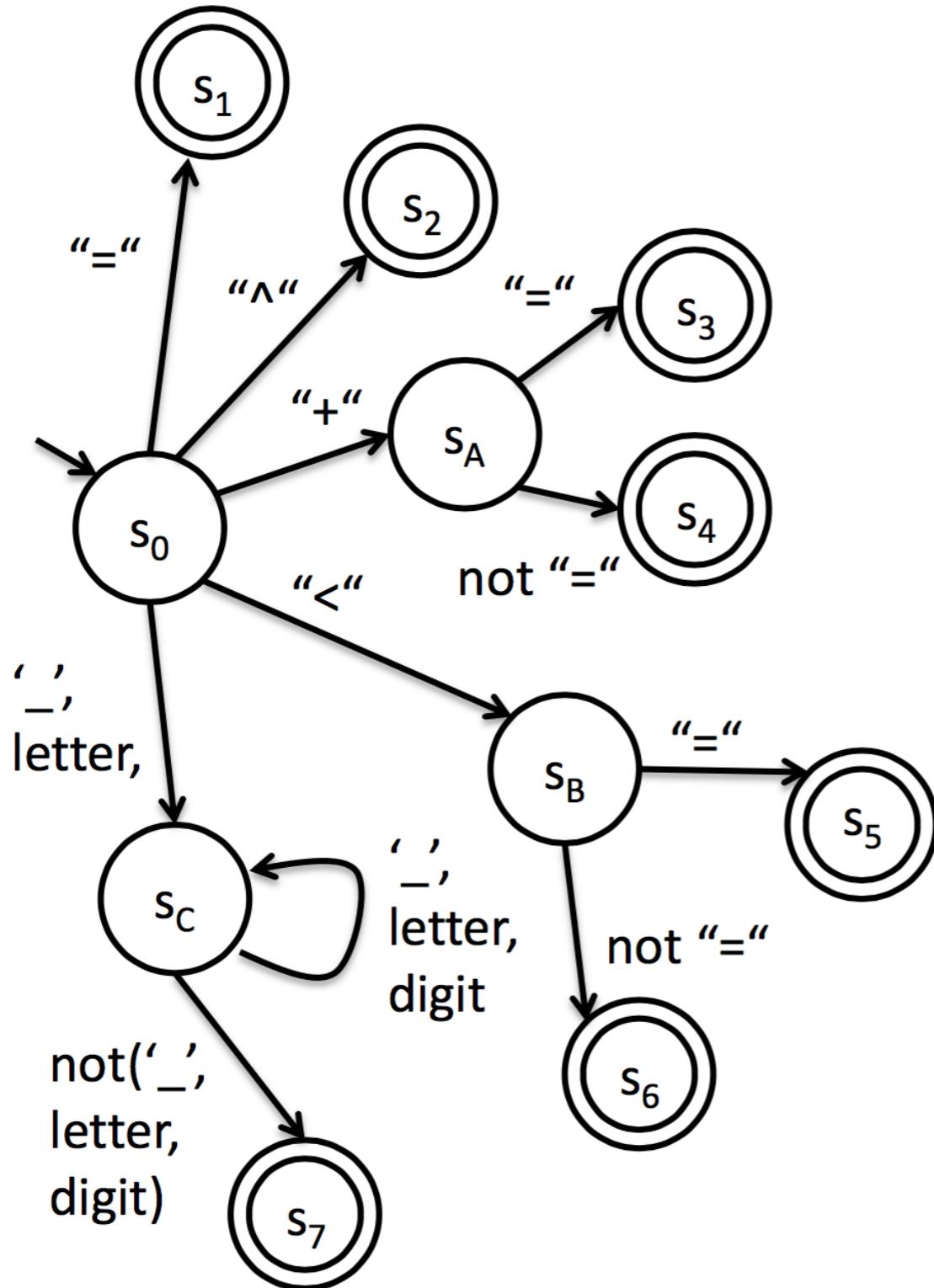
Identifiers ID follow C conventions

# Combined DFA

理解put back 1 char: 拿s4做例子

1+2=3 此时dfa接到+和 (not =) <这里是2>, 所以dfa判断这个符号并不是+=, 而是+, 但此时, 2仍然被程序需要, 但我们已经走过2了, 所以我们必须回退一次, 重新拿到2, 才能正确执行1+2

所以很好理解s7, s6为什么也需要这么做, 1+, 我们仍需要判定+, 1为ID, 所以也需要put back 1 char

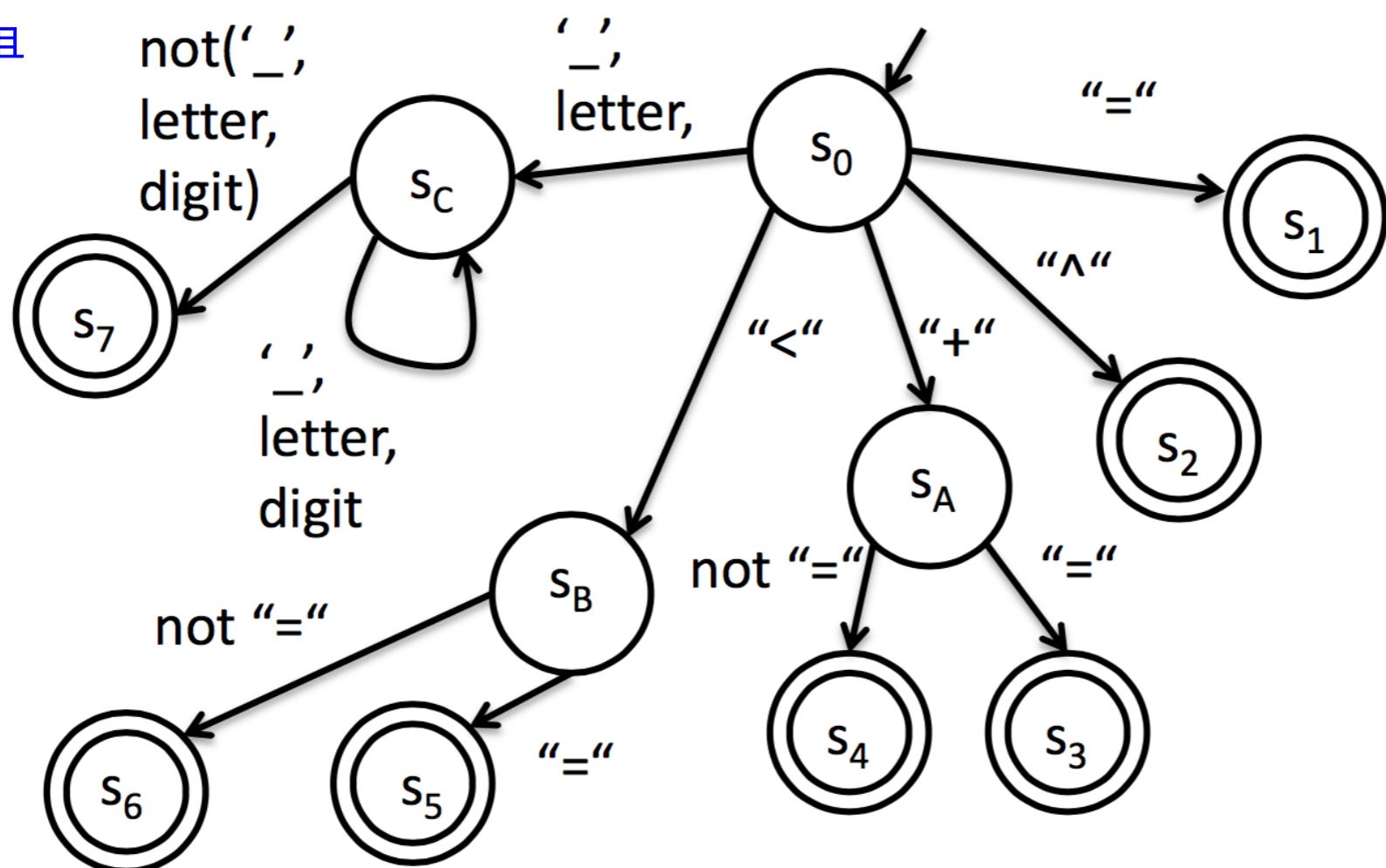


Token name	Regular Expression
ASSIGN	"="
INC	"+="
PLUS	"+"
EXP	"^"
LT	"<"
LEQ	"<="
ID	(letter _)(letter digit _)*

State	Action
S1	return ASSIGN
S2	return EXP
S3	return INC
S4	put back 1 char, return PLUS
S5	Return LEQ
S6	put back 1 char, return LT
S7	put back 1 char, return ID

	=	+	^	<	-	letter	digit	EOF	none
S <sub>0</sub>	Ret ASSIGN	S <sub>A</sub>	Ret EXP	S <sub>B</sub>	S <sub>C</sub>	S <sub>C</sub>		Ret EOF	
S <sub>A</sub>	Ret INC	Back 1, Ret PLUS	Back 1, Ret PLUS						
S <sub>B</sub>	Ret LEQ	Back 1, Ret LT	Back1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT
S <sub>C</sub>	Back 1, Ret ID	Back 1 Ret ID	Back 1, Ret ID	Back 1, Ret ID	S <sub>C</sub>	S <sub>C</sub>	S <sub>C</sub>	Back 1, Ret ID	Back 1, Ret ID

将指针向前移一位，并且  
返回ID(x)这个token



	=	+	^	<	-	letter	digit	EOF	none
S <sub>0</sub>	Ret ASSIGN	S <sub>A</sub>	Ret EXP	S <sub>B</sub>	S <sub>C</sub>	S <sub>C</sub>		Ret EOF	
S <sub>A</sub>	Ret INC	Back 1, Ret PLUS	Back 1, Ret PLUS						
S <sub>B</sub>	Ret LEQ	Back 1, Ret LT	Back1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT	Back 1, Ret LT
S <sub>C</sub>	Back 1, Ret ID	Back 1 Ret ID	Back 1, Ret ID	Back 1, Ret ID	S <sub>C</sub>	S <sub>C</sub>	S <sub>C</sub>	Back 1, Ret ID	Back 1, Ret ID

```

do{
    read char
    perform action / update state
    if (action was to return a token){
        start again in start state
    }
} (while not EOF or stuck);

```

# Lexical analyzer generators

aka scanner generators

The transformation from regexp to scanner is formally defined

Can write tools to synthesize a lexer automatically

Lex: unix scanner generator

Flex: fast lex

JLex: Java version of Lex

# JLex

Declarative specification

tell it what you want scanned, it will figure out the rest

**Input:** set of regexps + associated actions

xyz.jlex file

**Output:** Java source code for a scanner

xyz.jlex.java source code of scanner

# jlex format

3 sections separated by %%

user code section

directives

regular expressions + actions

```
//User Code Section (uninterpreted java code)
```

```
%%
```

```
//Directives Section
```

```
DIGIT = [0-9]  
LETTER = [a-zA-Z]  
WHITESPACE = [\040\t\n]
```

} Macro definitions

```
%state SPECIALINTSTATE — State declaration
```

```
//Configure for use with java CUP (Parser generator)  
%implements java_cup.runtime.Scanner  
%function next_token  
%type java_cup.runtime.Symbol
```

```
//End of file behavior
```

```
%eofval{  
System.out.println("All done");  
return null;  
%eofval}
```

```
//Turn on line counting  
%line
```

```
%%
```

```
//Regular Expression rules
```

# Rules section

Format is <regex>{code} where regex is a regular expression for a single token

can use macros from the directive sections in regex, surround with curly braces

## Conventions

chars represent themselves (except special characters)

chars inside “” represent themselves (except \)

## Regexp operators

| \* + ? () .

## Character class operators

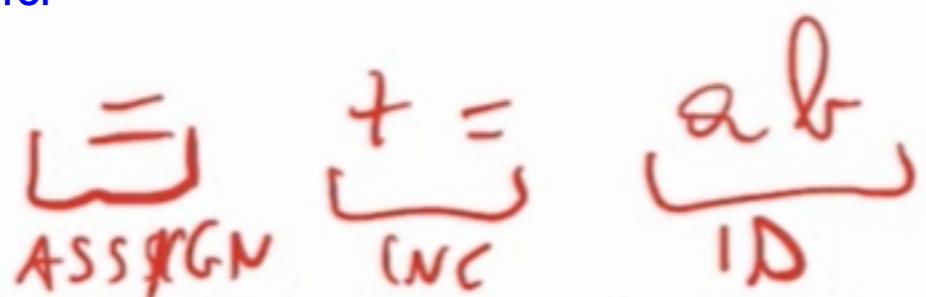
- range

^ not

\ escape

## REGEX

	action (written in Java)
"=	{ System.out.println(yyline + 1 + ": ASSIGN"); }
"+"	{ System.out.println(yyline + 1 + ": PLUS"); }
"^"	{ System.out.println(yyline + 1 + ": EXP"); }
"<"	{ System.out.println(yyline + 1 + ": LT"); }
"+="	{ System.out.println(yyline + 1 + ": INC"); }
"<="	{ System.out.println(yyline + 1 + ": LEQ"); }
{WHITESPACE}	{ } here we do nothing
({LETTER} "_") ({DIGIT} {LETTER} "_")*	{ System.out.println(yyline+1 + ": ID " + yytext());} if we find that we didnt match anything above, then it is an error { System.out.println(yyline + 1 + ": badchar"); }

  
ASSIGN   INC   ID

# Announcements

- P2 posted
- HW2 posted

# Context-free grammars (CFGs)

# Roadmap



Regex ~ Scanner

CFG ~ Parser

JLex ->scanner  
Java CUP ->parser

## Last time

- Regex == DFA
- JLex: a tool for generating (Java code for) Lexers/  
Scanners

## This time context free grammar

- CFGs, the underlying abstraction for Parsers

## Next week

- Java CUP: A tool for generating (Java code for) parser

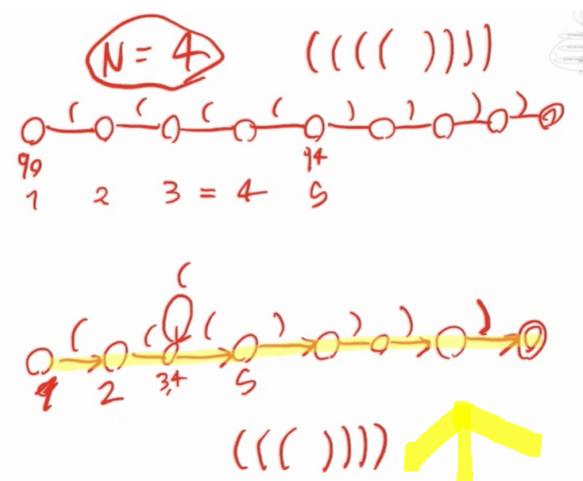
# RegExs Are Great!

Perfect for tokenizing a language

They do have some limitations

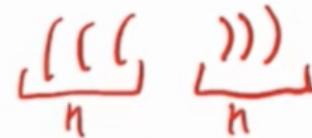
- Limited class of language that cannot specify all programming constructs we need
- No notion of structure

Let's explore both of these issues



# Limitations of RegExps

Cannot handle “matching”



E.g., language of balanced parentheses

$$L_{()} = \{ ({}^n )^n \text{ where } n > 0 \}$$

no DFA -> 让这些括号数量平衡

No DFA exists for this language

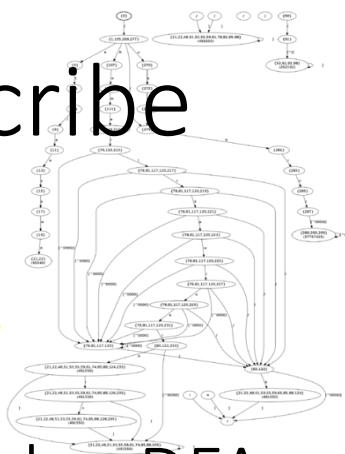
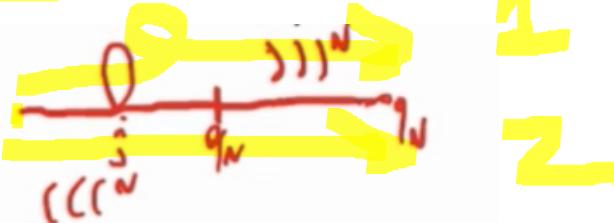
Intuition: A given FSM only has a fixed, finite amount of memory

- For an FSM, memory = the states
- With a fixed, finite amount of memory, how could an FSM remember how many “(“ characters it has seen?

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

By contradiction:

- Let's say there exists a DFA A for  $L_{()}$  and such a DFA has  $N$  states
- A has to accept the string  $(^N)^N$  with some path  $q_0q_1\dots q_N\dots q_{2N}$
- By *pigeonhole principle* some state has repeated:  $q_i = q_j$  for some  $i < j \leq N$
- Therefore the run  $q_0q_1\dots q_i q_{j+1}\dots q_N\dots q_{2N}$  is also accepting
- A accepts the string  $(^{N-(j-i)})^N$  not in  $L_{()}$  → contradiction!

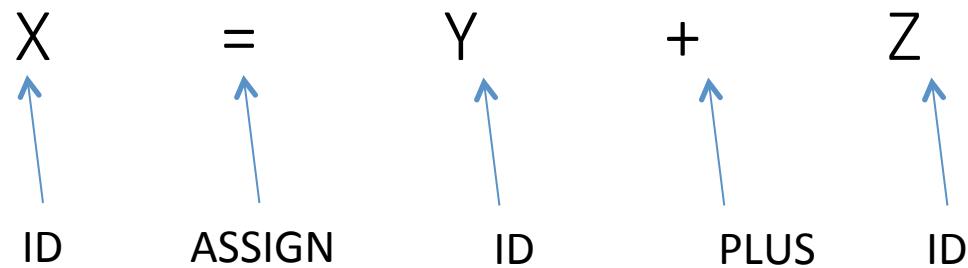


so 2 ways (one with the loop; and one without the loop) will both be accepted and the first one is contradicted

here we have 0-n, total of n+1 q's, but we have only n states, so there must be  $q_i, q_j (i \neq j)$  that is repeated;

# Limitations of RegEx: Structure

Our Enhanced-RegEx scanner can emit a stream of tokens:



... but this doesn't really enforce any order of operations

# The Chomsky Hierarchy



Turing machine



## LANGUAGE CLASS:

Recursively enumerable

expressive power

power

efficiency



Context-Sensitive

Context-Free

Regular

Happy medium?

context free 是一个很好的折中点

FSM



Noam Chomsky

# Context Free Grammars (CFGs)

A set of (recursive) rewriting rules to generate patterns of strings<sup>1</sup>

Can envision a “parse tree” that keeps structure

# CFG: Intuition

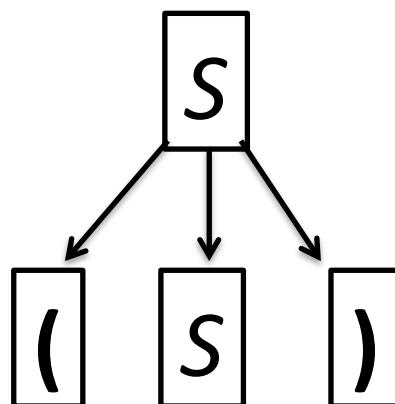
$$S \rightarrow (' S ')$$

A *rule* that says that you can rewrite S to be an S surrounded by a single set of parenthesis

Before applying rule

S

After applying rule



# Context Free Grammars (CFGs)

A CFG is a 4-tuple  $(N, \Sigma, P, S)$

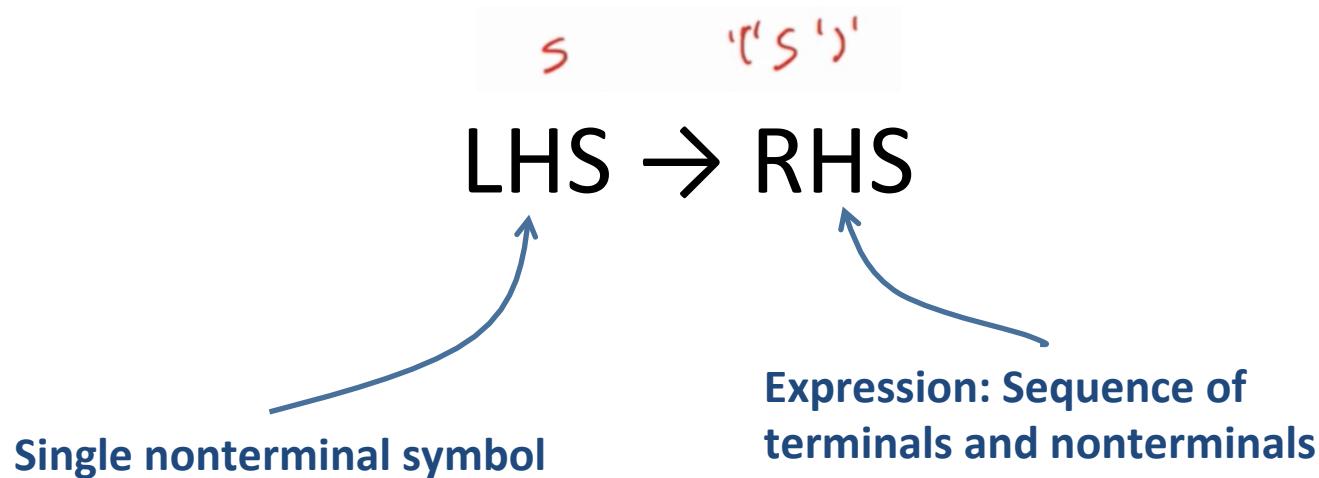
- $N$  is a set of non-terminals, e.g.,  $A, B, S\dots$
- $\Sigma$  is the set of terminals
- $P$  is a set of production rules
- $S$  (in  $N$ ) is the initial non-terminal symbol

# Context Free Grammars (CFGs)

A CFG is a 4-tuple  $(N, \Sigma, P, S)$

- $N$  is a set of non-terminals, e.g.,  $A, B, S\dots$
  - $\Sigma$  is the set of terminals
  - $P$  is a set of production rules
  - $S$  (in  $N$ ) is the initial non-terminal symbol
- If not otherwise specified, use the non-terminal that appears on the LHS of the first production as the start
- 
- The diagram consists of four blue curved arrows pointing from text labels to their corresponding components in the list:
  - An arrow points from "Placeholder / interior nodes in the parse tree" to the element  $N$ .
  - An arrow points from "Tokens from scanner" to the element  $\Sigma$ .
  - An arrow points from "Rules for deriving strings" to the element  $P$ .
  - An arrow points from the explanatory text at the bottom to the element  $S$ .

# Production Syntax



Examples:

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

$$S \rightarrow \epsilon$$

# Production Shorthand

Nonterm  $\rightarrow$  expression

$s \rightarrow (' s ')$

Nonterm  $\rightarrow \epsilon$

$s \rightarrow \epsilon$

*equivalently:*

Nonterm  $\rightarrow$  expression

$s \rightarrow (' s ')$

$| \epsilon$

$| \epsilon$

*equivalently:*

Nonterm  $\rightarrow$  expression  $| \epsilon$

$s \rightarrow (' s ') | \epsilon$

# Derivations

To derive a string:

- Start by setting “*Current Sequence*” to the start symbol
- Repeat:
  - Find a Nonterminal X in the Current Sequence
  - Find a production of the form  $X \rightarrow \alpha$
  - “Apply” the production: create a new “current sequence” in which  $\alpha$  replaces X
- Stop when there are no more non-terminals
- This process derives a string of terminal symbols

$$S \rightarrow (S) \rightarrow ((S)) \xrightarrow{S \rightarrow \epsilon} (())$$

$(\Sigma \cup N)^*$

$\Rightarrow$   
 $\Rightarrow^+$

# Derivation Syntax

these 2 are the same,  
the grammar moves  
one step

- We'll use the symbol  $\Rightarrow^+$  for *derives*
- We'll use the symbol  $\Rightarrow^+ +$  for *derives in one or more steps*
- We'll use the symbol  $\Rightarrow^* +$  for *derives in zero or more steps*

.....

$S \Rightarrow (S)$

$S \Rightarrow^+ ((S))$

$S \Rightarrow^* S$

$S \Rightarrow^+ (( ))$

$\Rightarrow$   
 $\Rightarrow^+$   
 $\Rightarrow^*$

# An Example Grammar

# An Example Grammar

## Terminals

begin

end

semicolon

assign

id

plus

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin**

**end**

**semicolon**

**assign**

**id**

**plus**

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin** } Program  
**end** } boundary

**semicolon**

**assign**

**id**

**plus**

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin** } Program  
**end** } boundary

**semicolon**      Represents “;”  
**assign**           Separates statements  
**id**  
**plus**

# An Example Grammar

For readability, bold and lowercase

## Terminals

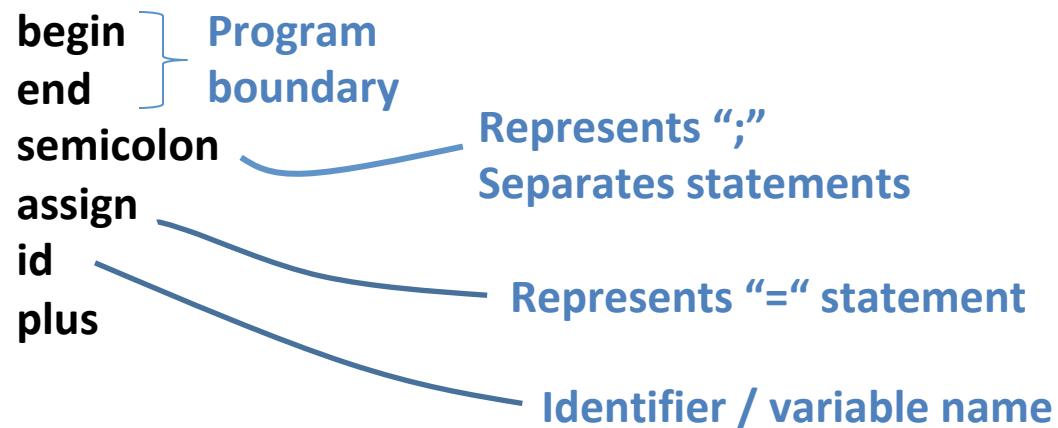
**begin**    } Program  
**end**      boundary  
**semicolon**  
**assign**  
**id**  
**plus**

Represents ";"  
Separates statements  
Represents "=" statement

# An Example Grammar

For readability, bold and lowercase

## Terminals



# An Example Grammar

For readability, bold and lowercase

## Terminals

<b>begin</b>	Program boundary	
<b>end</b>		
<b>semicolon</b>		Represents ";" Separates statements
<b>assign</b>		
<b>id</b>		Represents "=" statement
<b>plus</b>		Identifier / variable name Represents "+" expression

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin**

**end**

**semicolon**

**assign**

**id**

**plus**

## Nonterminals

*Prog*

*Stmts*

*Stmt*

*Expr*

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin**

**end**

**semicolon**

**assign**

**id**

**plus**

For readability, Italics and UpperCamelCase

## Nonterminals

*Prog*

*Stmts*

*Stmt*

*Expr*

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin**

**end**

**semicolon**

**assign**

**id**

**plus**

For readability, Italics and UpperCamelCase

## Nonterminals

*Prog* ————— Root of the parse tree

*Stmts*

*Stmt*

*Expr*

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin**

**end**

**semicolon**

**assign**

**id**

**plus**

For readability, Italics and UpperCamelCase

## Nonterminals

*Prog* ————— Root of the parse tree

*Stmts* ————— List of statements

*Stmt*

*Expr*

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin**

**end**

**semicolon**

**assign**

**id**

**plus**

For readability, Italics and UpperCamelCase

## Nonterminals

*Prog* ————— Root of the parse tree

*Stmts* ————— List of statements

*Stmt* ————— A single statement

*Expr*

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin**

**end**

**semicolon**

**assign**

**id**

**plus**

For readability, Italics and UpperCamelCase

## Nonterminals

*Prog* ————— Root of the parse tree

*Stmts* ————— List of statements

*Stmt* ————— A single statement

*Expr* ————— A mathematical expression

# An Example Grammar

For readability, bold and lowercase

## Terminals

**begin**

**end**

**semicolon**

**assign**

**id**

**plus**

Defines the syntax of legal programs

## Productions

$\text{Prog} \rightarrow \mathbf{begin} \text{Stmts end}$

$\text{Stmts} \rightarrow \text{Stmts} \mathbf{semicolon} \text{Stmt}$   
| Stmt

$\text{Stmt} \rightarrow \mathbf{id} \text{ assign Expr}$

$\text{Expr} \rightarrow \mathbf{id}$

| Expr plus id

For readability, Italics and UpperCamelCase

## Nonterminals

*Prog*

*Stmts*

*Stmt*

*Expr*

# An Example Grammar

For readability, bold and lowercase

## Terminals

begin	Program boundary
end	
semicolon	Represents ";" Separates statements
assign	Represents "=" statement
id	Identifier / variable name
plus	Represents "+" expression

For readability, Italics and UpperCamelCase

## Nonterminals

Prog	Root of the parse tree
Stmts	List of statements
Stmt	A single statement
Expr	A mathematical expression

Defines the syntax of legal programs

## Productions

$Prog \rightarrow begin\ Stmt\ s\ end$
$Stmt\ s \rightarrow Stmt\ s\ semicolon\ Stmt$
Stmt
$Stmt \rightarrow id\ assign\ Expr$
$Expr \rightarrow id$
Expr plus id

官方定义；牢记



## Productions

1.  $\text{Prog} \rightarrow \mathbf{begin} \text{Stmts} \mathbf{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts} \mathbf{semicolon} \text{Stmt}$
3.           |  $\text{Stmt}$
4.  $\text{Stmt} \rightarrow \mathbf{id} \mathbf{assign} \text{Expr}$
5.  $\text{Expr} \rightarrow \mathbf{id}$
6.           |  $\text{Expr} \mathbf{plus} \mathbf{id}$

## Productions

1.  $\text{Prog} \rightarrow \mathbf{begin} \text{Stmts end}$
2.  $\text{Stmts} \rightarrow \text{Stmts semicolon Stmt}$
3.           |  $\text{Stmt}$
4.  $\text{Stmt} \rightarrow \mathbf{id assign Expr}$
5.  $\text{Expr} \rightarrow \mathbf{id}$
6.           |  $\text{Expr plus id}$

## Derivation Sequence

## Productions

## Parse Tree

1.  $\text{Prog} \rightarrow \mathbf{begin} \text{Stmts end}$
2.  $\text{Stmts} \rightarrow \text{Stmts} \mathbf{semicolon} \text{Stmt}$
3.           |  $\text{Stmt}$
4.  $\text{Stmt} \rightarrow \mathbf{id assign Expr}$
5.  $\text{Expr} \rightarrow \mathbf{id}$
6.           |  $\text{Expr} \mathbf{plus id}$

## Derivation Sequence

## Productions

## Parse Tree

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts} \text{ end}$
2.  $\text{Stmts} \rightarrow \text{Stmts} \text{ semicolon } \text{Stmt}$
3.           |  $\text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.           |  $\text{Expr plus id}$

## Derivation Sequence

## Key

terminal

Nonterminal

Rule used

## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.           |  $\text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.           |  $\text{Expr plus id}$

## Parse Tree

Prog

## Derivation Sequence

Prog

## Key

terminal

Nonterminal

Rule used

## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts end}$
2.  $\text{Stmts} \rightarrow \text{Stmts semicolon Stmt}$
3.           |  $\text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.           |  $\text{Expr plus id}$

## Parse Tree

Prog

## Derivation Sequence

$\text{Prog} \Rightarrow^+ \text{begin } \text{Stmts end}$

1

## Key

terminal

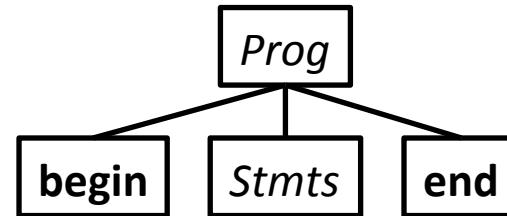
Nonterminal

Rule used

## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$

## Parse Tree



## Derivation Sequence

$\text{Prog} \Rightarrow^+ \text{begin } \text{Stmts } \text{end}$

1

## Key

terminal

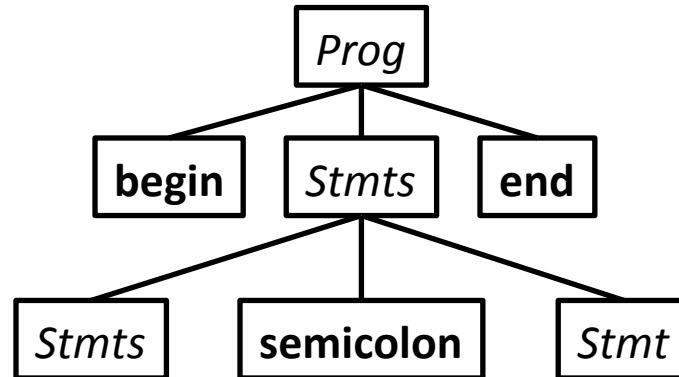
Nonterminal

Rule used

## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$

## Parse Tree



## Derivation Sequence

$\text{Prog} \Rightarrow^+ \text{begin } \text{Stmts } \text{end}$  1  
 $\Rightarrow^+ \text{begin } \text{Stmts } \text{semicolon } \text{Stmt } \text{end}$  2

## Key

terminal

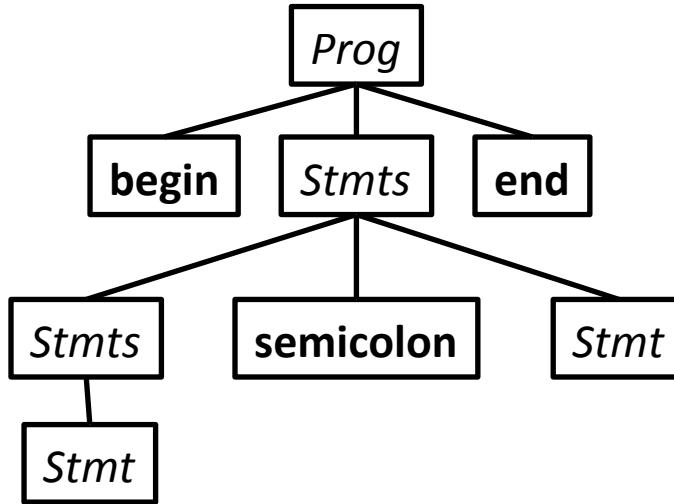
Nonterminal

Rule used

## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$

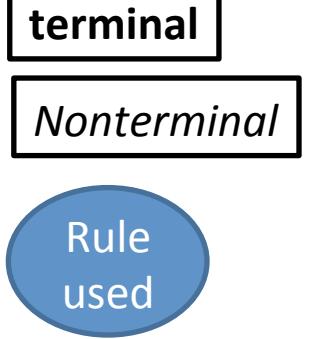
## Parse Tree



## Derivation Sequence

$\text{Prog} \Rightarrow^+ \text{begin } \text{Stmts } \text{end}$  ①  
 $\Rightarrow^+ \text{begin } \text{Stmts } \text{semicolon } \text{Stmt } \text{end}$  ②  
 $\Rightarrow^+ \text{begin } \text{Stmt } \text{semicolon } \text{Stmt } \text{end}$  ③

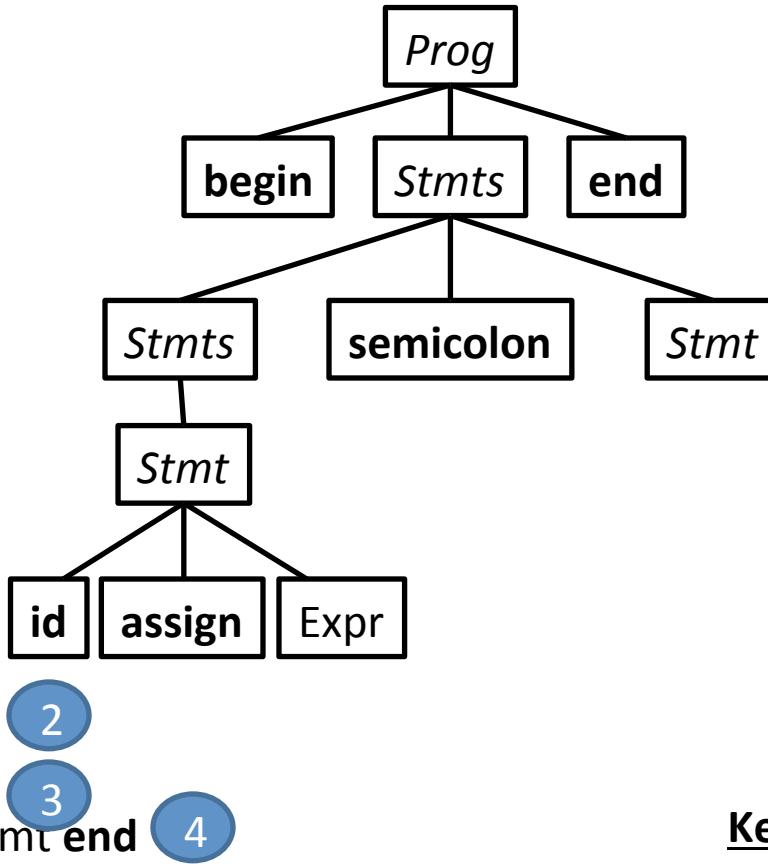
## Key



## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$

## Parse Tree



## Derivation Sequence

$\text{Prog} \Rightarrow^+ \text{begin } \text{Stmts } \text{end}$  ①  
 $\Rightarrow^+ \text{begin } \text{Stmts } \text{semicolon } \text{Stmt } \text{end}$  ②  
 $\Rightarrow^+ \text{begin } \text{Stmt } \text{semicolon } \text{Stmt } \text{end}$  ③  
 $\Rightarrow^+ \text{begin } \text{id assign Expr semicolon Stmt end}$  ④

## Key

terminal

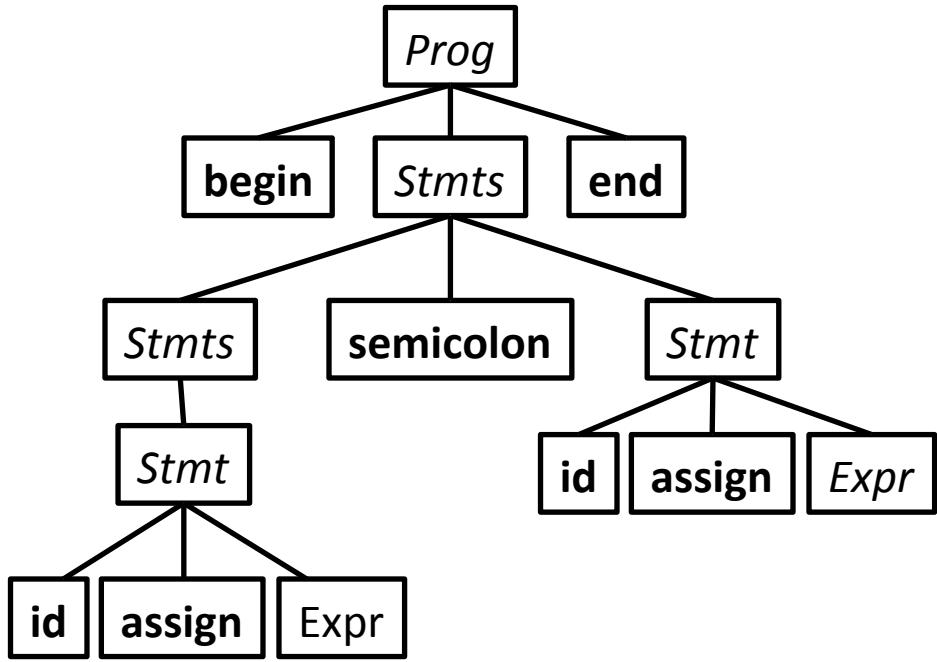
Nonterminal

Rule used

## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$

## Parse Tree



## Derivation Sequence

$\text{Prog} \Rightarrow^+ \text{begin } \text{Stmts } \text{end}$  ①  
 $\Rightarrow^+ \text{begin } \text{Stmts } \text{semicolon } \text{Stmt } \text{end}$  ②  
 $\Rightarrow^+ \text{begin } \text{Stmt } \text{semicolon } \text{Stmt } \text{end}$  ③  
 $\Rightarrow^+ \text{begin } \text{id assign Expr } \text{semicolon } \text{Stmt } \text{end}$  ④  
 $\Rightarrow^+ \text{begin } \text{id assign Expr } \text{semicolon } \text{id assign Expr } \text{end}$  ⑤

## Key

terminal

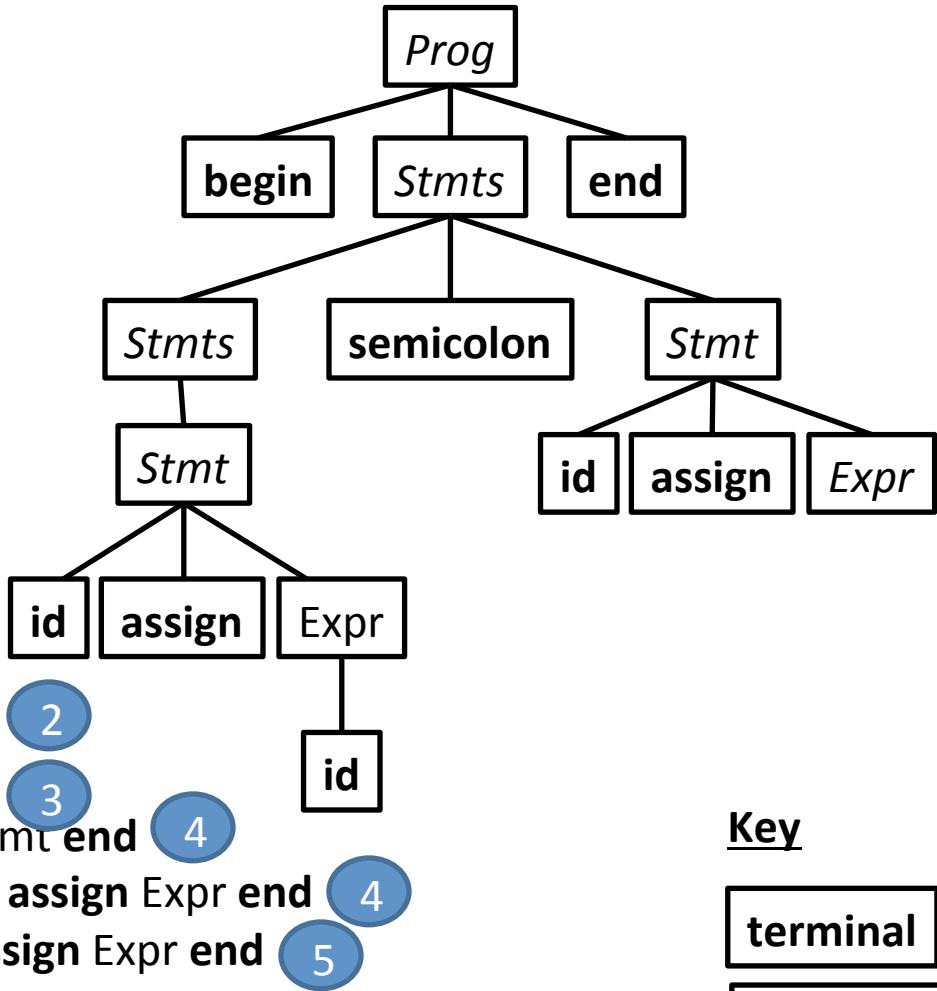
Nonterminal

Rule used

## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$

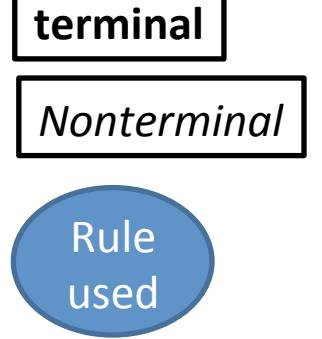
## Parse Tree



## Derivation Sequence

$\text{Prog} \Rightarrow^+ \text{begin } \text{Stmts } \text{end}$  ①  
 $\Rightarrow^+ \text{begin } \text{Stmts } \text{semicolon } \text{Stmt } \text{end}$  ②  
 $\Rightarrow^+ \text{begin } \text{Stmt } \text{semicolon } \text{Stmt } \text{end}$  ③  
 $\Rightarrow^+ \text{begin } \text{id assign Expr } \text{semicolon } \text{Stmt } \text{end}$  ④  
 $\Rightarrow^+ \text{begin } \text{id assign Expr } \text{semicolon } \text{id assign Expr } \text{end}$  ⑤  
 $\Rightarrow^+ \text{begin } \text{id assign id } \text{semicolon } \text{id assign Expr } \text{end}$

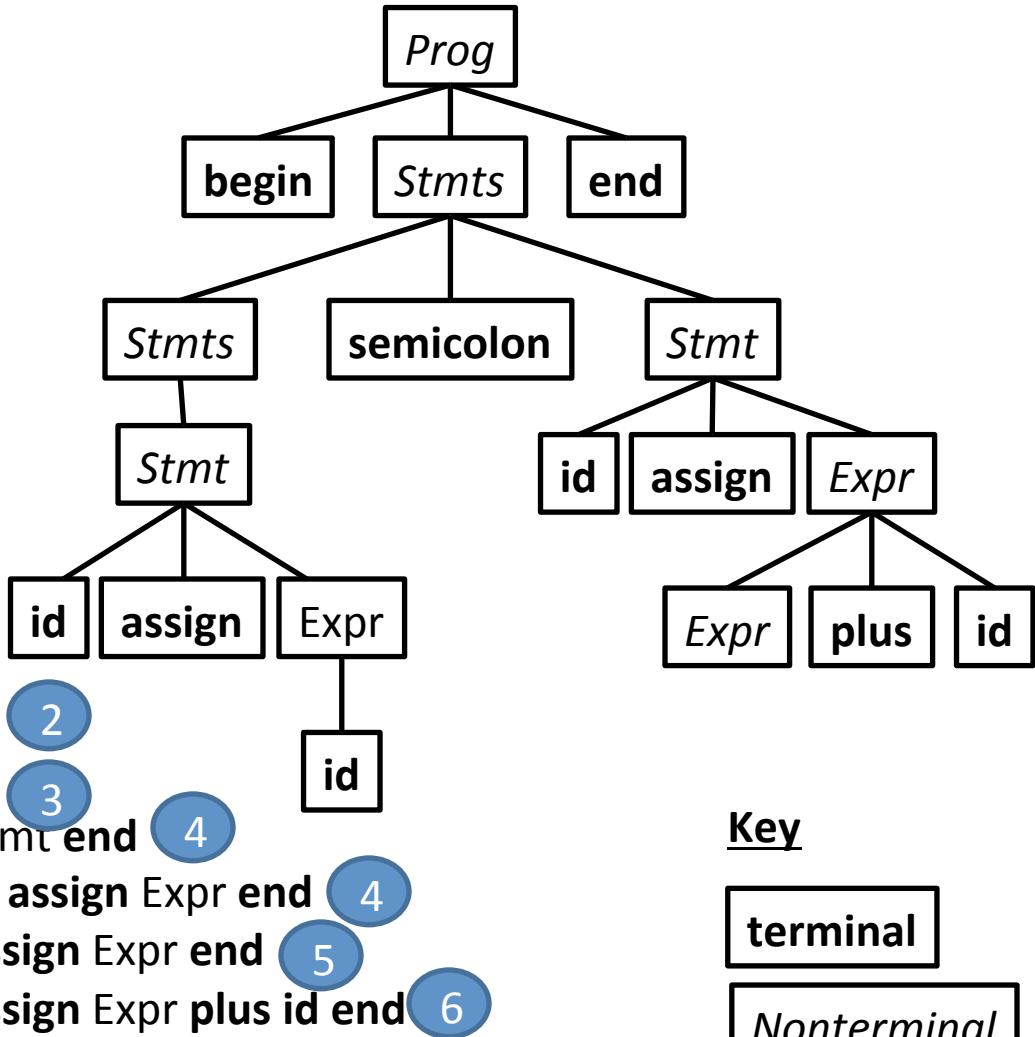
## Key



## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$

## Parse Tree



Key

terminal

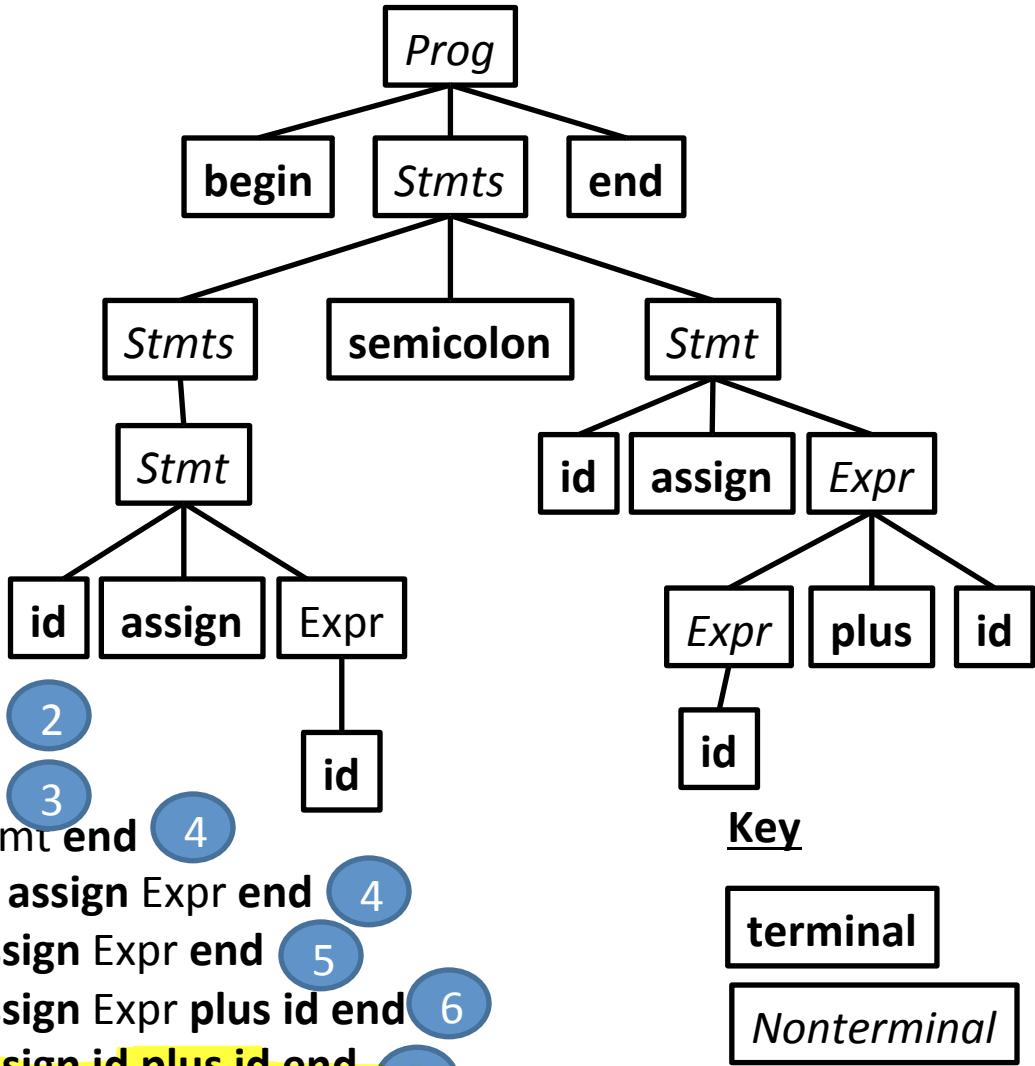
Nonterminal

Rule used

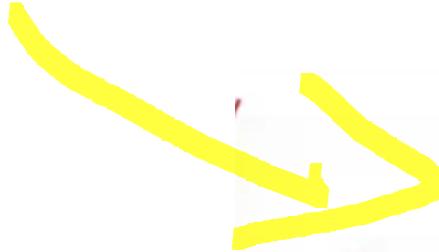
## Productions

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts } \text{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts } \text{semicolon } \text{Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$

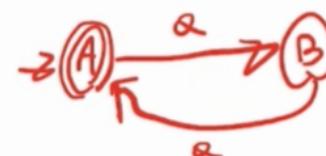
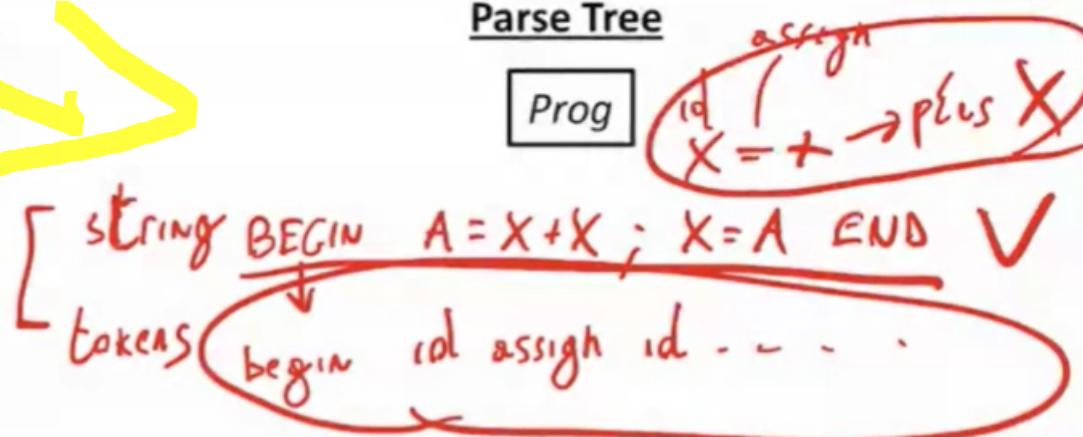
## Parse Tree



with 0 or more steps, prog can be expressed as these series of tokens



### Parse Tree



A five minute introduction

## MAKEFILE

$$\begin{aligned} A &\rightarrow \alpha B \mid \epsilon \\ B &\rightarrow \alpha A \end{aligned}$$

dfa can be easily converted to cfg

# Makefiles: Motivation

- Typing the series of commands to generate our code can be tedious
  - Multiple steps that depend on each other
  - Somewhat complicated commands
  - May not need to rebuild everything
- Makefiles solve these issues
  - Record a series of commands in a script-like DSL
  - Specify dependency rules and Make generates the results

# Makefiles: Basic Structure

<target>: <dependency list>

(tab) <command to satisfy target>

# Makefiles: Basic Structure

<target>: <dependency list>

(tab) <command to satisfy target>

## Example

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

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

# Makefiles: Basic Structure

<target>: <dependency list>

(tab) <command to satisfy target>

## Example

Example.class depends on example.java and IO.class

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

IO.class: IO.java  
javac IO.java

# Makefiles: Basic Structure

<target>: <dependency list>

(tab) <command to satisfy target>

## Example

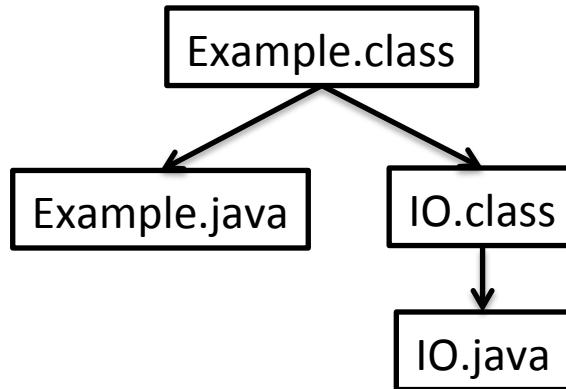
Example.class depends on example.java and IO.class

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

Example.class is generated by  
javac Example.java

IO.class: IO.java  
javac IO.java

# Makefiles: Dependencies



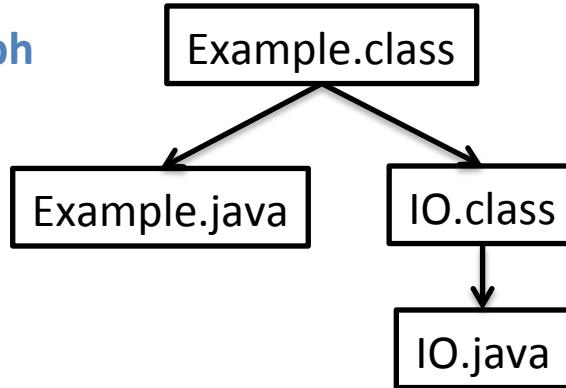
## Example

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

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

# Makefiles: Dependencies

Internal Dependency graph



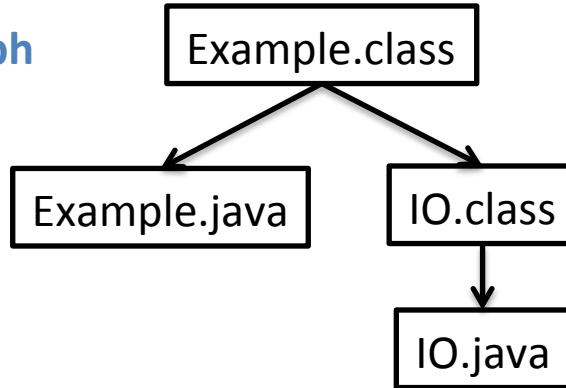
## Example

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

IO.class: IO.java  
javac IO.java

# Makefiles: Dependencies

Internal Dependency graph



A file is rebuilt if one of its dependencies changes

## Example

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

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

# Makefiles: Variables

You can thread common configuration values through your makefile

# Makefiles: Variables

You can thread common configuration values through your makefile

## Example

JC = /s/std/bin/javac

JFLAGS = -g

# Makefiles: Variables

You can thread common configuration values through your makefile

## Example

JC = /s/std/bin/javac

JFLAGS = -g    **Build for debug**

# Makefiles: Variables

You can thread common configuration values through your makefile

## Example

JC = /s/std/bin/javac

JFLAGS = -g    **Build for debug**

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

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

# Makefiles: Phony Targets

- You can run commands through make.
  - Write a target with no dependencies (called phony)
  - Will cause it to execute the command every time



# Makefiles: Phony Targets

- You can run commands through make.
  - Write a target with no dependencies (called phony)
  - Will cause it to execute the command every time

## Example

clean:

```
rm -f *.class
```

clean:

```
rm -f *.class ✓
```

```
rm -f *.java X
```



# Makefiles: Phony Targets

- You can run commands through make.
  - Write a target with no dependencies (called phony)
  - Will cause it to execute the command every time

## Example

clean:

```
    rm -f *.class
```

test:

```
    java -cp . Test.class
```



# Recap

- We've defined context-free grammars
  - More powerful than regular expressions
- Learned a bit about makefile
- Next time we'll look at grammars in more detail

# Defining syntax using CFGs

# Roadmap

Last time

- Defined context-free grammar

This time

- CFGs for syntax design
  - Language membership
  - List grammars
  - Resolving ambiguity

# CFG Review

terminal

- $G = (N, \Sigma, P, S)$
- $\xrightarrow{+}$  means *derives*  
*derives in 1 or more steps*
- CFG generates a string by applying productions until no non-terminals remain

Example: Nested parens

$$N = \{ Q \}$$

$$\Sigma = \{ (, ) \}$$

$$P = Q \xrightarrow{} (Q) \\ | \quad \varepsilon$$

$$S = Q$$

# Formal CFG Language Definition

Let  $G = (N, \Sigma, P, S)$  be a CFG. Then

$$w \in \Sigma^*$$

$L(G) = w \square S \Rightarrow^* + - w$  where

$S$  is the start nonterminal of  $G$

$w$  is a sequence of terminals or  $\epsilon$

# CFGs as Language Definition

CFG productions define the *syntax* of a language

1.  $\text{Prog} \rightarrow \mathbf{begin} \text{Stmts} \mathbf{end}$
2.  $\text{Stmts} \rightarrow \text{Stmts} \mathbf{semicolon} \text{Stmt}$
3.           |  $\text{Stmt}$
4.  $\text{Stmt} \rightarrow \mathbf{id} \mathbf{assign} \text{Expr}$
5.  $\text{Expr} \rightarrow \mathbf{id}$
6.           |  $\text{Expr} \mathbf{plus} \mathbf{id}$

We call this notation “*BNF*” or “*extended BNF*”

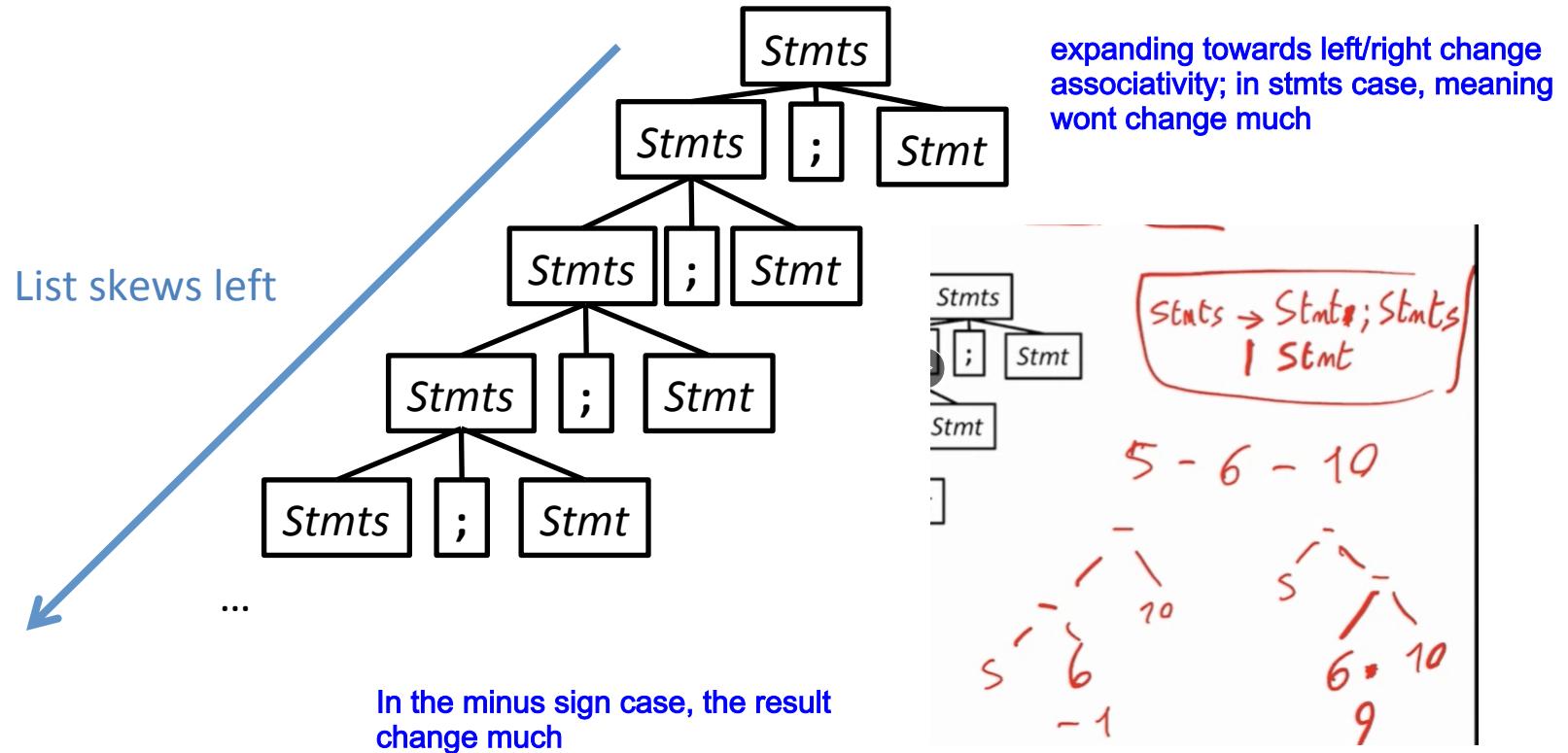
HTTP grammar using BNF:

- <http://www.w3.org/Protocols/rfc2616/rfc2616-sec2.html>

# List Grammars

- Useful to repeat a structure arbitrarily often

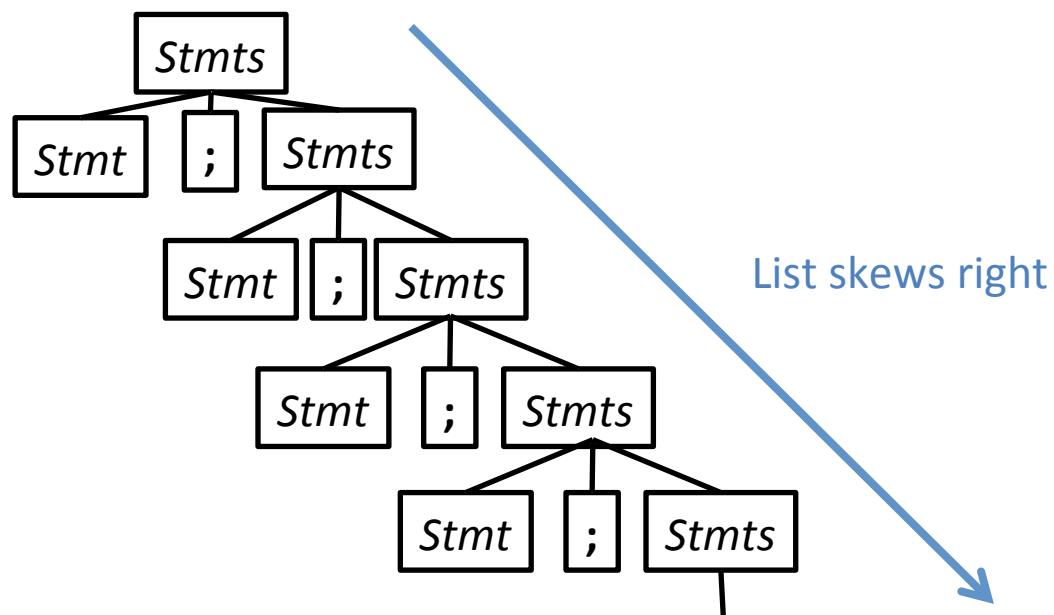
$Stmts \rightarrow Stmt \mid Semicolon \; Stmt \mid Semicolon \; Stmt \mid \dots$



# List Grammars

- Useful to repeat a structure arbitrarily often

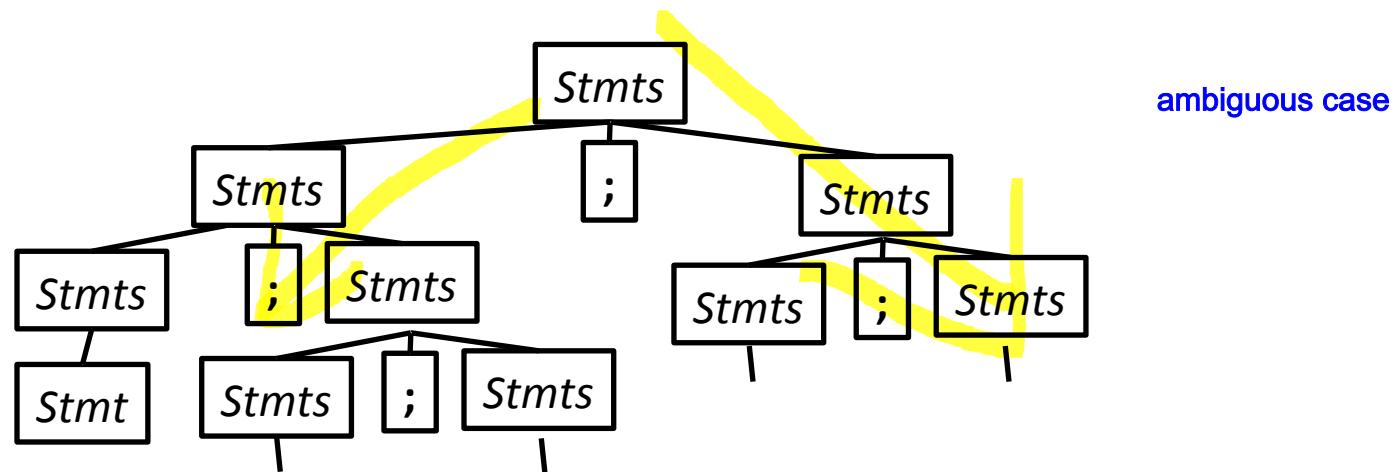
$Stmts \rightarrow Stmt \text{ semicolon } \underline{Stmts} \mid Stmt$



# List Grammars

- What if we allowed both “skews”?

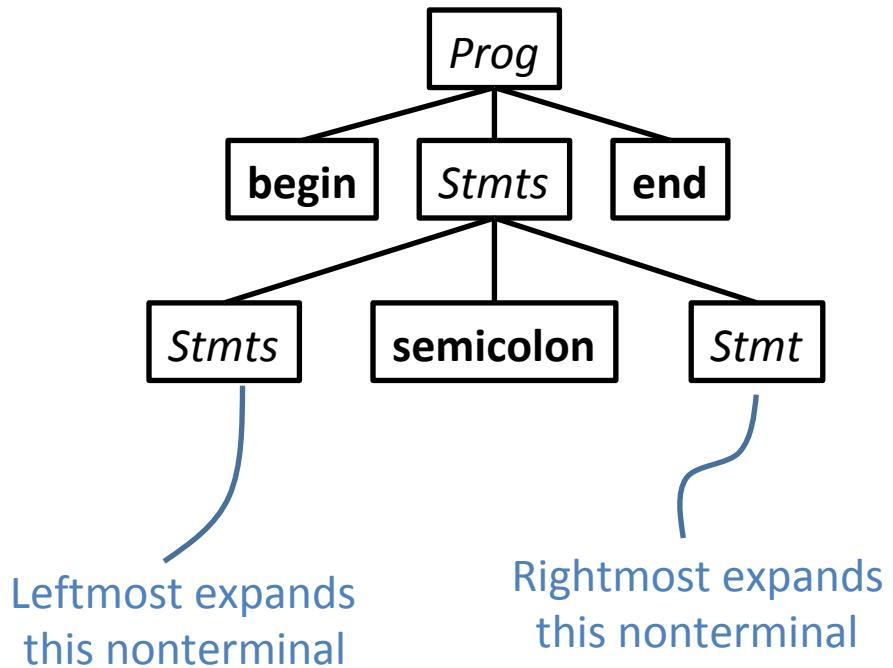
$Stmts \rightarrow Stmt \_ semicolon \_ Stmt \mid Stmt$



# Derivation Order

- Leftmost Derivation: always expand the leftmost nonterminal
- Rightmost Derivation: always expand the rightmost nonterminal

1.  $\text{Prog} \rightarrow \text{begin } \text{Stmts end}$
2.  $\text{Stmts} \rightarrow \text{Stmts semicolon Stmt}$
3.  $\quad | \quad \text{Stmt}$
4.  $\text{Stmt} \rightarrow \text{id assign Expr}$
5.  $\text{Expr} \rightarrow \text{id}$
6.  $\quad | \quad \text{Expr plus id}$



# Ambiguity

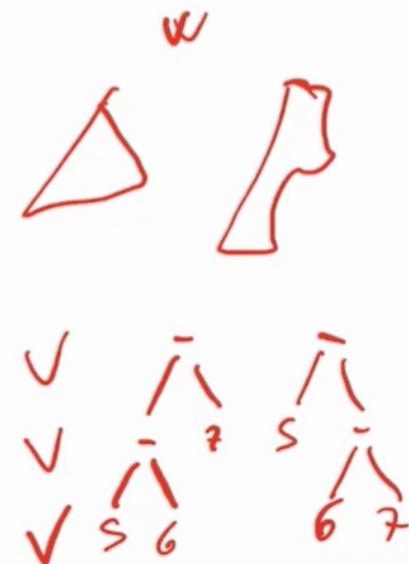
the same tree will produce different trees

Even with a fixed derivation order, it is possible to derive the same string in multiple ways

For Grammar G and string w

– G is ambiguous if there exists one of the following cases:

- >1 leftmost derivation of w
- >1 rightmost derivation of w
- > 1 parse tree for w

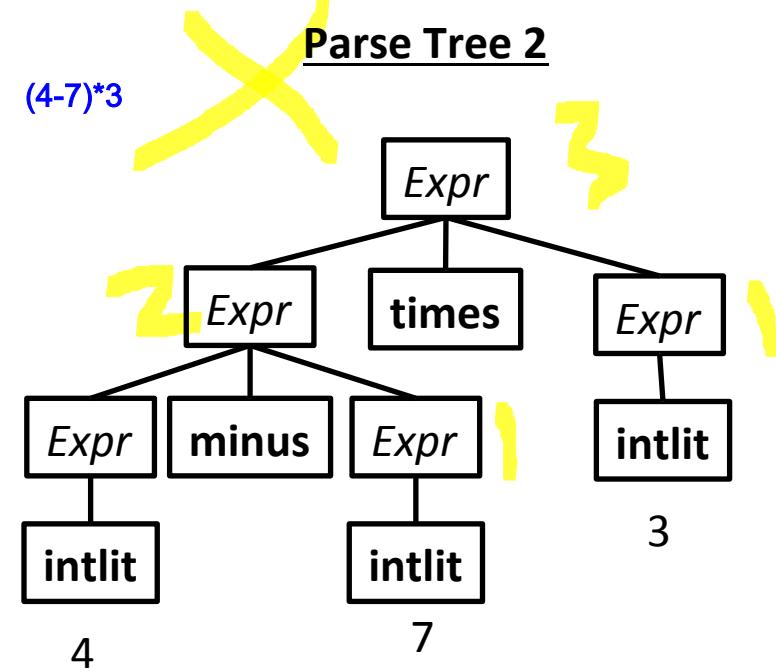
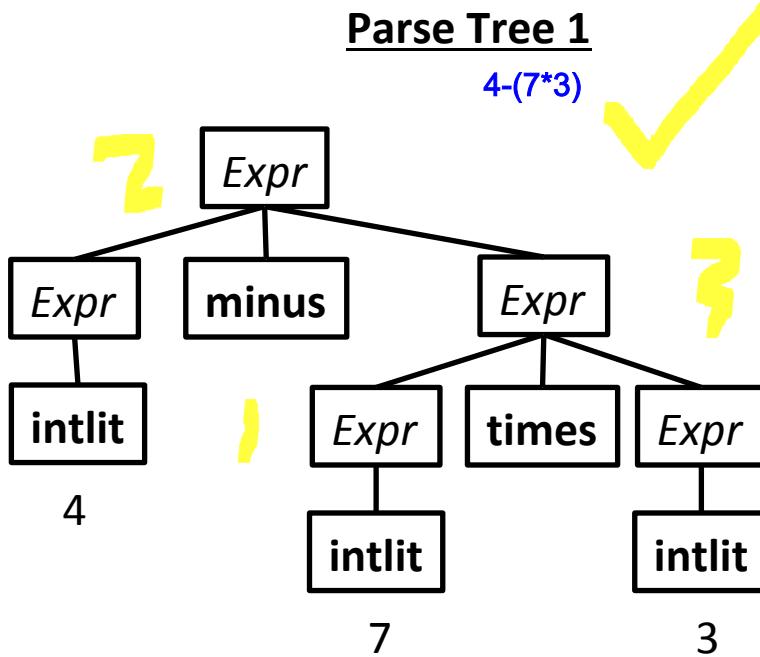


# Example: Ambiguous Grammars

*Expr* → *intlit*  
| *Expr minus Expr*  
| *Expr times Expr*  
| *lparen Expr rparen*

Derive the string  $4 - 7 * 3$   
(assume tokenization)

2 problems: associativity  
operator precedence

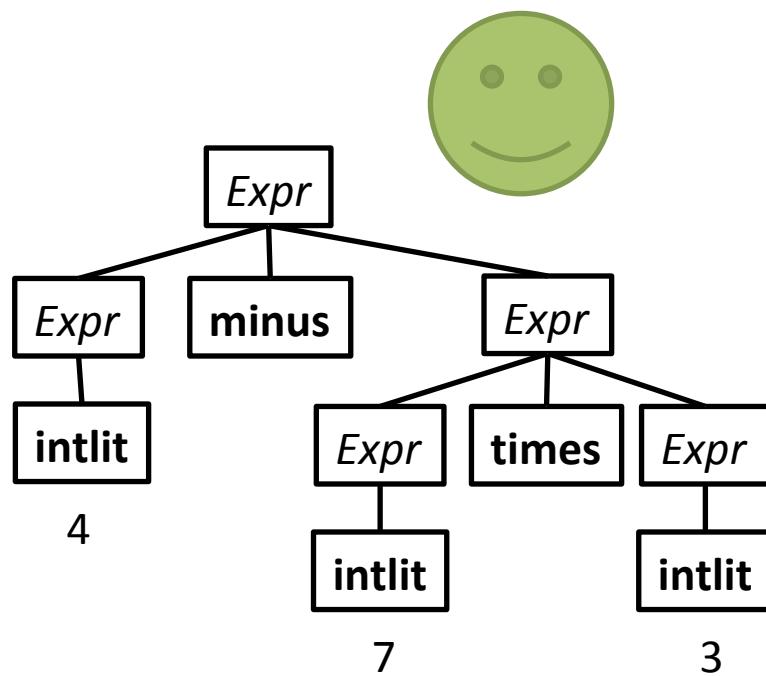


# Why is Ambiguity Bad?

2 ways to solve this: precedence and associativity

Eventually, we'll be using CFGs as the basis for our parser

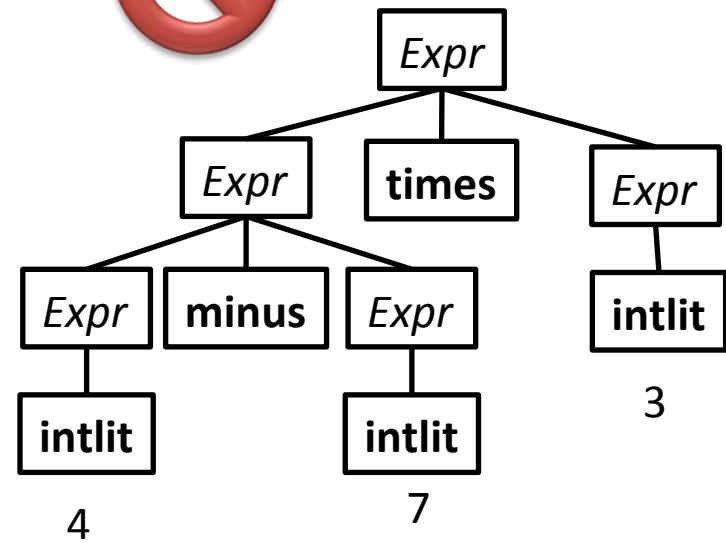
- Parsing is much easier when there is no ambiguity in the grammar
- The parse tree may mismatch user understanding!



$4 - 7 * 3$



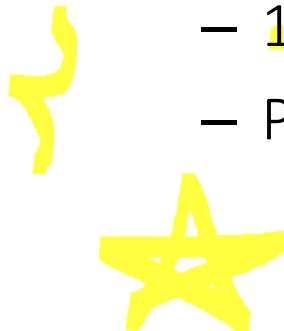
Operator  
precedence



# Resolving Grammar Ambiguity: Precedence

$Expr \rightarrow \text{intlit}$

|  $Expr \text{ minus } Expr$   
|  $Expr \text{ times } Expr$   
| **I**paren  $Expr$  rparen



Intuitive problem

- “Context-freeness”
- Nonterminals are the same for both operators

To fix precedence

- 1 nonterminal per precedence level
- Parse lowest level first

# Resolving Grammar Ambiguity: Precedence

$Expr \rightarrow intlit$

|  $Expr \text{ minus } Expr$

|  $Expr \text{ times } Expr$

|  $( Expr )$



$Expr \rightarrow Expr \text{ minus } Expr$

|  $Term$

$Term \rightarrow Term \text{ times } Term$

|  $Factor$

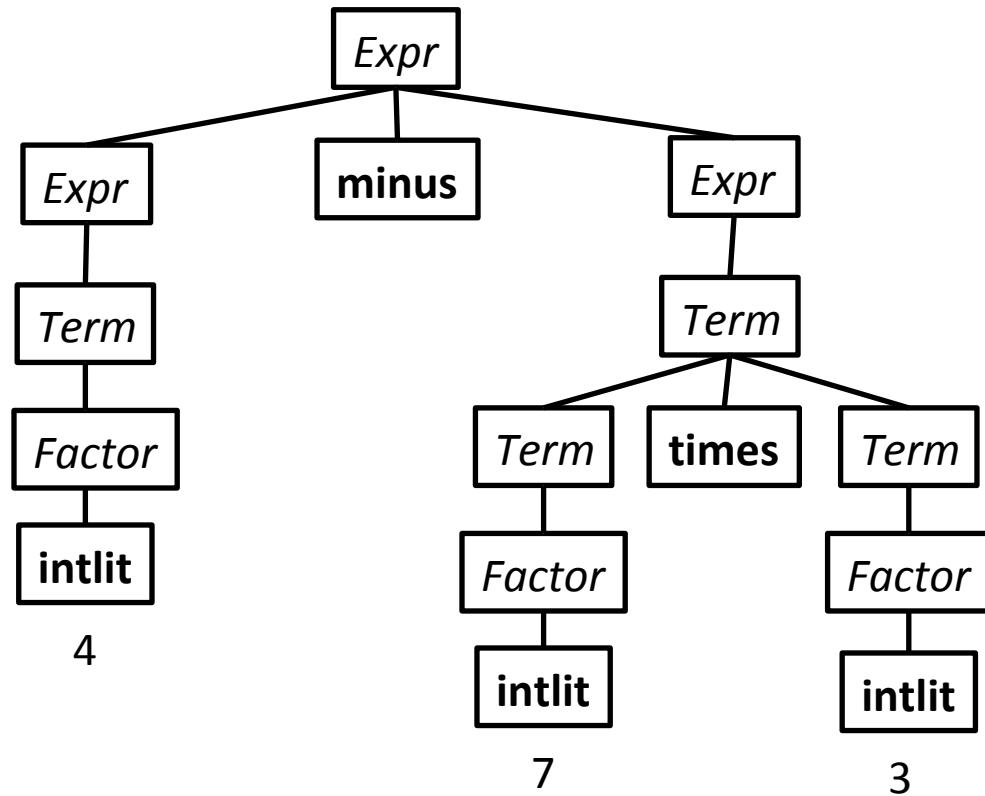
$Factor \rightarrow intlit$

|  $( Expr )$

these are  
just  
names

lowest precedence level first  
1 nonterm per precedence level

Derive the string  $4 - 7 * 3$



# Resolving Grammar Ambiguity: Precedence

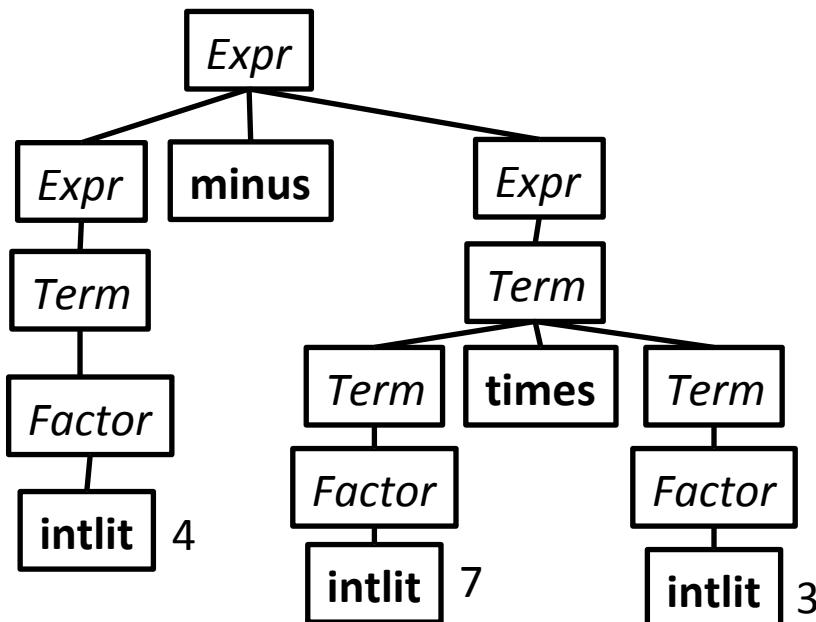
## Fixed Grammar

$Expr \rightarrow Expr \text{ minus } Expr$   
|  $Term$

$Term \rightarrow Term \text{ times } Term$   
|  $Factor$

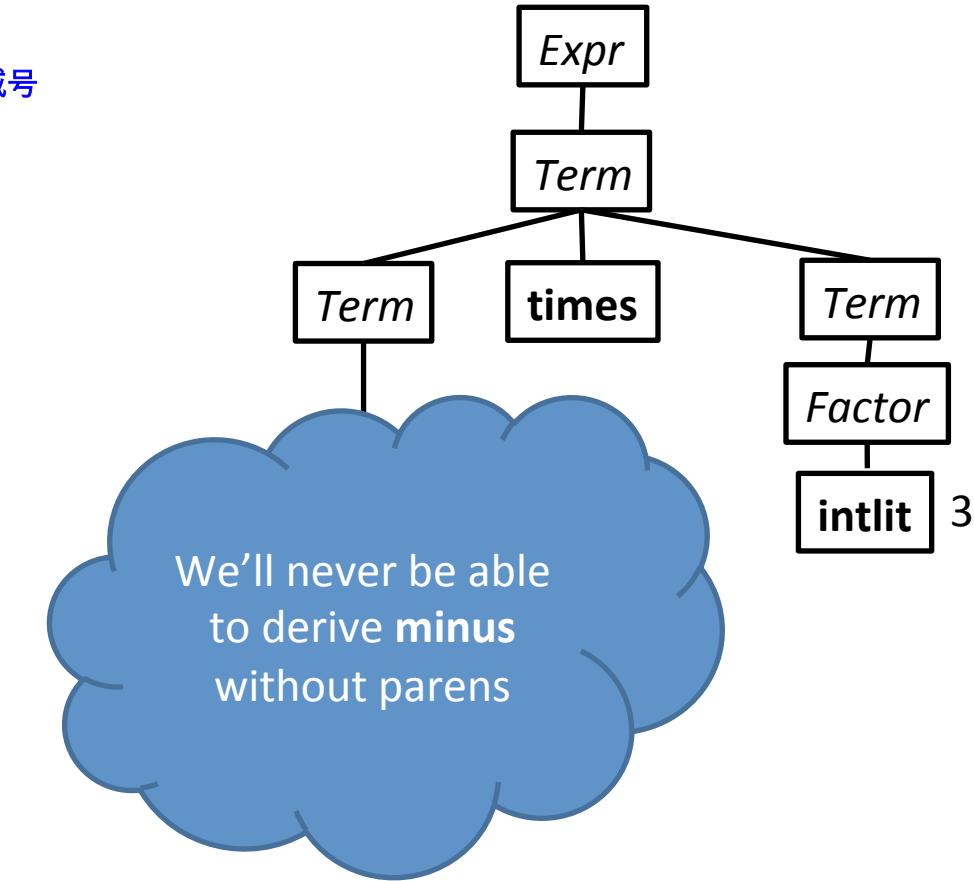
$Factor \rightarrow intlit$   
|  $\text{lparen } Expr \text{ rparen}$

先处理减号，因为减号  
优先级低于乘号



Derive the string  $4 - 7 * 3$

Let's try to re-build the wrong parse tree



# Did we fix all ambiguity?

## Fixed Grammar

*Expr* → *Expr minus Expr*

| *Term*

*Term* → *Term times Term*

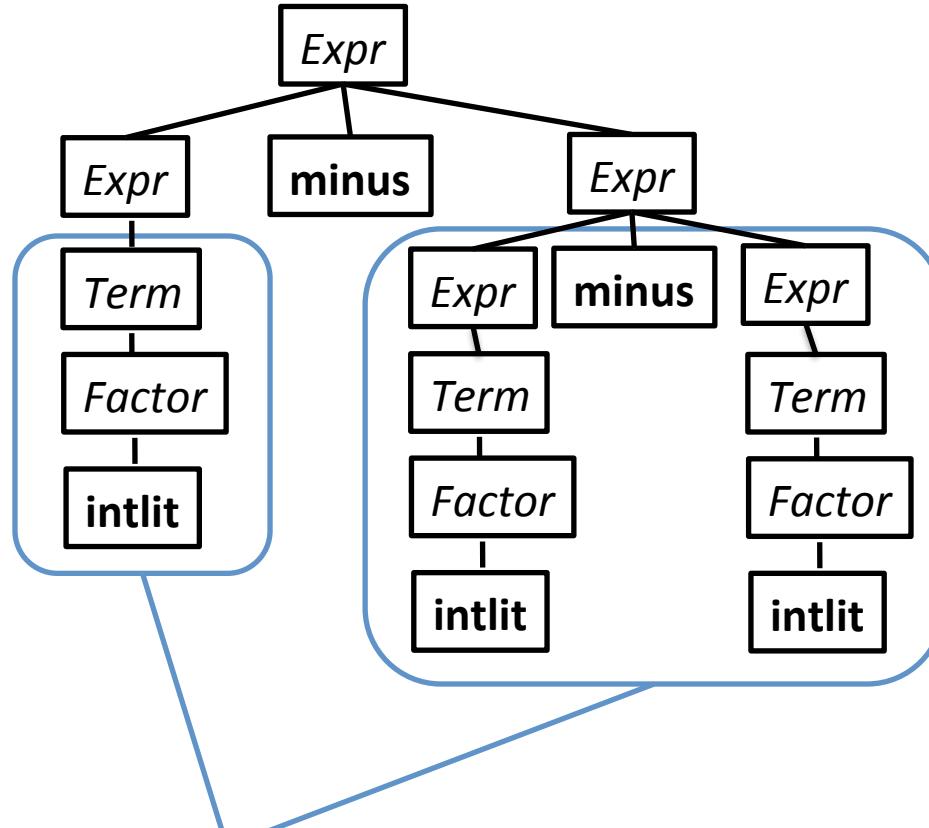
| *Factor*

*Factor* → **intlit**

| **Iparen Expr rparen**

NO!

Derive the string 4 - 7 - 3



These subtrees could have  
been swapped!

# Where we are so far

## Precedence

- We want correct behavior on  $4 - 7 * 9$
- A new nonterminal for each precedence level

## Associativity

- We want correct behavior on  $4 - 7 - 9$
- Minus should be *left associative*:  $a - b - c = (a - b) - c$
- Problem: the *recursion* in a rule like

$Expr \rightarrow Expr \text{minus } Expr$

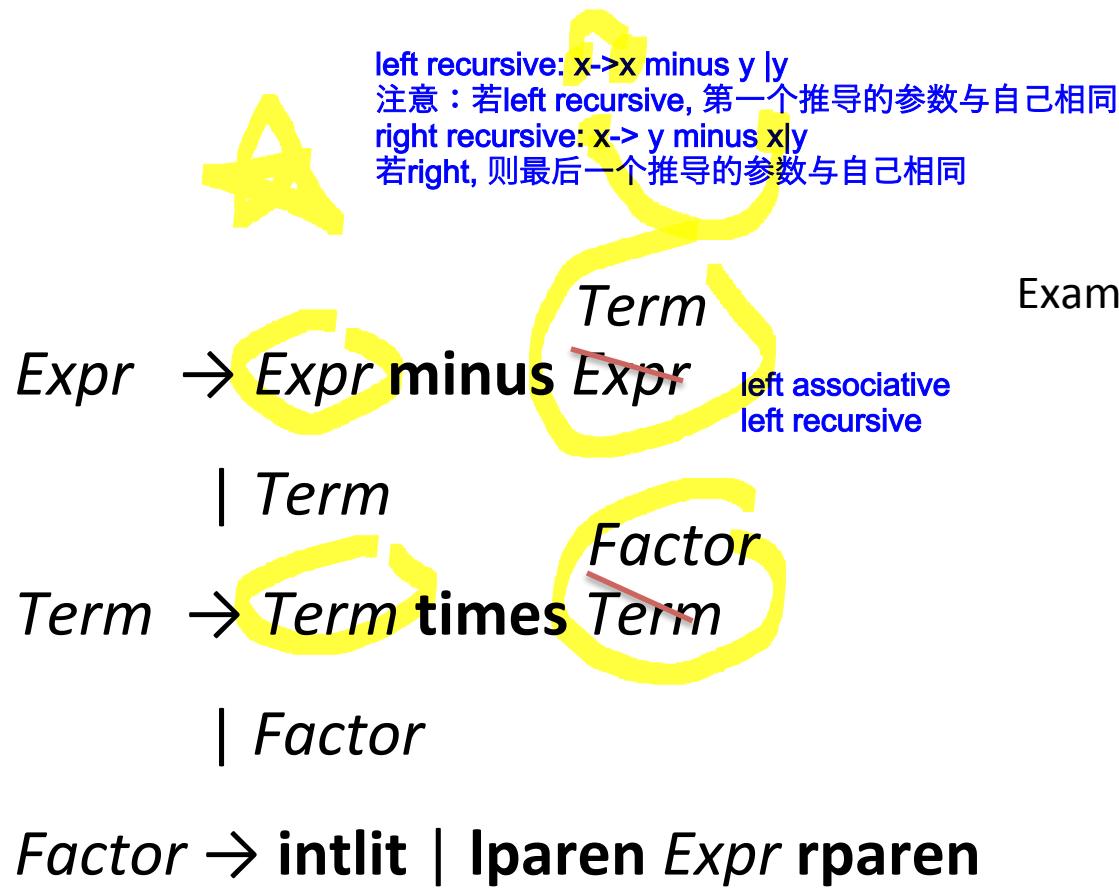
# Definition: Recursion in Grammars

- A grammar is *recursive* in (nonterminal)  $X$  if  $X \Rightarrow^* \alpha X \gamma$  for non-empty strings of symbols  $\alpha$  and  $\gamma$
- A grammar is *left-recursive* in  $X$  if  $X \Rightarrow^* \alpha X \gamma$  for non-empty string of symbols  $\gamma$
- A grammar is *right-recursive* in  $X$  if  $X \Rightarrow^* \alpha X$  for non-empty string of symbols  $\alpha$

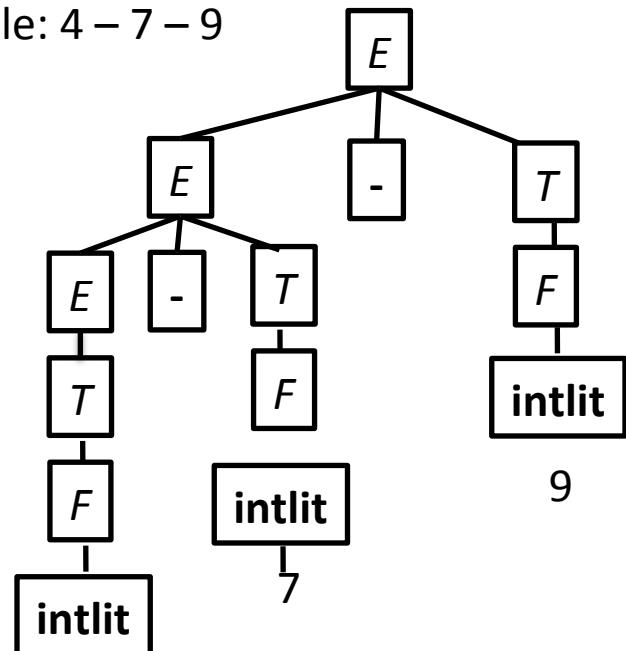
# Resolving Grammar Ambiguity: Associativity

Recognize left-assoc operators with left-associative productions

Recognize right-assoc operators with right-associative productions



Example: 4 – 7 – 9



# Resolving Grammar Ambiguity: Associativity

*Expr* → *Expr minus Term*

| *Term*

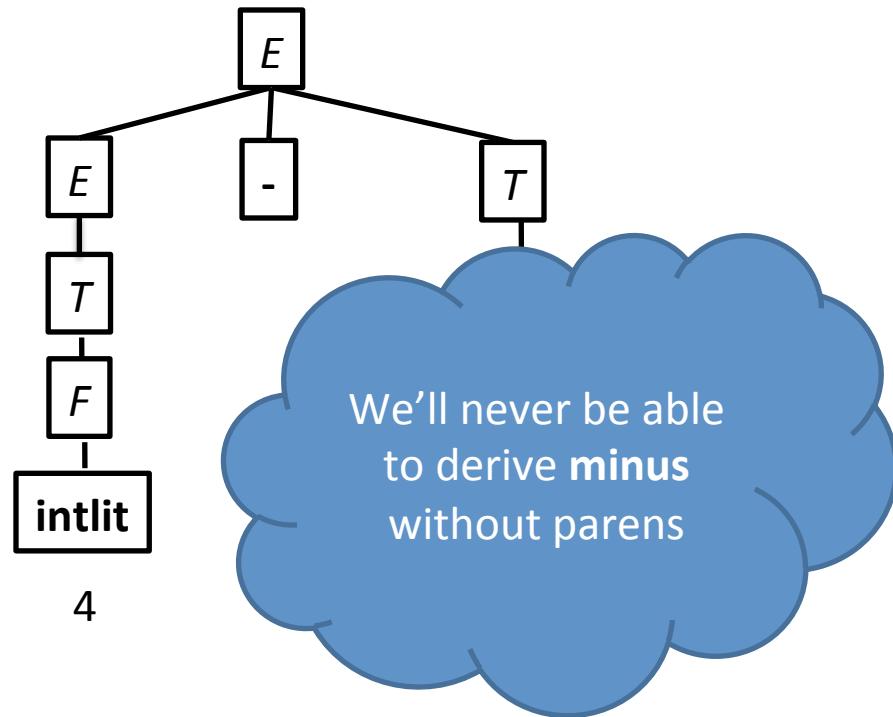
*Term* → *Term times Factor*

| *Factor*

*Factor* → **intlit** | **Iparen Expr rparen**

Example: 4 – 7 – 9

Let's try to re-build the wrong parse tree again

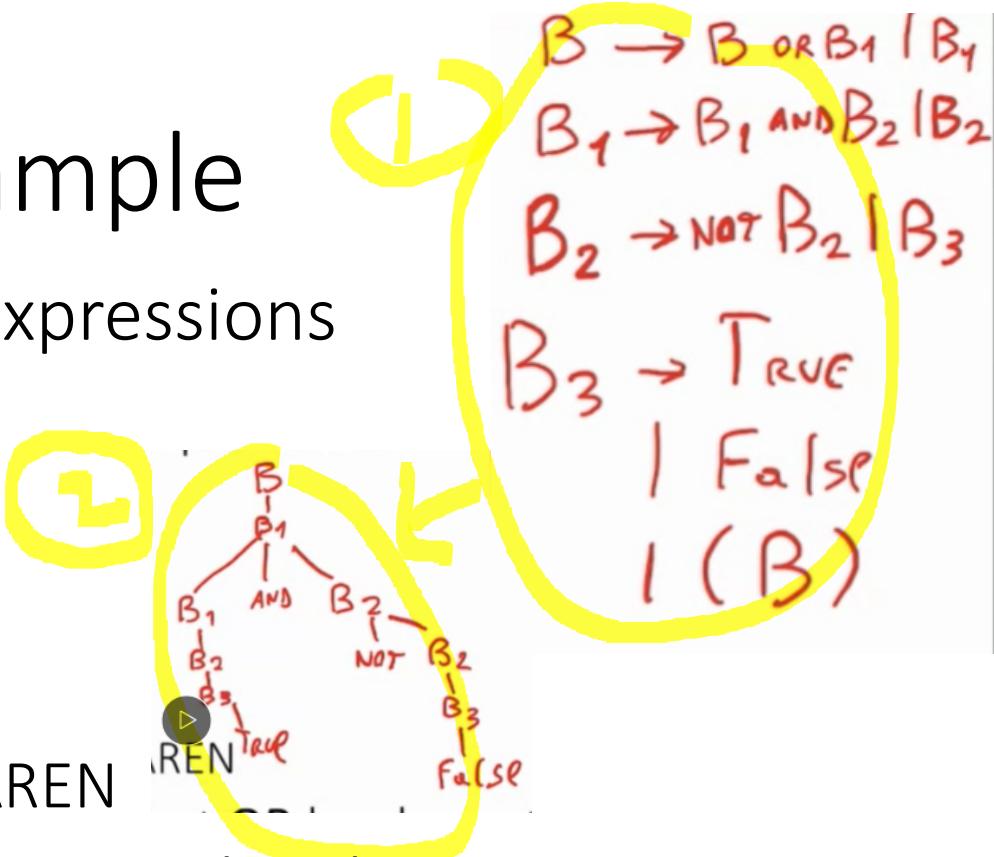


# Example

- Language of Boolean expressions

$bexp \rightarrow \text{TRUE}$

| FALSE  
|  $bexp \text{ OR } bexp$   
|  $bexp \text{ AND } bexp$   
| NOT  $bexp$   
| LPAREN  $bexp$  RPAREN



- Add nonterminals so that OR has lowest precedence, then AND, then NOT. Then change the grammar to reflect the fact that both AND and OR are left associative.
- Draw a parse tree for the expression:
  - true AND NOT true

# Another ambiguous example

Stmt →

if Cond then Stmt |

if Cond then Stmt else Stmt | ...

Consider this word in this grammar:

if a then if b then s else s2

How would you derive it?

# Summary

To understand how a parser works, we start by understanding **context-free grammars**, which are used to define the language recognized by the parser.  
terminal symbol

- (non)terminal symbol
- grammar rule (or production)
- derivation (leftmost derivation, rightmost derivation)
- parse (or derivation) tree
- the language defined by a grammar
- ambiguous grammar

# Announcements

Working in pairs is only allowed for programming assignments and not for homework problems

H3 has been posted

1

# Syntax Directed Translation

# CFGs so Far

## CFGs for Language *Definition*

- The CFGs we've discussed can generate/define languages of valid strings
- So far, we **start** by building a parse tree and **end** with some valid string

## CFGs for Language *Recognition*

- Start with a string and end with a parse tree for it

# CFGs for Parsing

Language Recognition isn't enough for a parser

- We also want to *translate* the sequence

Parsing is a special case of *Syntax-Directed Translation*

- Translate a sequence of tokens into a sequence of actions

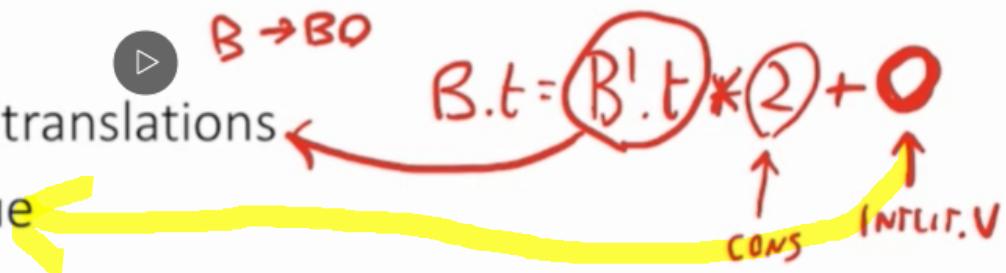
# Syntax Directed Translation

Augment CFG rules with translation rules (at least 1 per production)

- Define translation of LHS nonterminal as function of
  - Constants
  - RHS nonterminal translations
  - RHS terminal value

Assign rules bottom-up

- Constants
- RHS nonterminal translations
- RHS terminal value



# SDT Example

CFG

Rules actions

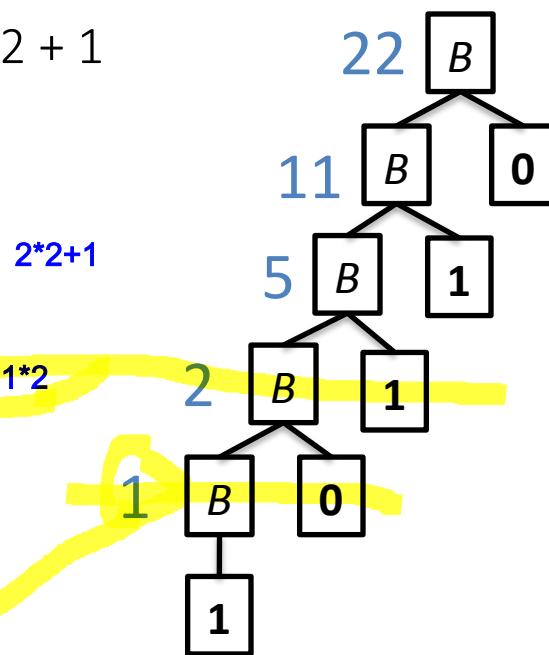
B -> 0       $B.\text{trans} = 0$

| 1       $B.\text{trans} = 1$

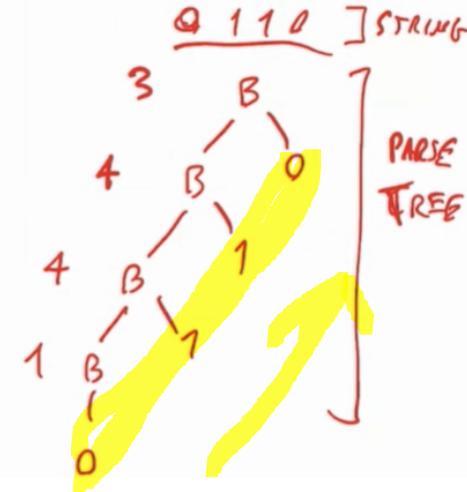
| B 0       $B.\text{trans} = B_2.\text{trans} * 2$

| B 1       $B.\text{trans} = B_2.\text{trans} * 2 + 1$

Translation is  
the value of  
the input



Input string  
10110



# SDT Example 2: Declarations

		an empty string
<u>CFG</u>		
<i>DList</i>	$\rightarrow \epsilon$	<i>DList.trans = " "</i>
	<i>DList Decl</i>	<i>DList.trans = Decl.trans + " " + DList<sub>2</sub>.trans</i>

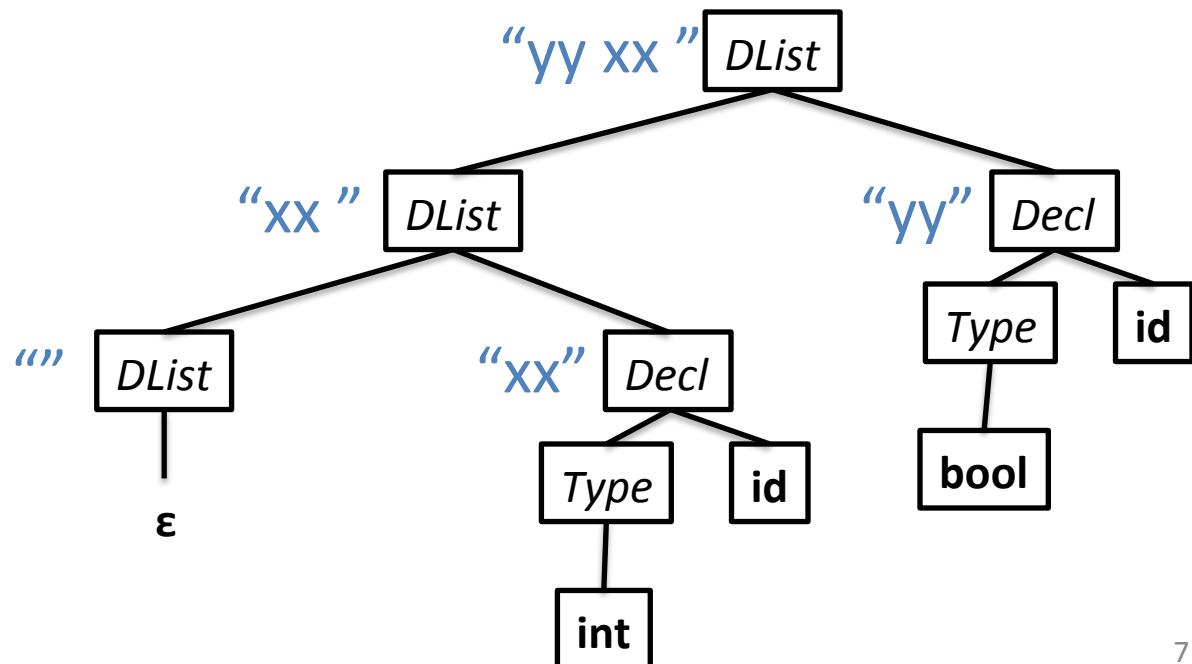
  

<i>Decl</i>	$\rightarrow$ Type id	<i>Decl.trans = id.value</i>
<i>Type</i>	$\rightarrow$ int	

	bool	
--	------	--

Input string  
int xx;  
bool yy;



Translation is a String of ids

# Exercise Time

Only add declarations of type int to the output String.

Augment the previous grammar:

## CFG

$DList$	$\rightarrow \epsilon$
	$DList\ Decl$
$Decl$	$\rightarrow Type\ id\ ;$
$Type$	$\rightarrow int$
	$bool$

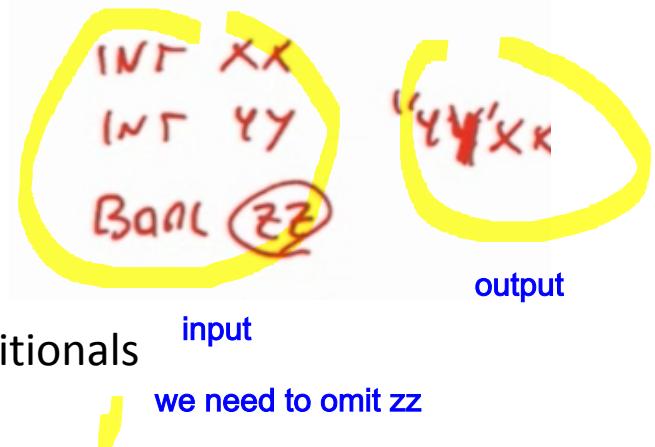
Different nonterms can have different types

## Rules

$DList$	$.trans = \epsilon$
	$= DList.trans + Decl.trans + DList_2.trans$

$Decl$   $.trans = id.value$

Rules can have conditionals



# SDT Example 2b: ints only

Translation is a String of **int** ids only

## CFG

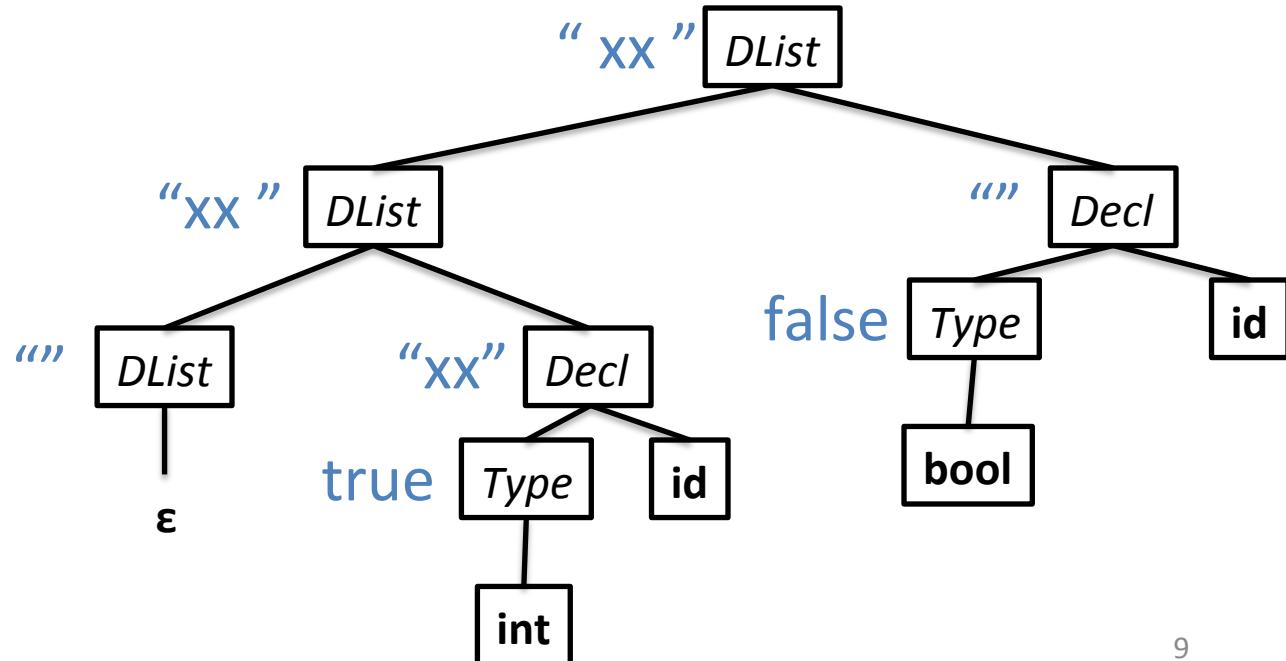
$DList$	$\rightarrow \epsilon$	$DList.trans = ""$
	$  Decl\ DList$	$DList.trans = Decl.trans + " " + DList_2.trans$
$Decl$	$\rightarrow Type\ id\ ;$	$if (Type.trans) \{Decl.trans = id.value\} else \{Decl.trans = "\"\}$
$Type$	$\rightarrow int$	$Type.trans = true$
	$  bool$	$Type.trans = false$

## Rules

Input string  
 int xx;  
 bool yy;

Different nonterms can have different types

Rules can have conditionals



# SDT for Parsing

In the previous examples, the SDT process assigned different types to the translation:

- Example 1: tokenized stream to an `integer value`
- Example 2: tokenized stream to a (java) `String`

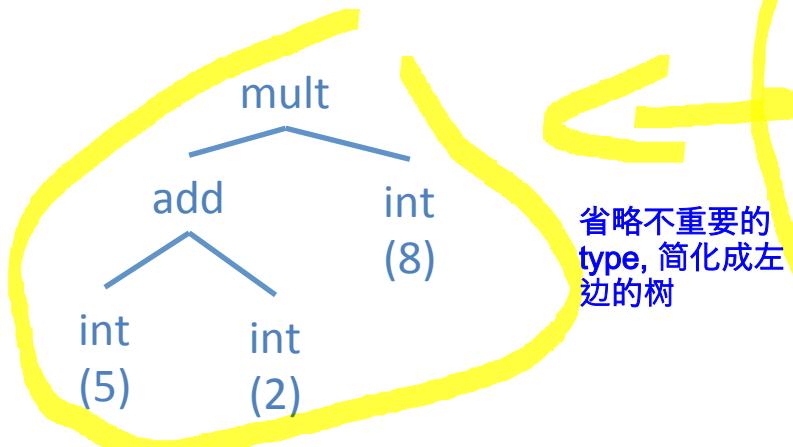
For parsing, we'll go from tokens to an Abstract-Syntax Tree (AST)

# Abstract Syntax Trees

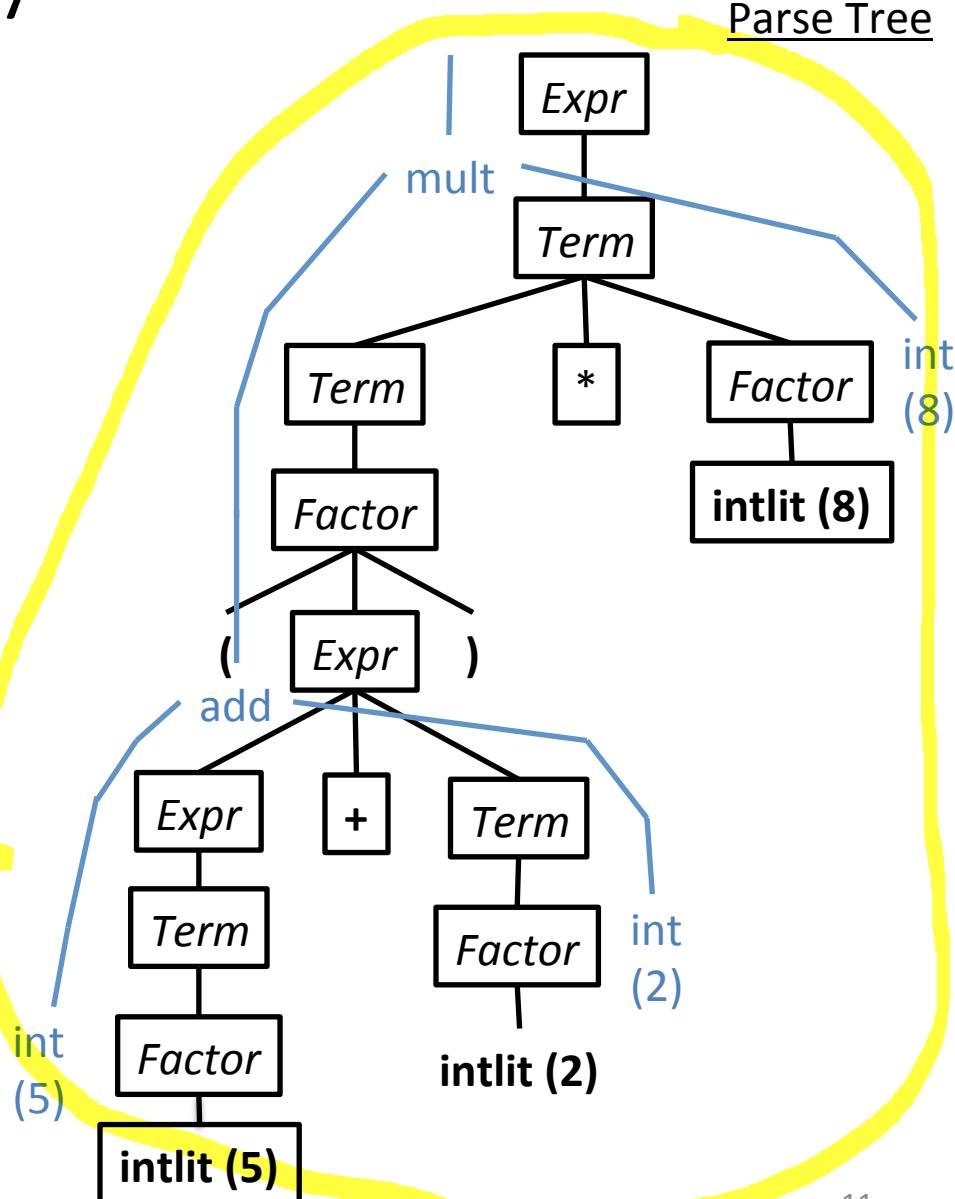
抽象语法树

- A condensed form of the parse tree
- Operators at internal nodes (not leaves)
- Chains of productions are collapsed
- Syntactic details omitted

Example:  $(5+2)*8$



省略不重要的  
type, 简化成左  
边的树



# Exercise #2

- Show the AST for:

$$(1 + 2) * (3 + 4) * 5 + 6$$

Expr  $\rightarrow$  Expr + Term  
| Term

Term  $\rightarrow$  Term \* Factor  
| Factor

Factor  $\rightarrow$  intlit  
| ( Expr )

Expr  $\rightarrow$  Expr + Term      *Expr1.trans = MkPlusNode(Expr2.trans, Term.trans)*

$\xrightarrow{\text{FACTOR} \rightarrow \text{INTLIT}}$  Factor.trans = MkINTLIT Node (INTLIT.v)

$\text{Factor.} t : E_{\text{expr.}} t$

# AST for Parsing

In previous slides we did our translation in two steps

- Structure the stream of tokens into a parse tree
- Use the parse tree to build an abstract syntax tree, throw away the parse tree

In practice, we will combine these into 1 step

**Question:** Why do we even need an AST?

- More of a “logical” view of the program
- Generally easier to work with

# AST Implementation

How do we actually represent an AST in code?

# ASTs in Code

Note that we've assumed a field-like structure in our SDT actions:

$$DList.trans = Decl.trans + " " + DList_2.trans$$

In our parser, we'll define classes for each type of nonterminal, and create a new nonterminal in each rule.

- In the above rule we might represent DList as

```
public class DList{  
    public String trans;  
}
```

- For ASTs: when we execute an SDT rule
  - we construct a new node object for the RHS
  - propagate its fields with the fields of the LHS nodes

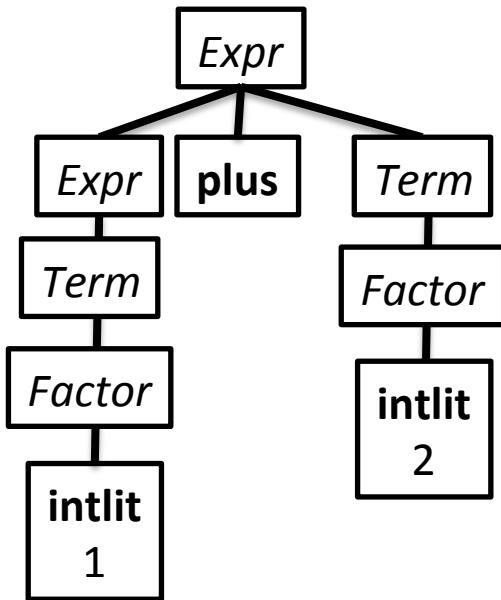
# Thinking about implementing ASTs

Consider the AST for a simple language of Expressions

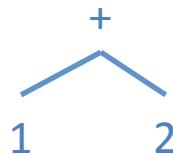
Input  
1 + 2

Tokenization  
intlit plus intlit

Parse Tree



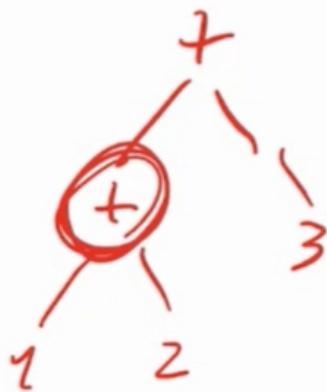
AST



Naïve AST Implementation

```
class PlusNode  
    IntNode left;  
    IntNode right;  
}
```

```
class IntNode{  
    int value;  
}
```

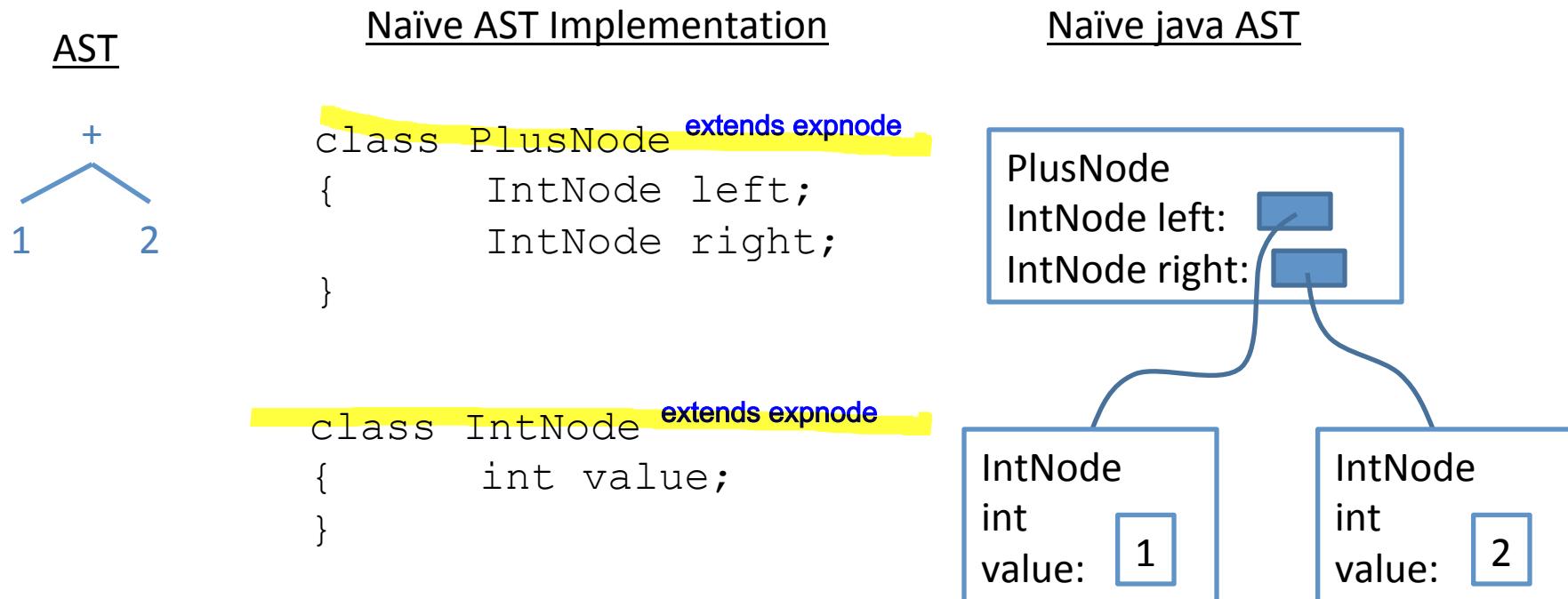


problem: 如果我们要以上的implementation, 我们需要一个+ node, 但我们只有left, right node, 所以说这个ast设计的不周全

# Thinking about implementing ASTs

Consider AST node classes

- We'd like the classes to have a common inheritance tree

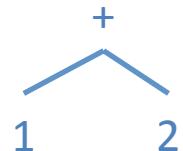


# Thinking about implementing ASTs

Consider AST node classes

- We'd like the classes to have a common inheritance tree

AST



Naïve AST Implementation

```
class PlusNode {  
    IntNode left;  
    IntNode right;  
}  
  
class IntNode {  
    int value;  
}
```

Make these extend  
ExpNode

Better java AST

```
PlusNode  
ExpNode left:  
ExpNode right:
```

```
IntNode  
int  
value: 1
```

```
IntNode  
int  
value: 2
```

# Implementing ASTs for Expressions

## CFG

Expr  $\rightarrow$  Expr + Term  
| Term

Term  $\rightarrow$  Term \* Factor  
| Factor

Factor  $\rightarrow$  intlit  
| ( Expr )

## Translation Rules

*Expr1.trans = new PlusNode(Expr2.trans, Term.trans)*

*Expr.trans = Term.trans*

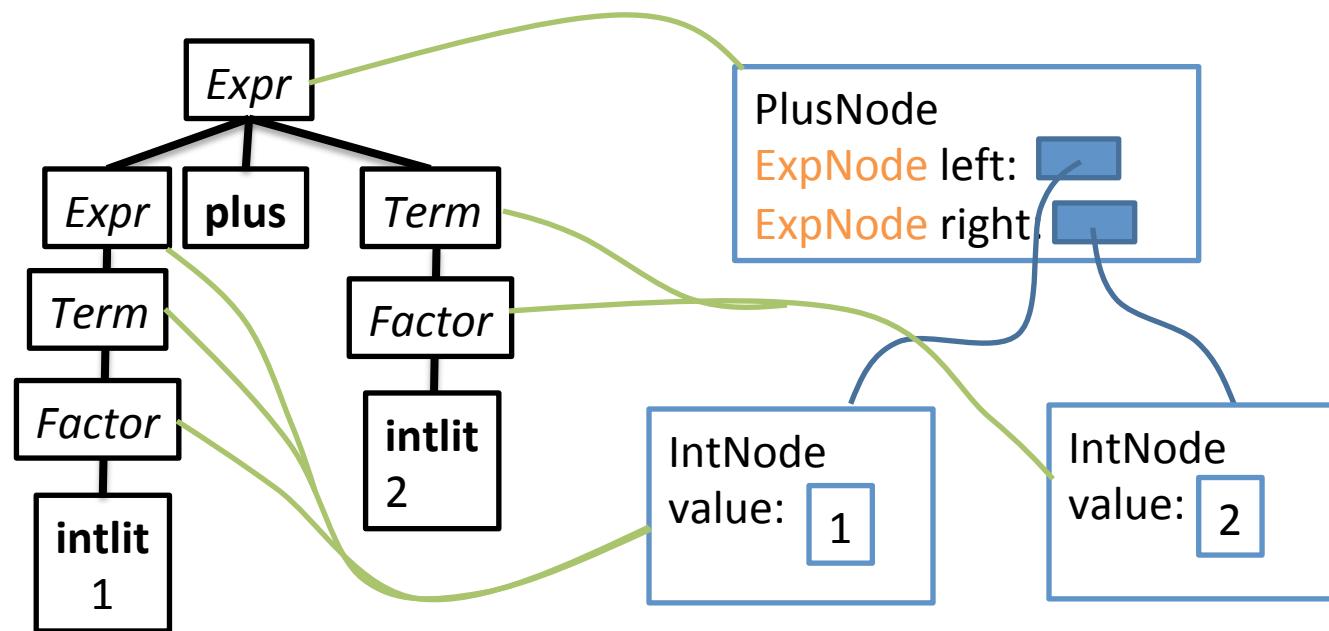
*Term1.trans = new TimesNode(Term2.trans, Factor.trans)*

*Term.trans = Factor.trans*

*Factor.trans = new IntNode(intlit.value)*

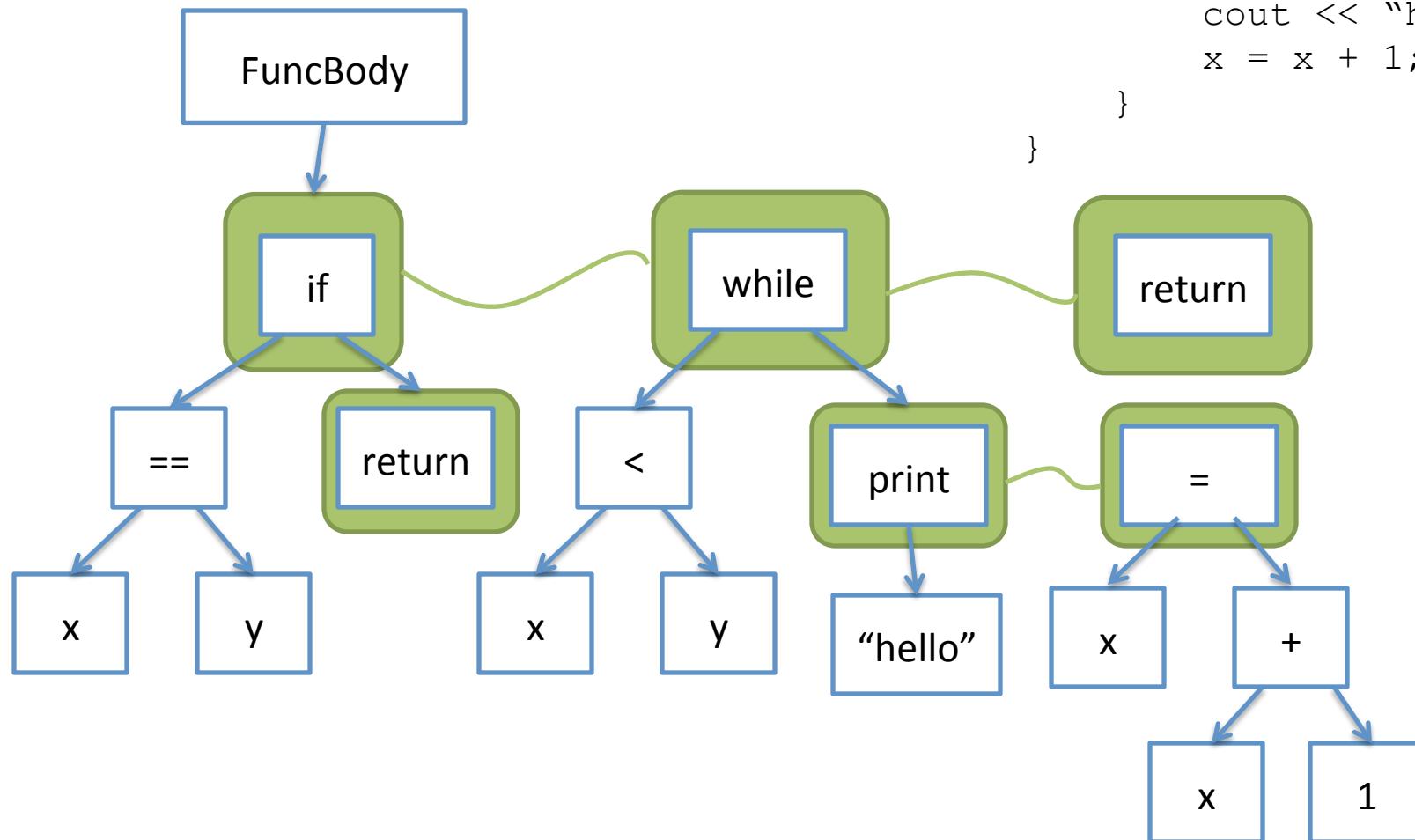
*Factor.trans = Expr.trans*

Example: 1 + 2



# An AST for a code snippet

```
void foo(int x, int y) {  
    if (x == y) {  
        return;  
    }  
    while (x < y) {  
        cout << "hello";  
        x = x + 1;  
    }  
}
```



# Summary (1 of 2)

Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser.

Today we learned about

- Syntax-Directed Translation (SDT)
  - Consumes a parse tree with actions
  - Actions yield some result
- Abstract Syntax Trees (ASTs)
  - The result of SDT for parsing in a compiler
  - Some practical examples of ASTs

在计算机科学中，抽象语法树（Abstract Syntax Tree, AST），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于 if-condition-then 这样的条件跳转语句，可以使用带有三个分支的节点来表示。

和抽象语法树相对的是具体语法树（通常称作分析树）。一般的，在源代码的翻译和编译过程中，语法分析器创建出分析树，然后从分析树生成AST。一旦AST被创建出来，在后续的处理过程中，比如语义分析阶段，会添加一些信息。

# Summary (2 of 2)

Scanner

Language abstraction: RegEx  
Output: Token Stream  
Tool: JLex  
Implementation: DFA walking via table

Parser

Language abstraction: CFG  
Output: AST by way of Parse Tree  
Tool: Java CUP  
Implementation: ???

Next time

Next week

Scanner

Language abstraction: RegEx  
Output: Token Stream  
Tool: JLex  
Implementation: DFA walking via table

P<sub>2</sub>

Parser

Language abstraction: CFG + SDE  
Output: AST by way of Parse Tree  
Tool: Java CUP thursday  
Implementation: ??? next weeks

# Java CUP

# Last Time

What do we want?

- An AST

When do we want it?

- Now!



# This Time

A little review of ASTs

The philosophy and use of a *Parser Generator*

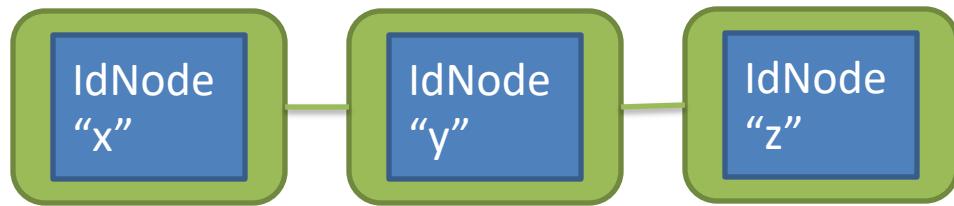
# Translating Lists

CFG

$IdList \rightarrow id$

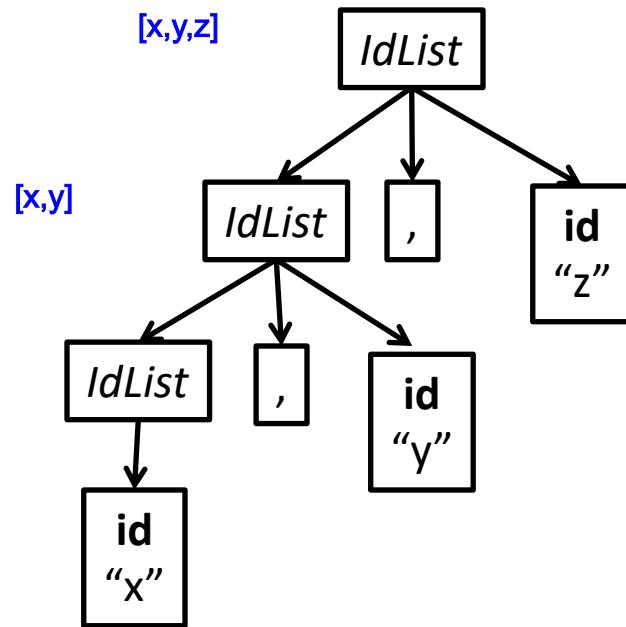
|  $IdList \text{ comma } id$

AST



Input

x, y, z



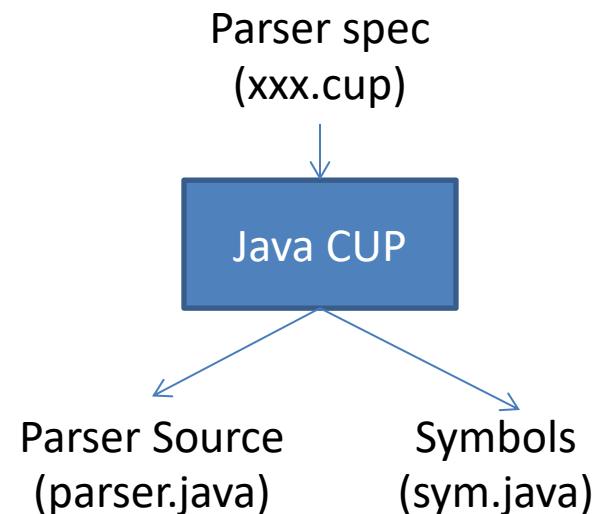
# Parser Generators

Tools that take an SDT spec and build an AST

- YACC: Yet Another Compiler Compiler
- Java CUP: Constructor of Useful Parsers

Conceptually similar to JLex

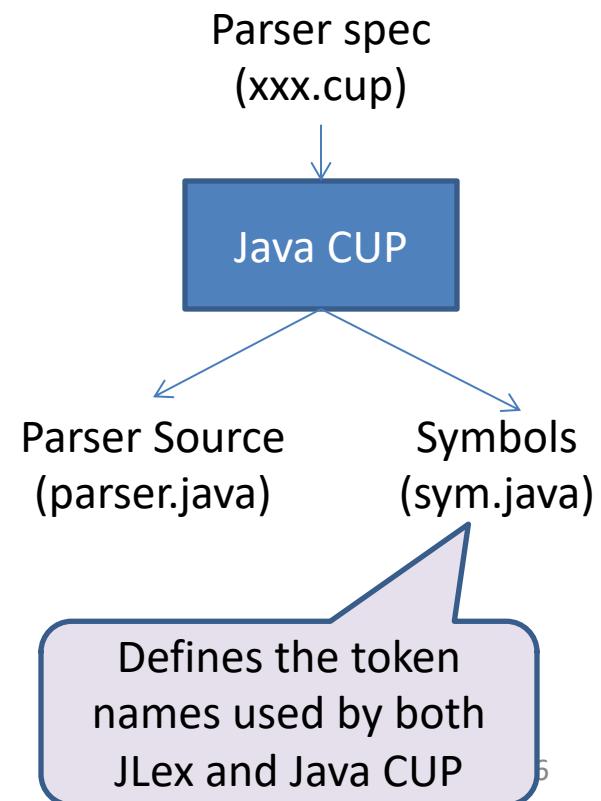
- Input: Language rules + actions
- Output: java code



# Java CUP

## Parser.java

- Constructor takes arg of type Scanner (i.e., yylex)
- Contains a parsing method
  - return: Symbol whose value contains translation of root nonterminal
- Uses output of JLex
  - Depends on scanner and TokenVals
- Uses defs of AST classes
  - Also in xxx.cup



# Java CUP Input Spec

Terminal & nonterminal declarations

Optional precedence and associativity declarations

Grammar with rules and actions

一般cup input由这三类input组成

这里遵循Lowest to highest priority, 即  
加法优先度最低, nonassoc less 最高

## Grammar rules

```
Expr ::= intliteral
      | id
      | Expr plus Expr
      | Expr times Expr
      | lparens Expr rparens
```

## Terminal and Nonterminals

```
terminal intliteral;
terminal id;
terminal plus;           lowest
terminal times;          precedence
terminal lparen;
terminal rparen;
non terminal Expr;
```

## Precedence and Associativity

```
precedence left plus;
precedence left times;
precedence nonassoc less;
```

# Java CUP Example

Assume ExpNode

Subclasses

- PlusNode, TimesNode have 2 children for operands
- IdNode has a String field
- IntLitNode has an int field

Assume Token classes

- IntLitTokenVal with field `intVal` for int literal token
- IdTokenVal with field `idVal` for identifier token

## Step 1: Add types to terminals

```
terminal IntLitTokenVal intliteral;  
terminal IdTokenVal id;  
terminal plus;  
terminal times;  
terminal lparen;  
terminal rparen;
```

```
non terminal ExpNode expr;
```

**Expr**

# Java CUP Example

```
Expr ::= intliteral
       { :
         : }
       | id
         { :
           : }
       | Expr plus Expr
         { :
           : }
       | Expr times Expr
         { :
           : }
       | lparen Expr rparen
         { :
           : }
       ;
;
```

# Java CUP Example

```
Expr ::= intliteral:i
       { :
         RESULT = new IntLitNode(i.intValue);
       }
     | id
       { :
         :
       }
     | Expr plus Expr
       { :
         :
       }
     | Expr times Expr
       { :
         :
       }
     | lparen Expr rparen
       { :
         :
       }
     ;
```

# Java CUP Example

all these nodes are expnode

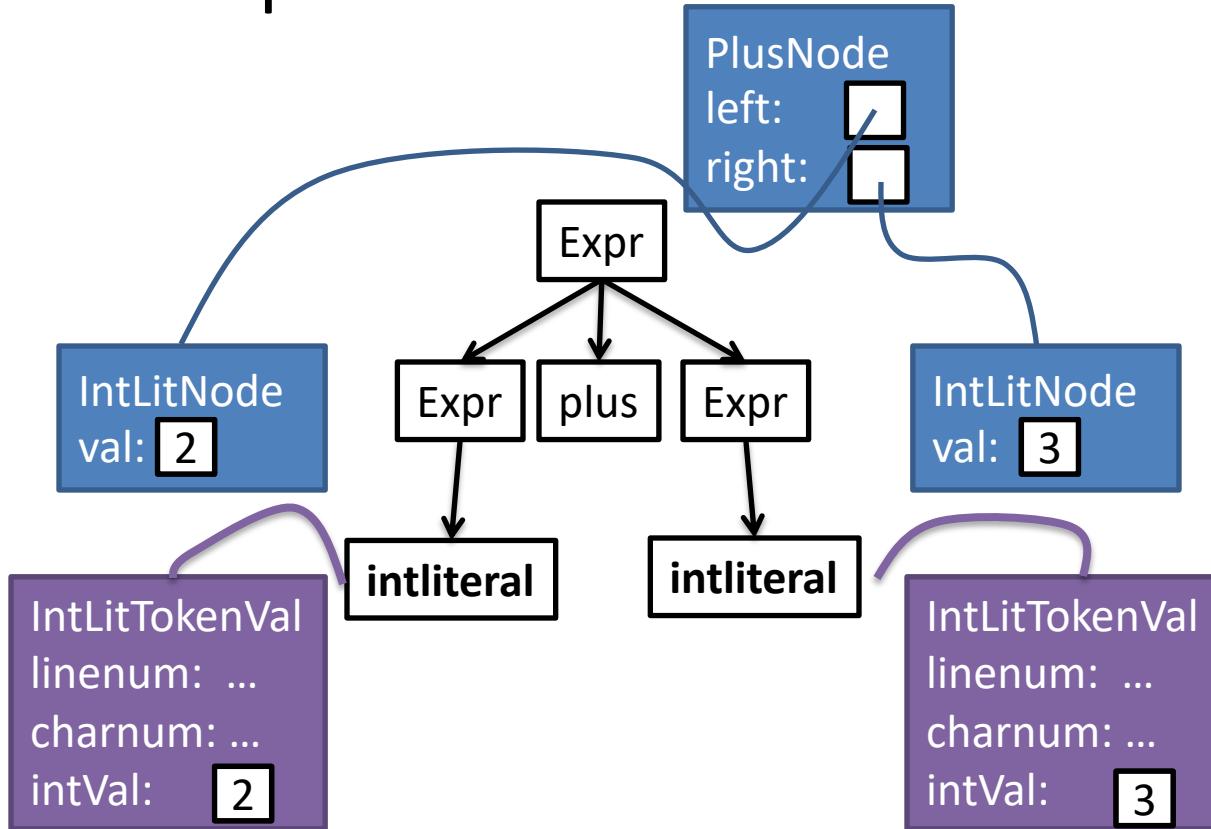
```
Expr ::= intliteral:i
      { :
          RESULT = new IntLitNode(i.intValue);
      }
      | id:i
      { :
          RESULT = new IdNode(i.idVal);
      }
      | Expr:e1 plus Expr:e2
      { :
          RESULT = new PlusNode(e1,e2);
      }
      | Expr:e1 times Expr:e2
      { :
          RESULT = new TimesNode(e1,e2);
      }
      | lparen Expr:e rparen
      { :
          RESULT = e;
      }
;
```

plus needs to define two elements

( ) is being omitted

# Java CUP Example

Input: 2 + 3



Purple = Terminal Token (Built by Scanner)  
Blue = Symbol (Built by Parser)

left associative -> 这个符号先行运算

right associative -> ~后行运算

# Java CUP Demo



# Announcements

Midterm is on March 10<sup>th</sup> in this room

# Bottom-up parsing algorithms

*Cocke–Younger–Kasami algorithm*

*and*

*Chomsky Normal Form*

# Last time

Showed how to use JavaCUP for getting ASTs  
But we never saw HOW the parser works

# This time

Dip our toe into parsing

- Approaches to Parsing
- CFG transformations
  - Useless non-terminals
  - Chomsky Normal Form: A form of grammar that's easier to deal with
- CYK: powerful, heavyweight approach to parsing

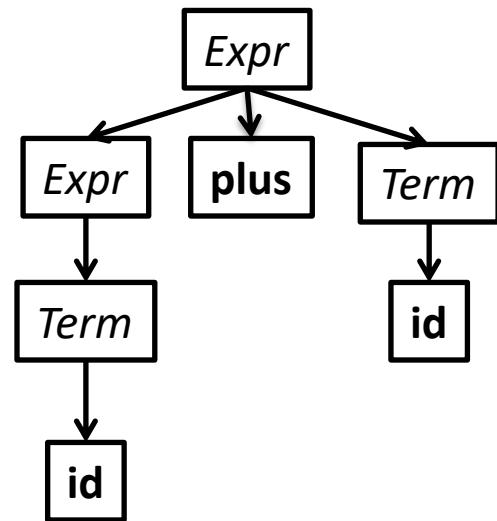
# Approaches to parsing

Top Down / “Goal driven”

- Start at root of parse tree, grow downward to match the string

Bottom Up / “Data Driven”

- Start at terminal, generate subtrees until you get to the start



# CYK: A general approach to Parsing (Cocke–Younger–Kasami algorithm)

n= # of tokens

Operates in  $O(n^3)$

this algo is not good enough

Works Bottom-Up

Only takes a grammar in Chomsky Normal Form

- This will not turn out to be a limitation

# Chomsky Normal Form

All rules must be one of two forms:

1  $X \rightarrow t$  (terminal)

2  $X \rightarrow A B$  or go to strictly two nonterminals, A, B

The only rule allowed to derive epsilon is the start S

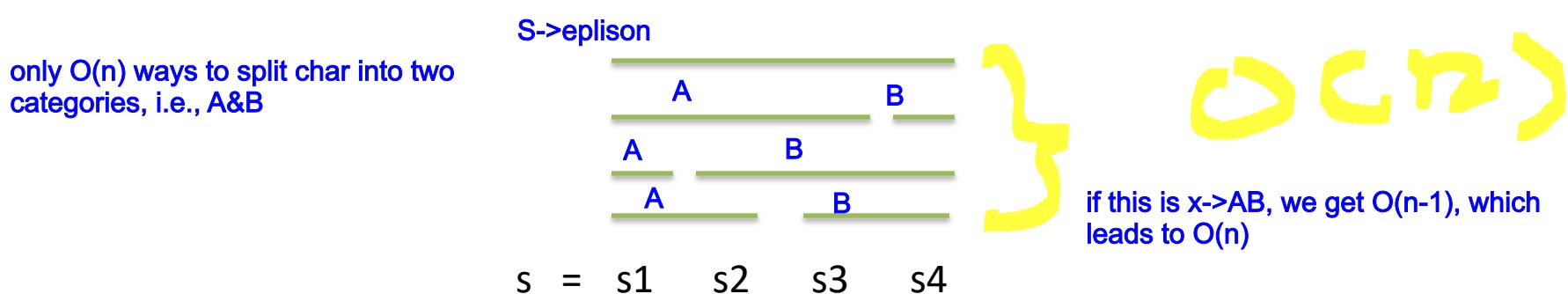
3  $S \rightarrow \epsilon$



$X \rightarrow aX$  ~~X~~  
 $X \rightarrow ABC$  ~~X~~

# What CNF buys CYK

- The fact that non-terminals come **in pairs** allows you to think of a subtree as a subspan of the input **can not be epsilon**
- The fact that non-terminals are **not nullable** (except for start) means that each subspan has at least one character



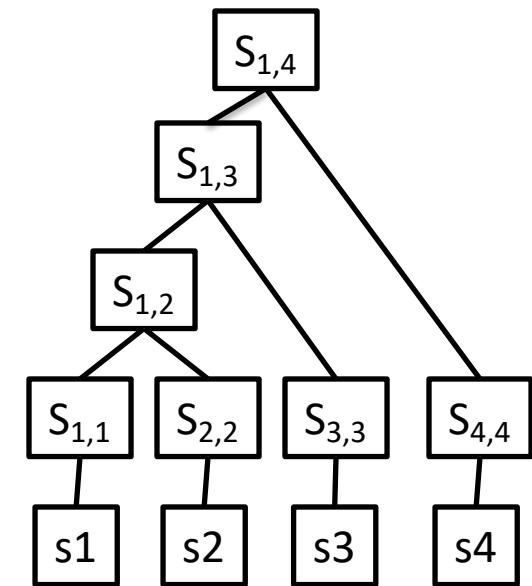
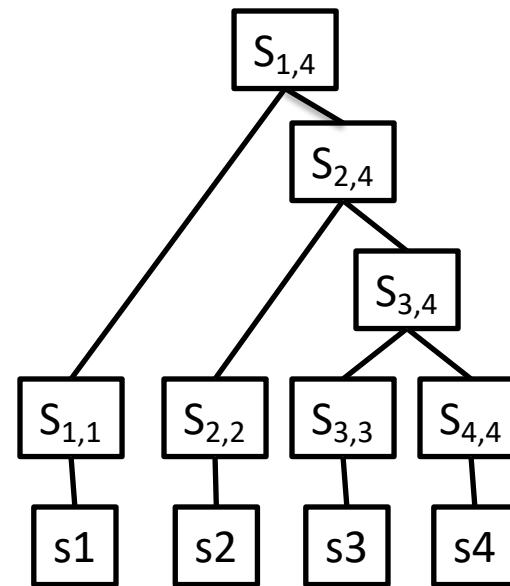
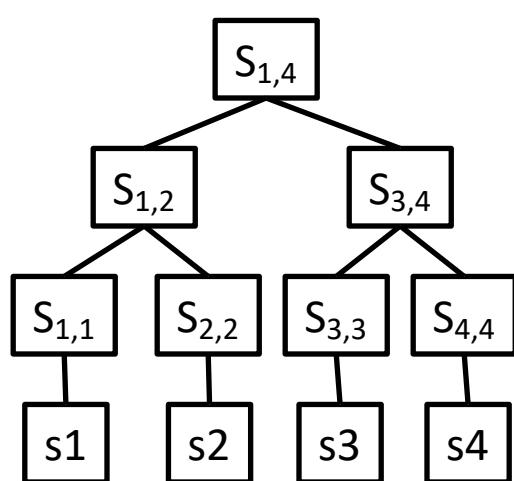
# CYK: Dynamic Programming

$X \rightarrow t$

Prods. form the leaves of the parse tree

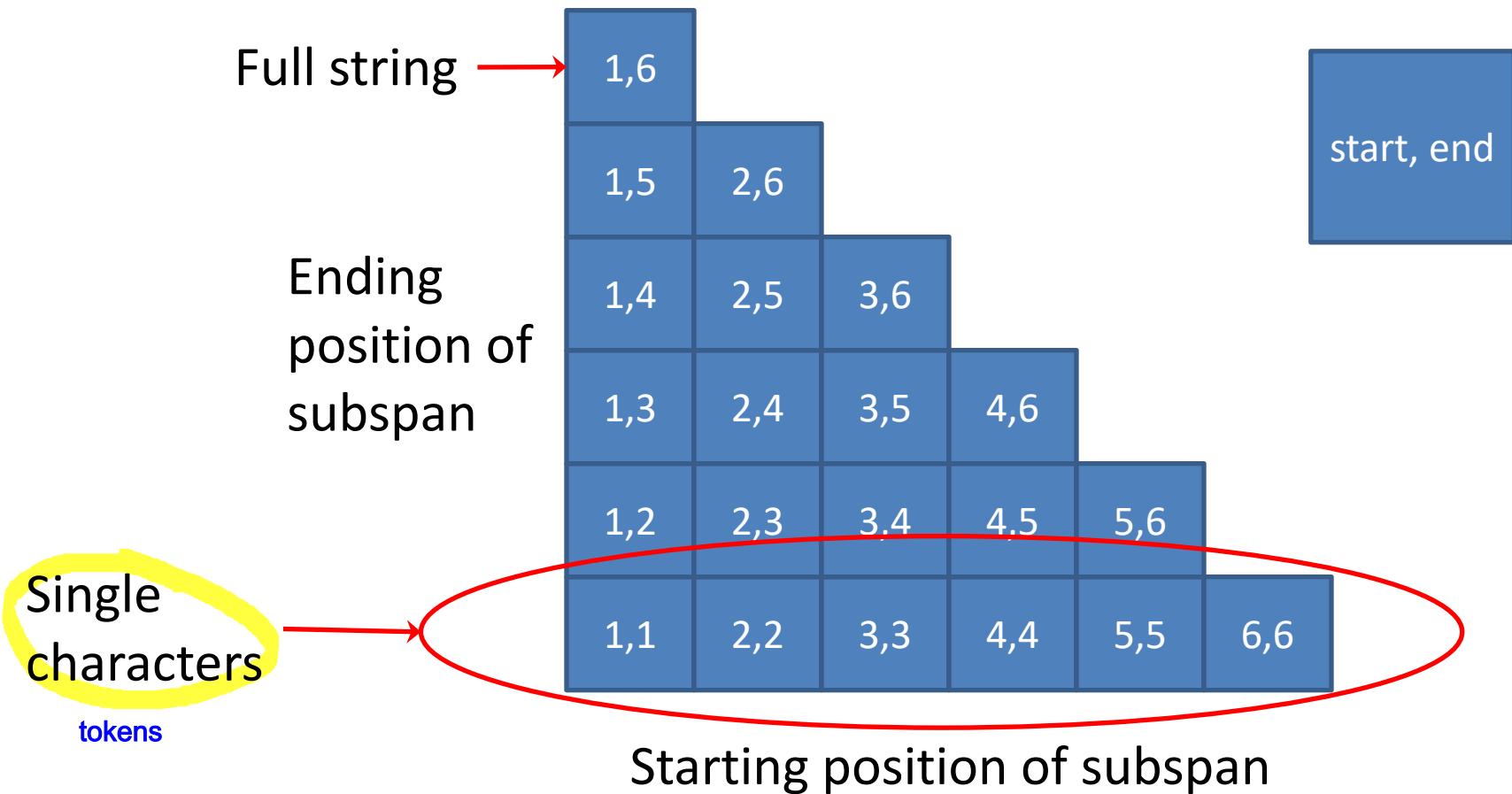
$X \rightarrow A B$

Form binary nodes



# Running CYK ...

Track every viable subtree from leaf to root. Here are all the subspans for a string of 6 terminals:

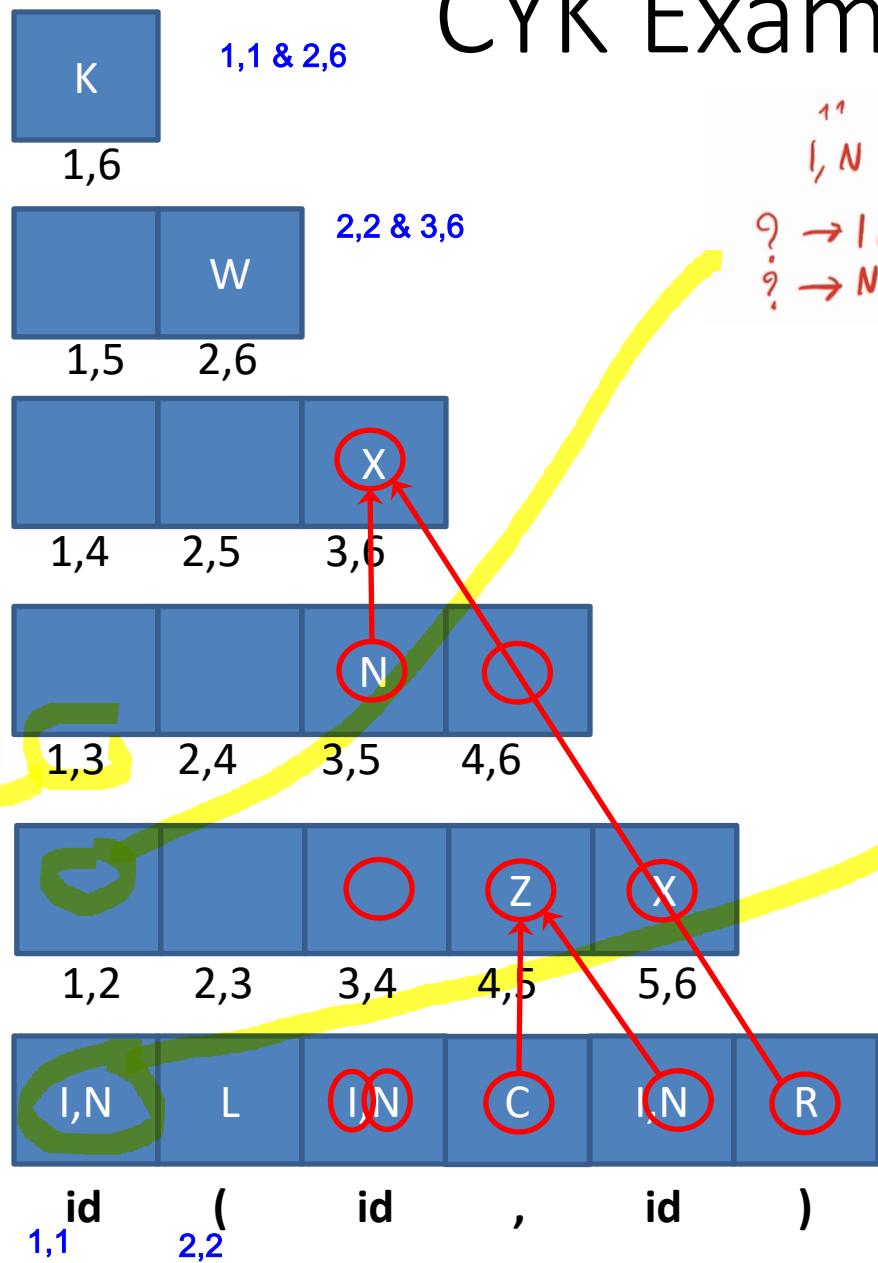


# CYK Example

no way to parse 1,3; since 2,3 and 1,2 is empty; we don't have empty results at the righttbl

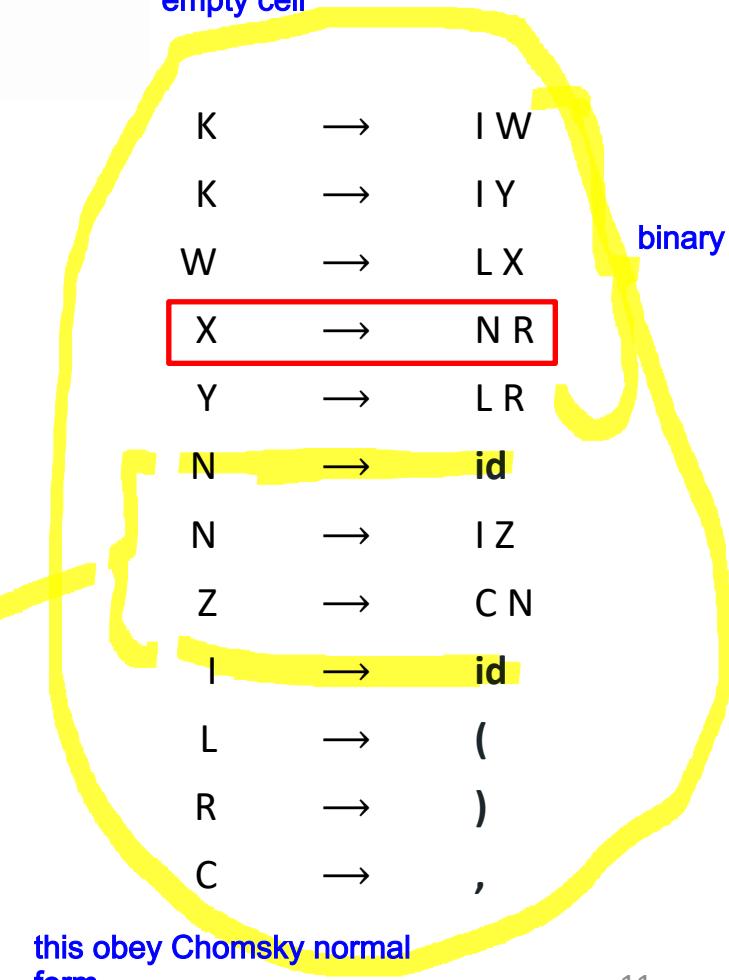
11 23  
12 33

len==1

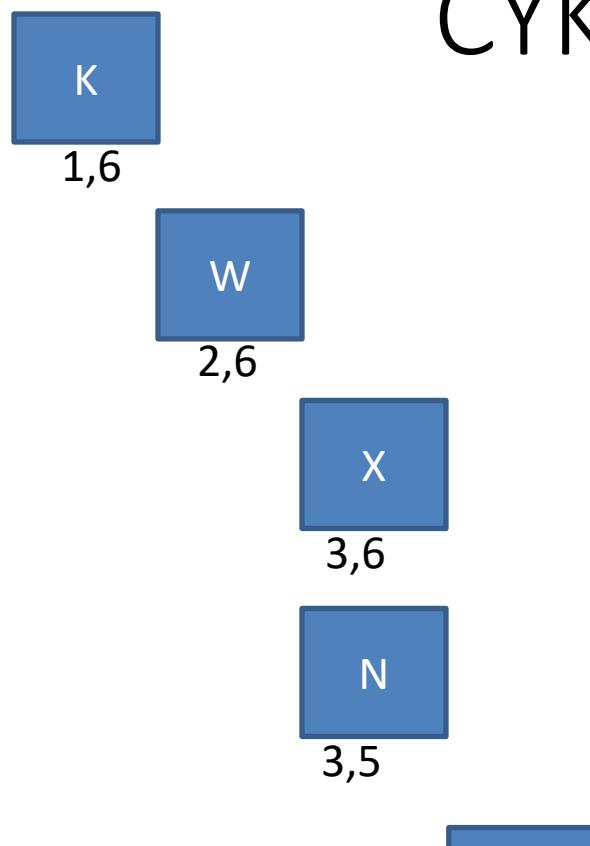


<sup>11</sup>  
I, N  
<sup>22</sup>  
L  
? → IL  
? → NL

no token -> either IL or NL; so this should be empty cell

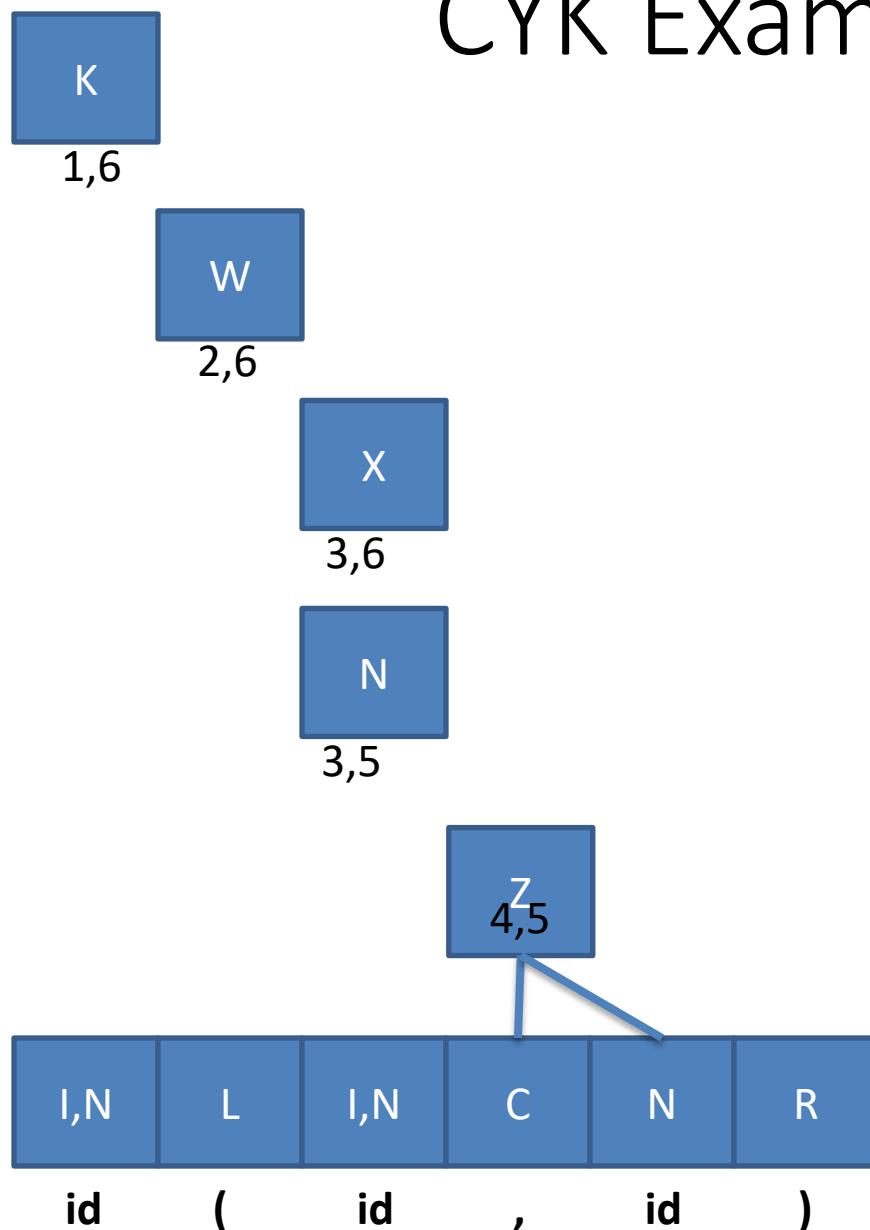


# CYK Example



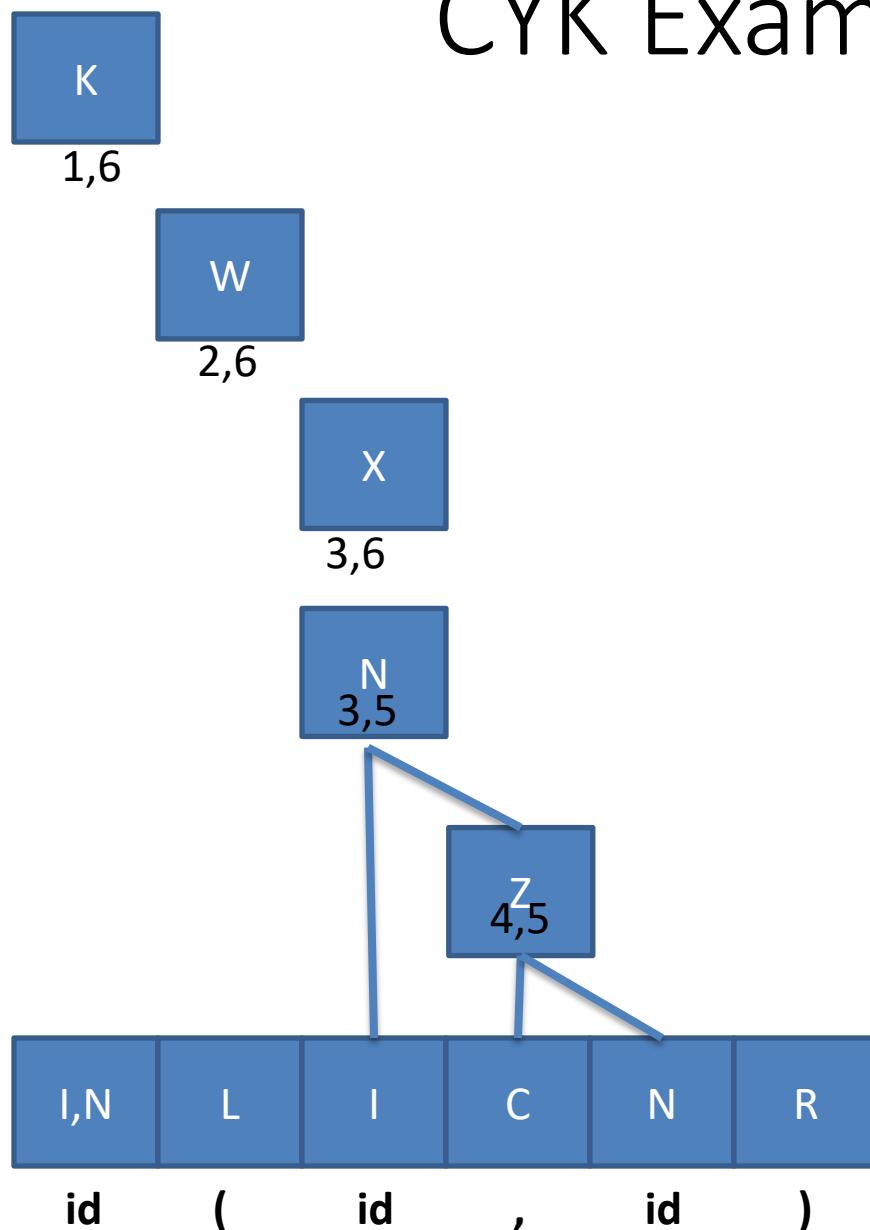
K	$\rightarrow$	I W
K	$\rightarrow$	I Y
W	$\rightarrow$	L X
X	$\rightarrow$	N R
Y	$\rightarrow$	L R
N	$\rightarrow$	id
N	$\rightarrow$	I Z
Z	$\rightarrow$	C N
I	$\rightarrow$	id
L	$\rightarrow$	(
R	$\rightarrow$	)
C	$\rightarrow$	,

# CYK Example

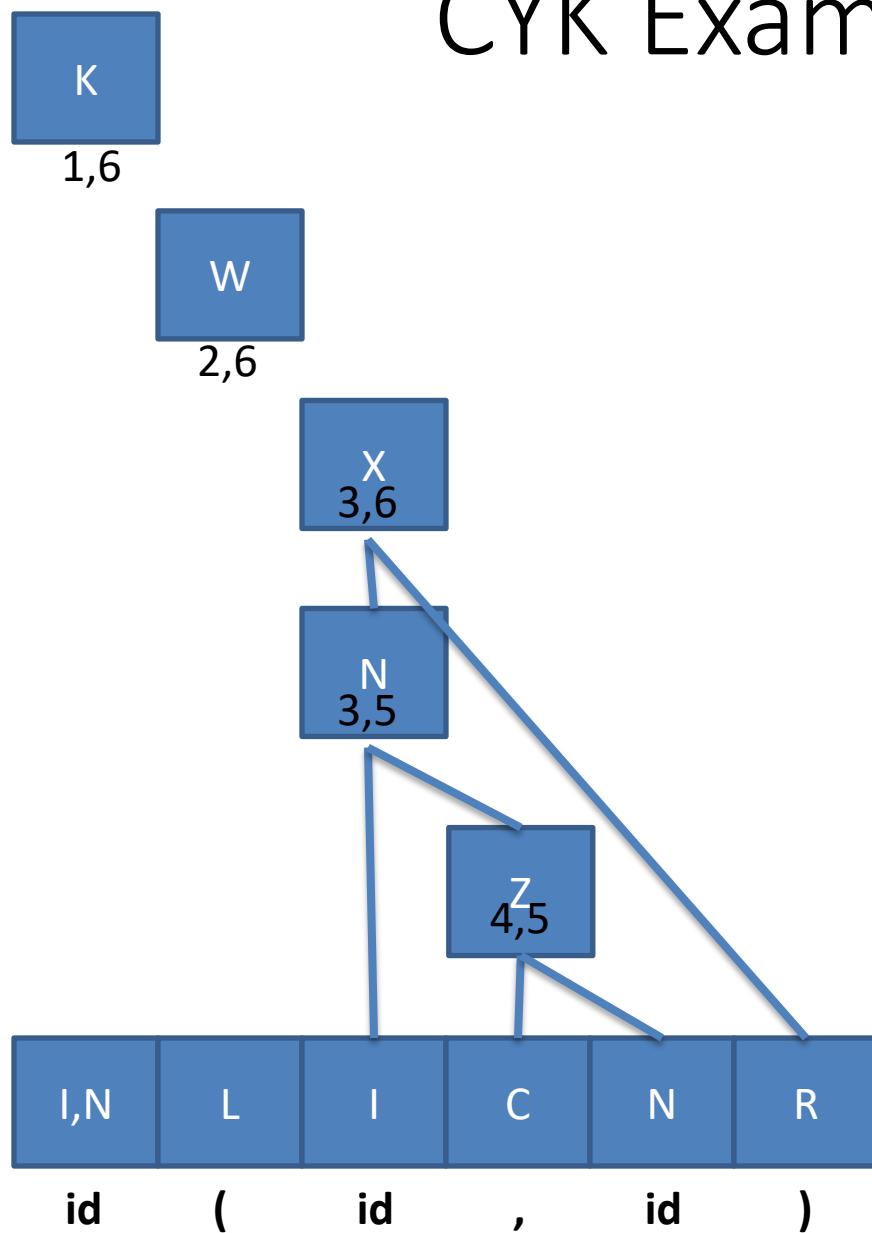


<b>K</b>	$\rightarrow$	I W
<b>K</b>	$\rightarrow$	I Y
<b>W</b>	$\rightarrow$	L X
<b>X</b>	$\rightarrow$	N R
<b>Y</b>	$\rightarrow$	L R
<b>N</b>	$\rightarrow$	<b>id</b>
<b>N</b>	$\rightarrow$	I Z
<b>Z</b>	$\rightarrow$	<b>C N</b>
<b>I</b>	$\rightarrow$	<b>id</b>
<b>L</b>	$\rightarrow$	<b>(</b>
<b>R</b>	$\rightarrow$	<b>)</b>
<b>C</b>	$\rightarrow$	<b>,</b>

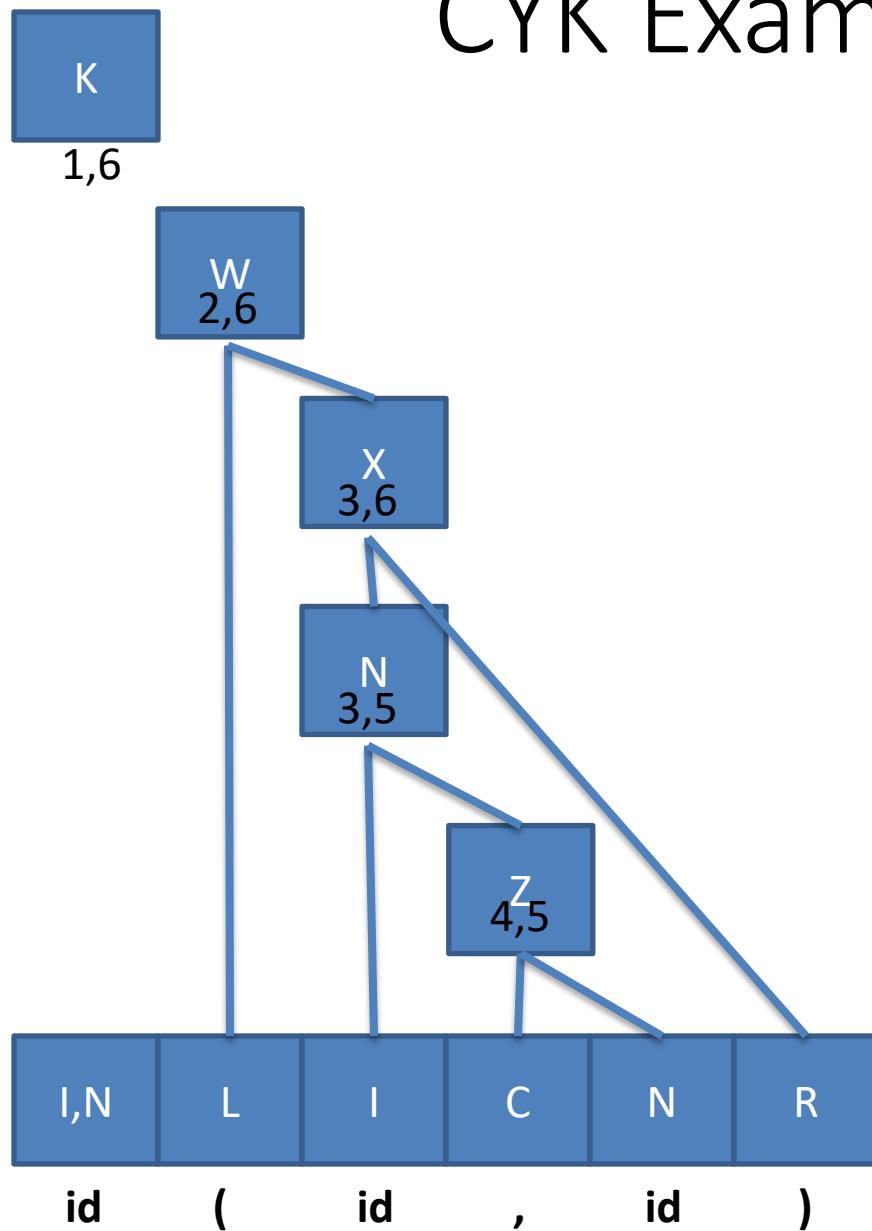
# CYK Example



# CYK Example

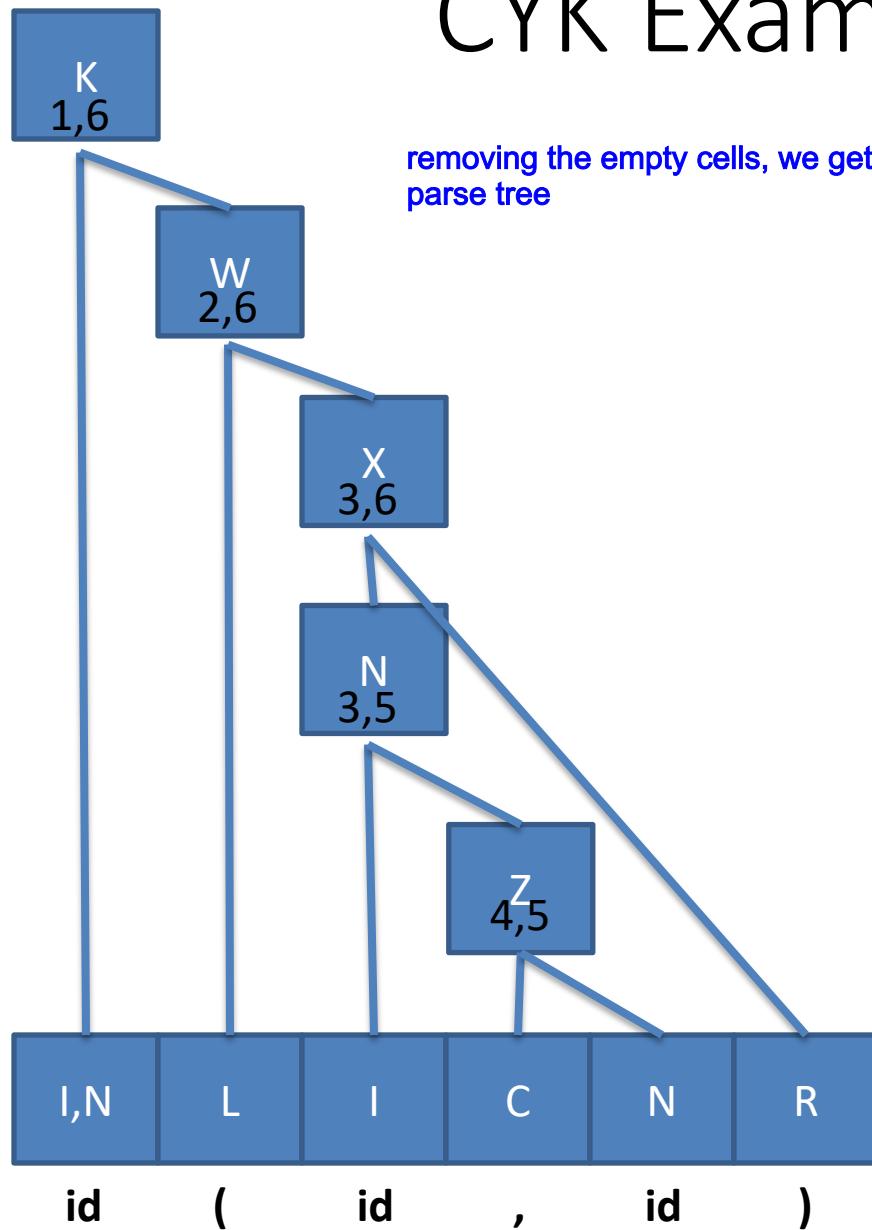


# CYK Example



<code>K</code>	$\rightarrow$	<code>I W</code>
<code>K</code>	$\rightarrow$	<code>I Y</code>
<code>W</code>	$\rightarrow$	<code>L X</code>
<code>X</code>	$\rightarrow$	<code>N R</code>
<code>Y</code>	$\rightarrow$	<code>L R</code>
<code>N</code>	$\rightarrow$	<code>id</code>
<code>N</code>	$\rightarrow$	<code>I Z</code>
<code>Z</code>	$\rightarrow$	<code>C N</code>
<code>I</code>	$\rightarrow$	<code>id</code>
<code>L</code>	$\rightarrow$	<code>(</code>
<code>R</code>	$\rightarrow$	<code>)</code>
<code>C</code>	$\rightarrow$	<code>,</code>

# CYK Example



K	$\rightarrow$	I W
K	$\rightarrow$	I Y
W	$\rightarrow$	L X
X	$\rightarrow$	N R
Y	$\rightarrow$	L R
N	$\rightarrow$	id
N	$\rightarrow$	I Z
Z	$\rightarrow$	C N
I	$\rightarrow$	id
L	$\rightarrow$	(
R	$\rightarrow$	)
C	$\rightarrow$	,

# Cleaning up our grammars

We want to avoid unnecessary work

- Remove *useless* rules



# Eliminating Useless Nonterminals

1. If a nonterminal cannot derive a sequence of terminal symbols then it is useless
2. If a nonterminal cannot be derived from the start symbol, then it is useless

所以，先从能output terminal的符号开始看，

# Eliminate Useless Nonterms

If a nonterminal cannot derive a sequence of terminal symbols, then it is useless

Mark all terminal symbols

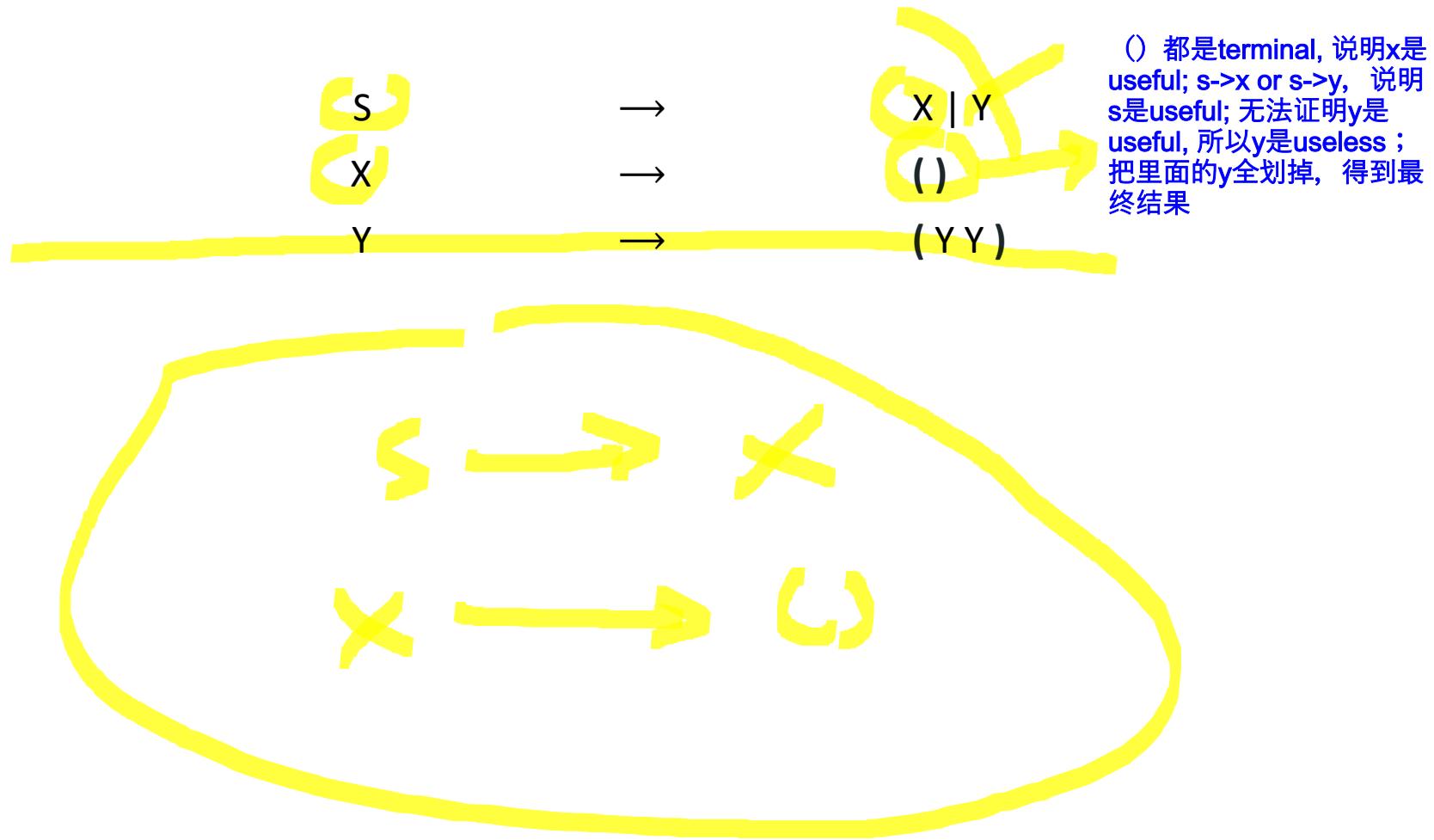
Repeat

If all symbols on the righthand side of a production are marked

mark the lefthand side

Until no more non-terminals can be marked

# Example:



# Eliminate Useless Nonterms

If a nonterminal  
cannot be derived  
from the start  
symbol, then it is  
useless

Mark the start symbol

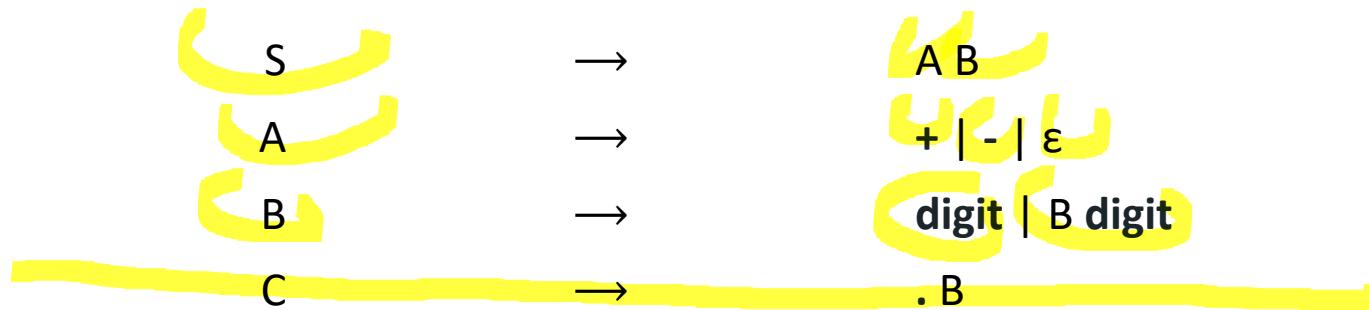
Repeat

If the lefthand side of a  
production is marked

mark all righthand  
non-terminal

Until no more non-terminals  
can be marked

# Example:



# Chomsky Normal Form

## 4 Steps

- Eliminate epsilon rules  $x \rightarrow \epsilon$
- Eliminate unit rules  $x \rightarrow A$
- Fix productions with terminals on RHS  $x \rightarrow A($
- Fix productions with  $> 2$  nonterminals on RHS  $x \rightarrow ABC$

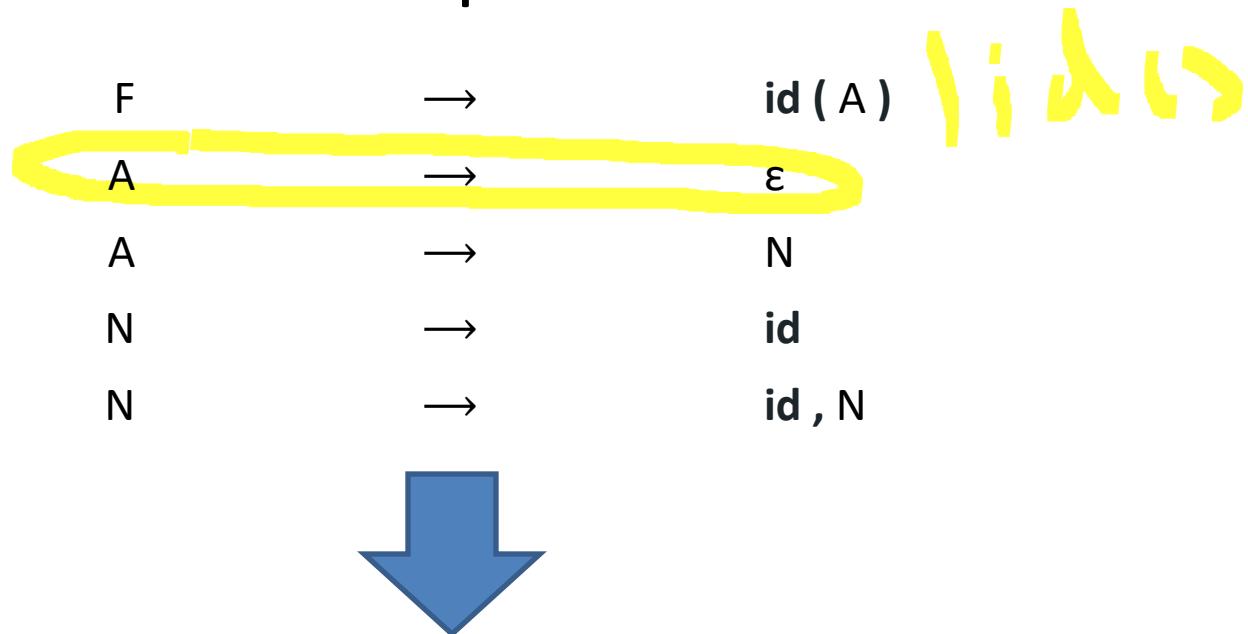
# Eliminate (Most) Epsilon Productions

If a nonterminal  $A$  immediately derives epsilon

- Make copies of all rules with  $A$  on the RHS and delete all combinations of  $A$  in those copies

# Example 1

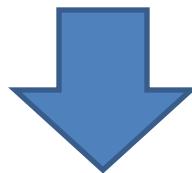
~~remove~~



$F$	$\rightarrow$	$\text{id} ( A )$
$F$	$\rightarrow$	$\text{id} ( )$
$A$	$\rightarrow$	$N$
$N$	$\rightarrow$	$\text{id}$
$N$	$\rightarrow$	$\text{id}, N$

# Example 2

$$\begin{array}{lll} X & \rightarrow & A \times A \times A \\ A & \rightarrow & \varepsilon \\ A & \rightarrow & z \end{array}$$



$$\begin{array}{lll} X & \rightarrow & A \times A \times A \\ | & & A \times A \times y \\ | & & A \times y \times A \\ | & & A \times y \\ | & & x \times A \times y \\ | & & x \times A \times y \\ | & & x \times y \times A \\ | & & x \times y \end{array}$$

it needs to include all possibilities : either A is epsilon or A is not

we have  $2^3$  options,  
 $2^{(\text{number of } A)}$

# Eliminate Unit Productions

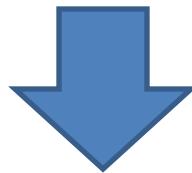
Productions of the form  $A \rightarrow B$  are called unit productions

Place B anywhere A could have appeared and remove the unit production

# Example 1

F	→	<b>id ( A )</b>
F	→	<b>id ( )</b>
A	→	N
N	→	<b>id</b>
N	→	<b>id , N</b>

everything with A is useless



F	→	<b>id ( N )</b>
F	→	<b>id ( )</b>
N	→	<b>id</b>
N	→	<b>id , N</b>

# Fix RHS Terminals

For productions with Terminals and something else on the RHS

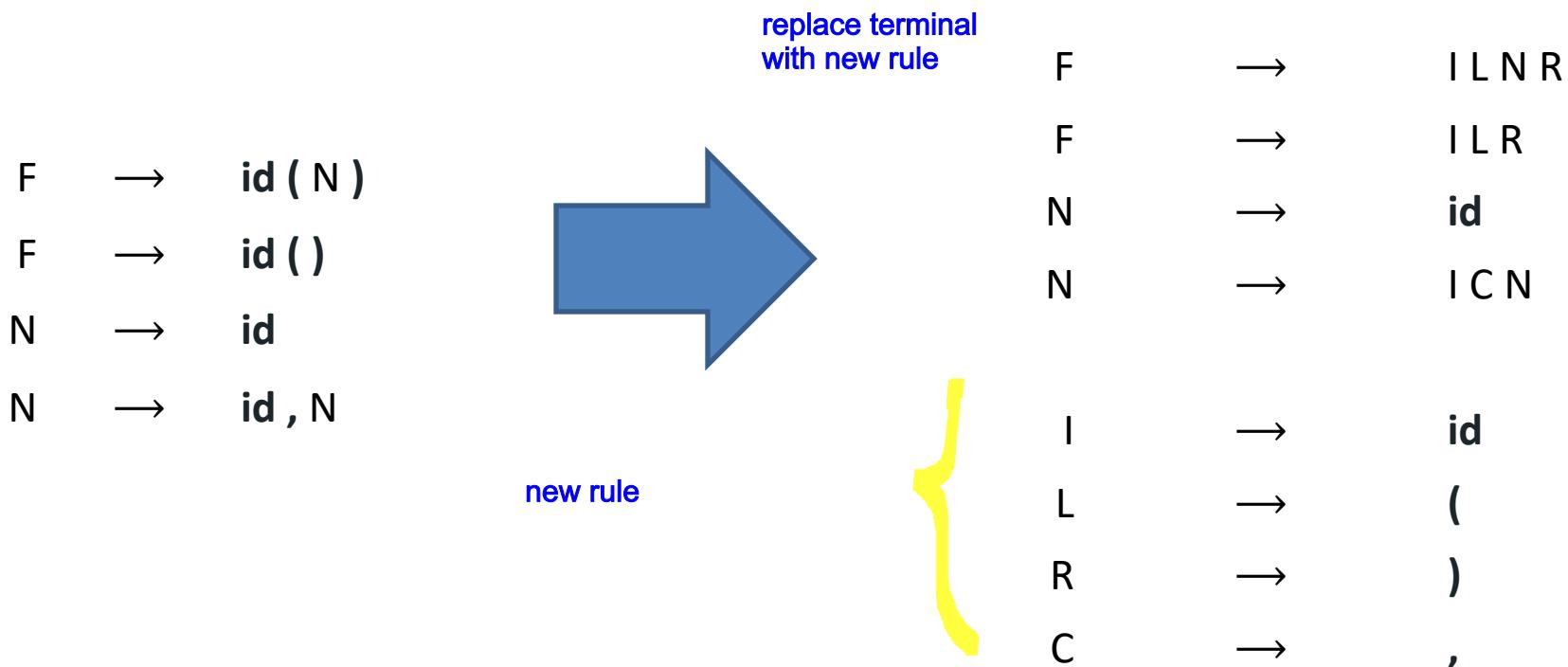
- For each terminal  $t$  add the rule

$$X \rightarrow t$$

Where  $X$  is a new non-terminal

- Replace  $t$  with  $X$  in the original rules

# Example



# Fix RHS Nonterminals

For productions with  $> 2$  Nonterminals on the RHS

- Replace all but the *first* nonterminal with a new nonterminal
- Add a rule from the new nonterminal to the replaced nonterminal sequence
- Repeat

# Example

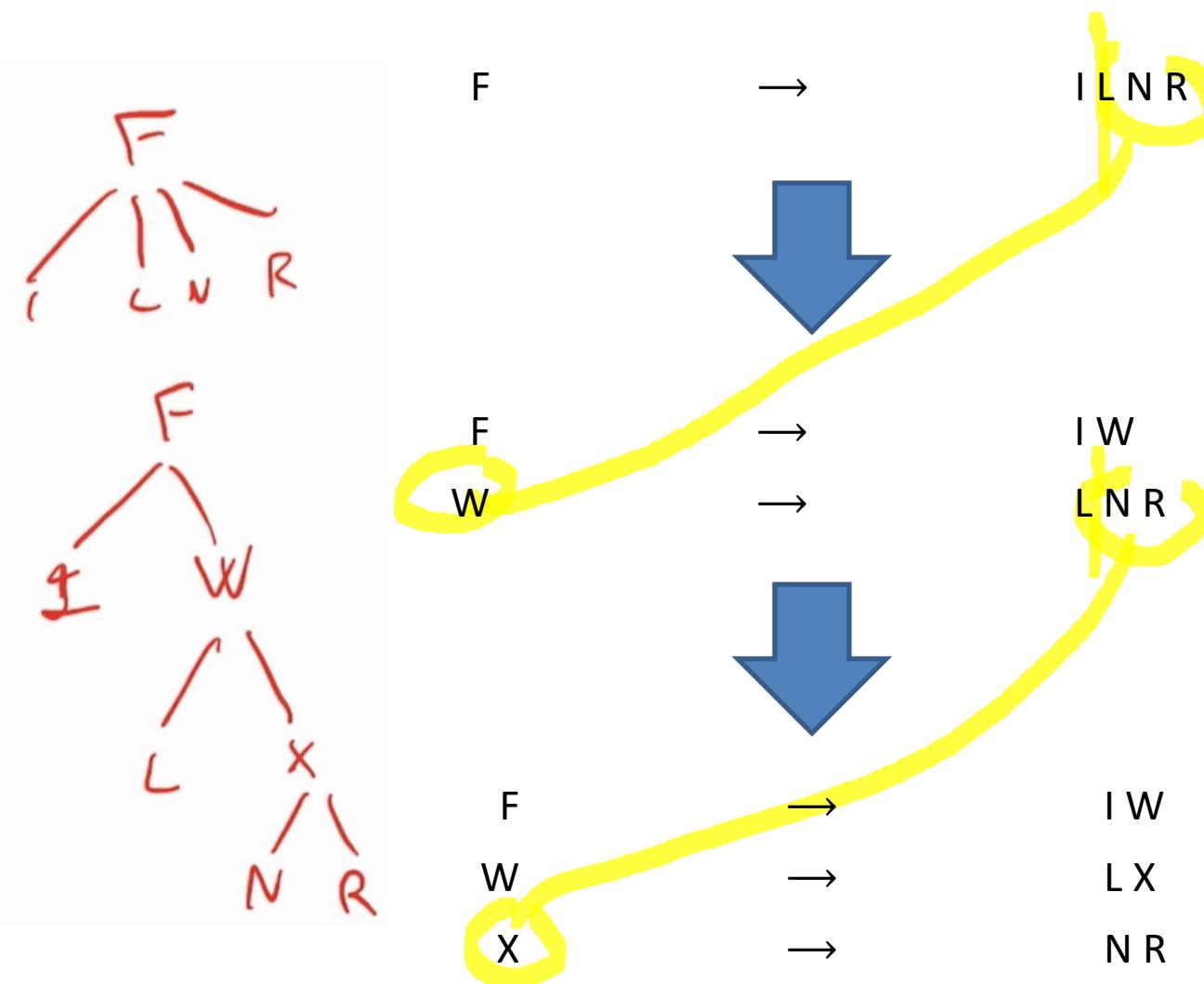


figure below explains why  
 $O(n^3)$

# Parsing is Tough

CYK parses an arbitrary CFG, but

- $O(n^3)$
- Too slow!

length of input OR size of grammar  
which is more important?  
ANS: length of input

For special class of grammars

- $O(n)$
- Includes LL(1) and LALR(1)

can run in all grammar

only restricted grammar, but with  
 $O(n)$

# Classes of Grammars

## LL(1)

- Scans input from Left-to-right (first L)
- Builds a Leftmost Derivation (second L)
- Can peek (1) token ahead of the token being parsed
- Top-down “predictive parsers”

implemented in Javacup and Bison

## LALR(1)

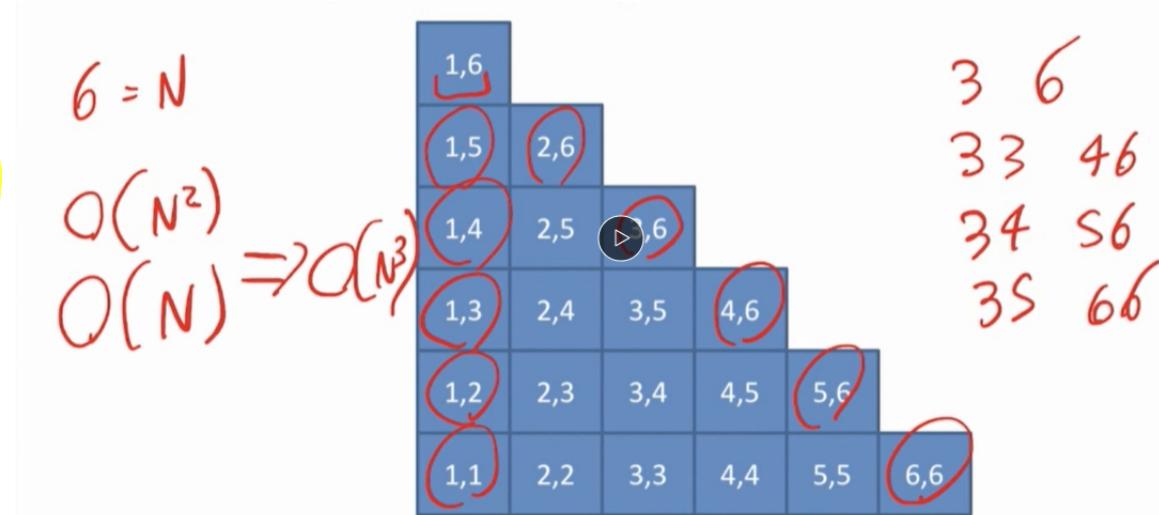
- Uses special lookahead procedure (LA)
- Scans input from Left-to-right (second L)
- Rightmost derivation (R)
- Can also peek (1) token ahead

bottom-up

LALR(1) strictly more powerful, much harder to understand

# In summary

We talked about how to parse with CYK and Chomsky Normal Form grammars



# Top-down parsing

# Parsing: Review of the Big Picture (1)

- Context-free grammars (CFGs)
  - Generation:  $G \rightarrow L(G)$
  - Recognition: Given  $w$ , is  $w \in L(G)$ ?
- Translation
  - Given  $w \in L(G)$ , create a ( $G$ ) parse tree for  $w$
  - Given  $w \in L(G)$ , create an AST for  $w$ 
    - The AST is passed to the next component of our compiler

# Parsing: Review of the Big Picture (2)

- Algorithms
  - CYK
  - Top-down (“recursive-descent”) for LL(1) grammars
    - How to parse, given the appropriate parse table for  $G$
    - How to construct the parse table for  $G$
  - Bottom-up for LALR(1) grammars
    - How to parse, given the appropriate parse table for  $G$
    - How to construct the parse table for  $G$

# Last time

## CYK

- Step 1: get a grammar in Chomsky Normal Form
- Step 2: Build all possible parse trees bottom-up
  - Start with runs of 1 terminal
  - Connect 1-terminal runs into 2-terminal runs
  - Connect 1- and 2- terminal runs into 3-terminal runs
  - Connect 1- and 3- or 2- and 2- terminal runs into 4 terminal runs
  - ...
  - If we can connect the entire tree, rooted at the start symbol, we've found a valid parse

# Some Interesting properties of CYK

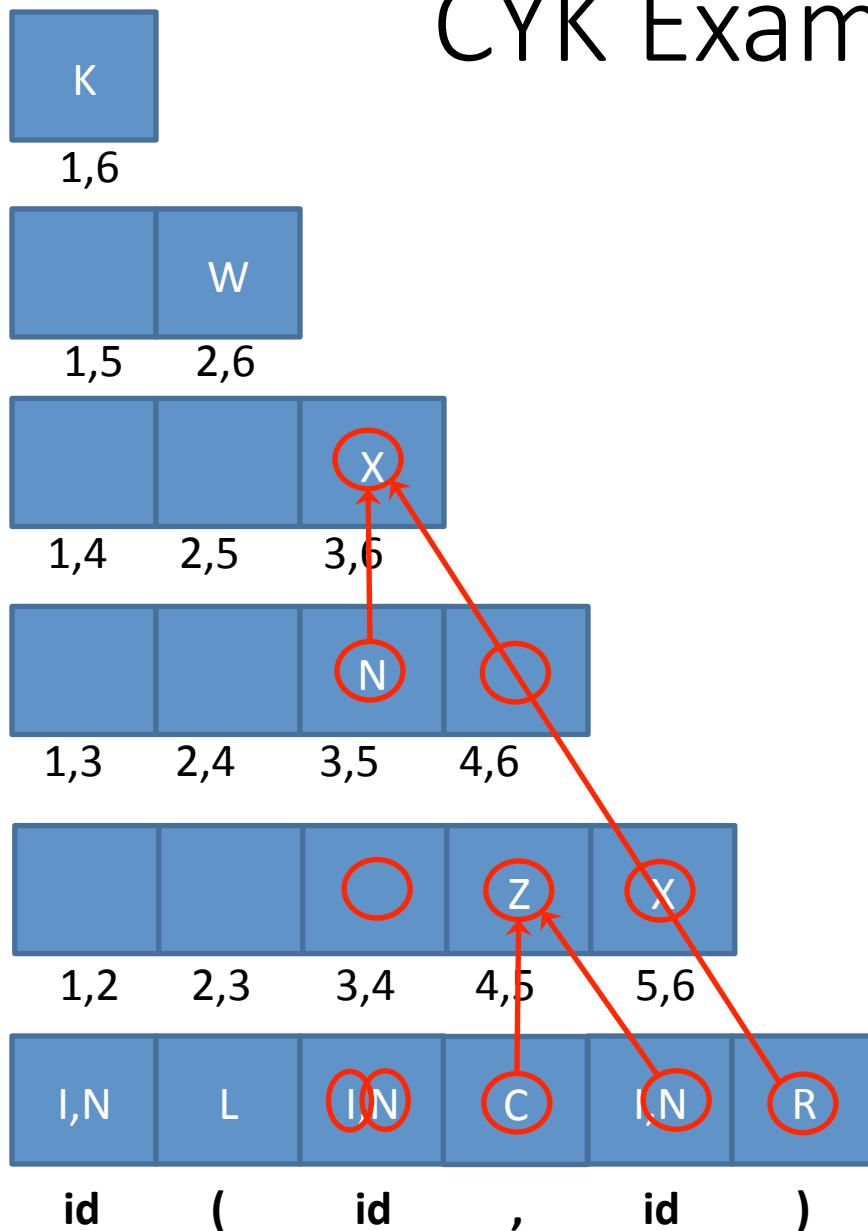
Very old algorithm

- Already well known in early 70s

No problems with ambiguous grammars:

- Gives a solution for *all* possible parse tree simultaneously

# CYK Example



K	→	I W
K	→	I Y
W	→	L X
X	→	N R
Y	→	L R
N	→	id
N	→	I Z
Z	→	C N
I	→	id
L	→	(
R	→	)
C	→	,

# Thinking about Language Design

## Balanced considerations

- Powerful enough to be useful
- Simple enough to be parseable

Syntax need not be complex for complex behaviors

- Guy Steele’s “Growing a Language”

[https://www.youtube.com/watch?v=\\_ahvzDzKdB0](https://www.youtube.com/watch?v=_ahvzDzKdB0)



# Restricting the Grammar

By restricting our grammars we can

- Detect ambiguity
- Build linear-time,  $O(n)$  parsers

LL(1) languages

- Particularly amenable to parsing
- Parseable by Predictive (top-down) parsers
  - Sometimes called recursive descent

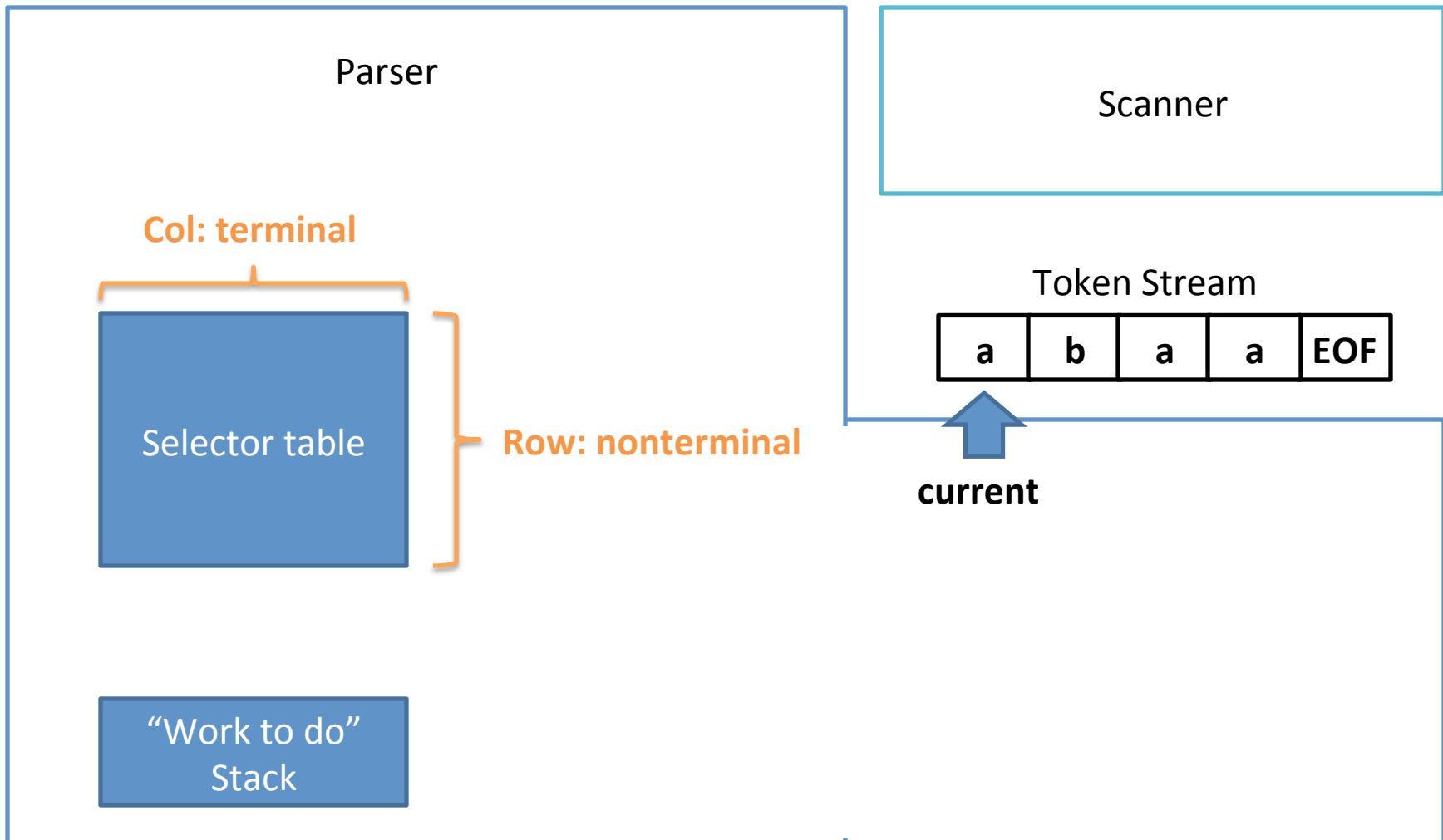
# Top-Down Parsers

Start at the **Start** symbol

Repeatedly: “predict” what production to use

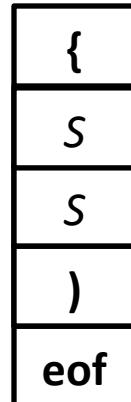
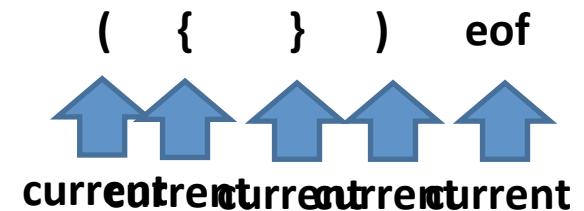
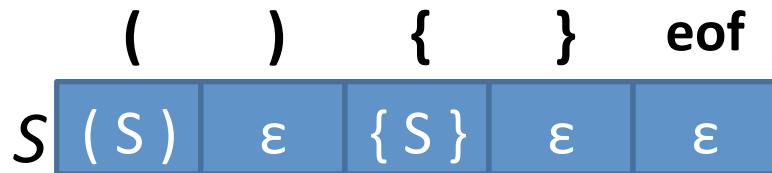
- Example: if the current token to be parsed is an **id**, no need to try productions that start with **intLiteral**
- This might seem simple, but keep in mind that a chain of productions may have to be used to get to the rule that handles, e.g., **id**

# Predictive Parser Sketch



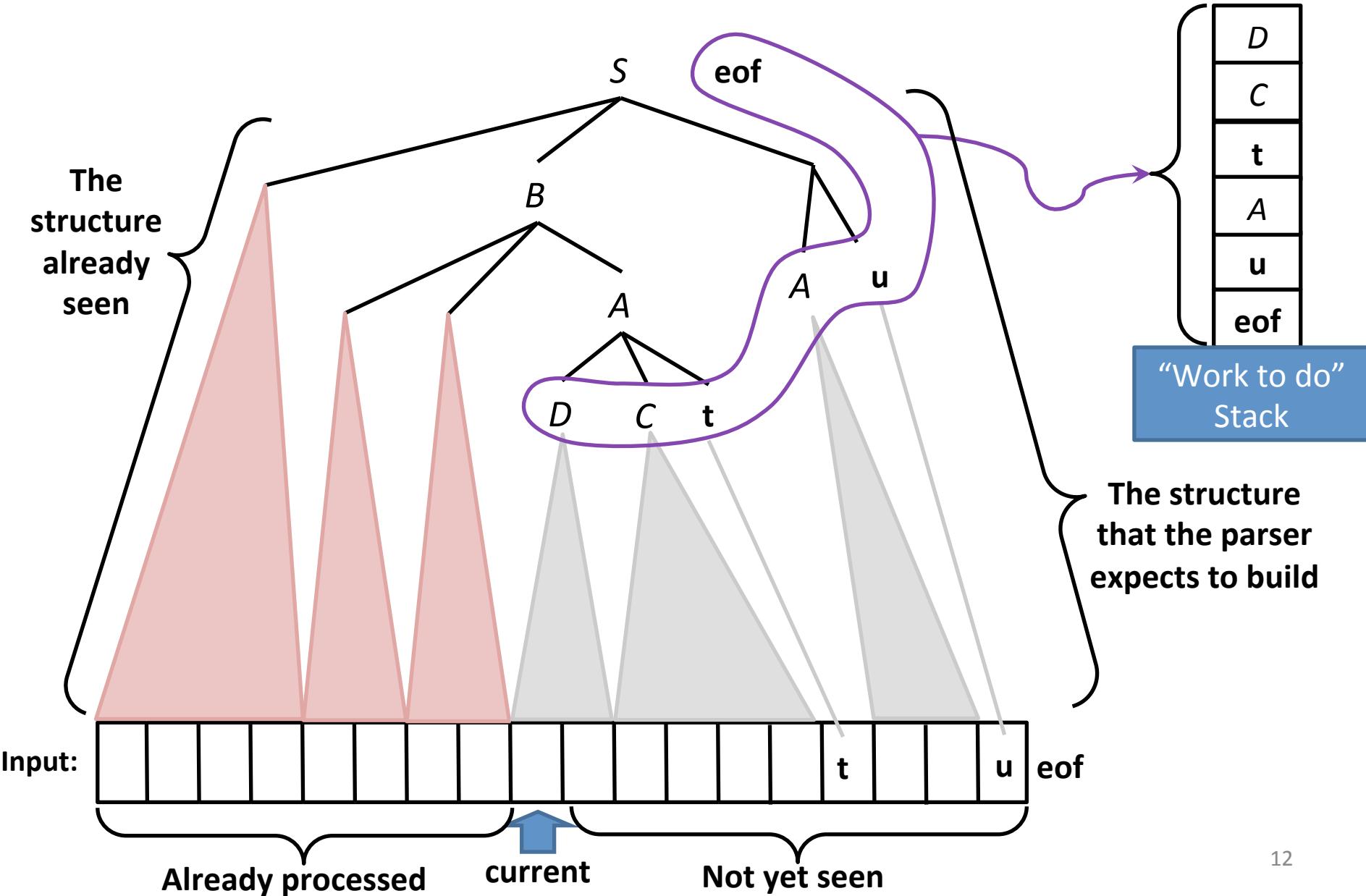
# Example

$$S \rightarrow ( S ) | \{ S \} | \epsilon$$



“Work to do”  
Stack

# A Snapshot of a Predictive Parser



# Algorithm

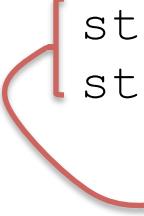
```
stack.push(eof)
stack.push(Start non-term)
t = scanner.getToken()
Repeat
    if stack.top is a terminal y
        match y with t
        pop y from the stack
        t = scanner.next_token()
    if stack.top is a nonterminal X
        get table[X,t]
        pop X from the stack
        push production's RHS (each symbol from Right to Left)
```

Until one of the following:

stack is empty  **accept**

stack.top is a terminal that doesn't match t

stack.top is a non-term and parse table entry is empty

 **reject**

# Example 2, bad input: You try

$S \rightarrow (S) \mid \{S\} \mid \epsilon$

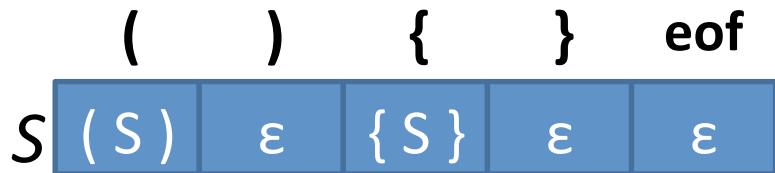
( ) { } eof  
S (S) | ε | {S} | ε | ε

INPUT

( ( } eof

# This Parser works great!

Given a single token we always knew exactly what production it started



# Two Outstanding Issues

1. How do we know if the language is LL(1)
  - Easy to imagine a Grammar where a single token is not enough to select a rule

*Any Idea?*

$$S \rightarrow ( S ) | \{ S \} | ()$$

2. How do we build the selector table?

It turns out that there is one answer to both:

If selector table has  $\leq 1$  production per cell, then grammar is LL(1)

# LL(1) Grammar Transformations

Necessary (but not sufficient) conditions for LL(1)  
Parsing:

- Free of left recursion
  - No nonterminal loops for a production
  - Why? Need to look past list to know when to cap it
- Left factored
  - No rules with common prefix
  - Why? We'd need to look past the prefix to pick rule

# Left-Recursion

Recall, a grammar such that  $A \rightarrow A\alpha$  is left recursive

A grammar is immediately left recursive if this can happen in one step:

$$A \rightarrow A\alpha \mid \beta$$

Fortunately, it's always possible to change the grammar to remove left-recursion without changing the language it recognizes

# Why Left Recursion is a Problem (Blackbox View)

CFG snippet:  $XList \rightarrow XList\ x \mid x$

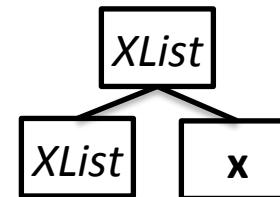
Current parse tree:  $XList$

Current token:  $x$

How should we grow the tree top-down?



**(OR)**



Correct if there are no more **xs**

Correct if there are more xs

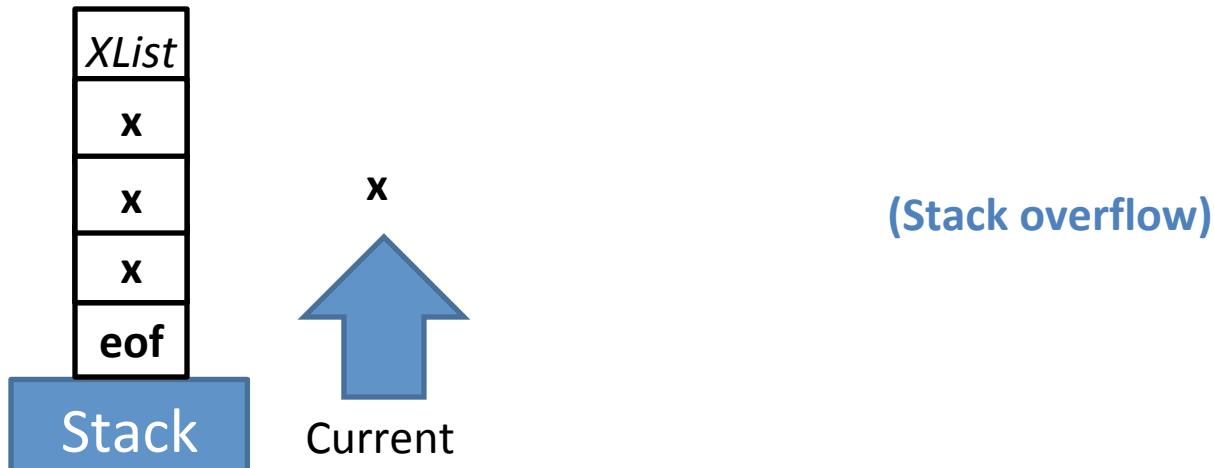
We don't know which without more lookahead

# Why Left Recursion is a Problem (Whitebox View)

CFG snippet:  $XList \rightarrow XList\ x \mid x$

Current parse tree:  $XList$        $x$        $\text{eof}$       Current token:  $x$

Parse table:       $XList$        $XList\ x$        $\epsilon$



# Removing Left-Recursion

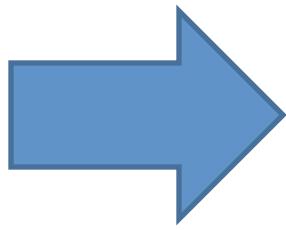
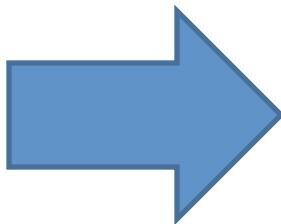
(for a single immediately left-recursive rule)

$$A \rightarrow A\alpha \mid \beta \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \\ \mid \varepsilon \end{array}$$

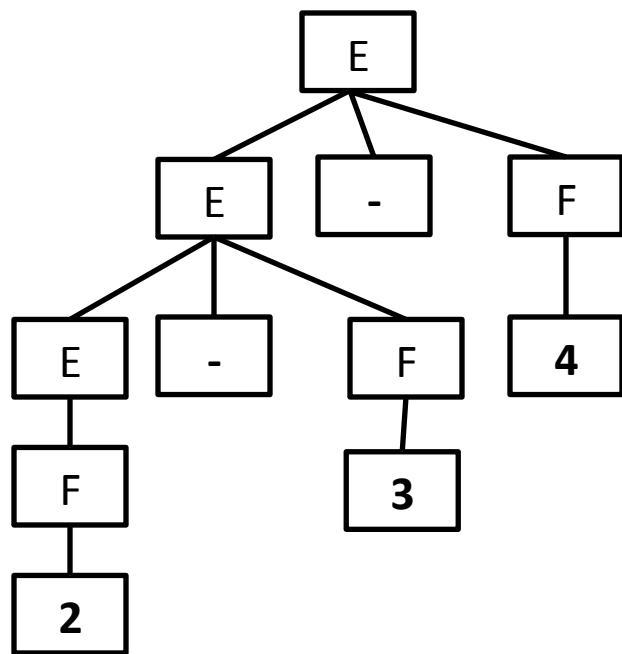
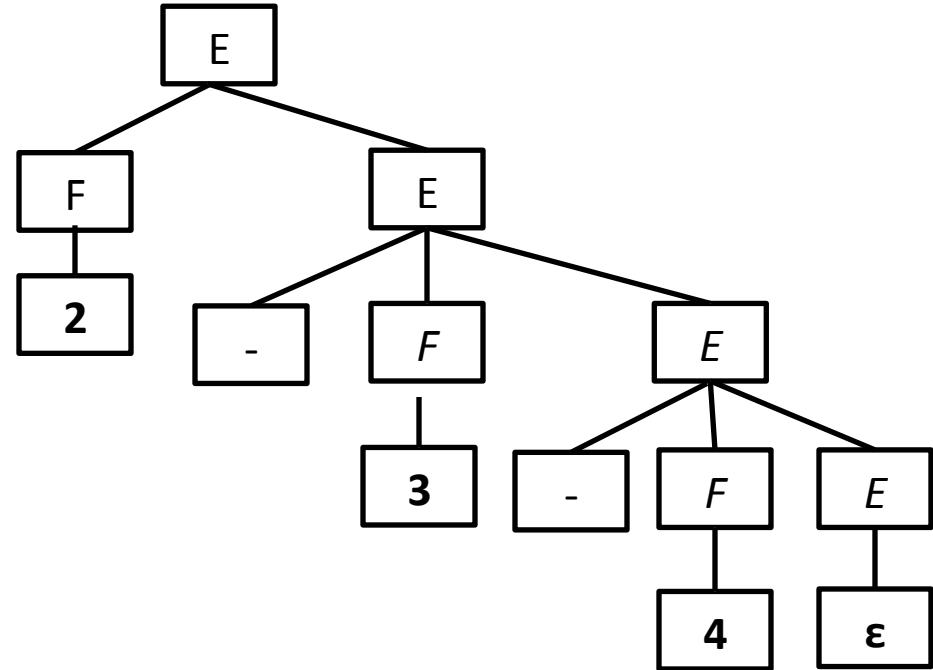


Where  $\beta$  does  
not begin with  $A$

# Example

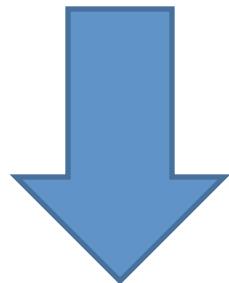
$$A \rightarrow A\alpha \mid \beta$$

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \\ &\mid \varepsilon \end{aligned}$$
$$\begin{aligned} Exp &\rightarrow Exp - Factor \\ &\mid Factor \\ Factor &\rightarrow intlit \mid ( Exp ) \end{aligned}$$

$$\begin{aligned} Exp &\rightarrow Factor Exp' \\ Exp' &\rightarrow - Factor Exp' \\ &\mid \varepsilon \\ Factor &\rightarrow intlit \mid ( Exp ) \end{aligned}$$

# Let's check in on the Parse Tree...

$$\begin{array}{lcl} \textit{Exp} & \rightarrow & \textit{Exp} - \textit{Factor} \\ & | & \textit{Factor} \end{array}$$
$$\textit{Factor} \rightarrow \text{intlit} \mid ( \textit{Exp} )$$

$$\begin{array}{lcl} \textit{Exp} & \rightarrow & \textit{Factor} \textit{Exp}' \\ \textit{Exp}' & \rightarrow & - \textit{Factor} \textit{Exp}' \end{array}$$
$$\mid \epsilon$$
$$\textit{Factor} \rightarrow \text{intlit} \mid ( \textit{Exp} )$$


... We'll fix that later

# General Rule for Removing Immediate Left-Recursion

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n \mid A \beta_1 \mid A \beta_2 \mid \dots \mid A \beta_m$$

$$\begin{aligned} A &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \\ A' &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A' \mid \varepsilon \end{aligned}$$

# Left Factored Grammars

If a nonterminal has two productions whose RHS have common prefix

→ Grammar is not left factored and not LL(1)

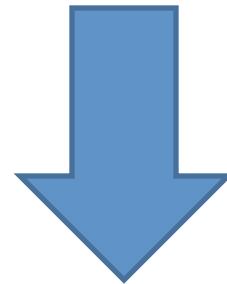
$$Exp \rightarrow ( \ Exp \ ) \mid ( )$$

**Not left factored**

# Left Factoring

Given productions of the form

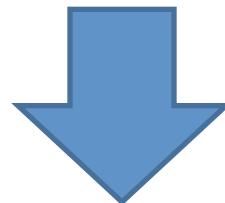
$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$$



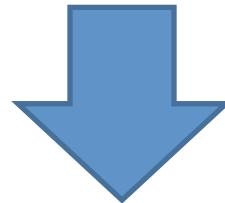
$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

# Combined Example

$$Exp \rightarrow ( Exp ) \mid Exp\ Exp \mid ()$$


Remove  
Immediate left-recursion

$$\begin{aligned}Exp &\rightarrow ( Exp ) Exp' \mid () Exp' \\Exp' &\rightarrow Exp\ Exp' \mid \epsilon\end{aligned}$$


Left-factoring

$$\begin{aligned}Exp &\rightarrow ( Exp'' \\Exp'' &\rightarrow Exp ) Exp' \mid ) Exp' \\Exp' &\rightarrow Exp\ Exp' \mid \epsilon\end{aligned}$$

# Where are we at?

We've set ourselves up for success in building the selection table

- Two things that prevent a grammar from being LL(1) were identified and avoided
  - Not Left-Factored grammars
  - Left-recursive grammars
- Next time
  - Build two data structures that combine to yield a selector table:
    - FIRST set
    - FOLLOW set

# Announcements

HW4 due today

HW5 assigned today

# Building a Predictive Parser

I.e., How to build the parse table for a  
recursive-descent parser

# Last Time: Intro LL(1) Predictive Parser

*Predict* the parse tree  
top-down

Parser structure

- 1 token of lookahead
- A stack tracking parse tree frontier
- Selector/parse table

Necessary conditions

- Left-factored
- Free of left-recursion



# Today: Building the Parse Table

Review Grammar transformations

- Why they are necessary
- How they work

Build the selector table

- $\text{FIRST}(X)$ : Set of terminals that can begin at a subtree rooted at  $X$
- $\text{FOLLOW}(X)$ : Set of terminals that can appear after  $X$

# Review of LL(1) Grammar Transformations

Necessary (but not sufficient conditions) for LL(1) parsing:

- Free of left recursion
  - “No left-recursive rules”
  - Why? Need to look past the list to know when to cap it
- Left-factored
  - “No rules with a common prefix, for any nonterminal”
  - Why? We would need to look past the prefix to pick the production

~~$A \rightarrow A \dots$~~   
if left recursion, can't make prediction

$A \rightarrow A B$

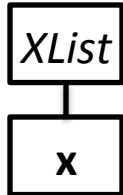
# Why Left Recursion is a Problem (Blackbox View)

CFG snippet:  $XList \rightarrow XList\ x \mid x$

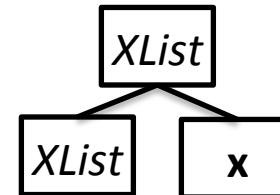
Current parse tree:  $XList$

Current token:  $x$

How should we grow the tree top-down?



(OR)



Correct if there are no more **xs**

Correct if there are more xs

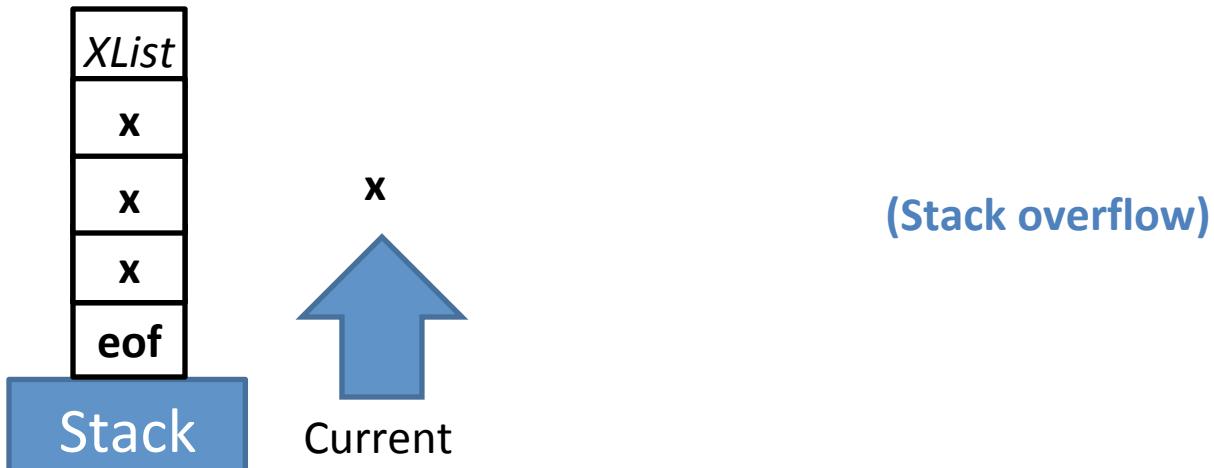
We don't know which without more lookahead

# Why Left Recursion is a Problem (Whitebox View)

CFG snippet:  $XList \rightarrow XList\ x \mid x$

Current parse tree:  $XList$        $x$        $\text{eof}$       Current token:  $x$

Parse table:       $XList$        $XList\ x$        $\epsilon$



# Left Recursion Elimination: Review

Replace

$$A \rightarrow A\alpha \mid \beta$$

With

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Head of the list

Where  $\beta$  does not start with  $A$  or may not be present

Preserve order (a list of  $\alpha$  starting with  $\beta$ ) but use right recursion

# Left Recursion Elimination: Ex1

$$A \rightarrow A\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{array}$$

$$E \rightarrow E \underbrace{\text{cross id}}_{\alpha} \mid \underbrace{\text{id}}_{\beta} \quad \Rightarrow \quad \begin{array}{c} \beta \\ | \\ E \rightarrow \mathbf{id} E' \\ E' \rightarrow \underbrace{\text{cross id } E'}_{\alpha} \mid \epsilon \end{array}$$

# Left Recursion Elimination: Ex2

$$A \rightarrow A\alpha | \beta \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | \epsilon \end{array}$$

$$\begin{array}{l} E \rightarrow E + T | T \\ T \rightarrow T * F | F \\ F \rightarrow ( E ) | \text{id} \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{l} E \rightarrow TE' \\ E' \rightarrow + TE' | \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' | \epsilon \\ F \rightarrow ( E ) | \text{id} \end{array}$$

# Left Recursion Elimination: Ex3

$$A \rightarrow A\alpha | \beta \quad \Rightarrow \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | \epsilon \end{array}$$

$$SList \rightarrow SList D | \epsilon$$

$$D \rightarrow Type \text{ id semi}$$

$$Type \rightarrow \text{bool} | \text{int}$$

$$SList \rightarrow D SList | \epsilon$$

$$D \rightarrow Type \text{ id semi}$$

$$Type \rightarrow \text{bool} | \text{int}$$

~~$$SList \rightarrow \epsilon SList'$$~~

~~$$SList' \rightarrow D SList' | \epsilon$$~~

$$D \rightarrow Type \text{ id semi}$$

$$Type \rightarrow \text{bool} | \text{int}$$

$D^*$

$S \rightarrow DS/\epsilon$

$S \rightarrow SD/\epsilon$

# Left Factoring: Review

Removing common prefix from grammar

Replace

$$A \rightarrow \boxed{\alpha \beta_1} \mid \dots \mid \alpha \beta_m \mid y_1 \mid \dots \mid y_n$$

With

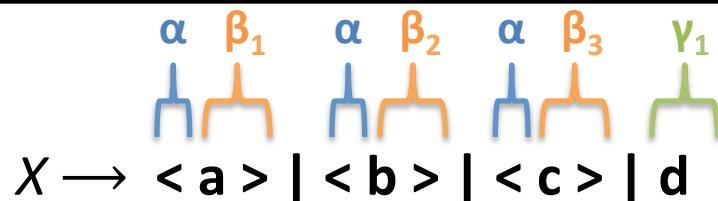
$$\begin{aligned} A &\rightarrow \boxed{\alpha A'} \mid y_1 \mid \dots \mid y_n \\ A' &\rightarrow \boxed{\beta_1} \mid \dots \mid \boxed{\beta_m} \end{aligned}$$

Where  $\beta_i$  and  $y_i$  are sequence of symbols with no common prefix  
 $y_i$  May not be present, one of the  $\beta$  may be  $\epsilon$

Squash all “problem” rules starting with  $\alpha$  together into one rule  $\alpha A'$   
Now  $A'$  represents the suffix of the “problem” rules

# Left Factoring: Example 1

$$A \rightarrow \alpha \beta_1 \mid \dots \mid \alpha \beta_m \mid y_1 \mid \dots \mid y_n \quad \xrightarrow{\hspace{1cm}} \quad A \rightarrow \alpha A' \mid y_1 \mid \dots \mid y_n \\ A' \rightarrow \beta_1 \mid \dots \mid \beta_m$$

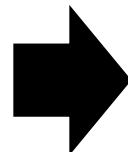


---

$$X \rightarrow <X' \mid \mathbf{d}$$
$$X' \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$$

Legend:  $\beta_1$  (orange),  $\beta_2$  (orange),  $\beta_3$  (orange),  $\gamma_1$  (green)

# Left Factoring: Example 2

$$A \rightarrow \alpha \beta_1 \mid \dots \mid \alpha \beta_m \mid y_1 \mid \dots \mid y_n$$

$$\begin{aligned} A &\rightarrow \alpha A' \mid y_1 \mid \dots \mid y_n \\ A' &\rightarrow \beta_1 \mid \dots \mid \beta_m \end{aligned}$$

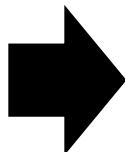
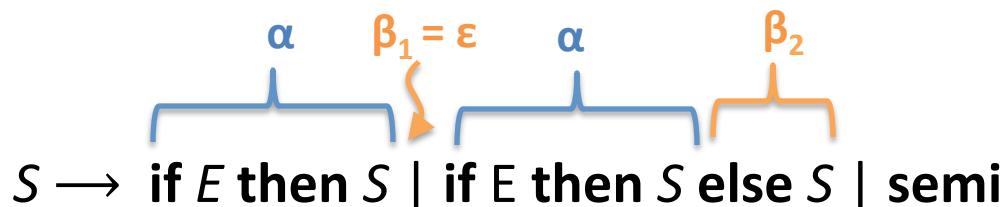
$\beta_1$                            $\beta_2$

$$\text{Stmt} \rightarrow \mathbf{id \ assign \ E} \mid \mathbf{id \ ( \ EList \ )} \mid \mathbf{return}$$
$$E \rightarrow \mathbf{intlit} \mid \mathbf{id}$$
$$Elist \rightarrow E \mid E \mathbf{comma} Elist$$

---

$$\text{Stmt} \rightarrow \mathbf{id \ Stmt'} \mid \mathbf{return}$$
$$\text{Stmt}' \rightarrow \mathbf{assign \ E} \mid \mathbf{( \ EList \ )}$$
$$E \rightarrow \mathbf{intlit} \mid \mathbf{id}$$
$$Elist \rightarrow E \mid E \mathbf{comma} Elist$$

# Left Factoring: Example 3

$$A \rightarrow \alpha \beta_1 \mid \dots \mid \alpha \beta_m \mid y_1 \mid \dots \mid y_n$$

$$\begin{aligned} A &\rightarrow \alpha A' \mid y_1 \mid \dots \mid y_n \\ A' &\rightarrow \beta_1 \mid \dots \mid \beta_m \end{aligned}$$


$E \rightarrow \text{boollit}$

---

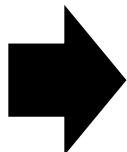
$S \rightarrow \text{if } E \text{ then } S S' \mid \text{semi}$

$S' \rightarrow \text{else } S \mid \epsilon$

$E \rightarrow \text{boollit}$

# Left Factoring: Not Always Immediate

$$A \rightarrow \alpha \beta_1 \mid \dots \mid \alpha \beta_m \mid y_1 \mid \dots \mid y_n$$



$$\begin{aligned} A &\rightarrow \alpha A' \mid y_1 \mid \dots \mid y_n \\ A' &\rightarrow \beta_1 \mid \dots \mid \beta_m \end{aligned}$$

This snippet yearns for left-factoring

$$\begin{aligned} S &\rightarrow A \mid C \mid \text{return} \\ A &\rightarrow \text{id assign } E \\ C &\rightarrow \text{id ( } E\text{List) } \end{aligned}$$

but we cannot! At least without *Inlining*

$$S \rightarrow \text{id assign } E \mid \text{id ( } E\text{List) } \mid \text{return}$$

看起来没问题，写成一行后发现可以进行left factoring

left factoring

$$S \rightarrow \text{id } X \mid \text{return}$$

$$X \rightarrow \text{assign } E \mid C \ldots$$

# Let's be more constructive

So far, we've only talked about what precludes us from building a predictive parser

It's time to actually build the parse table

# Building the Parse Table

What do we actually need to ensure arbitrary production  $A \rightarrow \alpha$  is the correct one to apply?

Assume  $\alpha$  is an arbitrary sequence of symbols.

1. What terminals could  $\alpha$  possibly start with  
→ we call this the FIRST set
2. What terminal could possibly come after  $A$   
→ we call this the FOLLOW set

# Why is FIRST Important?

Assume the top-of-stack symbol is  $A$  and current token is **a**

- Production 1:  $A \rightarrow \alpha$
- Production 2:  $A \rightarrow \beta$

FIRST lets us disambiguate:

- If **a** is in  $\text{FIRST}(\alpha)$ , we know Production 1 is a viable choice
- If **a** is in  $\text{FIRST}(\beta)$ , we know Production 2 is a viable choice
- If **a** is in only one of  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$ , we can predict the production we need

# FIRST Sets

$\text{FIRST}(\alpha)$  is the set of terminals that begin the strings derivable from  $\alpha$ , and also, if  $\alpha$  can derive  $\epsilon$ , then  $\epsilon$  is in  $\text{FIRST}(\alpha)$ .

Formally, let's write it together

$\text{FIRST}(\alpha) =$

$\text{FIRST}(S) = F(AB)$  since  $F(A)$  is epsilon,  $\text{FIRST}(S) = F(B)$

$$S \rightarrow A \ B$$

$$F(S) = F(AB) = F(A) \cup F(B)$$

$$A \rightarrow \epsilon$$

$$\text{First}(A) = \{\epsilon\}$$

$$B \rightarrow ID \mid \epsilon$$

$$\text{First}(B) = \{ID, \epsilon\}$$

$$S \rightarrow A \ B$$

$$A \rightarrow ( \mid \epsilon$$

$$B \rightarrow ID \mid \epsilon$$

$$F(S) = \{C, ID, \epsilon\}$$

$$F(S) = F(AB) = F(A) \cup F(B)$$

$$\text{First}(A) = \{C, \epsilon\}$$

$$\text{First}(B) = \{ID, \epsilon\}$$

# FIRST Sets

$\text{FIRST}(\alpha)$  is the set of terminals that begin the strings derivable from  $\alpha$ , and also, if  $\alpha$  can derive  $\epsilon$ , then  $\epsilon$  is in  $\text{FIRST}(\alpha)$ .

Formally, let's write it together

$$\text{FIRST}(\alpha) = \{t | (t \in \Sigma \wedge \alpha \Rightarrow^* t\beta) \vee (t = \epsilon \wedge \alpha \Rightarrow^* \epsilon)\}$$

terminal

Terminal  $\alpha$  can  
start with

$\alpha \Rightarrow^* \epsilon$

# FIRST Construction: Single Symbol

We begin by doing FIRST sets for a single, arbitrary symbol X

$$\text{FIRST}(a) = \{a\}$$

– If X is a terminal:  $\text{FIRST}(X) = \{ X \}$

$$\text{FIRST}(AB) = \text{FIRST}(A) \cup \text{FIRST}(B)$$

– If X is  $\epsilon$ :  $\text{FIRST}(\epsilon) = \{ \epsilon \}$

– If X is a nonterminal, for each  $X \rightarrow Y_1 Y_2 \dots Y_k$

- Put  $\text{FIRST}(Y_1) - \{\epsilon\}$  into  $\text{FIRST}(X)$
- If  $\epsilon$  is in  $\text{FIRST}(Y_1)$ , put  $\text{FIRST}(Y_2) - \{\epsilon\}$  into  $\text{FIRST}(X)$
- If  $\epsilon$  is also in  $\text{FIRST}(Y_2)$ , put  $\text{FIRST}(Y_3) - \{\epsilon\}$  into  $\text{FIRST}(X)$
- ...
- If  $\epsilon$  is in  $\text{FIRST}$  of all  $Y_i$  symbols, put  $\epsilon$  into  $\text{FIRST}(X)$

Repeat this step until there are no changes to any nonterminal's FIRST set

# $\text{FIRST}(X)$ Example

Building  $\text{FIRST}(X)$  for nonterm X

for each  $X \rightarrow Y_1 Y_2 \dots Y_k$

- Add  $\text{FIRST}(Y_1) - \{\epsilon\}$
- If  $\epsilon$  is in  $\text{FIRST}(Y_{1 \text{ to } i-1})$ : add  $\text{FIRST}(Y_i) - \{\epsilon\}$
- If  $\epsilon$  is in all RHS symbols, add  $\epsilon$

$$Exp \rightarrow Term \ Exp'$$

$$Exp' \rightarrow \mathbf{minus} \ Term \ Exp' \mid \epsilon$$

$$Term \rightarrow Factor \ Term'$$

$$Term' \rightarrow \mathbf{divide} \ Factor \ Term' \mid \epsilon$$

$$Factor \rightarrow \mathbf{intlit} \mid \mathbf{lparens} \ Exp \ rparens$$

$$\text{FIRST}(Factor) = \{ \mathbf{intlit}, \mathbf{lparens} \}$$

$$\text{FIRST}(Term') = \{ \mathbf{divide}, \epsilon \}$$

$$\text{FIRST}(Term) = \{ \mathbf{intlit}, \mathbf{lparens} \}$$

$$\text{FIRST}(Exp') = \{ \mathbf{minus}, \epsilon \}$$

$$\text{FIRST}(Exp) = \{ \mathbf{intlit}, \mathbf{lparens} \}$$

# FIRST( $\alpha$ )

We now extend FIRST to strings of symbols  $\alpha$

- We want to define FIRST for all RHS

Looks very similar to the procedure for single symbols

Let  $\alpha = Y_1 Y_2 \dots Y_k$

- Put  $\text{FIRST}(Y_1) - \{\varepsilon\}$  in  $\text{FIRST}(\alpha)$ 
  - If  $\varepsilon$  is in  $\text{FIRST}(Y_1)$ : add  $\text{FIRST}(Y_2) - \{\varepsilon\}$  to  $\text{FIRST}(\alpha)$
  - If  $\varepsilon$  is in  $\text{FIRST}(Y_2)$ : add  $\text{FIRST}(Y_3) - \{\varepsilon\}$  to  $\text{FIRST}(\alpha)$
  - ...
  - If  $\varepsilon$  is in  $\text{FIRST}$  of all  $Y_i$  symbols, put  $\varepsilon$  into  $\text{FIRST}(\alpha)$

# Building FIRST( $\alpha$ ) from FIRST(X)

## Building FIRST(X) for nonterm X

for each  $X \rightarrow Y_1 Y_2 \dots Y_k$

- Add  $\text{FIRST}(Y_1) - \{\epsilon\}$
- If  $\epsilon$  is in  $\text{FIRST}(Y_{1 \text{ to } i-1})$ : add  $\text{FIRST}(Y_i) - \{\epsilon\}$
- If  $\epsilon$  is in all RHS symbols, add  $\epsilon$

## Building FIRST( $\alpha$ )

Let  $\alpha = Y_1 Y_2 \dots Y_k$

- Add  $\text{FIRST}(Y_1) - \{\epsilon\}$
- If  $\epsilon$  is in  $\text{FIRST}(Y_{1 \text{ to } i-1})$ : add  $\text{FIRST}(Y_i) - \{\epsilon\}$
- If  $\epsilon$  is in all RHS symbols, add  $\epsilon$

# FIRST( $\alpha$ ) Example

## Building FIRST( $\alpha$ )

Let  $\alpha = Y_1 Y_2 \dots Y_k$

- Add  $\text{FIRST}(Y_1) - \{\epsilon\}$
- If  $\epsilon$  is in  $\text{FIRST}(Y_{1 \text{ to } i-1})$ : add  $\text{FIRST}(Y_i) - \{\epsilon\}$
- If  $\epsilon$  is in all RHS symbols, add  $\epsilon$

$E \rightarrow TX$   
 $X \rightarrow +TX \mid \epsilon$   
 $T \rightarrow FY$   
 $Y \rightarrow *FY \mid \epsilon$   
 $F \rightarrow (E) \mid id$

$\text{FIRST}(E) = \{(, id\}$   
 $\text{FIRST}(T) = \{(, id\}$   
 $\text{FIRST}(F) = \{(, id\}$   
 $\text{FIRST}(X) = \{+, \epsilon\}$   
 $\text{FIRST}(Y) = \{*, \epsilon\}$

$\text{FIRST}(TX) = \{(, id\}$   
 $\text{FIRST}(+TX) = \{+\}$   
 $\text{FIRST}(FY) = \{ (, id \}$   
 $\text{FIRST}(*FY) = \{ *\}$   
 $\text{FIRST}( ( E ) ) = \{ ( \}$   
 $\text{FIRST}( id ) = \{ id \}$

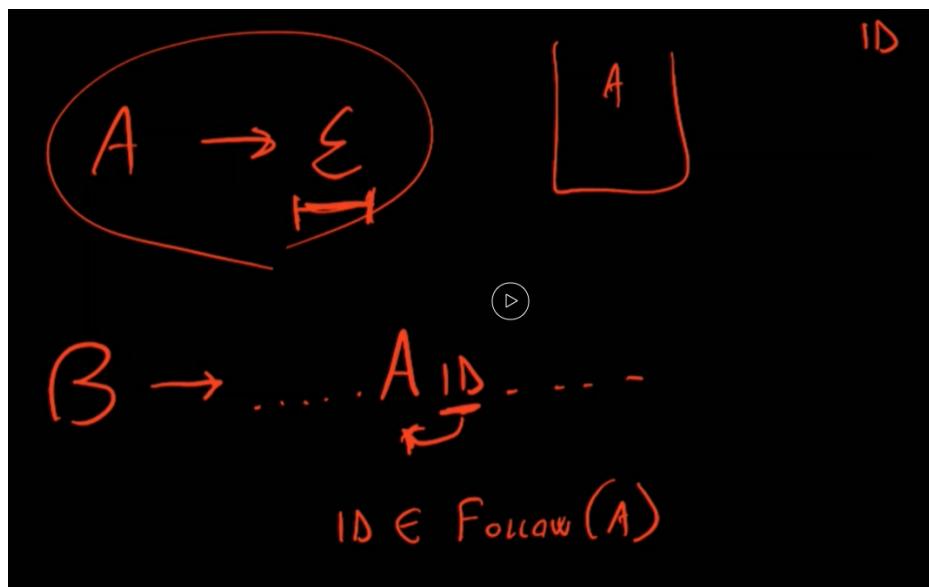
$\text{FIRST}(xF) = \{+, (, id\}$

x还有一个可能是epsilon, 所以要考虑F的情况

$\text{FIRST}(xy) = \{+, *, \epsilon\}$

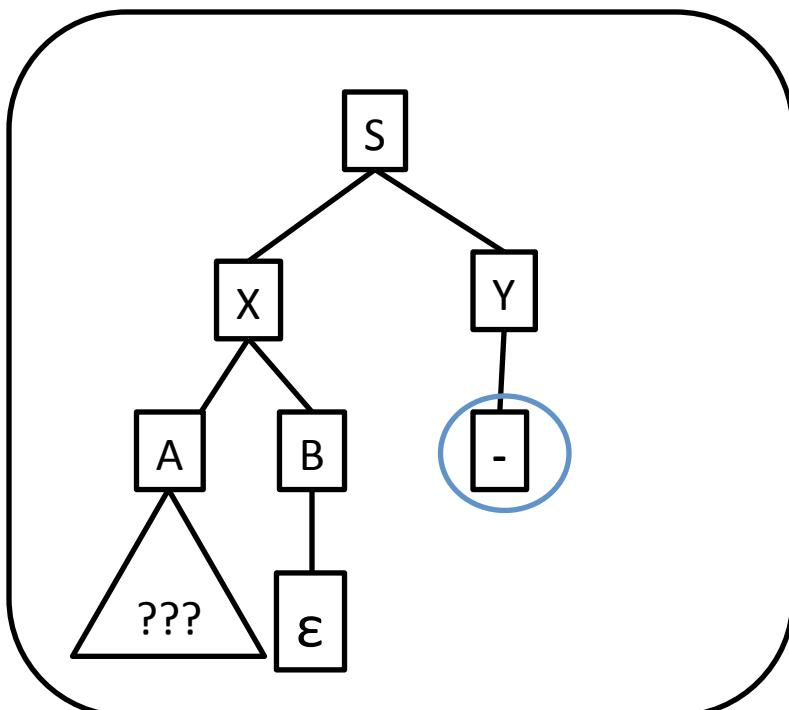
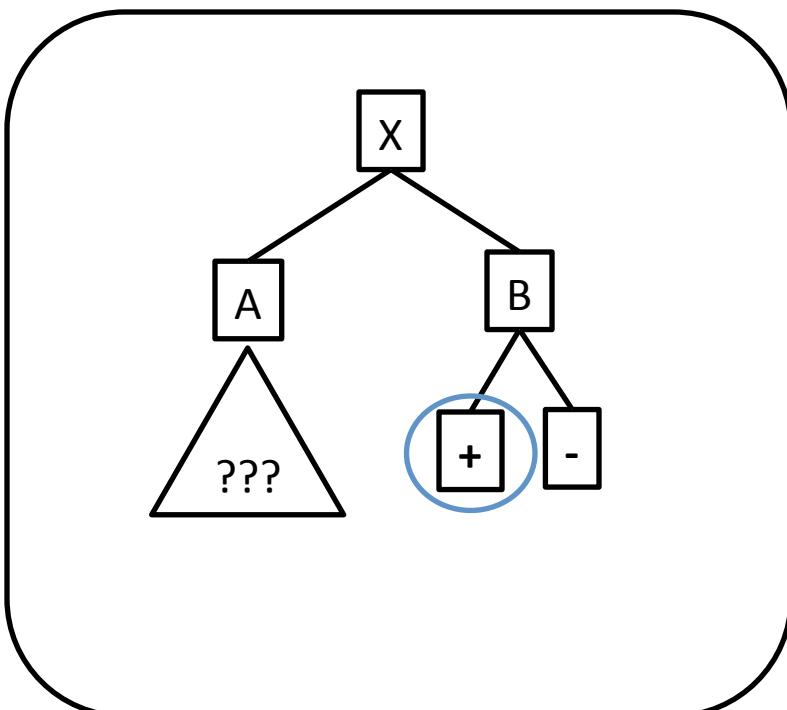
FIRST sets alone do not provide enough information to construct a parse table

If a rule R can derive  $\epsilon$ , we need to know what terminals can come just after R



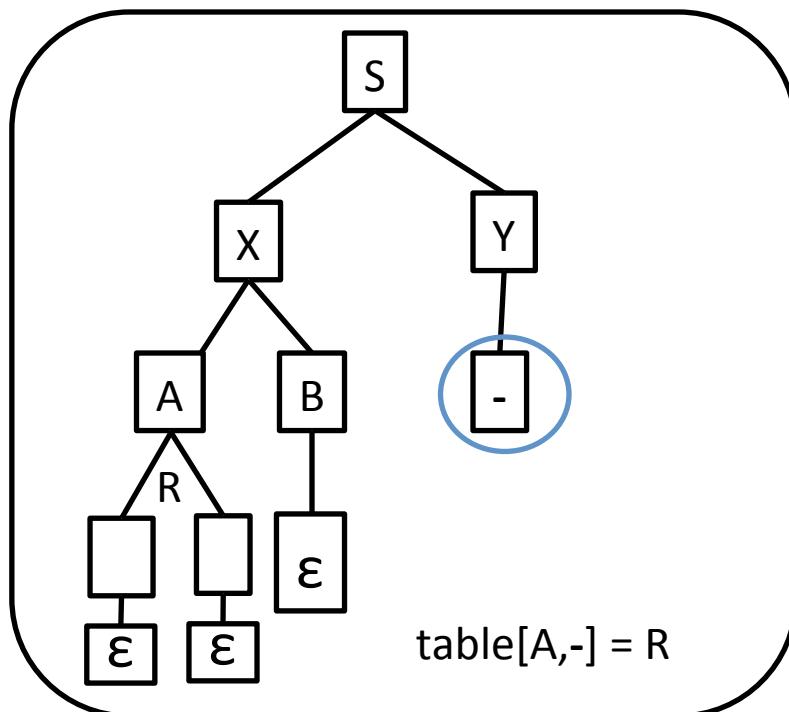
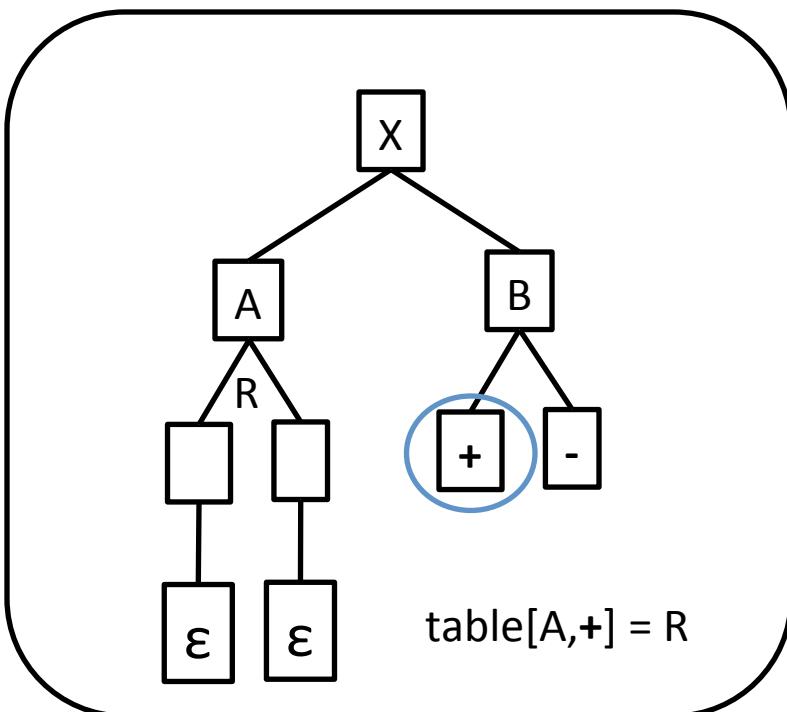
# FOLLOW Sets: Pictorially

For nonterminal A, FOLLOW(A) is the set of terminals that can appear immediately to the right of A



# FOLLOW Sets: Pictorially

For nonterminal A, FOLLOW(A) is the set of terminals that can appear immediately to the right of A



# FOLLOW Sets

For nonterminal A, FOLLOW(A) is the set of terminals that can appear immediately to the right of A

Let's write it together,

$\text{FOLLOW}(A) =$

# FOLLOW Sets

For nonterminal A, FOLLOW(A) is the set of terminals that can appear immediately to the right of A

Let's write it together,

$\text{FOLLOW}(A) =$

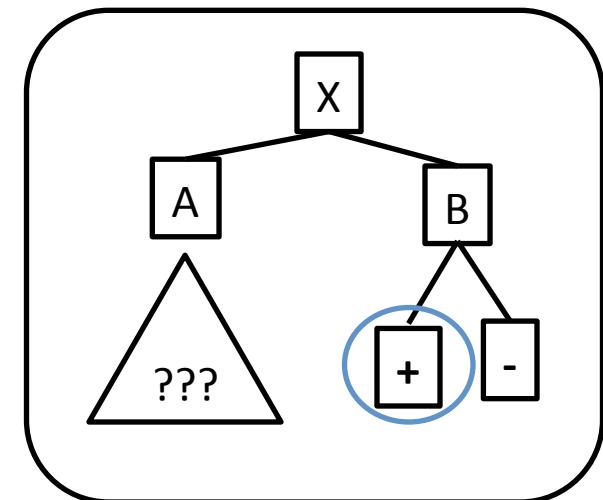
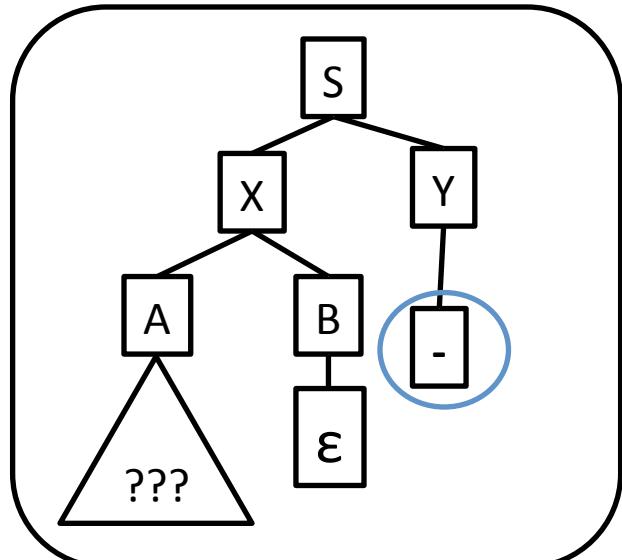
$$\{t \mid (t \in \Sigma \wedge S \Rightarrow t+ \alpha At\beta) \vee (t = EOF \wedge S \Rightarrow t* \alpha A)\}$$

# FOLLOW Sets: Construction

To build FOLLOW(A)

- If A is the start nonterminal, add **eof**  
**Where  $\alpha, \beta$  may be empty**
- For rules  $X \rightarrow \alpha A \beta$ 
  - Add FIRST( $\beta$ ) –  $\{\epsilon\}$
  - If  $\epsilon$  is in FIRST( $\beta$ ) or  $\beta$  is empty, add FOLLOW(X)

Continue building FOLLOW sets until reach a fixed point (i.e., no more symbols can be added)



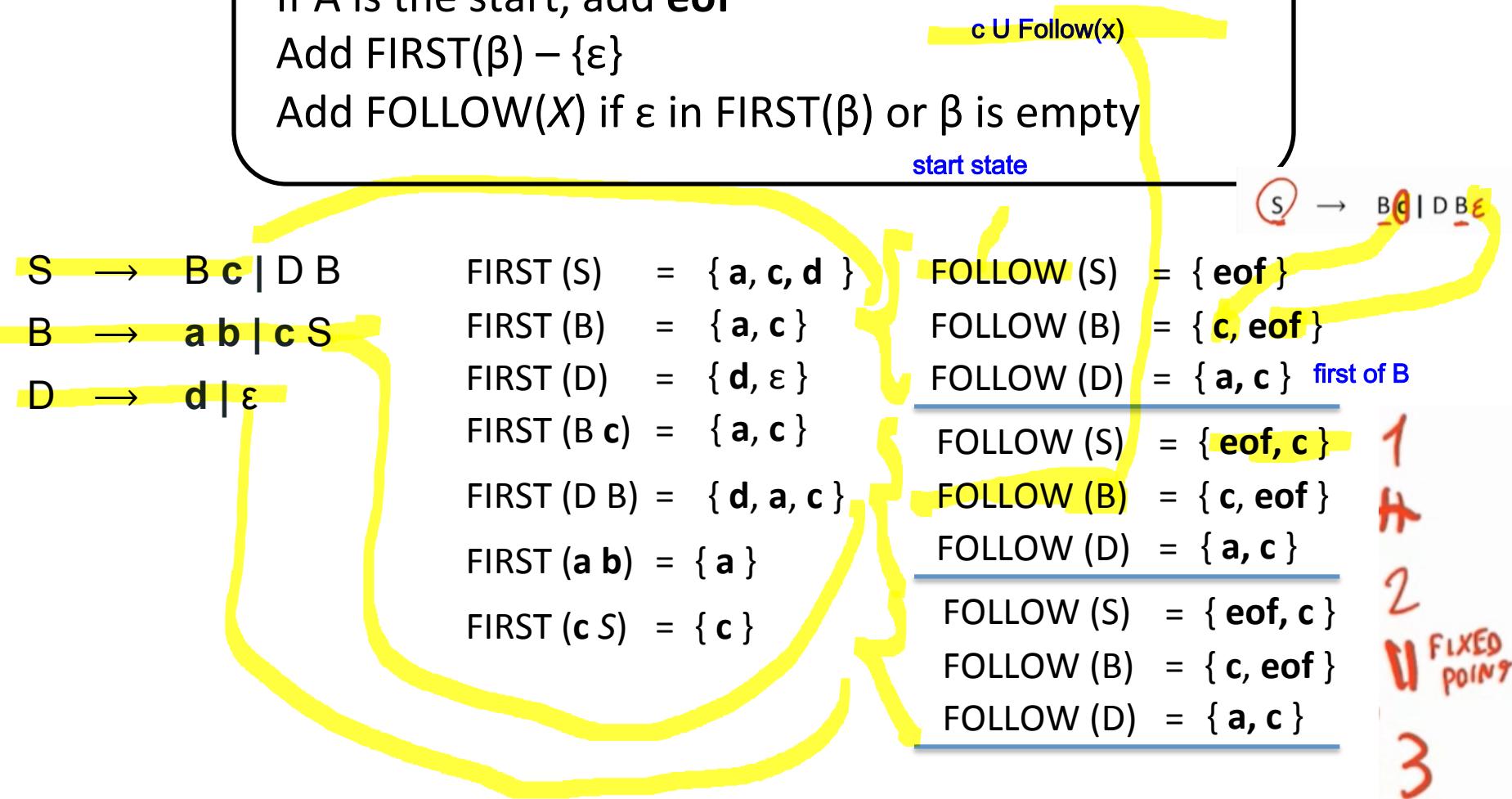
# FOLLOW Sets Example

FOLLOW(A) for  $X \rightarrow \alpha A \beta$

If A is the start, add **eof**

Add FIRST( $\beta$ ) – { $\epsilon$ }

Add FOLLOW(X) if  $\epsilon$  in FIRST( $\beta$ ) or  $\beta$  is empty



# Building the Parse Table

```
for each production  $X \rightarrow \alpha$  {  
    for each terminal  $t$  in FIRST( $\alpha$ ) {  
        put  $\alpha$  in Table[X] [ $t$ ]  
    }  
    if  $\epsilon$  is in FIRST( $\alpha$ ) {  
        for each terminal  $t$  in FOLLOW( $X$ ) {  
            put  $\alpha$  in Table[X] [ $t$ ]  
        }  
    }  
}
```

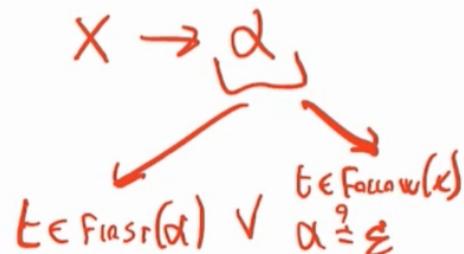


Table collision  $\Leftrightarrow$  Grammar is not in LL(1)

# Putting it all together

Build FIRST sets for each nonterminal

Build FIRST sets for each production's RHS

Build FOLLOW sets for each nonterminal

Use FIRST and FOLLOW to fill parse table for each production

# Tips n' Tricks

## FIRST sets

- Only contain alphabet terminals and  $\epsilon$  defined for everything
- Defined for arbitrary RHS and nonterminals
- Constructed by starting at the beginning of a production

## FOLLOW sets

- Only contain alphabet terminals and  $\text{eof}$
- Defined for nonterminals only
- Constructed by jumping into production

FIRST( $\alpha$ ) for  $\alpha = Y_1 Y_2 \dots Y_k$

Add FIRST( $Y_1$ ) - { $\epsilon$ }

If  $\epsilon$  is in FIRST( $Y_{1 \text{ to } i-1}$ ): add FIRST( $Y_i$ ) - { $\epsilon$ }

If  $\epsilon$  is in all RHS symbols, add  $\epsilon$

FOLLOW(A) for  $X \rightarrow \alpha A \beta$

If A is the start, add **eof**

Add FIRST( $\beta$ ) - { $\epsilon$ }

Add FOLLOW( $X$ ) if  $\epsilon$  in FIRST( $\beta$ ) or  $\beta$  empty

### Table[X][t]

```
for each production  $X \rightarrow \alpha$ 
  for each terminal  $t$  in FIRST( $\alpha$ )
    put  $\alpha$  in Table[X] [t]
  if  $\epsilon$  is in FIRST( $\alpha$ ) {
    for each terminal  $t$  in FOLLOW( $X$ ) {
      put  $\alpha$  in Table[X] [t]
```

$$\text{FIRST}(S) = \{a, c, d\} \quad \text{B, D}\backslash \text{epsilon}$$

$$\text{FIRST}(B) = \{a, c\}$$

$$\text{FIRST}(D) = \{d, \epsilon\}$$

$$\text{FIRST}(Bc) = \{a, c\}$$

$$\text{FIRST}(DB) = \{d, a, c\}$$

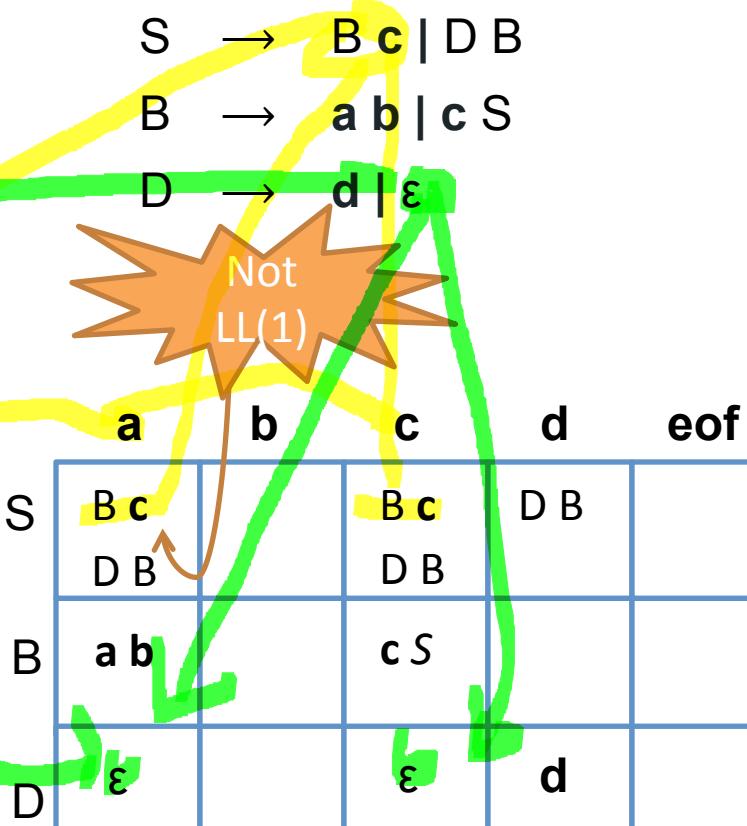
$$\text{FIRST}(ab) = \{a\}$$

$$\text{FIRST}(cS) = \{c\}$$

$$\text{FIRST}(d) = \{d\}$$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

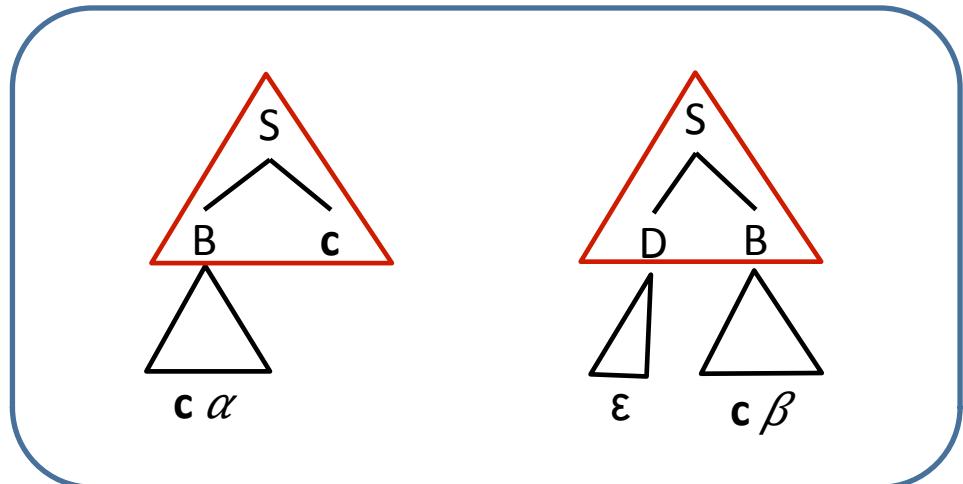
### CFG



# Why is a Table Collision a Problem?

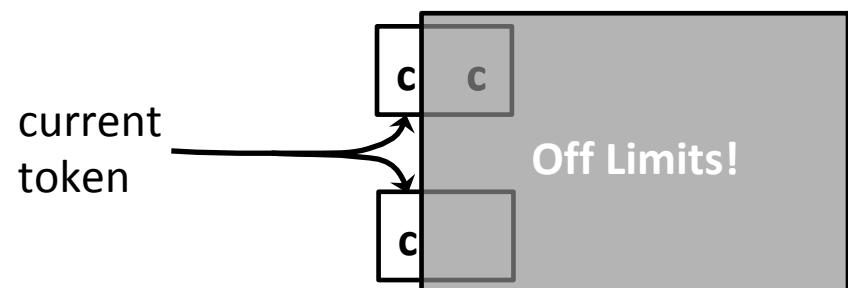
CFG

$$\begin{array}{l} S \rightarrow Bc \mid DB \\ B \rightarrow ab \mid cS \\ D \rightarrow d \mid \epsilon \end{array}$$



	a	b	c	d	
S	Bc		Bc	DB	
	DB		DB		
B	ab		cS		
D	ε		ε	d	

current  
token



# Recap

**FIRST** and **FOLLOW** sets define the parse table

If the grammar is LL(1), the table is unambiguous

- i.e., each cell has at most one entry

If the grammar is not LL(1) we can attempt a transformation sequence:

1. Remove left recursion
2. Left-factoring

**Next time:** Grammar transformations affect the structure of the parse tree. How does this affect syntax-directed translation (in particular, parse tree AST)?

# Announcement: Midterm Prep

Midterm is Tuesday, 3/10 here

List of topics

- Up to and including lecture on Tuesday's the 5th

Length 1hr 10min

No extra material allowed – just bring a pen

Sample midterm available online

- Recommended that you do this by Thursday
- We'll review it in class

# Syntax Directed Translation for Top-Down Parsing

# Last Time: Built LL(1) Predictive Parser

FIRST and FOLLOW sets define the parse table

If the grammar is LL(1), the table is unambiguous

- i.e., each cell has at most one entry

If the grammar is not LL(1) we can attempt a transformation sequence:

1. Remove left recursion
2. Left-factoring

Grammar transformations affect the structure of the parse tree. How does this affect syntax-directed translation (in particular, parse tree  $\rightarrow$  AST)?

# Today

Review Parse Table Construction

- 2 examples

Show how to do Syntax-Directed Translation  
using an LL(1) parser

FIRST( $\alpha$ ) for  $\alpha = Y_1 Y_2 \dots Y_k$

Add FIRST( $Y_1$ ) -  $\{\epsilon\}$

If  $\epsilon$  is in FIRST( $Y_{1 \text{ to } i-1}$ ): add FIRST( $Y_i$ ) -  $\{\epsilon\}$

If  $\epsilon$  is in all RHS symbols, add  $\epsilon$

FOLLOW(A) for  $X \rightarrow \alpha A \beta$

If A is the start, add **eof**

Add FIRST( $\beta$ ) -  $\{\epsilon\}$

Add FOLLOW( $X$ ) if  $\epsilon$  in FIRST( $\beta$ ) or  $\beta$  empty

### Table[X][t]

```

for each production  $X \rightarrow \alpha$ 
  for each terminal  $t$  in FIRST( $\alpha$ )
    put  $\alpha$  in Table[X] [ $t$ ]
  if  $\epsilon$  is in FIRST( $\alpha$ ) {
    for each terminal  $t$  in FOLLOW( $X$ ) {
      put  $\alpha$  in Table[X] [ $t$ ]
    }
  }

```

$$\text{FIRST}(S) = \{a, c, d\}$$

$$\text{FIRST}(B) = \{a, c\}$$

$$\text{FIRST}(D) = \{d, \epsilon\}$$

$$\text{FOLLOW}(S) = \{\text{eof}, c\}$$

$$\text{FIRST}(Bc) = \{a, c\}$$

$$\text{FOLLOW}(B) = \{c, \text{eof}\}$$

$$\text{FIRST}(DB) = \{d, a, c\}$$

$$\text{FOLLOW}(D) = \{a, c\}$$

$$\text{FIRST}(ab) = \{a\}$$

$$\text{FIRST}(cS) = \{c\}$$

$$\text{FIRST}(d) = \{d\}$$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

### CFG

$$S \rightarrow Bc \mid DB$$

$$B \rightarrow ab \mid cS$$

$$D \rightarrow d \mid \epsilon$$

$\text{FIRST}(D) \subseteq$

$\{\text{EOF}\} \cup \text{FIRST}(B)$

$\text{FIRST}(B)$



a b c d eof

S	B c DB		B c DB	DB	
B	ab		cS		
D	$\epsilon$		$\epsilon$	d	

FIRST( $\alpha$ ) for  $\alpha = Y_1 Y_2 \dots Y_k$

Add FIRST( $Y_1$ ) -  $\{\epsilon\}$

If  $\epsilon$  is in FIRST( $Y_{1 \text{ to } i-1}$ ): add FIRST( $Y_i$ ) -  $\{\epsilon\}$

If  $\epsilon$  is in all RHS symbols, add  $\epsilon$

FOLLOW(A) for  $X \rightarrow \alpha A \beta$

If A is the start, add **eof**

Add FIRST( $\beta$ ) -  $\{\epsilon\}$

Add FOLLOW( $X$ ) if  $\epsilon$  in FIRST( $\beta$ ) or  $\beta$  empty

Table[X][t]

```

for each production  $X \rightarrow \alpha$ 
  for each terminal  $t$  in FIRST( $\alpha$ )
    put  $\alpha$  in Table[X] [ $t$ ]
  if  $\epsilon$  is in FIRST( $\alpha$ ) {
    for each terminal  $t$  in FOLLOW( $X$ ) {
      put  $\alpha$  in Table[X] [ $t$ ]
  
```

CFG

$S \rightarrow ( S ) | \{ S \} | \epsilon$

$$\text{FIRST}(S) = \{\{, (, \epsilon\}$$

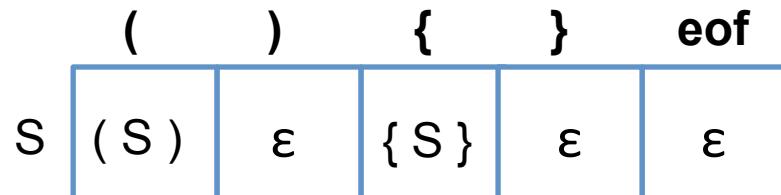
$$\text{FIRST}((S)) = \{( ( \}$$

$$\text{FIRST}(\{S\}) = \{\{ \}$$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

$$\text{FOLLOW}(S) = \{ \text{eof}, ), \} \}$$

start state always contains eof, and  
(S, {S} give )



FIRST( $\alpha$ ) for  $\alpha = Y_1 Y_2 \dots Y_k$

Add FIRST( $Y_1$ ) -  $\{\epsilon\}$

If  $\epsilon$  is in FIRST( $Y_{1 \text{ to } i-1}$ ): add FIRST( $Y_i$ ) -  $\{\epsilon\}$

If  $\epsilon$  is in all RHS symbols, add  $\epsilon$

FOLLOW(A) for  $X \rightarrow \alpha A \beta$

If A is the start, add **eof**

Add FIRST( $\beta$ ) -  $\{\epsilon\}$

Add FOLLOW( $X$ ) if  $\epsilon$  in FIRST( $\beta$ ) or  $\beta$  empty

Table[X][t]

```

for each production  $X \rightarrow \alpha$ 
  for each terminal  $t$  in FIRST( $\alpha$ )
    put  $\alpha$  in Table[X] [ $t$ ]
  if  $\epsilon$  is in FIRST( $\alpha$ ) {
    for each terminal  $t$  in FOLLOW( $X$ ) {
      put  $\alpha$  in Table[X] [ $t$ ]
    }
  }

```

$$\text{FIRST}(S) = \{+, \epsilon\}$$

$$\text{FIRST}(+S) = \{+\}$$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

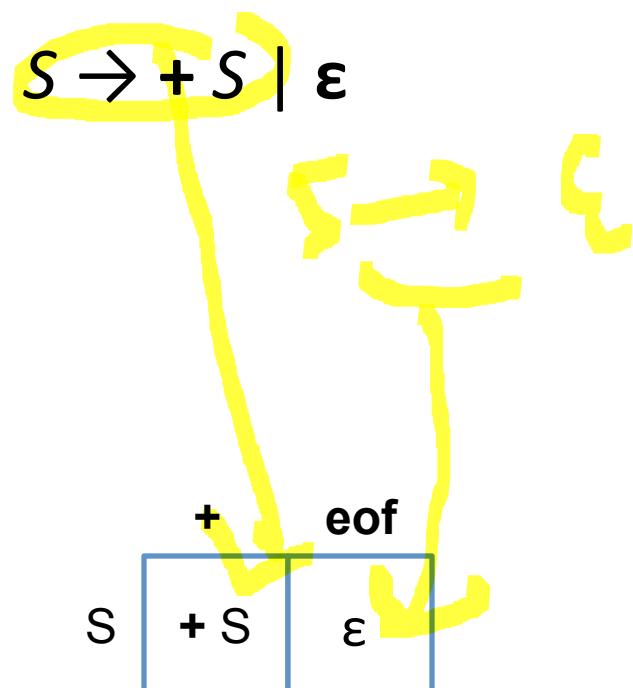
$$\text{FOLLOW}(S) = \{\text{eof}\}$$

$$\text{first}(S) = \text{first}(+S) \cup \text{first}(\epsilon)$$

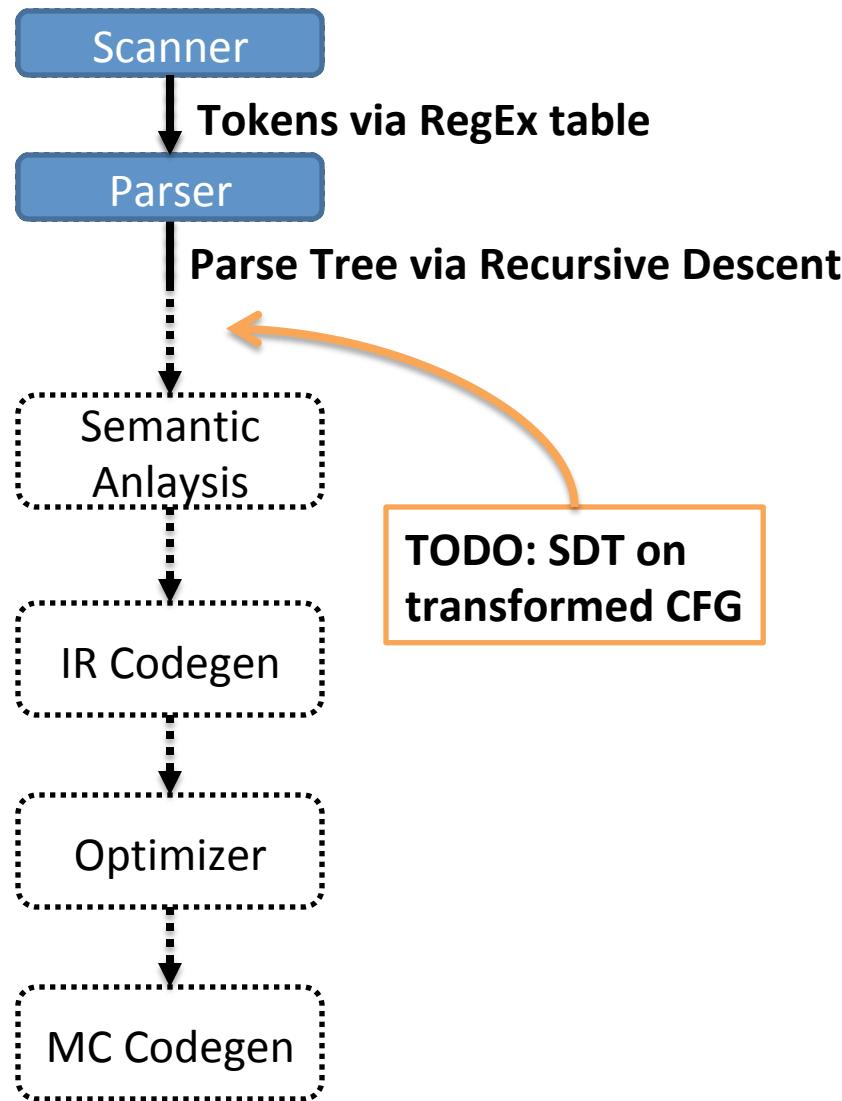
first(x), first( every elt in alpha)

follow set only contains terminals and eof!

CFG



# How's that Compiler Looking?



# Implementing SDT for LL(1) Parser

So far, SDT shown as second (bottom-up) pass over parse tree

The LL(1) parser never needed to explicitly build the parse tree (implicitly tracked via stack)

Naïve approach: build the parse tree explicitly

# Semantic Stack

Instead of building the parse tree, give parser second, *semantic* stack

- Holds nonterminals' translations
- SDT rules converted to
- Pop translations of RHS nonterm on
- Push computed translation of LHS nonterm on

Translation goal:

- Count the number of occurrences of matched pairs of rounded parens: “( ... )”
- Ignore occurrences of matched pairs of square brackets: “[ ... ]”

<u>CFG</u>	<u>SDT Rules</u>	<u>SDT Actions</u>
$\text{Expr} \rightarrow \epsilon$	$\text{Expr.trans} = 0$	push 0
( Expr )	$\text{Expr.trans} = \text{Expr}_2.\text{trans} + 1$	$\text{Expr}_2.\text{trans} = \text{pop}; \text{push } \text{Expr}_2.\text{trans} + 1$
[ Expr ]	$\text{Expr.trans} = \text{Expr}_2.\text{trans}$	$\text{Expr}_2.\text{trans} = \text{pop}; \text{push } \text{Expr}_2.\text{trans}$

Bottom Up  
parse tree

# Action Numbers

Need to define *when* to fire the SDT Action

- Not immediately obvious since SDT is bottom-up

Solution

- Number actions and put them on the symbol stack!
- Add action number symbols at end of the productions

## CFG

$$\begin{aligned} Expr \rightarrow & \varepsilon \ #1 \\ | \quad ( \text{Expr} ) \ #2 \\ | \quad [ \text{Expr} ] \ #3 \end{aligned}$$

## SDT Actions

- #1 push 0
- #2  $\text{Expr}_2.\text{trans} = \text{pop}; \text{push } \text{Expr}_2.\text{trans} + 1$
- #3  $\text{Expr}_2.\text{trans} = \text{pop}; \text{push } \text{Expr}_2.\text{trans}$

# Action Numbers: Example 1

## CFG

$$\begin{aligned} Expr \rightarrow & \varepsilon \ #1 \\ | & ( Expr ) \ #2 \\ | & [ Expr ] \ #3 \end{aligned}$$

## SDT Actions: Counting Max Paren Depth

- #1 push 0
- #2 Expr<sub>2</sub>.trans = pop; push(Expr<sub>2</sub>.trans + 1)
- #3 Expr<sub>2</sub>.trans = pop; push(Expr<sub>2</sub>.trans)

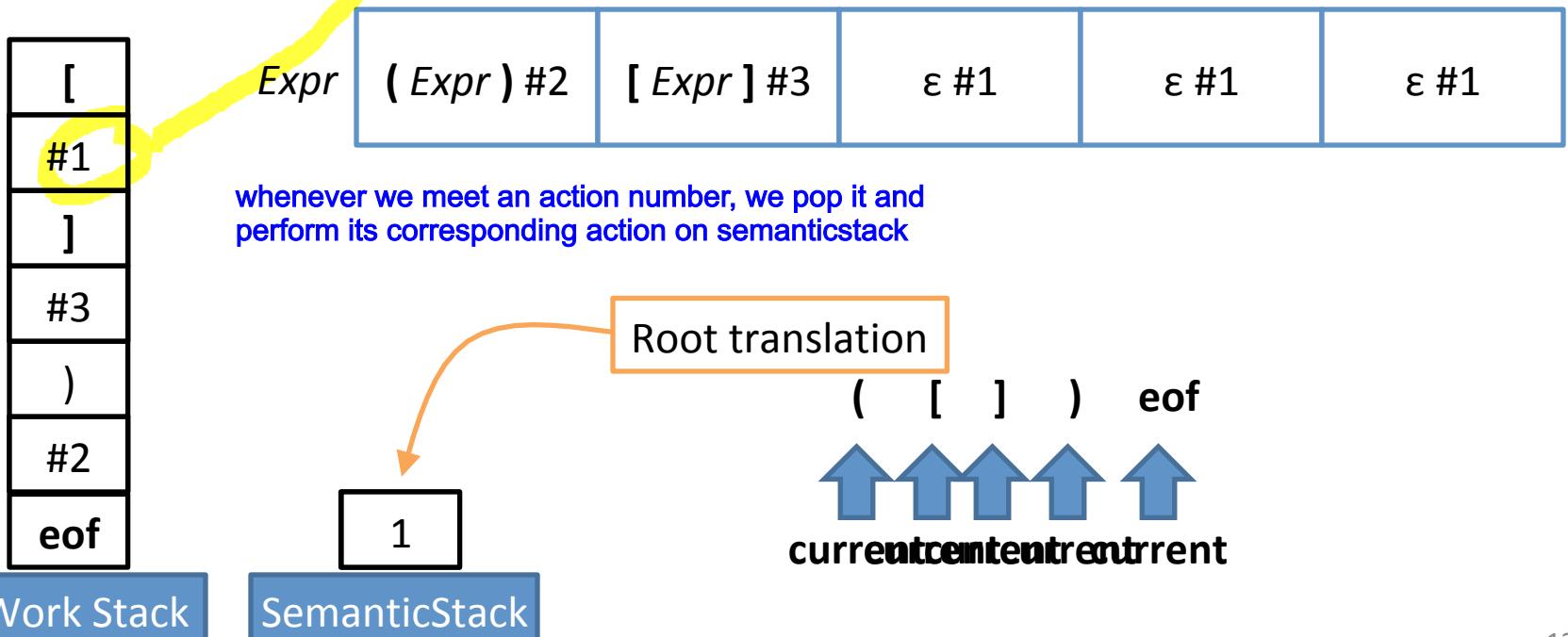
(

[

]

)

EOF



# No-op SDT Actions

## CFG

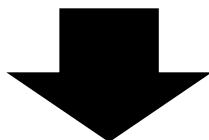
$$\begin{aligned} Expr \rightarrow & \varepsilon \ #1 \\ | & ( Expr ) \ #2 \\ | & [ Expr ] \ #3 \end{aligned}$$

## SDT Actions: Counting Max Paren Depth

#1 push 0  
#2  $Expr_2.trans = pop; push(Expr_2.trans + 1)$   
#3  $Expr_2.trans = pop; push(Expr_2.trans)$

this is "peephole optimization"

Useless rule



## CFG

$$\begin{aligned} Expr \rightarrow & \varepsilon \ #1 \\ | & ( Expr ) \ #2 \\ | & [ Expr ] \end{aligned}$$

## SDT Actions: Counting Max Paren Depth

#1 push 0  
#2  $Expr_2.trans = pop; push(Expr_2.trans + 1)$

# Placing Action Numbers

Action numbers go after their corresponding nonterminals, before their corresponding terminal

Translations popped right to left in action

## CFG

*Expr* → *Expr + Term* #1

|  
*Term*

*Term* → *Term \* Factor*#2

|  
*Factor*

*Factor* → #3 **intlit**

## SDT Actions

#1 tTrans = pop ; eTrans = pop ; push(tTrans + eTrans)

#2 fTrans = pop; tTrans = pop ; push(fTrans \* tTrans)

#3 push(**intlit.value**)

# Placing Action Numbers: Example

Write SDT Actions and place action numbers to get the product of a *ValList* (i.e. multiply all elements)

## CFG

$$List \rightarrow Val\ List'\ #1$$

$$List' \rightarrow Val\ List'\ #2$$

$$| \quad \epsilon \ #3$$

$$Val \rightarrow \#4\ intlit$$

1. push val
2. push list
3. reach action#1
4. pop list
5. pop val

we push left to right, but  
we pop right to left

ltrans, vtrans correspond to val and list  
respectively

## SDT Actions

- #1 LTrans = pop ; vTrans = pop ; push(LTrans \* vTrans)
- #2 LTrans = pop; vTrans = pop ; push(LTrans \* vTrans)
- #3 push(1)
- #4 push(intlit.value)

first, put their corresponding action number;  
then, write SDT actions

# Action Numbers: Benefits

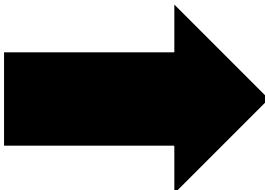
Plans SDT actions using the work stack

**Robust** to previously introduced grammar transformations

## CFG

$$\begin{array}{l} Expr \rightarrow Expr + Term \#1 \\ \quad | \quad Term \\ Term \rightarrow Term * Factor \#2 \\ \quad | \quad Factor \\ Factor \rightarrow \#3 intlit \end{array}$$

can maintain the previous grammar structure easily with action number, so we call it robust


$$\begin{array}{l} Expr \rightarrow Term Expr' \\ Expr' \rightarrow + Term \#1 Expr' \\ \quad | \quad \epsilon \\ Term \rightarrow Factor Term' \\ Term' \rightarrow * Factor \#2 Term' \\ \quad | \quad \epsilon \\ Factor \rightarrow \#3 intlit \end{array}$$

## SDT Actions

- #1 tTrans = pop ; eTrans = pop ; push(tTrans + eTrans)
- #2 fTrans = pop; tTrans = pop ; push(fTrans \* tTrans)
- #3 push(intlit.value)

# Example: SDT on Transformed Grammar

## CFG

*Expr* → *Term Expr'*

*Expr'* → + *Term #1 Expr'*  
|  $\epsilon$

*Term* → *Factor Term'*

*Term'* → \* *Factor #2 Term'*  
|  $\epsilon$

*Factor* → #3 **intlit**  
| ( *Expr* )

## SDT Actions

- #1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
- #2 fTrans = pop; tTrans = pop ; push(tTrans \* fTrans)
- #3 push(**intlit.value**)

# Example: SDT on Transformed Grammar

## CFG

$\text{Expr} \rightarrow \text{Term Expr}'$   
 $\text{Expr}' \rightarrow + \text{Term} \#1 \text{Expr}'$   
 $\quad | \quad \epsilon$   
 $\text{Term} \rightarrow \text{Factor Term}'$   
 $\text{Term}' \rightarrow * \text{Factor} \#2 \text{Term}'$   
 $\quad | \quad \epsilon$   
 $\text{Factor} \rightarrow \#3 \text{intlit}$   
 $\quad | \quad ( \text{Expr} )$

## SDT

#1    tTr  
...  
...

$\text{First}(\text{Factor}) = \{ \text{intlit}, ( \ } )$   
 $\text{First}(\text{Term}') = \{ *, \ }$   
 $\text{First}(\text{Term}) = \{ \text{intlit}, ( \ } )$   
 $\text{First}(\text{Expr}') = \{ +, \ }$   
 $\text{First}(\text{Expr}) = \{ \text{intlit}, ( \ } )$

$\text{First}(\text{Term Expr}') = \{ \text{intlit}, ( \ } )$   
 $\text{First}(+ \text{Term} \#1 \text{Expr}') = \{ + \}$   
 $\text{First}() = \{ \ }$   
 $\text{First}(\text{Factor Term}') = \{ \text{intlit}, ( \ } )$   
 $\text{First}(* \text{Factor} \#2 \text{Term}) = \{ * \}$   
 $\text{First}() = \{ \ }$   
 $\text{First}(\#3 \text{intlit}) = \{ \text{intlit} \}$   
 $\text{First}( ( \text{Expr} ) ) = \{ ( \ } )$

$\text{Follow}(\text{Expr}) = \{ \text{eof}, ) \ }$   
 $\text{Follow}(\text{Expr}') = \{ \text{eof}, ) \ }$   
 $\text{Follow}(\text{Term}) = \{ +, \text{eof}, ) \ }$   
 $\text{Follow}(\text{Term}') = \{ +, \text{eof}, ) \ }$   
 $\text{Follow}(\text{Factor}) = \{ *, +, \text{eof}, ) \ }$

$\text{trans} + \text{tTrans}$   
 $\text{ans} * \text{fTrans}$

# Example: SDT on Transformed Grammar

## CFG

$$\begin{aligned}
 Expr &\rightarrow Term \ Expr' \\
 Expr' &\rightarrow + \ Term \ #1 \ Expr' \\
 &\quad | \quad \epsilon \\
 Term &\rightarrow Factor \ Term' \\
 Term' &\rightarrow * \ Factor \ #2 \ Term' \\
 &\quad | \quad \epsilon \\
 Factor &\rightarrow \#3 \ intlit \\
 &\quad | \quad ( \ Expr \ )
 \end{aligned}$$

## SDT Actions

- #1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
- #2 fTrans = pop; tTrans = pop ; push(tTrans \* fTrans)
- #3 push(intlit.value)

	+	*	(	)	intlit	eof
<i>Expr</i>				<i>Term Expr'</i>		<i>Term Expr'</i>
<i>Expr'</i>	+ <i>Term #1 Expr'</i>				$\epsilon$	$\epsilon$
<i>Term</i>				<i>Factor Term'</i>		<i>Factor Term'</i>
<i>Term'</i>	$\epsilon$	* <i>Factor #2 Term'</i>			$\epsilon$	$\epsilon$
<i>Factor</i>			( <i>Expr</i> )		#3 intlit	

## CFG

$Expr \rightarrow Term\ Expr'$   
 $Expr' \rightarrow +\ Term\ #1\ Expr'$   
 $\quad | \quad \epsilon$   
 $Term \rightarrow Factor\ Term'$   
 $Term' \rightarrow *\ Factor\ #2\ Term'$   
 $\quad | \quad \epsilon$   
 $Factor \rightarrow \#3\ intlit$   
 $\quad | \quad ( Expr )$

we first need to build the table using CFG together with FIRST and FOLLOW set rules; after that, we don't need to care about CFG

## SDT Actions

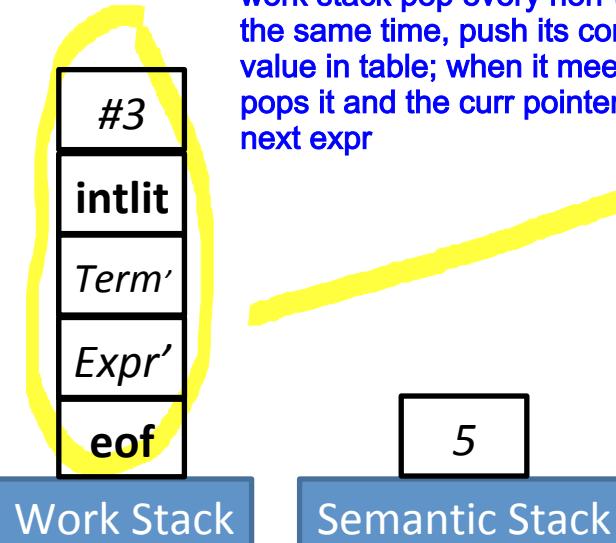
- #1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
- #2 fTrans = pop; tTrans = pop ; push(tTrans \* fTrans)
- #3 push(intlit.value)

	+	*	(	)	intlit	eof
<i>Expr</i>					<i>Term Expr'</i>	
<i>Expr'</i>	+ Term #1 Expr'				$\epsilon$	$\epsilon$
<i>Term</i>					<i>Factor Term'</i>	
<i>Term'</i>	$\epsilon$	* Factor #2 Term'			$\epsilon$	$\epsilon$
<i>Factor</i>			( Expr )		#3 intlit	

work stack pop every non-terminal and at the same time, push its corresponding value in table; when it meets terminal, it pops it and the curr pointer move to the next expr

e.g., when we at 5, 5 is intlit and expr, so we push term expr'; pop term, find table value <term, intlit=5>, push factor term'; pop factor, find table value<term, intlit=5>, push #3 intlit;

Input: 5 + 3 \* 2 eof



then, we pop #3, call action #3, push(intlit=5) to semantic stack; then, stack top is intlit, a terminal, so we pop it and move the pointer to the next expr, i.e., "+"

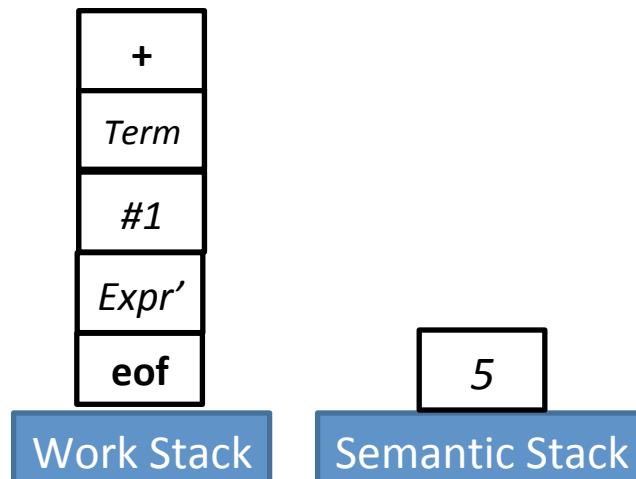
## CFG

$Expr \rightarrow Term\ Expr'$   
 $Expr' \rightarrow +\ Term\ #1\ Expr'$   
 $\quad | \quad \epsilon$   
 $Term \rightarrow Factor\ Term'$   
 $Term' \rightarrow *\ Factor\ #2\ Term'$   
 $\quad | \quad \epsilon$   
 $Factor \rightarrow \#3\ intlit$   
 $\quad | \quad ( Expr )$

## SDT Actions

- #1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
- #2 fTrans = pop; tTrans = pop ; push(tTrans \* fTrans)
- #3 push(intlit.value)

	+	*	(	)	intlit	eof
<i>Expr</i>				<i>Term Expr'</i>		<i>Term Expr'</i>
<i>Expr'</i>	+ <i>Term #1 Expr'</i>				$\epsilon$	$\epsilon$
<i>Term</i>				<i>Factor Term'</i>		<i>Factor Term'</i>
<i>Term'</i>	$\epsilon$	* <i>Factor #2 Term'</i>			$\epsilon$	$\epsilon$
<i>Factor</i>			( <i>Expr</i> )		#3 intlit	



Input: 5 + 3 \* 2 eof  
  
**concurrent**

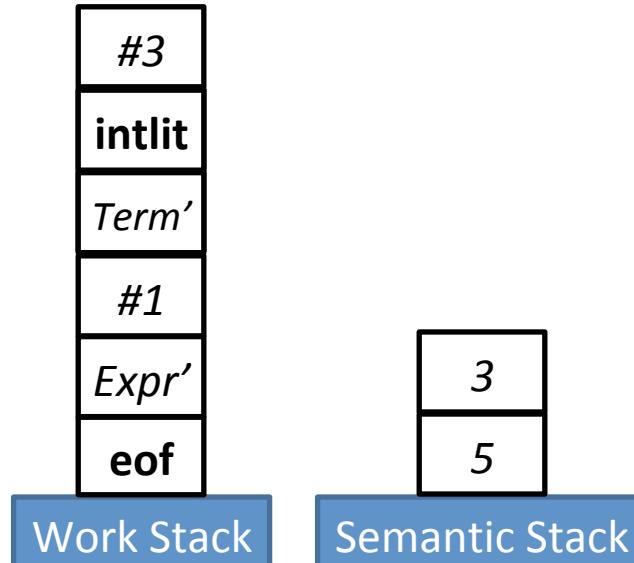
## CFG

$Expr \rightarrow Term\ Expr'$   
 $Expr' \rightarrow +\ Term\ #1\ Expr'$   
 $\quad | \quad \epsilon$   
 $Term \rightarrow Factor\ Term'$   
 $Term' \rightarrow *\ Factor\ #2\ Term'$   
 $\quad | \quad \epsilon$   
 $Factor \rightarrow \#3\ intlit$   
 $\quad | \quad ( Expr )$

## SDT Actions

- #1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
- #2 fTrans = pop; tTrans = pop ; push(tTrans \* fTrans)
- #3 push(intlit.value)

	+	*	(	)	intlit	eof
<i>Expr</i>				<i>Term Expr'</i>		<i>Term Expr'</i>
<i>Expr'</i>	+ <i>Term #1 Expr'</i>				$\epsilon$	$\epsilon$
<i>Term</i>				<i>Factor Term'</i>		<i>Factor Term'</i>
<i>Term'</i>	$\epsilon$	* <i>Factor #2 Term'</i>			$\epsilon$	$\epsilon$
<i>Factor</i>			( <i>Expr</i> )		#3 intlit	



Input: 5 + 3 \* 2 eof  
  
**current**

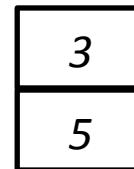
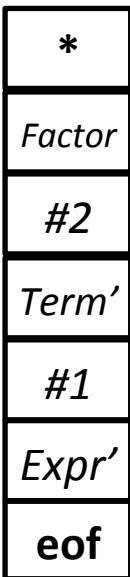
## CFG

$Expr \rightarrow Term\ Expr'$   
 $Expr' \rightarrow +\ Term\ #1\ Expr'$   
 $\quad | \quad \epsilon$   
 $Term \rightarrow Factor\ Term'$   
 $Term' \rightarrow *\ Factor\ #2\ Term'$   
 $\quad | \quad \epsilon$   
 $Factor \rightarrow \#3\ intlit$   
 $\quad | \quad ( Expr )$

## SDT Actions

- #1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
- #2 fTrans = pop; tTrans = pop ; push(tTrans \* fTrans)
- #3 push(intlit.value)

	+	*	(	)	intlit	eof
<i>Expr</i>				<i>Term Expr'</i>		<i>Term Expr'</i>
<i>Expr'</i>	+ <i>Term #1 Expr'</i>				$\epsilon$	$\epsilon$
<i>Term</i>				<i>Factor Term'</i>		<i>Factor Term'</i>
<i>Term'</i>	$\epsilon$	* <i>Factor #2 Term'</i>			$\epsilon$	$\epsilon$
<i>Factor</i>			( <i>Expr</i> )		#3 intlit	



Input: 5 + 3 \* 2 eof



Work Stack

Semantic Stack

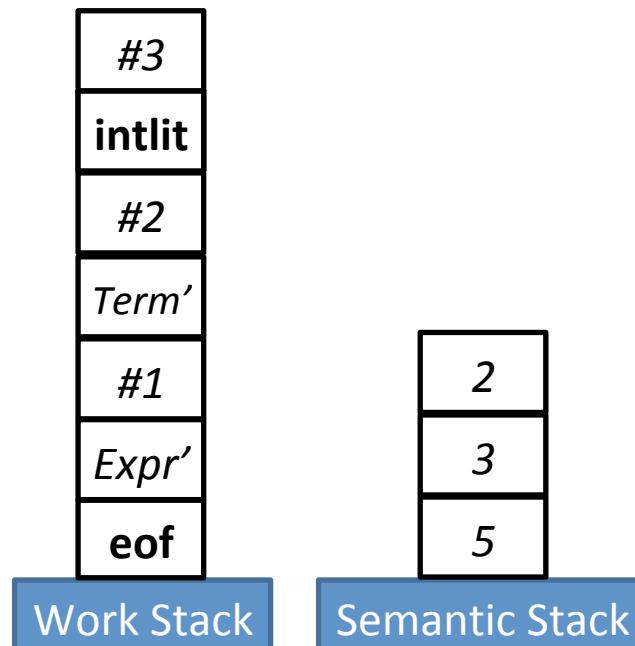
## CFG

$Expr \rightarrow Term\ Expr'$   
 $Expr' \rightarrow +\ Term\ #1\ Expr'$   
 $\quad | \quad \epsilon$   
 $Term \rightarrow Factor\ Term'$   
 $Term' \rightarrow *\ Factor\ #2\ Term'$   
 $\quad | \quad \epsilon$   
 $Factor \rightarrow \#3\ intlit$   
 $\quad | \quad ( Expr )$

## SDT Actions

- #1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
- #2 fTrans = pop; tTrans = pop ; push(tTrans \* fTrans)
- #3 push(intlit.value)

	+	*	(	)	intlit	eof
<i>Expr</i>				<i>Term Expr'</i>		<i>Term Expr'</i>
<i>Expr'</i>	+ <i>Term #1 Expr'</i>				$\epsilon$	$\epsilon$
<i>Term</i>				<i>Factor Term'</i>		<i>Factor Term'</i>
<i>Term'</i>	$\epsilon$	* <i>Factor #2 Term'</i>			$\epsilon$	$\epsilon$
<i>Factor</i>			( <i>Expr</i> )		#3 intlit	



Input: 5 + 3 \* 2 eof  
  
**current**

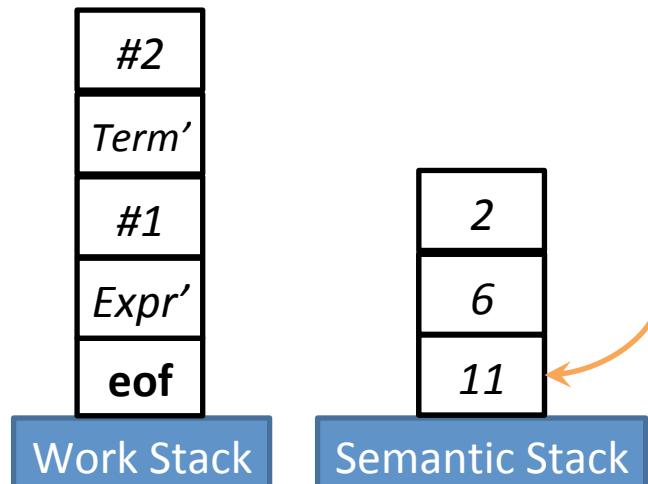
## CFG

$Expr \rightarrow Term\ Expr'$   
 $Expr' \rightarrow +\ Term\ #1\ Expr'$   
 $\quad | \quad \epsilon$   
 $Term \rightarrow Factor\ Term'$   
 $Term' \rightarrow *\ Factor\ #2\ Term'$   
 $\quad | \quad \epsilon$   
 $Factor \rightarrow \#3\ intlit$   
 $\quad | \quad ( Expr )$

## SDT Actions

- #1 tTrans = pop ; eTrans = pop ; push(eTrans + tTrans)
- #2 fTrans = pop; tTrans = pop ; push(tTrans \* fTrans)
- #3 push(intlit.value)

	+	*	(	)	intlit	eof
<i>Expr</i>				<i>Term Expr'</i>		<i>Term Expr'</i>
<i>Expr'</i>	+ Term #1 <i>Expr'</i>				$\epsilon$	$\epsilon$
<i>Term</i>				<i>Factor Term'</i>		<i>Factor Term'</i>
<i>Term'</i>	$\epsilon$	* Factor #2 <i>Term'</i>			$\epsilon$	$\epsilon$
<i>Factor</i>			( <i>Expr</i> )		#3 intlit	



Input: 5 + 3 \* 2 eof  
  
**current**

# What about ASTs?

Push and pop AST nodes on the stack

Keep field references to nodes that we pop

## CFG

$$\begin{array}{l} \textit{Expr} \rightarrow \textit{Expr} + \textit{Term} \ #1 \\ \quad | \quad \textit{Term} \\ \textit{Term} \rightarrow \#2 \text{ intlit} \end{array}$$

## Transformed CFG

$$\begin{array}{l} \textit{Expr} \rightarrow \textit{Term} \textit{Expr}' \\ \textit{Expr}' \rightarrow + \textit{Term} \ #1 \textit{Expr}' \\ \quad | \quad \varepsilon \\ \textit{Term} \rightarrow \#2 \text{ intlit} \end{array}$$

## “Evaluation” SDT Actions

- #1 tTrans = pop ;  
eTrans = pop ;  
push(eTrans + tTrans)
- #2 push(intlit.value)

## “AST” SDT Actions

- #1 tTrans = pop ;  
eTrans = pop ;  
push(new PlusNode(tTrans, eTrans))
- #2 push(new IntLitNode(intlit.value))

# AST Example

## Transformed CFG

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + T \#1 E' \\ | & \quad \epsilon \\ T &\rightarrow \#2 \text{ intlit} \end{aligned}$$

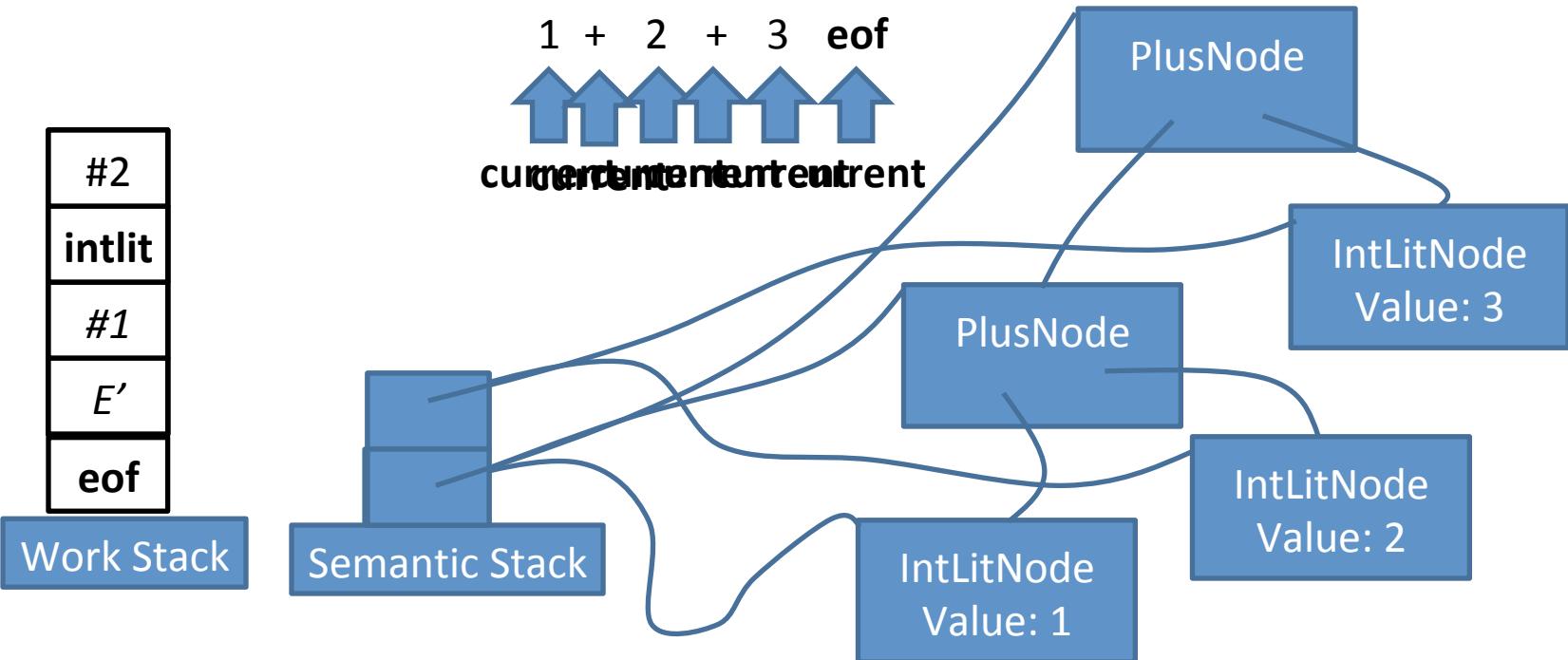
## “AST” SDT Actions

```
#1 tTrans = pop ;
eTrans = pop ;
push(new PlusNode(tTrans, eTrans))

#2 push(new IntLitNode(intlit.value))
```

intlit      +      EOF

	TE'		
E		+ T #1 E'	$\epsilon$
T	#2 intlit		



# We now have an AST

At this point, we have completed the frontend  
for (a) compiler

- Only recognize LL(1)

LL(1) is not a great class of languages

```
if (e1)
    stmt1
if (e2)
    stmt2
else
    stmt3
```

**Grammar Snippet**  
IfStmt -> **if lparens Exp rparens Stmt**  
         | **if lparens Exp rparens Stmt else Stmt**

# Announcements

Midterm review this Thursday

Try the midterm I put online

Homework solutions now available

# Semantic Analysis with Emphasis on Name Analysis

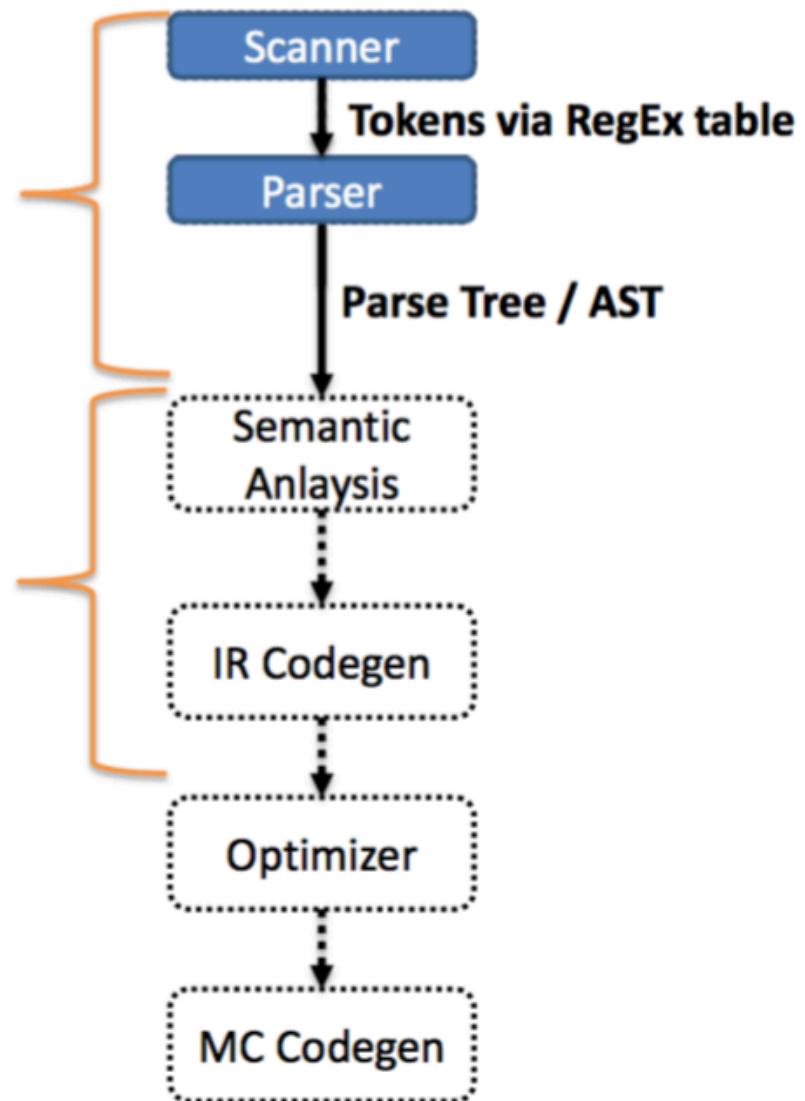
You'll need this for P4

We'll get back to Parsing next week

# Where we are at

So far, we've only defined the structure of a program—aka the syntax

We are now diving into the semantics of the program



# Semantics: The Meaning of a Program

The parser can guarantee that the program is structurally correct

The parser does not guarantee that the program makes sense:

- `void var;`
- Undeclared variables
- Ill-typed statements

```
int doubleRainbow;  
doubleRainbow = true;
```

language dependent

# Static Semantic Analysis

static-> compile time  
dynamic -> run time

Two phases

– Name analysis (aka name resolution)

• For each scope

– Process declarations, add them to symbol table

– Process statements, update IDs to point to their entry

– Type analysis

• Process statements

– Use symbol table info to determine the type of each expression (and sub-expression)

# Why do we need this phase?

## Code generation

- Different operations use different instructions:
  - Consistent variable access
  - Integer addition vs floating point addition
  - Operator overloading

## Optimization

- Symbol table knows where a variable is used
  - Can remove dead code
  - Can weaken the type (e.g., int -> bool)
  - NOTE: pointers can make this occasionally impossible

## Error checking

# Semantic Error Analysis

For non-trivial programming languages, we run into fundamental undecidability problems

- Does the program halt?
- Can the program crash?

Sometimes practical feasibility as well

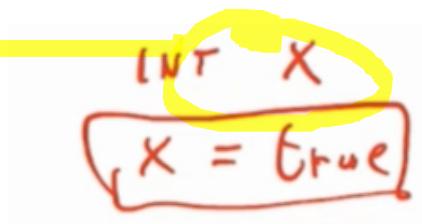
- Combinations thread interleavings
- Inter-procedural dataflow

# Catch Obvious Errors

We cannot guarantee absence of errors...

...but we can at least catch some:

- Undeclared identifiers
- Multiply declared identifiers
- Ill-typed terms

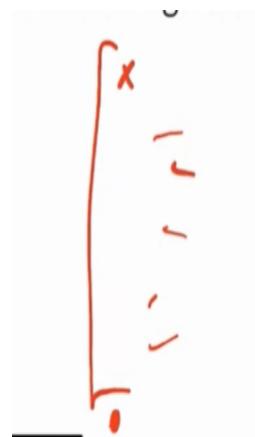


# Name analysis

Associating ids with their uses

Need to bind names before we can type uses

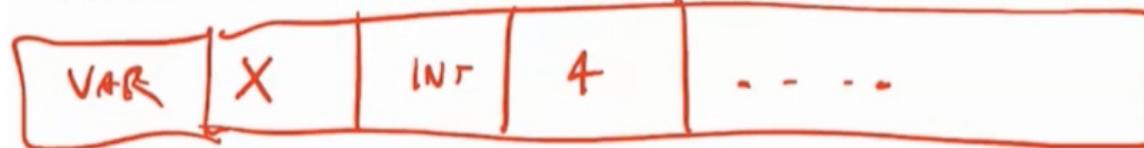
- What definitions do we need about identifiers?
  - Symbol table
- How do we bind definitions and uses together?
  - Scope



scope-> the segment of code that  
x lives

# Symbol table entries

Table that binds a name to information we need



What information do you think we need?

- Kind (struct, variable, function, class)
- Type (int, int × string → bool, struct)
- Nesting level
- Runtime location (where it's stored in memory)

# Symbol table operations

- Insert entry      declare a var
- Lookup      use a var
- Add new table      new scope founded, if statement, loop
- Remove/forget a table      the scope ends



When do you think we use these operations?

# Scope: the lifetime of a name

Block of code in which a name is visible/valid

No scope

- Assembly / FORTRAN

Static / most nested scope

- Should be familiar – C / Java / C++

```
void func() {  
    int a;  
}  
  
void soul(int b) {  
    if (b) {  
        int c = 2;  
    }  
}
```

**MANY DECISIONS RELATED TO  
SCOPE!!**

# Static vs Dynamic Scope

## Static

- Correspondence between a variable use / decl is known at compile time

## Dynamic

- Correspondence determined at runtime

if static, this will throw an error since x is not defined within this scope

if dynamic, this will work and print x as 10, 30.5

```
void main() {  
    f1();  
    f2();  
}  
  
void f1() {  
    int x = 10;  
    g();  
}  
  
void f2() {  
    String x = "hello";  
    f3();  
    g();  
}  
  
void f3() {  
    double x = 30.5;  
}  
  
void g() {  
    print(x);  
}
```

# Exercises

```
class animal {  
    // methods  
    void attack(int animal) {  
        for (int animal=0; animal<10; animal++) {  
            int attack;  
        }  
    }  
  
    int attack(int x) {  
        for (int attack=0; attack<10; attack++) {  
            int animal;  
        }  
    }  
  
    void animal() { }  
  
    // fields  
    double attack;  
    int attack;  
    int animal;  
}
```

not overloading: overloading requires different types of params, different return type wont work

What uses and declarations are OK in this Java code?

var attack and method attack -> overloading

# Exercises

```
void main() {  
    int x = 0;  
    f1();  
    g();  
    f2();  
}
```

```
void f1() {  
    int x = 10;  
    g();  
}
```

```
void f2() {  
    int x = 20;  
    f1();  
    g();  
}
```

```
void g() {  
    print(x);  
}
```

10 0 10 20

What does this print,  
assuming dynamic scoping?

these three are actually different var, while  
we are in scope, we always look for latest  
one

# Variable shadowing

Do we allow names to be reused in nesting relations?

What about when the kinds are different?

for java, above is error, below is Okay

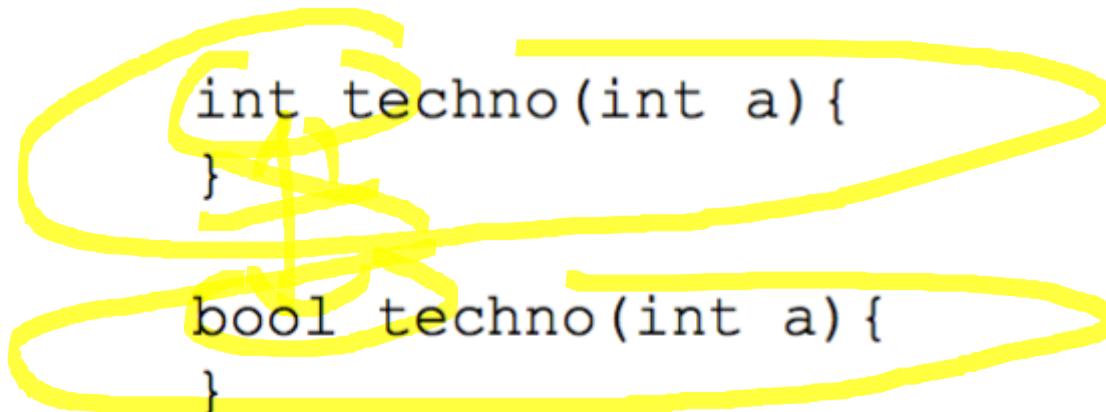
```
void smoothJazz(int a) {  
    int a;  
    if (a) {  
        int a;  
        if (a) {  
            int a;  
        }  
    }  
}
```

```
void hardRock(int a) {  
    int hardRock;  
}
```

# Overloading

Same name different type

for java, 1,3,4 or 2,3,4



int techno(int a) {  
}  
  
bool techno(int a) {  
}

bool techno(bool a) {  
}

bool techno(bool a, bool b) {  
}

# Forward references

Use of a name before it is added to symbol table  
How do we implement it?

```
void country() {  
    western();  
}  
  
void western() {  
    country();  
}
```

Requires two passes over the program  
– 1 to fill symbol table, 1 to use it

# Example

```
int k=10, x=20;
```

```
void foo(int k) {  
    int a = x;  
    int x = k;  
    int b = x;  
    while (...) {  
        int x;  
        if (x == k) {  
            int k, y;  
            k = y = x;  
        }  
        if (x == k) {  
            int x = y;  
        }  
    }  
}
```

Determine which uses correspond to which declarations

y is out of scope

# Example

```
int (1)k=10, (2)x=20;

void (3)foo(int (4)k) {
    int (5)a = x(2);
    int (6)x = k(4);
    int (7)b = x(6);
    while (...) {
        int (8)x;
        if (x(8) == k(4)) {
            int (9)k, (10)y;
            k(9) = y(10) = x(8);
        }
        if (x(8) == k(4)) {
            int (11)x = y(ERROR);
        }
    }
}
```

Determine which uses correspond  
to which declarations

# Name analysis for our language

Time to make some decisions

- What scoping rules will we allow?
- What info does our project compiler need in its symbol table?
- Relevant for P4

# Our language is statically scoped

Designed for ease of symbol table use

- global scope + nested scopes
- all declarations are made at the top of a scope
- declarations can always be removed from table at end of scope

```
int a;  
void fun() {  
    int b;  
    int c;  
    int d;  
    b = 0;  
    if (b == 0) {  
        int d;  
    }  
    c = b;  
    d = b + c;  
}
```

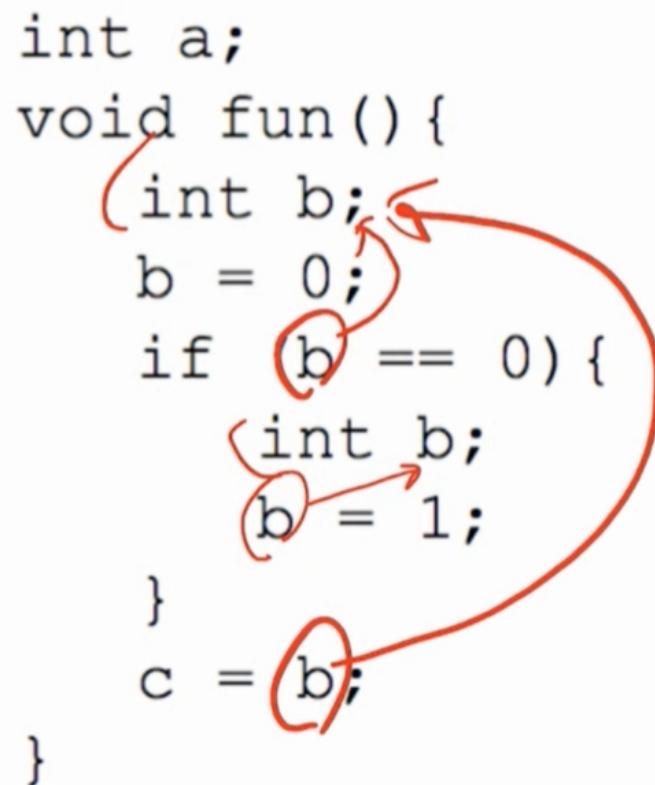
# Our language: Nesting

Like Java or C, we'll use most deeply nested scope to determine binding

- Shadowing

- Variable shadowing allowed
- Struct definition shadowing allowed

```
int a;  
void fun() {  
    int b;  
    b = 0;  
    if (b == 0) {  
        int b;  
        b = 1;  
    }  
    c = b;  
}
```



# Our language: Symbol table implementation

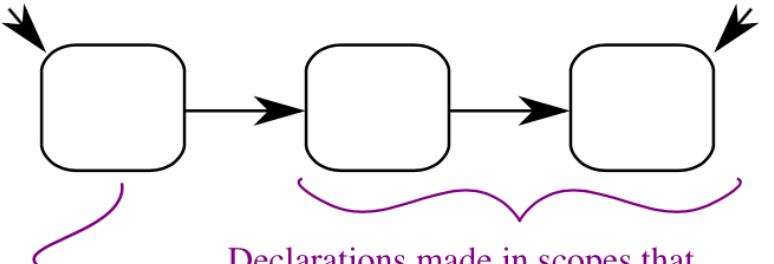
We want the symbol table to efficiently add an entry when we need it, remove it when we're done with it

We'll go with a list of hashmaps

- This makes sense since we expect to remove a lot of names from scope at once
- You did most of this in P1

# Example

front of list



Declarations  
made in S

Declarations made in scopes that enclose S. Each hashtable in the list corresponds to one scope (i.e. contains all declarations for that scope)

A code snippet illustrating scope nesting:

```
void f(int a, int b) {
    double x;
    while (...) {
        int x, y;
        ...
    }
    void g() {
        f();
    }
}
```

The code is annotated with red numbers 1, 2, and 3 indicating scopes:

- Scope 1 (outermost): The entire function body.
- Scope 2: The block of the `while` loop.
- Scope 3: The block of the `int x, y` declaration.

Red ovals highlight variable names `x` and `y` which are shadows of variables declared in outer scopes.

x: int, 3  
y: int, 3

Declarations  
in the loop

a: int, 2  
b: int, 2  
x: double, 2

Declarations in f

f: (int,int) -> void, 1    g: () -> void, 1

Global Declarations

1

# Our language: Symbol kinds

Symbol kinds (= types of identifiers)

- Variables
  - Carries a name, primitive type
- Function declarations
  - Carries a name, return type, list of parameter types
- Struct definitions
  - Carries a name, list of fields (types with names), size

# Our language: Sym class implementation

There are many ways to implement your symbols  
Here's one suggestion

- Sym class for variable definitions
- FnSym subclass for function declarations
- StructDefSym for struct type definitions
  - Contains it's OWN symbol table for it's field definitions
- StructSym for when you want an instance of a struct

# Implementing name analysis with an AST

At this point, we're done with the Parse Tree

- All subsequent processing done on the AST + symbol table

Walk the AST, much like the unparse() method

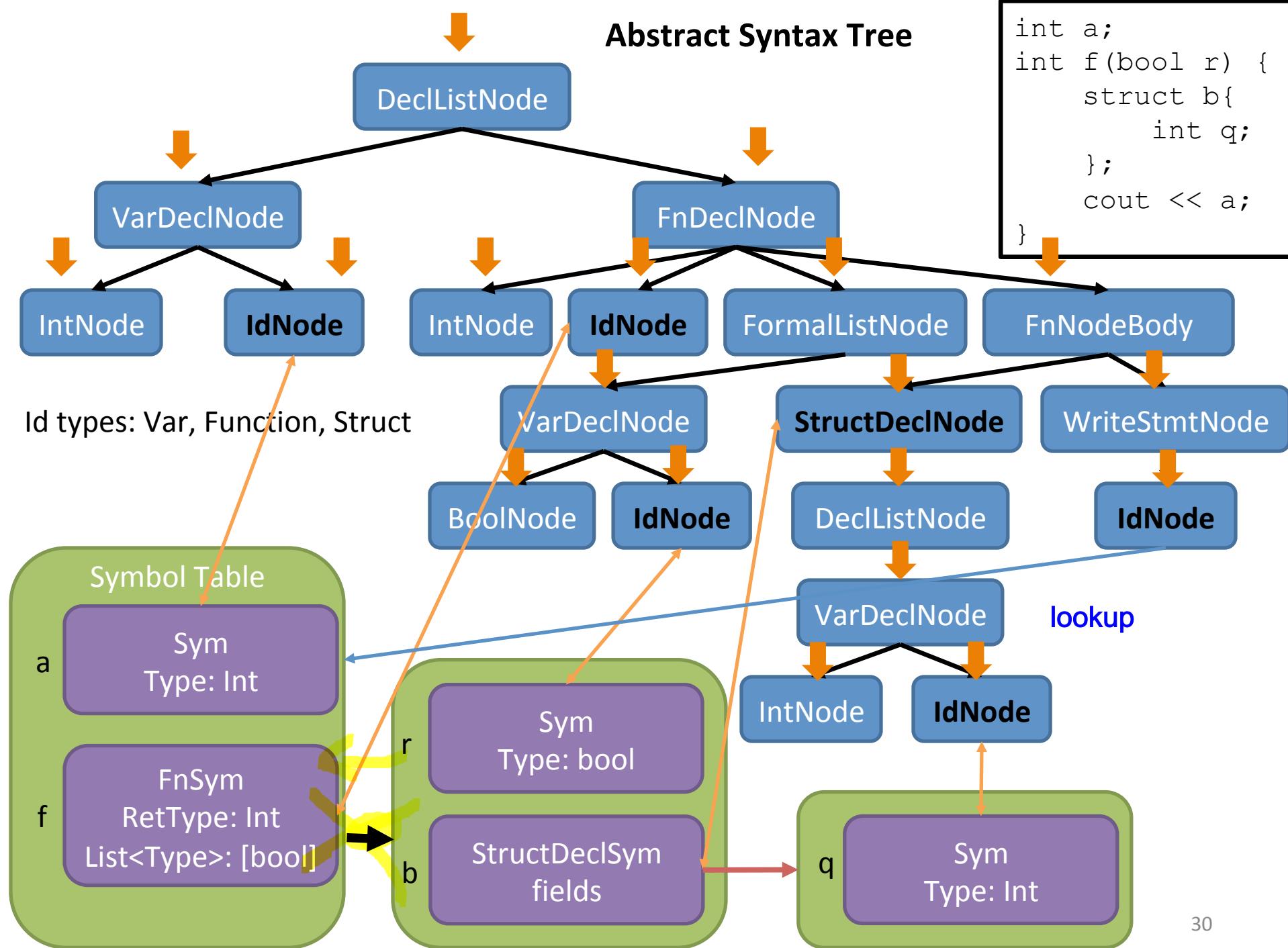
- Augment AST nodes where names are used (both declarations and uses) with a link to the relevant object in the symbol table
- Put new entries into the symbol table when a declaration is encountered

# Abstract Syntax Tree

```

int a;
int f(bool r) {
    struct b{
        int q;
    };
    cout << a;
}

```



Id types: Var, Function, Struct

Symbol Table

a  
  Sym  
  Type: Int

f  
  FnSym  
  RetType: Int  
  List<Type>: [bool]

r  
b  
  Sym  
  Type: bool

StructDeclSym  
  fields

q  
  Sym  
  Type: Int

lookup

# LR Bottom-up Parsing

# Roadmap

Last class

- Name analysis

Previous-ish last class

- LL(1)

Today's class

- LR Parsing
  - SLR(1)

# Lecture Outline

## Bottom-Up parsing

- Talk about the language class / theory
- Describe the state that it keeps / intuition
- Show how it works
- Show how it is built

# LL(1) Not Powerful Enough for all PL

Left-recursion

Not left factored

Doesn't mean LL(1) is bad

- Right tool for simple parsing jobs



```
stmtList ::= stmtList stmt  
          | /* epsilon */  
          ;
```

# We Need a *Little* More Power

Could increase the lookahead

- Up until the mid 90s, this was considered impractical

Could increase the runtime complexity

- CYK has us covered there

Could increase the memory complexity

- i.e. more elaborate parse table

# LR Parsers

Left-to-right scan of the input file

Reverse **rightmost** derivation

Advantages

- Can recognize almost any programming language
- Time and space  $O(n)$  in the input size
- More powerful than the corresponding LL parser i.e.  
 $LL(1) < LR(1)$   $LL(2) < LR(2)$

Disadvantages

- More **complex** parser generation
- Larger parse tables

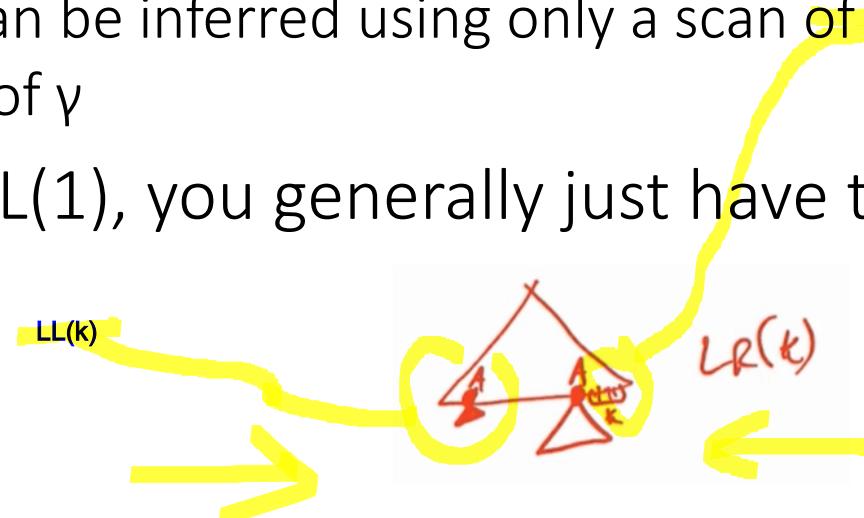
# LR Parser Power

Let  $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow w$  be a rightmost derivation, where  $w$  is a terminal string

Let  $\alpha A \gamma \Rightarrow \alpha \beta \gamma$  be a step in the derivation

- So  $A \rightarrow \beta$  must have been a production in the grammar
- $\alpha \beta \gamma$  must be some  $\alpha_i$  or  $w$
- A grammar is LR( $k$ ) if for every derivation step,  $A \rightarrow \beta$  can be inferred using only a scan of  $\alpha \beta$  and at most  $k$  symbols of  $\gamma$

Much like LL(1), you generally just have to go ahead and try it



# LR Parser types

LR(1)

- Can recognize any DCFG
- Can experience blowup in parse table size

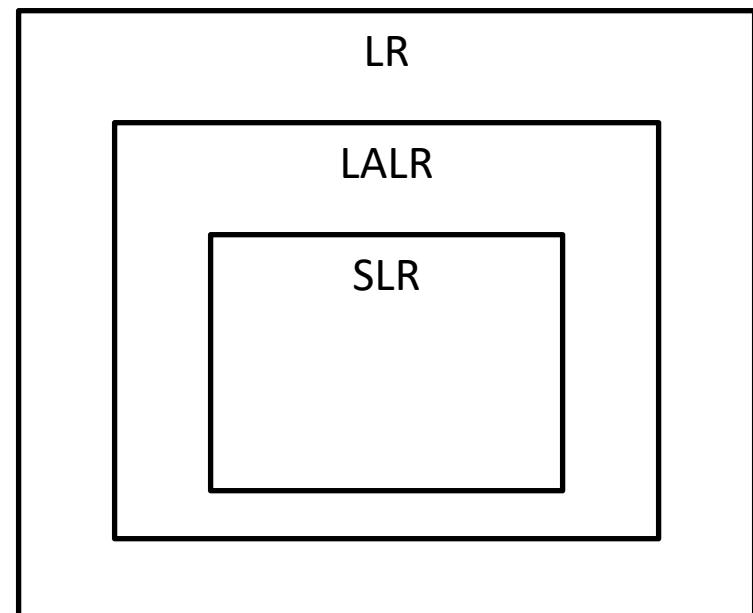
LALR(1)

SLR(1)

- Both proposed at the same time to limit parse table size

deterministic context free grammar

Recognizable by a deterministic PDA



# Which parser should we use?

Different variants mostly differ in how they build the parse table, we can still talk about all the family in general terms

- Today we'll cover SLR
- Pretty easy to learn LALR from there

## LALR(1)

- Generally considered a good compromise between parse table size and expressiveness
- Class for Java CUP, yacc, and bison Antlr

# How does Bottom-up Parsing work?

Already seen 1 such parser: CYK

- Simultaneously tracked every possible parse tree
- LR parsers work in a similar same way

Contrast to top-down parser

- We know exactly where we are in the parse
- Make predictions about what's next

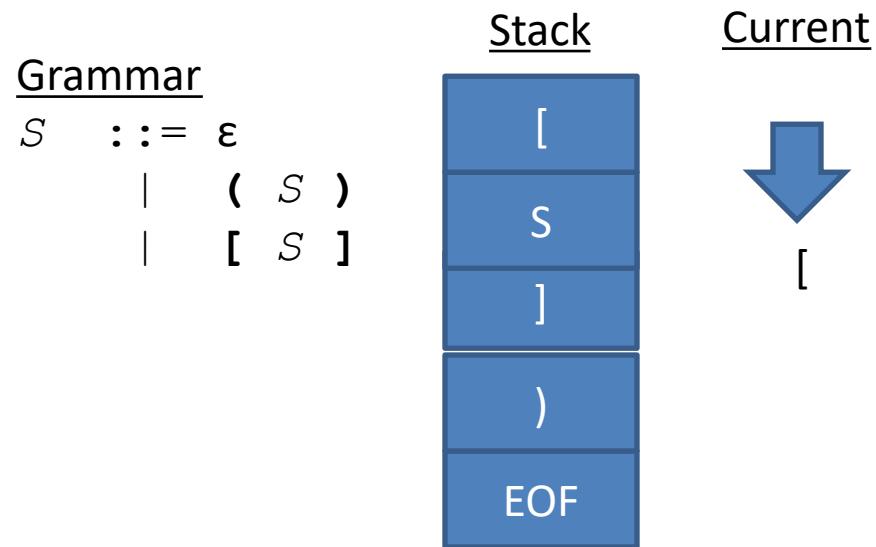
# Parser State

## Top-down parser state

- Current token
- Stack of symbols
  - Represented what we expect in the rest of our descent to the leaves
- Worked down and to the left through tree

## Bottom-up parser state

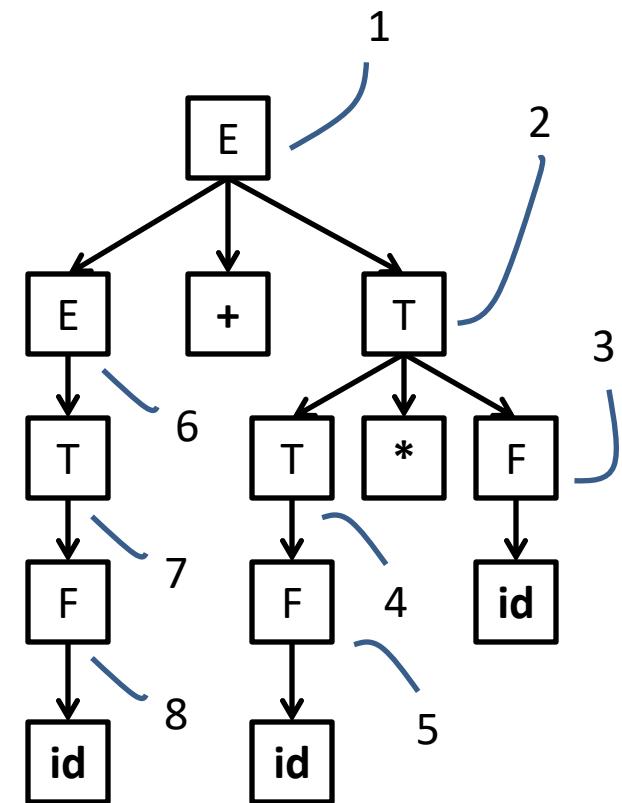
- Also maintains a stack and token
  - Represents summary of input we've seen
- Works upward and to the right through the tree
- Also has an auxiliary state machine to help disambiguate rules



# LR Derivation Order

Let's remember derivation orders again

Reverse	Rightmost derivation	从右至左开始扩展
8	$E \Rightarrow E + T$	
7	$\Rightarrow E + T * F$	
6	$\Rightarrow E + T * id$	
5	$\Rightarrow E + F * id$	
4	$\Rightarrow E + id * id$	
3	$\Rightarrow T + id * id$	
2	$\Rightarrow F + id * id$	
1	$\Rightarrow id + id * id$	



# Parser Operations

Top-down parser

- *Scan* the next input token
- *Push* a bunch of RHS symbols
- *Pop* a single symbol

Bottom-up parser

- *Shift* an input token into a stack item
- *Reduce* a bunch of stack items into a new parent item (on the stack)

# Parser Actions: Simplified view

<u>Stack</u>	<u>Input</u>	<u>Action</u>
	id + id * id EOF	shift(id)
id	+ id * id EOF	reduce by F → id
F	+ id * id EOF	reduce by T → F
T	+ id * id EOF	reduce by E → T
E	+ id * id EOF	shift +
E +	id * id EOF	shift id
E + id	* id EOF	reduce by F → id
E + F	* id EOF	reduce by T → F
E + T	* id EOF	shift *
E + T *	id EOF	shift id
E + T * id	EOF	reduce by F → id
E + T * F	EOF	reduce by T → T * F
E + T	EOF	reduce by E → E + T
E	EOF	accept

bottom up approach

```

graph TD
    E[E] --> E1[E]
    E --> P[+]
    E --> T1[T]
    E1 --> E2[E]
    P --> T2[T]
    T1 --> T3[T]
    T3 --> M[*]
    T3 --> F1[F]
    F1 --> ID1[id]
    T2 --> F2[F]
    F2 --> ID2[id]
  
```

# Stack Items

Note that the previous slide was called “simplified”

Stack elements are representative of symbols

- Actually known as items
  - Indicate a production and a position within the production

$$X \rightarrow \alpha \cdot B \beta$$

to mark where we are in this production

in LR, alpha is in the stack

in LR(1), B beta is in the stack

- Means

- we are in a production of X
- We believe we've parsed (arbitrary) symbol string  $\alpha$
- We could handle a production of B
- After that we'll have  $\beta$

# Stack Item Examples

## Example 1

$PList \rightarrow ( . IDList )$

We've parsed the ( and are now moving onto IDList

## Example 2

$PList \rightarrow ( IDList . )$

You have parsed ( IDList and are about to parse )

## Example 3

$PList \rightarrow ( IDList ) .$

I'm done with this production

## Example 4

$PList \rightarrow . ( IDList )$

I'm about to start this production

# Stack Item State

You may not know  
exactly which item you  
are parsing

LR Parsers actually track  
the set of states that you  
*could* have been in

## Grammar snippet

$S \rightarrow A$

$A \rightarrow B$

|  $C$

$B \rightarrow D \text{ id}$

$C \rightarrow \text{id } E$

$D \rightarrow \text{id } E$

$\{S \rightarrow .A, A \rightarrow .B, A \rightarrow .C, \dots\}$

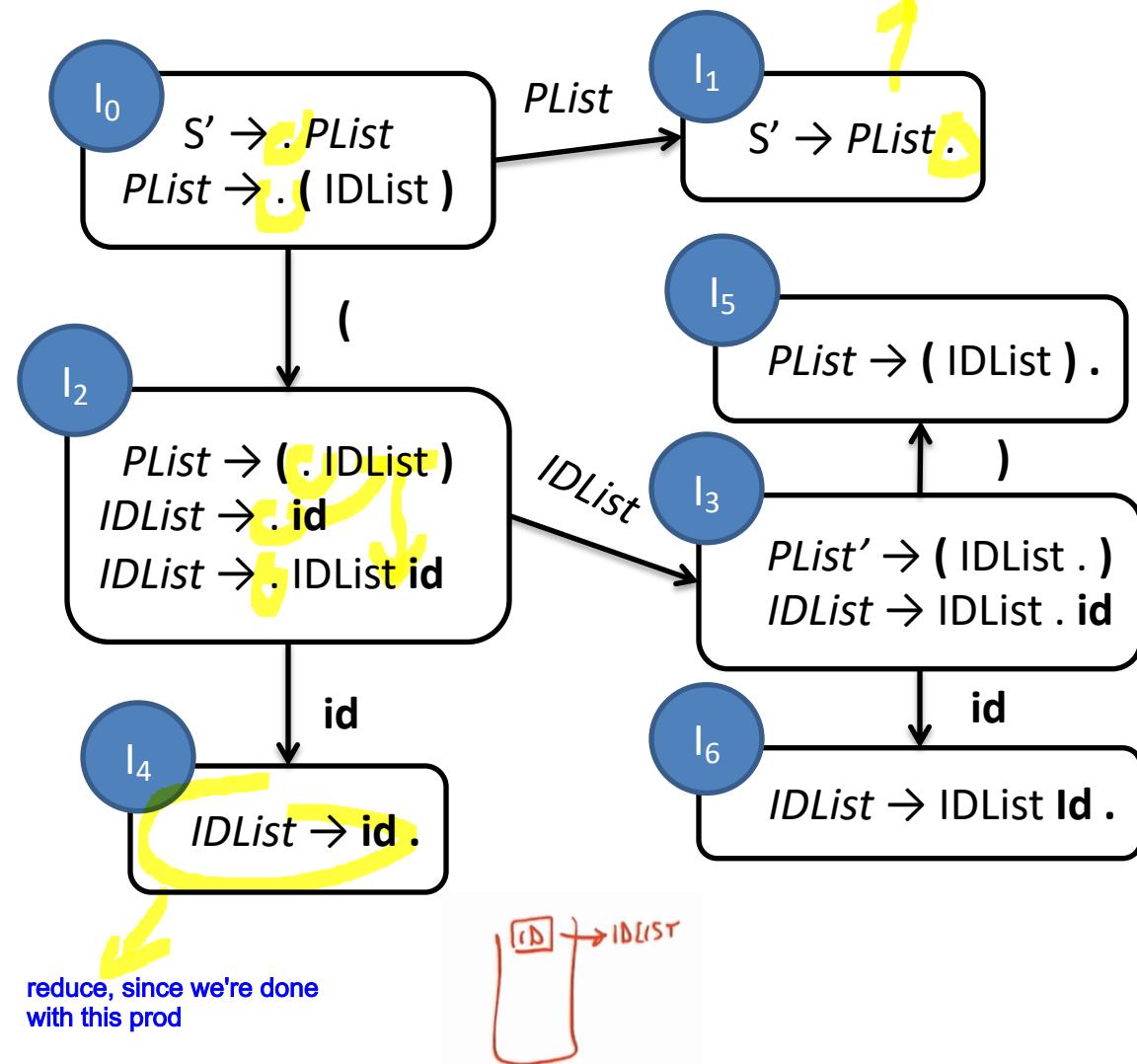
# LR Parser FSM

done  
1

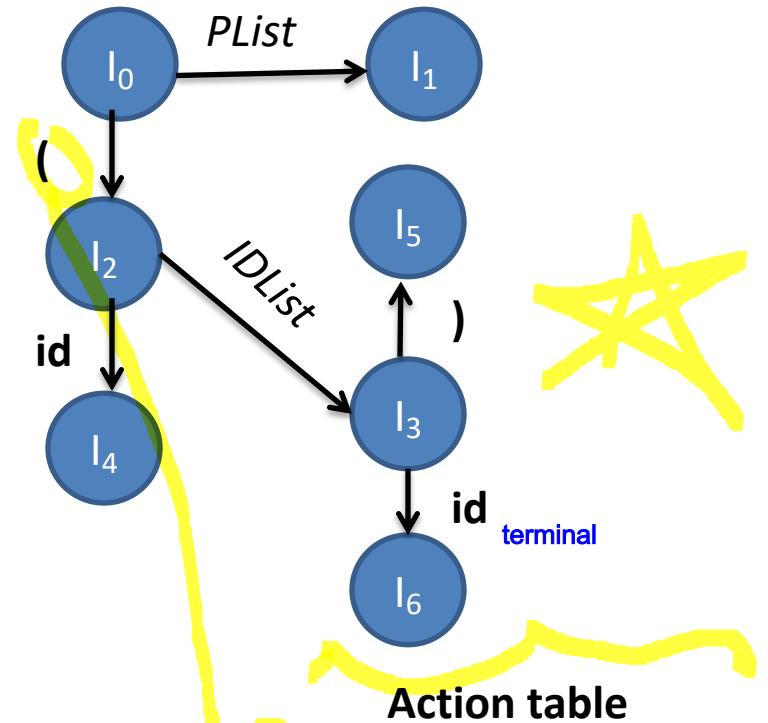
这里就是first set 的逻辑

## Grammar G

$S' \rightarrow PList$   
 $PList \rightarrow ( IDList )$   
 $IDList \rightarrow id$   
 $IDList \rightarrow IDList id$



# Automaton as a table



- Shift corresponds to taking a terminal edge
- Reduce corresponds to taking a nonterminal edge

Action table

	(	)	id	eof	<i>PList</i>	<i>IDList</i>
0	S 2				1	
1						
2			S 4			3
3		S 5	S 6			
4						
5						
6						

GoTo table

nonterminal

Shift and go to state 6

# How do we know when to reduce?

	Action table				GoTo table	
	(	)	id	eof	<i>PList</i>	<i>IDList</i>
0	S 2				1	
1						
2			S 4			3
3		S 5	S 6			
4		R ③	R ③			
5				R ②		
6		R ④	R ④			

## Grammar G

- ①  $S' \rightarrow PList$
- ②  $PList \rightarrow ( IDList )$
- ③  $IDList \rightarrow id$
- ④  $IDList \rightarrow IDList id$

Only see terminals in the input

Actually do reduce steps in 2 phases

- Action table will tell us when to reduce (and how much)
- GoTo will tell us where to... go to

# How do we know we're done?

	Action table				GoTo table	
	(	)	id	eof	<i>PList</i>	<i>IDList</i>
0	S 2				1	
1				☺		
2			S 4			3
3		S 5	S 6			
4		R ③	R ③			
5				R ②		
6		R ④	R ④			

Add an accept token  
Any other cell is an error

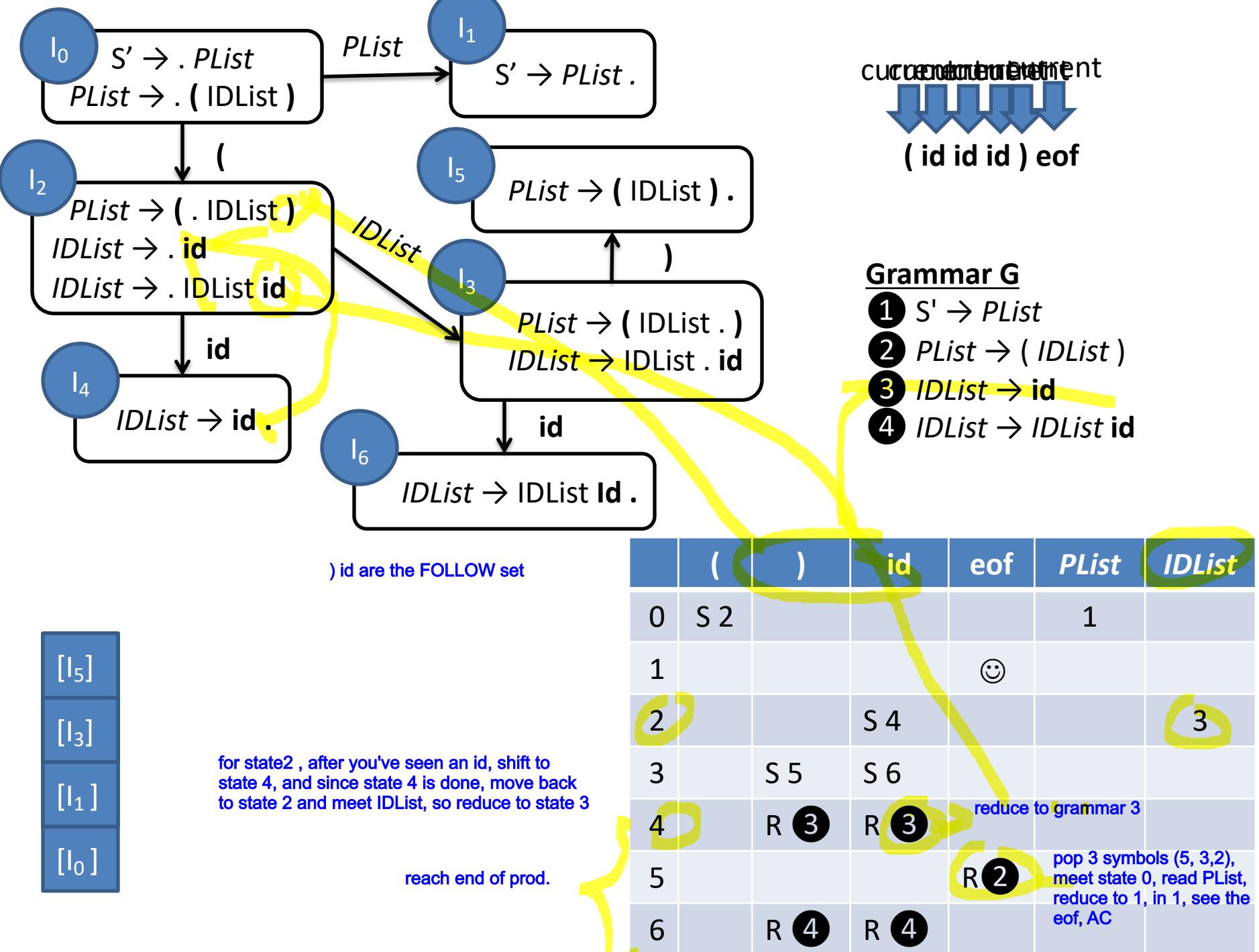
## Grammar G

- 1  $S' \rightarrow PList$
- 2  $PList \rightarrow ( IDList )$
- 3  $IDList \rightarrow id$
- 4  $IDList \rightarrow IDList id$

# Full Parse Table Operation

```
Initialize stack
a = scan()
do forever
    t = top-of-stack (state) symbol
    switch action[t, a] {
        case shift s:
            push(s)
            a = scan()
        case reduce by A → alpha:
            for i = 1 to length(alpha) do pop() end
            t = top-of-stack symbol
            push(goto[t, A])
        case accept:
            return( SUCCESS )
        case error:
            call the error handler
            return( FAILURE )
    }
end do
```

# Example Time



Seems that LR Parser works great  
What could possibly go wrong?

# LR Parser State Explosion

Tracking sets of states  
can cause the size of the  
FSM to blow up



The SLR and LALR  
variants exist to combat  
this explosion

Slight modification to  
item and table form



# Building the SLR Automaton

Uses 2 sets

- $\text{Closure}(I)$ 
  - What is the set of items we could be in?
  - Given  $I$ : what is the set of items that could be mistaken for  $I$  (reflexive)
- $\text{Goto}(s, X)$ 
  - If we are in state  $I$ , where might we be after parsing  $X$ ?

Vaguely reminiscent of FIRST and FOLLOW

# Closure Sets

Put I itself into Closure(I)

While there exists an item in Closure(I) of form

$X \rightarrow \alpha . B \beta$

such that there is a production  $B \rightarrow \gamma$

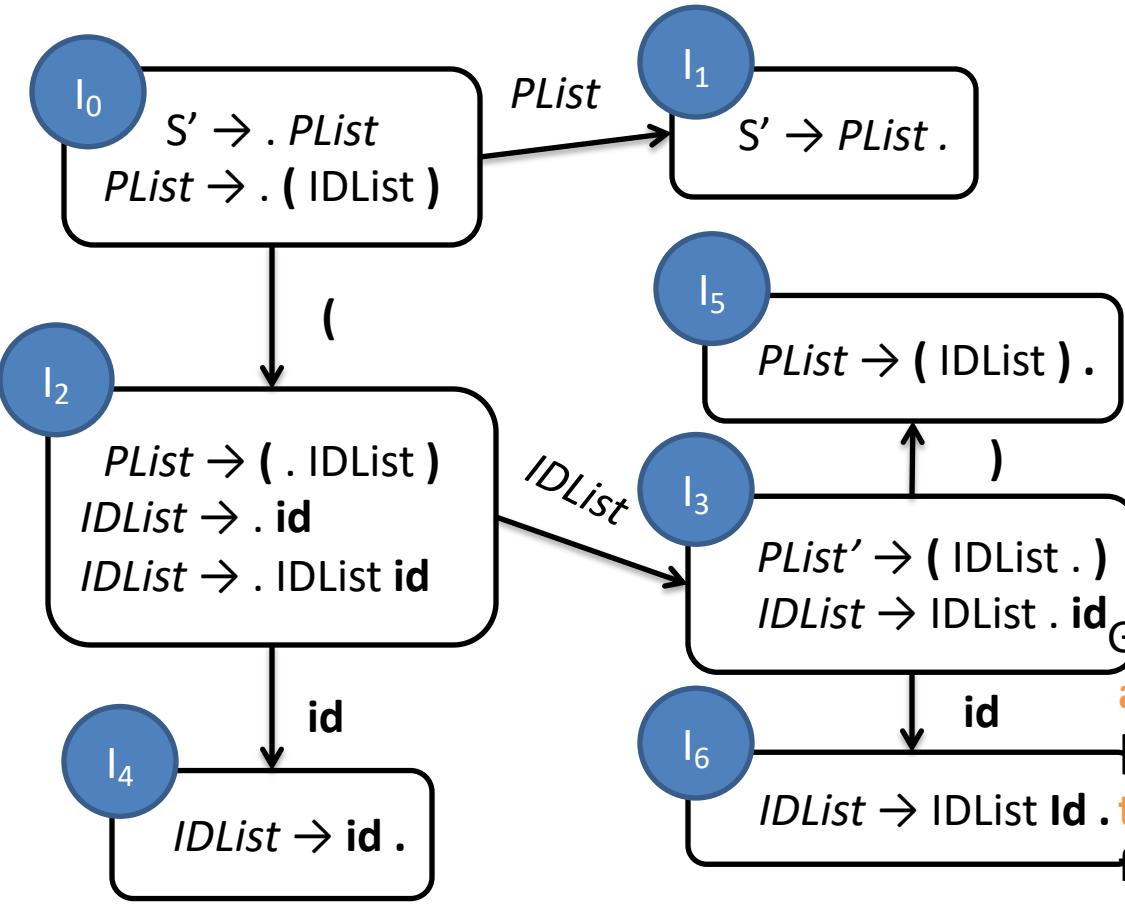
and  $B \rightarrow . \gamma$  is not in Closure(I)

add  $B \rightarrow . \gamma$  to Closure(I)

# GoTo Sets

$\text{Goto}(l, X) =$

$\text{Closure}(\{ A \rightarrow \alpha X . B \mid A \rightarrow \alpha . X \beta \text{ is in } l \})$



## Grammar G

$S' \rightarrow PList$   
 $PList \rightarrow ( IDList )$   
 $IDList \rightarrow id$   
 $IDList \rightarrow IDList id$

## GoTo(j, X)

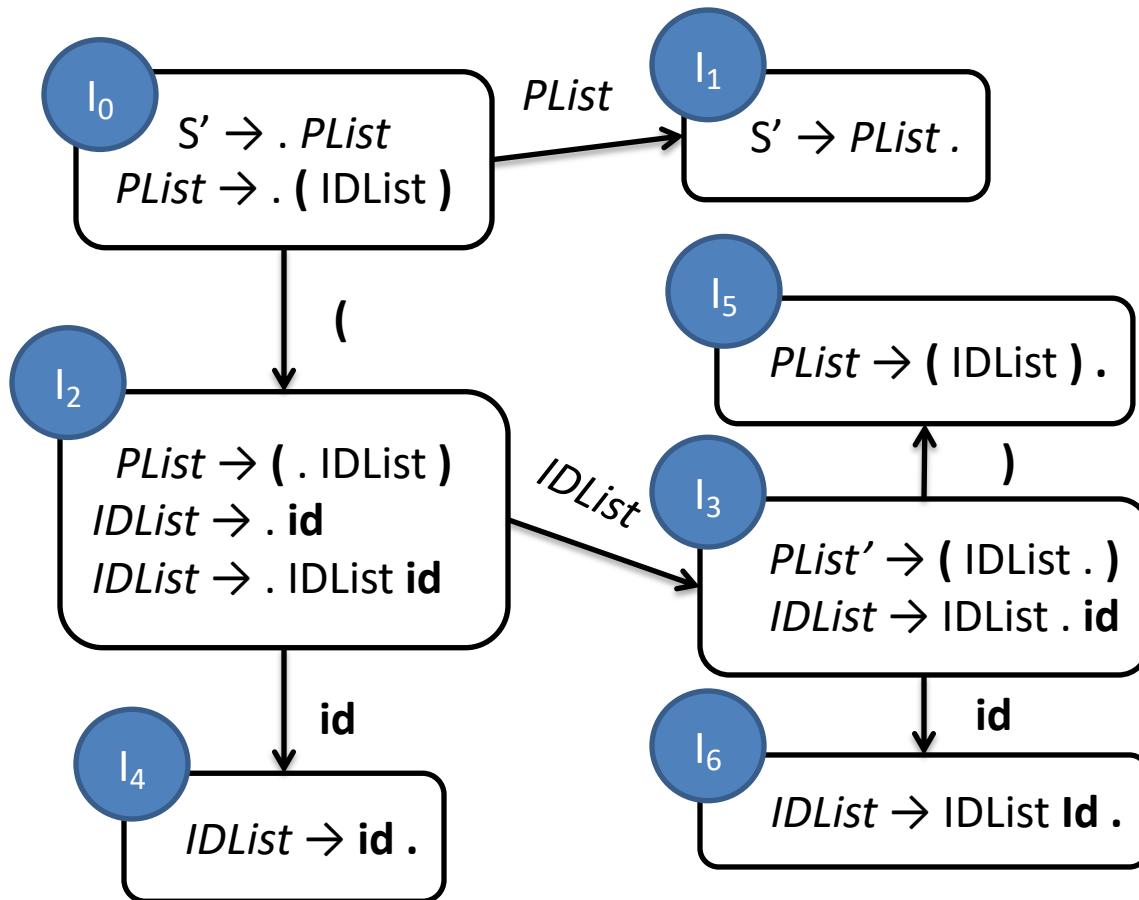
Put closure of  $\{I\}$  items  
 Repeat for  $X \beta$  s.t.  $A \rightarrow \alpha X \beta \in I$   
 $X \rightarrow \alpha B \beta \in \text{Closure}(I)$  s.t.  
 $\exists B \rightarrow \gamma$ , add  $B \rightarrow \gamma$  to  $\text{Closure}(I)$   
 GoTo( $j$ ,  $X \beta$ )  
 GoTo( $j$ ,  $\beta$ )  
 all items  $\alpha A \beta \in \text{Closure}(I)$   
 all items  $A \rightarrow \alpha IDList \beta$   
 [1]  $PList \rightarrow S \rightarrow PList \beta$   
 [1]  $PList \rightarrow ( IDList ) \beta$   
 those where  $A \rightarrow PList \beta \in I_0$   
 for [1]  $PList \rightarrow S \rightarrow PList \beta$ ,  $PList \beta \in I_0$   
 those where  $A \rightarrow PList \beta \in I_2$   
 set to closure of the following:  
 for [1]  $PList \rightarrow ( IDList ) \beta$ ,  $IDList \beta \in I_2$   
 for [2]  $PList \rightarrow IDList \beta$ ,  $IDList \beta \in I_2$  in  $I_2$   
 Item  $\beta$  does nothing here  $IDList \beta \in G$   
 set to closure is  
 {  $IDList \rightarrow PList \beta$  }  
 {  $PList \rightarrow ( IDList ) \beta$ ,  $IDList \rightarrow ( IDList . ) \beta$  }  
 $IDList \rightarrow . IDList id \beta$   
 Only terminals after . so closure done  
 Done with closure, and GoTo

## Parse Table Construction

- 1: Add new start  $S'$  and  $S' \rightarrow S$
- 2: Build State  $I_0$  for  $\text{Closure}(\{S' \rightarrow . S\})$
- 3: Saturate FSM:

for each symbol  $X$  s.t. there is an item in state  $j$  containing  $. X$   
add transition from state  $j$  to state for  $\text{GoTo}(j, X)$

# From FSM to parse table(s)

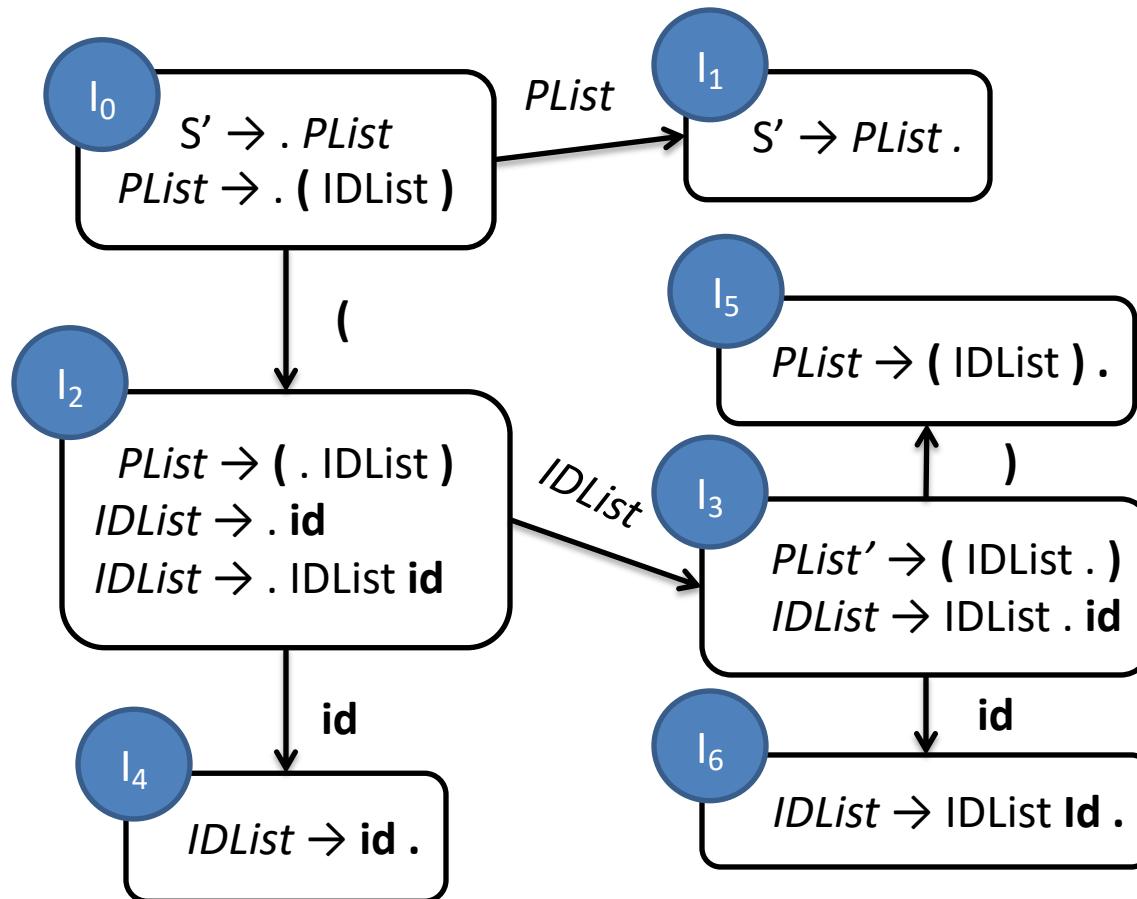


Need to connect the FSM back to the grammar

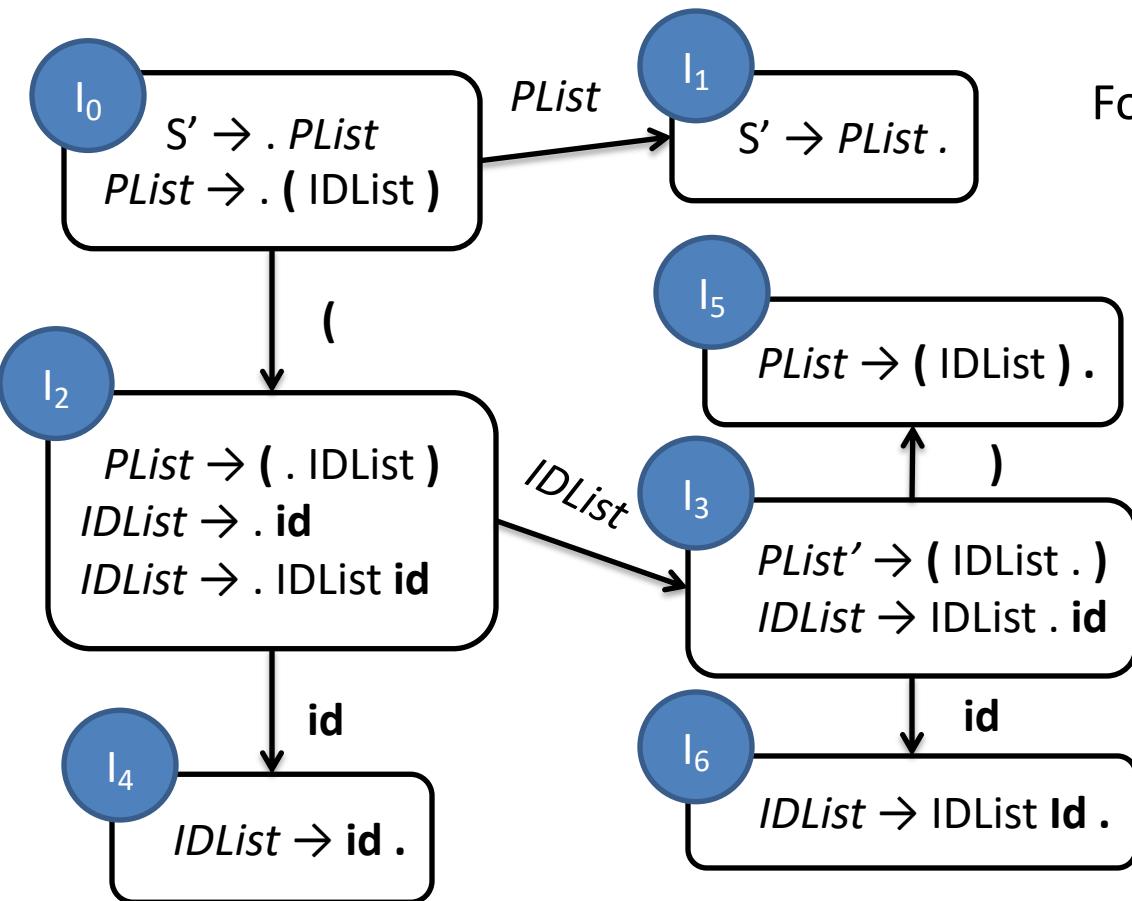
## Grammar G

- 1  $S' \rightarrow PList$
- 2  $PList \rightarrow ( IDList )$
- 3  $IDList \rightarrow id$
- 4  $IDList \rightarrow IDList id$

# Can Now Build Action and GoTo Tables



# Building the GoTo Table



For every nonterminal  $X$   
if there is an  $(i,j)$  edge on  $X$   
set  $\text{GoTo}[i,X] = j$

	<i>PList</i>	<i>IDList</i>
0	1	
1		
2		3
3		
4		
5		
6		

# Building the Action Table

If state  $i$  includes item  $A \rightarrow \alpha . t \beta$

- where  $t$  is a terminal
- and there is an  $(i,j)$  transition on  $t$
- set  $\text{Action}[i,t] = \text{shift } j$

If state  $i$  includes item  $A \rightarrow \alpha .$

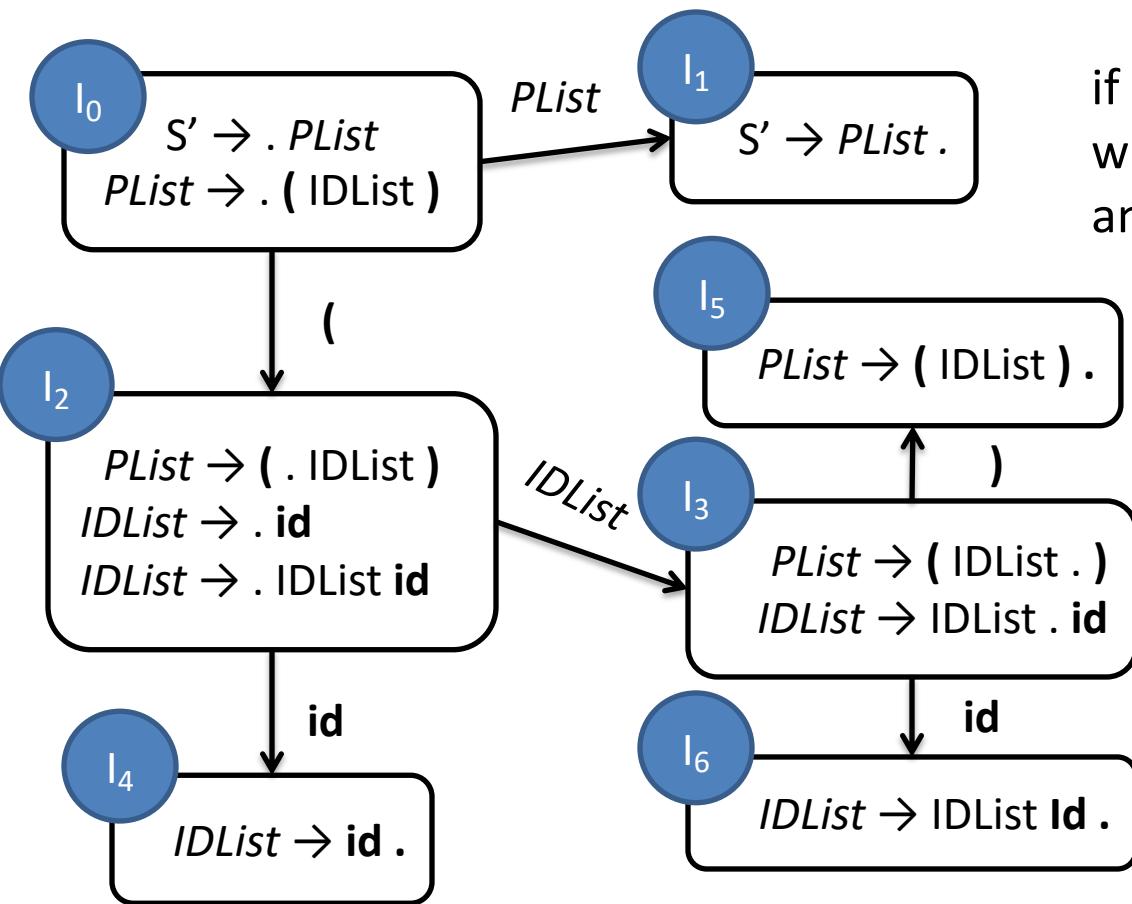
- where  $A$  is not  $S'$
- for each  $t$  in  $\text{FOLLOW}(A)$ :
- set  $\text{Action}[i,t] = \text{reduce by } A \rightarrow \alpha$

If state  $i$  includes item  $S \rightarrow S .$

- set  $\text{Action}[i, \text{eof}] = \text{accept}$

All other entries are error actions

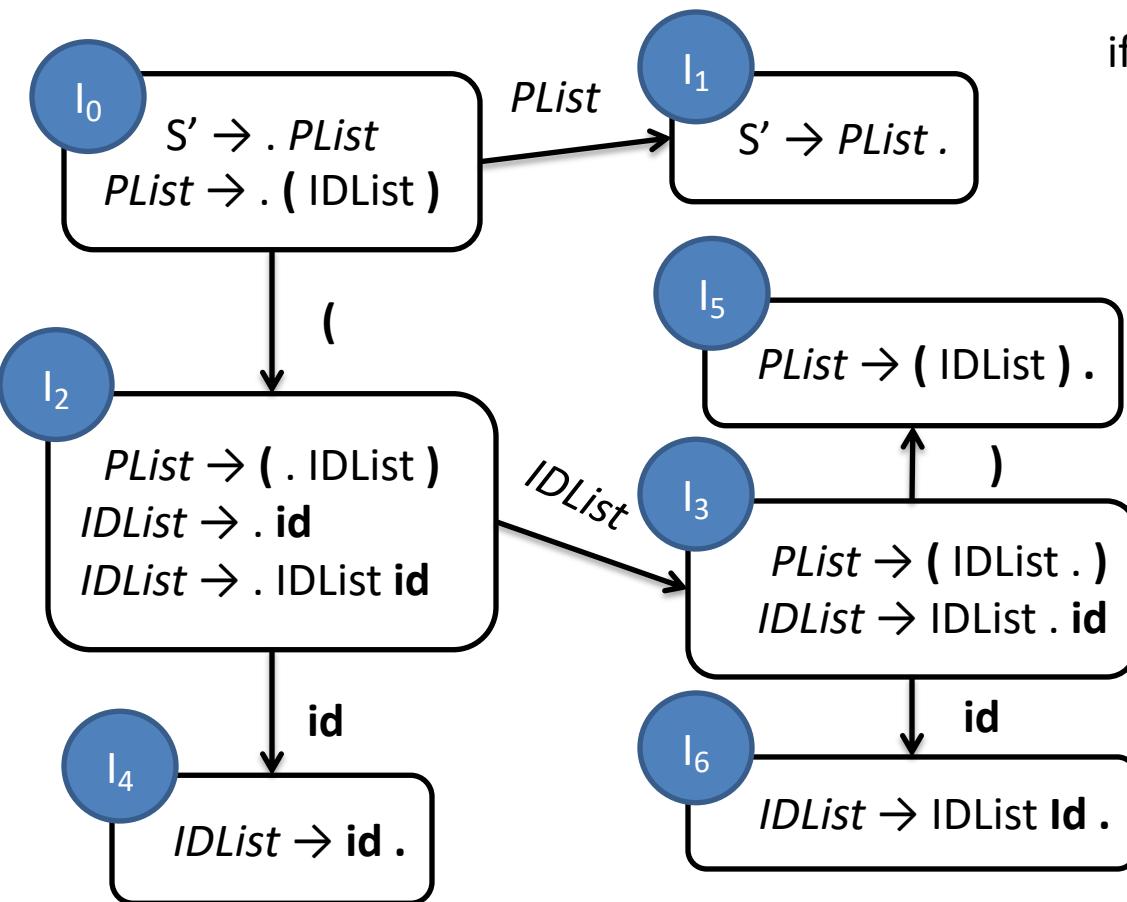
# Action Table: Shift



if state  $i$  includes item  $A \rightarrow \alpha . t \beta$   
 where  $t$  is a terminal  
 and there is an  $(i,j)$  transition on  $t$   
 set  $\text{Action}[i,t] = \text{shift } j$

	(	)	id	eof
0	S 2			
1				
2			S 4	
3		S 5	S 6	
4				
5				
6				

# Action Table: Reduce



if state  $i$  includes item  $A \rightarrow \alpha$ .  
 where  $A$  is not  $S'$   
 for each  $t$  in FOLLOW( $A$ ):  
 set Action[ $i, t$ ] = reduce by  $A \rightarrow \alpha$

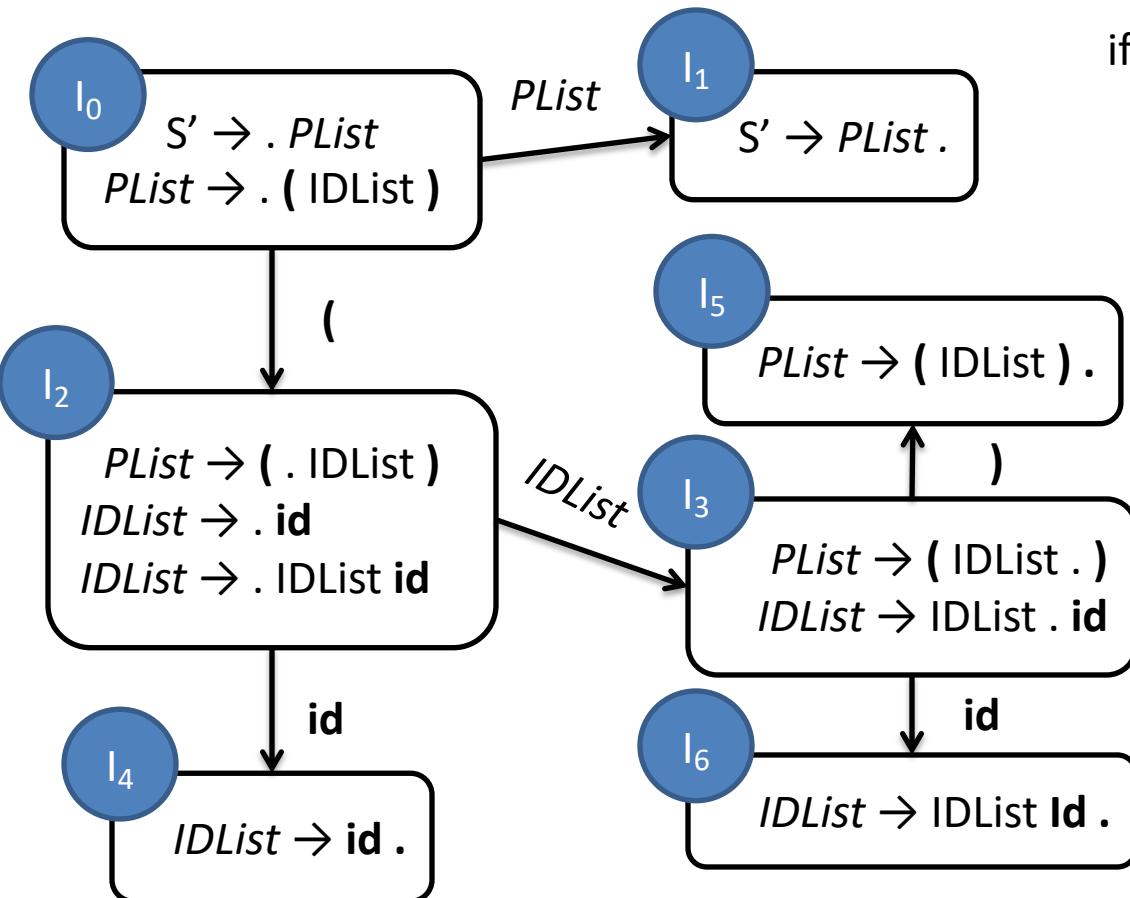
FOLLOW( $IDList$ ) = { ), id }  
 FOLLOW( $PList$ ) = { eof }

	(	)	id	eof
0	S 2			
1				
2			S 4	
3		S 5	S 6	
4		R 3	R 3	
5				R 2
6		R 4	R 4	

## Grammar G

- 1  $S' \rightarrow PList$
- 2  $PList \rightarrow ( IDList )$
- 3  $IDList \rightarrow id$
- 4  $IDList \rightarrow IDList id$

# Action Table: Accept



if state  $i$  includes item  $S' \rightarrow S .$   
set  $\text{Action}[i, \text{eof}] = \text{accept}$

	(	)	id	eof
0	S 2			
1				☺
2			S 4	
3		S 5	S 6	
4		R 3	R 3	
5				R 2
6		R 4	R 4	

## Grammar G

- 1  $S' \rightarrow PList$
- 2  $PList \rightarrow ( IDList )$
- 3  $IDList \rightarrow id$
- 4  $IDList \rightarrow IDList id$

# Some Final Thoughts on LR Parsing

A bit complicated to build the parse table

- Fortunately, algorithms exist

Still not as powerful as CYK

lr 的缺点

- Shift/reduce: action table cell includes S and R
- Reduce/reduce: cell include > 1 R rule

SDT similar to LL(1)

- Embed SDT action numbers in action table
- Fire off on reduce rules

done with parsing

# Types

# Roadmap

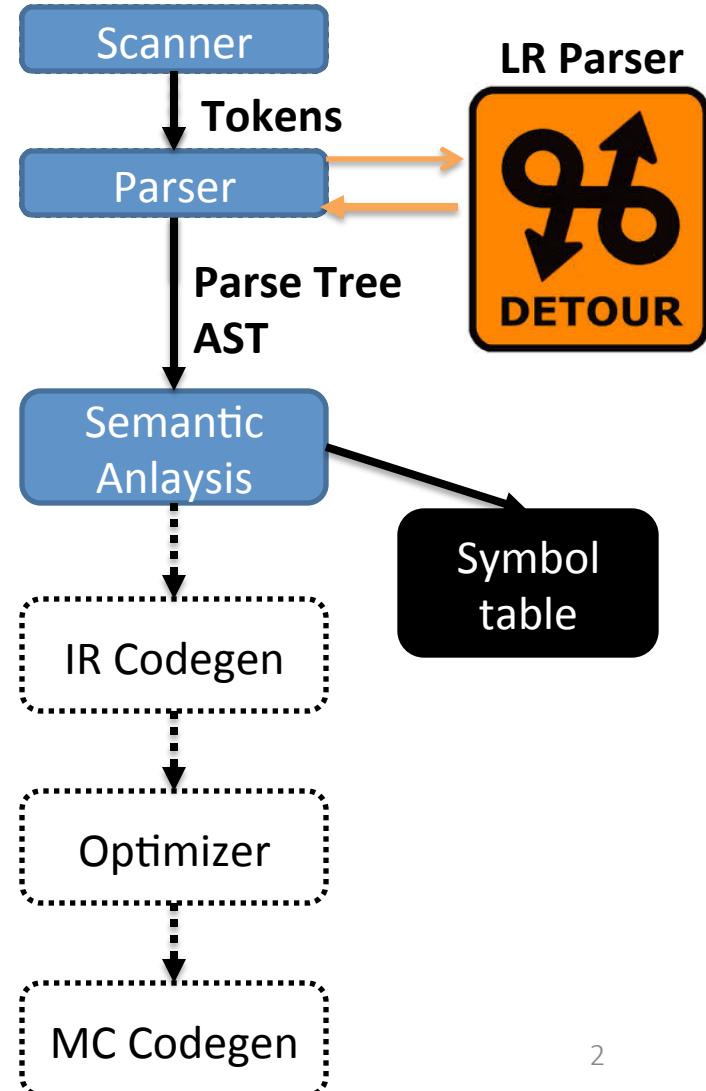
Back from LR Parsing Detour

Name analysis

- Static vs dynamic
- Scope

Today

- Type checking



# Lecture Outline

## Type Safari

- Type system concepts
- Type system vocabulary

## For our language

- Type rules
- How to apply type rules

## Data representation

- Moving towards actual code generation
- Brief comments about types in memory

# Say, What *is* a Type?

Short for “data type”

- Classification identifying kinds of data
- A set of possible values which a variable can possess
- Operations that can be done on member values
- A representation (perhaps in memory)

# Type Intuition

You can't do this:

```
int a = 0;
```

```
int * pointer = &a;
```

```
float fraction = 1.2;
```

```
a = pointer + fraction;
```

**... or can you?**

# Components of a type system

Primitive types + operators for building more complex types

- int, bool, void, class, function, struct

Means of determining if types are compatible

- Can disparate types be combined? How?

Rules for inferring type of an expression

# Type Rules

For every operator (including assignment)

- What types can the operand have?
- What type is the result?

Examples

```
double a;
```

```
int b;
```

```
a = b; Legal in Java, C++
```

```
b = a; Legal in C++, not in Java
```

# Type Coercion

Implicit cast from one data type to another

- Float to int

Narrow form: type promotion

- When the destination type can represent the source type
- float to double

# Types of Typing I: When do we check?

## Static typing

- Type checks are made before execution of the program (compile-time)

C Java

## Dynamic typing

- Type checks are made during execution (runtime)

## Combination of the two

- Java (downcasting vs cross-casting)

Java

# Object o = ...  
A o = (A) o

# Example: Casting

## Cross-casting (static check)

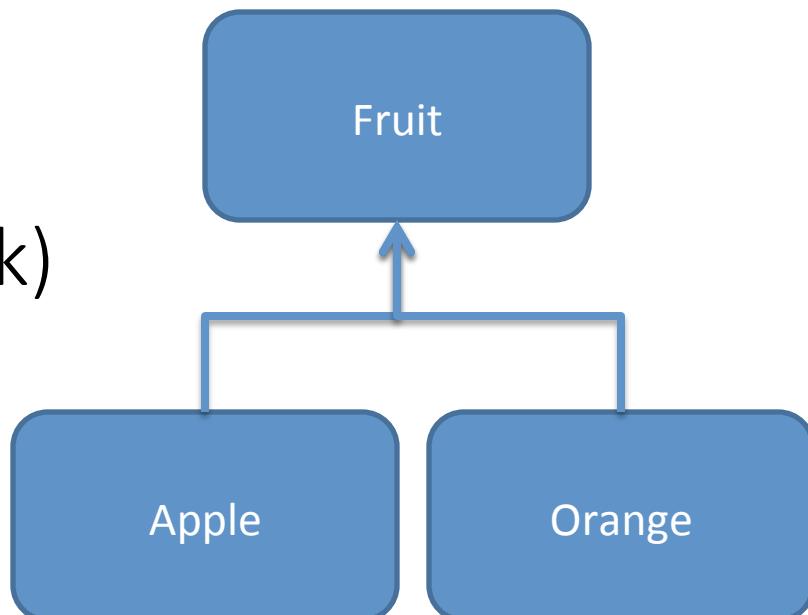
```
Apple a = new Apple();  
Orange o = (Orange)a;
```

cross-casting wont compile in java

## Downcasting (dynamic check)

```
Fruit f = new Apple();  
if ( ... ) { if conditions becomes true, then java  
throws an error  
    f = new Orange();  
}  
  
Apple two = (Apple)f;
```

can compile in java



# Static vs Dynamic Tradeoffs

Static typing is where the type is bound to the variable. Types are checked at compile time.

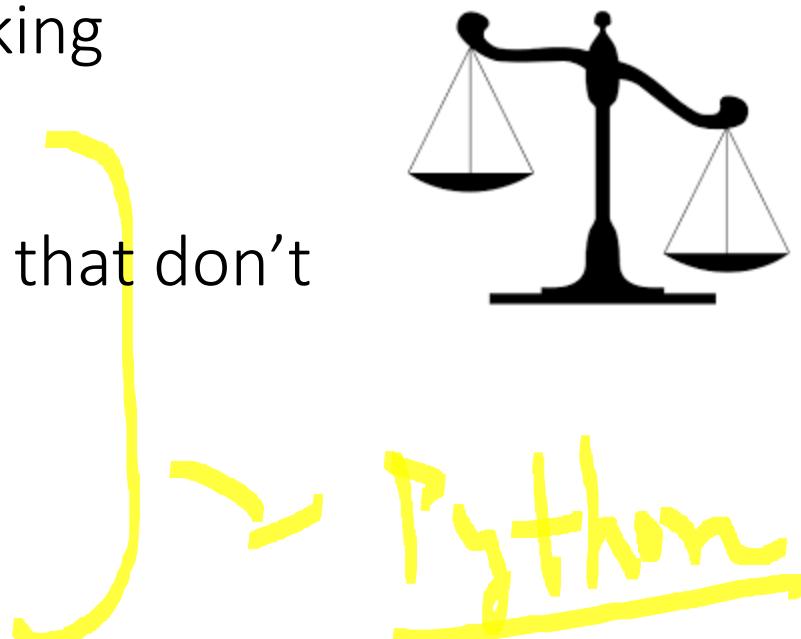
Dynamic typing is where the type is bound to the value. Types are checked at run time.

## Statically typed

- Compile-time optimization
- Compile-time error checking

## Dynamically typed

- Avoid dealing with errors that don't matter
- Some added flexibility
- Runtime failures



# Duck Typing

Type is defined by the methods  
and properties

```
class bird:  
    def quack(): print("quack!")  
  
class mechaBird:  
    def quack(): print("101011...")
```

How do we check?

- Runtime modifications to allow duck typing (Duck punching)



# Types of Typing II: What do we check?

Strong vs weak typing

- Degree to which type checks are performed
- Degree to which type errors are allowed to happen at runtime
- Continuum without precise definitions

# Strong v Weak

No universal definitions but...

- Statically typed is often considered stronger (fewer type errors possible)
- The more implicit casts allowed the weaker the type system
- The fewer checks performed at runtime the weaker

# Strong v Weak Example

C (weaker)

```
union either{  
    int i;  
    float f;  
} u;  
u.i = 12;  
float val = u.f;
```

StandardML (stronger)

```
real(2) + 2.0
```

pointer can convert to any types just by casting

# Fancier types

**Dependent types** can be used to reason about computation

- Reverse takes a list of int of length  $n$  and returns a list of length  $n$

**Resource types** can be used to reason about program complexity

- The program only type-checks if it runs in poly time

Very hard to reason about, but strong guarantees

# Type Safety

## Type safety

- All successful operations must be allowed by the type system
- Java was explicitly designed to be type safe
  - If you have a variable with some type, it is guaranteed to be of that type
- C is not
- C++ is a little better

**Computer scientist Ross Tate working to tame Java 'wildcards'**

*By Bill Steele*

A Cornell computer scientist has just discovered that the Java computer language, designed to be safe, is not so safe after all, and now he is working to find a solution.

# Type Safety Violations

C

- Format specifier

```
printf("%s", 1);
```

- Memory safety

```
struct big{  
    int a[100000];  
};  
struct big * b = malloc(1);
```

C++

- **Unchecked casts**

```
class T1{ char a};  
class T2{ int b; };  
int main{  
    T1 * myT1 = new T1();  
    T2 * myT2 = new T2();  
    myT1 = (T1*)myT2;  
}
```

# Type System for Our Language

# Our type system

Primitive types

- int, bool, string, void

Type constructors

- struct

Coercion

- bool cannot be used as an int in our language (nor vice-versa)

# Our Type Errors I

Arithmetic operators must have **int** operands

Equality operators **==** and **!=**

- Operands must have same type
- Can't be applied to
  - Functions (but CAN be applied to function results)
  - struct name
  - struct variables

Other relational operators must have **int** operands

Logical operators must have **bool** operands

# Our Type Errors II

## Assignment operator

- Must have operands of the same type
- Can't be applied to
  - Functions (but CAN be applied to function results)
  - struct name
  - struct variables

For `cin >> x;`

- x cannot be function struct name, struct variable

For `cout << x;`

- x cannot be function struct name, struct variable

Condition of if, while must be boolean

# Our Type Errors III

Invoking (aka calling) something that's not a function

Invoking a function with

- Wrong number of args
- Wrong types of args
  - Also will not allow struct or functions as args

Returning a value from a void function

Not returning a value in a non-void function

Returning wrong type of value in a non-void function

# Type Checking

Structurally similar to nameAnalysis

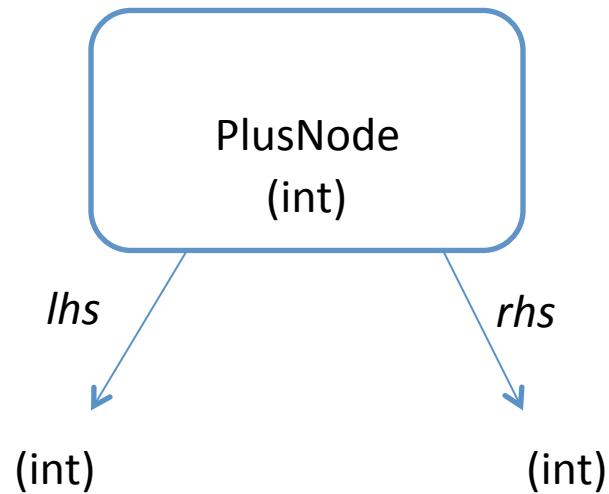
- Historically, intermingled with nameAnalysis and done as part of attribute “decoration”

Add a typeCheck method to AST nodes

- Recursively walk the AST checking types of sub-expressions
- Let’s look at a couple of examples

# Type Checking: Binary Operator

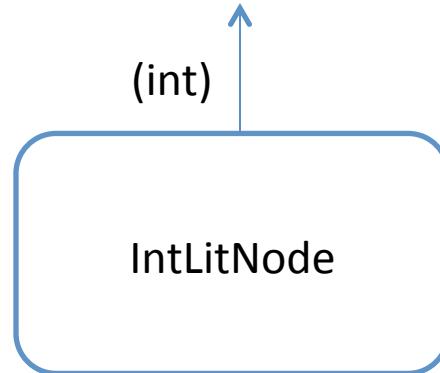
- Get the type of the LHS
- Get the type of the RHS
- Check that the types are compatible for the operator
- Set the *kind* of the node be a value
- Set the *type* of the node to be the type of the operation's result



# Type “Checking”: Literal

Cannot be wrong

- Just pass the type of the literal up the tree

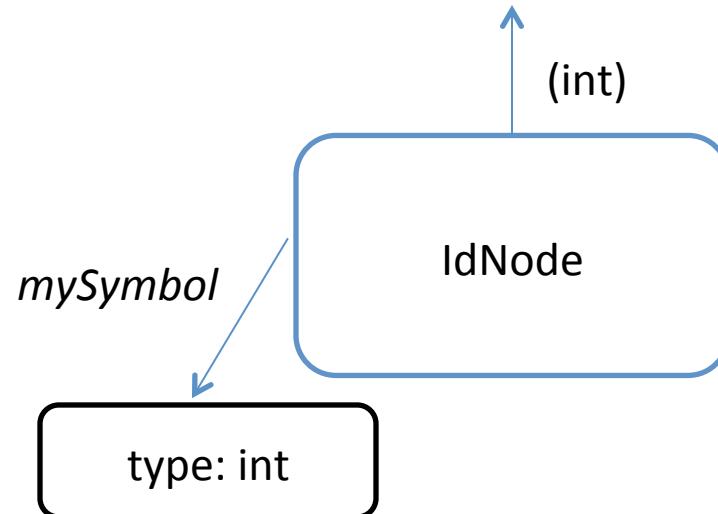


# Type Checking: IdNode

Look up the type of the declaration

- There should be a symbol “linked” to the node

Pass symbol type up the tree



# Type Checking: Others

Other node types follow these same principles

- Function calls
  - Get type of each actual argument
  - Match against the formal argument (check symbol)
  - Send the return type up the tree
- Statement
  - No type

# Type Checking: Errors

We'd like all *distinct* errors at the same time

- Don't give up at the first error
- Don't report the same error multiple times

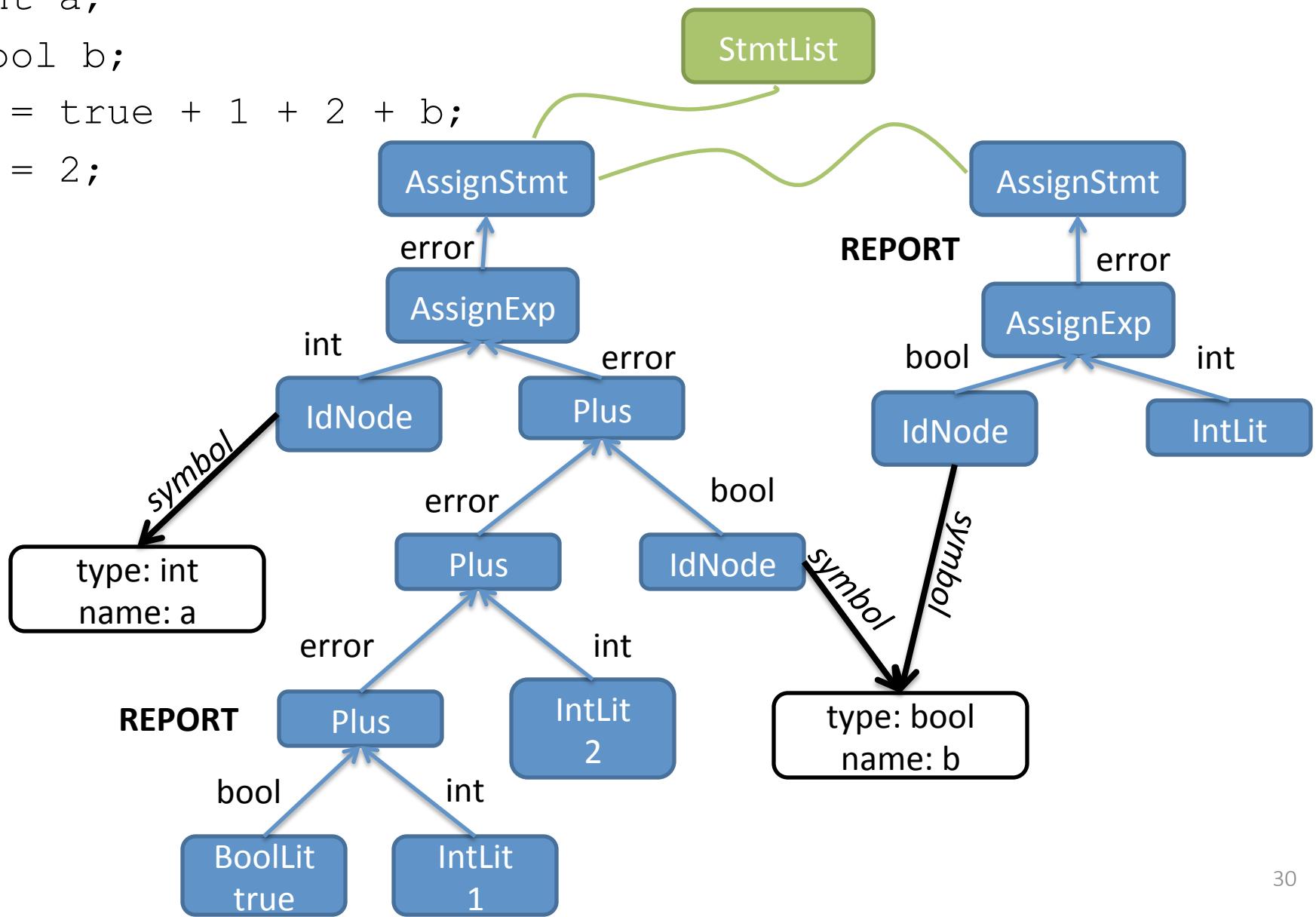
Introduce an internal **error** type

- When type incompatibility is discovered
  - Report the error
  - Pass **error** up the tree
- When you get error as an operand
  - Don't (re)report an error
  - Again, pass **error** up the tree



# Error Example

```
int a;
bool b;
a = true + 1 + 2 + b;
b = 2;
```



# Looking Towards Next Lecture

Look at data (and therefore types) is represented in the machine

Start very abstract, won't talk about an actual architecture for awhile

Assembly has no intrinsic notion of types. We'll have to add code for type checking ourselves

# Runtime Environments

# Roadmap

## Type checking

- Went through a couple of type system design points
- Inferred the types of expressions in our language
- Showed how to propagate type errors

## Today

- Begin looking at how to lower code down to assembly

# Outline

Talk about what a runtime environment is

Discuss the “semantic gap”

- The difference between level of abstraction in source code and executables

How memory is laid out in an abstract machine

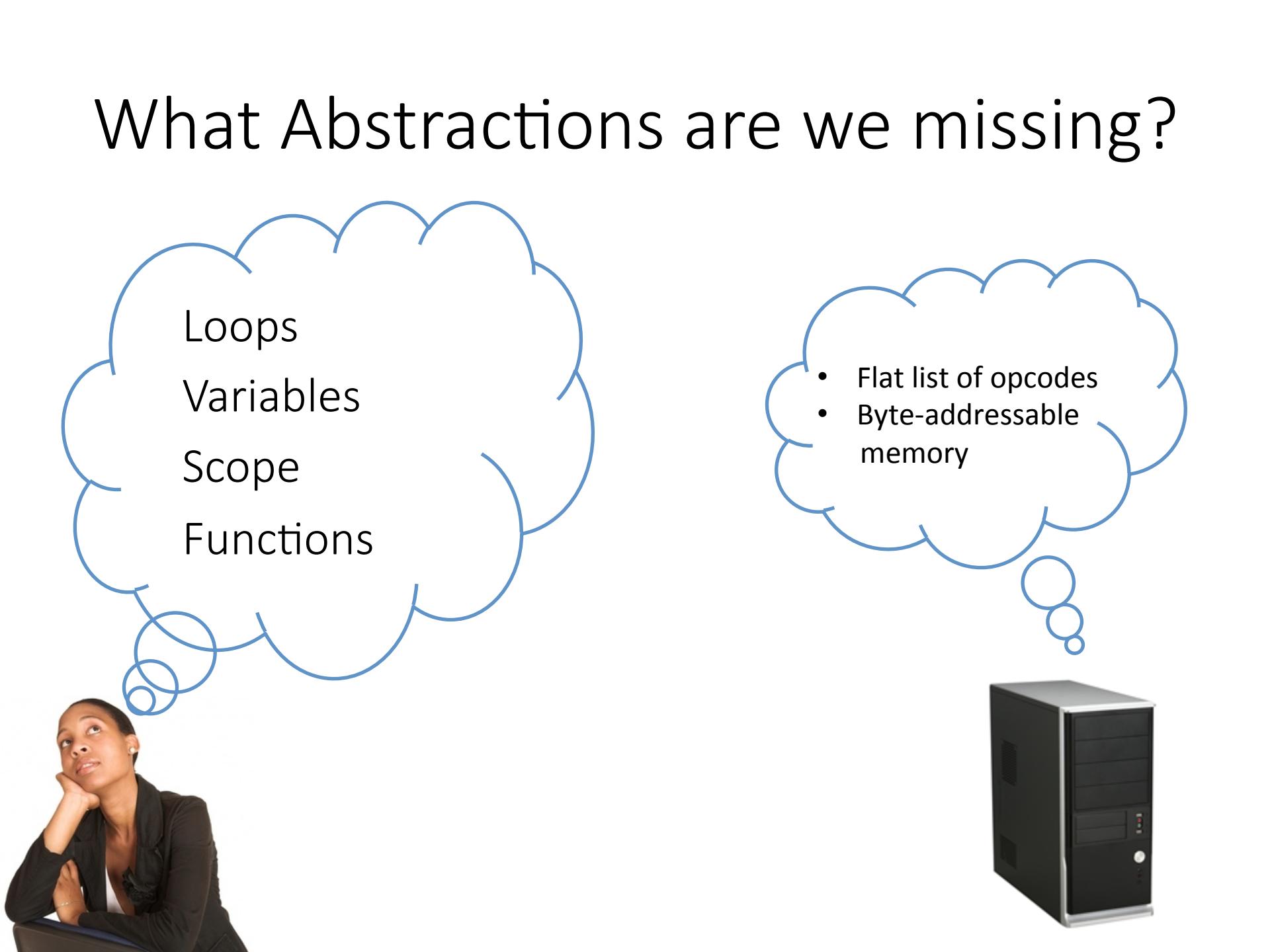
# WYSINWYX

What You See (in source code) Is Not What You eXecute

- We think in terms of high-level abstractions
- Many of these abstractions have no explicit representation in machine code



# What Abstractions are we missing?



Loops  
Variables  
Scope  
Functions

- Flat list of opcodes
- Byte-addressable memory



# Runtime Environment

Underlying software and hardware configuration assumed by the program

- May include an OS (may not!)
- May include a virtual machine

# The Role of the Operating System

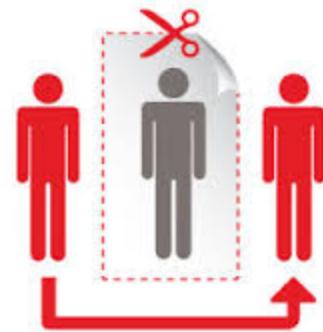
Program piggybacks on the OS

- Provides functions to access hardware
- Provides illusion of uniqueness
- Enforces some boundaries on what is allowed

# Mediation is Slow

It's up to the compiler to use the runtime environment as best it can

- Limited number of very fast registers with which to do computation
- Comparatively large region of memory to hold data
- Some basic instructions from which to build more complex behaviors



# Conventions

Assembly code enforces very few rules

- We'll have to structure the way we access memory ourselves

These conventions help to guarantee that isolated code can work together

- Allows modularity
- Increase efficiency



# Issues to consider

## Variables

- How do we store them?
- How do we access them?

## Functions as straight-line code

- How do we simulate function calls?
- How do we simulate function entry?
- How do we simulate function return?

# General Memory Layout

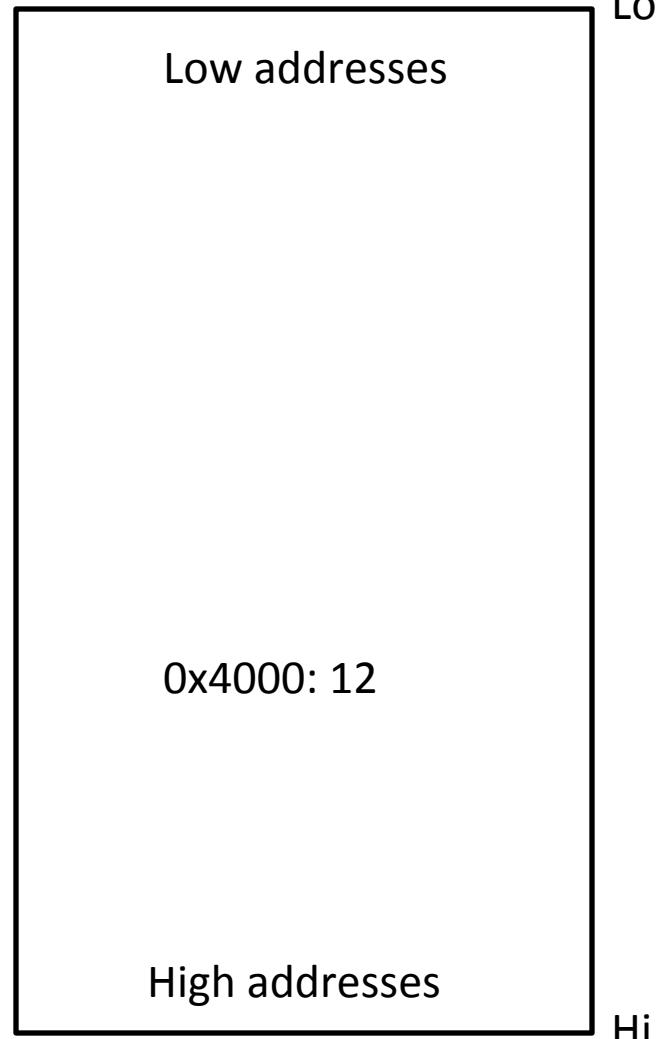
We can think of program memory as a single array

Addressable via memory cell

- Represent using a hex value

Very common to represent program memory as a “tower”

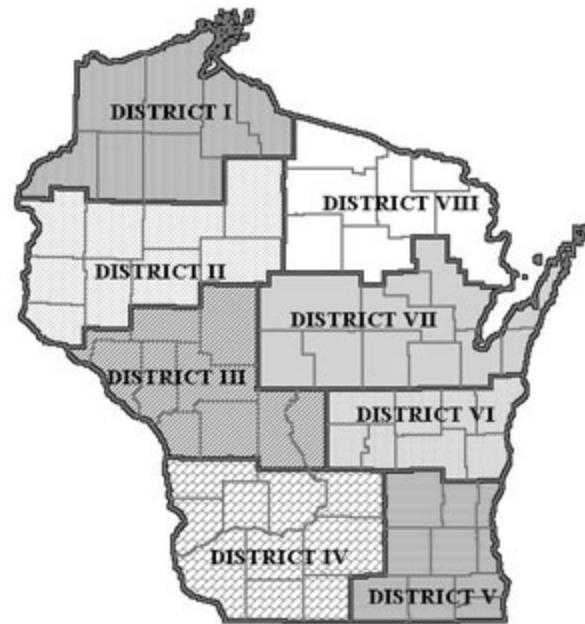
- Low addresses at the “top”
- High addresses at the “bottom”



# How do we divide up memory?

## Goals

- Flexibility
- Efficiency
- Speed



# Memory Layout : Static Allocation

Region for global memory

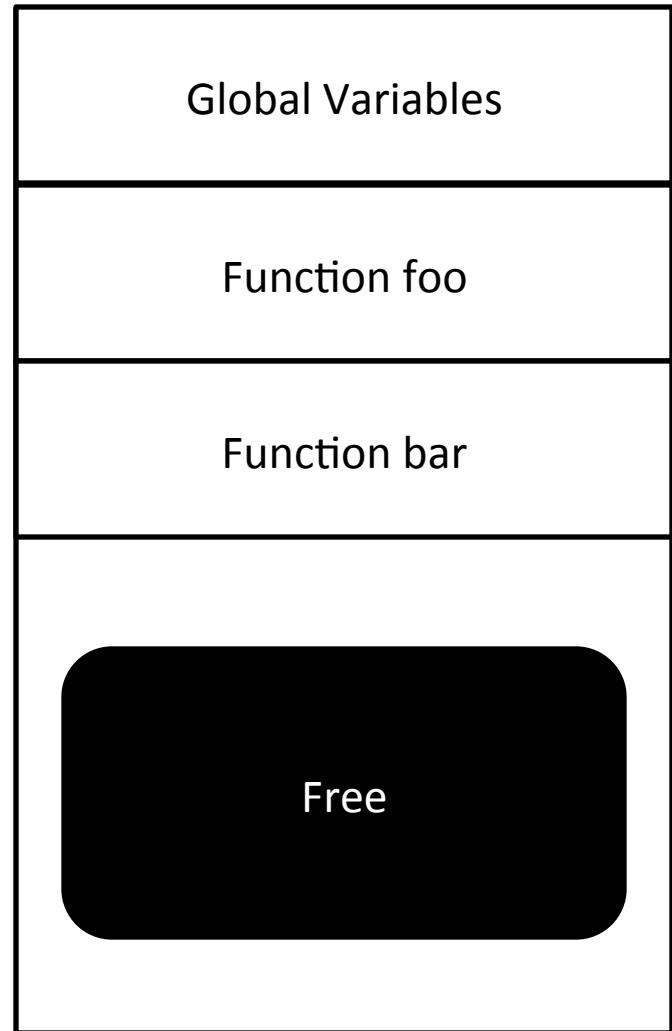
1 “frame” for each subroutine of the program

- Memory “slot” for each local, param
- “slot” for caller

Fast but

- Any drawbacks?

doesn't support recursion;  
static allocation only works if we run our programs using a finite amount of space



# Memory Layout: The Stack

we use this layout instead

Keep the function frame idea, but allocate per invocation

- AKA activation records
- We don't statically know how many frames we might have
- Fix a point in memory grow from there



# A Closer look at Activation Records (ARs)

Push a new frame on function entry

Pop the frame on function exit

To keep size down, we can put static data in the global area

- In particular, strings

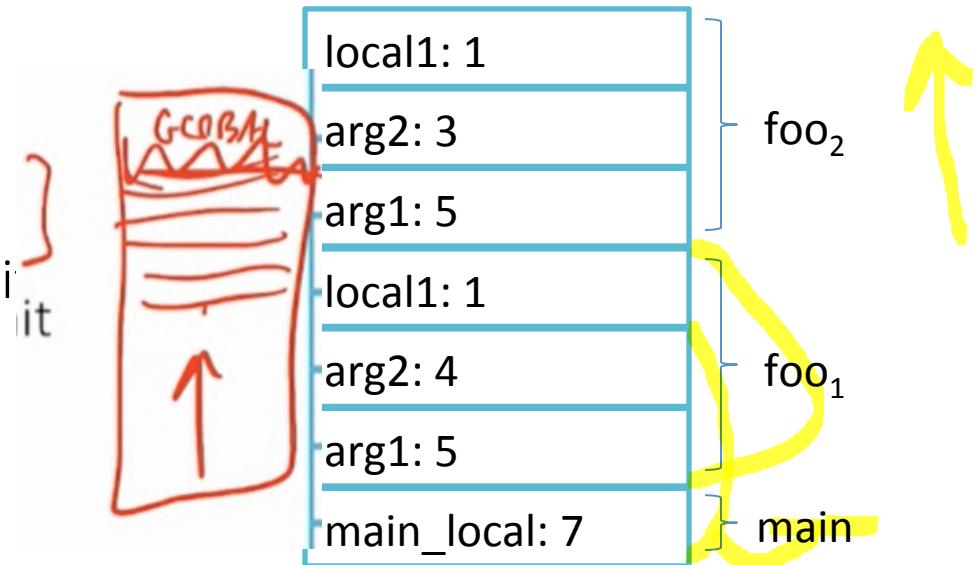
Allows conceptually infinite recursion depth

- In practice, we'll eventually hit the global data

stackoverflow occurs

```
foo(int arg1, int arg2){  
    int local1 = arg1 - arg2;  
    if (local1 > 0) { foo( arg1, 3); }  
}  
main(){  
    int main_local = 7;  
    foo(5, 4);  
}
```

**Disclaimer:**  
High-level idea only



# Activation Records: Dynamic Locals

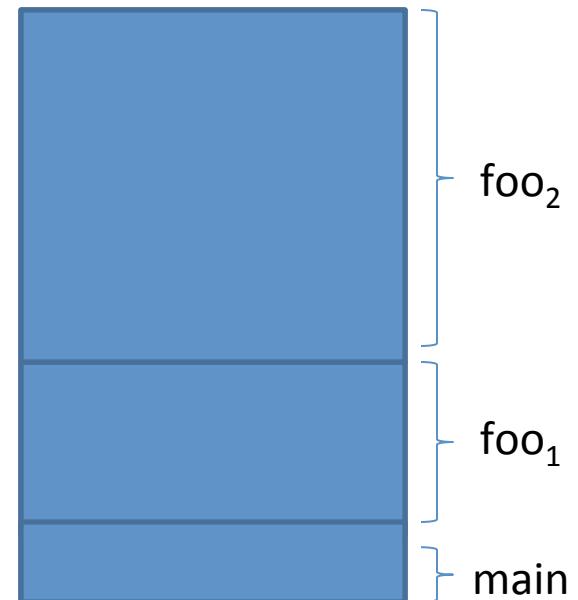
The stack can handle local variables whose size is unknown

- Grow the frame as needed during its execution

This means stack size is unknown at compile time!

- Store the previous frame's boundaries in the current frame

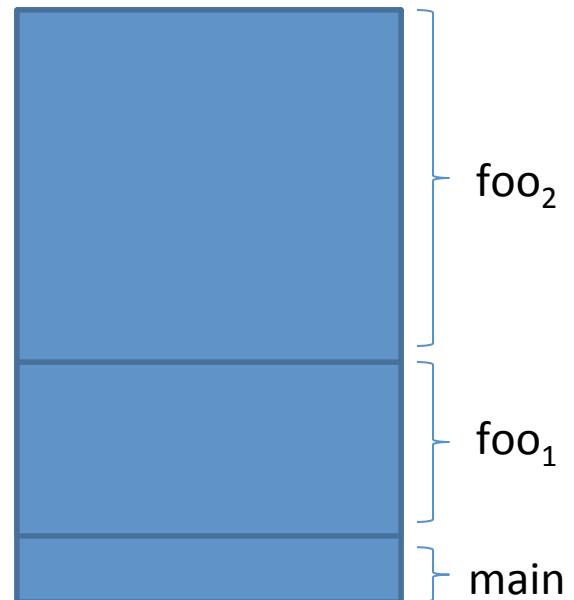
```
foo(int arg) {  
    int locArr[arg];  
    ...  
    foo(arg * 2);  
}  
main(int argc, char * argv[]) {  
    int main_local = 7;  
    foo(argc);  
}
```



# Activation Record: Summary

## Things in the frame

- Local variable values
- Space for the caller's frame
  - Data context
    - Enough info to remember the boundaries of the frame we called from (the caller)
  - Control context
    - Enough info to know what line of code we were at when we made the call



# Non-Local Dynamic Memory

Surely we don't want *all* data allocated in a function call to disappear on return

Don't know how much space we'll need

- Can allocate many such objects
- Can be sized dynamically

```
public makeList() {  
    Node n = new Node();  
    Node t = new Node();  
    n.next = t;  
    return n;  
}
```

# The Heap

Region of memory  
independent of the  
stack

Allocate at program's  
command

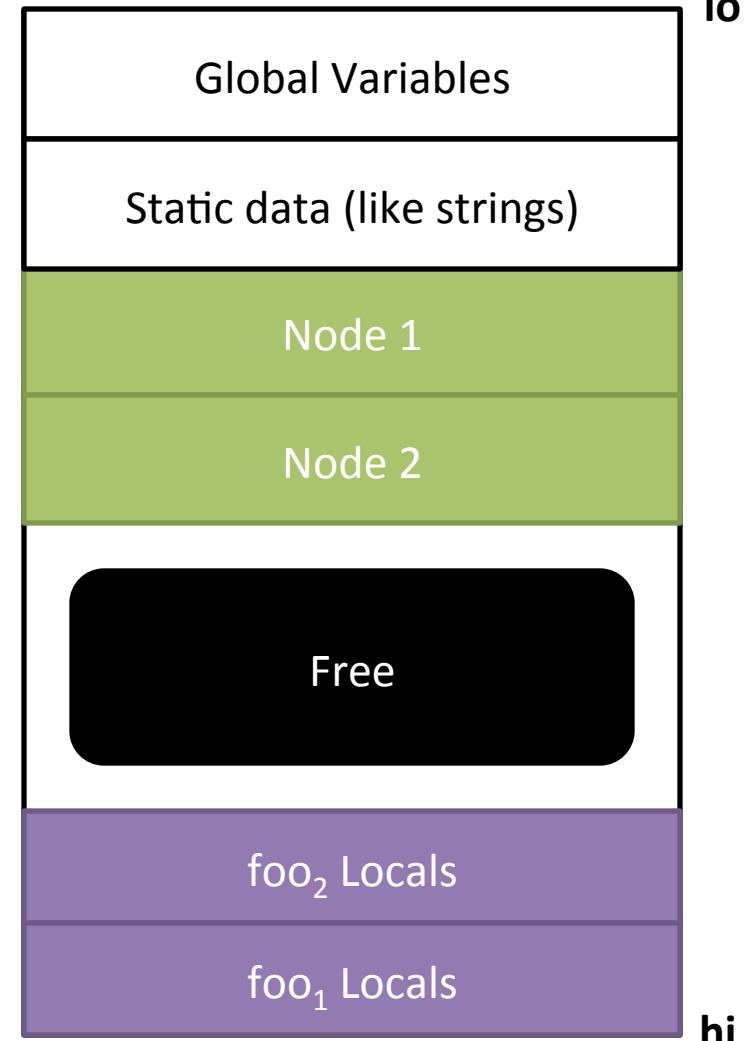
How do we get rid of it?

- Ask programmer to specify when it's unused
- Can track automatically when it's unused JAVA

Heap grows  
towards high  
memory

c

Stack grows  
towards low  
memory



# Function Calls

Where convention meets implementation

- Function calls are so common that their semantics are partially encoded into architecture
- Registers often have “nicknames” that hint at their purpose in representing ARs
- Some instructions implement “shortcuts” for building up and breaking down ARs



# When are we “in” a function?

register id

\$ip the *instruction pointer* tracks the line of code we are executing. It tracks “where we are at” in the program

If the instruction pointer points to code that was generated for some function, we’ll say we’re in that function

```
#1 int summation(int max) {  
#2     int sum = 1;  
#3     for (int k = 1 ; k <=  
max ; k++) {  
#4         sum += k;  
#5     }  
#6     return sum;  
#7 }  
#8 void main(){  
#9     int x = summation(4);  
#10    cout << x;  
#11 }
```

\$ip: #2

# Caller / Callee relationship

## Caller

- The function doing the invocation

## Callee

- The function being invoked

Note that this is a per-call relationship

- main is the caller at line 5
- v is the callee at line 5



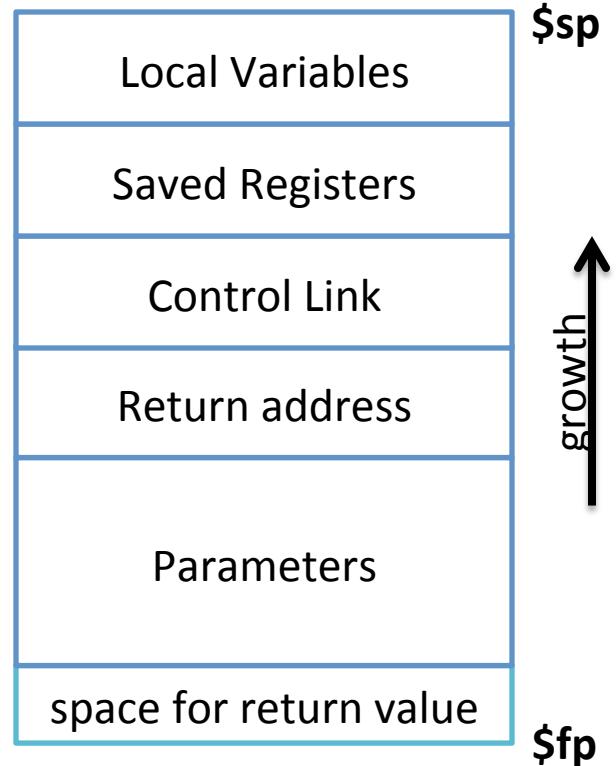
```
1. void v() {  
2. }  
3.  
4. int main() {  
$ip → 5.      v();  
6. }
```

# How ARs are *Actually* Implemented

Two registers track the stack

- Frame pointer (\$fp) tracks the base of the frame
- Stack pointer (\$sp) tracks the top of the stack

Low memory addresses



High memory addresses

# Function Entry: Caller Responsibilities

Low memory addresses

Store the *caller-saved* registers in it's own AR

Set up actual params

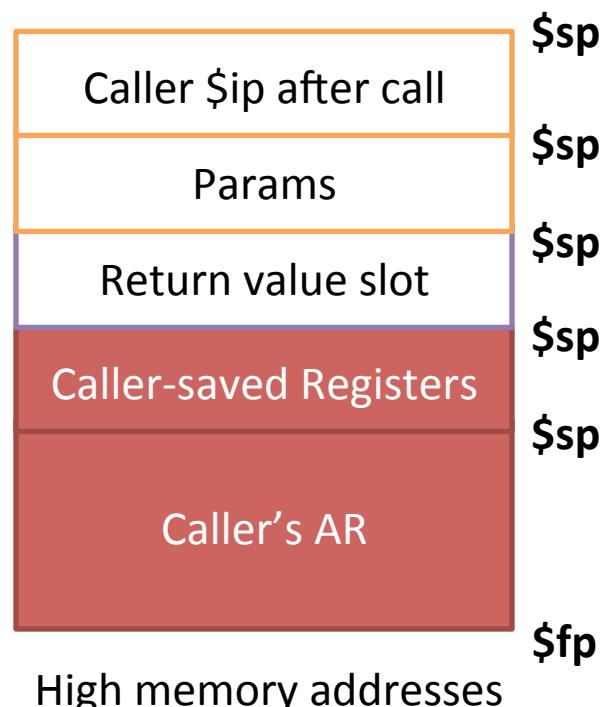
- Set aside a slot for the return value
- Push parameters onto the stack

Copy return address out of \$ip

- It's about to get obliterated

Jump to the Callee's first instruction

\$ip      Callee entry



# Function Entry: Callee Responsibilities

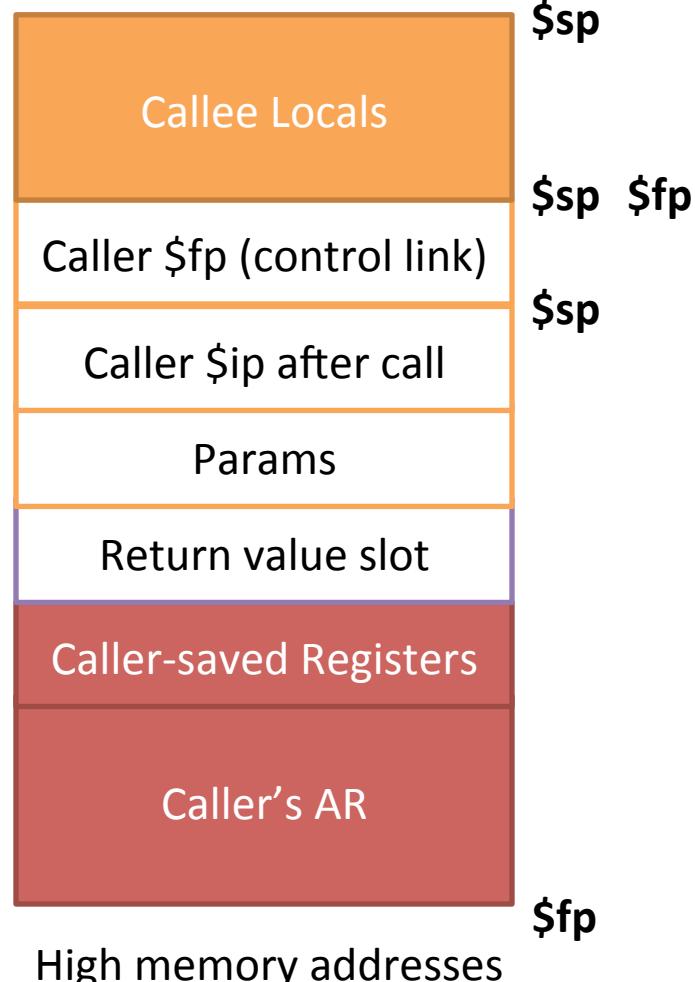
Save \$fp since we need to restore it later

Update the base of the new AR to be to end of the old AR

Save *callee-saved* registers if necessary

Make space for locals

Low memory addresses



# Function Exit: Callee Responsibilities

Set the return value

Restore callee-saved registers

Grab stored return address

Restore *old \$sp*: fixed (negative)  
offset from the current base of the  
stack

Restore *old \$fp*: also from stack

Jump to the stored return address

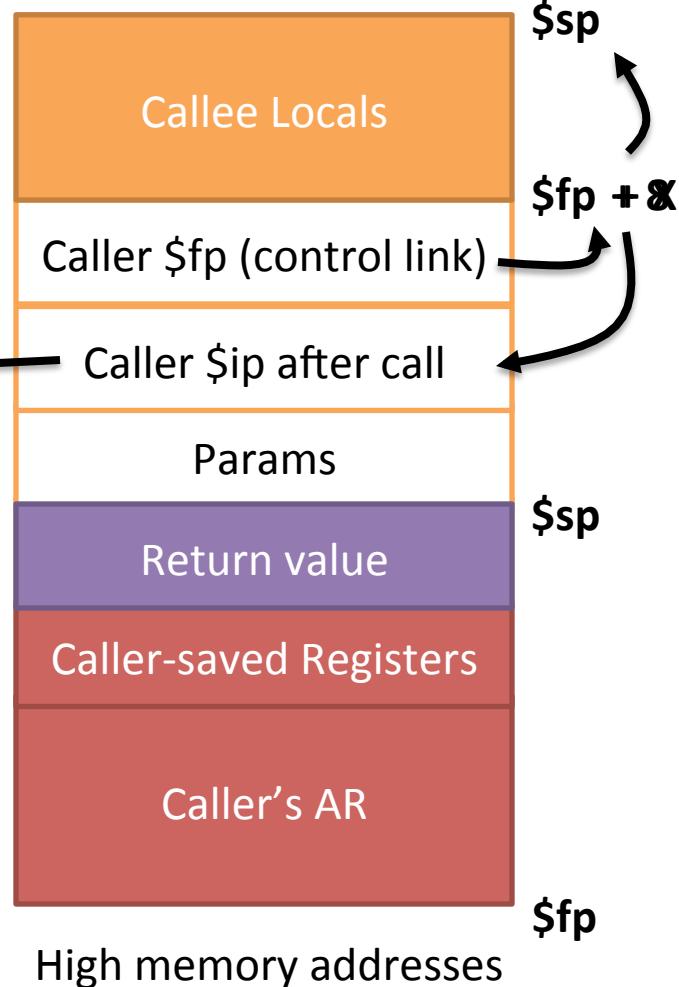
\$ra

After Call site

\$ip

After Call site

Low memory addresses



# Function Exit: Caller Responsibilities

Grab the return value (pop or copy from register)

Restore caller-saved Registers

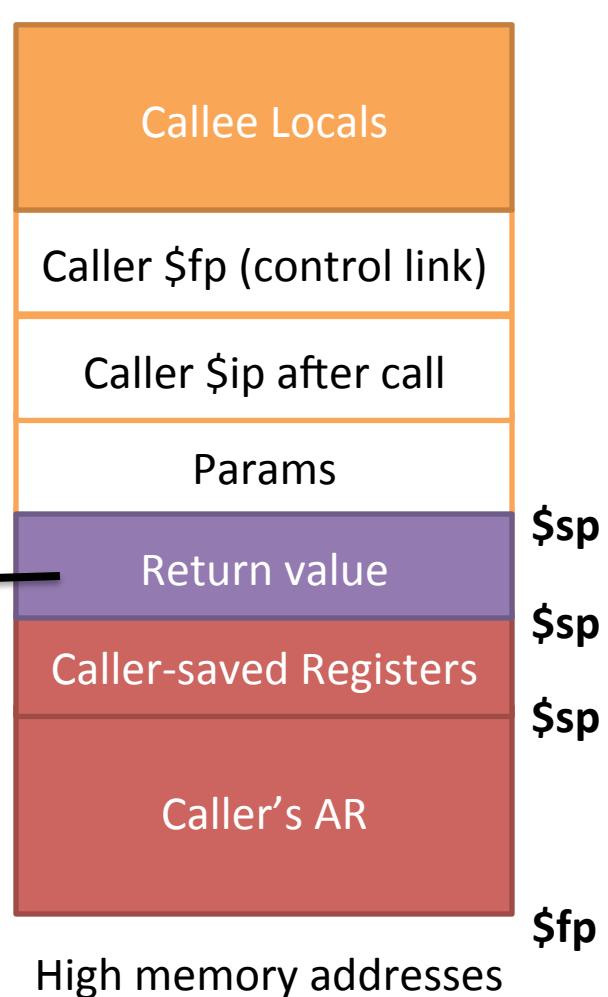
Low memory addresses

\$2

Return val

\$ip

After Call site



# Example

```
#1 int summation(int max) {  
#2     int sum = 1;  
#3     for (int k = 1 ; k <= max ; k++) {  
#4         sum += k;  
#5     }  
#6     return sum;  
#7 }  
#8 void main() {  
#9     int x = summation(4);  
#10    cout << x;  
#11 }
```

# Hardware Support for Functions

## Calls

- JAL (Jump and Link): MIPS instruction that puts **\$ip** in **\$ra** then, sets **\$ip** to a given address
- Call: x86 instruction that pushes **\$ip** directly onto the stack, then sets **\$ip** to given address

## Return

- JR (Jump Return): MIPS instruction that sets **\$ip** to **\$ra**
- ret: x86 instruction that pops directly off the stack into **\$ip** SPARC “Sliding Windows”
- Crazy system where caller registers are automatically saved, new set of callee saved registers automatically exposed

# Next Time

## MIPS

- We will fix a concrete runtime environment, not just a pseudo-code machine

## Variable access

- We've shown how to store variables
- How do we actually access them?
  - What about scope?

# Parameter Passing

# Roadmap

Last time

- Discussed runtime environments
- Described some conventions for assembly
  - Functions via stack
  - Dynamic memory via a heap

Today

- Propagating values from one function to another

# Outline

## Parameter Passing

- Different styles
- What they mean
- How they look on the stack

# Vocabulary

Define a couple of terms  
that are helpful to talk  
about parameters

We've already obliquely  
talked about some of  
these



# L- and R- Values

## L-Value

- A value with a place of storage

## R-Value

- A value that may not have storage

```
b = 2;  
a = 1;  
a = b+b;
```

# Memory references

## Pointer

- A variable whose value is a memory address

## Aliasing

- When two or more variables hold same address

# Parameter Passing

In definition:

```
void v(int a, int b, bool c) { ... }
```

- Terms
  - Formals / formal parameters / parameters

In call:

```
v(a+b,8,true);
```

- Terms
  - Actuals / actual parameters / arguments



# Types of Parameter Passing

We'll talk about 4 different varieties

- Some of these are more used than others
- Each has it's own advantages / uses

# Pass by Value

On function call

- *Values* of actuals are copied into the formals
- C and java always pass by value

```
void fun(int a) {  
    a = 1;  
}  
void main() {  
    int i = 0;  
    fun(i);  
    print(i);  
}
```

# Pass by Reference

On function call

- The address of the actuals  
are *implicitly* copied

```
void fun(int a) {  
    a = 1;  
}  
  
void main() {  
    int i = 0;  
    fun(i);  
    print(i);  
}
```

# Language Examples

Pass by value

- C and Java

Pass by reference

- Allowed in C++ and Pascal

# Wait, Java is Pass by Value?

All non-primitive L-values are pointers

```
void fun(int a, Point p) {  
    a = 1;  
    p.x = 5;  
}  
void main() {  
    int i = 0;  
    Point k = new Point(1, 2);  
    fun(i, k);  
}
```

# Java – pass by value

```
public static void main( String[] args ) {
    Dog aDog = new Dog("Max");
    foo(aDog);

    if (aDog.getName().equals("Max")) {
        System.out.println( "Java passes by value." );
    } else if (aDog.getName().equals("Fifi")) {
        System.out.println( "Java passes by reference." );
    }
}

public static void foo(Dog d) {
    d.getName().equals("Max");
    d = new Dog("Fifi");
    d.getName().equals("Fifi");
}
```

Demo

# Pass by Value-Result

When function is called

- Value of actual is passed

When function returns

- Final values are copied back to the actuals

Used by Fortran IV, Ada

- As the language examples show, not very modern

# Pass by Value-Result – Example 1

```
int x = 1;           // a global variable

void f(int & a)
{
    a = 2;
    // when f is called from main, a and x are aliases
    x = 0;
}

main()
{
    f(x);
    cout << x;    // 0 with call by ref, 2 with call by
                   // value-result
}
```

# Pass by Value-Result – Example 2

```
void f(int &a, int &b)
{
    a = 2;
    b = 4;
}

main()
{
    int x;
    f(x, x);
    cout << x; // Undefined different output
with
                // different compilers
}
```

# Pass by Name

Conceptually works as follows:

- When a function is called
  - Body of the callee is **rewritten** with the **text** of the argument
- Like macros in C / C++

# Call-by-need / lazy evaluation

```
int f(x, y)  
{ return x+y; }
```

```
main()  
{  
    int x = f(5, 6); //x=5+6  
    cout << x;          //x is now  
                        evaluated  
}
```

# Implementing parameter passing

Let's talk about how this actually is going to work in memory



# Let's draw out the memory

```
int g;  
void f (int x, int y, int z) {  
    x = 3; y = 4; z = y;  
}
```

```
void main () {  
    int a = 1, b = 2, c = 3;  
    f(a,b,c);  
    f(a+b, 7, 8);  
}
```

Consider pass-by-value and  
pass-by reference

# Bad use of R-Values

Can prevent programs that are valid in pass by value from working in pass by reference

- Literals (for example) do not have locations in memory

We will rely on the type checker to catch bad use of R-values

# Let's draw out the memory again

```
int g;  
void f(int x, int y, int z) {  
    x = 3; y = 4; z = y;  
}
```

```
void main() {  
    int a = 1, b = 2, c = 3;  
    f(a, b, g);  
    f(a+b, 7, 8);  
}
```

Consider pass by value-result  
and pass by name

# Efficiency Considerations

## Pass by Value

- Copy values into AR (slow)
- Access storage directly in function (fast)

## Pass by Reference

- Copy address into AR (fast)
- Access storage via indirection (slow)

## Pass by Value-result

- Strictly slower than pass by value
- Also need to know where to copy locations back

# Object Handling

```
void alter(Point pt, Position  
pos){  
    pos = pt.p;  
    pos.x++;  
    pos.y++;  
}  
  
void main(){  
    Position loc;  
    Point dot;  
    // ... initialize loc with  
    // x=1,y=2  
    // ... initialize dot with loc  
    alter(dot, loc);  
}
```

```
class Point{  
    Position p;  
}
```

```
class Position{  
    int x, y;  
}
```

In java, loc and dot are pointers to objects (on the heap)

In C++, loc and dot are objects with no indirection (on the stack)

# Roadmap

We learned about parameter passing

- By-value, by-reference, by-value-result, by-name
- How values traverse the stack

Next time

- Allocating variables

# Runtime Access to Variables

# Roadmap

Last Time

- Parameter passing strategies

This time

- How do we deal with variables and scope
- How do we store variables on stack

# Scope

We mostly worry about 3 flavors

- Local
  - Declared and used in the same function
  - Further divided into “block” scope in your language
- Global
  - Declared at the outermost level of the program
- Non-local
  - For static scope: variables declared in an outer nested sub-program
  - For dynamic scope: variables declared in the calling context

# Local variables: Examples

What are the local variables here?

```
int fun(int a, int b) {  
    int c;                      c is local to the function  
    c = 1;  
    if (a == 0) {  
        int d;  ← d is local to the if statement  
        d = 4;  
    }  
}
```

# How do we access the Stack?

Need a little MIPS knowledge

- Full tutorial next week
- General anatomy of a MIPS instruction

opcode   Operand1   Operand2

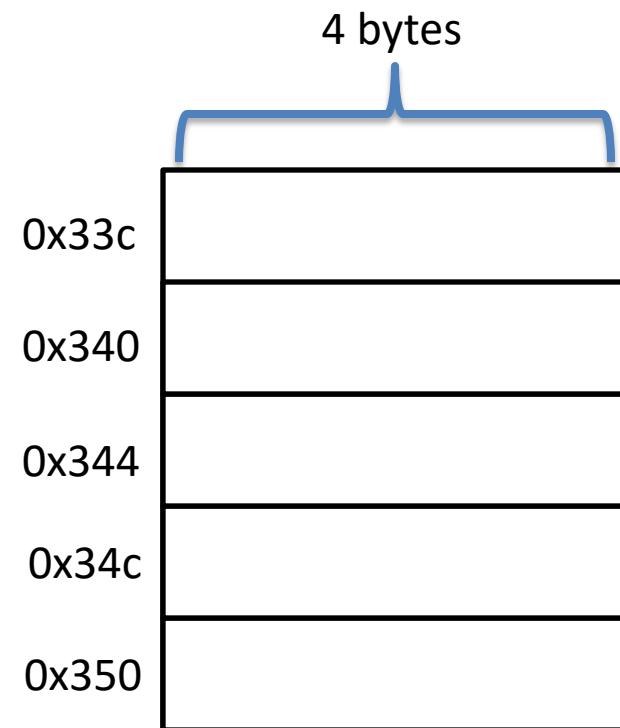
# How do we access the Stack?

read

Use “load” and  
“store” instructions

write

- Recall that every memory cell has an address
- Calculate that memory address, then move data from/to that address



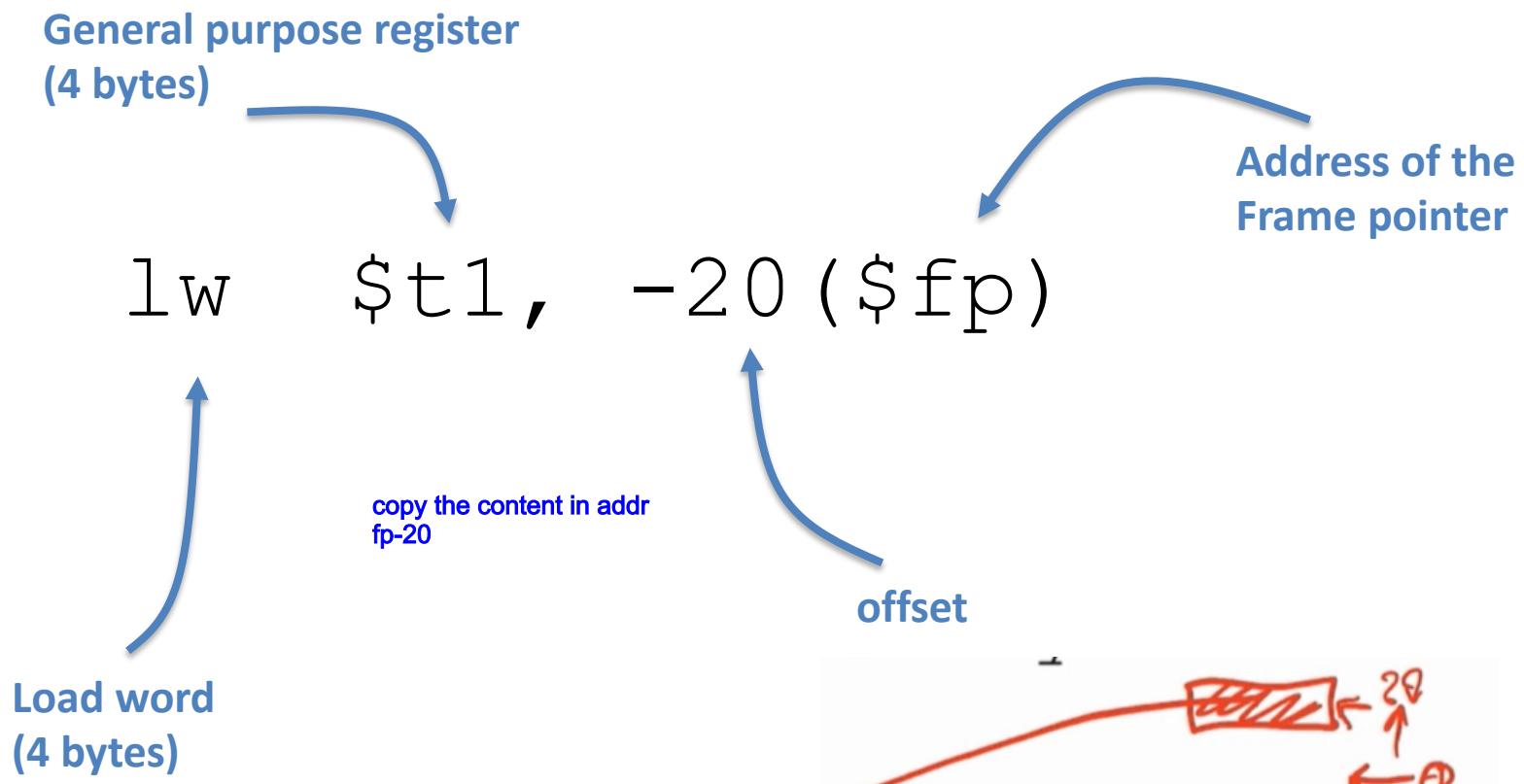
# Basic memory operations

lw register memoryAddress

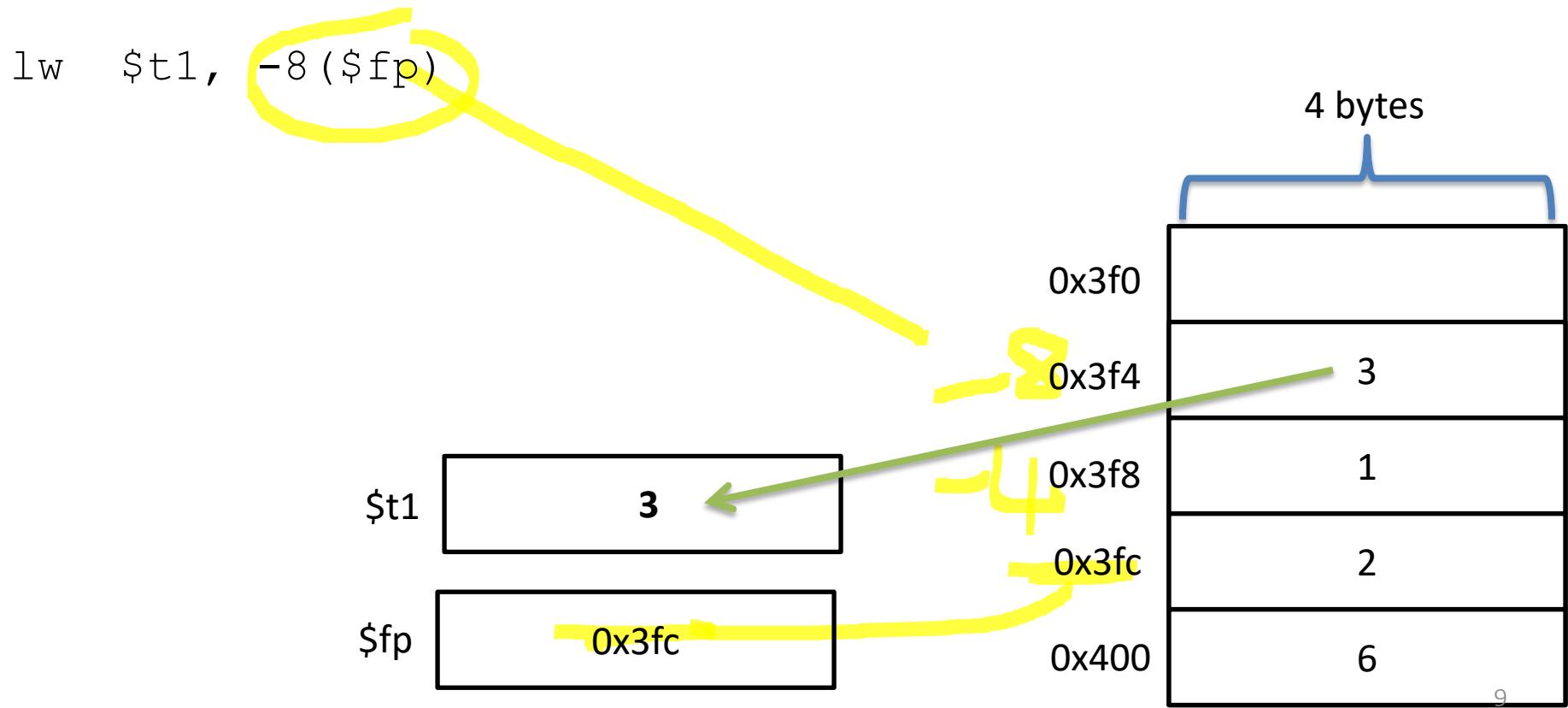
sw register memoryAddress

# Load Word Example

opcode register memoryAddress

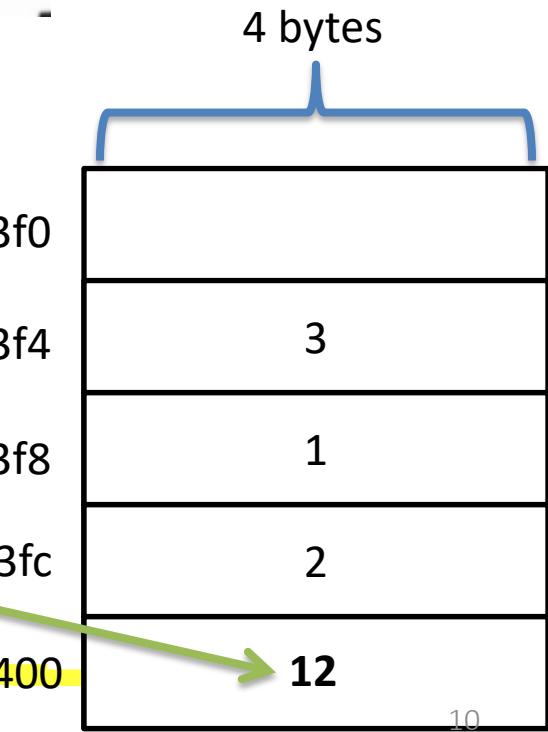
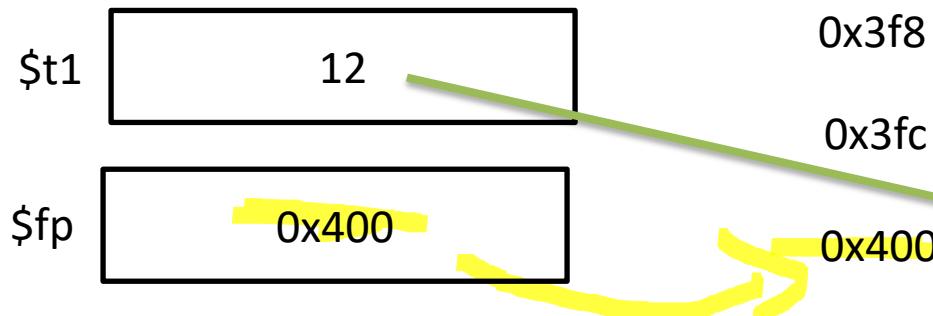
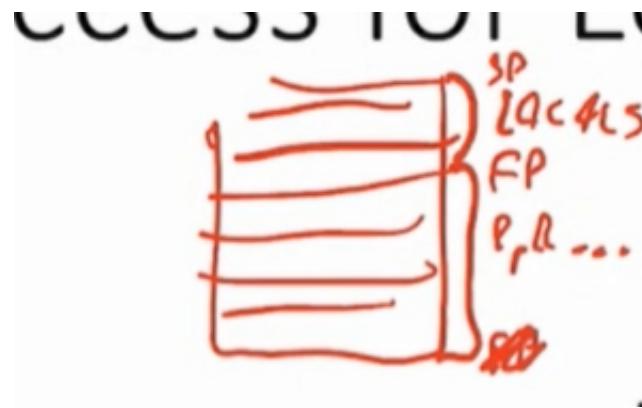


# Load Word in Action



# Store Word in Action

sw \$t1, 0(\$fp)

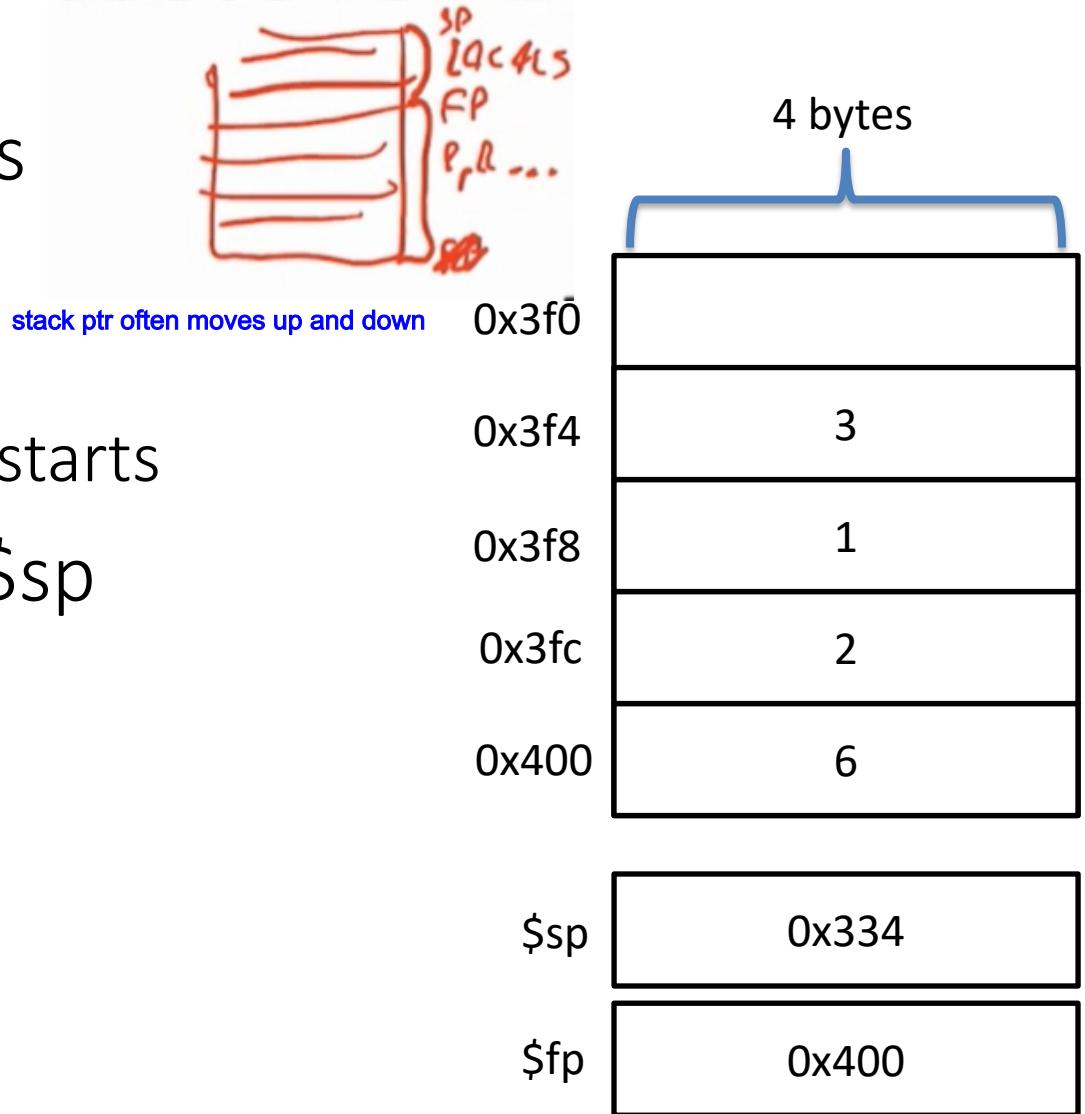


# Relative Access for Locals

Why do we access locals from \$fp?

- That's where the activation record starts

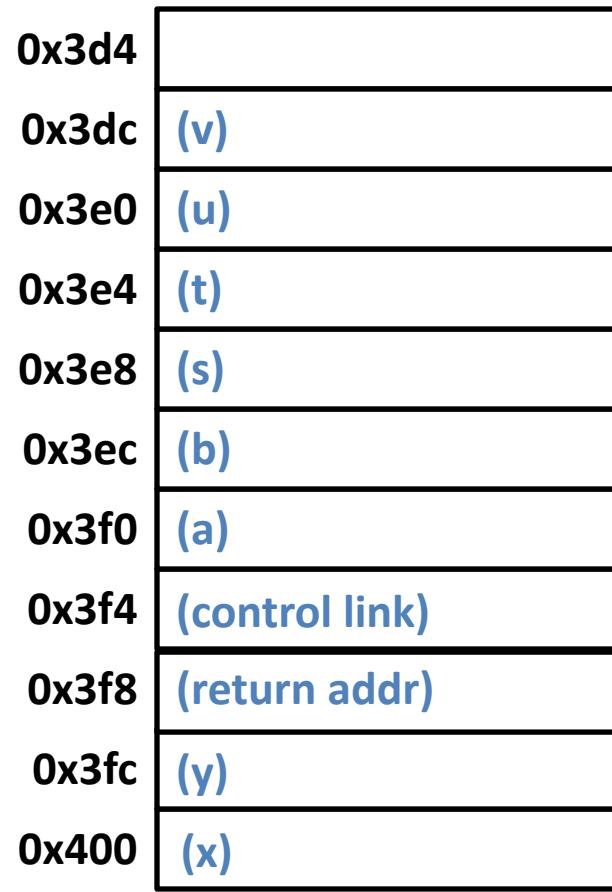
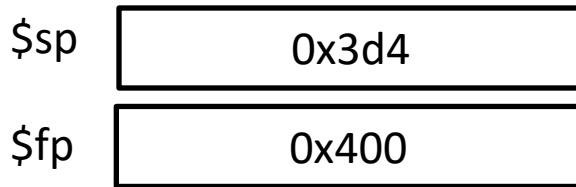
What if we used \$sp instead?



# Simple Memory-Allocation Scheme

Reserve a slot for each variable in the function

```
int test (int x, int y) {  
    int a, b;  
    if (x) {  
        int s;  
    } else {  
        int t, u, v;  
        u = b + y;  
    }  
}
```



# Simple Memory-Allocation Algorithm

**For each function**

Set offset = 0

for each parameter

    add name to symbol table

    offset -= size of parameter

offset -= size of return address

offset -= size of control link

offset -= size of callee saved registers

for each local

    add name to symbol table

    offset -= size of variable

# Simple Memory-Allocation Implementation

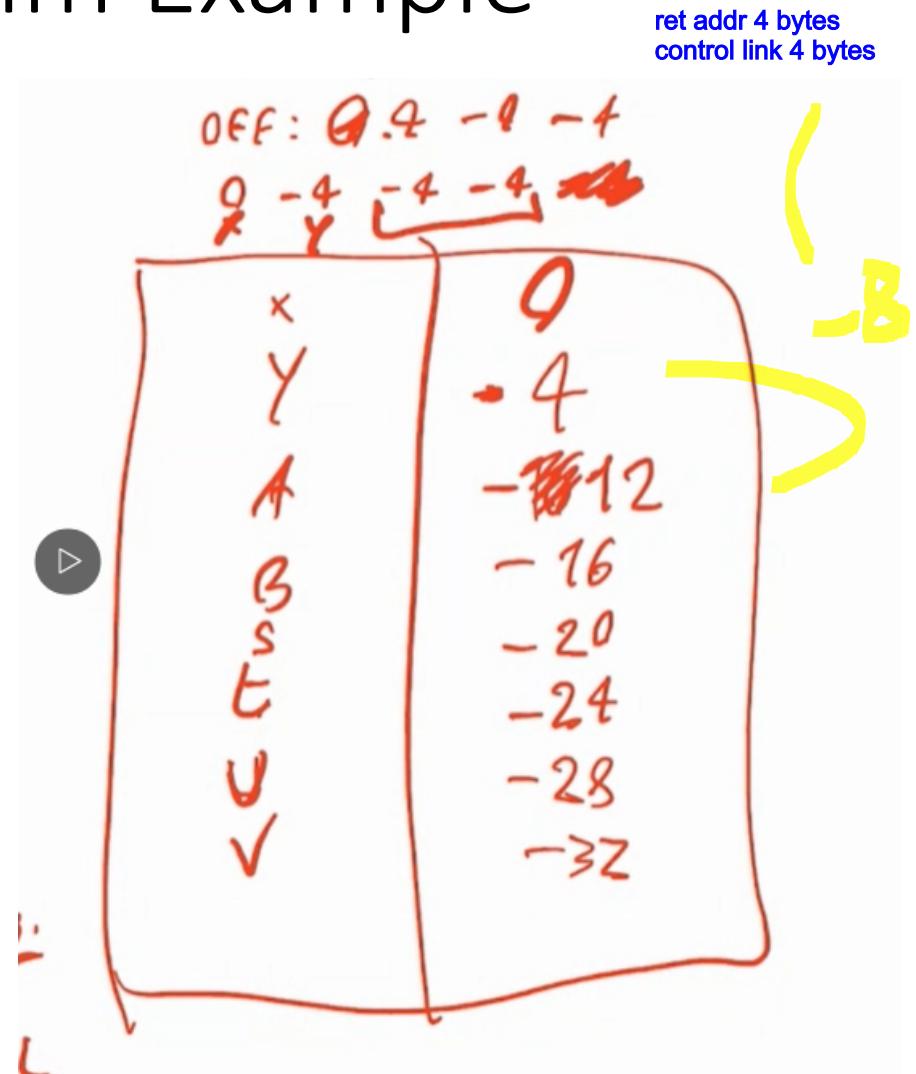
Add an offset field to each symbol table entry

During name analysis, add the offset along with the name (Wait until Project 6 to do this)

Walk the AST performing decrements at each declaration node

# Algorithm Example

```
int test (int x, int y) {  
    int a, b;  
    if (x) {  
        int s;  
    } else {  
        int t, u, v;  
        u = b + y;  
    }  
}
```



# Handling Global Variables

In a sense, globals easier to handle than locals

- Space allocated directly at compile time instead of indirectly via \$fp and \$sp registers
- Never needs to be deallocated

Place in static data area

- In MIPS, handling with a special storage directive
- Variables referred to by name, not by address

# Memory Region Example

```
.data  
_x: .word 10  
_y: .byte 1  
_z: .asciiz "I am a string"  
.text  
lw $t0, _x    #Load from x into $t0  
sw $t0, _x    #Store from $t0 into x
```

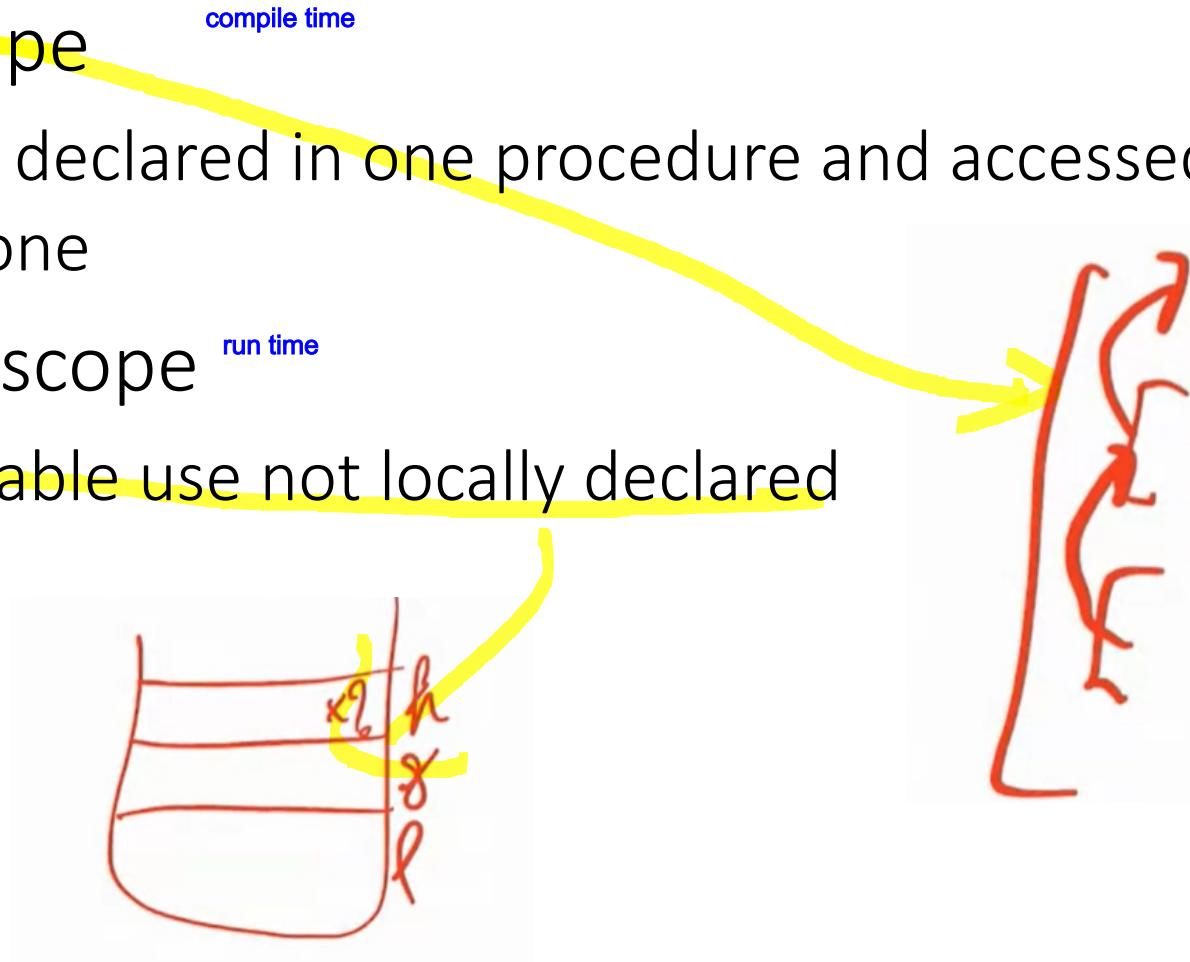
# Accessing non-local variables

## Static scope

- Variable declared in one procedure and accessed in a nested one

## Dynamic scope

- Any variable use not locally declared



# Static non-local scope example

Each function has it's own AR

- Inner function accesses the outer AR

```
function main () {  
    int a = 0;  
    function subprog () {  
        a = a + 1;  
    }  
}
```

e.g., lambda, anonymous function in Java

this is called closure, and the example here is impure, since it changes variable a, but a doesn't belong to its scope

at compile time, a in func subprog() refers to a in main()

# Static non-local scope memory access

```
void procA() { // level 1
    int x, y;
    void procB() { // level 2
        void procC() { //level 3
            int z;
            void procD() {
                int x;
                x = z + y;
                procB();
            }
            x = 4;
            z = 2;
            procB();
            procD();
        }
        x = 3;
        y = 5;
    }
}
```

notice that x in procD() doesn't share the same scope with x=4 here

# Access Links

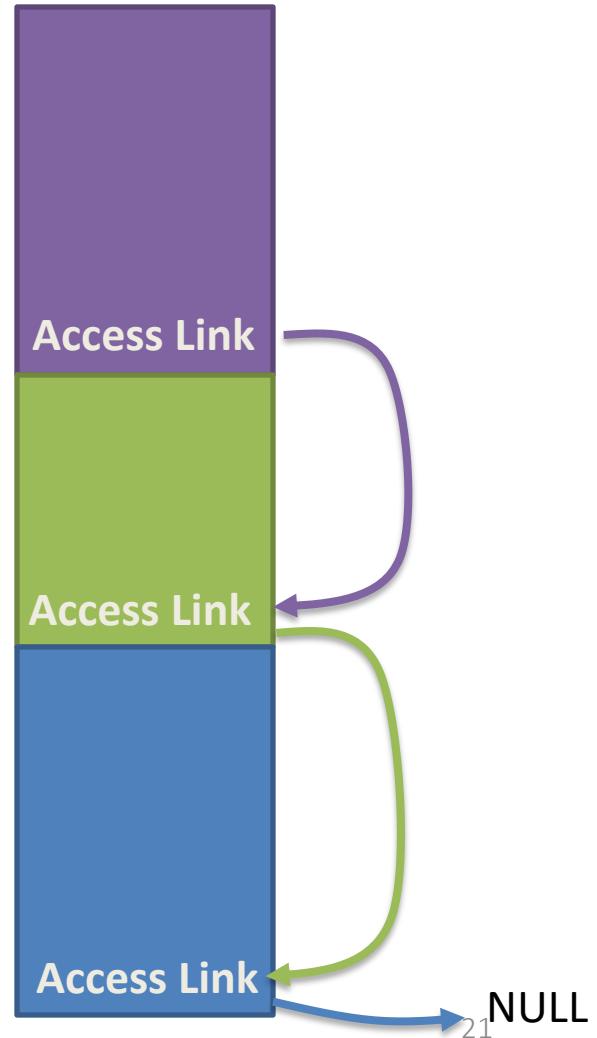
Add an additional field to the AR

- Points to the locals area of the outer function
- Sometimes called the static link (since it refers to the static nesting)

Level 3  
AR

Level 2  
AR

Level 1  
AR

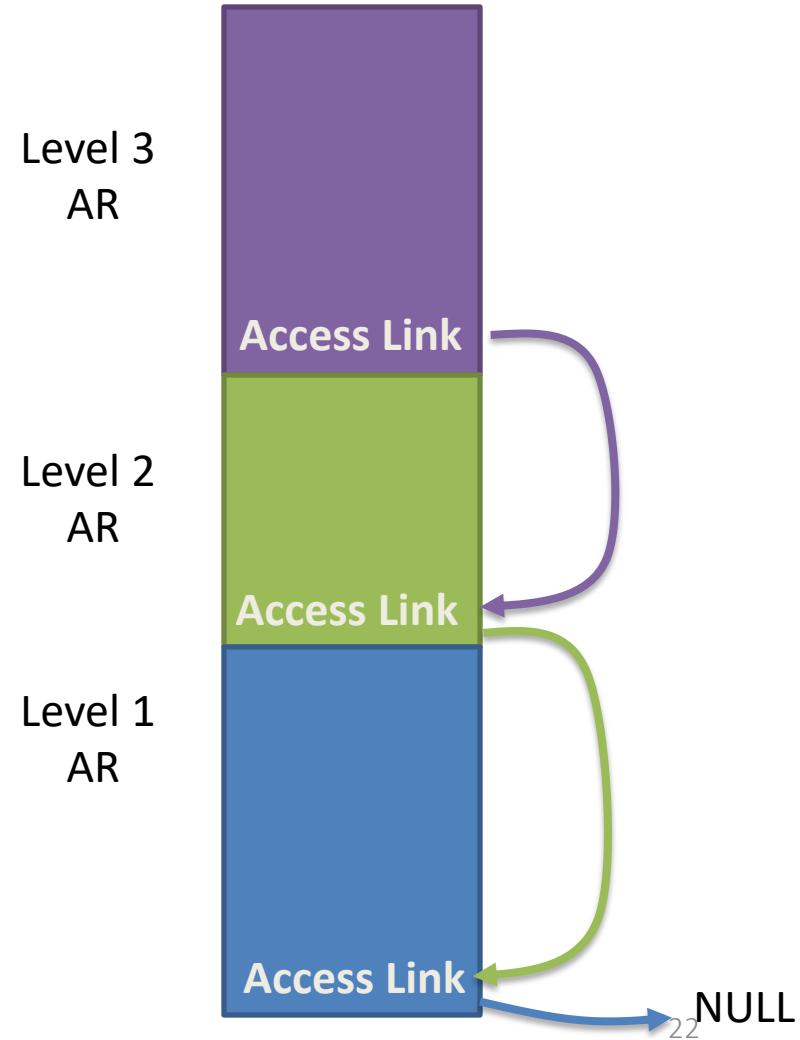
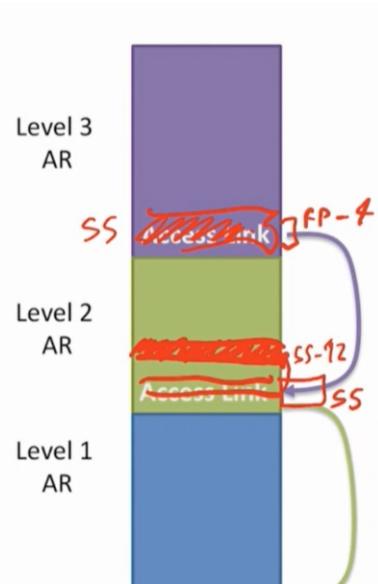


# How Access Links Work

We know how many *levels* to traverse statically

- Example: In nesting level 3 and the variable is in nesting level 1: go back access links

$(3 - 1)$  2 levels



# Setting up access links

Using 1 access link

```
lw $t0, -4 ($fp)
```

```
lw $t0, -12 ($t0)
```

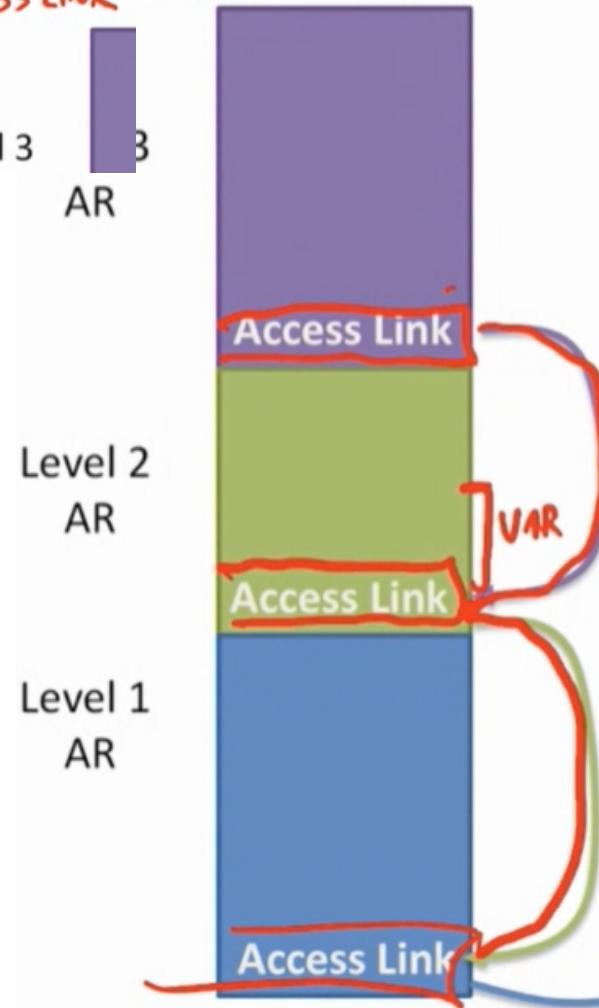
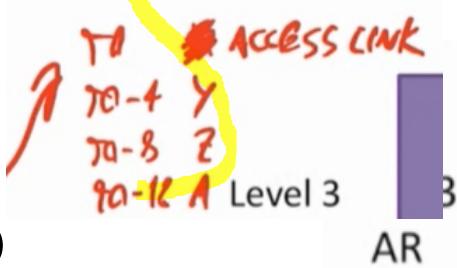
Where \$fp -4 is the location of the access link and the variable in the outer scope is at offset 12 from outer AR

Using 2 access links

```
lw $t0, -4 ($fp)
```

```
lw $t0, ($t0)
```

```
lw $t0, -12 ($t0)
```



# Thinking about access links

We know the variable we want to access statically

Why don't we just index into the parent's AR using a large positive offset from \$fp?

```
lw    $t0  38($fp)
```

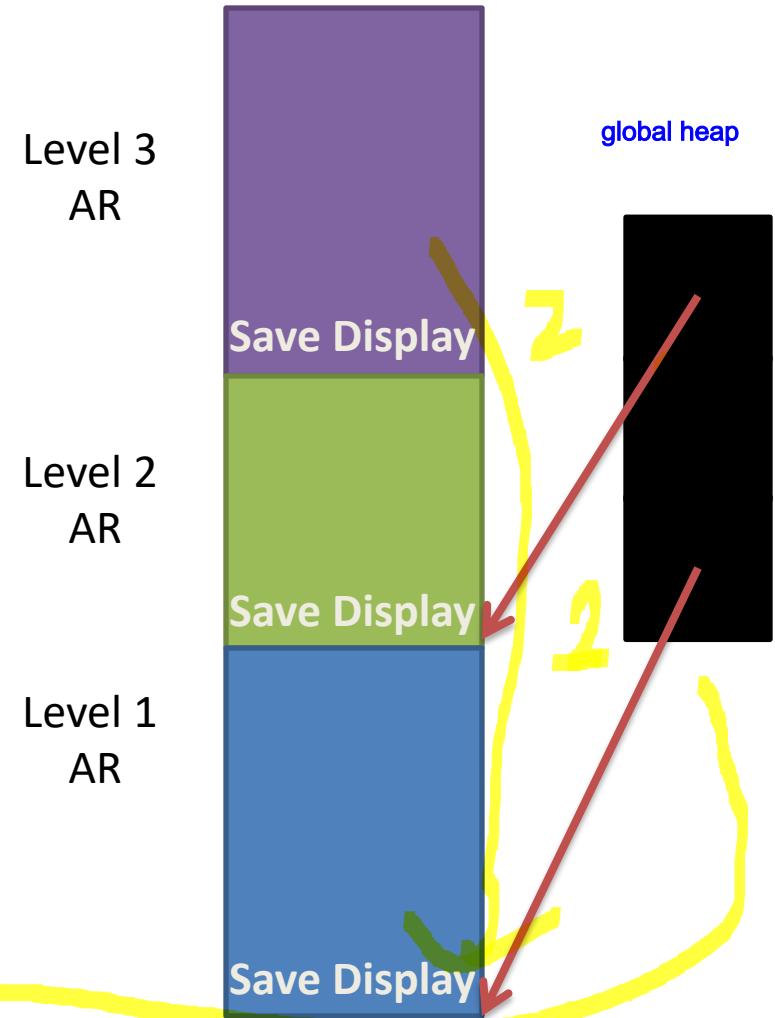
# Displays

High-level idea:

- Keep the transitive effects of multiple access link traversals
- Uses a side-table of this info

Tradeoffs vs Access Links?

- Faster to call far up the hierarchy
- Takes extra space



# Questions about Static Scope?

# Dynamic non-local scope example

```
function main() {  
    a = 0;  
    fun();  
}  
e.g., js julia
```

```
function fun() {  
    a = a + 1;  
}
```

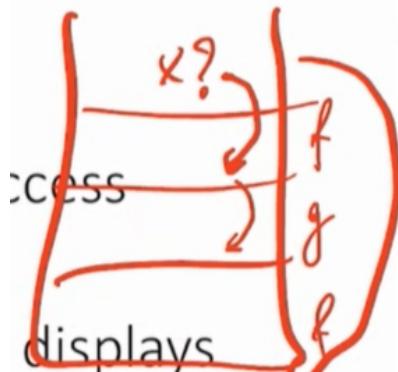
# Dynamic Scope Storage

## Key point

- We don't know *which* non-local variable we are referring to

Two ways to set up dynamic access

1. Deep Access – somewhat similar to Access links
2. Shallow Access – somewhat similar to displays



# Deep Access

If the variable isn't local

- Follow the control link to the caller's AR
- Check to see if it defines the variable
- If not, follow the next control link down the stack

Note that we somehow need to know if a variable is defined *by name* in an AR

- Usually means we'll have to associate a name with a stack slot

# Shallow Access

Keep a table with an entry for each variable declaration

- Compile a direct reference to that entry
- At a function call
  - Save all locals in the *caller*'s AR
  - Restore locals when the *callee* is finished

# Roadmap

We learned about variable access

- Local vs global variables
- Static vs dynamic scopes

Next time

- We'll start getting into the details of MIPS
- Code generation

# Code Generation

# Roadmap

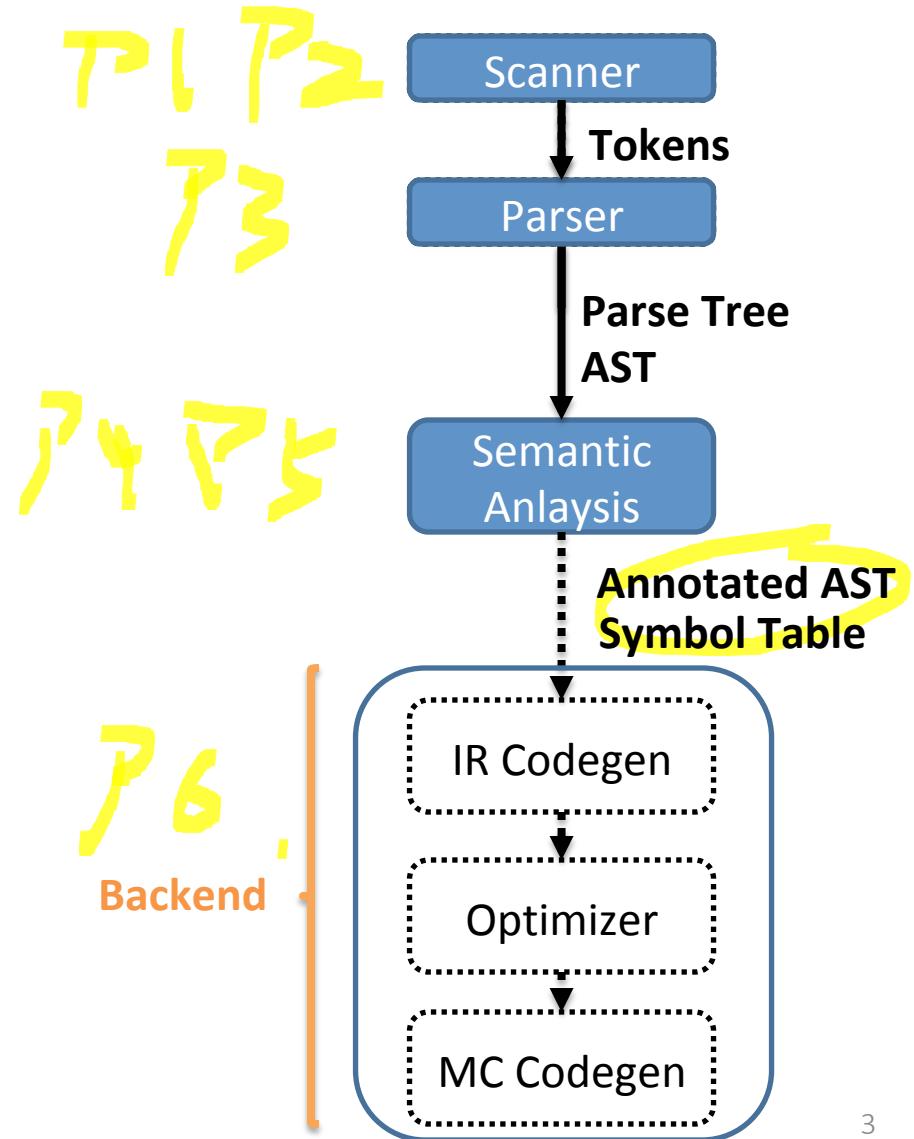
Last time, we learned about variable access

- Local vs global variables
- Static vs dynamic scopes

Today

- We'll start getting into the details of MIPS
- Code generation

# Roadmap



# The Compiler Back-end

Unlike front-end, we can skip phases without sacrificing correctness

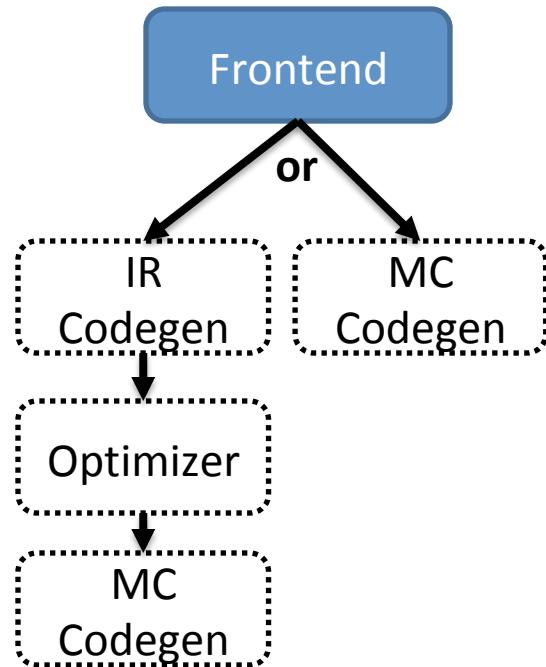
Actually have a couple of options

- What phases do we do
- How do we order our phases

# Outline

## Possible compiler designs

- Generate IR code or MC code directly?
- Generate during SDT or as another phase?



# How many passes do we want?

Fewer passes

- Faster compiling
- Less storage requirements
- May increase burden on programmer

More passes

- Heavyweight
- Can lead to better modularity
- We'll go with this approach for our language

# To Generate IR Code or Not?

Generate Intermediate Representation:

- More amenable to optimization
- More flexible output options
- Can reduce the complexity of code generation

Go straight to machine code:

- Much faster to generate code (skip 1 pass, at least)
- Less engineering in the compiler

# What might the IR Do?

Provide illusion of infinitely many registers

“Flatten out” expressions

- Does not allow build-up of complex expressions

3AC (Three-Address Code)

- Pseudocode-machine style instruction set
- Every operator has at most 3 operands

$$\begin{array}{ll} X = Y + Z + A & P = Y + Z \\ & X = P + A \end{array}$$

↳ expressions

ADD Y Z P

# 3AC Example

```
if      (x + y * z > x * y + z)
      a = 0;
b = 2;
```

```
tmp1 = y * z
tmp2 = x+tmp1
tmp3 = x*y
tmp4 = tmp3+z
```

if (tmp2 <= tmp4) goto L

```
a = 0
```

L: b = 2

here if condition is false, we jump to b=2;

if condition is true, we just move to a=0;

# 3AC Instruction Set

## Assignment

- $x = y \text{ op } z$
- $x = \text{op } y$
- $x = y$

## Jumps

- if ( $x \text{ op } y$ ) goto  $L$

## Indirection

- $x = y[z]$
- $y[z] = x$
- $x = \&y$
- $x = *y$
- $*y = x$

## Call/Return

- param  $x,k$
- retval  $x$
- call  $p$
- enter  $p$
- leave  $p$
- return
- retrieve  $x$

## Type Conversion

- $x = \text{AtoB } y$

## Labeling

- label  $L$

## Basic Math

- times, plus, etc.

# 3AC Representation

Each instruction represented using a structure called a “quad”

- Space for the operator
- Space for each operand
- Pointer to auxilary info
  - Label, successor quad, etc.

Chain of quads sent to an architecture specific machine code generation phase

# 3AC LLVM Example

Demo

# Direct machine code generation

## Option 1

- Have a chain of quad-like structures where each element is a machine-code instruction
- Pass the chain to a phase that writes to file

## Option 2

- Write code directly to the file
- Greatly aided by assembly conventions here
- Assembler allows us to use function names, labels in output

# Our language: skip the IR

## Traverse AST

- Add codeGen methods to the AST nodes
- Directly write corresponding code into file

# Correctness/Efficiency Tradeoffs

Two high-level goals

1. Generate correct code
2. Generate *efficient* code

It can be difficult to achieve both of these at the same time

- Why?

# Simplifying assumptions

Make sure we don't have to worry about running out of registers

we assume infinite number of reg.

- We'll put all function arguments on the stack
- We'll make liberal use of the stack for computation
  - Only use \$t1 and \$t0 for computation

# The CodeGen Pass

We'll now go through a high-level idea of how the topmost nodes in the program are generated

# The Effect of Different Nodes

Many nodes simply structure their results

- ProgramNode.codeGen
  - call codeGen on the child
- List node types (e.g., StmtList)
  - call codeGen on each element in turn
- DeclNode
  - StructDeclNode – no code to generate!
  - FnDeclNode – generate function body
  - VarDeclNode – varies on context! Globals v locals



# Global Variable Declarations

Source code:

```
int name;
```

```
struct MyStruct instance;
```

In varDeclNode

Generate:

```
.data  
.align 4 #Align on word boundaries  
_name: .space N # (N is the size of variable)
```



width=32 bits=4 bytes

LLVM

# Generating Global Variable Declaration

```
.data  
    .align 4    #Align on word boundaries  
_name: .space N    #(N is the size of variable)
```

How do we know the size?

- For scalars, well defined: int, bool (4 bytes)
- structs, 4 \* size of the struct

We can calculate this during name analysis

# Generating Function Definitions

Need to generate

- Preamble
  - Sort of like the function signature
- Prologue
  - Set up the function
- Body
  - Perform the computation
- Epilogue
  - Tear down the function

→ build frame  
→ destroy frame

# MIPS crash course

## Registers

Register	Purpose
\$sp	stack pointer
\$fp	frame pointer
\$ra	return address
\$v0	used for system calls and to return int values from function calls, including the syscall that reads an int
\$f0	used to return double values from function calls, including the syscall that reads a double
\$a0	used for output of int and string values
\$f12	used for output of double values
\$t0 - \$t7	temporaries for ints
\$f0 - \$f30	registers for doubles (used in pairs; i.e., use \$f0 for the pair \$f0, \$f1)

# Program structure

## Data

- Label: .data
- Variable names & size; heap storage

## Code

- Label: .text
- Program instructions
- Starting location: **main**
- Ending location

# Data

name:      type    value(s)

– E.g.

- v1:      .word      10
- a1:      .byte      ‘a’ , ’b’
- a2:      .space      40

– 40 here is allocated space – no value is initialized

# Mem Instructions

*word*

**lw** register\_destination, RAM\_source

– copy word (4 bytes) at source RAM location to destination register.

*byte*

**lb** register\_destination, RAM\_source

– copy byte at source RAM location to low-order byte of destination register

**li** register\_destination, value

– load immediate value into destination register

# Mem instructions

**sw register\_source, RAM\_dest**

– store word in source register into RAM destination

**sb register\_source, RAM\_dest**

– store byte in source register into RAM destination

# Arithmetic instructions

unsigned

	add	\$t0,\$t1,\$t2	$t_0 = t_1 + t_2$
	sub	\$t2,\$t3,\$t4	
	addi	\$t2,\$t3, 5	
	addu	\$t1,\$t6,\$t7	
	subu	\$t1,\$t6,\$t7	
	mult	\$t3,\$t4	Stores result in \$lo $t_0 = t_3 \times t_4$
	div	\$t5,\$t6	Stores result in \$lo and Remainder in \$hi
	mfhi	\$t0	
	mflo	\$t1	

# Control instructions

	b	target
less than	beq	\$t0,\$t1,target
	blt	\$t0,\$t1,target
less or equal	ble	\$t0,\$t1,target
	bgt	\$t0,\$t1,target
	bge	\$t0,\$t1,target
not equal	bne	\$t0,\$t1,target
jump return	j	target
	jr	\$t3
jal	sub_label	# "jump and link"
		Jump and store return address in \$31



# TODO

Watch ALL MIPS and SPIM tutorials online

– [pages.cs.wisc.edu/~loris/cs536s18/resources.html](https://pages.cs.wisc.edu/~loris/cs536s18/resources.html)

MIPS tutorial

[https://minnie.tuhs.org/CompArch/Resources/  
mips\\_quick\\_tutorial.html](https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html)

# Roadmap

## Today

- Talked about compiler backend design points
- Decided to go with direct to machine code design for our language

## Next time:

- Run through what actual codegen pass will look like

# Code Generation, Continued

# How to be a MIPS Master

It's really easy to get confused with assembly

- Try writing a program by hand before having the compiler generate it
- Draw lots of pictures of program flow
- Have your compiler output detailed comments

Get help

- Post on piazza

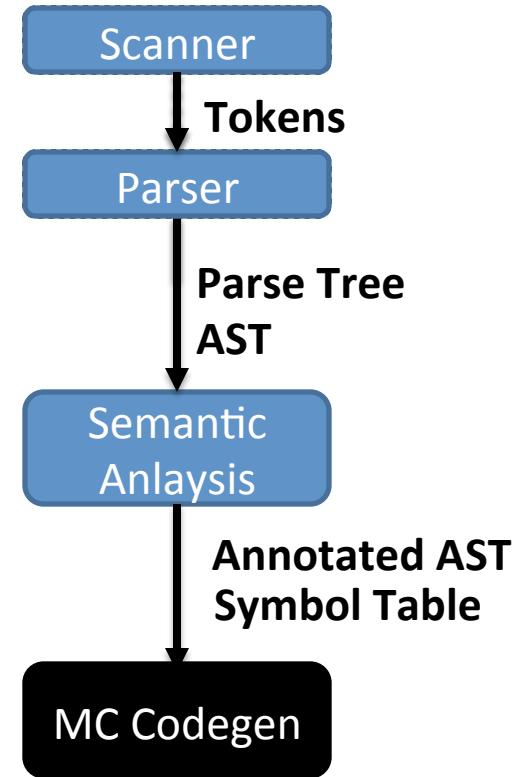
# Roadmap

Last time:

- Talked about compiler backend design points
- Decided to go with direct to machine code design for our language

This time:

- Run through what the actual codegen pass will look like



# Review: Global Variables

Showed you one way to do declaration last time:

```
.data  
.align 2  
_name: .space 4
```

Simpler form for primitives:

```
.data  
_name: .word <value>
```

# Review: Functions

Preamble

- Sort of like the function signature

Prologue

- Set up the function

Body

- Do the thing

Epilogue

- Tear down the function

# Function Preambles

```
int f(int a, int b) {           .text
    int c = a + b;             f:
    int d = c - 7;            #... Function body ...
    return c;
}
```

**This label gives us something to jump to**

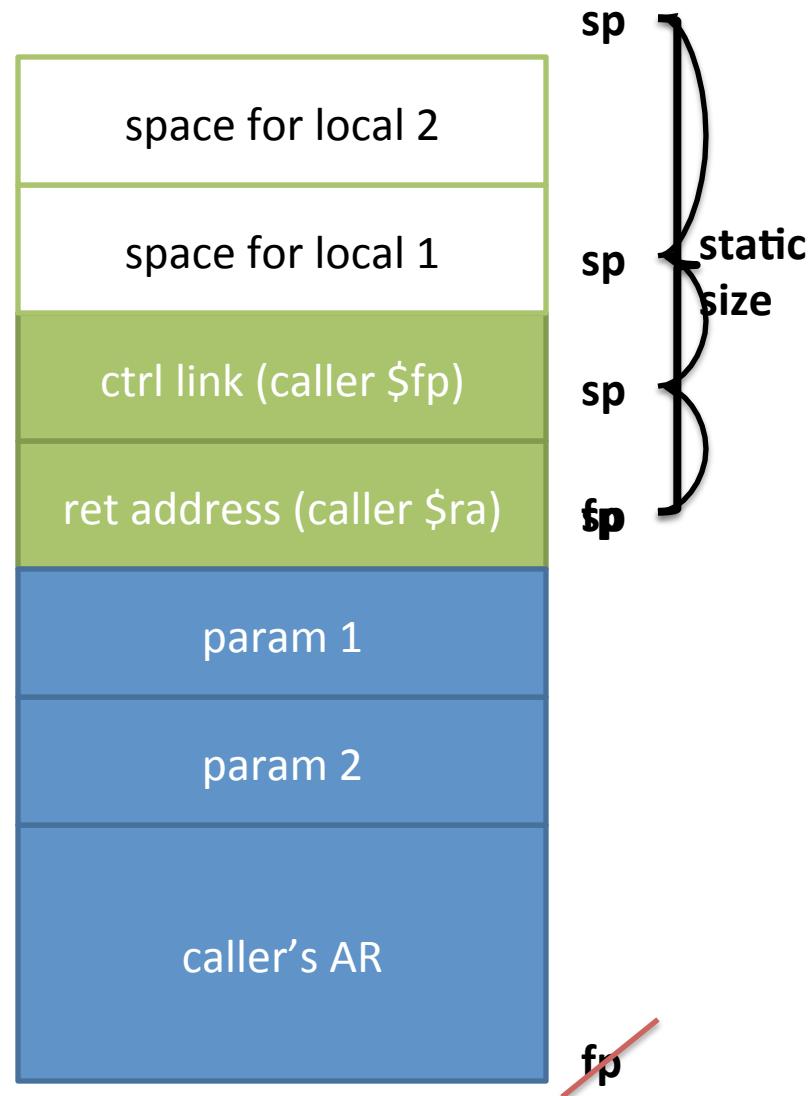
```
jal f
```

# Function Prologue

Recall our view of the Activation Record

1. save the return address
2. save the frame pointer
3. make space for locals
4. update the frame ptr

*low mem*  
↑  
*high mem*



# Function Prologue: MIPS

Recall our view of the Activation Record

1. save the return address
2. save the frame pointer
3. make space for locals
4. update the frame ptr

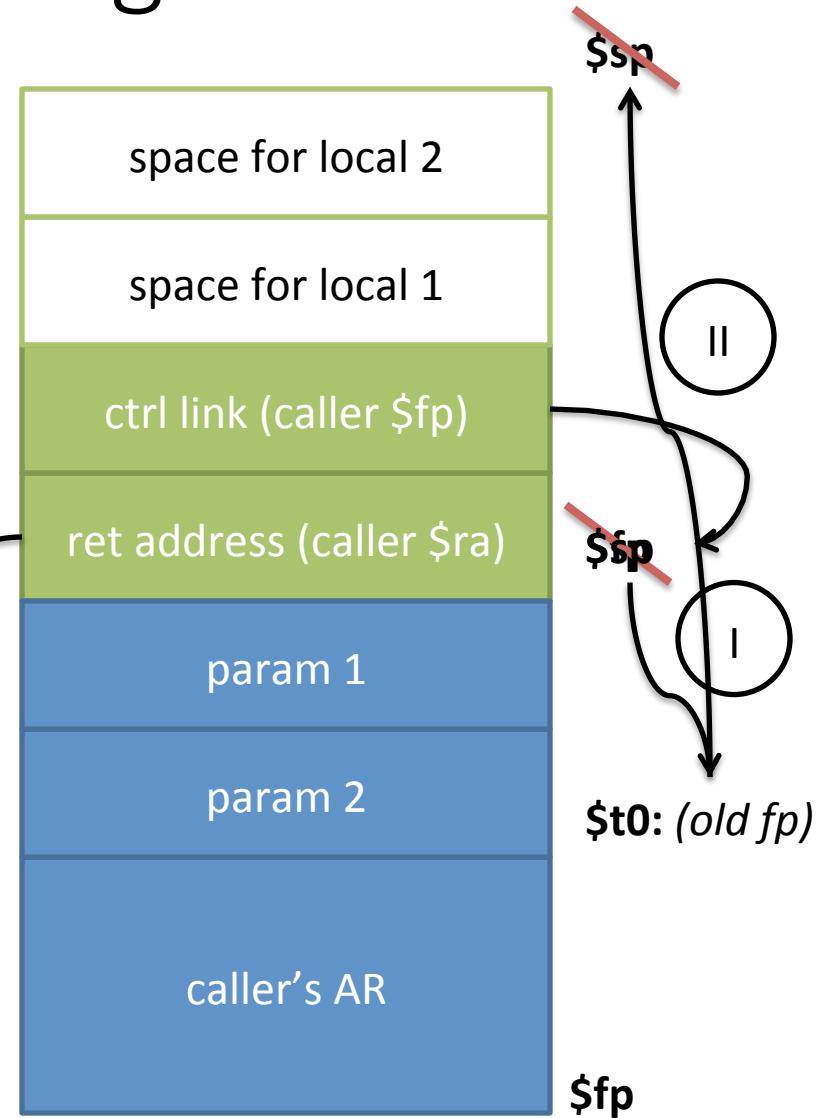
```
.text
f:
    sw $ra 0($sp)      #call  lnk
    subu $sp $sp 4     # (push)
    sw $fp 0($sp)      #ctrl  lnk
    subu $sp $sp 4     # (push)
    subu $sp $sp 8     #locals
    addu $fp $sp 16    #update fp
```

# Function Epilogue

## Restore Caller AR

1. restore return address
2. restore frame pointer
3. restore stack pointer
4. return control

$\$ra: (old \$ra)$



# Function Epilogue: MIPS

## Restore Caller AR

1. restore return address
2. restore frame pointer
3. restore stack pointer
4. return control

load-> content of location=mem of loc 100 contains content 5

LW REGISTAR MEM ADDRESS  
Move Register Register  
LW t<sub>0</sub> 0(R)  $\Rightarrow$  t<sub>0</sub>=5  
Move t<sub>0</sub> R  $\Rightarrow$  t<sub>0</sub>=100



move -> content of R=addr 100

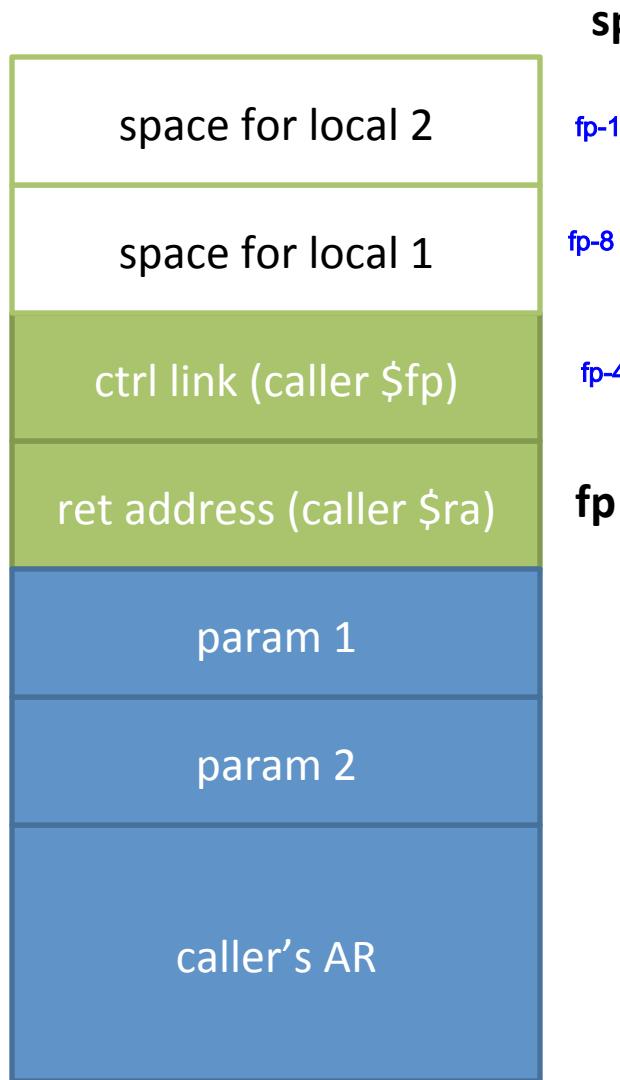
```
.text
f:
    sw $ra 0($sp)
    subu $sp $sp 4
    sw $fp 0($sp)
    subu $sp $sp 4
    subu $sp $sp 8
    addu $fp $sp 16
    #... Function body ...
    lw $ra, 0($fp)
    move $t0, $fp
    lw $fp, -4($fp)
    move $sp, $t0
    jr $ra
```

# Function Body

Obviously, quite different based on content

- Higher-level data constructs
  - Loading parameters, setting return
  - Evaluating expressions
- Higher-level control constructs
  - Performing a call
  - Loops
  - Ifs

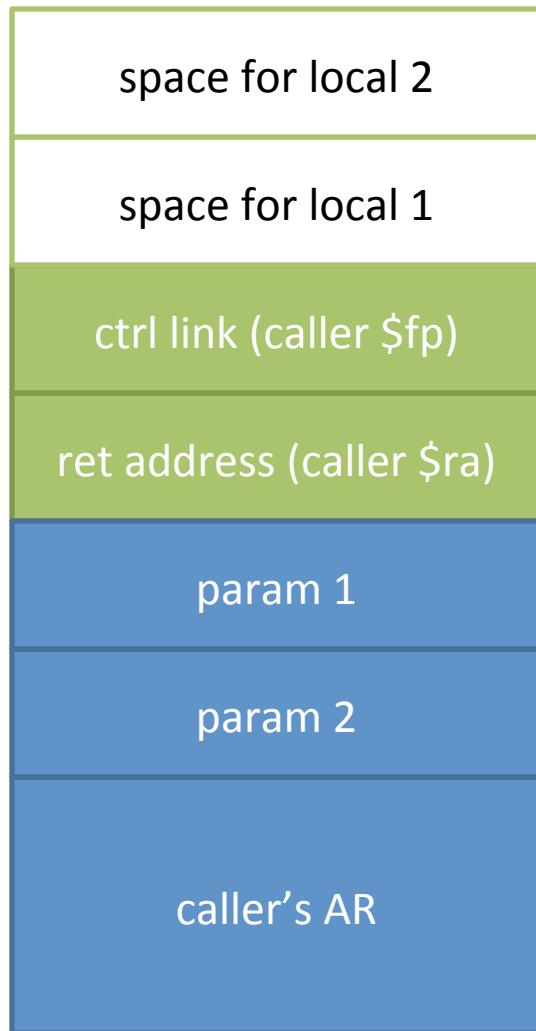
# Function Locals



```
.text
f:
# ... prologue ...
lw $t0, -8($fp)    first local
lw $t1, -12($fp)   second local
```

# ... epilogue ... #

# Function Returns



sp

fp

.text

f:

# ... prologue ... #

lw \$t0, -8(\$fp)

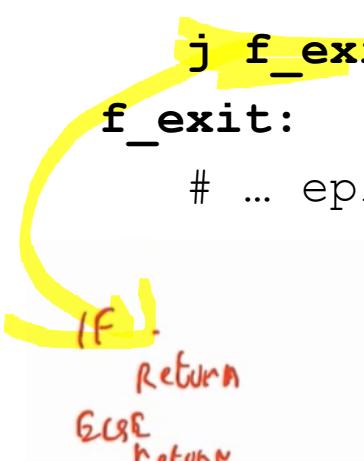
lw \$t1, -12(\$fp)

lw \$v0, -8(\$fp) return local 1;  
v0 is used as return reg

j f\_exit

f\_exit:

# ... epilogue ... #



14

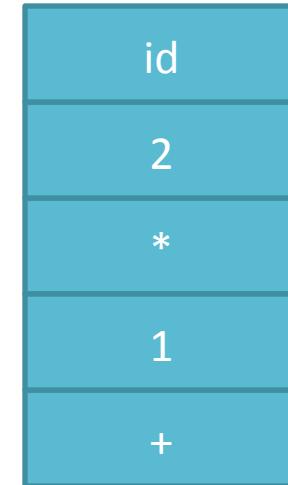
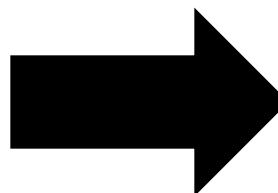
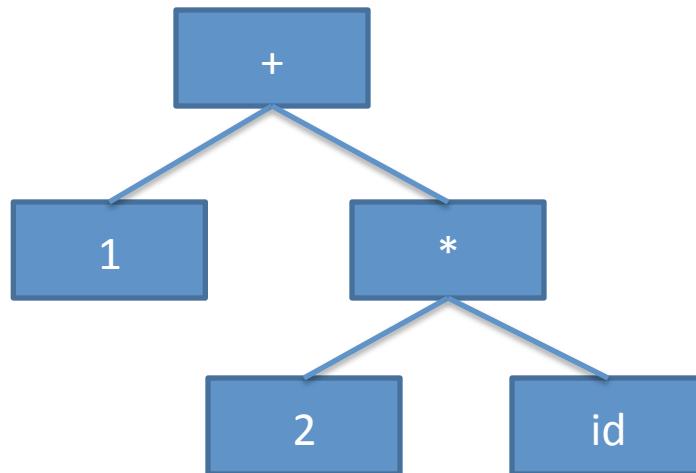
# Function Body: Expressions

Goal

- Serialize (“flatten”) an expression tree

Use the same insight as the parser

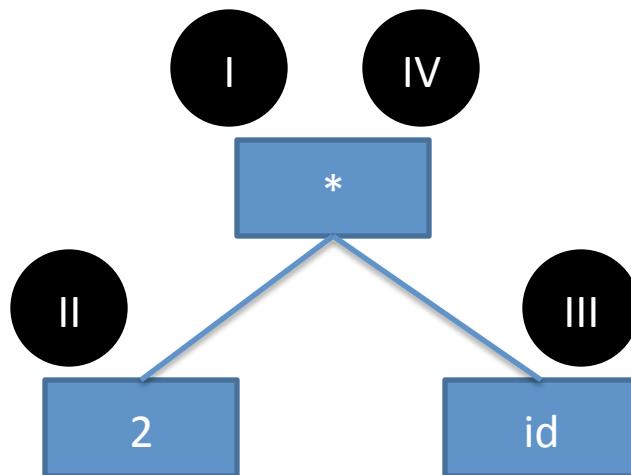
- Use a work stack and a post-order traversal



# Serialized Psuedocode

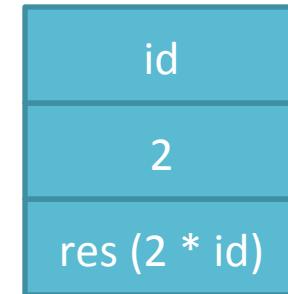
## Key insight

- Use the stack pointer location as “scratch space”
- At operands: push value onto the stack
- At operators: pop source values from stack, push result



push 2  
push id  
pop id into t1  
pop 2 into t0  
mult t0 \* t1 into t0  
push t0

$\$t1 = id$   
 ~~$\$t0 = 2$~~   $2 * id$



# Serialized MIPS

L1: push 2  
L2: push id  
L3: pop id into t1  
L4: pop 2 into t0  
L5: mult t0 \* t1 into t0  
L6: push t0

lw loads word

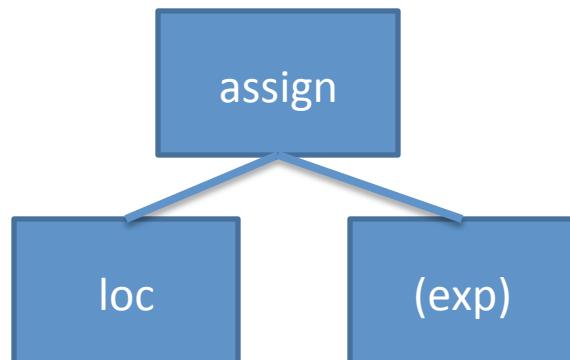
li loads integer  
store it on top of stack  
sub stack ptr by 4; 增高  
stack size by 4 bytes

```
L1: li $t0 2
    sw $t0 0($sp)
    subu $sp $sp 4
L2: lw $t0 id
    sw $t0 0($sp)
    subu $sp $sp 4
L3: lw $t1 4($sp)
    addu $sp $sp 4
L4: lw $t0 4($sp)
    addu $sp $sp 4
L5: mult $t0 $t0 $t1
L6: sw $t0 0($sp)
    subu $sp $sp 4
```

# Stmts

By the end of the expression, our stack isn't exactly as we left it

- Contains the result of the expression
- This is by design



- 1) Compute RHS expr on stack
- 2) Compute LHS *location* on stack
- 3) Pop LHS into \$t1
- 4) Pop RHS into \$t0
- 5) Store value \$t0 at address \$t1

# Simple Assign, You Try

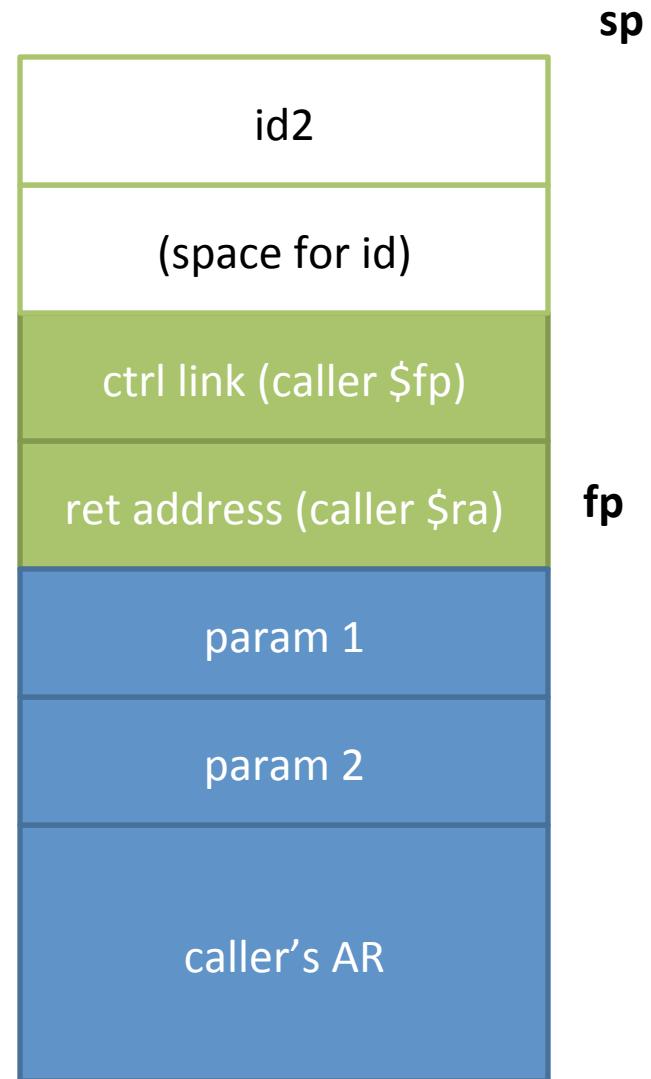
Generate stack-machine style MIPS code for

$\text{id} = 1 + 2;$

## Algorithm

- 1) Compute RHS expr on stack
- 2) Compute LHS *location* on stack
- 3) Pop LHS into \$t1
- 4) Pop RHS into \$t0
- 5) Store value \$t0 at address \$t1

$1+2$   
l<sub>k</sub>  $t_1 \rightarrow 10$   
 $t_0 = 3$   
l  $3 \rightarrow t_1$



# Dot Access

Fortunately, we know the offset from the base of a struct to a certain field statically

- The compiler can do the math for the slot address
- This isn't true for languages with pointers!

```
struct Demo inst;  
struct Demo inst2;  
inst.b.c = inst2.b.c + 1;
```

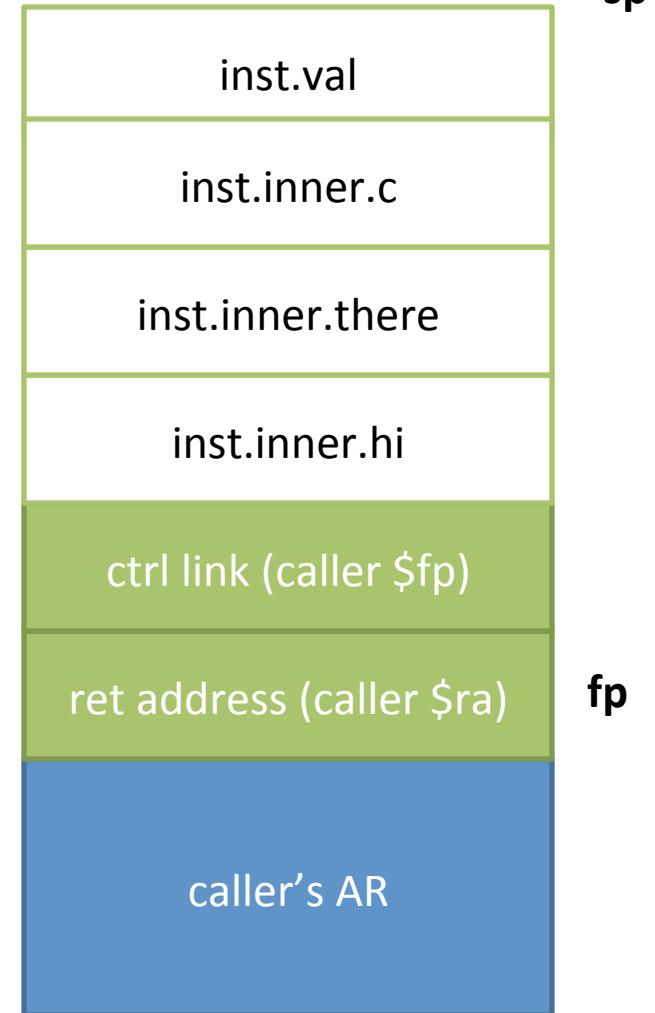
load this address

load this value

# Dot Access Example

```
void v() {
    struct Inner{
        bool hi;
        int there;
        int c;
    };
    struct Demo{
        struct Inner b;
        int val;
    };
    struct Demo inst;
    inst.b.c = inst.b.c;
}
```

**inst is based at -8(\$fp )**  
**field b.c is -8 off the base**



## LHS

```
subu $t0 $fp 16  
sw $t0 0($sp)
```

## RHS

```
lw $t0 -16($fp)  
sw $t0 0($sp)  
subu $sp $sp 4
```

# Control Flow Constructs

Function Calls

Loops

Ifs

We do these next time

# Function Call Example

```
int f(int arg1, int arg2){  
    return 2;  
}
```

```
int main() {  
    int a;  
    a = f(a, 4);  
}
```

```
li $t0 4          # push arg 2  
sw $t0 0($sp)    #  
subu $sp $sp 4   #  
lw $t0 -8($fp)   # push arg 1  
sw $t0 0($sp)    #  
subu $sp $sp 4   #  
jal f            # goto f  
addu $sp $sp 8   # tear down params  
sw $v0 -8($fp)   # retrieve result
```

Return to  
original state

# Summary

Today:

- Got the basics of MIPS
- CodeGen for *some* AST node types

Next time:

- Do the rest of the AST nodes
- Introduce control flow graphs

# Function Call

Two tasks:

- Put argument *values* on the stack (pass-by-value semantics)
- Jump to the callee preamble label
- Bonus 3<sup>rd</sup> task: save *live* registers
  - (We don't have any in a stack machine)
- Semi-bonus 4<sup>th</sup> task: retrieve result value

# Code Generation for Control Flow Constructs

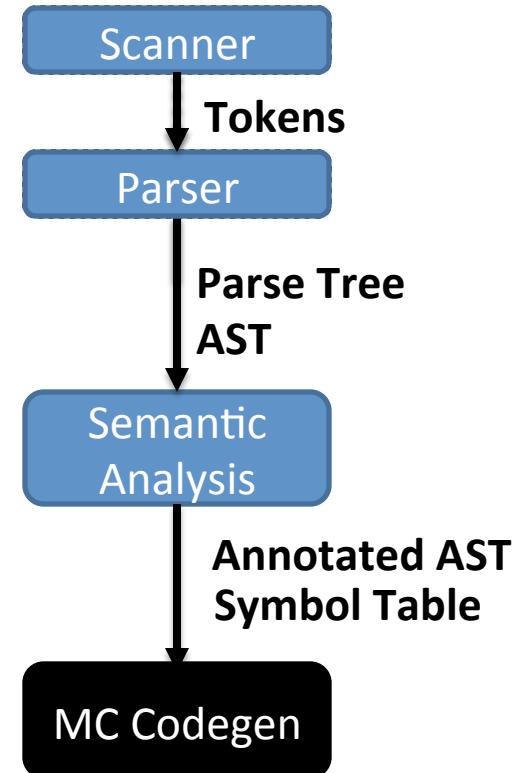
# Roadmap

Last time:

- Got the basics of MIPS
- CodeGen for *some* AST node types

This time:

- Do the rest of the AST nodes
- Introduce control flow graphs

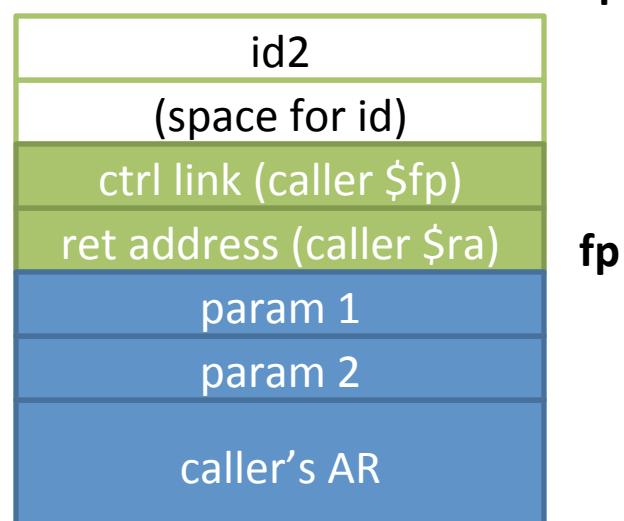


# A Quick Warm-Up: MIPS for $\text{id} = 1 + 2;$

	li	\$t0	1
push 1	sw	\$t0	0 (\$sp)
	subu	\$sp	\$sp 4
push 2	li	\$t0	2
	sw	\$t0	0 (\$sp)
	subu	\$sp	\$sp 4
pop opR	lw	\$t1	4 (\$sp)
	addu	\$sp	\$sp 4
pop opL	lw	\$t0	4 (\$sp)
	addu	\$sp	\$sp 4
Do 1+2	add	\$t0	\$t0 \$t1
push RHS	sw	\$t0	0 (\$sp)
	subu	\$sp	\$sp 4
push LHS	la	\$t0	-8 (\$fp)
	sw	\$t0	0 (\$sp)
	subu	\$sp	\$sp 4
pop LHS	lw	\$t1	4 (\$sp)
	addu	\$sp	\$sp 4
pop RHS	lw	\$t0	4 (\$sp)
	addu	\$sp	\$sp 4
Do assign	sw	\$t0	0 (\$t1)

## General-Purpose Algorithm

- 1) Compute RHS expr on stack
- 2) Compute LHS *location* on stack
- 3) Pop LHS into \$t1
- 4) Pop RHS into \$t0
- 5) Store value \$t0 at address \$t1

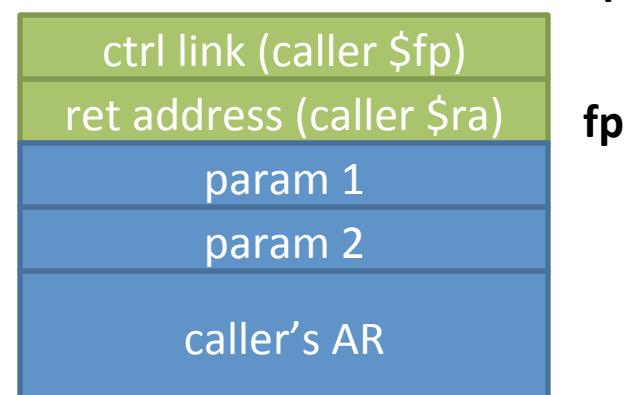


# Same Example if id was Global

	li	\$t0	1
push 1	sw	\$t0	0 (\$sp)
	subu	\$sp	\$sp 4
	li	\$t0	2
push 2	sw	\$t0	0 (\$sp)
	subu	\$sp	\$sp 4
	lw	\$t1	4 (\$sp)
pop opR	addu	\$sp	\$sp 4
	lw	\$t0	4 (\$sp)
pop opL	addu	\$sp	\$sp 4
Do 1+2	add	\$t0	\$t0 \$t1
push RHS	sw	\$t0	0 (\$sp)
	subu	\$sp	\$sp 4
push LHS	la	\$t0	-8 (\$ip) <span style="color:red">id</span>
	sw	\$t0	0 (\$sp)
	subu	\$sp	\$sp 4
pop LHS	lw	\$t1	4 (\$sp)
	addu	\$sp	\$sp 4
pop RHS	lw	\$t0	4 (\$sp)
	addu	\$sp	\$sp 4
Do assign	sw	\$t0	0 (\$t1)

## General-Purpose Algorithm

- 1) Compute RHS expr on stack
- 2) Compute LHS *location* on stack
- 3) Pop LHS into \$t1
- 4) Pop RHS into \$t0
- 5) Store value \$t0 at address \$t1



# Do We Need LHS computation ?

This is a bit much when the LHS is a variable

- We end up doing a single load to find the address, then a store, then a load
- We know a lot of the computation at compile time

# Static v Dynamic Computation

Static

- Perform the computation at compile-time

Dynamic

- Perform the computation at runtime

As applied to memory addresses...

- Global variable location
- Local variable
- Field offset

# More Complex LHS addresses

Chain of dereferences

java: a.b.c.d

Array cell address

arr[1]

arr[c]

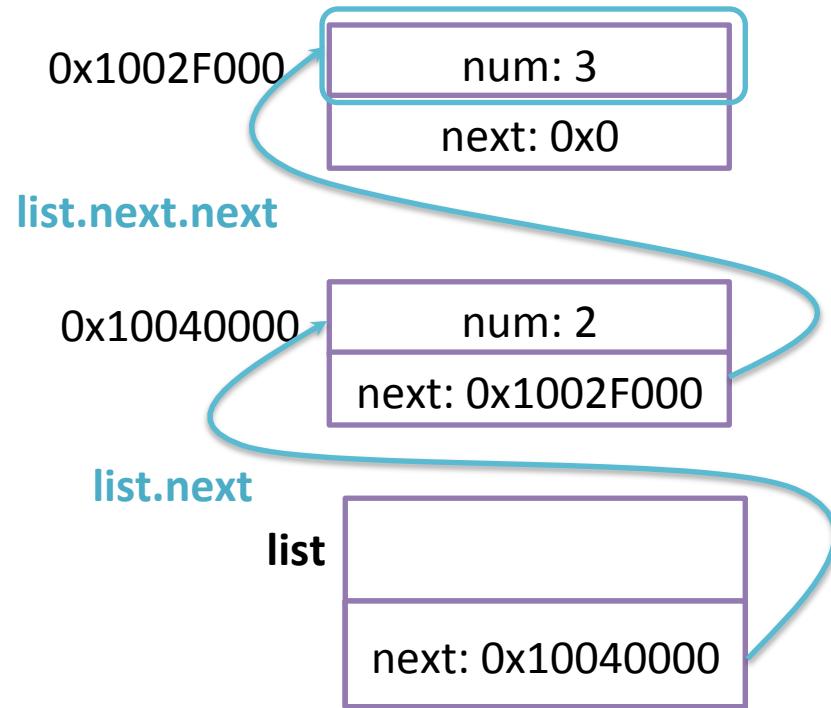
arr[1][c]

arr[c][1]

# Dereference Computation

```
struct LinkedList{  
    int num;  
    struct LinkedList& next;  
}  
  
list.next.next.num = list.next.num  
  
multi-step code to           multi-step code to  
load this address             load this value
```

- Get base addr of list
- Get offset to next field
- Load value in next field
- Get offset to next field
- Load value in next field
- Get offset to num field
- Load that address



# Control Flow Constructs

Function Calls

Loops

Ifs

# Function Call

Two tasks:

- Put argument *values* on the stack (pass-by-value semantics)
- Jump to the callee preamble label
- Bonus 3<sup>rd</sup> task: save *live* registers
  - (We don't have any in a stack machine)
- Semi-bonus 4<sup>th</sup> task: retrieve result value

# Function Call Example

```
int f(int arg1, int arg2) {  
    return 2;  
}  
  
int main() {  
    int a;  
    a = f(a, 4);  
}  
  
li $t0 4          # push arg 2  
sw $t0 0($sp)    #  
subu $sp $sp 4   #  
lw $t0 -8($fp)   # push arg 1  
sw $t0 0($sp)    #  
subu $sp $sp 4   #  
jal f            # goto f  
addu $sp $sp 8   # tear down params  
sw $v0 -8($fp)   # retrieve result
```

# We Need a New Tool

## Control Flow Graph

- Important representation for program optimization
- Helpful way to visualize source code

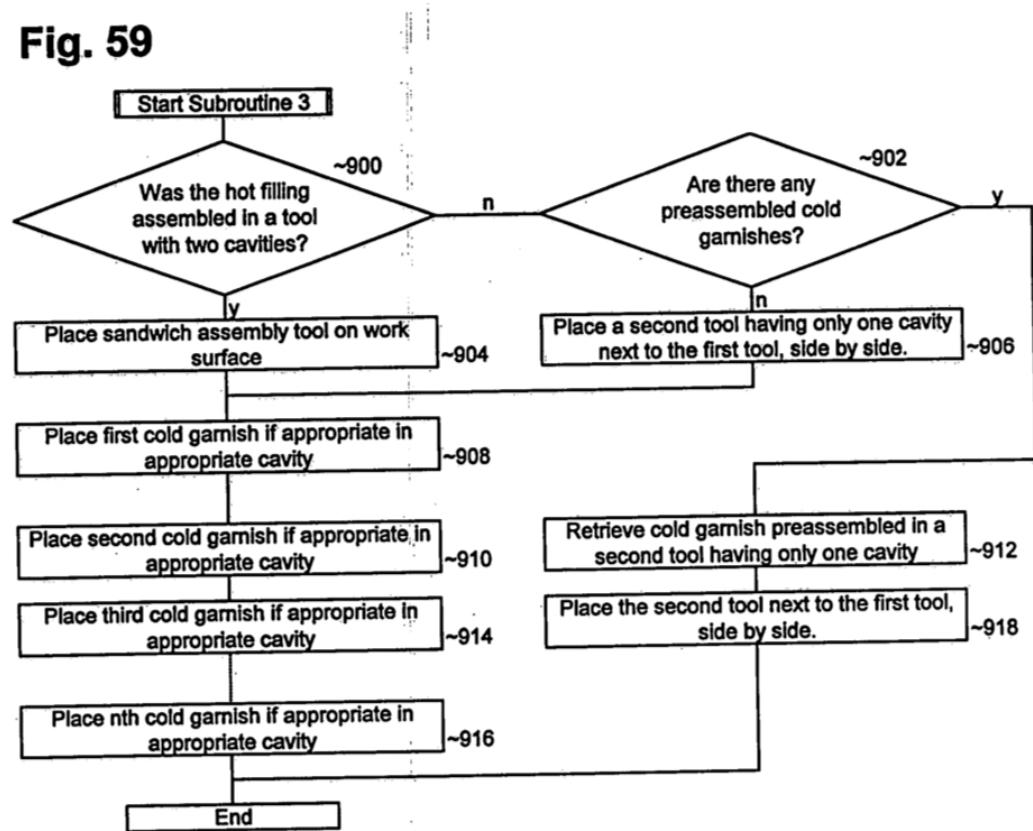


# Control Flow Graphs: the Other CFG

Think of a CFG like a flowchart

- Each block is a set of instructions
- Execute the block, decide on next block

Fig. 59

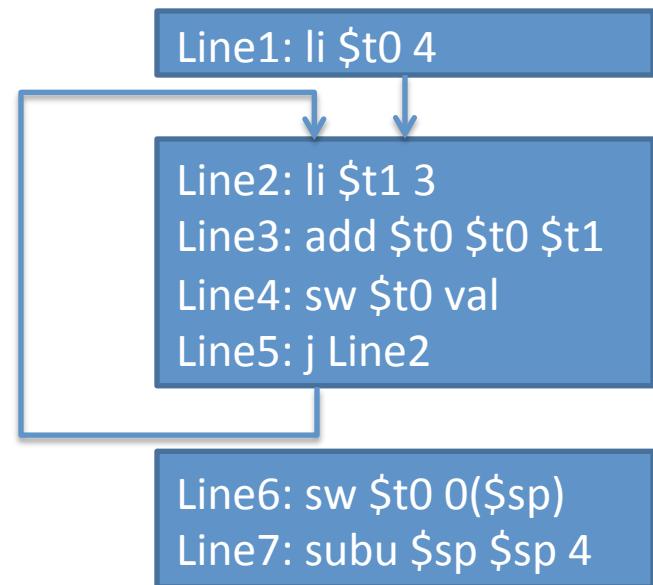


# Basic Blocks

## Nodes in the CFG

Largest run of instructions  
that will always be  
executed in sequence

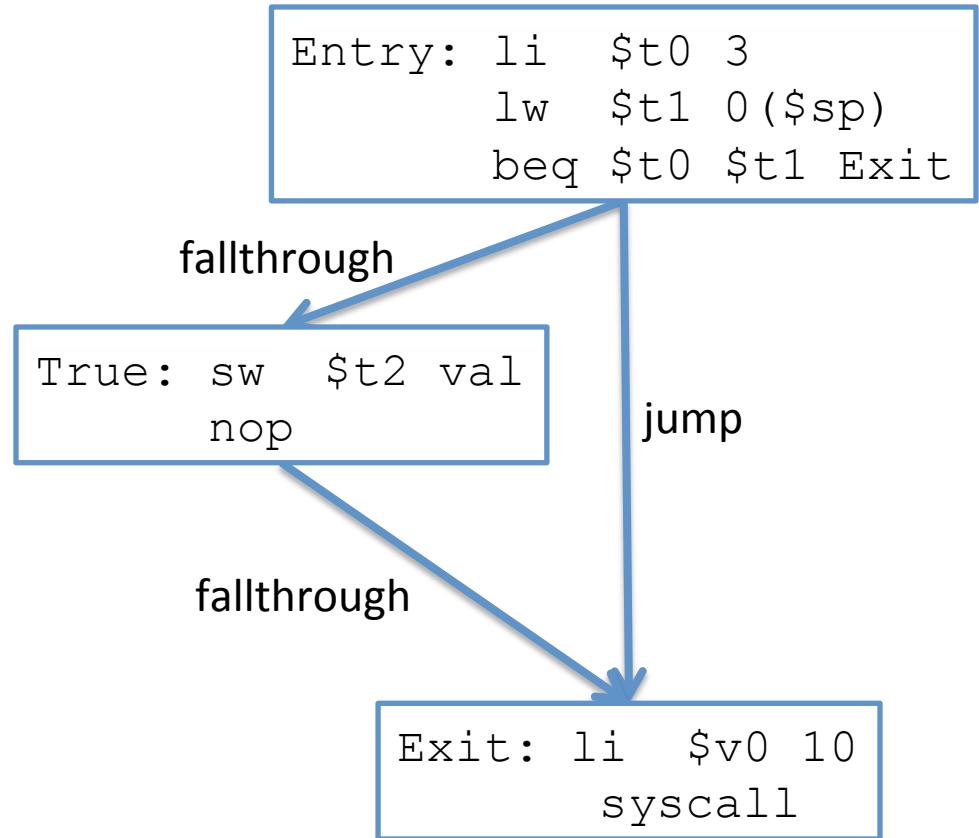
Line1: li \$t0 4  
Line2: li \$t1 3  
Line3: add \$t0 \$t0 \$t1  
Line4: sw \$t0 val  
Line5: j Line2  
Line6: sw \$t0 0(\$sp)  
Line7: subu \$sp \$sp 4



# Conditional Blocks

Branch instructions  
cause a node to have  
multiple out-edges

```
Entry: li $t0 3
       lw $t1 0($sp)
       beq $t0 $t1 Exit
True:  sw $t2 val
       nop
Exit: li $v0 10
      syscall
```



# Generating If-thenStmts

First, get label for the exit

Generate the head of the if

- Make jumps to the (not-yet placed!) exit label

Generate the true branch

- Write the body of the true node

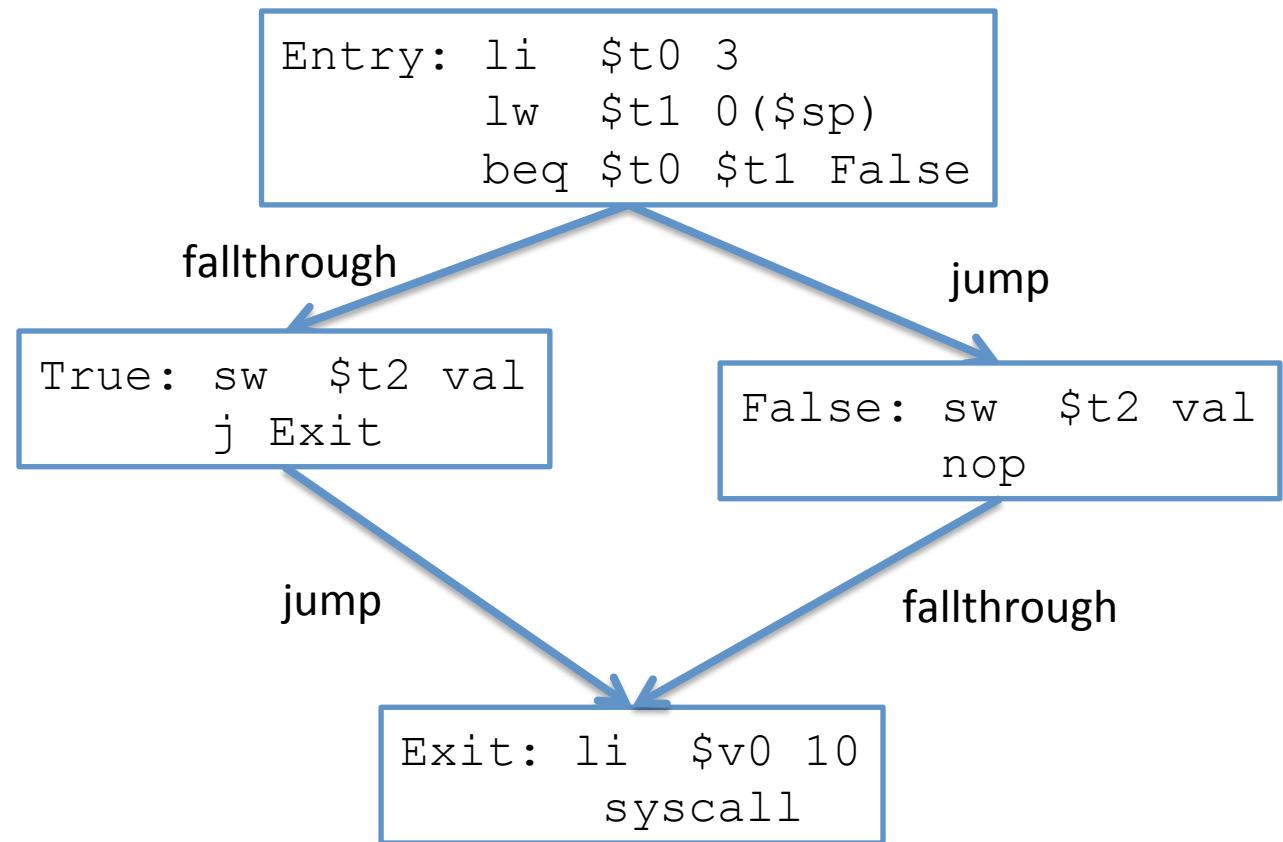
Place the exit label

# If-thenStmts

```
...  
if (val == 1) {  
    val = 2;  
}  
...  
  
lw $t0 val          # evaluate condition LHS  
sw $t0 0($sp)       # push onto stack  
subu $sp $sp 4      #  
li $t0 1             # evaluate condition RHS  
sw $t0 0($sp)       # push onto stack  
subu $sp $sp 4      #  
lw $t1 4($sp)        # pop RHS into $t1  
addu $sp $sp 4       #  
lw $t0 4($sp)        # pop LHS into $t0  
addu $sp $sp 4       #  
bne $t0 $t1 L_0     # skip if condition false  
li $t0 2             # Loop true branch  
sw $t0 val  
nop  
L_0:  
...  
# end true branch  
# branch successor
```

# Conditional Blocks

```
Entry: li $t0 3  
       lw $t1 0($sp)  
       beq $t0 $t1 False  
True:  sw $t2 val  
       j Exit  
False: sw $t2 val2  
       nop  
Exit:  li $v0 10  
       syscall
```



# Generating If-then-Else Stmt

First, get name for the false and exit labels

Generate the head of the if

- Make jumps to the (not-yet placed!) exit and false labels

Generate the true branch

- Write the body of the true node
- Jump to the (not-yet placed!) exit label

Generate the false branch

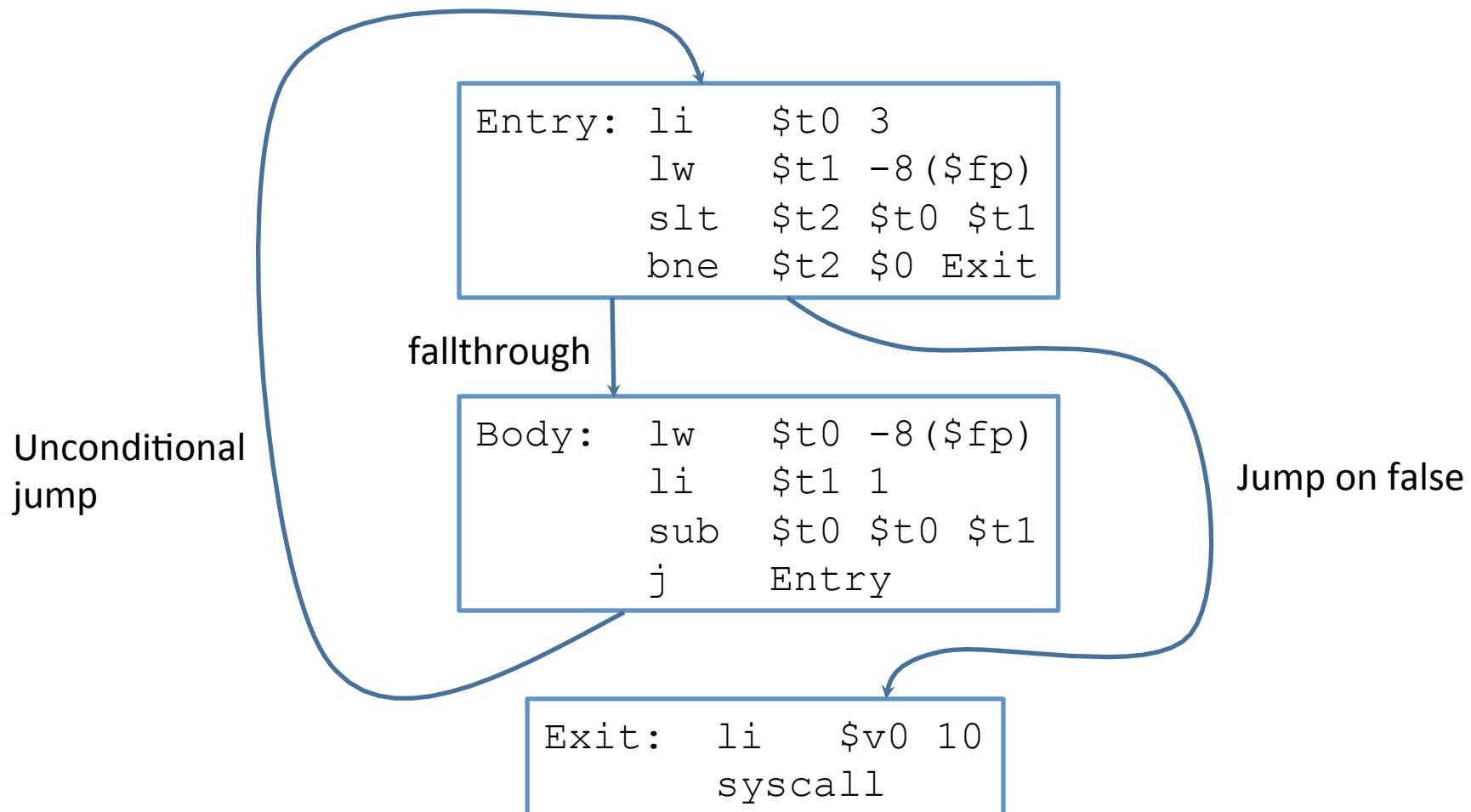
- Place the false label
- Write the body of the false node

Place the exit label

# If-then-Else Stmt

```
...  
if (val == 1) {  
    val = 2;  
} else {  
    val = 3;  
}  
...  
lw $t0 val          # evaluate condition LHS  
sw $t0 0($sp)       # push onto stack  
subu $sp $sp 4      #  
li $t0 1            # evaluate condition RHS  
sw $t0 0($sp)       # push onto stack  
subu $sp $sp 4      #  
lw $t1 4($sp)        # pop RHS into $t1  
addu $sp $sp 4       #  
lw $t0 4($sp)        # pop LHS into $t0  
addu $sp $sp 4       #  
bne $t0 $t1 L_1  # branch if condition  
                    # false  
li $t0 2            # Loop true branch  
sw $t0 val  
j L_0              # end true branch  
L_1:                # false branch  
...  
L_0:                # branch successor
```

# While Loops CFG



# Generating While Loops

Very similar to if-then stmts

- Generate a bunch of labels
- Label for the head of the loop
- Label for the successor of the loop

At the end of the loop body

- Unconditionally jump back to the head

# While Loop

```
while (val == 1) {
    val = 2;
}

L_0:
    lw $t0 val          # evaluate condition LHS
    sw $t0 0($sp)       # push onto stack
    subu $sp $sp 4      #
    li $t0 1             # evaluate condition RHS
    sw $t0 0($sp)       # push onto stack
    subu $sp $sp 4      #
    lw $t1 4($sp)        # pop RHS into $t1
    addu $sp $sp 4      #
    lw $t0 4($sp)        # pop LHS into $t0
    addu $sp $sp 4      #
    bne $t0 $t1 L_1     # branch loop end
    li $t0 2             # Loop body
    sw $t0 val
    j L_0                # jump to loop head
L_1:
    ...

```

# A Note on Conditionals

We lack instructions for branching on most relations

- No “branch if reg1 < reg2”
- Instead we use the slt “set less than”
  - `slt $t2 $t1 $t0`
    - $\$t2$  is 1 when  $\$t1 < \$t0$
    - $\$t2$  otherwise set to 0

# P6 Helper Functions

Generate (opcode, ...args...)

- Generate("add", "T0", "T0", "T1")
  - writes out add \$t0, \$t0, \$t1
- Versions for fewer args as well

Generate indexed (opcode, “Reg1”, “Reg2”, offset)

GenPush(reg) / GenPop(reg)

NextLabel() – Gets you a unique label

GenLabel(L) –Places a label

# Questions?

Looking forward

- More uses of the CFG
- Program analysis
- Optimization

Homework: see QtSpim resources

# QtSpim

# Special topics

CompCert: a formally verified compiler

What would you like to hear about?

# Course evaluations

Please complete them

# Optimization

# Roadmap

Last time:

- CodeGen for the remainder of AST nodes
- Introduced the control-flow graph

This time:

- Optimization Overview
- Discuss a couple of optimizations
  - Review CFGs

# OPTIMIZATION OVERVIEW

# Optimization goals

What are we trying to accomplish?

- Traditionally, speed
- Lower power
- Smaller footprint
- Bug resilience?

The fewer instructions the better



# Optimization guarantees

Informally: Don't change the program's output

- We may relax this to “Don't change the program's output *on good input*”
- This can actually be really hard to do

# Optimization difficulties

There's no perfect way to check equivalence of two arbitrary programs

- If there was we could use it to solve the halting problem
- We'll attempt to perform behavior-preserving transformations

# Program analysis

A perspective on optimization

- Recognize some behavior in a program
- Replace it with a “better” version

Constantly plagued by the halting problem

- We can only use approximate algorithms to recognize behavior

# Program behavior

Two terms in program analysis / behavior detection:

- Soundness: All results that are output are valid
- Completeness: All results that are valid are output

These terms are necessarily mutually exclusive

- If an algorithm was sound *and* complete, it would either:
  1. Solve the halting problem
  2. Detect a trivial property

# Back to optimization

We want our optimizations to be *sound* transformations

- In other words, they are always valid, but will miss some behaviors



# You may be thinking...

I'm sad because this makes optimization seem pretty limited



Cheer up! Our optimization may be able to detect many *practical* instances of the behavior

# Now you may be thinking...

I'm happy because I'm guaranteed that my optimization won't do any harm



Settle down! Our optimization still needs to be efficient

# Or maybe you are thinking...

I don't know how to feel about any of this  
without understanding how often it comes up



# What can we do?

We can pick some low-hanging fruit



# EXAMPLE OPTIMIZATIONS

# Peephole optimization

A naïve code generator tends to output some silly code

- Err on the side of correctness over efficiency

Pattern-match the most obvious problems

# CFG for program analysis

Consider the following sequence of instructions:

```
push { sw    $t0 0 ($sp)
       subu $sp $sp 4
pop  { lw    $t0 4 ($sp)
       addu $sp $sp 4
```

We'd like to remove this sequence...

- Is it sound to do so?
- Maybe not!

# Review: the CFG

Program as a flowchart

Nodes are “Basic Blocks”

Edges are control transfers

- Fallthrough
- Jump
- *Maybe* function calls

# CFG for optimization

We can limit our peephole optimizations to *intra-block* analysis

- This ensures, by definition, that no jumps will intrude on the sequence

We will assume for the rest of our peephole optimizations that instruction sequences are in one block

# Peephole examples

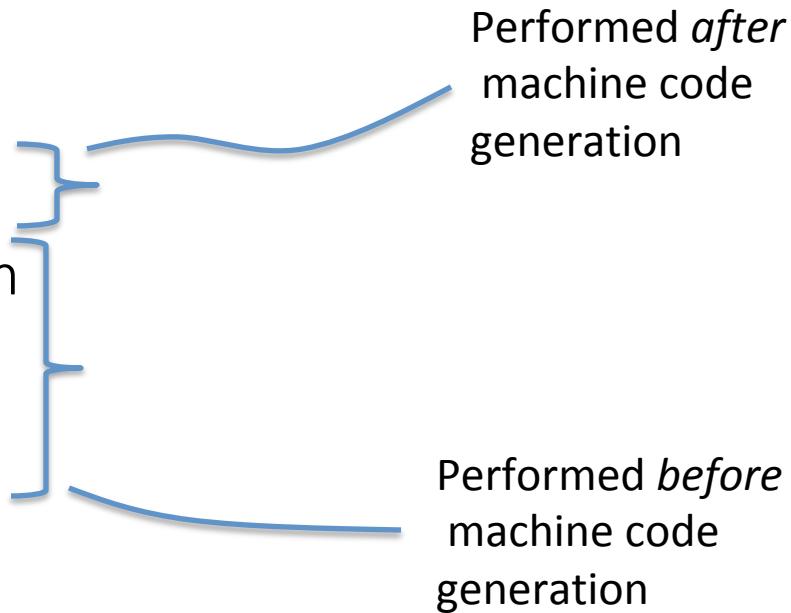
Called “peephole” optimization because we are conceptually sliding a small window over the code, looking for small patterns



# Outline

Four different optimizations

- Peephole optimization
- Loop-Invariant Code Motion
- For-loop strength reduction
- Copy propagation



# Peephole optimization 1

Remove no-op sequences

- Push followed by pop

```
push { sw    $t0 0 ($sp)
       subu $sp $sp 4
       lw    $t0 4 ($sp)
       addu $sp $sp 4
```

- Add/sub 0

```
addu $t1 $t1 0
```

- Mul/div 1

```
mul $t2 $t2 1
```

# Peephole optimization 2

Simplify sequences

- Ex. Store then load
- Strength reduction

```
sw    $t0  -8 ($fp)
lw    $t0  -8 ($fp)
mul   $t1  $t1  2
add   $t2  $t2  1
```

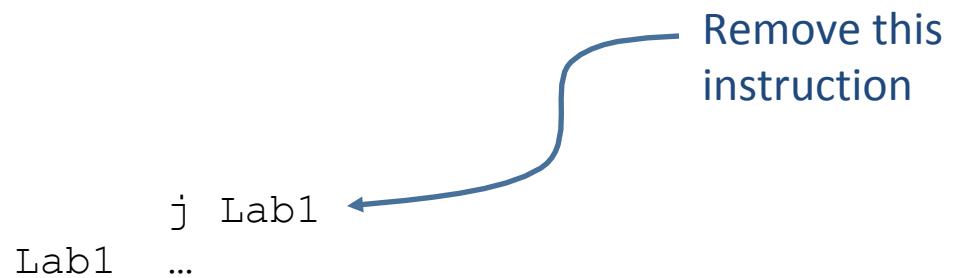
Useless instruction

shift-left \$t1

inc \$t2

# Peephole optimization 3

Jump to next instruction



# Loop invariant code motion

## Loop Invariant Code Motion (LICM)

- Don't duplicate effort in a loop

### Goal

- Pull code out of the loop
- “Loop hoisting”

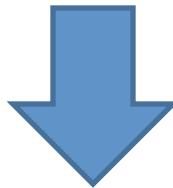
Important due to “hot spots”

- Most execution time due to small regions of deeply-nested loops

# LICM example

```
for (i=0; i<100; i++) {  
    for (j=0; j<100; j++) {  
        for (k=0; k<100; k++) {  
            A[i][j][k] = i*j*k  
        }  
    }  
}
```

Sub-expression  
*invariant* with respect to  
Innermost loop



```
for (i=0; i<100; i++) {  
    for (j=0; j<100; j++) {  
        temp = i * j  
        for (k=0; k<100; k++) {  
            A[i][j][k] = temp * k  
        }  
    }  
}
```

# LICM: When should we do it?

In the previous example,  
showed LICM on source  
code

At IR level, more candidate  
operations

Assembly might be *too*  
low-level

- Need a guarantee that the  
loop is *natural*
  - No jumps into the loop

```
tmp0 = FP + offsetA
for (i=0; i<100; i++) {
    tmp1 = tmp0 - i*40000
    for (j=0; j<100; j++) {
        tmp2 = ind2
        tmp3 = i*j
        for (k=0; k<100; k++) {
            T0 = tmp3 * k
            T1 = tmp2 - k*4
            store T0, 0(T1)
        }
    }
}
```

# LICM: How should we do it?

Two factors, which really generalize to optimization:

- Safety
  - Is the transformation semantics-preserving?
    - Make sure the operation is truly loop-invariant
    - Make sure ordering of events is preserved
- Profitability
  - Is there any advantage to moving the instruction?
    - May end up moving instructions that are never executed
    - May end up performing more intermediate computation than necessary

# Other loop optimizations

## Loop unrolling

- For a loop with a small, constant number of iterations, we may actually save time by just placing every copy of the loop body in sequence (no jumps)
- May also consider doing multiple iterations within the body

## Loop fusion

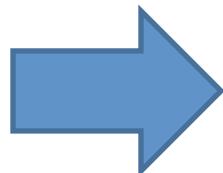
- Merge two sequential, independent loops into a single loop body (fewer jumps)

# Jump optimizations

**Disclaimer: Require some extra conditions**

## Jump around jump

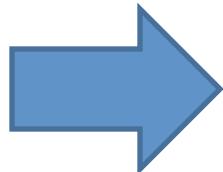
```
beq $t0,$t1,Lab1  
j Lab2  
Lab1: ...  
...  
Lab2: ...
```



```
bne $t0,$t1,Lab2  
Lab1: ...  
...  
Lab2: ...
```

## Jump to jump

```
j Lab1  
...  
Lab1: j Lab2  
...  
Lab2: ...
```



```
j Lab2  
...  
Lab1: j Lab2  
...  
Lab2: ...
```

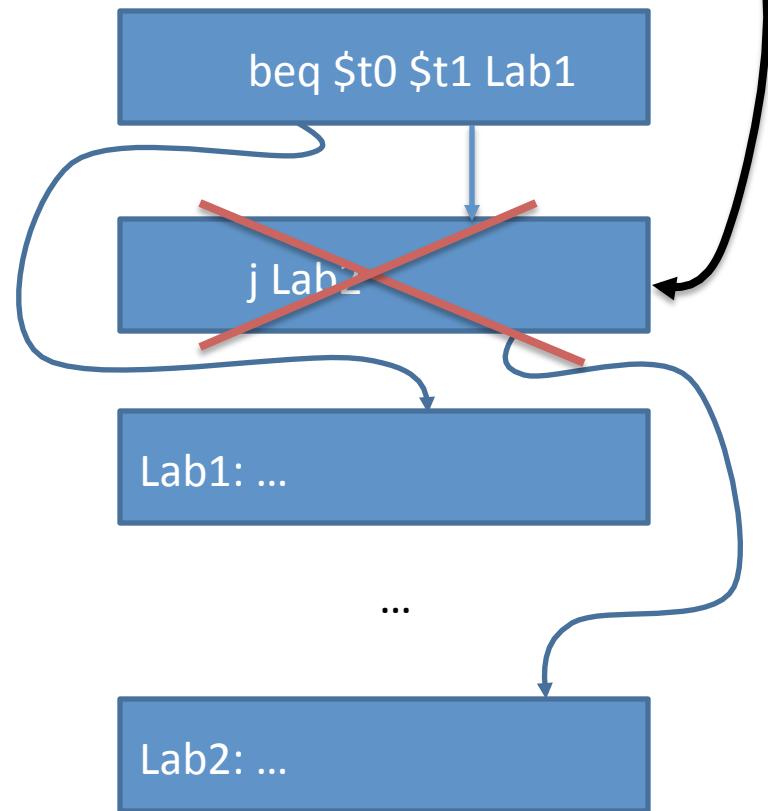
# Intraprocedural analysis

The past two optimizations had some caveats

- There may be a jump into your eliminated code

We'd like to introduce a control-flow concept beyond basic blocks:

- Guarantee that block1 must be executed in order to get to block2
  - This goes by a pretty boring name





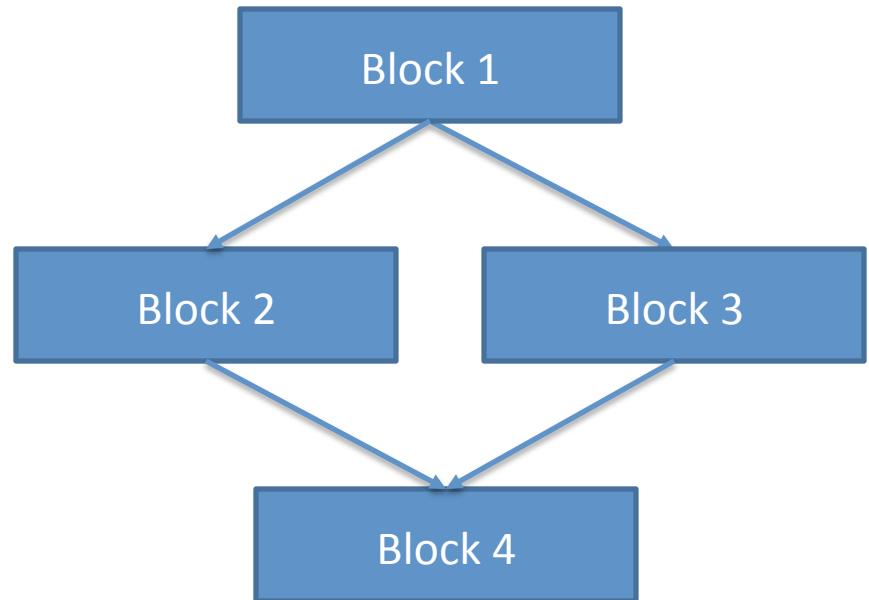
**Domination**

# Dominators and post-dominators

Control Flow Graph

We say that block A dominates block B if A must be executed before B is executed

We say that block A postdominates block B if A must be executed after B



# Semantics preserving

Do we really need semantics preserving optimizations?

Are there examples where we don't?

# In summary

Today

- Saw the basics of optimizations
- Soundness vs completeness
- Peephole and simple optimizations

Next time

- Wrap up optimizations
- Basics of static analysis

# Announcements

P6, H10 assigned

# Final exam

Final exam will cover *all* material covered in course – i.e., all compiler phases

- Focus will be on after-midterm material

There will be some questions on special topics

100 minutes exam, similar in format to midterm

# Optimization Frameworks

# Roadmap

Last time:

- Optimization overview
  - Soundness and completeness
- Simple optimizations
  - Peephole
  - LICM

This time:

- More Optimization
- Analysis frameworks

# Outline

Review Dominators

Introduce some more advanced concepts

- Static single assignment (SSA)
- Dataflow propagation

# **DOMINATOR REVIEW**

# Dominator terms

Domination (A dominates B):

- to reach block B, you must have gone through block A

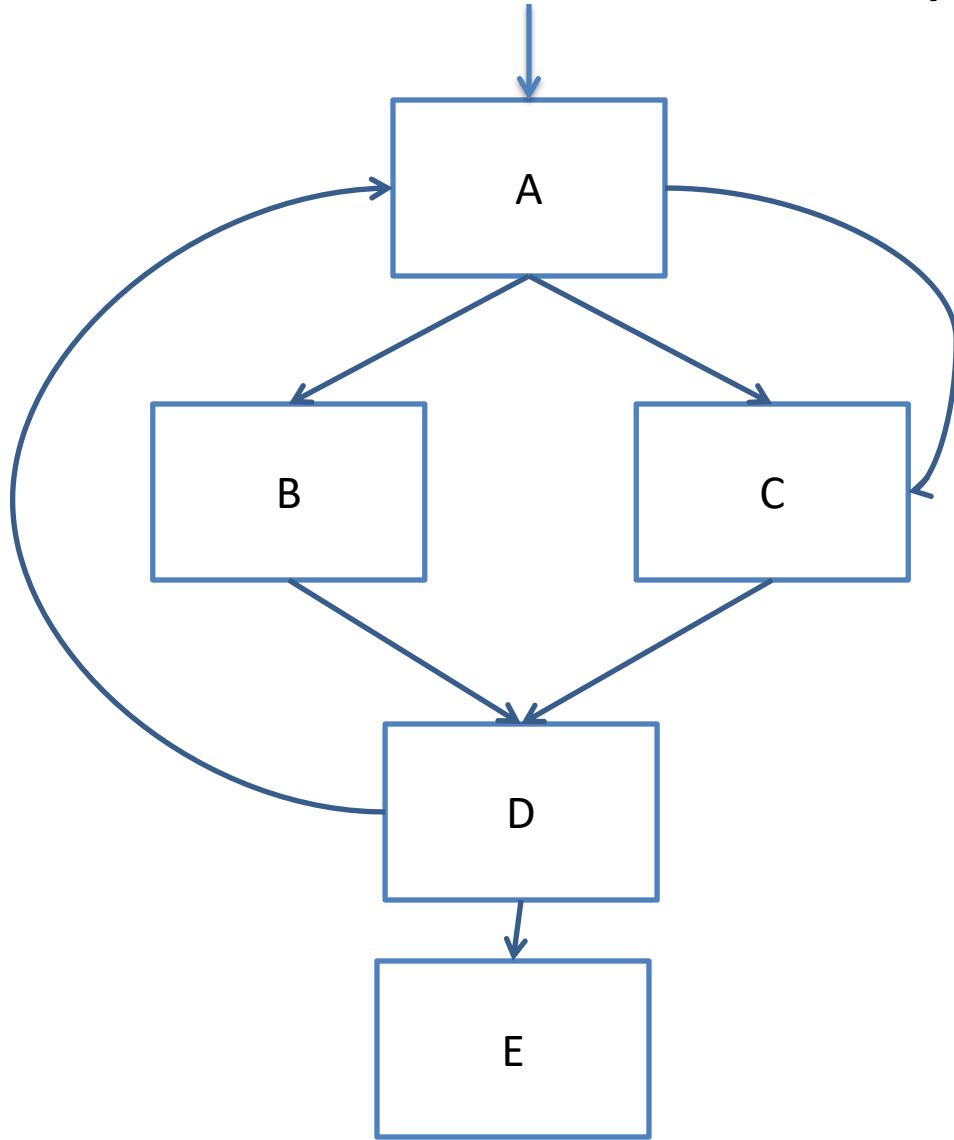
Strict Domination (A strictly dominates B)

- A dominates B and A is not B

Immediate Domination (A immediately dominates B)

- A immediately dominates B if A dominates B and has no intervening dominators

# Dominator example

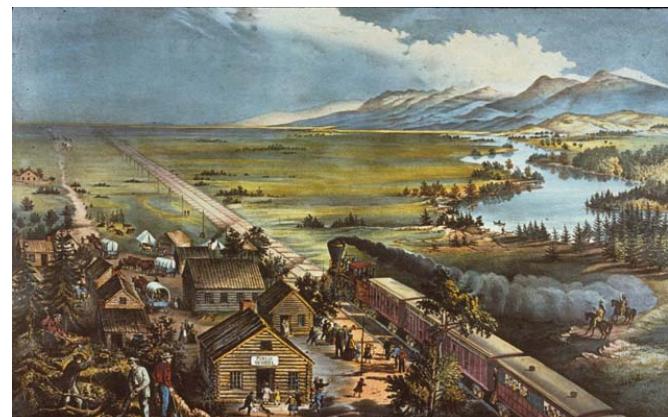


# Dominance frontier

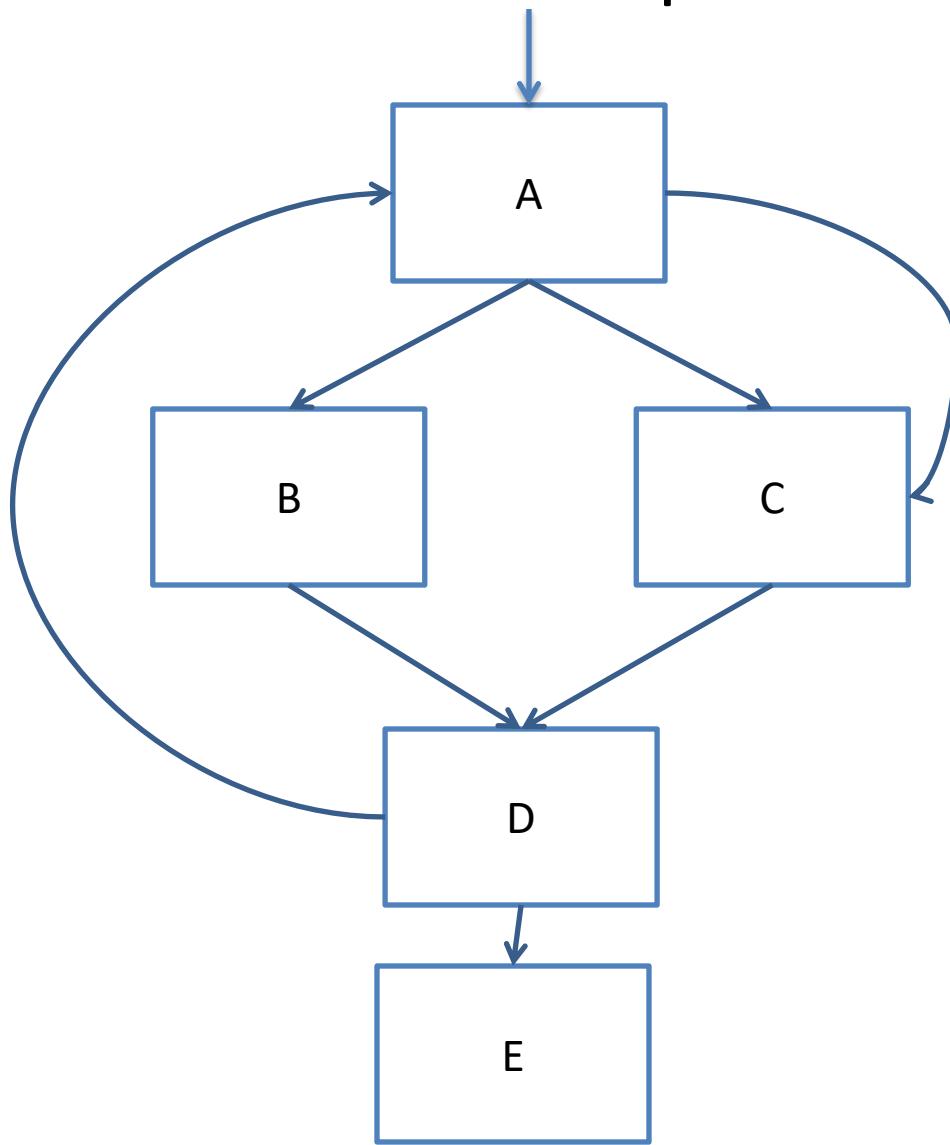
*Definition:*

For a block  $x$ , the dominance frontier of  $x$  is the set of nodes  $Y$  such that

- $x$  dominates an immediate predecessor of  $Y$
- $x$  does not strictly dominate  $Y$



# Example



# **STATIC SINGLE ASSIGNMENT**

# Goal of SSA Form

Build an intermediate representation of the program in which each variable is assigned a value in at most 1 program point:



$x = 1$   
 $z = 2$   
 $y = 3$



$x = y$   
 $z = y$   
 $w = z$



$x = 1$   
 $x = 2$   
 $y = 3$



$i = 0;$   
 $\text{while}( i < 10) \{$   
           $k = i + 1;$   
     $\}$

Statically: There is at most *one* assignment statement  
that assigns to k

Dynamically: k can be assigned to *multiple* times

# Conversion

We'll make new variables to carry over the effect of the original program



$x = 1$

$x = x$

$y = x$



$x_1 = 1$

$x_2 = x_1$

$y_1 = x_2$

# Benefits of SSA Form

There are some obvious advantages to this format for program analysis

- Easy to see the *live range* of a given variable  $x$  assigned to in statement  $s$ 
  - The region from “ $x = \dots;$ ” until the last use(s) of  $x$  before  $x$  is redefined
  - In SSA form, from “ $x_i = \dots;$ ” to all uses of  $x_i$ , e.g., “ $\dots = f(\dots, x_i, \dots);$ ”
- Easy to see when an assignment is *useless*
  - We have “ $x_i = \dots;$ ” and there are *no uses* of  $x_i$  in any expression or assignment RHS
  - “ $x_i = \dots;$ ” is a useless assignment”
  - “ $x_i = \dots;$ ” is dead code”

In other words, some useful information is pre-computed, or at least easily recoverable

Warning 1: Dead code = useless assignments + unreachable code

# Optim

At “if ( $b < 4$ )”,  $b$  is only reached by “ $b = 2;$ ”  
Therefore, the else branch is unreachable  
(dead), and can be removed

# Helps

## Dead-Code Elimination

```
int a = 0;  
int b = 2;
```

```
if (g < 12) {  
    a = 1;  
} else {  
    if (b < 4) {  
        a = 2;  
    } else {  
        a = 3;  
    }  
}  
b = a;  
return 2;
```

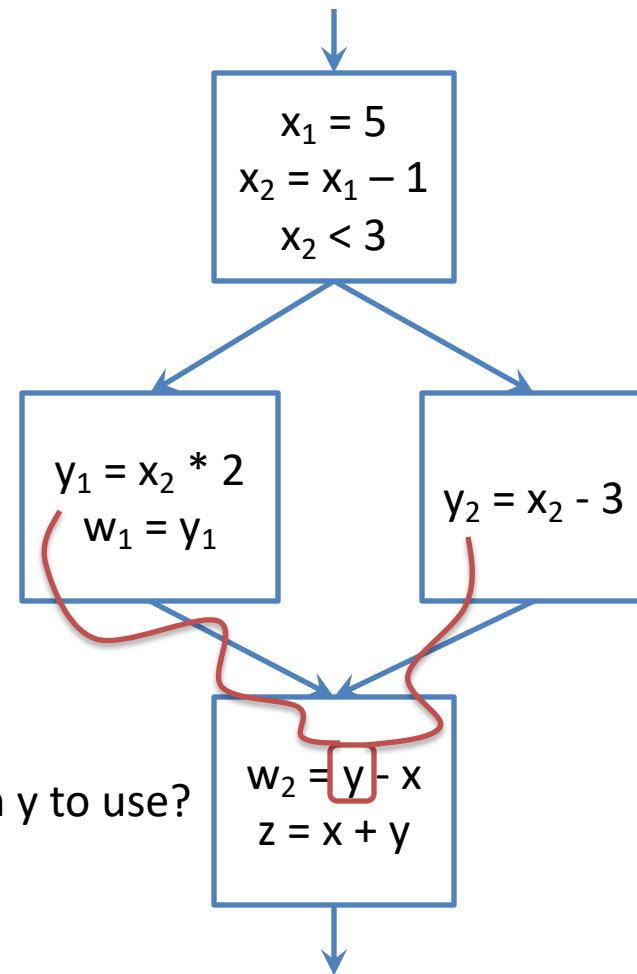
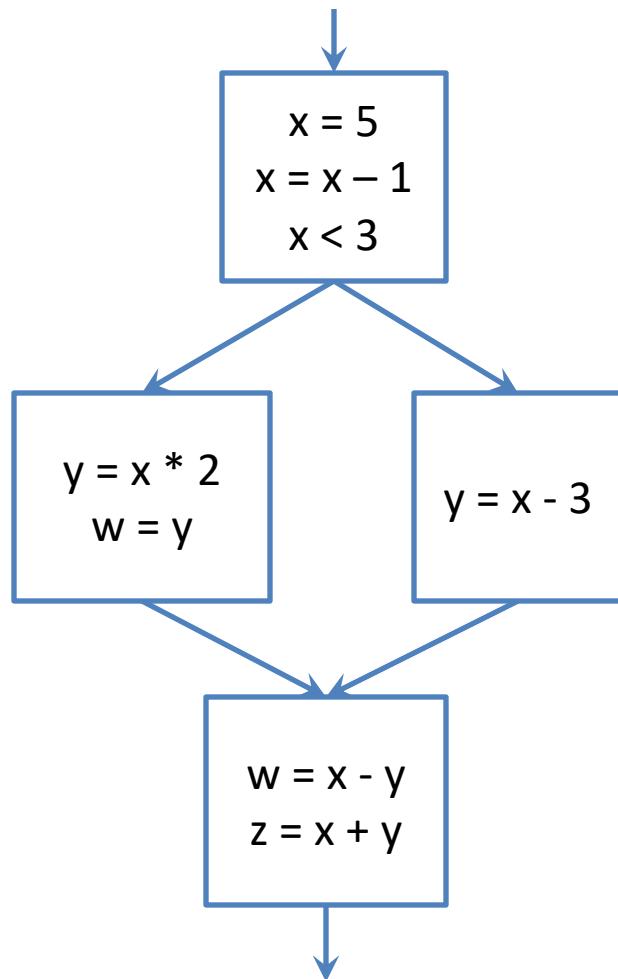
```
int a1 = 9;  
int b1 = 2;  
  
if (g1 < 12) {  
    a2 = 1;  
} else {  
    if (b1 < 4) {  
        a3 = 2;  
    } else {  
        a4 = 3;  
    }  
    a5 = φ(a3, a4);  
}  
a6 = φ(a2, a5);  
b2 = a6;  
return 2;
```

# Optimizations Where SSA Helps

## Constant-propagation/constant-folding

```
int a = 30;           6
int b = 9; - (a / 5);
int c;
c = b * 4; true 12
if (c>=10) {
    c = 2; - 10; 2
}
return 4; ^ 260 4 a);
```

# What About Conditionals?



Which  $y$  to use?

# Phi Functions ( $\phi$ )

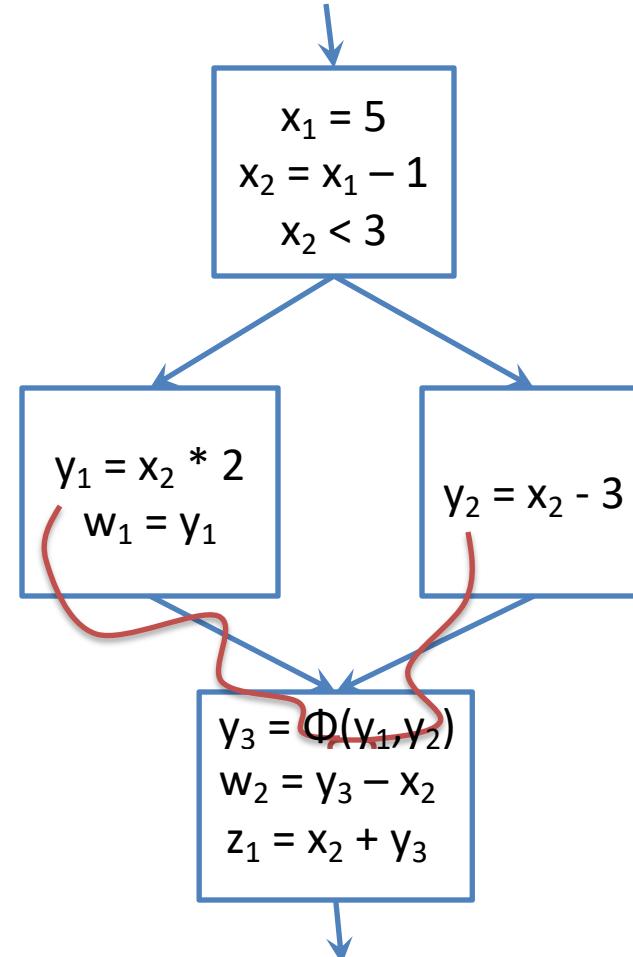
We introduce a special symbol  $\Phi$  at such points of confluence

$\Phi$ 's arguments are all the instances of variable  $y$  that might be the most recently assigned variant of  $y$

Returns the “correct” one

Do we need a  $\Phi$  for  $x$ ?

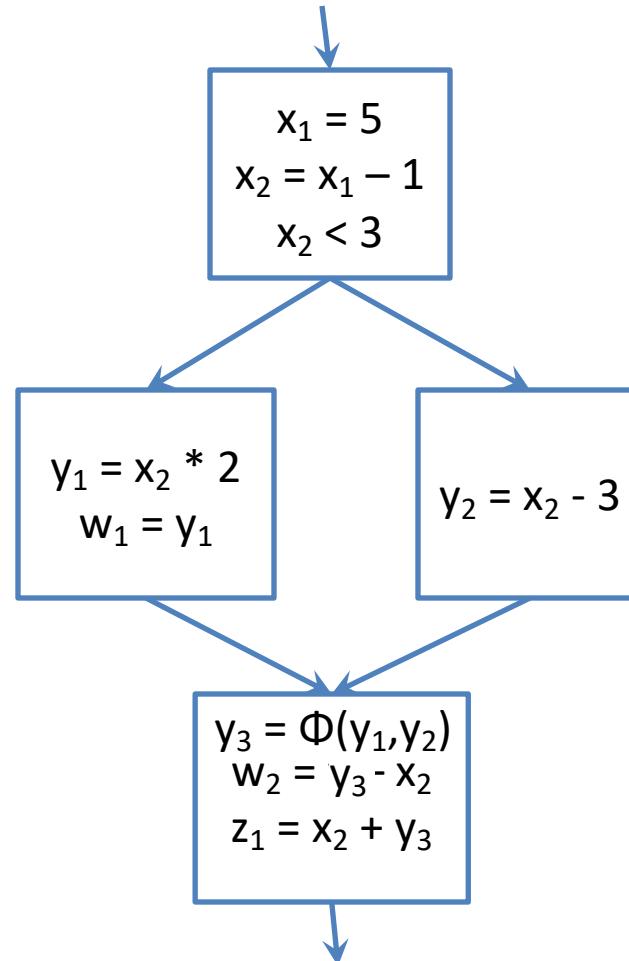
– No!



# Computing Phi-Function Placement

Intuitively, we want to figure out cases where there are multiple assignments that can reach a node

To be safe, we can place a  $\Phi$  function for each assignment at every node in the *dominance frontier*



# Pruned Phi Functions

This criterion causes a bunch of useless  $\Phi$  functions to be inserted

- Cases where the result is never used “downstream”  
(useless)

*Pruned SSA* is a version where useless  $\Phi$  nodes are suppressed

# DATAFLOW ANALYSIS

# Dataflow framework idea

Many analyses can be formulated as how data is transformed over the control flow graph

Propagate static information from:

- the beginning of a single basic block
- the end of a single basic block
- The join points of multiple basic blocks

# Dataflow framework idea

Meet Lattice

Transfer function

- How data is propagated from one end of a basic block to the other

Meet operation

- Means of combining lattice between blocks

# Dataflow analysis direction

## Forward analysis

- Start at the beginning of a function's CFG, work along the control edges

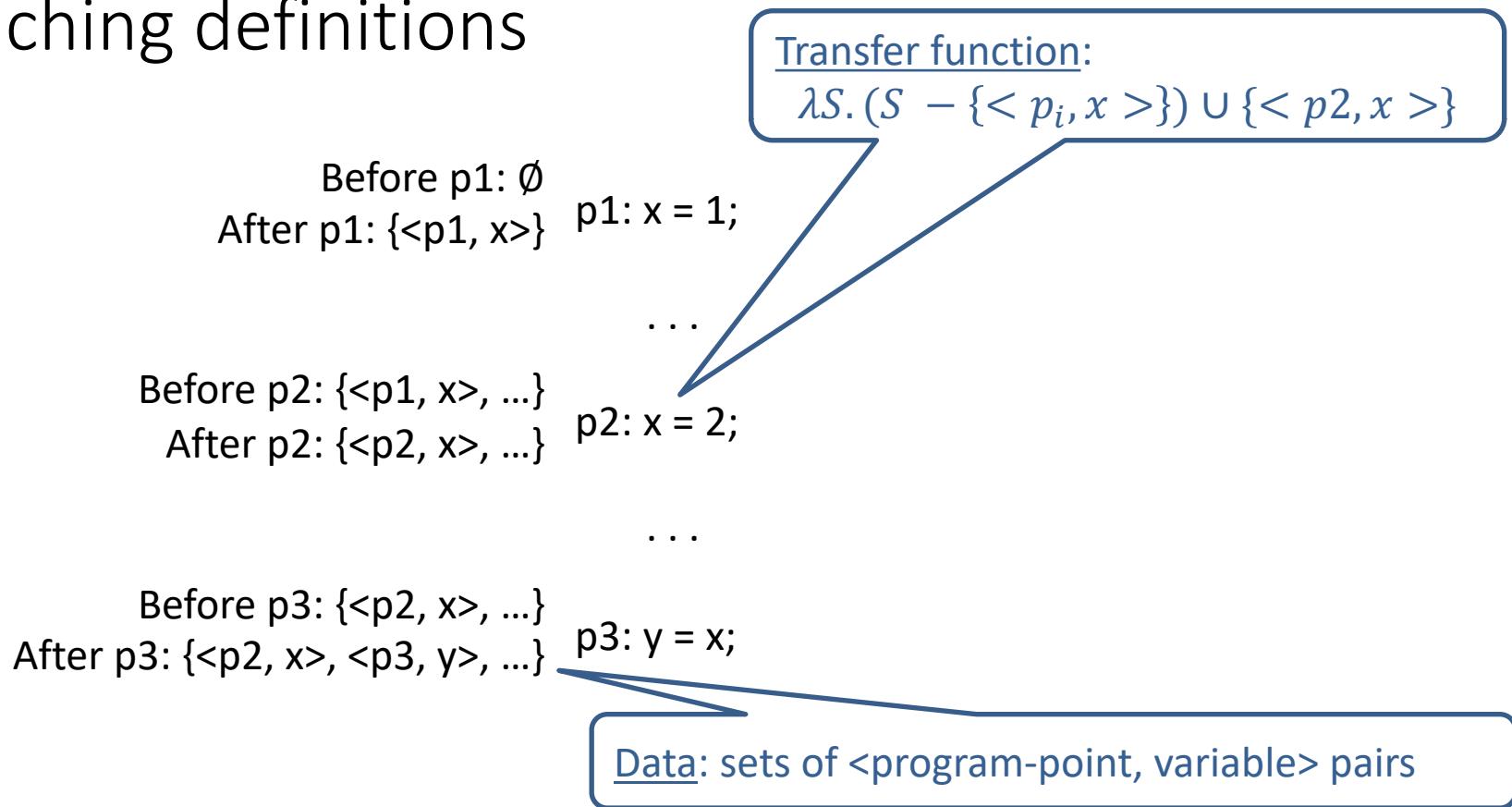
## Backwards analysis

- Start at the end of a function's CFG, work against the control edges

Continuously propagate values until there is no change

# Dataflow-Analysis Example 1

## Reaching definitions



Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.

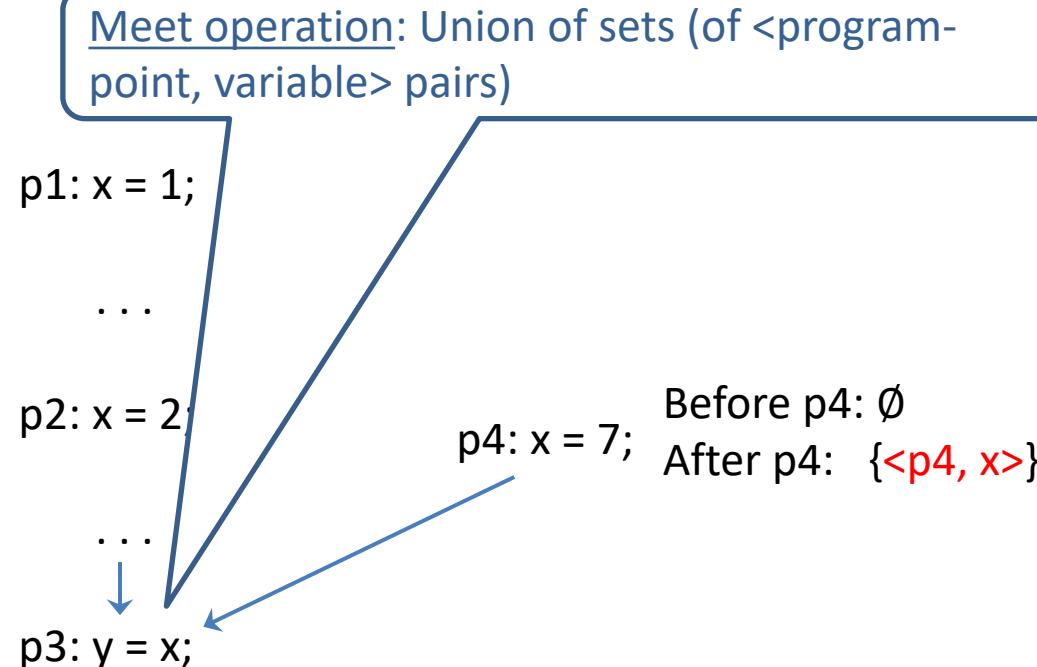
# Dataflow-Analysis Example 1

## Reaching definitions

Before p1:  $\emptyset$   
After p1: { $\langle p1, x \rangle$ }

Before p2: { $\langle p1, x \rangle, \dots$ }  
After p2: { $\langle p2, x \rangle, \dots$ }

Before p3: { $\langle p2, x \rangle, \langle p4, x \rangle, \dots$ }  
After p3: { $\langle p2, x \rangle, \langle p3, y \rangle, \langle p4, x \rangle, \dots$ }

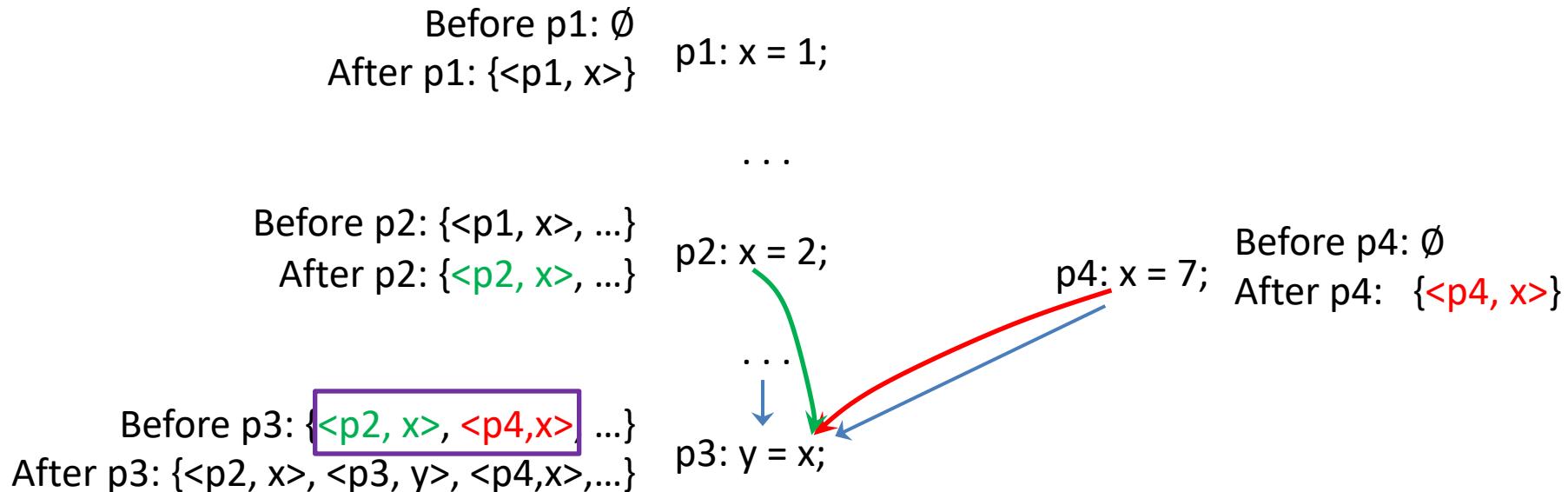


Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.

# Dataflow-Analysis Example 1

Reaching definitions: Why is it useful?

Answers the question “Where could this variable have been defined?”



# Dataflow-Analysis Example 2

## Live Variables

Before p1:  $\emptyset$   
After p1:  $\{x\}$

Before p2:  $\{x\}$   
After p2:  $\{x, y\}$   
Before p3:  $\{x, y\}$   
After p3:  $\emptyset$

Before p4:  $\emptyset$   
After p4:  $\{x\}$   
Before p5:  $\{x\}$   
After p5:  $\{x\}$   
Before p6:  $\{x\}$   
After p6:  $\emptyset$

p1:  $x = 1;$

if (...) {  
p2:  $y = 0;$

p3:  $z = x + y;$

}

p4:  $x = 2;$

p5:  $z = 3;$

p6:  $\text{cout} \ll x;$

Transfer function:

$$\lambda S. (S - \{z\}) \cup \{x, y\}$$

Data: sets of variables

$z$  is not live after p5, and  
thus p5 is a useless  
assignment (= dead code)

# The end: or is it?

Covered a broad range of topics

- Some formal concepts
- Some practical concepts

What we skipped

- Linking and loading
- Interpreters
- Register allocation
- Performance analysis / Proofs