

Checkin 3

Write a Flex rule for Fortran real literals:

- An optional sign (+ or -)
- One of the following:
 - An integer
 - One or more digits followed by a '.' followed by 0 or more digits
 - a '.' followed by one or more digits

Checkin 3

Write a Flex rule for Fortran real literals:

- An optional sign (+ or -)
- One of the following:
 - An integer
 - One or more digits followed by a '.' followed by 0 or more digits
 - a '.' followed by one or more digits

Administrivia

Housekeeping

P1 Out

KU | EECS | Drew Davidson



EECS 665

COMPILER

CONSTRUCTION

Lecture 3

Syntactic Definition

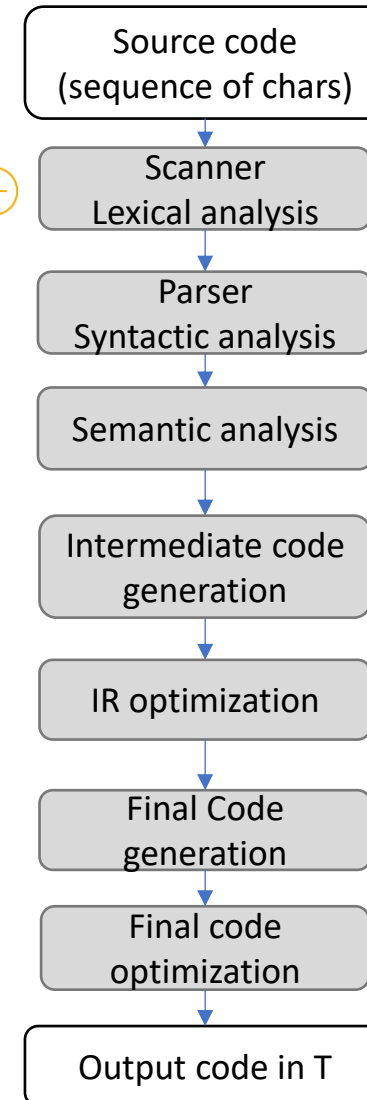
Compiler Construction

Progress Pics

Done:

- Shown RegEx as a good token formalism
- Lexical specification
- Lexical recognition

In progress



Last Time

Lecture Review



Replace sub-RegExes with sub-FSMs “bottom-up” in the expression tree

Use the ϵ -closure to “bypass middleman” states and transitions

Create a DFA that tracks all possible states the NFA could be in

Good news

DFAs have a natural implementation (use a 2-D array)

Bad news

DFAs don't exactly do tokenization

Today's Lecture

Outline

Finish lexical analysis

- How to build a scanner

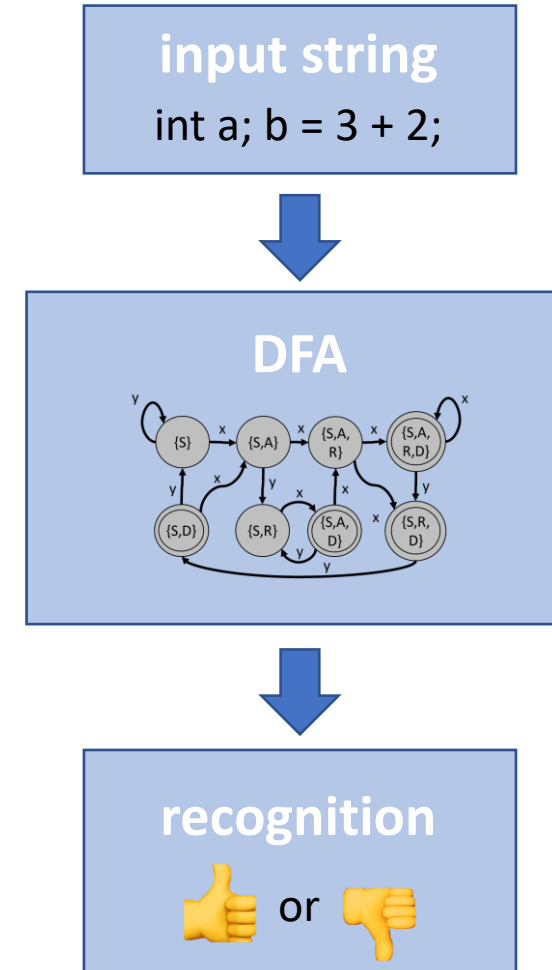
Begin discussing syntax

- How to specify the syntax
of a programming
language

DFA $\neq$ Tokenizer

Limitations

- Finite automata only check for language membership of a string (recognition)
- The Scanner needs to
 - Break the input into many different tokens
 - Know what characters comprise the token



Lecture Outline

Syntactic Definition

From DFAs to Tokenizer

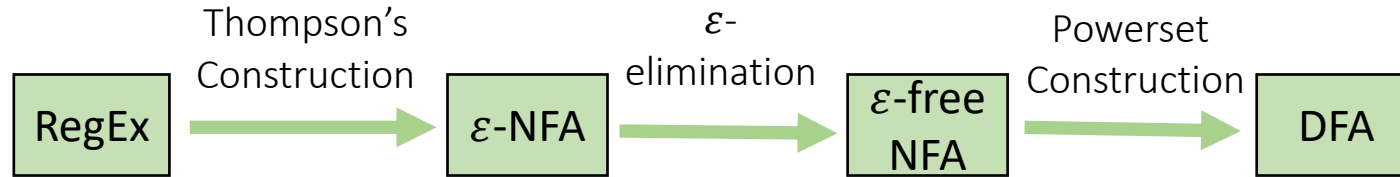
- Algorithms
- Implementation details

How Language Syntax is Formally Defined: CFGs

- Why we need context-free grammars
- How we use context-free grammars

From RegExes to Tokenizer

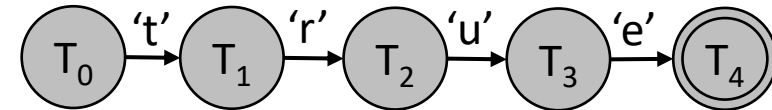
Algorithms



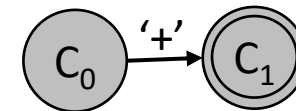
Language

true { Token(**true**) }
'+' { Token(**cross**) }
'-' { Token(**dash**) }

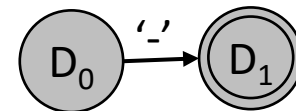
true



cross

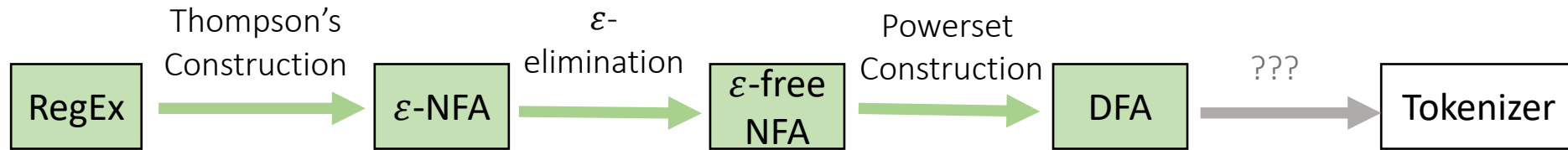


dash



From RegExes to Tokenizer

Algorithms



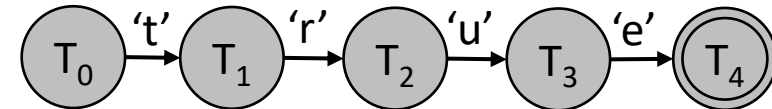
1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

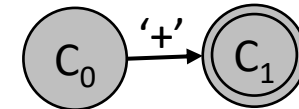
Language

true { Token(**true**) }
'+' { Token(**cross**) }
'-' { Token(**dash**) }

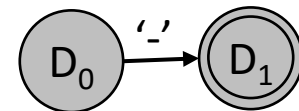
true



cross



dash



From RegExes to Tokenizer

Algorithms

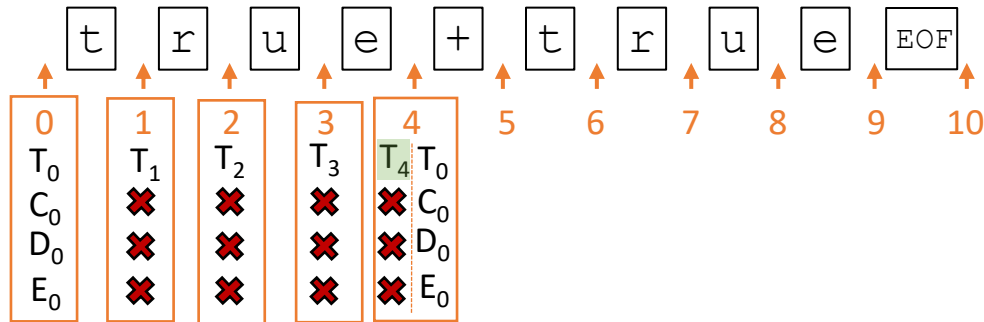
1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

Language

true {Token(**true**)}
'+' {Token(**cross**)}
'-' {Token(**dash**)}

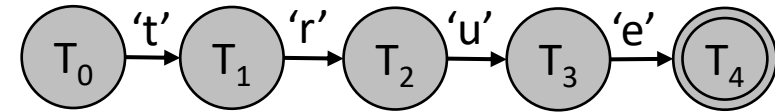
Char Stream



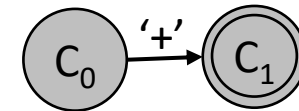
Token Stream

true
[0,4)

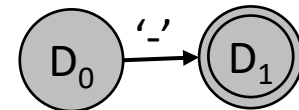
true



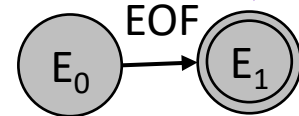
cross



dash



eof



Special State:
Return EOF
Accept stream

From RegExes to Tokenizer

Algorithms

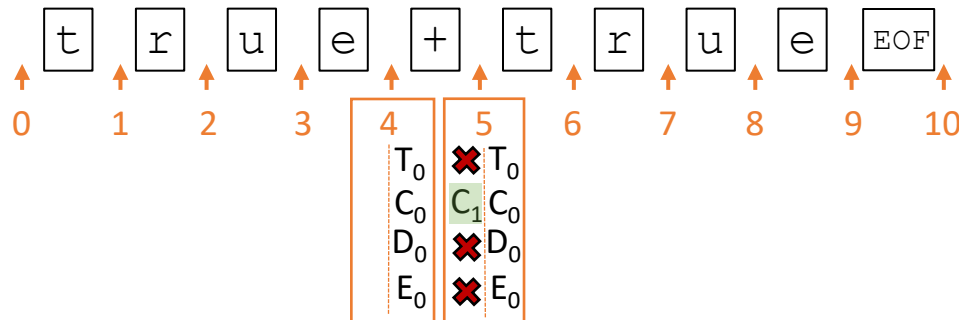
1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

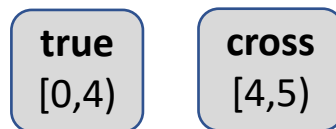
Language

true {Token(**true**)}
'+' {Token(**cross**)}
'-' {Token(**dash**)}

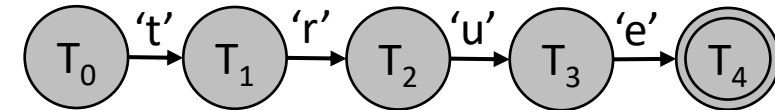
Char Stream



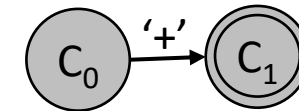
Token Stream



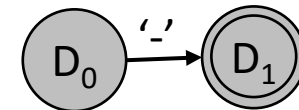
true



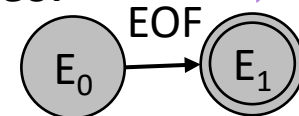
cross



dash



eof



Special State:
Return EOF
Accept stream

From RegExes to Tokenizer

Algorithms

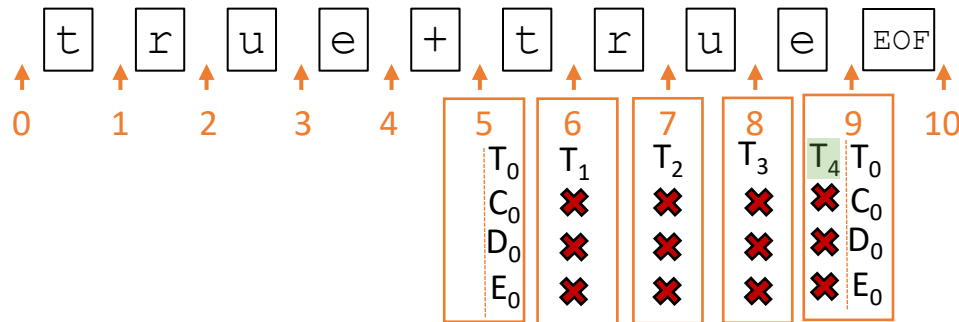
1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

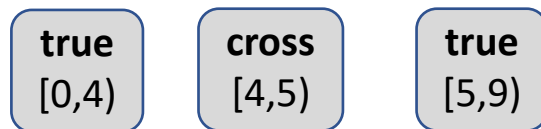
Language

true {Token(**true**)}
'+' {Token(**cross**)}
'-' {Token(**dash**)}

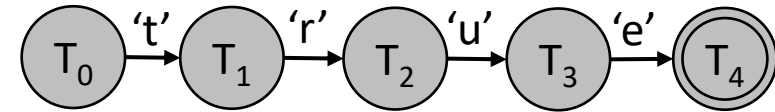
Char Stream



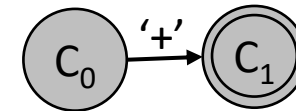
Token Stream



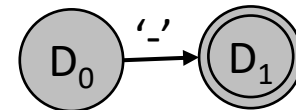
true



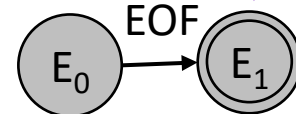
cross



dash



eof



Special State:
Return EOF
Accept stream

From RegExes to Tokenizer

Algorithms

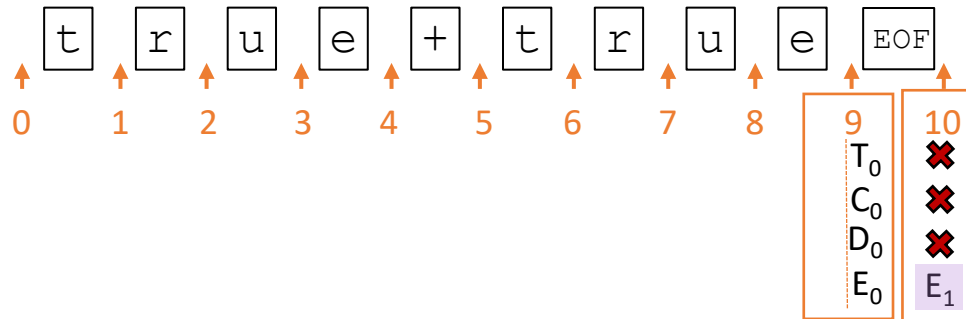
1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

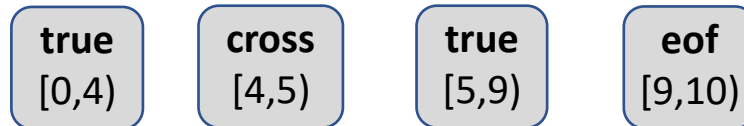
Language

true {Token(**true**)}
'+' {Token(**cross**)}
'-' {Token(**dash**)}

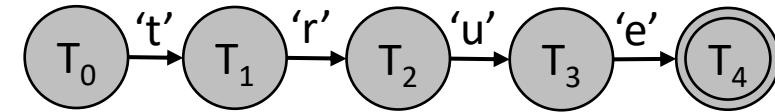
Char Stream



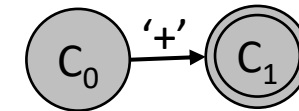
Token Stream



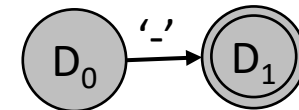
true



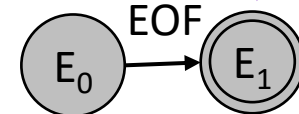
cross



dash



eof



Special State:

Return EOF
Accept stream

From RegExes to Tokenizer

Algorithms

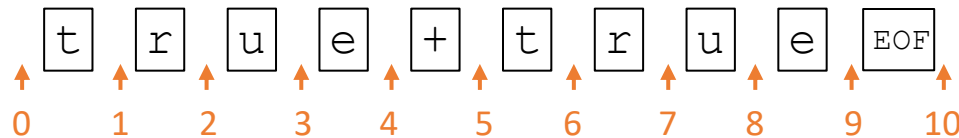
1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

Language

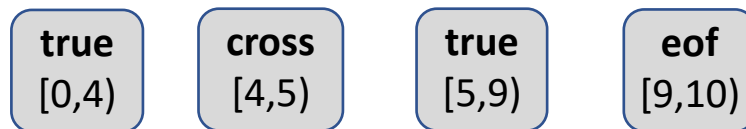
true {Token(**true**)}
'+' {Token(**cross**)}
'-' {Token(**dash**)}

Char Stream

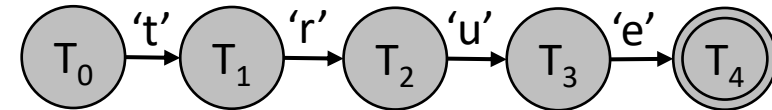


Accept!

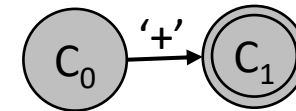
Token Stream



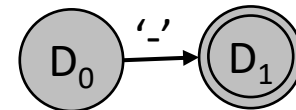
true



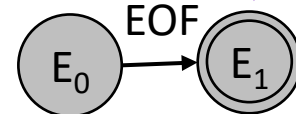
cross



dash



eof



Special State:
Return EOF
Accept stream

Problem

What happens when token languages overlap / prefix each other?

From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

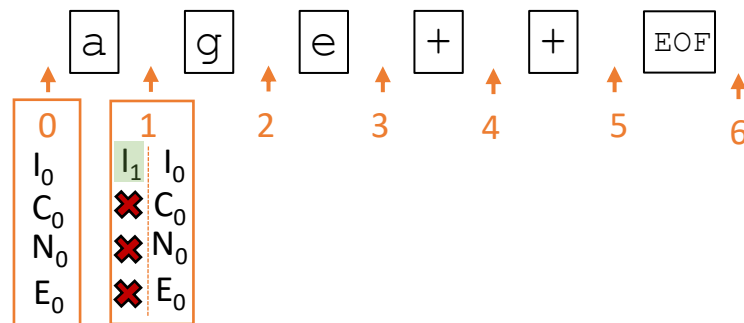
Problem

What happens when token languages overlap / prefix each other?

Language

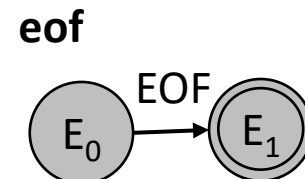
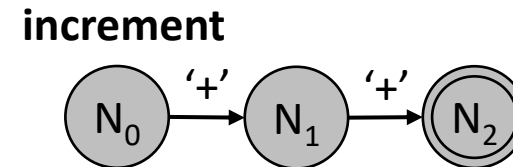
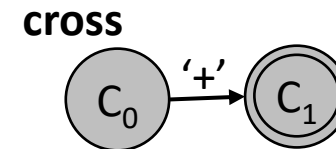
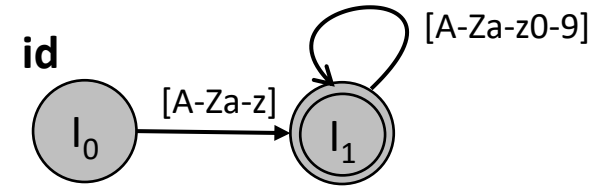
$[A-Za-z][A-Za-z0-9]^*$ { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream



Token Stream

id: a
[0,1)



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

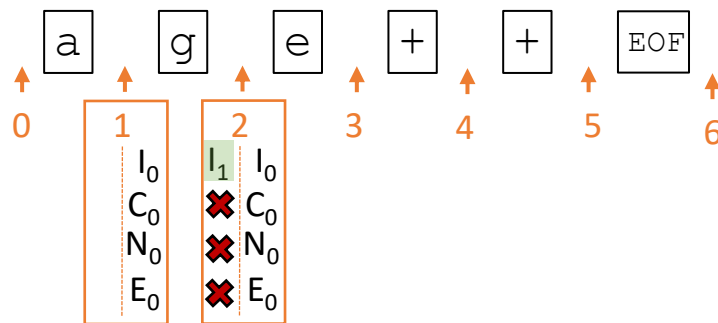
Problem

What happens when token languages overlap / prefix each other?

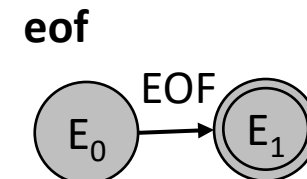
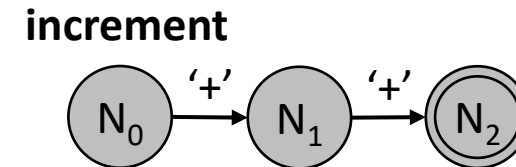
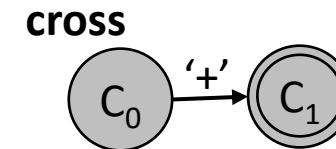
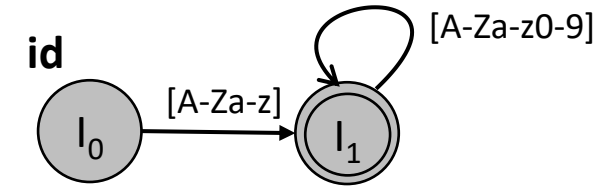
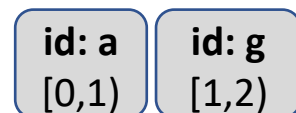
Language

[A-Za-z][A-Za-z0-9]* { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream



Token Stream



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

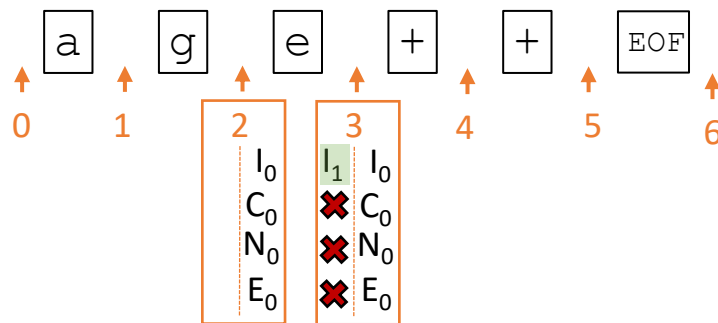
Problem

What happens when token languages overlap / prefix each other?

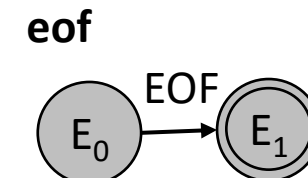
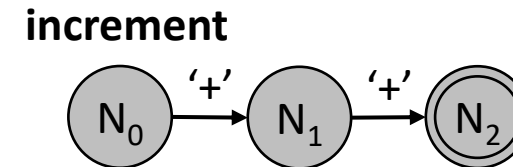
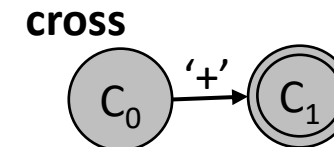
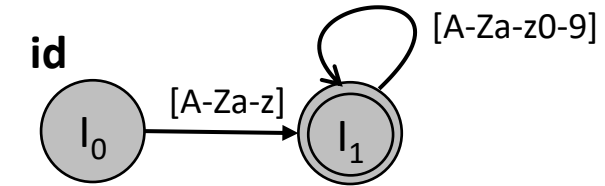
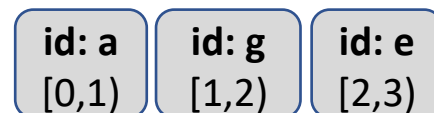
Language

$[A-Za-z][A-Za-z0-9]^*$ { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream



Token Stream



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

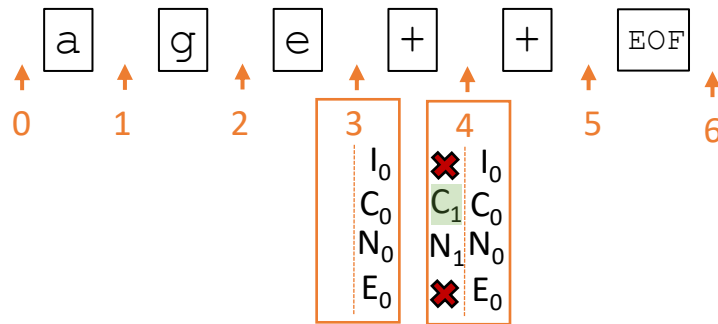
Problem

What happens when token languages overlap / prefix each other?

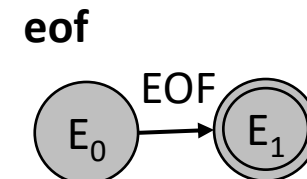
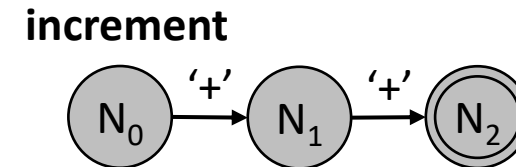
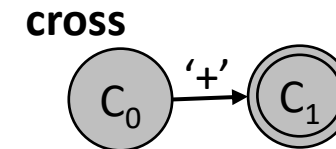
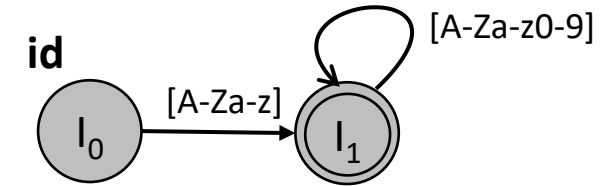
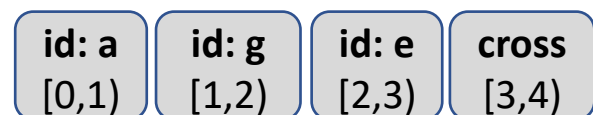
Language

[A-Za-z][A-Za-z0-9]* { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream



Token Stream



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

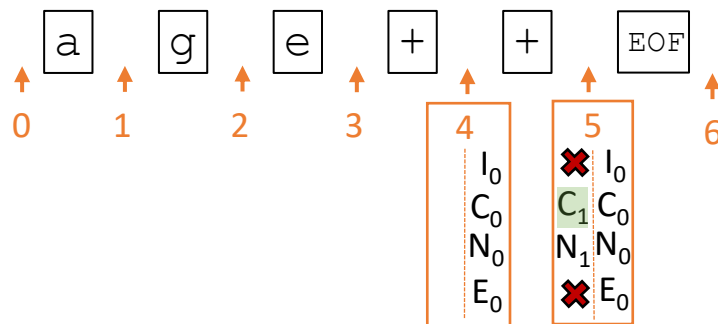
Problem

What happens when token languages overlap / prefix each other?

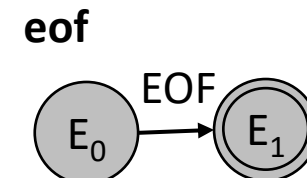
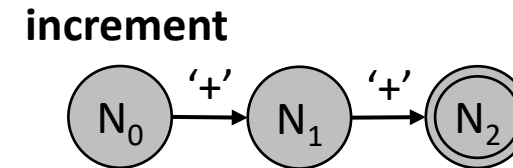
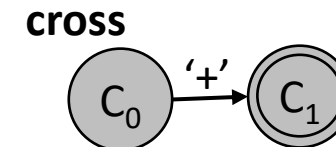
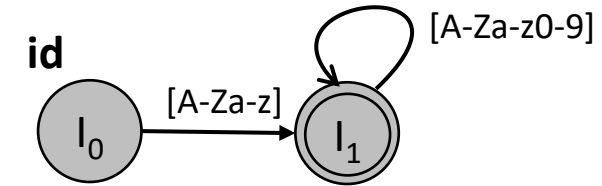
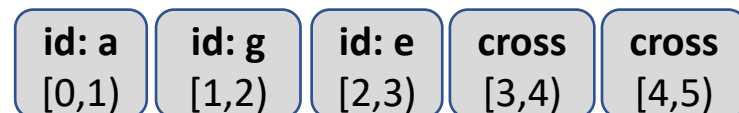
Language

$[A-Za-z][A-Za-z0-9]^*$ { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream



Token Stream



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

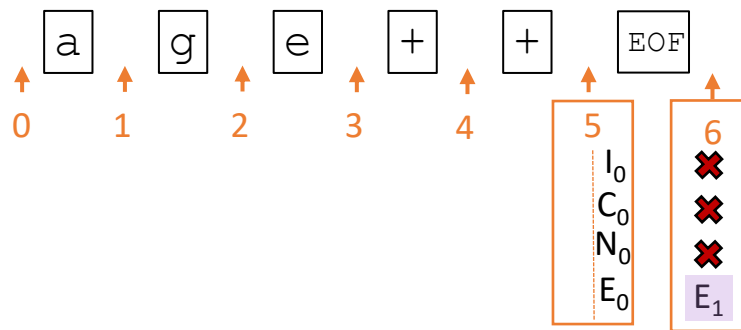
Problem

What happens when token languages overlap / prefix each other?

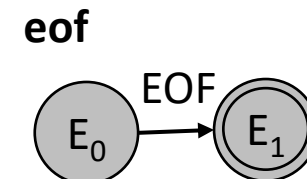
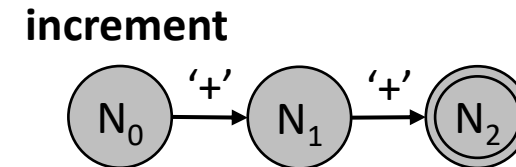
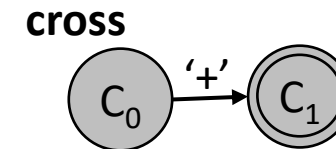
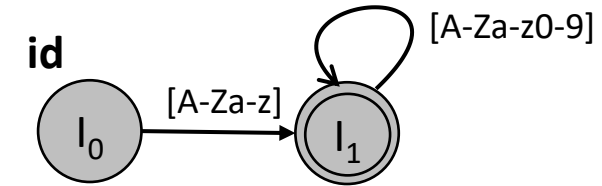
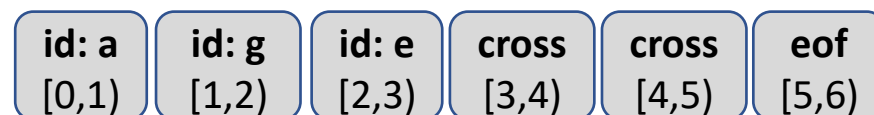
Language

$[A-Za-z][A-Za-z0-9]^*$ { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream



Token Stream



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

Problem

What happens when token languages overlap / prefix each other?

Language

[A-Za-z][A-Za-z0-9]* { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream

a g e + + EOF

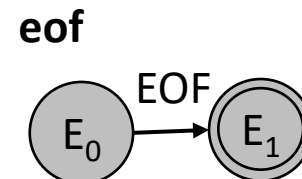
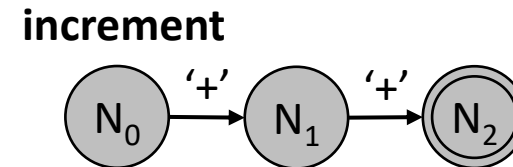
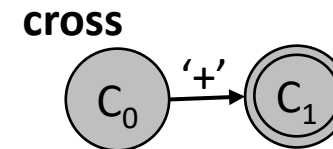
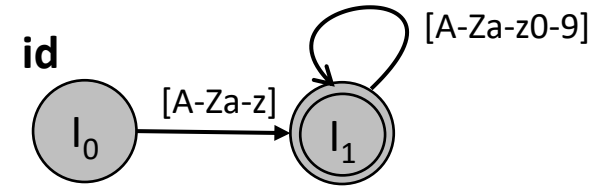
Accept,
but...



shortest
match!

Token Stream

id: a [0,1)	id: g [1,2)	id: e [2,3)	cross [3,4)	cross [4,5)	eof [5,6)
----------------	----------------	----------------	----------------	----------------	--------------



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

Problem

What happens when token languages overlap / prefix each other?

Language

$[A-Za-z][A-Za-z0-9]^*$ { Token(id) }

{ Token(greedy) }

Char Stream

a g

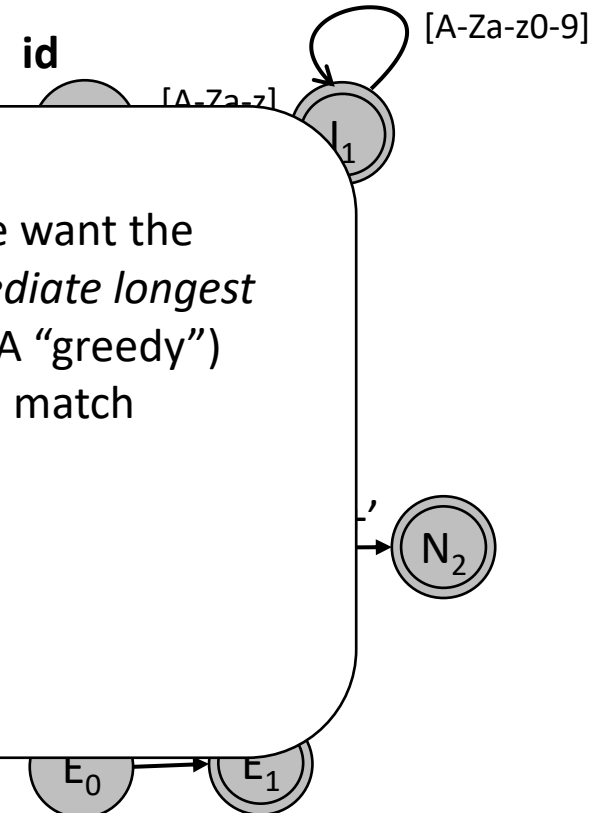
Accept
but...

Token Stream

id: a [0,1)	id: g [1,2)	[2,3)	[3,4)	[4,5)	[5,6)
----------------	----------------	-------	-------	-------	-------



We want the
immediate longest
(AKA "greedy")
match



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

NEW Idea (good)

Consume char stream to **reject** states: return **last accepted** token, restart DFAs with that char

Language

$[A-Za-z][A-Za-z0-9]^*$ { Token(id) }

{ Token(erase) }

Char Stream

a g

Accept
but...

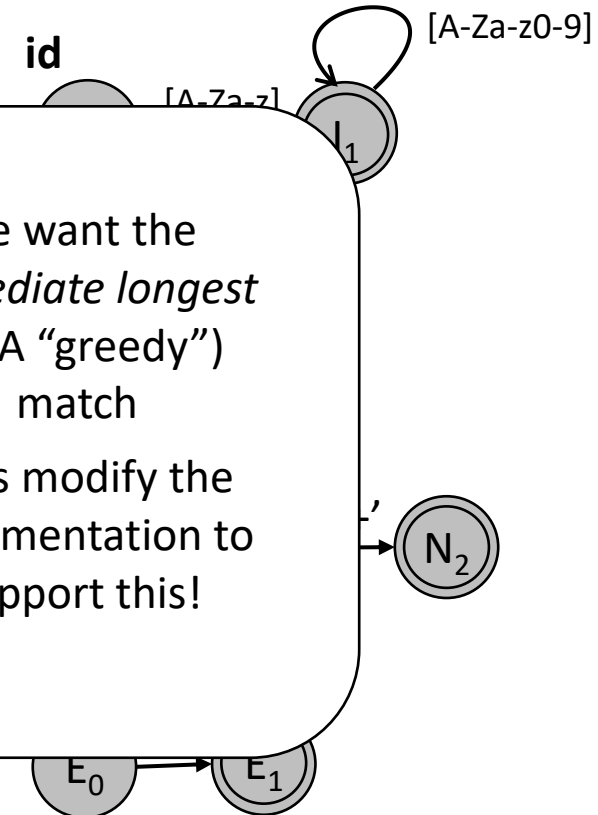
Token Stream

id: a [0,1) id: g [1,2) [2,3) [3,4) [4,5) [5,6)



We want the
immediate longest
(AKA "greedy")
match

Let's modify the
Implementation to
support this!



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

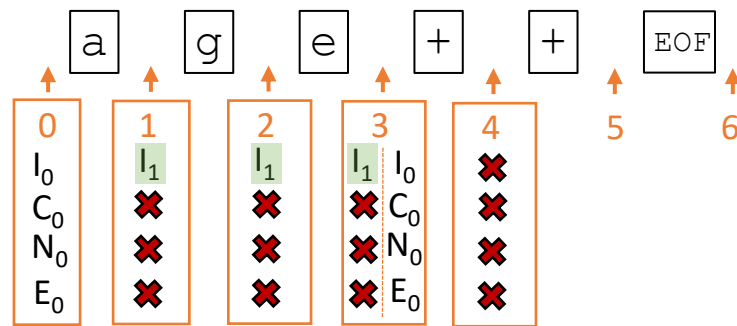
NEW Idea (good)

Consume char stream to **reject** states: return **last accepted** token, restart DFAs with that index

Language

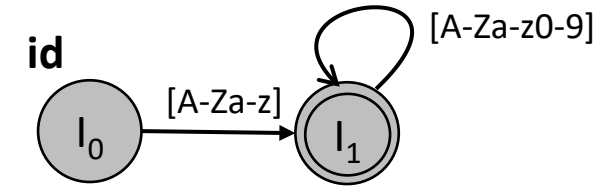
[A-Za-z][A-Za-z0-9]* { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream

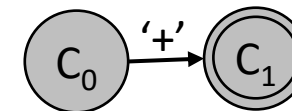


Token Stream

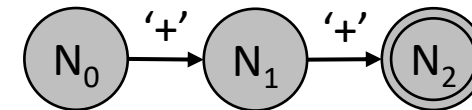
id: age
[0,3)



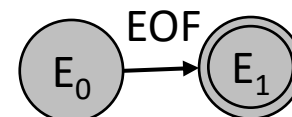
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

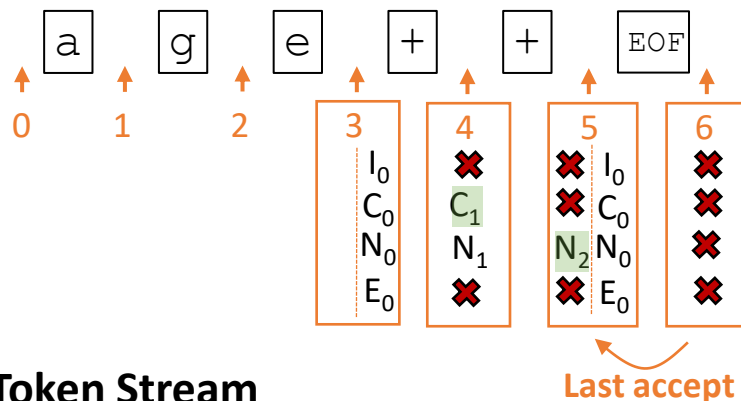
NEW Idea (good)

Consume char stream to **reject** states: return **last accepted** token, restart DFAs with that index

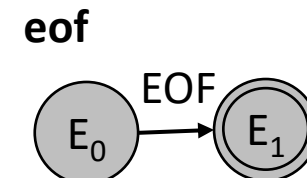
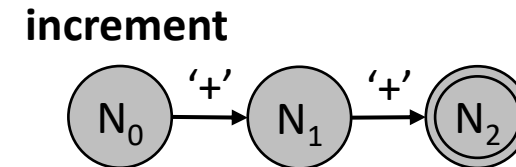
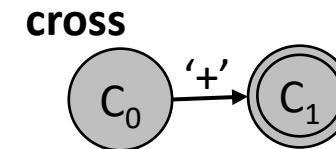
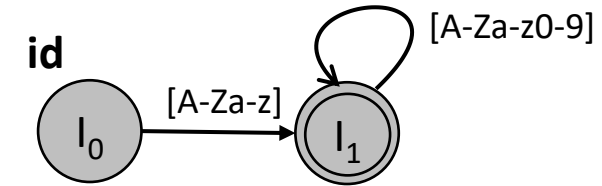
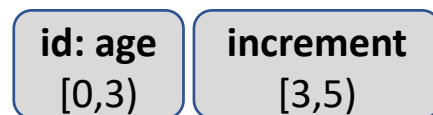
Language

$[A-Za-z][A-Za-z0-9]^*$ { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream



Token Stream



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

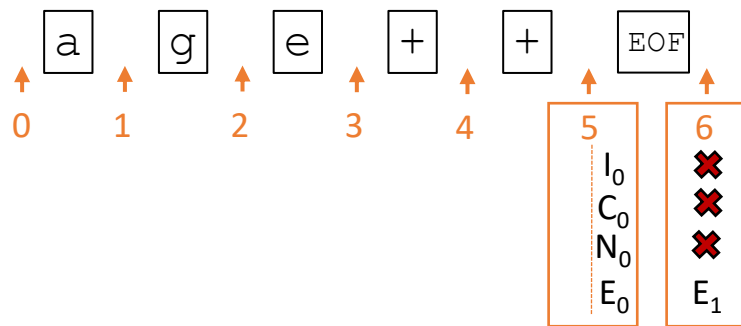
NEW Idea (good)

Consume char stream to **reject** states: return **last accepted** token, restart DFAs with that index

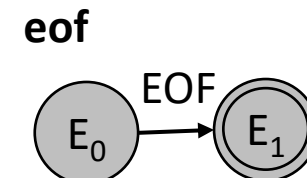
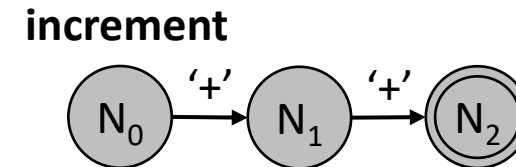
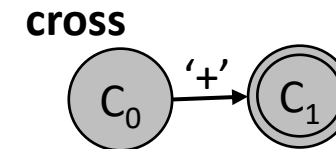
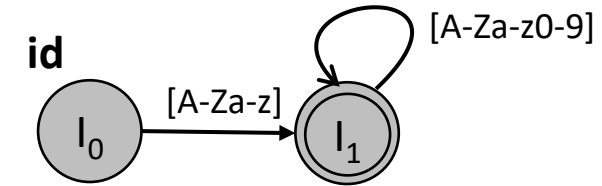
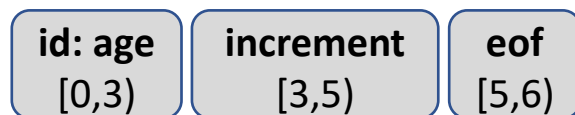
Language

$[A-Za-z][A-Za-z0-9]^*$ { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream



Token Stream



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

Consume char stream to **accept** state: return accepted token, restart DFAs with next char

NEW Idea (good)

Consume char stream to **reject** states: return **last accepted** token, restart DFAs with that index

Language

[A-Za-z][A-Za-z0-9]* { Token(**id**) }
+ { Token(**cross**) }
++ { Token(**increment**) }

Char Stream

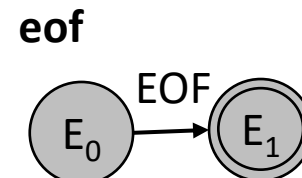
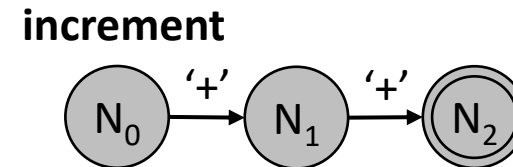
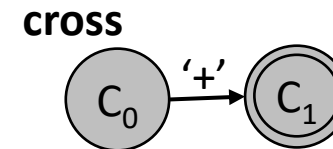
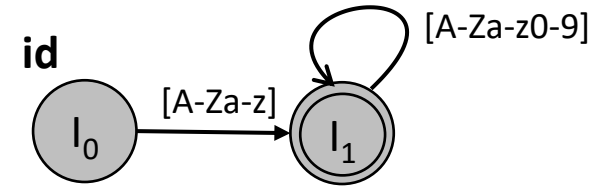
a g e + + EOF



Accept
longest
match!

Token Stream

id: age increment eof
[0,3) [3,5) [5,6)



Tokenizer Action Tables

Implementation

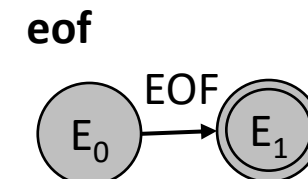
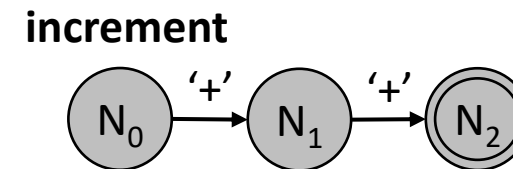
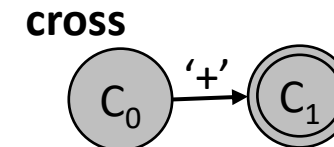
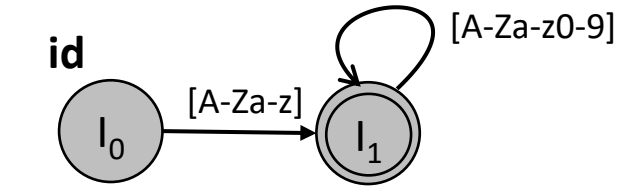
NEW Idea (good)

Consume char stream to **reject** states: return **last accepted** token, restart DFAs with that index

Language

[A-Za-z][A-Za-z0-9]* { Token(**id**) }
 + { Token(**cross**) }
 ++ { Token(**increment**) }

	+	letter	digit	EOF		Token
I ₀		I ₁			I ₀	
I ₁		I ₁	I ₁		I ₁	id
C ₀	C ₁				C ₀	
C ₁					C ₁	cross
N ₀	N ₁				N ₀	
N ₁	N ₂				N ₁	
N ₂					N ₂	increment
E ₀				E ₁	E ₀	
E ₁					E ₁	eof + accept



Tokenizer Action Tables

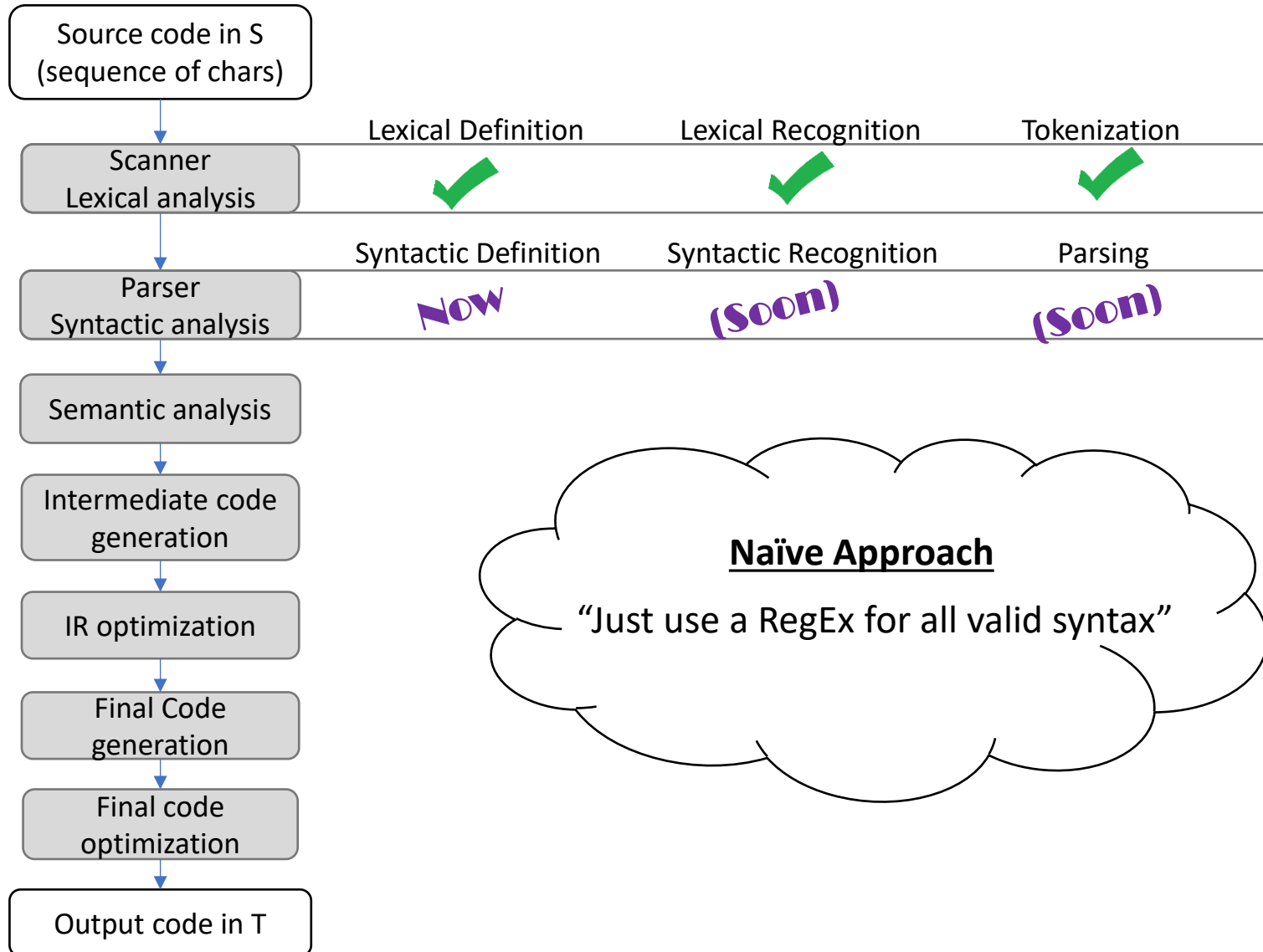
Implementation

	+	letter	digit	EOF		Token
I ₀		I ₁			I ₀	
I ₁		I ₁	I ₁		I ₁	id
C ₀	C ₁				C ₀	
C ₁					C ₁	cross
N ₀	N ₁				N ₀	
N ₁	N ₂				N ₁	
N ₂					N ₂	increment
E ₀				E ₁	E ₀	
E ₁					E ₁	eof + accept

This basic machinery lets us implement a scanner from a RegEx spec!

Lexical Analysis Done

Lecture 3 Preview



Regular Languages Lack Strength

How Languages are Defined: CFGs

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

Char Stream

X \t Y = Z + EOF

Token Stream

ID: X ID: Y ASSIGN ID: Z PLUS EOF



Cute, but weak

Observation: scanner ignores token order

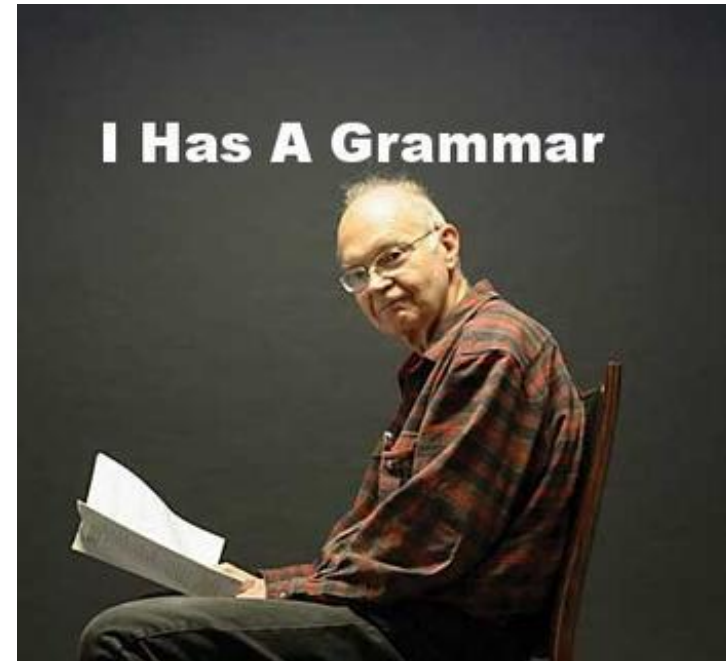
Audience Q: Could you enforce construct order another RegEx?

Answer: Nope! RegEx simply can't capture all PL structures (e.g. parentheses nesting)

Defining Languages with Grammars

Syntactic Definition: How we use CFGs

- A set of (recursive) rewriting rules to rewrite sequence of symbols
- Any “completed” sequence represents a string in the language



Defining Languages with Grammars

Syntactic Definition: How we use CFGs

- A set of (recursive) rewriting rules to rewrite sequence of symbols
- Any “completed” sequence represents a string in the language

CFG = (N, Σ, P, S) where:

- N : set of nonterminal symbols
- Σ : set of terminal symbols
- P : set of productions
- S : start nonterminal in N

Rules where

LHS: a single nonterminal symbol

RHS: a sequence of any symbols

Defining Languages with Grammars

Syntactic Definition: How we use CFGs

Example:

$$N = \{ A \}$$

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

$$S = A$$

$$P = \left\{ \begin{array}{l} P1: A \rightarrow (A) \\ P2: A \rightarrow \epsilon \end{array} \right\}$$

CFG = (N, Σ , P, S) where:

- N: set of nonterminal symbols
- Σ : set of terminal symbols
- P: set of productions
- S: start nonterminal in N

Defining Languages with Grammars

Syntactic Definition: How we use CFGs

Example:

$$N = \{ A \}$$

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

$$S = A$$

$$P = \left\{ \begin{array}{l} \text{P1: } A \rightarrow (A) \\ \text{P2: } A \rightarrow \epsilon \end{array} \right\}$$

Producing a string

A

Begin with start symbol

$A \rightarrow (A)$

Apply production P1
(a *derivation step*, denoted $\Rightarrow$)

(A)

Get a new symbol string

$A \rightarrow (A)$

Apply production P1 again

$((A))$

Get a new symbol string

$A \rightarrow \epsilon$

Apply another production in P

$(())$

Get a new symbol string

All terminals, this string is in language
(a *sentence*)

Simplifying Notation: Shorthand

Syntactic Definition: How we use CFGs

Example:

$$N = \{ A \}$$

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

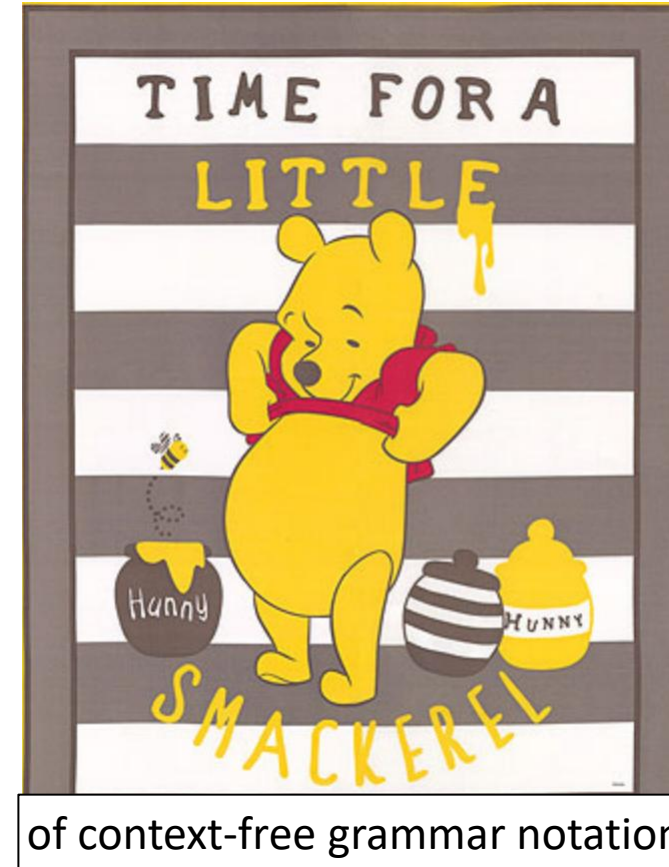
$$S = A$$

← Say S Implicit: LHS of top production

$$P = \left\{ \begin{array}{l} \text{P1: } A \rightarrow (A) \\ \text{P2: } A \rightarrow \varepsilon \end{array} \right\}$$

Say N and Σ Implicit:
Whatever symbols
appears in productions

Collapse rules with
the same LHS
using bar



Simplifying Notation: Shorthand

Syntactic Definition: How we use CFGs

Example:

$N = \{ A \}$

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

$S = A$

$P = \left\{ \begin{array}{l} \text{P1: } A \rightarrow (A) \\ \text{P2: } A \rightarrow \epsilon \end{array} \right\}$

Say N and Σ Implicit:
Whatever symbols
appears in productions

← Say S Implicit: LHS of top production

Collapse rules with
the same LHS
using bar

Denote grammar as

$A ::= (A)$

$A ::= \epsilon$

or equivalently as

$A ::= (A)$
 $\quad \mid \epsilon$

or even

$A ::= (A) \mid \epsilon$

Simplifying Notation: Shorthand

Syntactic Definition: How we use CFGs

$$A \rightarrow (A)$$

$$A ::= (\text{“BNF”})$$

Denote grammar as

$$A ::= (A)$$

$$A ::= \epsilon$$

or equivalently as

$$A ::= (A)$$

$$| \epsilon$$

or even

$$A ::= (A) | \epsilon$$

Some languages denoted in BNF

Syntactic Definition: How we use CFGs

$$A ::= (A) \\ | \epsilon$$

$F ::= \mathbf{b} \ G \ \mathbf{y} \ \mathbf{e}$	$F \Rightarrow \mathbf{b} \ G \ \mathbf{y} \ \mathbf{e} \Rightarrow \mathbf{b} \ \mathbf{y} \ \mathbf{e}$
$ \ \mathbf{see} \ \mathbf{ya}$	$F \Rightarrow \mathbf{b} \ G \ \mathbf{y} \ \mathbf{e} \Rightarrow \mathbf{b} \ G \ \mathbf{y} \ \mathbf{y} \ \mathbf{e} \Rightarrow \mathbf{b} \ \mathbf{y} \ \mathbf{y} \ \mathbf{e}$
$G ::= G \ \mathbf{y}$	$F \Rightarrow \mathbf{see} \ \mathbf{ya}$
$ \ \epsilon$	

$Y ::= \mathbf{a} \ Y$	$Y \Rightarrow \mathbf{a} \ Y \Rightarrow \mathbf{a} \ \mathbf{a} \ Y \Rightarrow \mathbf{a} \ \mathbf{a} \ \mathbf{a} \ Y \Rightarrow \dots$
$Z ::= \mathbf{w} \ \mathbf{t} \ \mathbf{f}$	Accepts no strings (not even the empty string)

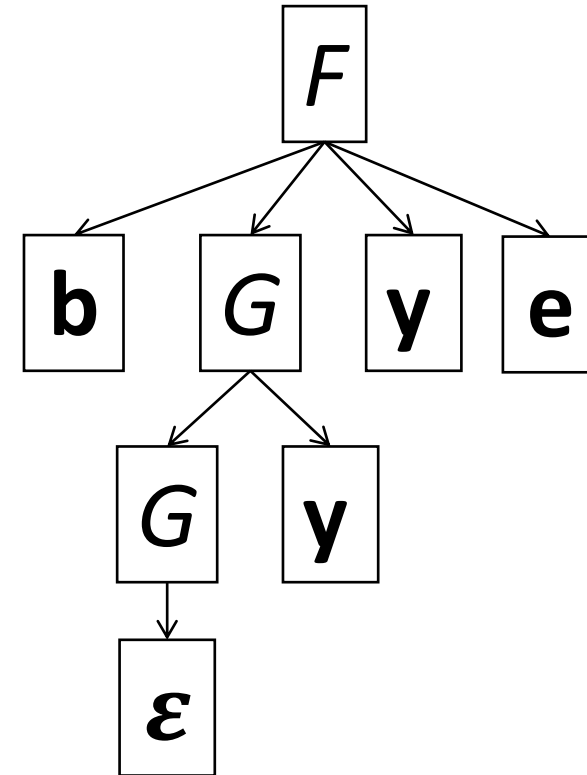
Parse Trees

Syntactic Definition: How we use CFGs

Represent Derivations

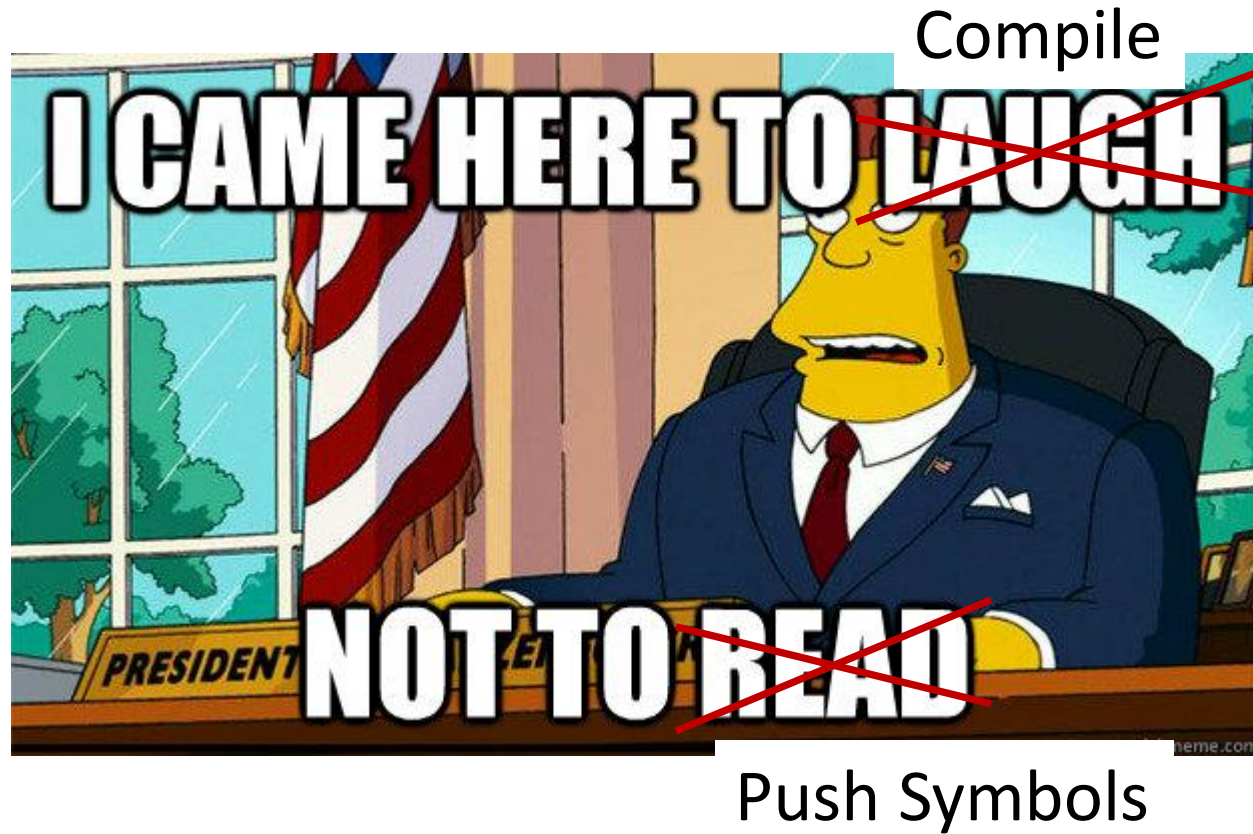
- Nodes are symbols in a tree
- Rooted at start symbol
- Children are derivation step
- Leaves are final string (if all terminals)

F $\Rightarrow$ b G y $e \Rightarrow b$ G y y $e \Rightarrow b$ y y e



CFG use in the Compiler

Syntactic Definition: How we use CFGs



CFG use in the Compiler

Syntactic Definition: How we use CFGs

CFG for PL Syntactic Structure

Productions specify valid programs

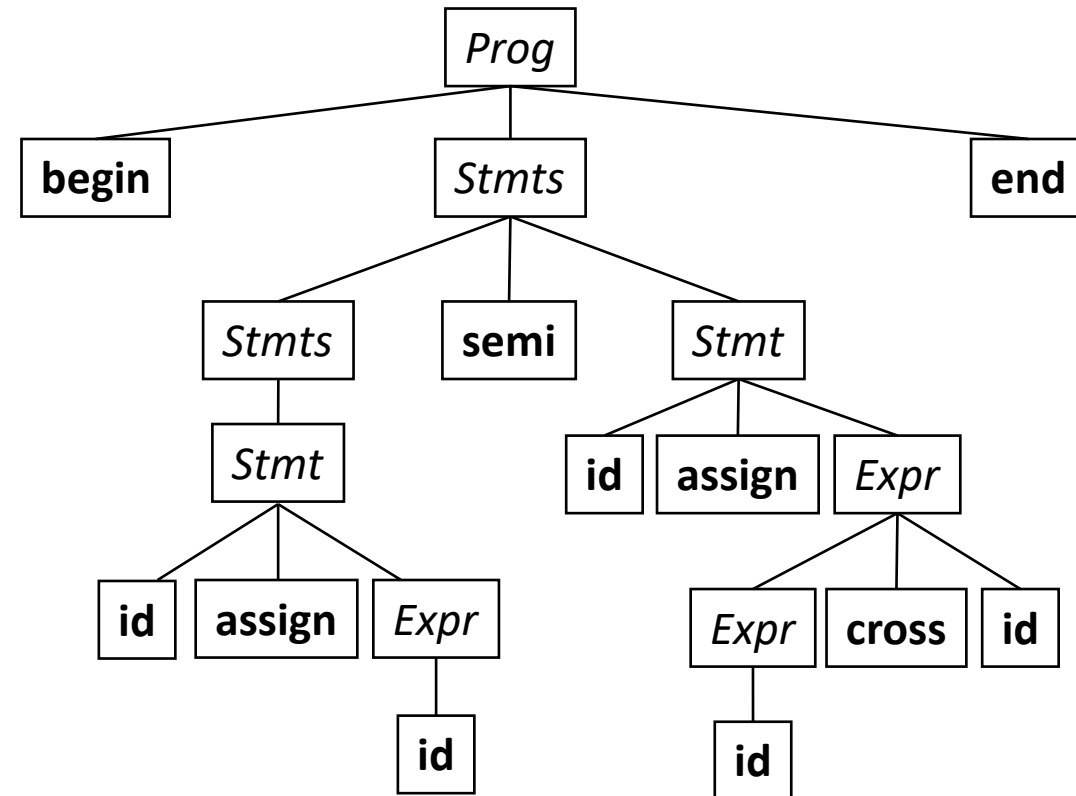
- Let the terminals be the tokens in the language
- Let the nonterminals be the groupings that form language constructs
 - (loops, statements, functions, calls, etc)
- The grammar will recognize (or reject) the stream of tokens from the Lexer

Let's see an example
with this grammar

Productions

1. *Prog* ::= **begin** *Stmts* **end**
2. *Stmts* ::= *Stmts* **semi** *Stmt*
3. | *Stmt*
4. *Stmt* ::= **id** **assign** *Expr*
5. *Expr* ::= **id**
6. | *Expr* **cross** **id**

Parse Tree



Derivation Sequence

Prog
⇒ **begin** Stmts **end** Prod. 1
⇒ **begin** Stmts **semi** Stmt **end** Prod. 2
⇒ **begin** Stmts **semi** Stmt **end** Prod. 3
⇒ **begin** Stmt **semi** Stmt **end** Prod. 4
⇒ **begin** id assign Expr **semi** Stmt **end** Prod. 4
⇒ **begin** id assign Expr **semi** id assign Expr **end** Prod. 5
⇒ **begin** id assign id **semi** id assign Expr **end** Prod. 6
⇒ **begin** id assign id **semi** id assign Expr cross id **end** Prod. 5
⇒ **begin** id assign id **semi** id assign id cross id **end**

Productions

1. *Prog* ::= **begin** *Stmts* **end**
2. *Stmts* ::= *Stmts* **semi** *Stmt*
3. | *Stmt*
4. *Stmt* ::= **id** **assign** *Expr*
5. *Expr* ::= **id**
6. | *Expr* **cross** **id**

End of Lecture

Syntactic Definition

Next Time

Parsing - Beyond specification for CFGs

- Extracting the *correct* tree from a token stream

Time Permitting

- Proof sketch: why RegExs can't match PL constructs

