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

Checkin Checkout: C2

Describe a lexical error in C

emojis

```
1 #include "stdio.h"
2 int main(){
3         int 😊 = 2;
4         int 💩 = 3;
5         printf("%d\n", 😊 + 💩);
6 }
```

A misspelled type name (imt instead of int)

Describe a semantic error in C

;;;

Administrivia

Housekeeping

P1 Out

KU | EECS| Drew Davidson

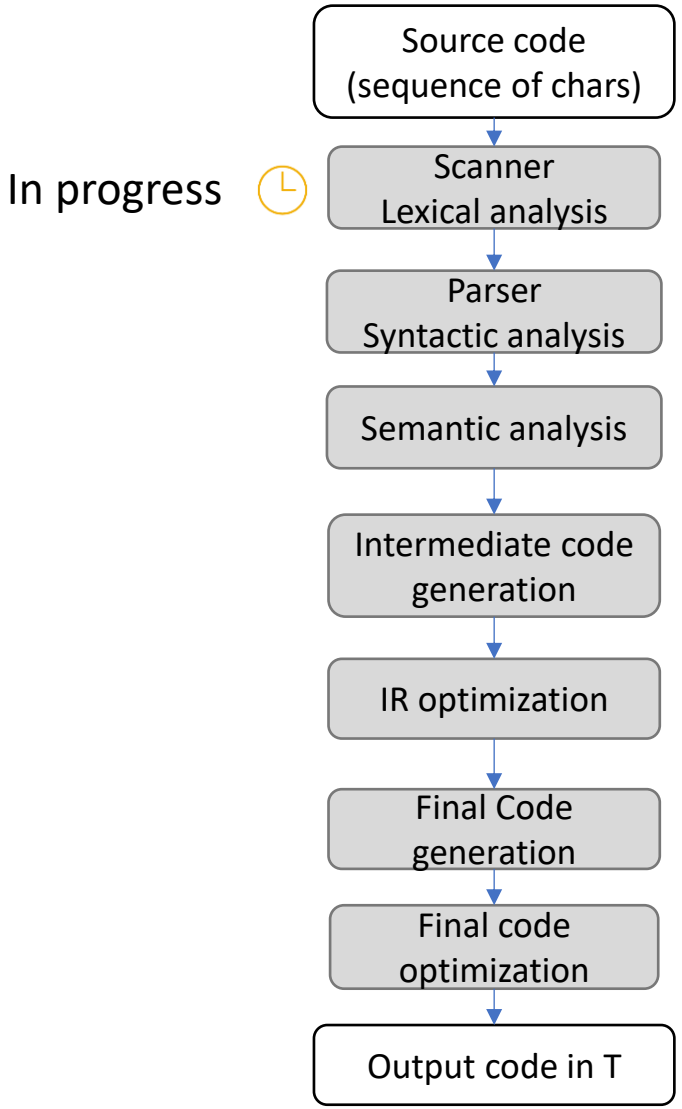
Lecture 3
Syntactic Definition

Compiler Construction

Progress Pics

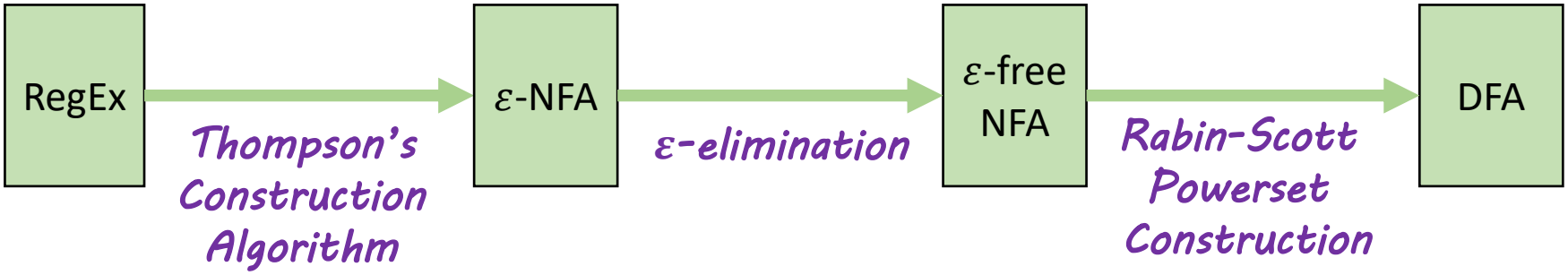
Done:

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



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

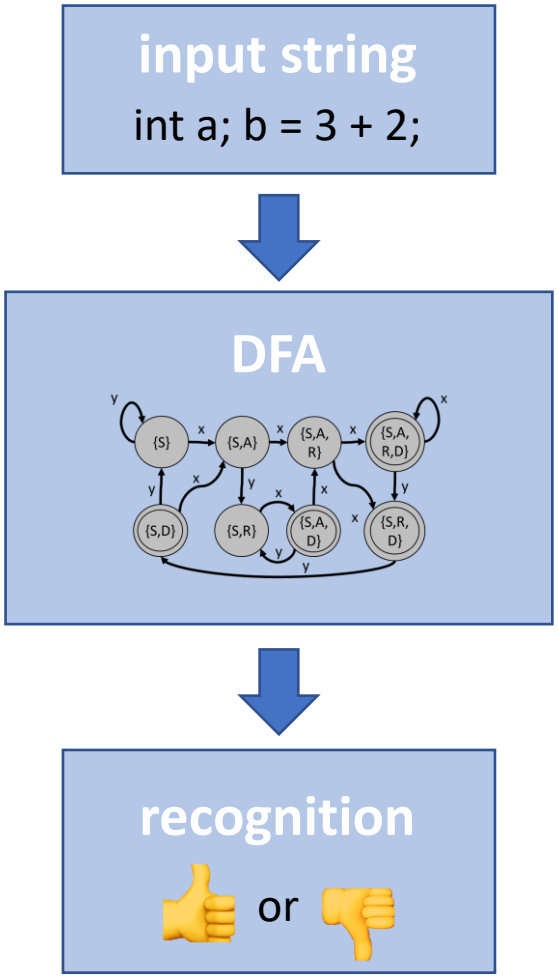
Begin discussing syntax

- How to specify the syntax of a programming language

DFA ≠ 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

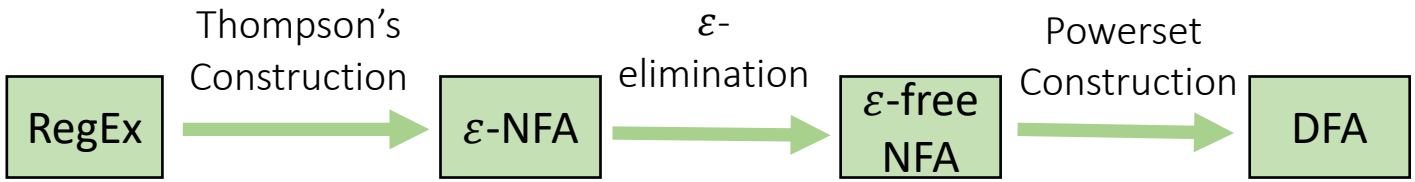
- Algorithms
- Implementation details

How Language Syntax is Formally Defined: CFGs

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

From RegExes to Tokenizer

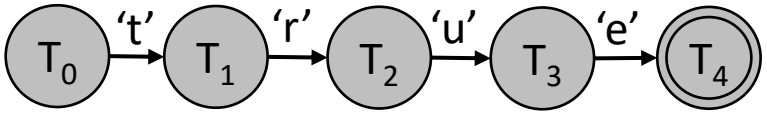
Algorithms



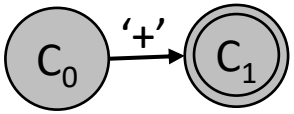
Language

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

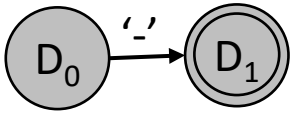
true



cross

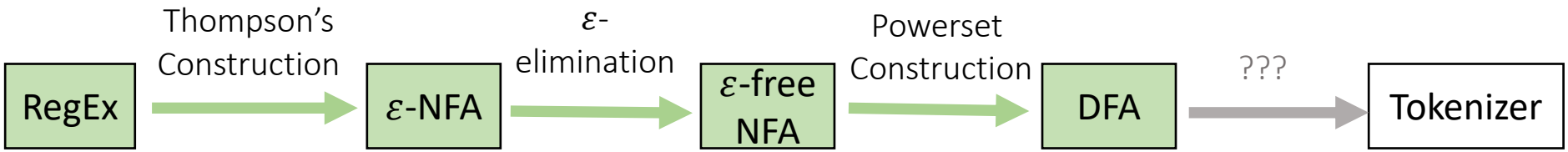


dash



From RegExes to Tokenizer

Algorithms



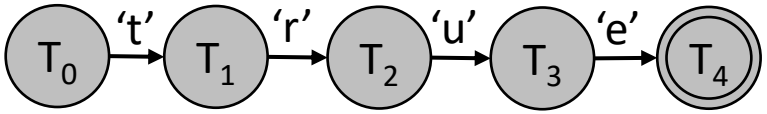
1st Idea (flawed)

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

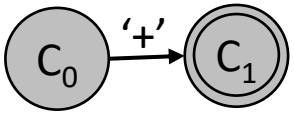
Language

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

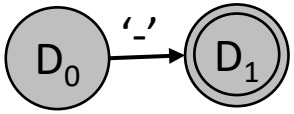
true



cross



dash



From RegExes to Tokenizer

Algorithms

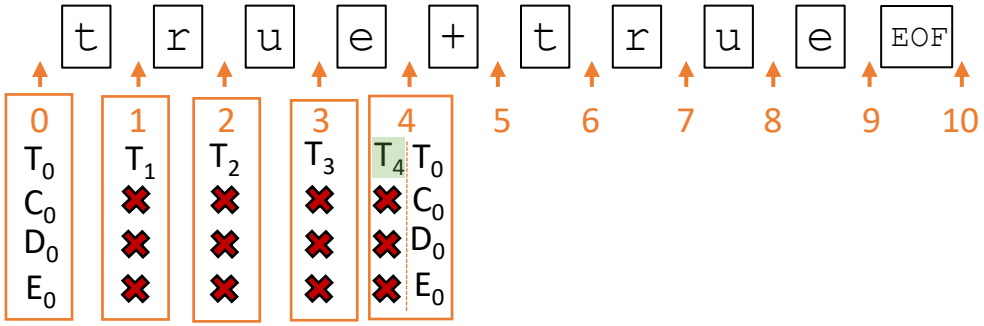
1st Idea (flawed)

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

Language

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

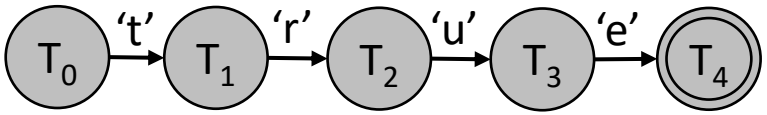
Char Stream



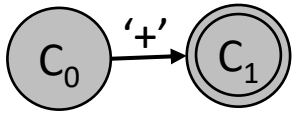
Token Stream

true
[0,4)

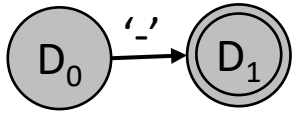
true



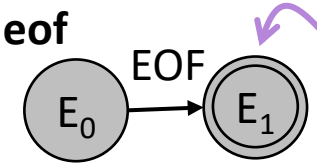
cross



dash



eof



Special State:
Return EOF
Accept stream

From RegExes to Tokenizer

Algorithms

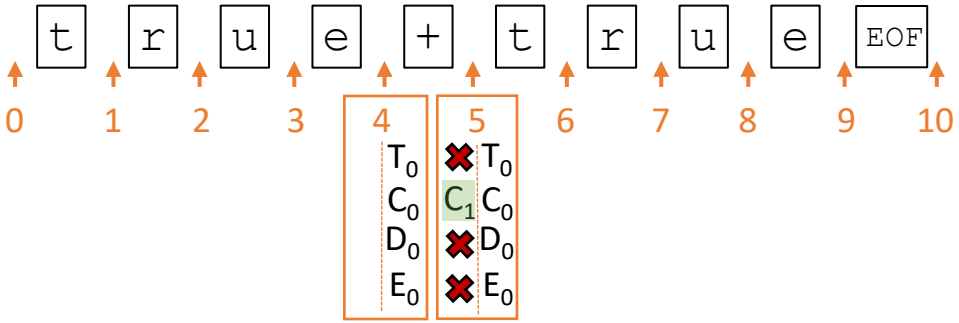
1st Idea (flawed)

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

Language

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

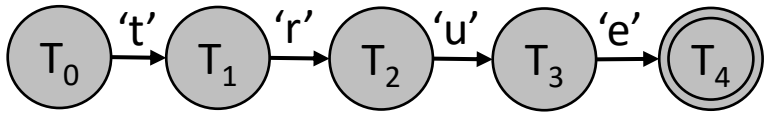
Char Stream



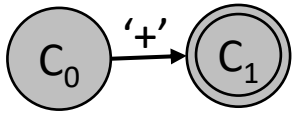
Token Stream

true	**cross**
[0,4)	[4,5)

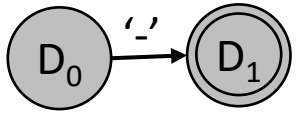
true



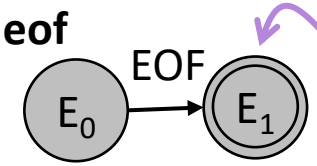
cross



dash



eof



Special State:
Return EOF
Accept stream

From RegExes to Tokenizer

Algorithms

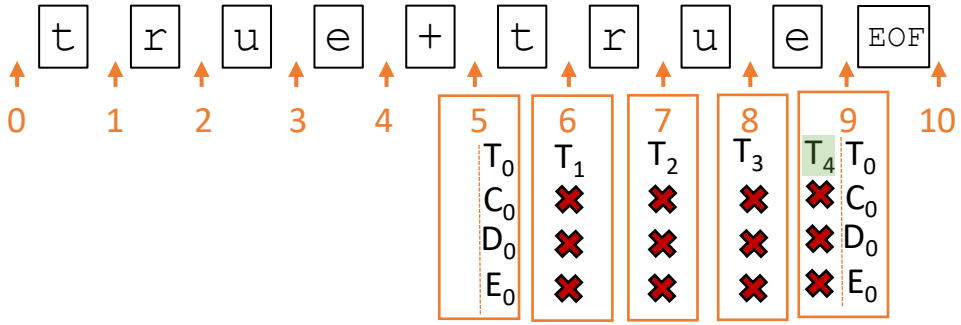
1st Idea (flawed)

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

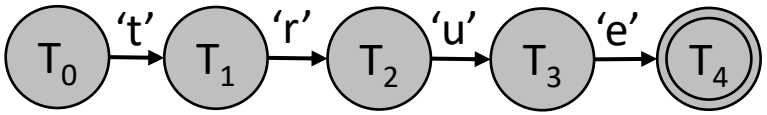
Language

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

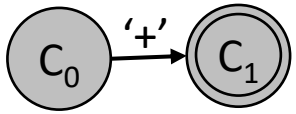
Char Stream



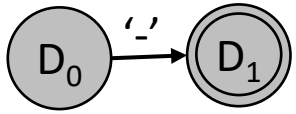
true



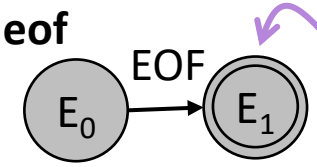
cross



dash



eof



Special State:
Return EOF
Accept stream

Token Stream

true [0,4)	**cross** [4,5)	**true** [5,9)

From RegExes to Tokenizer

Algorithms

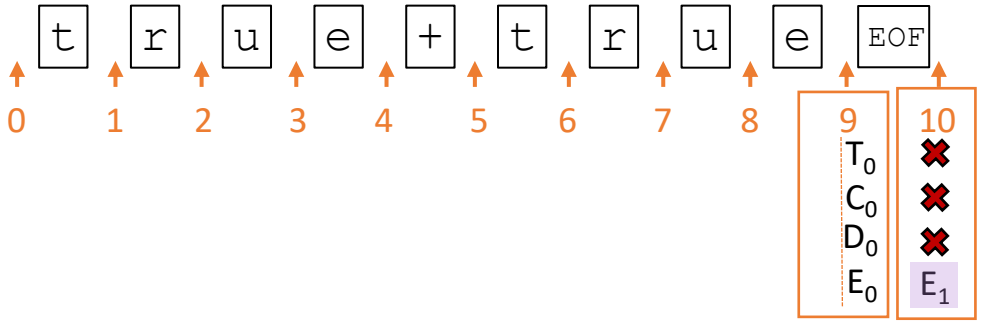
1st Idea (flawed)

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

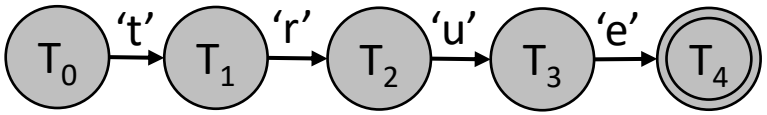
Language

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

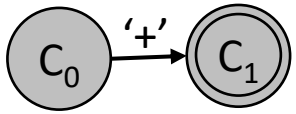
Char Stream



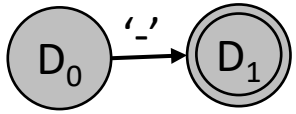
true



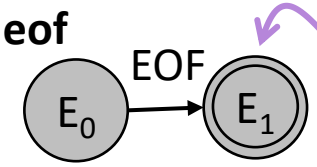
cross



dash



eof



Special State:
Return EOF
Accept stream

Token Stream

true [0,4)	**cross** [4,5)	**true** [5,9)	**eof** [9,10)

From RegExes to Tokenizer

Algorithms

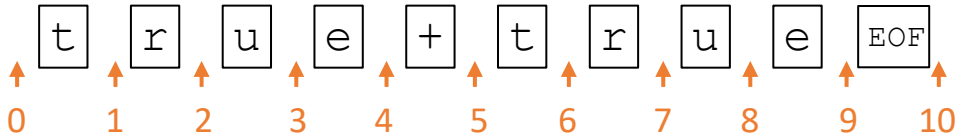
1st Idea (flawed)

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

Language

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

Char Stream

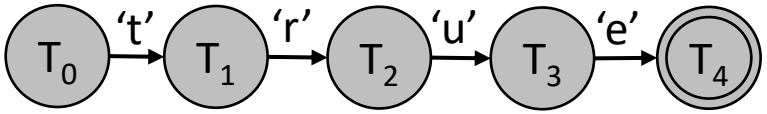


Accept!

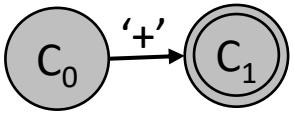
Token Stream

true [0,4)	**cross** [4,5)	**true** [5,9)	**eof** [9,10)

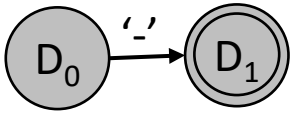
true



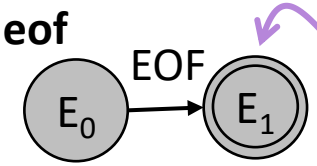
cross



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

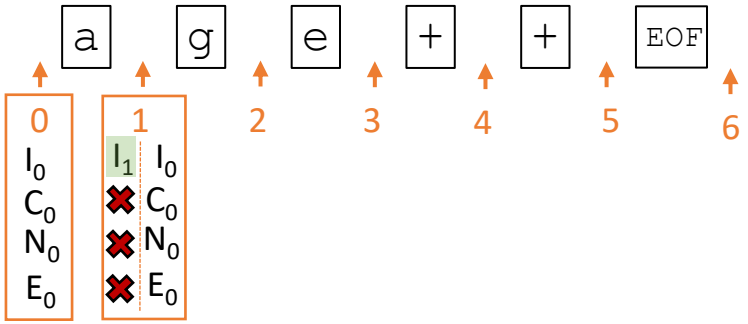
Problem

What happens when token languages overlap / prefix each other?

Language

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

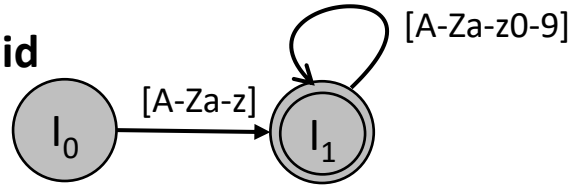
Char Stream



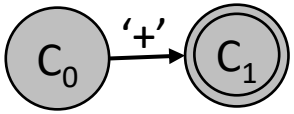
Token Stream

id: a
[0,1)

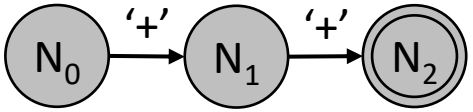
id



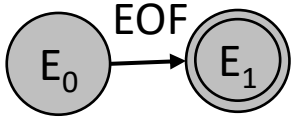
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

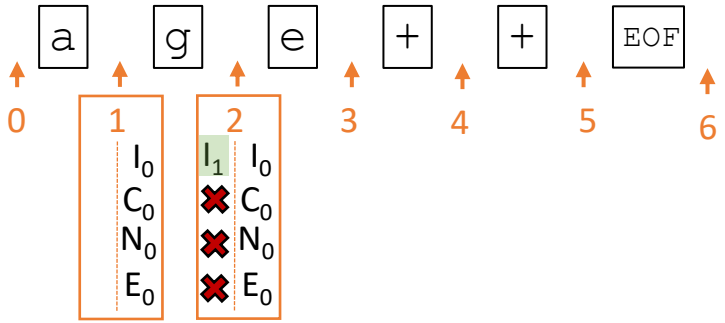
Problem

What happens when token languages overlap / prefix each other?

Language

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

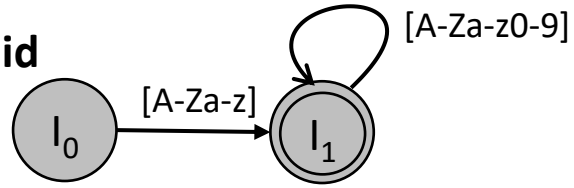
Char Stream



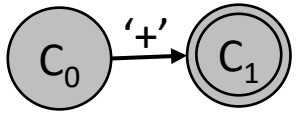
Token Stream

id: a [0,1)	**id: g** [1,2)

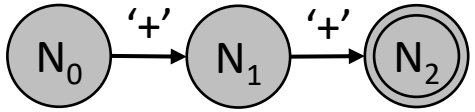
id



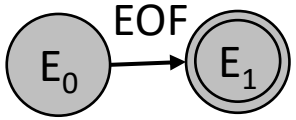
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

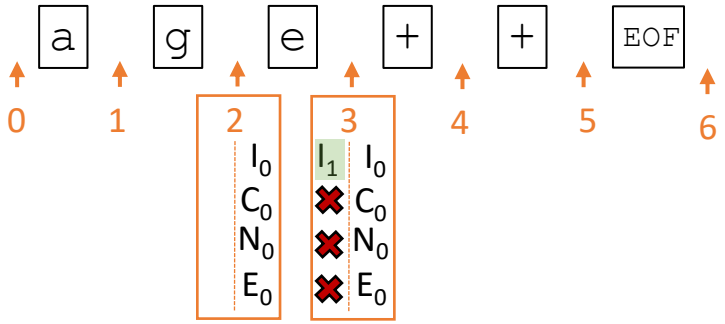
Problem

What happens when token languages overlap / prefix each other?

Language

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

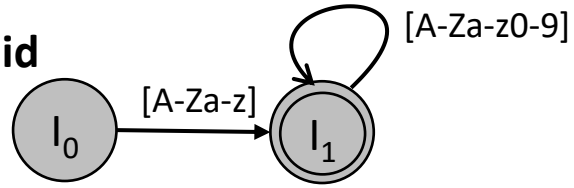
Char Stream



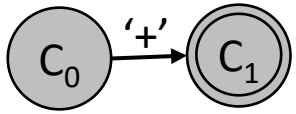
Token Stream

id: a [0,1)	**id: g** [1,2)	**id: e** [2,3)

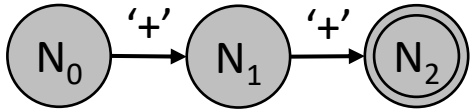
id



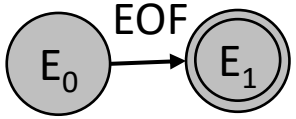
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

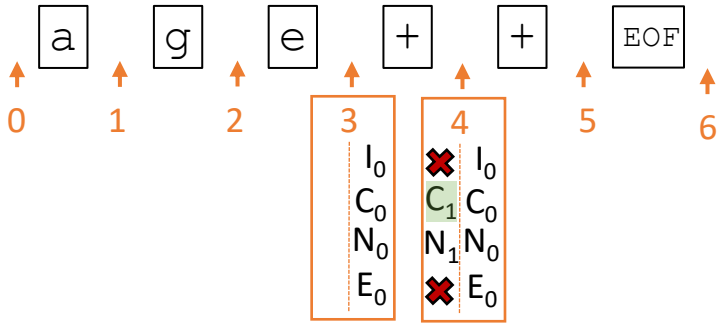
Problem

What happens when token languages overlap / prefix each other?

Language

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

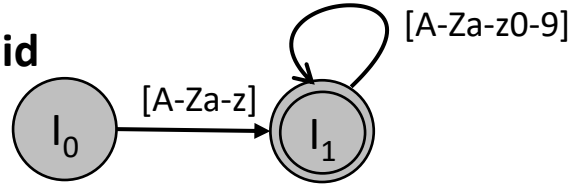
Char Stream



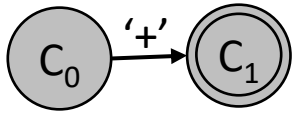
Token Stream

id: a	id: g	id: e	cross
[0,1)	[1,2)	[2,3)	[3,4)

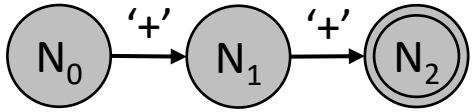
id



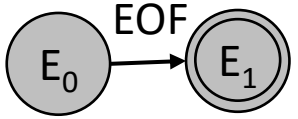
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

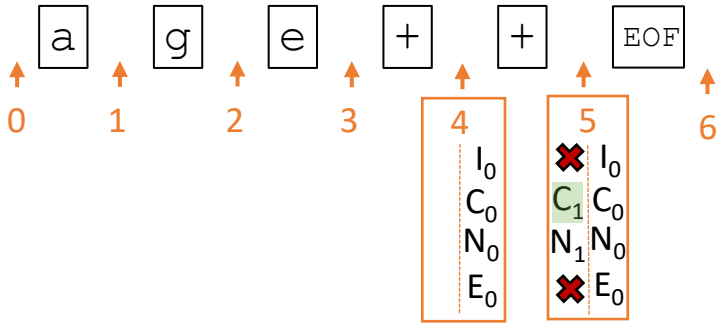
Problem

What happens when token languages overlap / prefix each other?

Language

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

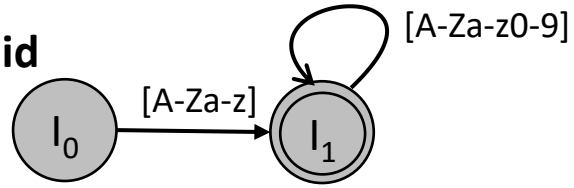
Char Stream



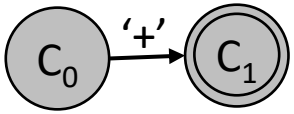
Token Stream

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

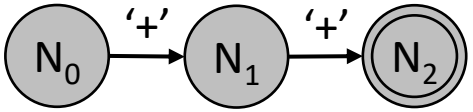
id



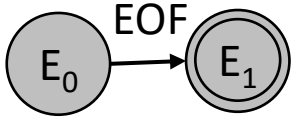
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

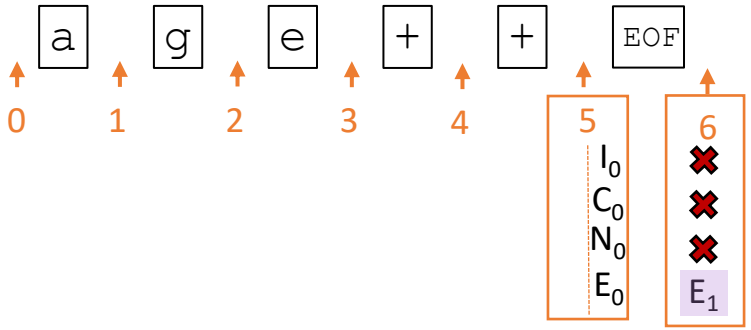
Problem

What happens when token languages overlap / prefix each other?

Language

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

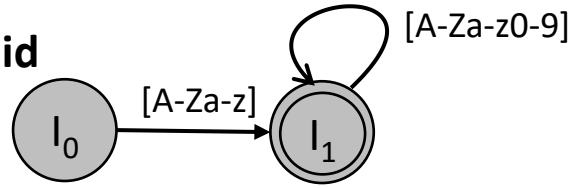
Char Stream



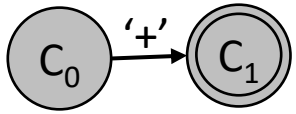
Token Stream

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

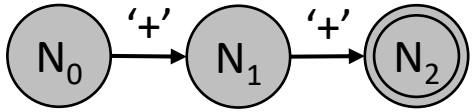
id



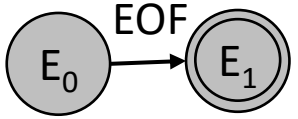
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

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

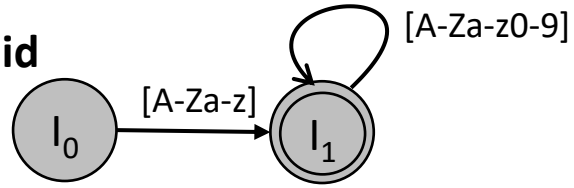
Accept, but…

shortest match!

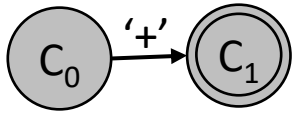
Token Stream

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

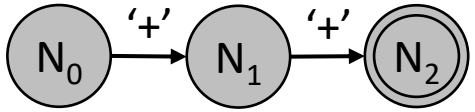
id



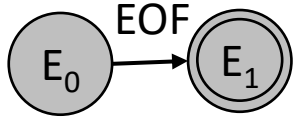
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

Problem

What happens when token languages overlap / prefix each other?

Language

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

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

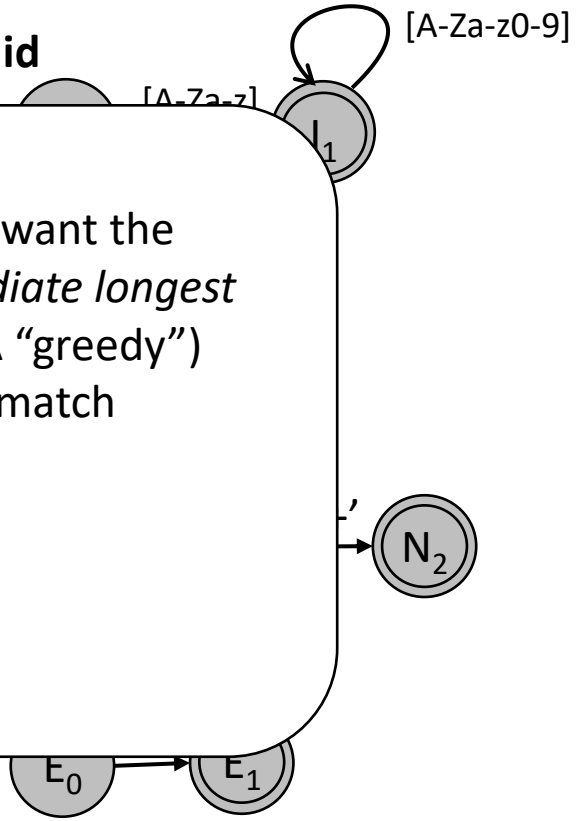
Char Stream

a g

Accept
but…

Token Stream

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



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**) }

Char Stream

a	g

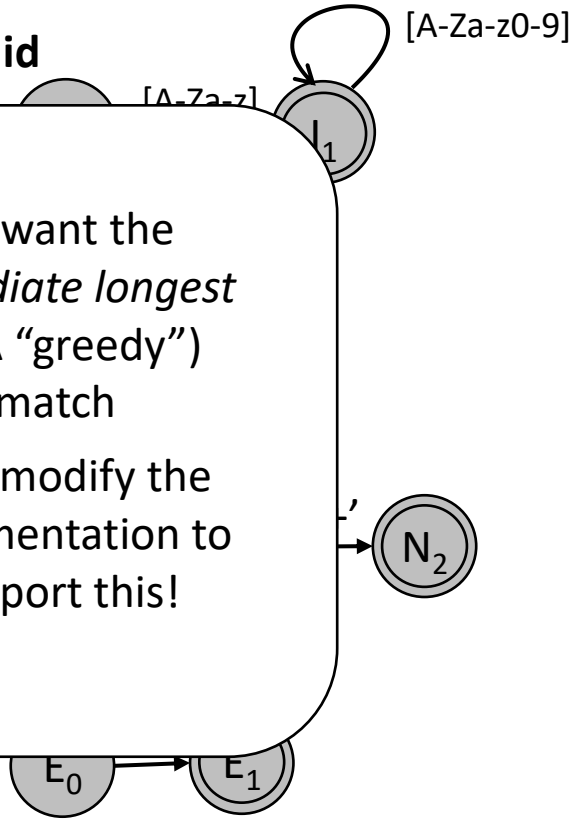
Accept
but…

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

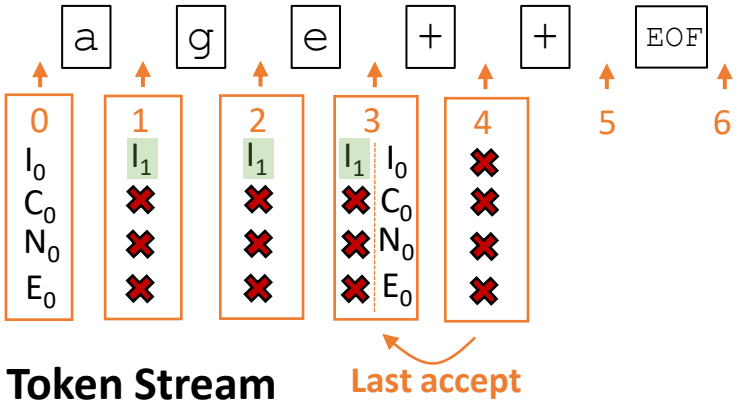
NEW Idea (good)

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

Language

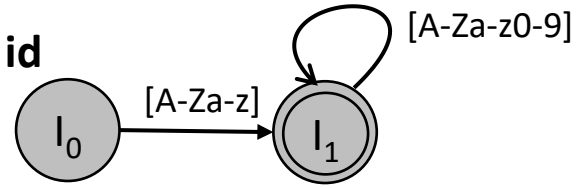
[A-Za-z][A-Za-z0-9]*	{ Token(**id**) }
+	{ Token(**cross**) }
++	{ Token(**increment**) }

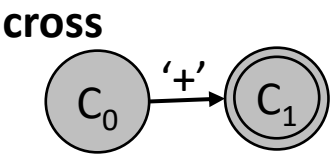
Char Stream

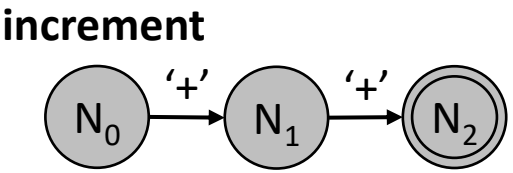


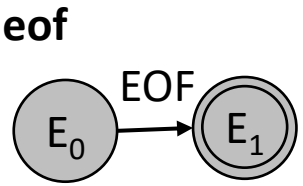
Token Stream

id: age
[0,3)









From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

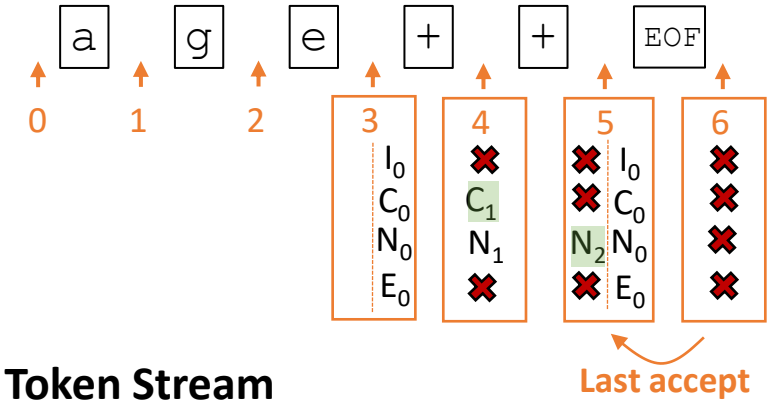
NEW Idea (good)

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**) }

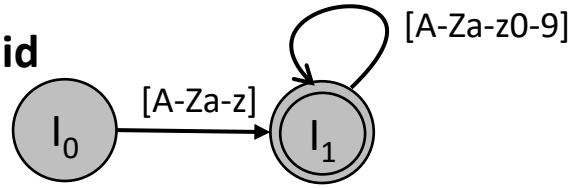
Char Stream



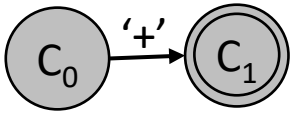
Token Stream

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

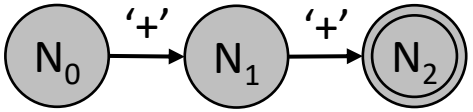
id



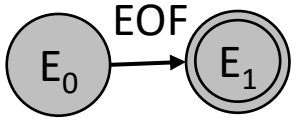
cross



increment



eof



From RegExes to Tokenizer

Algorithms

1st Idea (flawed)

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

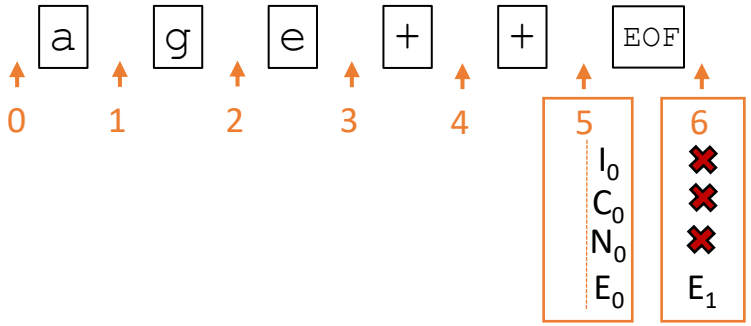
NEW Idea (good)

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**) }

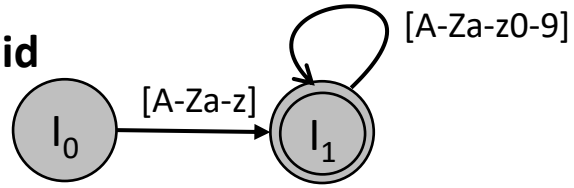
Char Stream



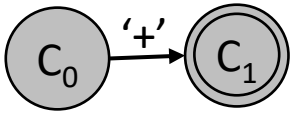
Token Stream

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

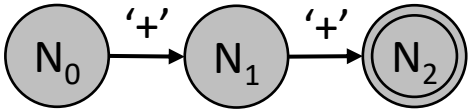
id



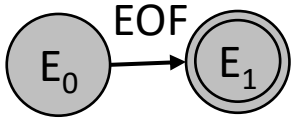
cross



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

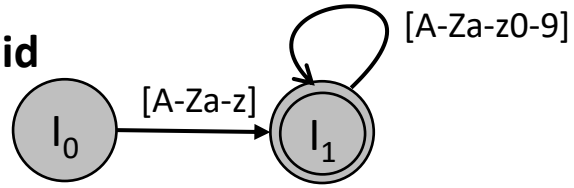
a	g	e	+	+	EOF

Accept longest match!

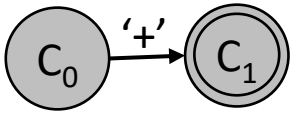
Token Stream

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

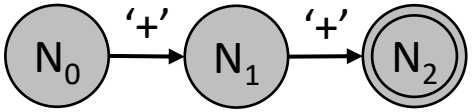
id



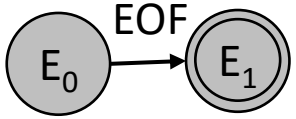
cross



increment



eof



Tokenizer Action Tables

Implementation

NEW Idea (good)

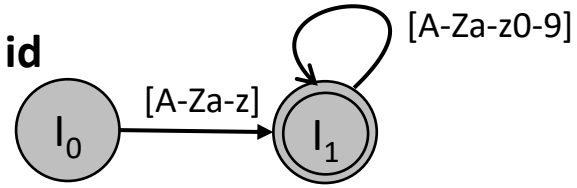
Consume char stream to **reject** states: return **last accepted** token, restart DFAs with that index

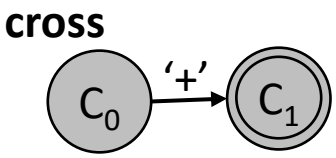
Language

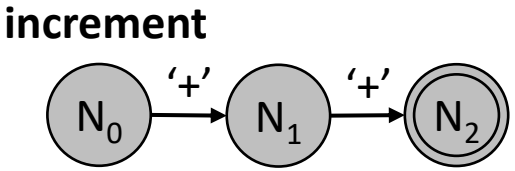
[A-Za-z][A-Za-z0-9]*	{ Token(**id**) }
+	{ Token(**cross**) }
++	{ Token(**increment**) }

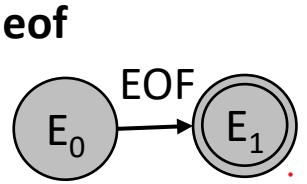
	+	letter	digit	EOF
I_0		I_1		
I_1		I_1	I_1	
C_0	C_1			
C_1				
N_0	N_1			
N_1	N_2			
N_2				
E_0				E_1
E_1				

	Token
I_0	
I_1	**id**
C_0	
C_1	**cross**
N_0	
N_1	
N_2	**increment**
E_0	
E_1	**eof** + accept









Tokenizer Action Tables

Implementation

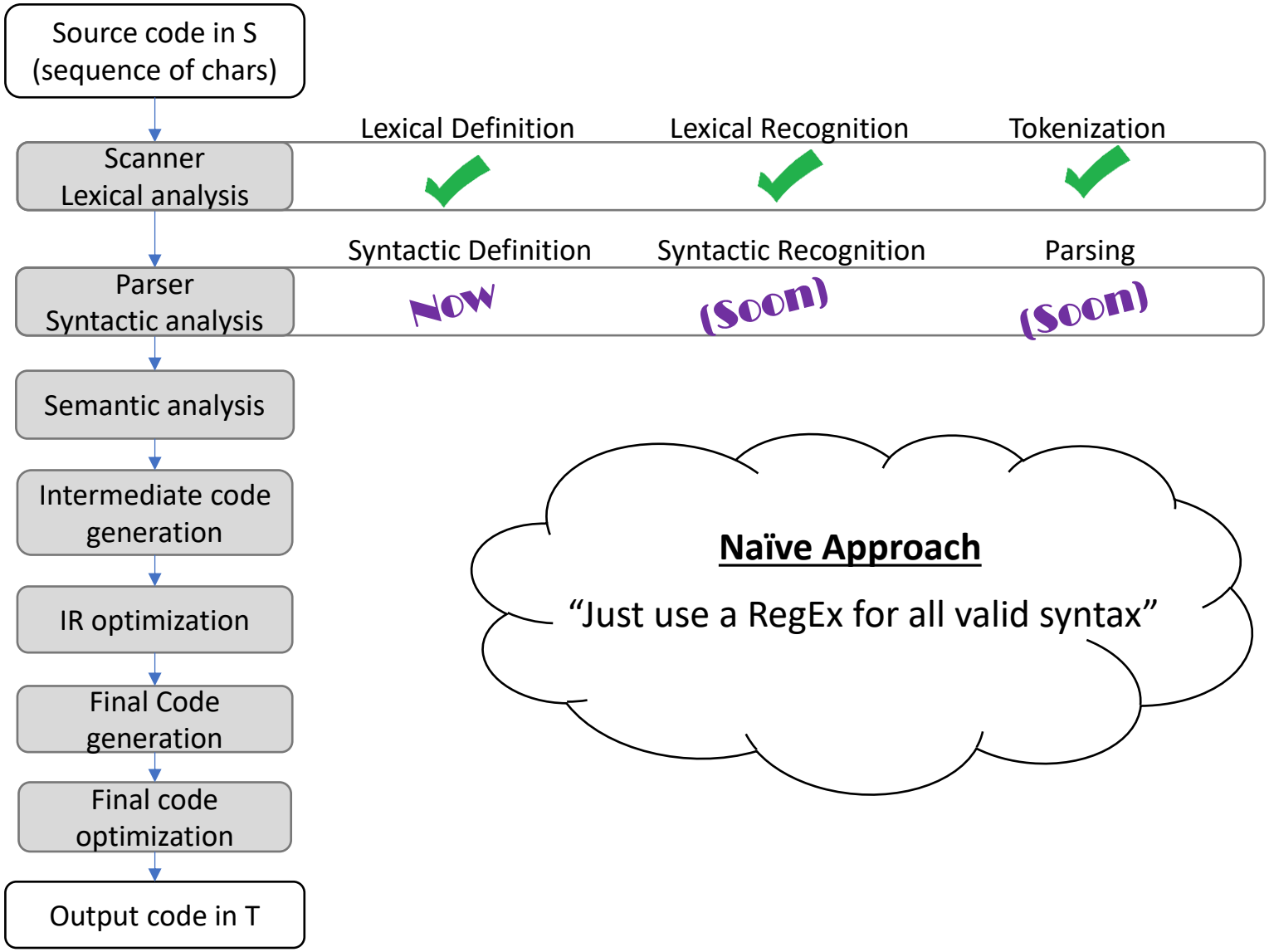
	+	letter	digit	EOF
I_0		I_1		
I_1		I_1	I_1	
C_0	C_1			
C_1				
N_0	N_1			
N_1	N_2			
N_2				
E_0				E_1
E_1				

	Token
I_0	
I_1	**id**
C_0	
C_1	**cross**
N_0	
N_1	
N_2	**increment**
E_0	
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

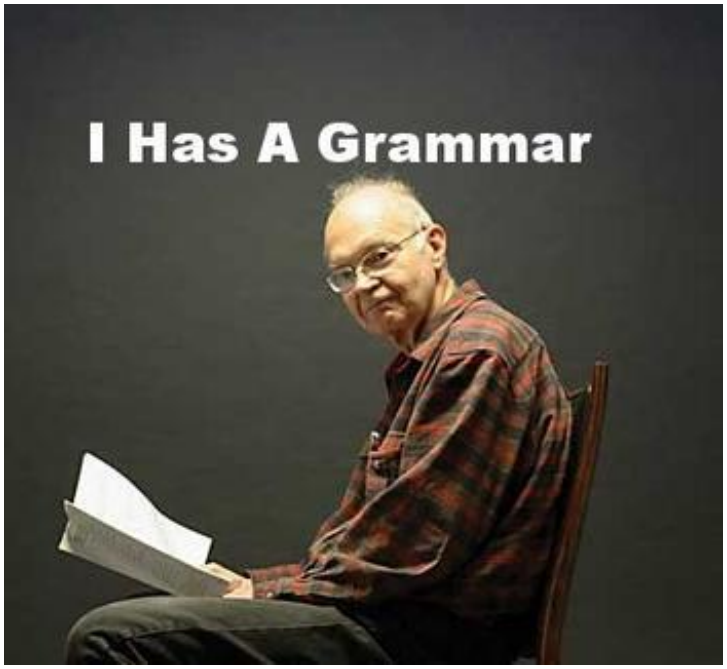
Audience Q: Could you enforce construct order another RegEx?

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:

$\mathrm{N} = \{ A \}$

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

$\mathrm{S} = A$

$$\mathrm{P} = \left\{ \begin{array}{ll} \mathrm{P1}: & A \rightarrow (A) \\ \mathrm{P2}: & A \rightarrow \boldsymbol{\varepsilon} \end{array} \right\}$$

CFG = $(\mathrm{N}, \Sigma, \mathrm{P}, \mathrm{S})$ 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:

$N = \{ A \}$

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

$S = A$

$$P = \begin{Bmatrix} \text{P1: } A \rightarrow (A) \\ \text{P2: } A \rightarrow \varepsilon \end{Bmatrix}$$

Producing a string

A	Begin with start symbol
$A \rightarrow (A)$	Apply production P1 (a *derivation step, denoted* ⇒)
(A)	Get a new symbol string
$A \rightarrow (A)$	Apply production P1 again
$((A))$	Get a new symbol string
$A \rightarrow \varepsilon$	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:

$N = \{ A \}$

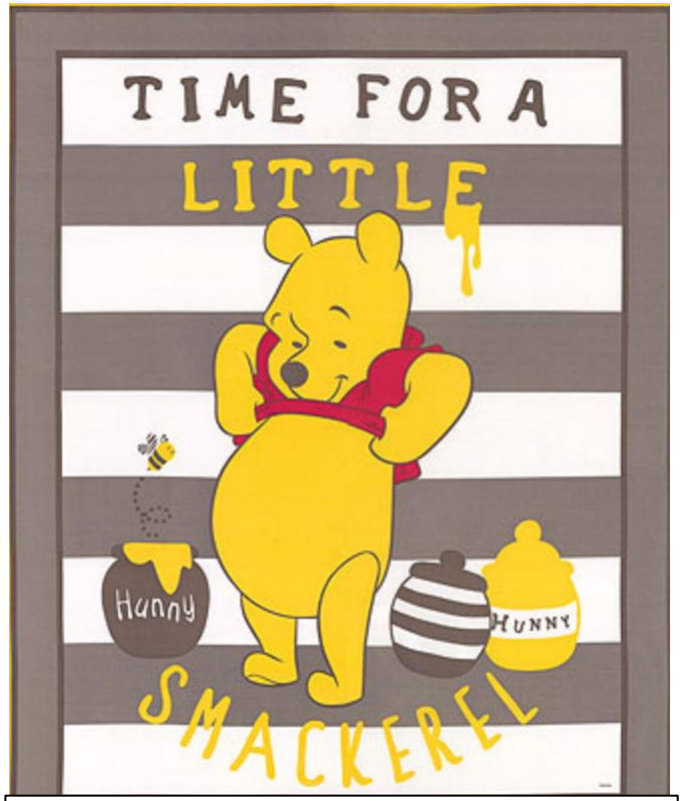
$\Sigma = \{ (,) \}$

Say N and Σ Implicit: Whatever symbols appears in productions

$S = A$ ← Say S Implicit: LHS of top production

$$P = \begin{Bmatrix} \text{P1: } A \rightarrow (A) \\ \text{P2: } A \rightarrow \varepsilon \end{Bmatrix}$$

Collapse rules with the same LHS using bar



of context-free grammar notation

Simplifying Notation: Shorthand

Syntactic Definition: How we use CFGs

Example:

$N = \{ A \}$

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

Say N and Σ Implicit: Whatever symbols appears in productions

$S = A$ ← Say S Implicit: LHS of top production

$$P = \begin{Bmatrix} \text{P1: } A \rightarrow (A) \\ \text{P2: } A \rightarrow \varepsilon \end{Bmatrix}$$

Collapse rules with the same LHS using bar

Denote grammar as

$$A ::= (A)$$
$$A ::= \varepsilon$$

or equivalently as

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

or even

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

Simplifying Notation: Shorthand

Syntactic Definition: How we use CFGs

"BNF"

Denote grammar as

A ::= **(** A **)**

A ::= $\boldsymbol{\varepsilon}$

or equivalently as

A ::=(A)

| $\boldsymbol{\varepsilon}$

or even

A ::= (A) | $\boldsymbol{\varepsilon}$

Some languages denoted in BNF

Syntactic Definition: How we use CFGs

A ::= (A)
| ε

F ::= **b** G **y e**
| **see ya**
G ::= G **y**
| ε

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

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

$F \Rightarrow$ **see ya**

Y ::= **a** Y
Z ::= **w t f**

$Y \Rightarrow$ a Y $\Rightarrow$ a a Y $\Rightarrow$ a a a Y $\Rightarrow$ …

Accepts no strings
(not even the empty string)

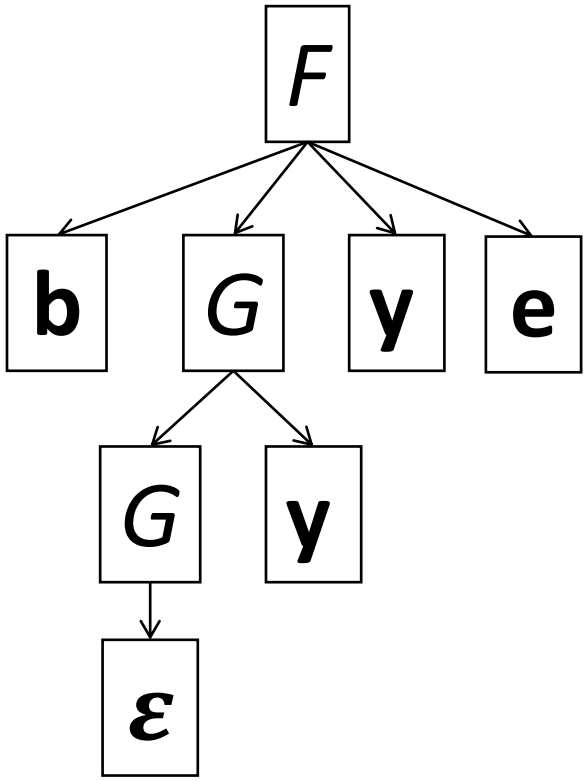
Parse Trees

Syntactic Definition: How we use CFGs

$F \Rightarrow \mathbf{b}\ G\ \mathbf{y}\ \mathbf{e} \Rightarrow \mathbf{b}\ G\ \mathbf{y}\ \mathbf{y}\ \mathbf{e} \Rightarrow \mathbf{b}\ \mathbf{y}\ \mathbf{y}\ \mathbf{e}$

Represent Derivations

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



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**

Derivation Sequence

Prog
Prod. 1

⇒ **begin** Stmts **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

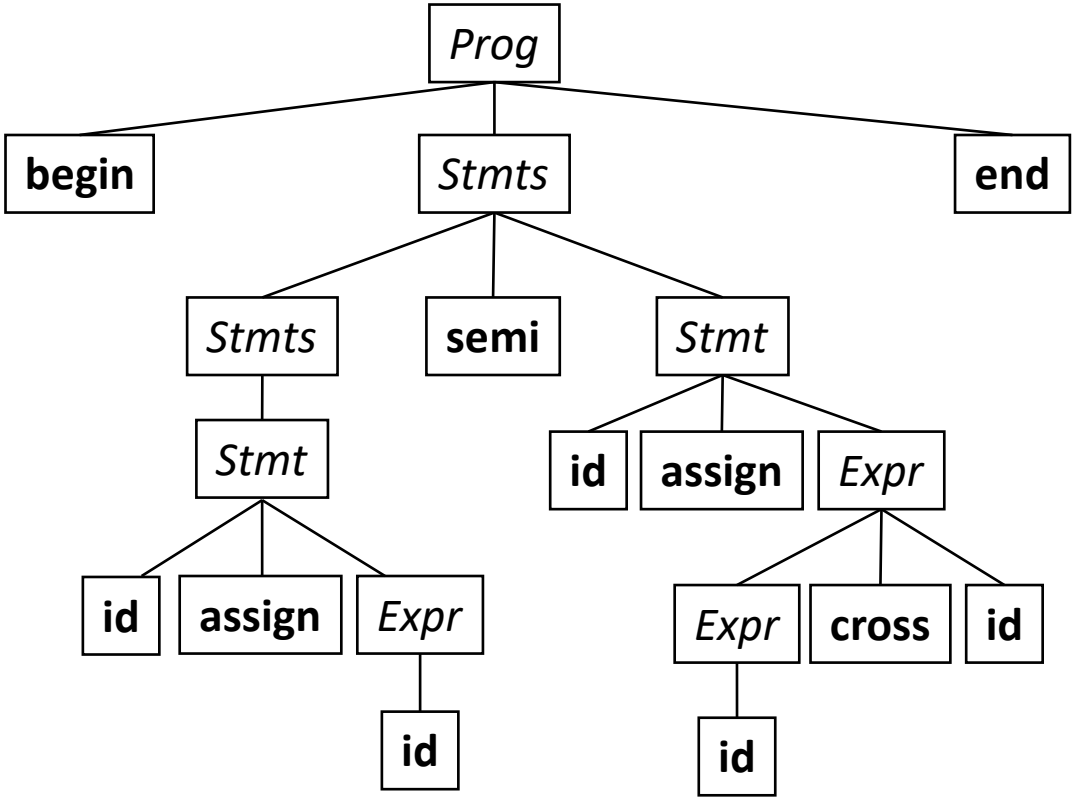
⇒ **begin id assign** *Expr* **semi id assign** *Expr* **end**
Prod. 5

⇒ **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**

Parse Tree



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