Check-In C9

Review: FIRST sets

Calculate FIRST sets for the Grammar

S ::= **lpar** X **rpar**
X ::= **id comma** *X*
| ε

Announcements

Housekeeping

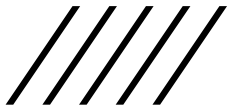
Quick word on academic misconduct

F L I P P E D
W E D N E S D A Y

Quiz 1

Topics:

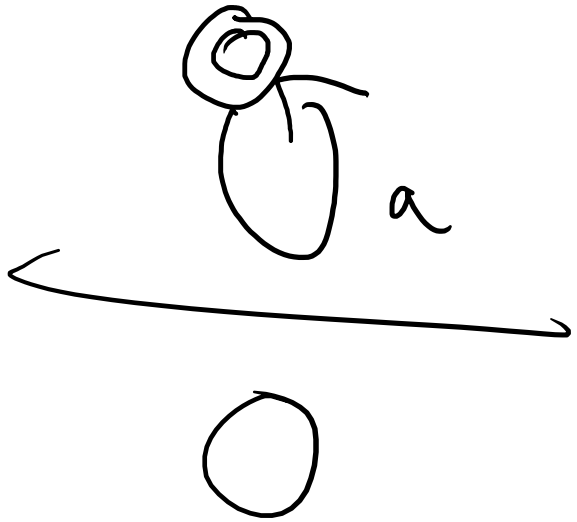
- Phases of the compiler
- Regex to NFA-with-epsilon-transitions
- NFA epsilon elimination
- epsilon-closure sets
- Rabin-Scott Powerset Construction
- Regular Expressions
- DFAs
- State transition tables
- Tokenizers
- Flex patterns
- Context-Free Grammars
- Ambiguous Grammars
- Fixing Precendence in CF Grammars
- Fixing Associativity in CF Grammars
- Backus Normal Form Notation
- Parse Trees
- Syntax-Directed Definitions
- Syntax-Directed Translation

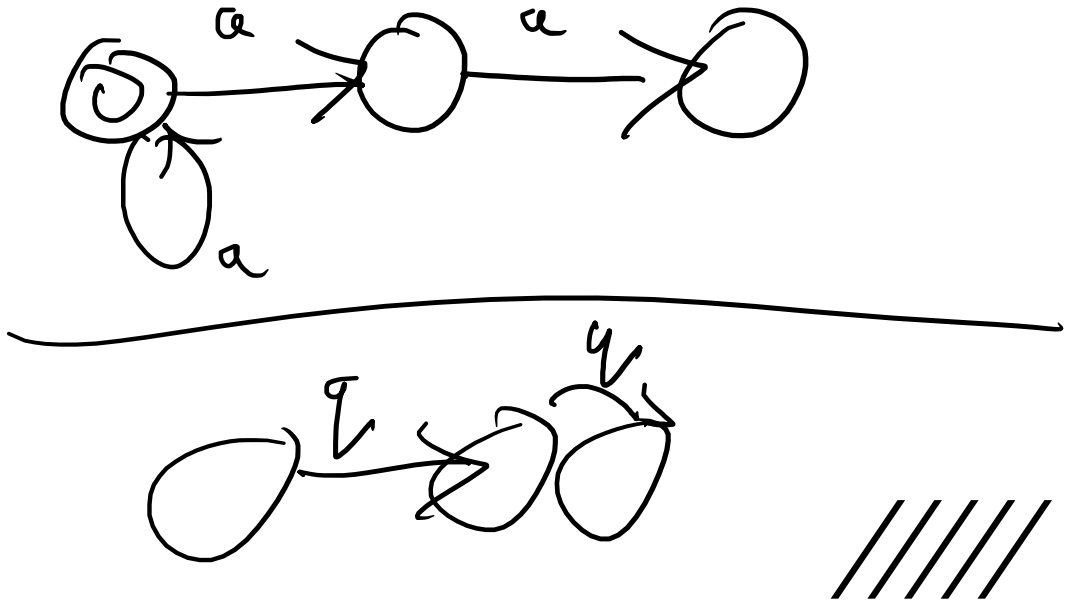


Quiz 1: Question 1, Part 1

(Assume, for this question, that you are the captain of a pirate ship)

Your first mate claims that there exists an NFA with 3 states, but that there exists an equivalent DFA with only 1 state. Is your first mate correct? If so, explain why. If not, give an example of such an NFA / DFA pair

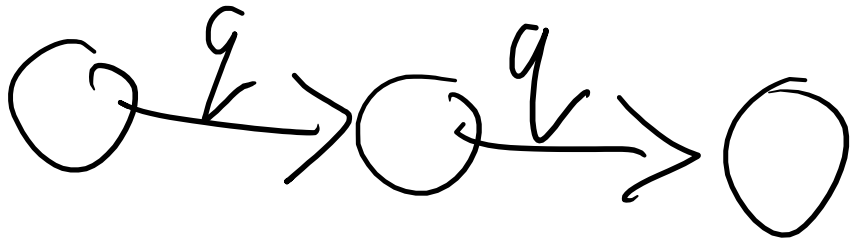


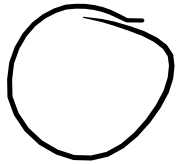


Quiz 1: Question 1, Part 2

(Assume, for this question, that you are the captain of a pirate ship)

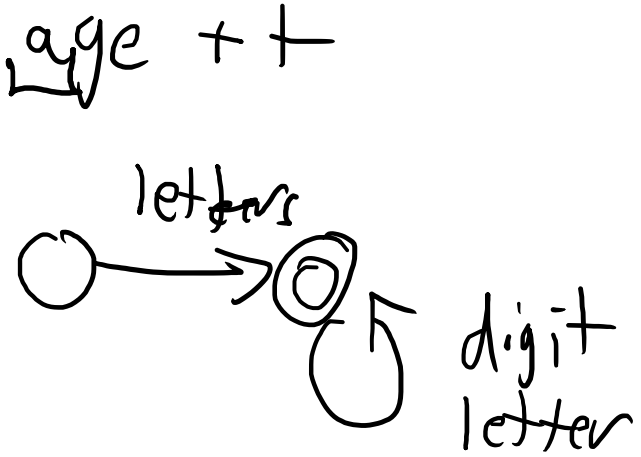
Your helmsman claims that there exists an NFA with 3 states, and that any equivalent DFA has at least 8 states. Is your helmsman correct? If so, give an example of such an NFA / DFA pair. If not, explain why





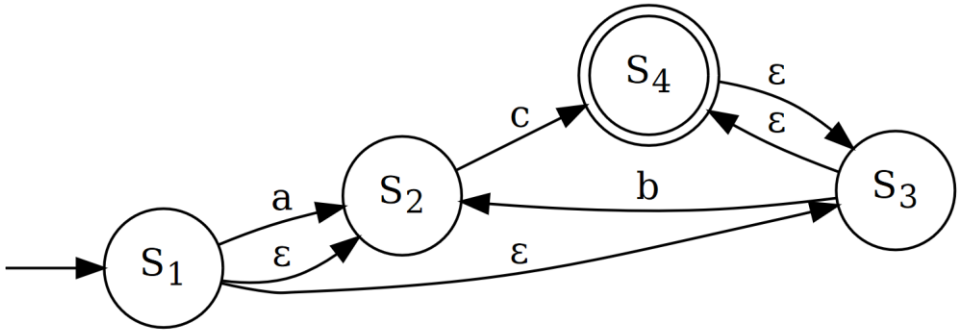
Quiz 1: Question 2

We described a method by which DFAs could be used to create a tokenizer (i.e. a translator from a stream to characters to a stream of tokens). The tokenizer had to backtrack over the input string even after an accepting state had been been found for one of the token DFAs. Explain why the tokenizer has to backtrace and give an example of a token language where backtracking is necessary.



Quiz 1: Question 3

Write the ε-closure of each state in the following finite state machine



S_1 : { S_1, S_2, S_3, S_4 }

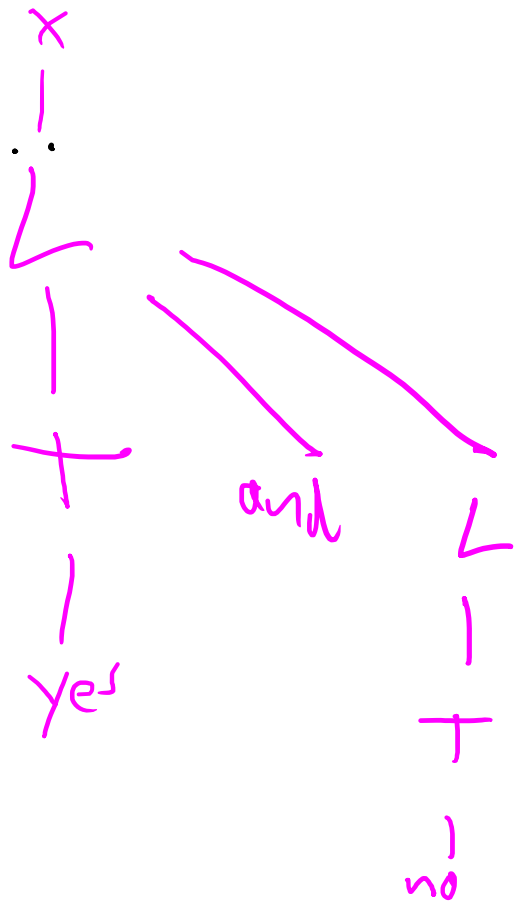
S_2 : { S_2 }

S_3 : { S_3, S_4 }

S_4 : { S_4, S_3 }

Quiz 1: Question 4, Part I

Write out an grammar that recognizes strings that are sequences of the tokens **yes** and **no**, delimited by the **and** token, in any order. In other words, your grammar should recognize **yes and no and no and yes** , as well as **yes**, as well as the empty string. It should NOT recognize **and yes and no**. Your grammar can be ambiguous if you want.



$$X ::= \varepsilon \mid L$$

$$L ::= T \text{ and } L \mid T$$

$$T ::= \text{yes} \mid \text{no}$$

Quiz 1: Question 4, Part 2

yes and no

Add a syntax-definition of your Part I grammar such that any string is translated to the count of yes tokens minus the count of no tokens. For example yes and yes translates to 2. no and yes translates to 0, and no and no translates to -2.

P_1 $X ::= \epsilon$ { $\$\$ = 0$ }

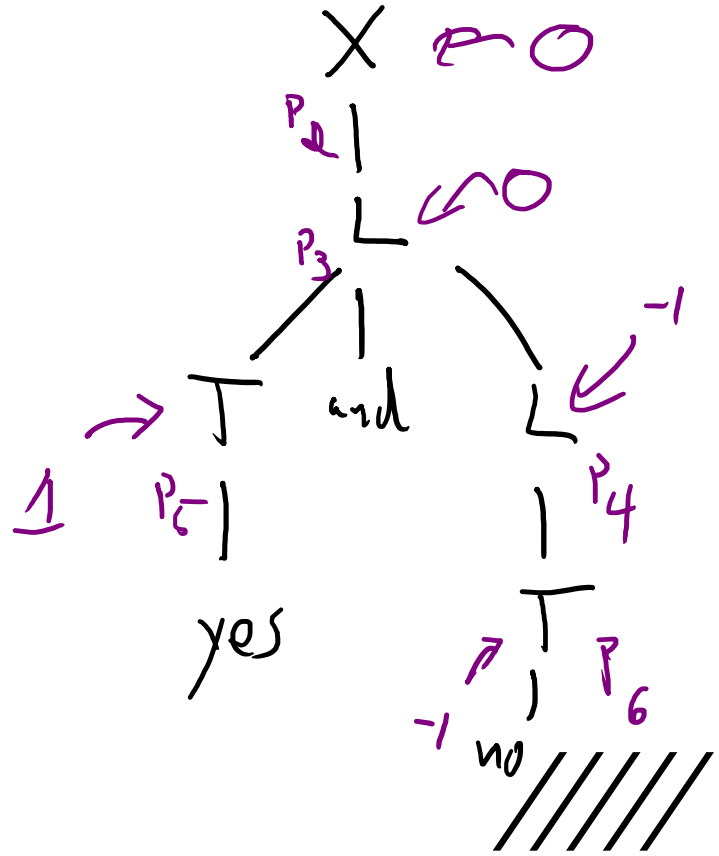
P_2 $X ::= L$ { $\$\$ = \$1$ }

P_3 $L ::= T$ and L { $\$\$ = \$1 + \$3$ }

P_4 $L ::= T$ { $\$\$ = \$1$ }

P_5 $T ::=$ yes { $\$\$ = 1$ }

P_6 $T ::=$ no { $\$\$ = -1$ }



Quiz 1: Question 5

Imagine the following is the rules section of a Flex spec:

```
goo         {std::cout << "1\n"; }
g(o+)od     {std::cout << "2\n"; }
.           {std::cout << "3\n"; }
[.\n]       {std::cout << "4\n"; }
```

What does this spec print on the following input: goodgoop?

Bonus #1

$X ::= X$

$\mid \varepsilon$

$X ::= \varepsilon$

$X ::= \varepsilon$

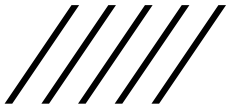
Create a Context Free Grammar that only accepts the empty string, but is still ambiguous

$X ::= Y$

$X ::= Y Z$

$Y ::= \varepsilon$

$Z ::= \varepsilon$



Bonus #2

Consider the following grammar:

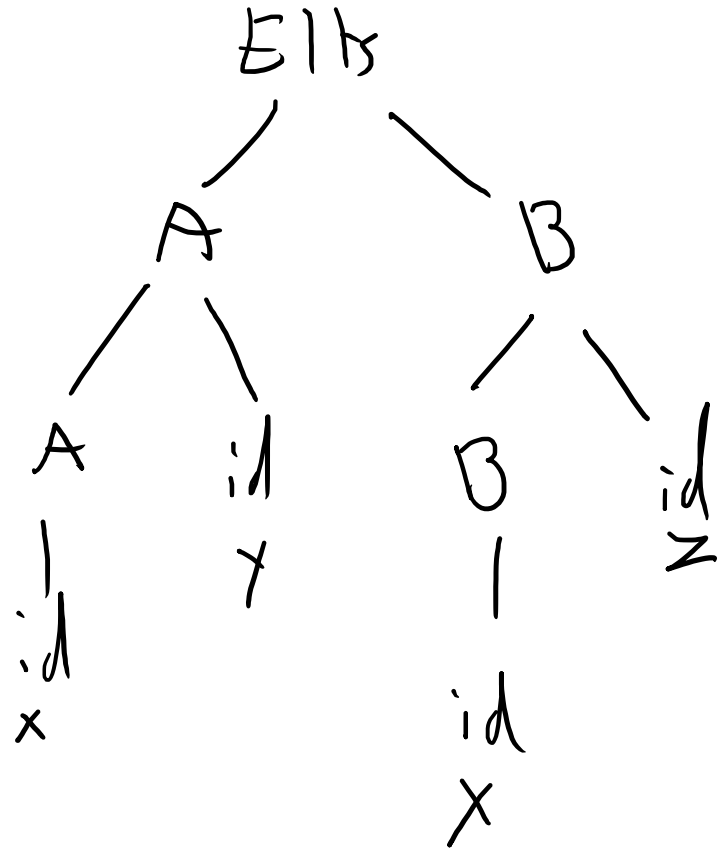
Elts ::= *A* **plus** *B* $$ = B - A

A ::= *A* **id** $$ = $1 ∪ $2

| **id** $$ = {$1.val}

B ::= *B* **id** $$ = $1 ∪ $2

| **id** $$ = {$1.val}



Assume that a tokenizer has already imbued each **id** with a value field that contains the name of the identifier. Write an SDD that sets the *Elt* translation to be set of identifier names mentioned under a *B* nonterminal, but never mentioned under an *A* nonterminal.

Bonus #3

Consider the following grammar:

```
E ::= E - T
    |  T
T ::= T * F
    |  F
F ::= intlit
```

Alter the above expression grammar such that it can recognize expressions with the exponentiation operator ^ e.g. allows the expression 1 - 2 ^ 3. Make sure that your changes do not introduce syntactic ambiguity and that exponetiation is *right-associative* and has the highest precedence of any operator.

Bonus #4

Describe the difference between an abstract syntax-tree and a parse tree.