6.110 Re-lecture 1

Regular expressions, automata, grammars, parse trees

4 lectures in 1

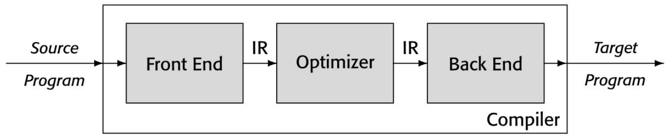
Focus on theory

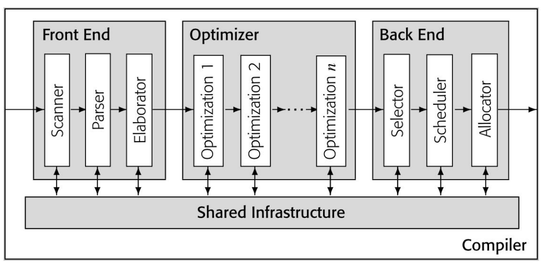
Ideal: 90 minutes

Stop me

Plan

- Overview of compiler components and optimizations
- Regular expressions
- Finite automata
- Duality and constructions
- Tokenization
- Context-free grammar
- Ambiguity
- Precedence





Front end

```
# Comment 1
 # Comment 2

# Factorial:

def fact(  x\
):

  if x == -1:
    return 1.j

  elif x ==0:

    return 1
  else:

        return x* fact(x

- 1)

s = "foo\
\\ \n\'\""
```

```
# Comment 1
 # Comment 2

# Factorial:

def fact(  x\
):

  if x == -1:
    return 1.j

  elif x ==0:

    return 1
  else:

        return x* fact(x

- 1)

s = "foo\
\\ \n\'\""
```

```
(KEYWORD def)
(ID "fact")
(PUNCT "(")
(ID "x")
(PUNCT ")")
(PUNCT ":")
(NEWLINE)
(INDENT)
(KEYWORD if)
(ID "x")
(PUNCT "==")
(PUNCT "-")
(LIT 1)
(PUNCT ":")
(NEWLINE)
(INDENT)
(KEYWORD return)
(LIT +1.i)
(NEWLINE)
(DEDENT)
(KEYWORD elif)
(ID "x")
(PUNCT "==")
(LIT 0)
(PUNCT ":")
(NEWLINE)
(INDENT)
(KEYWORD return)
(LIT 1)
(NEWLINE)
(DEDENT)
(KEYWORD else)
(PUNCT ":")
(NEWLINE)
(INDENT)
(KEYWORD return)
(ID "x")
(PUNCT "*")
(ID "fact")
(PUNCT "(")
(ID "x")
(PUNCT "-")
(LIT 1)
(PUNCT ")")
(NEWLINE)
(DEDENT)
(DEDENT)
(ID "s")
(PUNCT "=")
(LIT "foo\\ \n\'\"")
(NEWLINE)
(ID "fact")
(PUNCT "(")
(LIT 20)
(PUNCT ")")
(NEWLINE)
(ENDMARKER)
```

```
# Comment 1
 # Comment 2

# Factorial:

def fact(  x\
):

  if x == -1:
    return 1.j

  elif x ==0:

    return 1
  else:

        return x* fact(x

- 1)

s = "foo\
\\ \n\'\""
```

```
Module(
  body=[
    FunctionDef(
      name='fact',
      args=arguments(
        posonlyargs=[],
        args=[
          arg(arg='x')],
        kwonlyargs=[],
        kw_defaults=[],
        defaults=[]),
      body=[
        If(
          test=Compare(
            left=Name(id='x', ctx=Load()),
            ops=[
              Eq()],
            comparators=[
              UnaryOp(
                op=USub(),
                operand=Constant(value=1))]),
          body=[
            Return(
              value=Constant(value=1j))],
          orelse=[
      …
```

Intermediate representation

$$a \leftarrow a \times 2 \times b \times c \times d$$

$$a \leftarrow a \times 2 \times b \times c \times d$$

$$\begin{aligned} t_0 &\leftarrow a \times 2 \\ t_1 &\leftarrow t_0 \times b \\ t_2 &\leftarrow t_1 \times c \\ t_3 &\leftarrow t_2 \times d \\ a &\leftarrow t_3 \end{aligned}$$

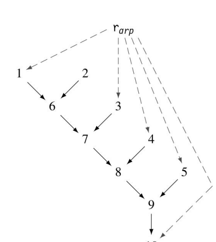
$$a \leftarrow a \times 2 \times b \times c \times d$$

loadAI	r_{arp}, @a	$\Rightarrow$	r_a	// load 'a'
loadI	2	$\Rightarrow$	r_2	// constant 2 into r_2
loadAI	r_{arp}, @b	$\Rightarrow$	r_b	// load 'b'
loadAI	r_{arp}, @c	$\Rightarrow$	r_c	// load 'c'
loadAI	r_{arp}, @d	$\Rightarrow$	r_d	// load 'd'
mult	r_a, r_2	$\Rightarrow$	r_a	// $r_a \leftarrow a \times 2$
mult	r_a, r_b	$\Rightarrow$	r_a	// $r_a \leftarrow (a \times 2) \times b$
mult	r_a, r_c	$\Rightarrow$	r_a	// $r_a \leftarrow (a \times 2 \times b) \times c$
mult	r_a, r_d	$\Rightarrow$	r_a	// $r_a \leftarrow (a \times 2 \times b \times c) \times d$
storeAI	r_a	$\Rightarrow$	r_{arp}, @a	// write r_a back to 'a'

$$a \leftarrow a \times 2 \times b \times c \times d$$

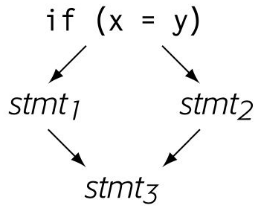
1	loadAI	r_{arp}, @a	$\Rightarrow$	r_a
2	loadI	2	$\Rightarrow$	r_2
3	loadAI	r_{arp}, @b	$\Rightarrow$	r_b
4	loadAI	r_{arp}, @c	$\Rightarrow$	r_c
5	loadAI	r_{arp}, @d	$\Rightarrow$	r_d
6	mult	r_a, r_2	$\Rightarrow$	r_a
7	mult	r_a, r_b	$\Rightarrow$	r_a
8	mult	r_a, r_c	$\Rightarrow$	r_a
9	mult	r_a, r_d	$\Rightarrow$	r_a
10	storeAI	r_a	$\Rightarrow$	r_{arp}, @a

(a) Example Code from Chapter 1



(b) Dependence Graph for the Example

```
if (x = y)
  then stmt1
  else stmt2
stmt3
```



Back end

$$a \leftarrow a \times 2 \times b \times c \times d$$

```
loadAI   r_arp, @a  ⇒ r_a        // load 'a'
loadI    2          ⇒ r_2        // constant 2 into r_2
loadAI   r_arp, @b  ⇒ r_b        // load 'b'
loadAI   r_arp, @c  ⇒ r_c        // load 'c'
loadAI   r_arp, @d  ⇒ r_d        // load 'd'
mult     r_a, r_2   ⇒ r_a        // r_a ← a × 2
mult     r_a, r_b   ⇒ r_a        // r_a ← (a × 2) × b
mult     r_a, r_c   ⇒ r_a        // r_a ← (a × 2 × b) × c
mult     r_a, r_d   ⇒ r_a        // r_a ← (a × 2 × b × c) × d
storeAI  r_a        ⇒ r_arp, @a  // write r_a back to 'a'
```

```
addi    sp,sp,-32
sw      ra,28(sp)
sw      s0,24(sp)
addi    s0,sp,32
sw      a0,-20(s0)
sw      a1,-24(s0)
sw      a2,-28(s0)
sw      a3,-32(s0)
lw      a4,-20(s0)
lw      a5,-24(s0)
mul     a4,a4,a5
lw      a5,-28(s0)
mul     a4,a4,a5
lw      a5,-32(s0)
mul     a5,a4,a5
slli    a5,a5,1
mv      a0,a5
lw      ra,28(sp)
lw      s0,24(sp)
addi    sp,sp,32
jr      ra
```

Formal languages

Alphabet

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

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

$\Sigma = \{\mathtt{false}, \mathtt{true}\}$

Σ = English words

String

abcdababab

11100011001

ε

~~00000...~~

'i' 'like' 'six' 'oh' 'three' 'five'

Language

L = {1, 01, 10, 001, 010, 100, 0001, 0010, 0100, 1000, 00001, 00010, 00100, 01000, 10000, 000001, ...}

(assuming Σ = {0, 1})

L = set of binary strings that contain exactly one 1

(assuming Σ = {0, 1})

L(s) = whether s contains exactly one 1 (yes or no)

(assuming Σ = {0, 1})

L = set of decimal numbers that are divisible by 3

(assuming Σ = {0, 1, 2, ..., 9})

L = set of valid hexadecimal numbers

(assuming Σ = ASCII characters)

L = set of syntactically valid Python programs

(assuming Σ = ASCII characters)

L = set of syntactically valid Python programs

(assuming Σ = Python tokens)

L = set of Python source interpretable without error

Regular languages

Regular expression

`(617|857)-253-(0|1|..|9)(0|1|..|9)(0|1|..|9)(0|1|..|9)`

0 Empty string

0 Empty string
ε

1 A letter from Σ

`0`

2 Concatenation

$a \cdot b$

2 Concatenation

ab

2 Concatenation

234324

3 Alternation

$a \mid b$

3 Alternation/Union

$a \cup b$

3 Alternation

`0|1`

3 Alternation

`000|001|100|101`

3 Alternation

`(0|1)0(0|1)`

3 Alternation

`(617|857)-253-(0|1|..|9)(0|1|..|9)(0|1|..|9)(0|1|..|9)`

4 Kleene star

*a**

4 Kleene star

`(0|1)*`

4 Kleene star

`0*10*`

"regex"

[0-9A-Fa-f]

[01]{8}

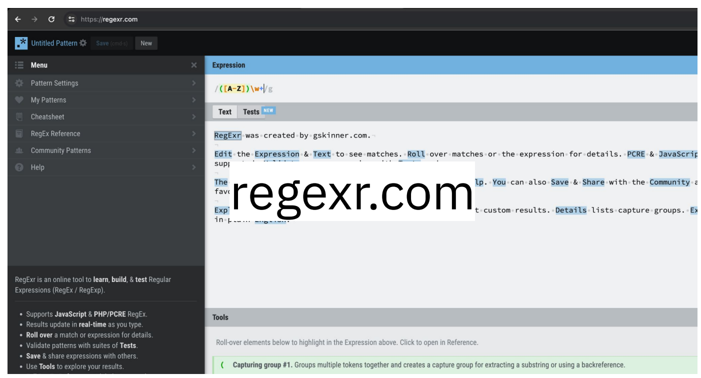
0|(1[01]*)

0x[0-9A-Fa-f]+

0x[0-9A-Fa-f][0-9A-Fa-f]*

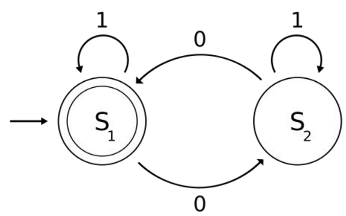
617-253-\d{4}

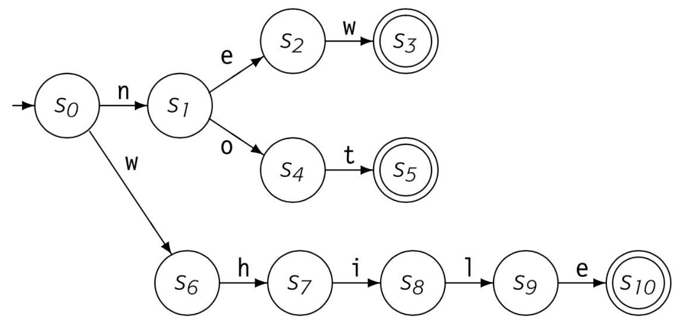
.*



Deterministic Finite Automata

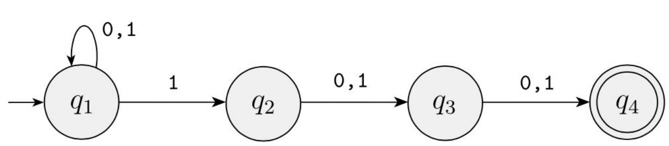
$M = (Q, \Sigma, \delta, q_0, F)$



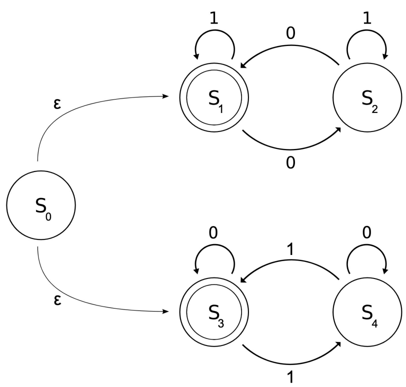


Non-deterministic Finite Automata

L = set of binary strings containing a 1 in the third position from the end



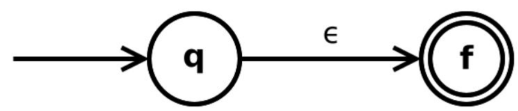
L = set of binary strings with even number of 0s or even number of 1s



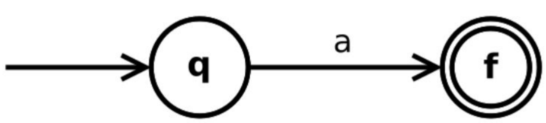
Regex → NFA

Thompson's construction

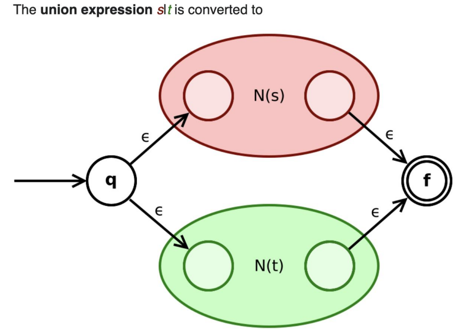
The **empty-expression** ε is converted to

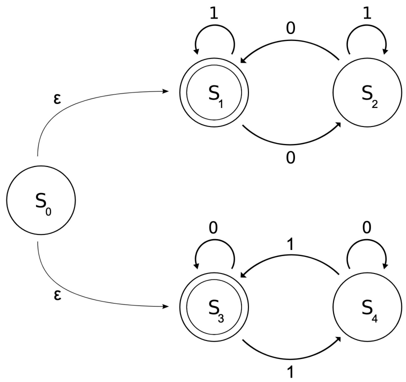


A **symbol** *a* of the input alphabet is converted to

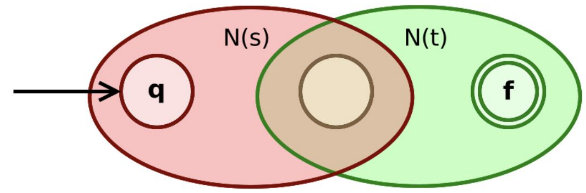


The **union expression** *s*|*t* is converted to

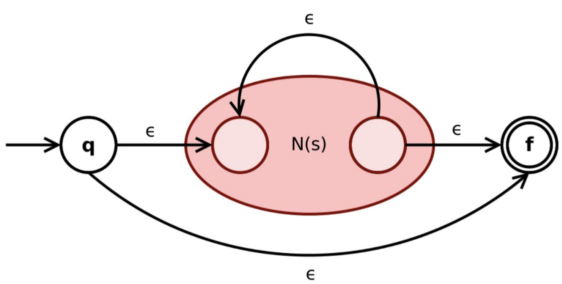




The **concatenation expression** *st* is converted to

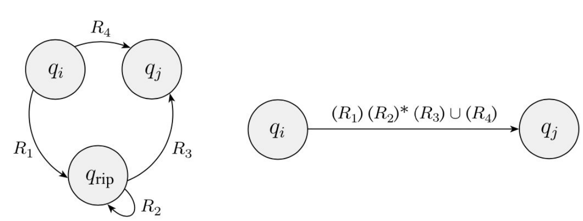


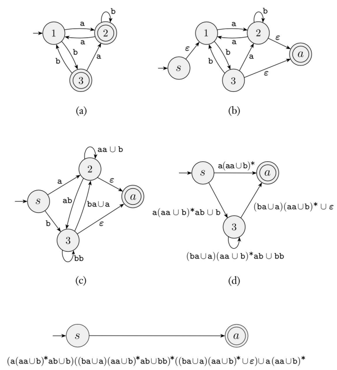
The **Kleene star expression** *s** is converted to



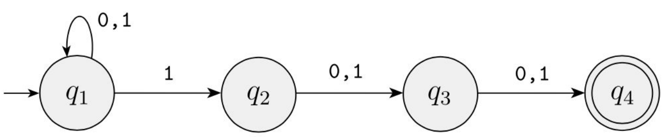
DFA/NFA → Regex

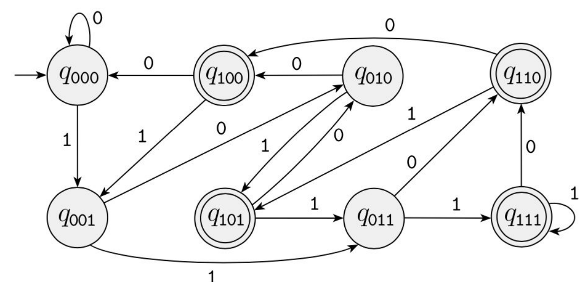
Generalized NFA





NFA → DFA





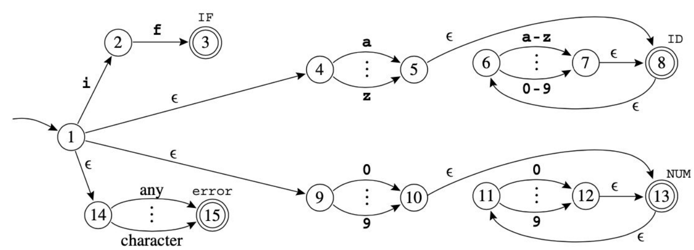
DFA → NFA

Trivial

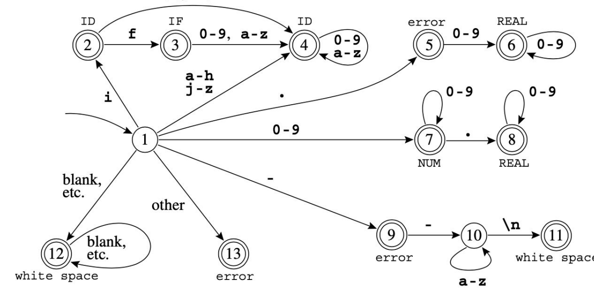
DFA Minimization

Hard

Why?



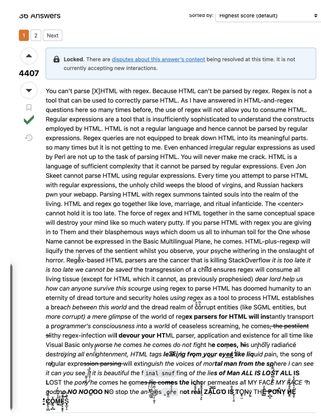
Greedy



L = set of binary strings that start with 0s, followed by an equal number of 1s

Pumping lemma

Take 6.045 or 6.840.


36 Answers

Sorted by: Highest score (default)

1 2 Next

4407

Locked. There are disputes about this answer's content being resolved at this time. It is not currently accepting new interactions.

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the transgression of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach *between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimpse* of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a world of ceaseless screaming, he comes, ~~the pestilent sl~~ithy regex-infection wil**l devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, ~~hi~~**s unholy radiancé destro*ying all enlightenment, HTML tags* ***leaking from your eyes like liq****uid* pain, the song of re~~gular expression parsing~~ will ext*inguish the voices of mor****tal man from the sp****here I can see it can you see it it is beautiful* the f `inal snuf` fing *of the lie****s of Man ALL IS LOST ALL IS** LOST the *pony he comes* he c~~omes he~~ **comes the ich**or permeates *al*l MY FACE *MY FACE* ᵒh god *no* ***NO NOO*OO N**Θ stop *the an*gles `are` **not real ZALGO IS TONY THE PONY HE COMES**

For the quiz, you should know how to:

- Simulate regex/DFA/NFA
- Design a regex/DFA/NFA
- Convert regex to NFA
- Convert NFA to DFA

How to practice: Do textbook exercises!

Context-free grammar

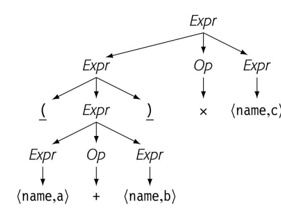
⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → `a` | `the`
⟨NOUN⟩ → `boy` | `girl` | `flower`
⟨VERB⟩ → `touches` | `likes` | `sees`
⟨PREP⟩ → `with`

1	*Expr*	→	(*Expr*)	4	*Op*	→	+
2		|	*Expr Op Expr*	5		|	-
3		|	`name`	6		|	×
				7		|	÷

1	*Expr*	→	(*Expr*)	4	*Op*	→	+
2		|	*Expr Op Expr*	5		|	-
3		|	`name`	6		|	×
				7		|	÷

(a + b) × c

1	*Expr*	→	(*Expr*)	4	*Op*	→	+
2		|	*Expr Op Expr*	5		|	-
3		|	`name`	6		|	×
				7		|	÷



Ambiguity

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *Stmt* `else` *Stmt*
3		|	*Other*

Left factoring

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *Stmt* `else` *Stmt*
3		|	*Other*

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *WithElse* `else` *Stmt*
3		|	*Other*
4	*WithElse*	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
5		|	*Other*

Precedence climbing

1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* *Expr*
3		|	`name`

4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

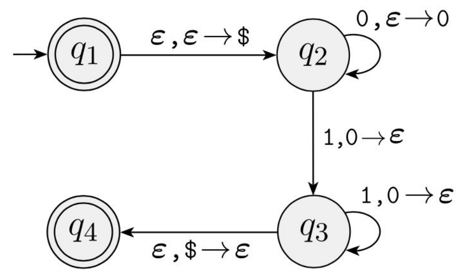
1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* *Expr*
3		|	`name`

4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	`num`
9		|	`name`

Pushdown Automata

L = set of binary strings that start with 0s, followed by an equal number of 1s



Remember this?

```
expression ::= resize ('|' resize)*;
resize     ::= primitive ('@' size)*;
size       ::= (number 'x' number);
primitive  ::= filename | '(' expression ')';

topToBottomOperator ::= '---' '-'*;
filename            ::= [A-Za-z0-9.][A-Za-z0-9._-]*;
number              ::= [0-9]+;
whitespace          ::= [ \\t\\r\\n]+;
```

Extended Backus–Naur form*

(* in spirit)

```
A  ::= B* C
```

```
A  ::= A' C
A' ::= ε | BA'
```

For the quiz, you should know how to:

- Parse a string using a given grammar (draw parse trees)
- Eliminate ambiguity
- Fix precedence issues
    - Make sure you understand the arithmetic examples.
    - Reminder: You can collaborate/ask for help on miniquiz.

How to practice: Do textbook exercises!

Top-down parsing

Recursive descent parser

<rant>

Use first principles

Ask TAs

</rant>

Project 1!

Left factoring (again)

Factor	→	`name`
	|	`name` [*ArgList*]
	|	`name` (*ArgList*)
ArgList	→	*Expr MoreArgs*
MoreArgs	→	, *Expr MoreArgs*
	|	ϵ

Factor	→	`name` *Arguments*
Arguments	→	[*ArgList*]
	|	(*ArgList*)
	|	ϵ
ArgList	→	*Expr MoreArgs*
MoreArgs	→	, *Expr MoreArgs*
	|	ϵ

Left recursion

```
Expr   ::= Expr + Term
         | Expr - Term
         | Term;
Term   ::= Term × Factor
         | Term ÷ Factor
         | Factor;
Factor ::= ( Expr )
         | num
         | name;
```

$$
\begin{array}{lcl}
Fee & \rightarrow & Fee\ \alpha \\
 & | & \beta
\end{array}
\qquad
\begin{array}{lcl}
Fee & \rightarrow & \beta\ Fee' \\
Fee' & \rightarrow & \alpha\ Fee' \\
 & | & \epsilon
\end{array}
$$

```
Expr   ::= Term Expr';
Expr'  ::= + Term Expr'
         | - Term Expr'
         | ε;
Term   ::= Factor Term'
Term'  ::= × Factor Term'
         | ÷ Factor Term'
         | ε;
Factor ::= ( Expr )
         | num
         | name
```

```
Expr   ::= Term ((+|-) Term)*
Term   ::= Factor ((×|÷) Factor)*
Factor ::= ( Expr )
         | num
         | name;
```

Indirect left recursion

Constraint propagation

$$NT \rightarrow \varepsilon \\ \Rightarrow \\ NT \rightarrow^{*} \varepsilon$$

$$NT_0 \rightarrow NT_1 NT_2 \ldots \text{ and } NT_i \rightarrow^{*} \varepsilon \\ \Rightarrow \\ NT_0 \rightarrow^{*} \varepsilon$$

$$\mathbf{T} \in \text{First}(\mathbf{T})$$

$$x \in \text{First}(S) \\ \Rightarrow \\ x \in \text{First}(S\, S_1\, S_2\, S_3 \ldots)$$

$$x \in \text{First}(S) \\ \Rightarrow \\ x \in \text{First}(S\beta)$$

$$\text{First}(S) \subseteq \text{First}(S\beta)$$

$$x \in \text{First}(\beta) \quad \text{and} \quad NT \to^{*} \varepsilon \\ \Rightarrow \\ x \in \text{First}(NT\,\beta)$$

$$x \in \text{First}(S\beta) \quad \text{and} \quad (NT \to S\beta) \\ \Rightarrow \\ x \in \text{First}(NT)$$