CSC 240

CONTEXT FREE LANGUAGES

WAYS TO DESCRIBE A LANGUAGE

Finite automata, which recognize strings in the language.

Regular expressions, which describe strings in the language.

Context-Free grammar, which describe structure of the language.

THE ADD AND SUBTRACT LANGUAGE

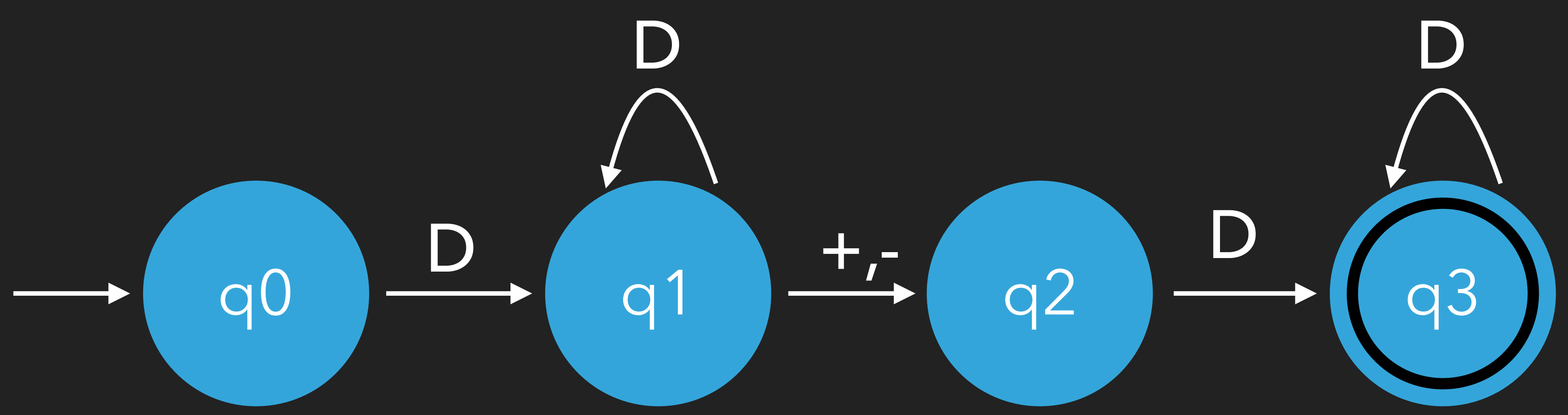
L = {w | w is a pair of integers which are added or subtracted}

L = {“4 + 2”, 5 - 1”, “20 - 14”, …}

D = {0…9}

Σ = {D, +, -}

$R = D^{+} (+ \cup -) D^{+}$



THE ADD AND SUBTRACT LANGUAGE

L = {w | w is a pair of integers which are added or subtracted}

L = {“4 + 2”, 5 - 1”, “20 - 14”, …}

integer → Operator → integer

$$R = D^{+} (+ \cup -) D^{+}$$

Expression

Context-Free Grammar of L

Expression → integer Operator integer

Operator → +

Operator → -

Context-Free Grammars

Variables, also known as nonterminal symbols.

Terminals, consisting of the alphabet of the grammar.

Substitution rules, also known as production rules.

Starting Symbol, which must be a nonterminal.

Expression → integer Operator integer
Operator → +
Operator → -

Context-Free Grammars

$G = (V, \Sigma, R, S)$

- V: A finite set called variables.
- Σ: A finite set called the terminals (must be disjoint from V).
- R: A finite set of substitution rules.
- S: is the start variable, where $S \in V$.

CFG FORMALIZED

L = {w | w is a pair of integers which are added or subtracted}

G = ({Expression, Operator} , { $\mathbb{Z}$, +, -}, R, Expression)

Expression → integer Operator integer

Operator → +

Operator → -

DERIVATIONS

L = {w | w is a pair of integers which are added or subtracted}
or a single integer value}

Grammar

Expression → integer
Expression → integer Operator integer
Operator → +
Operator → -

Derivation

Expression ⇒ integer Operator integer
⇒ integer + integer

Expression ⇒ integer Operator integer
⇒ integer - integer

Expression ⇒ integer

DESIGNING CFG - FUNCTION PROTOTYPE

L = {w | w is a valid C++ Function Prototype }

```
void printSummary();

int doubleNumber(int x);

float calculateAverage(int x, int y);
```

building grammars is a recursive process, so watch for places where you need a base case.

Prototype → Return name (Args) ;

Return → Type | void

Type → int | double

Args → ε | ArgList

ArgList → SingleArt | ArgList, SingleArg

SingleArg → Type name

CONVERT REGEX TO CFG

0*1

Expression → Zeros 1

Zeros → ε | Zeros0

building grammars is a recursive process, so watch for places where you need a base case.

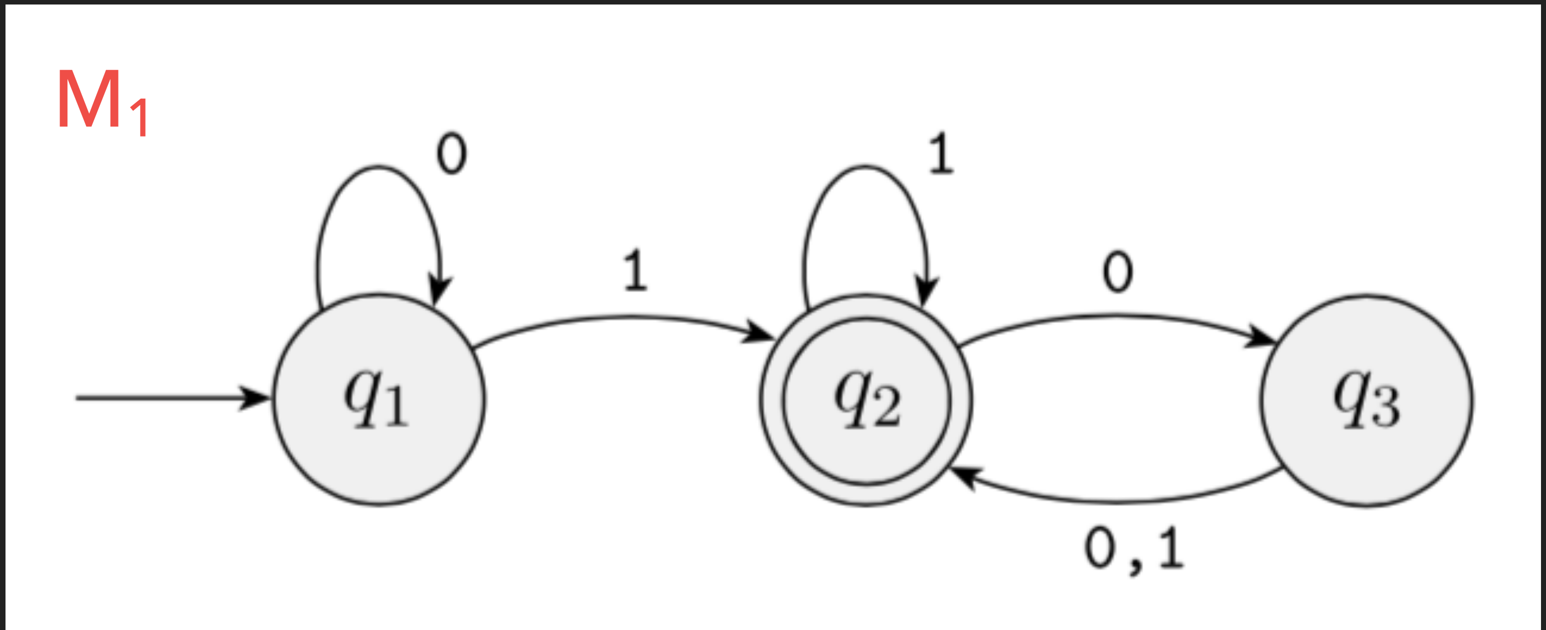
a(b ∪ c*)

Expression → ε Options

Options → b | Cees

Cees → ε | Ceesc

DFA TO CFG



$\mathscr{L}(M_1)$ = { w | w contains at least one 1 and an even number of 0's follow the last 1 }

01100

0110

Grammar

q1 → 0 q1 | 1 q2

q2 → 1 q2 | 0 q3 | ε

q3 → 1 q2 | 0 q2

Derivation

q1 ⇒ 0 q1
⇒ 01 q2
⇒ 011 q2
⇒ 0110 q3
⇒ 01100 q2
⇒ 01100ε
⇒ 01100 ✓

q1 ⇒ 0 q1
⇒ 01 q2
⇒ 011 q2
⇒ 0110 q3 ✗

RLS ARE CFLS BUT CFLS ARE NOT ALL RLS

$L = \{ a^n b^n \mid n \in \mathbb{N} \}$

Can we build an NFA for this language?

No! (Remember to watch the Pumping Lemma video)

This language requires an infinite amount of memory and no FINITE automata has INFINITE memory.

$L = \{ a^n b^n \mid n \in \mathbb{N} \}$

Can we build a CFG for this language?

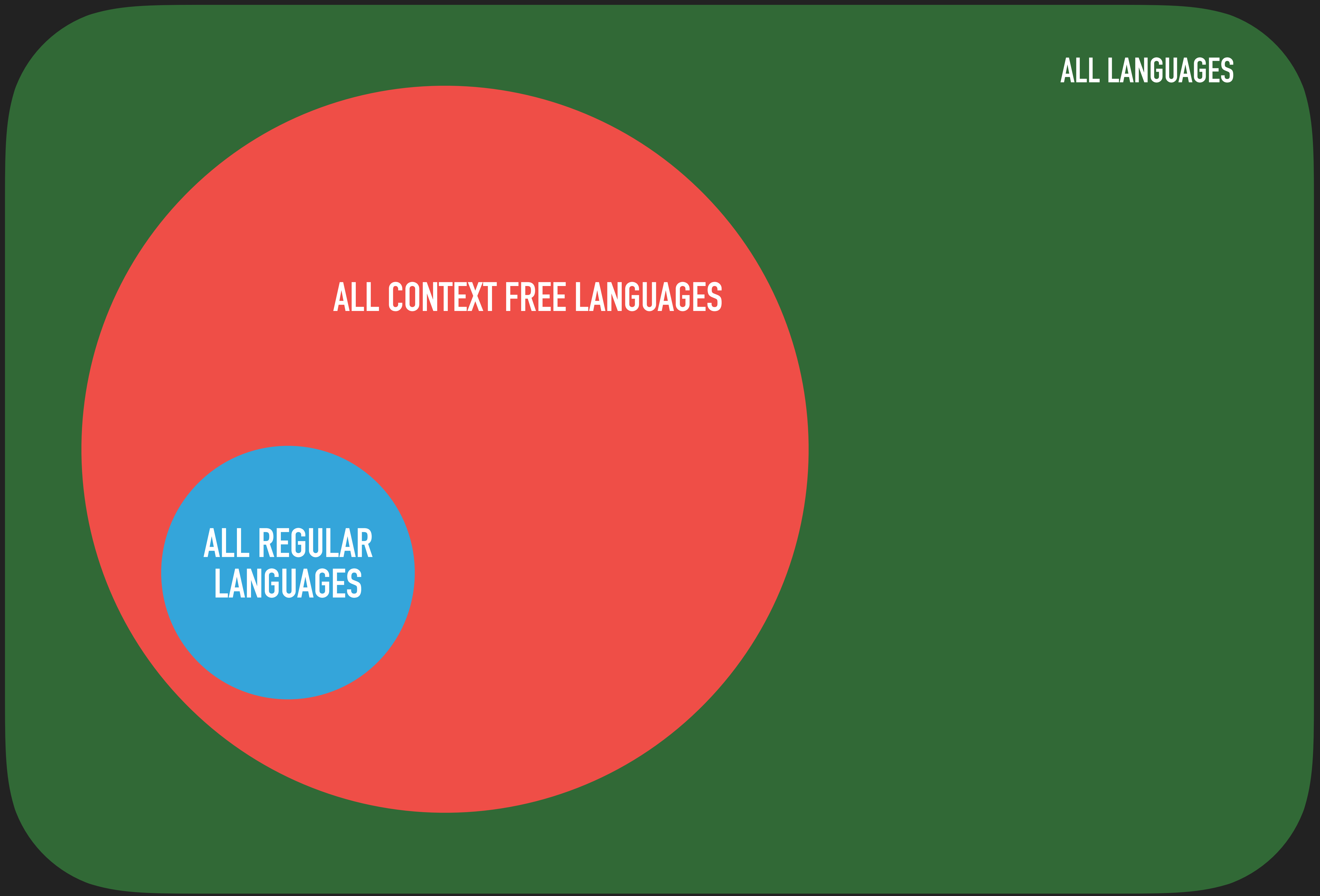
$X \rightarrow aXb \mid \varepsilon$

Derivation

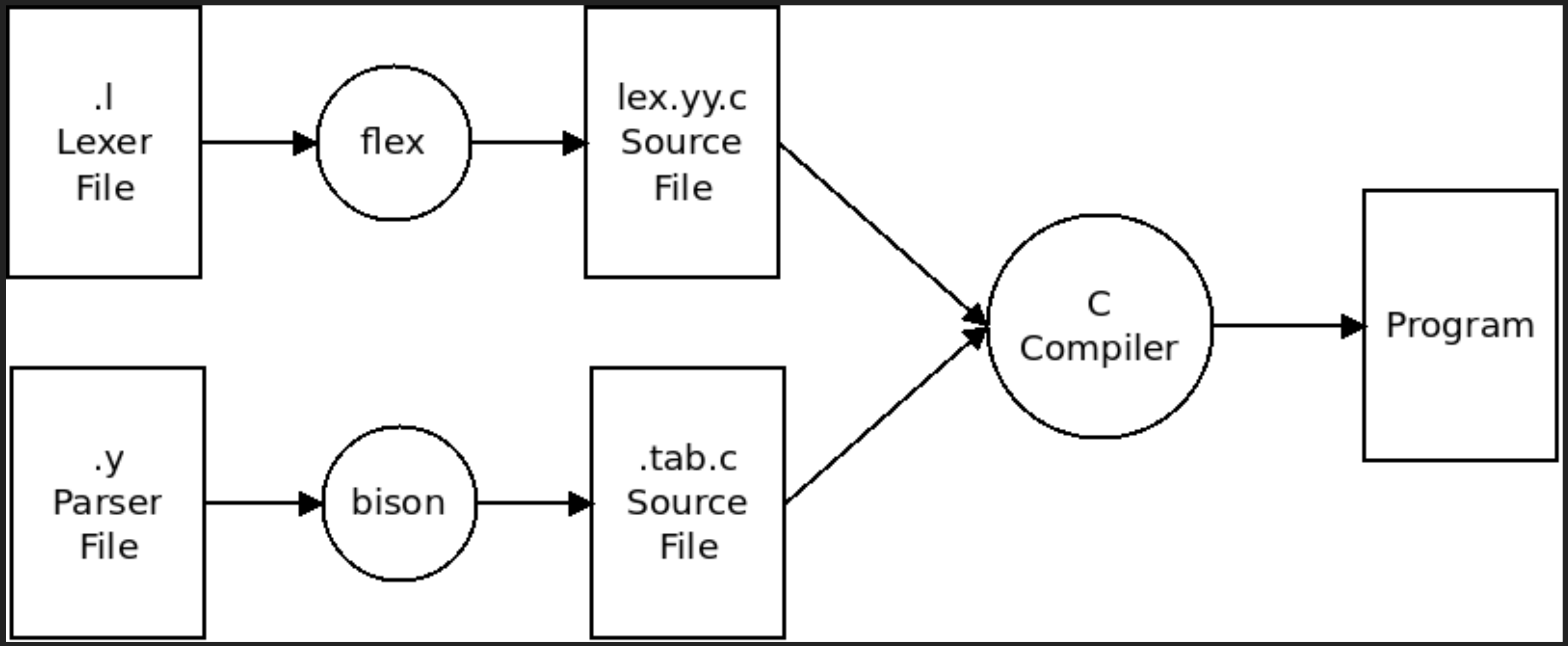
$$
\begin{aligned}
X &\Rightarrow aXb \\
&\Rightarrow aaXbb \\
&\Rightarrow aaaXbbb \\
&\Rightarrow aaa\varepsilon bbb \\
&\Rightarrow aaabbb
\end{aligned}
$$

Because of their recursive nature, a CFG allows us to describe languages that require INFINITE memory.

RLS ARE CFLS BUT CFLS ARE NOT ALL RLS



CFGS IN PARSERS



http://cs.umw.edu/~finlayson/class/fall13/cpsc401/notes/08-bison.html