Context-Free Grammars

Describing Languages

- We've seen two models for the regular languages:
  - **Automata** accept precisely the strings in the language.
  - **Regular expressions** describe precisely the strings in the language.
- Finite automata ***recognize*** strings in the language.
  - Perform a computation to determine whether a specific string is in the language.
- Regular expressions ***match*** strings in the language.
  - Describe the general shape of all strings in the language.

Context-Free Grammars

- A ***context-free grammar*** (or ***CFG***) is an entirely different formalism for defining a class of languages.
- Goal: Give a procedure for listing off all strings in the language.
- CFGs are best explained by example...

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \mathbf{E\ Op}\ (\mathbf{E}) \\
\Rightarrow\ & \mathbf{E\ Op}\ (\mathbf{E\ Op\ E}) \\
\Rightarrow\ & \mathbf{E}\ \texttt{*}\ (\mathbf{E\ Op\ E}) \\
\Rightarrow\ & \texttt{int * (}\mathbf{E\ Op\ E}\texttt{)} \\
\Rightarrow\ & \texttt{int * (int }\mathbf{Op\ E}\texttt{)} \\
\Rightarrow\ & \texttt{int * (int }\mathbf{Op}\texttt{ int)} \\
\Rightarrow\ & \texttt{int * (int + int)}
\end{aligned}
$$

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

$\mathbf{E} \rightarrow$ `int`

$\mathbf{E} \rightarrow \mathbf{E}\ \mathbf{Op}\ \mathbf{E}$

$\mathbf{E} \rightarrow$ `(`$\mathbf{E}$`)`

$\mathbf{Op} \rightarrow$ `+`

$\mathbf{Op} \rightarrow$ `-`

$\mathbf{Op} \rightarrow$ `*`

$\mathbf{Op} \rightarrow$ `/`

$\mathbf{E}$

$\Rightarrow \mathbf{E}\ \mathbf{Op}\ \mathbf{E}$

$\Rightarrow \mathbf{E}\ \mathbf{Op}$ `int`

$\Rightarrow$ `int` $\mathbf{Op}$ `int`

$\Rightarrow$ `int / int`

Context-Free Grammars

- Formally, a context-free grammar is a collection of four objects:
  - A set of ***nonterminal symbols*** (also called ***variables***),
  - A set of ***terminal symbols*** (the ***alphabet*** of the CFG)
  - A set of ***production rules*** saying how each nonterminal can be converted by a string of terminals and nonterminals, and
  - A ***start symbol*** (which must be a nonterminal) that begins the derivation.

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

Some CFG Notation

- Capital letters in **Bold Red Uppercase** will represent nonterminals.
  - i.e. **A**, **B**, **C**, **D**
- Lowercase letters in `blue monospace` will represent terminals.
  - i.e. `t`, `u`, `v`, `w`
- Lowercase Greek letters in ***gray italics*** will represent arbitrary strings of terminals and nonterminals.
  - i.e. α, γ, ω

A Notational Shorthand

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

A Notational Shorthand

$$\mathbf{E} \to \texttt{int} \mid \mathbf{E}\ \mathbf{Op}\ \mathbf{E} \mid \texttt{(}\mathbf{E}\texttt{)}$$

$$\mathbf{Op} \to \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

Derivations

$$E \to E\ Op\ E \mid \texttt{int} \mid (E)$$
$$Op \to + \mid * \mid - \mid /$$

$$\begin{aligned}
& E \\
\Rightarrow\ & E\ Op\ E \\
\Rightarrow\ & E\ Op\ (E) \\
\Rightarrow\ & E\ Op\ (E\ Op\ E) \\
\Rightarrow\ & E\ *\ (E\ Op\ E) \\
\Rightarrow\ & \texttt{int}\ *\ (E\ Op\ E) \\
\Rightarrow\ & \texttt{int}\ *\ (\texttt{int}\ Op\ E) \\
\Rightarrow\ & \texttt{int}\ *\ (\texttt{int}\ Op\ \texttt{int}) \\
\Rightarrow\ & \texttt{int}\ *\ (\texttt{int} + \texttt{int})
\end{aligned}$$

- A sequence of steps where nonterminals are replaced by the right-hand side of a production is called a ***derivation***.
- If string α derives string ω, we write $\alpha \Rightarrow^* \omega$.
- In the example on the left, we see $E \Rightarrow^*$ `int * (int + int)`.

The Language of a Grammar

- If G is a CFG with alphabet Σ and start symbol **S**, then the ***language of G*** is the set

$$\mathscr{L}(G) = \{ \boldsymbol{\omega} \in \Sigma^* \mid \mathbf{S} \Rightarrow^* \boldsymbol{\omega} \}$$

- That is, $\mathscr{L}(G)$ is the set of strings derivable from the start symbol.
- Note: $\boldsymbol{\omega}$ must be in Σ^*, the set of strings made from terminals. Strings involving nonterminals aren't in the language.

More Context-Free Grammars

- Chemicals!
  - $C_{19}H_{14}O_5S$
  - $Cu_3(CO_3)_2(OH)_2$
  - MnO_4^-
  - S^{2-}

Form → **Cmp** | **Cmp Ion**

Cmp → **Term** | **Term Num** | **Cmp Cmp**

Term → **Elem** | `(`**Cmp**`)`

Elem → `H` | `He` | `Li` | `Be` | `B` | `C` | …

Ion → `+` | `-` | **IonNum** `+` | **IonNum** `-`

IonNum → `2` | `3` | `4` | ...

Num → `1` | **IonNum**

CFGs for Chemistry

Form → **Cmp** | **Cmp Ion**

Cmp → **Term** | **Term Num** | **Cmp Cmp**

Term → **Elem** | `(`**Cmp**`)`

Elem → `H` | `He` | `Li` | `Be` | `B` | `C` | …

Ion → `+` | `-` | **IonNum** `+` | **IonNum** `-`

IonNum → `2` | `3` | `4` | ...

Num → `1` | **IonNum**

Form

⇒ **Cmp Ion**

⇒ **Cmp Cmp Ion**

⇒ **Cmp Term Num Ion**

⇒ **Term Term Num Ion**

⇒ **Elem Term Num Ion**

⇒ `Mn` **Term Num Ion**

⇒ `Mn` **Elem Num Ion**

⇒ `MnO` **Num Ion**

⇒ `MnO` **IonNum Ion**

⇒ MnO_4 **Ion**

⇒ MnO_4^-

CFGs for Programming Languages

```
BLOCK  → STMT
       | { STMTS }

STMTS  → ε
       | STMT STMTS

STMT   → EXPR;
       | if (EXPR) BLOCK
       | while (EXPR) BLOCK
       | do BLOCK while (EXPR);
       | BLOCK
       | …

EXPR   → var
       | const
       | EXPR + EXPR
       | EXPR – EXPR
       | EXPR = EXPR
       | …
```

CFGs for Programming Languages

```
BLOCK  → STMT
        | { STMTS }

STMTS  →  ε
        | STMT STMTS

STMT   →  EXPR;
        | if (EXPR) BLOCK
        | while (EXPR) BLOCK
        | do BLOCK while (EXPR);
        | BLOCK
        | …

EXPR   →    var
        |   const
        |   EXPR + EXPR
        |   EXPR – EXPR
        |   EXPR = EXPR
        |   …
```

BLOCK

CFGs for Programming Languages

BLOCK → **STMT**
  | `{` **STMTS** `}`

STMTS → ε
  | **STMT STMTS**

STMT → **EXPR**`;`
  | `if (`**EXPR**`)` **BLOCK**
  | `while (`**EXPR**`)` **BLOCK**
  | `do` **BLOCK** `while (`**EXPR**`);`
  | **BLOCK**
  | …

EXPR → `var`
  | `const`
  | **EXPR** `+` **EXPR**
  | **EXPR** `–` **EXPR**
  | **EXPR** `=` **EXPR**
  | …

`{`
  STMTS
`}`

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** `+` **EXPR**
| **EXPR** `–` **EXPR**
| **EXPR** `=` **EXPR**
| …

```
{
    STMT
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
　　| `{` **STMTS** `}`

STMTS → ε
　　| **STMT STMTS**

STMT → **EXPR**`;`
　　| `if (`**EXPR**`)` **BLOCK**
　　| `while (`**EXPR**`)` **BLOCK**
　　| `do` **BLOCK** `while (`**EXPR**`);`
　　| **BLOCK**
　　| …

EXPR → `var`
　　| `const`
　　| **EXPR** `+` **EXPR**
　　| **EXPR** `–` **EXPR**
　　| **EXPR** `=` **EXPR**
　　| …

```
{
    EXPR;
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** `+` **EXPR**
| **EXPR** `–` **EXPR**
| **EXPR** `=` **EXPR**
| …

`{`
EXPR `=` **EXPR**`;`
STMTS
`}`

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** `+` **EXPR**
| **EXPR** `–` **EXPR**
| **EXPR** `=` **EXPR**
| …

```
{
    var = EXPR;
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** + **EXPR**
| **EXPR** – **EXPR**
| **EXPR** = **EXPR**
| …

```
{
    var = EXPR * EXPR;
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** `+` **EXPR**
| **EXPR** `–` **EXPR**
| **EXPR** `=` **EXPR**
| …

```
{
    var = var * EXPR;
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT → EXPR;
     | if (EXPR) BLOCK
     | while (EXPR) BLOCK
     | do BLOCK while (EXPR);
     | BLOCK
     | …

EXPR → var
     | const
     | EXPR + EXPR
     | EXPR – EXPR
     | EXPR = EXPR
     | …
```

```
{
    var = var * var;
    STMTS
}
```

CFGs for Programming Languages

BLOCK → STMT
| { STMTS }

STMTS → ε
| STMT STMTS

STMT → EXPR;
| if (EXPR) BLOCK
| while (EXPR) BLOCK
| do BLOCK while (EXPR);
| BLOCK
| …

EXPR → var
| const
| EXPR + EXPR
| EXPR – EXPR
| EXPR = EXPR
| …

```
{
    var = var * var;
    STMT
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT →  EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR →   var
      |  const
      |  EXPR + EXPR
      |  EXPR – EXPR
      |  EXPR = EXPR
      |  …
```

```
{
    var = var * var;
    if (var) BLOCK
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** `+` **EXPR**
| **EXPR** `–` **EXPR**
| **EXPR** `=` **EXPR**
| …

```
{
    var = var * var;
    if (var) STMT
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT  →  EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR  →    var
      |    const
      |    EXPR + EXPR
      |    EXPR – EXPR
      |    EXPR = EXPR
      |    …
```

```
{
    var = var * var;
    if (var) EXPR;
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
  | `{` **STMTS** `}`

STMTS → ε
  | **STMT STMTS**

STMT → **EXPR**`;`
  | `if (`**EXPR**`)` **BLOCK**
  | `while (`**EXPR**`)` **BLOCK**
  | `do` **BLOCK** `while (`**EXPR**`);`
  | **BLOCK**
  | …

EXPR → `var`
  | `const`
  | **EXPR** + **EXPR**
  | **EXPR** – **EXPR**
  | **EXPR** = **EXPR**
  | …

```
{
    var = var * var;
    if (var) EXPR = EXPR;
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
  | `{` **STMTS** `}`

STMTS → ε
  | **STMT STMTS**

STMT → **EXPR**`;`
  | `if (`**EXPR**`)` **BLOCK**
  | `while (`**EXPR**`)` **BLOCK**
  | `do` **BLOCK** `while (`**EXPR**`);`
  | **BLOCK**
  | …

EXPR → `var`
  | `const`
  | **EXPR** + **EXPR**
  | **EXPR** – **EXPR**
  | **EXPR** = **EXPR**
  | …

```
{
    var = var * var;
    if (var) var = EXPR;
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** + **EXPR**
| **EXPR** – **EXPR**
| **EXPR** = **EXPR**
| …

```
{
    var = var * var;
    if (var) var = const;
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** + **EXPR**
| **EXPR** – **EXPR**
| **EXPR** = **EXPR**
| …

```
{
    var = var * var;
    if (var) var = const;
    STMT
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT →  EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR →    var
      |   const
      |   EXPR + EXPR
      |   EXPR – EXPR
      |   EXPR = EXPR
      |   …
```

```
{
    var = var * var;
    if (var) var = const;
    while (var) BLOCK
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** + **EXPR**
| **EXPR** – **EXPR**
| **EXPR** = **EXPR**
| …

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        STMTS
    }
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
　　　　| `{` **STMTS** `}`

STMTS → ε
　　　　| **STMT STMTS**

STMT → **EXPR**`;`
　　　　| `if (`**EXPR**`)` **BLOCK**
　　　　| `while (`**EXPR**`)` **BLOCK**
　　　　| `do` **BLOCK** `while (`**EXPR**`);`
　　　　| **BLOCK**
　　　　| …

EXPR → `var`
　　　　| `const`
　　　　| **EXPR** + **EXPR**
　　　　| **EXPR** – **EXPR**
　　　　| **EXPR** = **EXPR**
　　　　| …

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        STMT
        STMTS
    }
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT → EXPR;
     | if (EXPR) BLOCK
     | while (EXPR) BLOCK
     | do BLOCK while (EXPR);
     | BLOCK
     | …

EXPR → var
     | const
     | EXPR + EXPR
     | EXPR – EXPR
     | EXPR = EXPR
     | …
```

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        EXPR;
        STMTS
    }
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| `{` **STMTS** `}`

STMTS → `ε`
| **STMT STMTS**

STMT → **EXPR**`;`
| `if (`**EXPR**`)` **BLOCK**
| `while (`**EXPR**`)` **BLOCK**
| `do` **BLOCK** `while (`**EXPR**`);`
| **BLOCK**
| …

EXPR → `var`
| `const`
| **EXPR** `+` **EXPR**
| **EXPR** `–` **EXPR**
| **EXPR** `=` **EXPR**
| …

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        EXPR = EXPR;
        STMTS
    }
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
| { **STMTS** }

STMTS → ε
| **STMT STMTS**

STMT → **EXPR**;
| if (**EXPR**) **BLOCK**
| while (**EXPR**) **BLOCK**
| do **BLOCK** while (**EXPR**);
| **BLOCK**
| …

EXPR → var
| const
| **EXPR** + **EXPR**
| **EXPR** – **EXPR**
| **EXPR** = **EXPR**
| …

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        var = EXPR;
        STMTS
    }
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT  → EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR  → var
      | const
      | EXPR + EXPR
      | EXPR – EXPR
      | EXPR = EXPR
      | …
```

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        var = EXPR + EXPR;
        STMTS
    }
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT  → EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR  →   var
      |   const
      |   EXPR + EXPR
      |   EXPR – EXPR
      |   EXPR = EXPR
      |   …
```

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        var = var + EXPR;
        STMTS
    }
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT  →  EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR  →    var
      |    const
      |    EXPR + EXPR
      |    EXPR – EXPR
      |    EXPR = EXPR
      |    …
```

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        var = var + const;
        STMTS
    }
    STMTS
}
```

CFGs for Programming Languages

BLOCK → **STMT**
  | `{` **STMTS** `}`

STMTS → `ε`
  | **STMT STMTS**

STMT → **EXPR**`;`
  | `if (`**EXPR**`)` **BLOCK**
  | `while (`**EXPR**`)` **BLOCK**
  | `do` **BLOCK** `while (`**EXPR**`);`
  | **BLOCK**
  | …

EXPR → `var`
  | `const`
  | **EXPR** `+` **EXPR**
  | **EXPR** `–` **EXPR**
  | **EXPR** `=` **EXPR**
  | …

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        var = var + const;
    }
    STMTS
}
```

CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS

STMT  →  EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR  →  var
      |  const
      |  EXPR + EXPR
      |  EXPR – EXPR
      |  EXPR = EXPR
      |  …
```

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        var = var + const;
    }
}
```

Context-Free Languages

- A language L is called a ***context-free language*** (or CFL) if there is a CFG G such that $L = \mathscr{L}(G)$.
- Questions:
  - What languages are context-free?
  - How are context-free and regular languages related?

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$$\mathbf{S} \rightarrow \texttt{a*b}$$

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$$\mathbf{S} \rightarrow \mathbf{A}\mathtt{b}$$

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$$\mathbf{S} \to \mathbf{A}\mathtt{b}$$

$$\mathbf{A} \to \mathbf{A}\mathtt{a} \mid \varepsilon$$

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

S → `a(b|c*)`

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

S → `a`**X**

X → `(b|c*)`

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$

$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathtt{c*}$$

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathbf{C}$$

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathbf{C}$$
$$\mathbf{C} \rightarrow \mathbf{C}\mathtt{c} \mid \varepsilon$$

Regular Languages and CFLs

- ***Theorem:*** Every regular language is context-free.
- ***Proof Idea:*** Use the construction from the previous slides to convert a regular expression for L into a CFG for L. ■

The Language of a Grammar

- Consider the following CFG G:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

The Language of a Grammar

- Consider the following CFG G:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

$$\boxed{\mathbf{S}}$$

The Language of a Grammar

- Consider the following CFG G:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	S	b

The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

$\boxed{\mathtt{a}}$ $\boxed{\mathbf{S}}$ $\boxed{\mathtt{b}}$

The Language of a Grammar

- Consider the following CFG G:

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	**a**	**S**	**b**	**b**

The Language of a Grammar

- Consider the following CFG G:

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	**a**	**S**	**b**	**b**

The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	a	a	S	b	b	b

The Language of a Grammar

- Consider the following CFG G:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	a	a		S		b	b	b

The Language of a Grammar

- Consider the following CFG *G*:

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	a	a	a	S	b	b	b	b

The Language of a Grammar

- Consider the following CFG G:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	a	a	a		b	b	b	b

The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	a	a	a	b	b	b	b

The Language of a Grammar

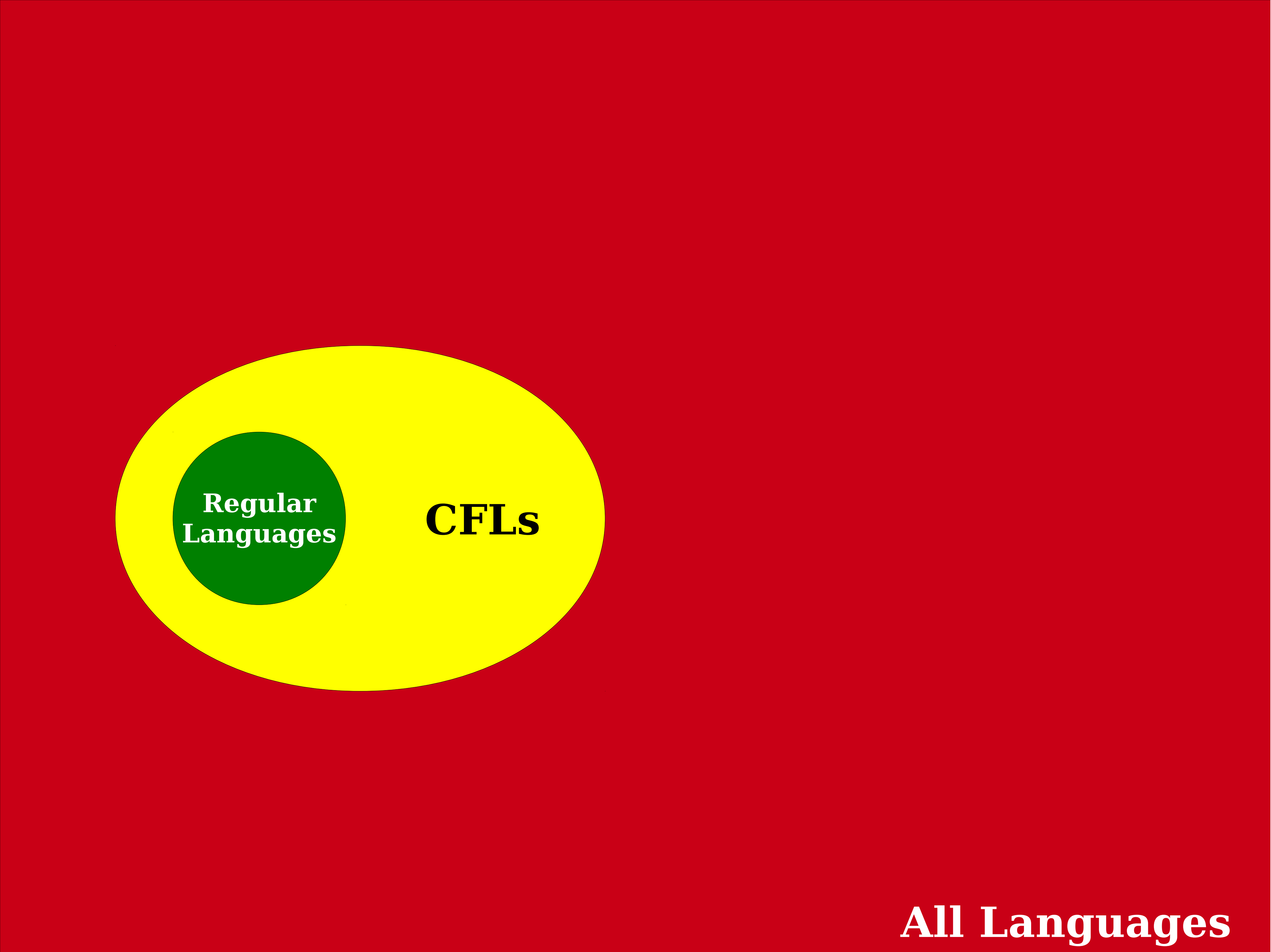
- Consider the following CFG G:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	a	a	a	b	b	b	b

$$\mathscr{L}(G) = \{\ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\ \}$$


Regular
Languages
CFLs
All Languages

Time-Out for Announcements!

Problem Set Six

- Problem Set Six goes out right now, is due next Monday at 2:15PM.
  - Explore context-free languages and the limits of regular languages!
- Problem Set Five was due at the start of class; is due on Wednesday at 2:15PM sharp with a late day.

Extra Practice Problems

- Based on your feedback, I've assembled a set of eight extra practice problems on the material so far.
- Available on the course website; solutions will go out later this week.

Midterm Regrades

- All midterm regrades were due at the start of today's lecture.
- Brought a regrade to class? Feel free to hand it in to Keith right after class today. That's okay with us.
- SCPD students – deadline is this Wednesday at the start of class.

Asking Questions

- We're getting a lot of great questions through the staff list and Piazza – please keep asking them!
- If you have Piazza questions that don't give away key insights, consider making them public. There are a lot of great private questions right now!
- Please avoid asking questions after 10:00PM the night before a problem set comes due – we will tend not to answer questions asked after this point.

Second Midterm Exam

- The second midterm exam is next Thursday, November 13 from 7PM – 10PM.
  - Covers material up through and including topics from PS6; focus is on topics from PS4 – PS6.
  - Same format as last time: closed-book, closed-computer, open one page of notes.
  - Location information TBA.
- Practice midterm next Monday night, November 10, from 7PM – 10PM, location TBA.
- Need to take the exam at an alternate time? Email Maesen ASAP to set up an alternate exam time.

Your Questions!

“It has been over a year since I took a CS class. I want to do CS107, but I am worried that my coding is too rusty. What are some good ways to brush up on coding over winter break?”

“Why is HTML not a regular language? I have always heard not to parse HTML with regular expressions because it is not a regular language, but why is this?”

“Do you have a dog?”

Back to CS103!

Designing CFGs

- Like designing DFAs, NFAs, and regular expressions, designing CFGs is a craft.
- When thinking about CFGs:
  - **Think recursively:** Build up bigger structures from smaller ones.
  - **Have a construction plan:** Know in what order you will build up the string.

Designing CFGs

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid w$ is a palindrome $\}$
- We can design a CFG for L by thinking inductively:
  - Base case: ε, **a**, and **b** are palindromes.
  - If ω is a palindrome, then **a**ω**a** and **b**ω**b** are palindromes.

$$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$$

Designing CFGs

- Let $\Sigma = \{$ **(**, **)** $\}$ and let $L = \{w \in \Sigma^* \mid w$ is a string of balanced parentheses $\}$
- We can think about how we will build strings in this language as follows:
  - The empty string is balanced.
  - Any two strings of balanced parentheses can be concatenated.
  - Any string of balanced parentheses can be parenthesized.

$$\mathbf{S} \rightarrow \mathbf{SS} \mid (\mathbf{S}) \mid \varepsilon$$

Designing CFGs: Watch Out!

- When designing CFGs, remember that each nonterminal can be expanded out independently of the others.
- Let $\Sigma = \{$**a**, $\stackrel{?}{=}\}$ and let $L = \{$ **a**$^n\stackrel{?}{=}$**a**$^n \mid n \in \mathbb{N}\}$. Is the following a CFG for L?

$$\mathbf{S} \rightarrow \mathbf{X}\stackrel{?}{=}\mathbf{X}$$

$$\mathbf{X} \rightarrow \mathtt{a}\mathbf{X} \mid \varepsilon$$

$$\begin{aligned}
&\mathbf{S}\\
\Rightarrow\ &\mathbf{X}\stackrel{?}{=}\mathbf{X}\\
\Rightarrow\ &\mathtt{a}\mathbf{X}\stackrel{?}{=}\mathbf{X}\\
\Rightarrow\ &\mathtt{aa}\mathbf{X}\stackrel{?}{=}\mathbf{X}\\
\Rightarrow\ &\mathtt{aa}\stackrel{?}{=}\mathbf{X}\\
\Rightarrow\ &\mathtt{aa}\stackrel{?}{=}\mathtt{a}\mathbf{X}\\
\Rightarrow\ &\mathtt{aa}\stackrel{?}{=}\mathtt{a}
\end{aligned}$$

Finding a Build Order

- Let $\Sigma = \{\texttt{a}, \stackrel{?}{=}\}$ and let $L = \{\texttt{a}^n\stackrel{?}{=}\texttt{a}^n \mid n \in \mathbb{N}\}$.
- To build a CFG for L, we need to be more clever with how we construct the string.
- **Idea:** Build from the ends inward.
- Gives this grammar: $\mathbf{S} \to \texttt{a}\mathbf{S}\texttt{a} \mid \stackrel{?}{=}$

$$
\begin{aligned}
&\mathbf{S} \\
\Rightarrow\ &\texttt{a}\mathbf{S}\texttt{a} \\
\Rightarrow\ &\texttt{aa}\mathbf{S}\texttt{aa} \\
\Rightarrow\ &\texttt{aaa}\mathbf{S}\texttt{aaa} \\
\Rightarrow\ &\texttt{aaa}\stackrel{?}{=}\texttt{aaa}
\end{aligned}
$$

Designing CFGs: A Caveat

- Let $\Sigma = \{$**`l`**, **`r`**$\}$ and let $L = \{ w \in \Sigma^* \mid w$ has the same number of **`l`**'s and **`r`**'s $\}$
- Is this a grammar for L?

$$\mathbf{S} \rightarrow \mathtt{l}\mathbf{S}\mathtt{r} \mid \mathtt{r}\mathbf{S}\mathtt{l} \mid \varepsilon$$

- Can you derive the string **`lrrl`**?

Designing CFGs: A Caveat

- When designing a CFG for a language, make sure that it
  - generates all the strings in the language and
  - never generates a string outside the language.
- The first of these can be tricky – make sure to test your grammars!
- You'll design your own CFG for this language on the next problem set.

CFG Caveats II

- Is the following grammar a CFG for the language $\{ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N} \}$?

$$\mathbf{S} \to \mathtt{a}\mathbf{S}\mathtt{b}$$

- What strings can you derive?
  - Answer: **None!**
- What is the language of the grammar?
  - Answer: **Ø**
- When designing CFGs, make sure your recursion actually terminates!

Function Prototypes

- Let Σ = {**void**, **int**, **double**, **name**, **(**, **)**, **,**, **;**}.
- Let's write a CFG for C-style function prototypes!
- Examples:
  - `void name(int name, double name);`
  - `int name();`
  - `int name(double name);`
  - `int name(int, int name, int);`
  - `void name(void);`

Function Prototypes

- Here's one possible grammar:
  - **S** → **Ret** `name` `(`**Args**`)` `;`
  - **Ret** → **Type** | `void`
  - **Type** → `int` | `double`
  - **Args** → ε | `void` | **ArgList**
  - **ArgList** → **OneArg** | **ArgList**, **OneArg**
  - **OneArg** → **Type** | **Type** `name`

Summary of CFG Design

- Look for recursive structures where they exist – it can help guide you toward a solution.
- Keep the build order in mind – often, you'll build two totally different parts of the string concurrently.
- Use different nonterminals to represent different structures.

Next Time

- **Turing Machines**
  - What does a computer with unbounded memory look like?
  - How do you program them?