Announcements

- Project 2 Assigned
  - JLex for C Flat
  - Find a partner if you want
- Reminder: Homework 2
  - Due 9/23 (Tuesday)

Roadmap

- Last time
  - JLex for generating Lexers
- This time
  - CFGs, the underlying abstraction for Parsers

RegExs Are Great!

- Perfect for tokenizing a language
- They do have some limitations
  - Limited class of language that cannot specify all programming constructs we need
  - No notion of structure
- Let's explore both of these issues

Limitations of RegExs

- Cannot handle “matching”
  - Eg: language of balanced parentheses

    L = **{** $(^x)^x$ where x > 1**}**

    cannot be matched
  - Intuition:

    An FSM can only handle a finite depth of parentheses that we can handle

    let’s see a diagram…

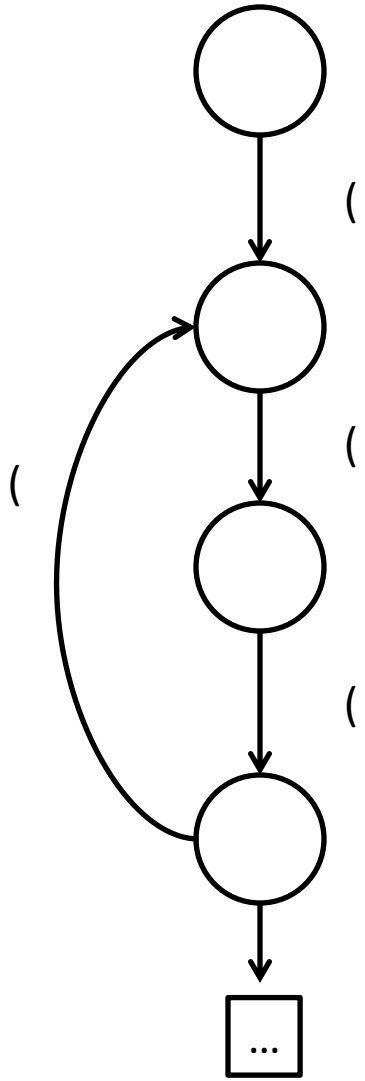
Limitations of RegExs: Balanced Parens

Assume F is an FSM that recognized L. Let N be the number of states in F'.

Feed N+1 left parens into N

By the pidgeonhole principle, we must have revisited some state s on two input characters i and j.

By the definition of F, there must be a path from s to a final state. But this means that it accepts some suffix of closed parens at input i and j, but both cannot be correct



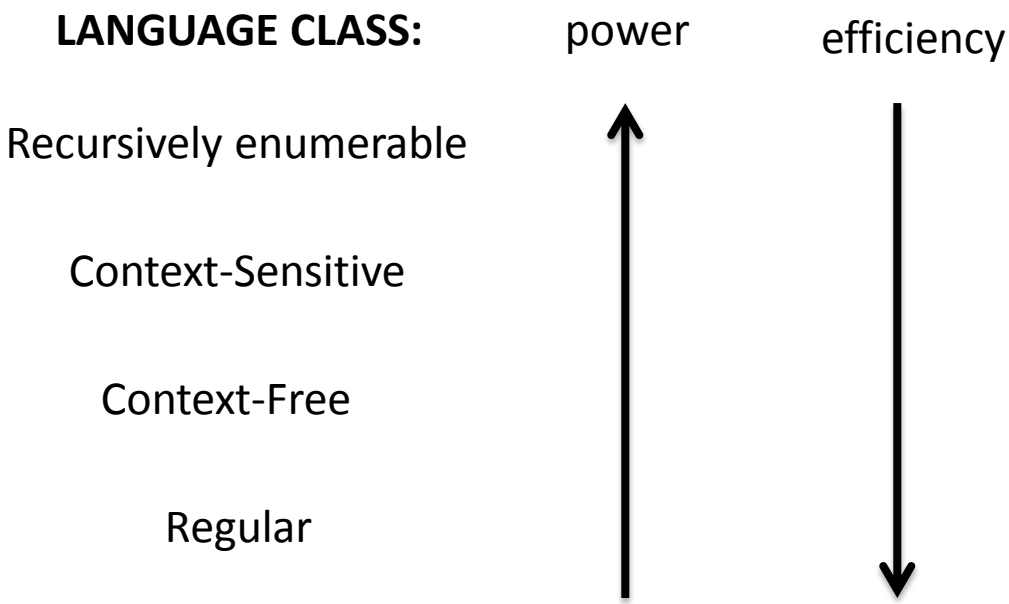
Limitations of RegEx: Structure

- Our Enhanced-RegEx scanner can emit a stream of tokens:

  X = Y + Z

  … but this doesn't really enforce any order of operations

The Chomsky Hierarchy



Context Free Grammars (CFGs)

- A set of (recursive) rewriting rules to generate patterns of strings
- Can envision a “parse tree” that keeps structure

CFG: Intuition

$S \rightarrow$ **(** S **)**

Before applying rule **After applying rule**

Context Free Grammars (CFGs)

- Formally, a 4-tuple:
  - N is the set of nonterminal symbols
  - $\sum$ is the set of terminal symbols
  - P is the set of productions
  - S is the start nonterminal in N

Production Syntax

LHS → RHS

Production Shorthand

Nonterm → expression

Nonterm→ ε

equivalently:

Nonterm → expression

| ε

equivalently:

Nonterm → expression | ε

Derivations

- To derive a string:
  - Start by setting "*Current Sequence*" to the start symbol
  - Repeat:
    - Find a Nonterminal X in the Current Sequence
    - Find a production of the form $X \rightarrow \alpha$
    - "Apply" the production: create a new "current sequence" in which α replaces X
  - Stop when there are no more nonterminals

Derivation Syntax

- We'll use the symbol $\Rightarrow$ for *derives*
- We'll use the symbol $\overset{+}{\Rightarrow}$ for *derives in one or more steps*
- We'll use the symbol $\overset{*}{\Rightarrow}$ for *derives in zero or more steps*

An Example Grammar

Terminals
begin
end
semicolon
assign
id
plus

Nonterminals
Prog
Stmts
Stmt
Expr

Productions

Prog → **begin** Stmts **end**

Stmts → Stmts **semicolon** Stmt

| Stmt

Stmt → **id assign** Expr

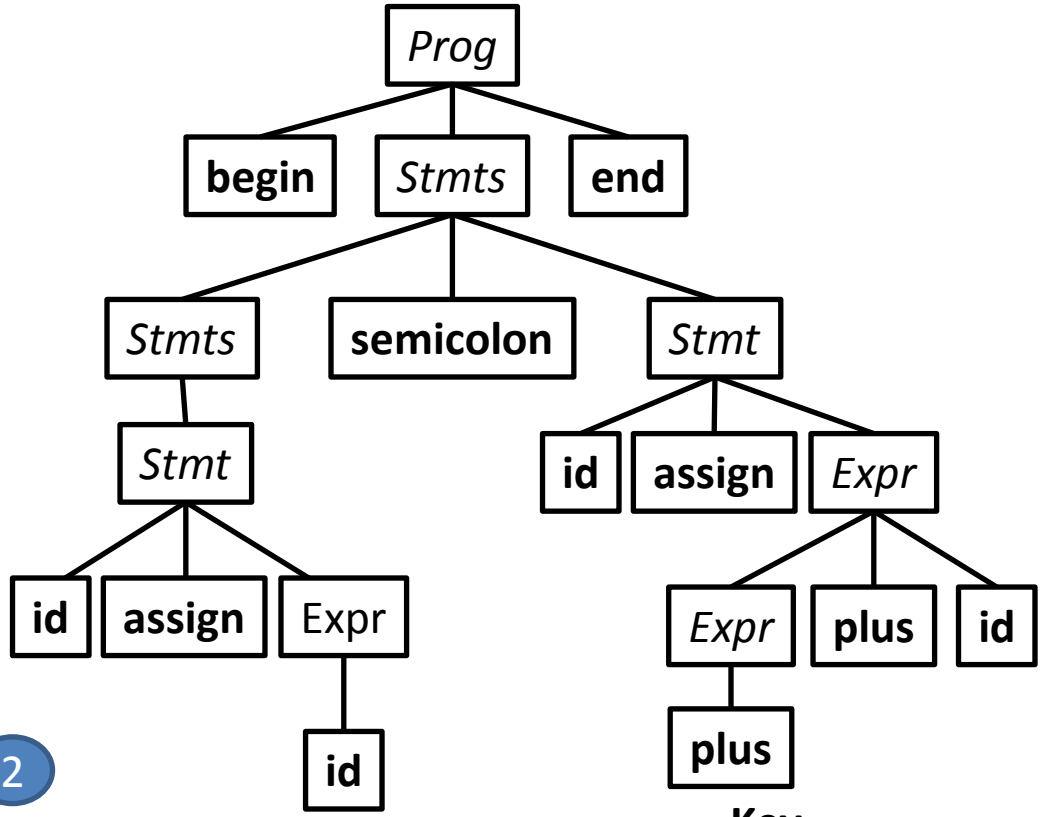
Expr → **id**

| Expr **plus id**

Productions

1. *Prog* → **begin** *Stmts* **end**
2. Stmts → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



Derivation Sequence

Prog ⇒ **begin** Stmts **end** (1)

⇒ **begin** Stmts **semicolon** Stmt **end** (2)

⇒ **begin** Stmt **semicolon** Stmt **end** (3)

⇒ **begin id assign** Expr **semicolon** Stmt **end** (4)

⇒ **begin id assign** Expr **semicolon id assign** Expr **end** (4)

⇒ **begin id assign id semicolon id assign** Expr **end** (5)

⇒ **begin id assign id semicolon id assign** Expr **plus id end** (6)

⇒ **begin id assign id semicolon id assign id plus id end** (5)

Key

terminal

Nonterminal

Rule used

Makefiles: Motivation

- Typing the series of commands to generate our code can be tedious
  - Multiple steps that depend on each other
  - Somewhat complicated commands
  - May not need to rebuild everything
- Makefiles solve these issues
  - Record a series of commands in a script-like DSL
  - Specify dependency rules and Make generates the results

Makefiles: Basic Structure

<target>: <dependency list>

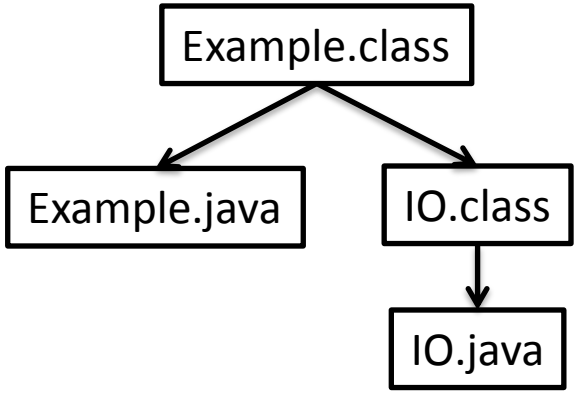
(tab) <command to satisfy target>

Example

```
Example.class: Example.java IO.class
     javac Example.java

IO.class: IO.java
     javac IO.java
```

Makefiles: Dependencies



Example

```
Example.class: Example.java IO.class
    javac Example.java

IO.class: IO.java
    javac IO.java
```

Makefiles: Variables

You can thread common configuration values through your makefile

Example

JC = /s/std/bin/javac

JFLAGS = -g **Build for debug**

```
Example.class: Example.java IO.class
	$(JC) $(JFLAGS) Example.java

IO.class: IO.java
	$(JC) $(JFLAGS) IO.java
```

Makefiles: Phony Targets

- You can run commands through make.
  - Write a target with no dependencies (called phony)
  - Will cause it to execute the command every time

Example

```
clean:
	rm –f *.class
test:
	java –cp . Test.class
```