Context-free grammars

Roadmap

- Last time
  - Regex == DFA
  - 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…

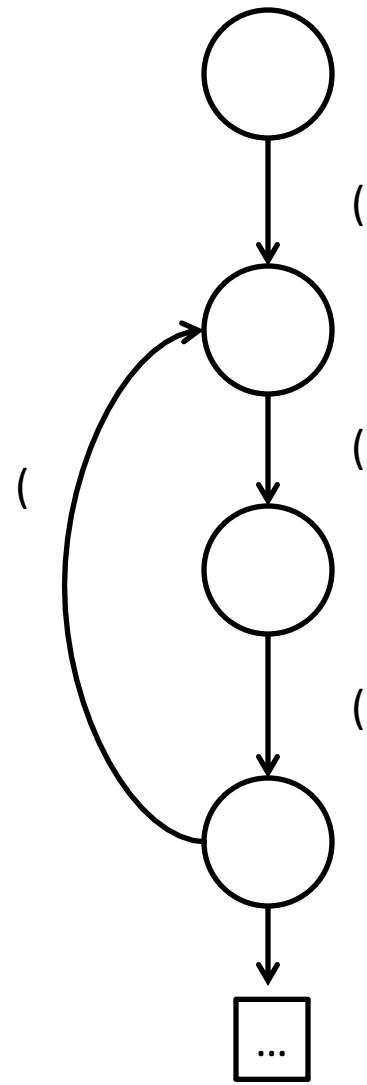
Limitations of RegExs: Balanced Parens

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 *pigeonhole* 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
↑	↑	↑	↑	↑
ID	ASSIGN	ID	PLUS	ID

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

The Chomsky Hierarchy

Turing machine

LANGUAGE CLASS: power efficiency

Recursively enumerable

Context-Sensitive

Context-Free

Happy medium?

Regular

FSM

Noam Chomsky

Context Free Grammars (CFGs)

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

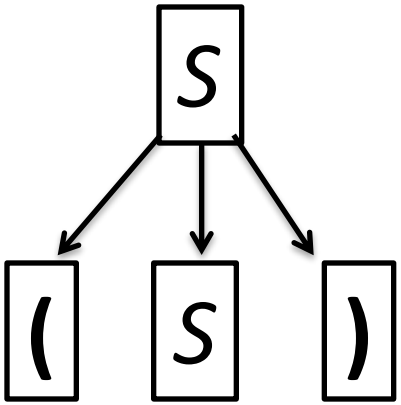
CFG: Intuition

$$\underbrace{S \rightarrow (\ S\ )}$$

A *rule* that says that you can rewrite S to be an S surrounded by a single set of parens

Before applying rule

After applying rule



CFGs recognize the language of trees where all the leaves are terminals

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

Context Free Grammars (CFGs)

占位符
Placeholder / interior nodes in the parse tree

- 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

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

Placeholder / interior nodes in the parse tree

Tokens from scanner

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

Placeholder / interior nodes in the parse tree

Tokens from scanner

Rules for deriving strings

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

Placeholder / interior nodes in the parse tree

Tokens from scanner

Rules for deriving strings

If not otherwise specified, use the non-terminal that appears on the LHS of the first production as the start

Production Syntax

LHS → RHS

Single nonterminal symbol

Expression: Sequence of terminals and nonterminals

Production Shorthand

Nonterm → expression

Sequence of terms and nonterms

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

An Example Grammar

<u>Terminals</u>
begin
end
semicolon
assign
id
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin } Program boundary
end
semicolon
assign
id
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end

Program boundary

semicolon

Represents ";"
Separates statements

assign
id
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end

Program boundary

semicolon

Represents ";"
Separates statements

assign

Represents "=" statement

id
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end

Program boundary

semicolon

Represents ";"
Separates statements

assign

Represents "=" statement

id

Identifier / variable name

plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end

Program boundary

semicolon

Represents ";"
Separates statements

assign

Represents "=" statement

id

Identifier / variable name

plus

Represents "+" expression

An Example Grammar

For readability, bold and lowercase

<u>Terminals</u>

begin
end
semicolon
assign
id
plus

<u>Nonterminals</u>

Prog
Stmts
Stmt
Expr

An Example Grammar

For readability, bold and lowercase

<u>Terminals</u>

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

<u>Nonterminals</u>

Prog
Stmts
Stmt
Expr

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

Nonterminals

Prog — Root of the parse tree
Stmts
Stmt
Expr

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

Nonterminals

Prog — Root of the parse tree
Stmts — List of statements
Stmt
Expr

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

Nonterminals

Prog — Root of the parse tree
Stmts — List of statements
Stmt — A single statement
Expr

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

Nonterminals

Prog — Root of the parse tree
Stmts — List of statements
Stmt — A single statement
Expr — A mathematical expression

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

Nonterminals

Prog
Stmts
Stmt
Expr

Defines the syntax of legal programs

Productions

Prog → **begin** Stmts **end**

Stmts → Stmts **semicolon** Stmt
| Stmt

Stmt → **id** **assign** Expr

Expr → **id**
| Expr **plus id**

An Example Grammar

For readability, bold and lowercase

Terminals

begin } Program boundary
end
semicolon — Represents ";" Separates statements
assign — Represents "=" statement
id — Identifier / variable name
plus — Represents "+" expression

For readability, Italics and UpperCamelCase

Nonterminals

Prog — Root of the parse tree
Stmts — List of statements
Stmt — A single statement
Expr — A mathematical expression

Defines the syntax of legal programs

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**

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**

Derivation Sequence

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

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

Key

terminal

Nonterminal

Rule used

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**

Derivation Sequence

Prog

Parse Tree

Prog

Key

terminal

Nonterminal

Rule used

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**

Derivation Sequence

Prog ⇒ **begin** Stmts **end** 1

Parse Tree

Prog

Key

terminal

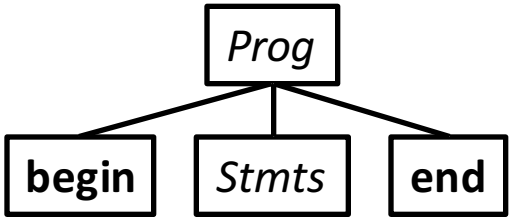
Nonterminal

Rule used

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

Key

terminal

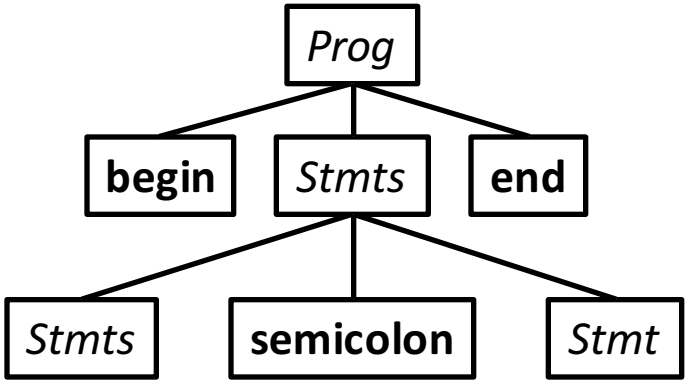
Nonterminal

Rule used

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)

Key

terminal

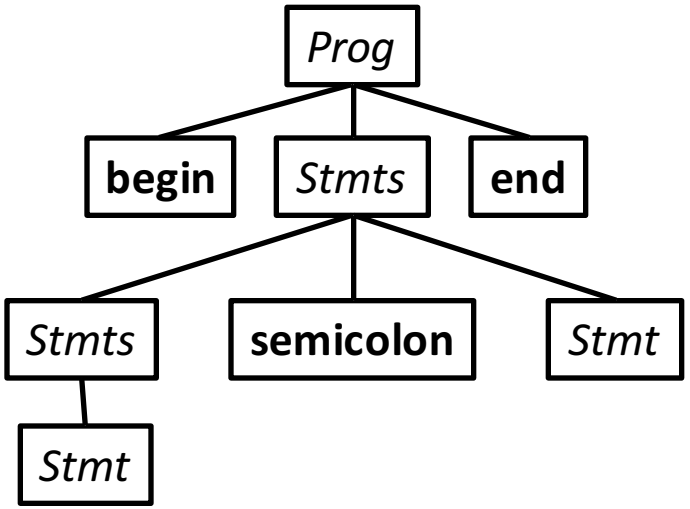
Nonterminal

Rule used

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)

Key

terminal

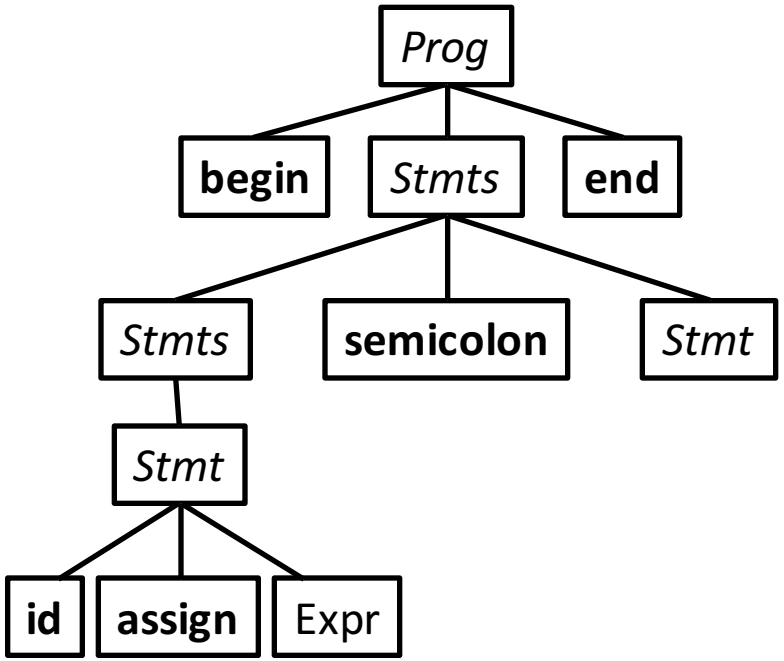
Nonterminal

Rule used

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)

Key

terminal

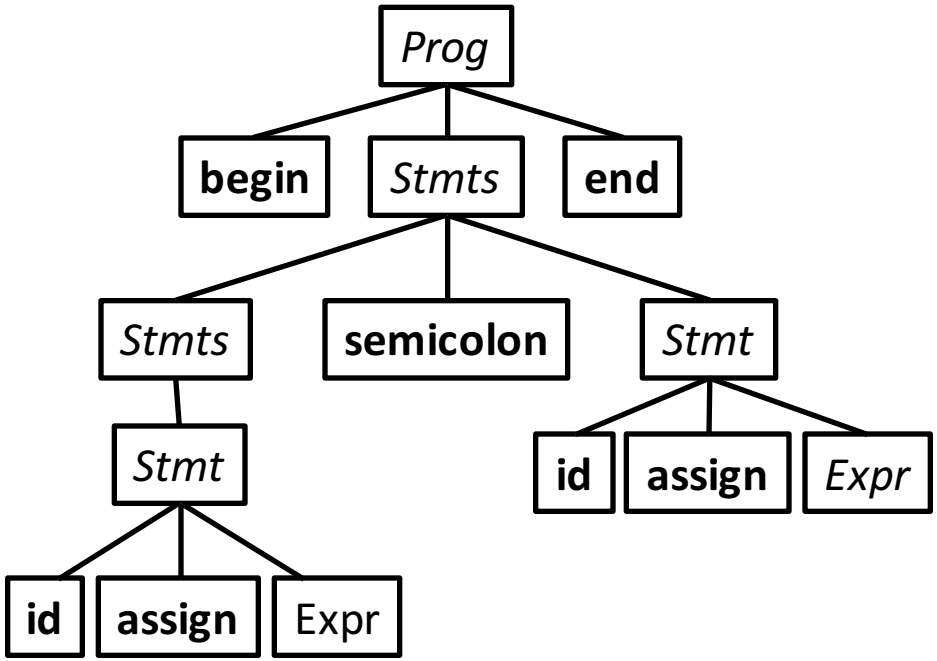
Nonterminal

Rule used

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)

Key

terminal

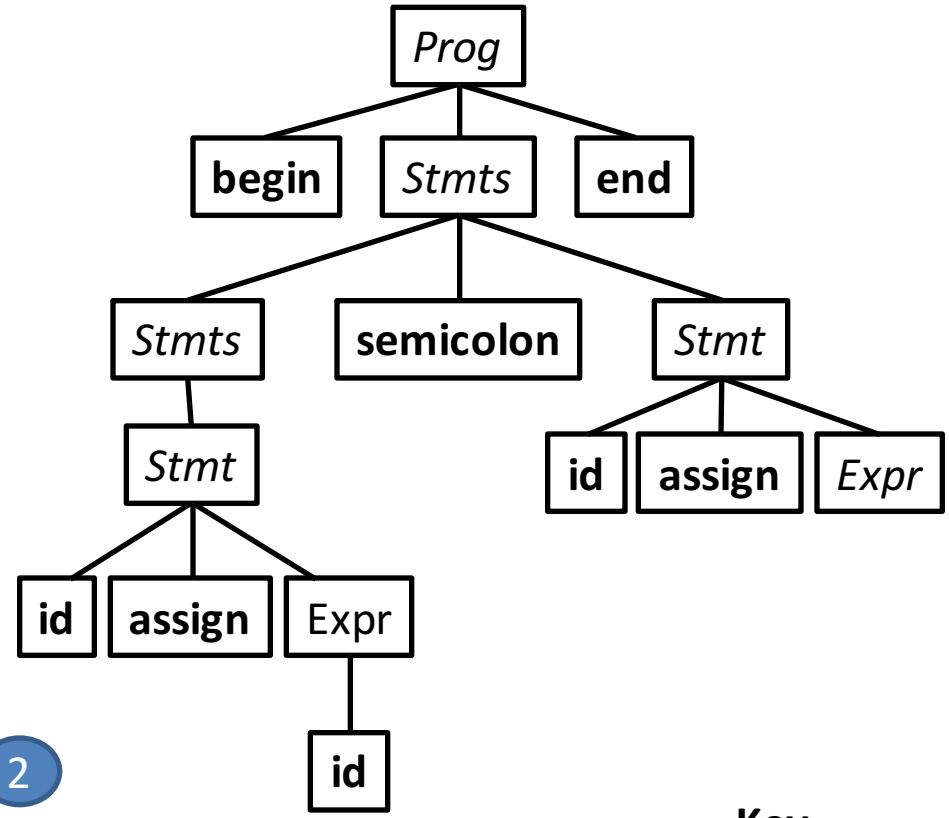
Nonterminal

Rule used

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)

Key

terminal

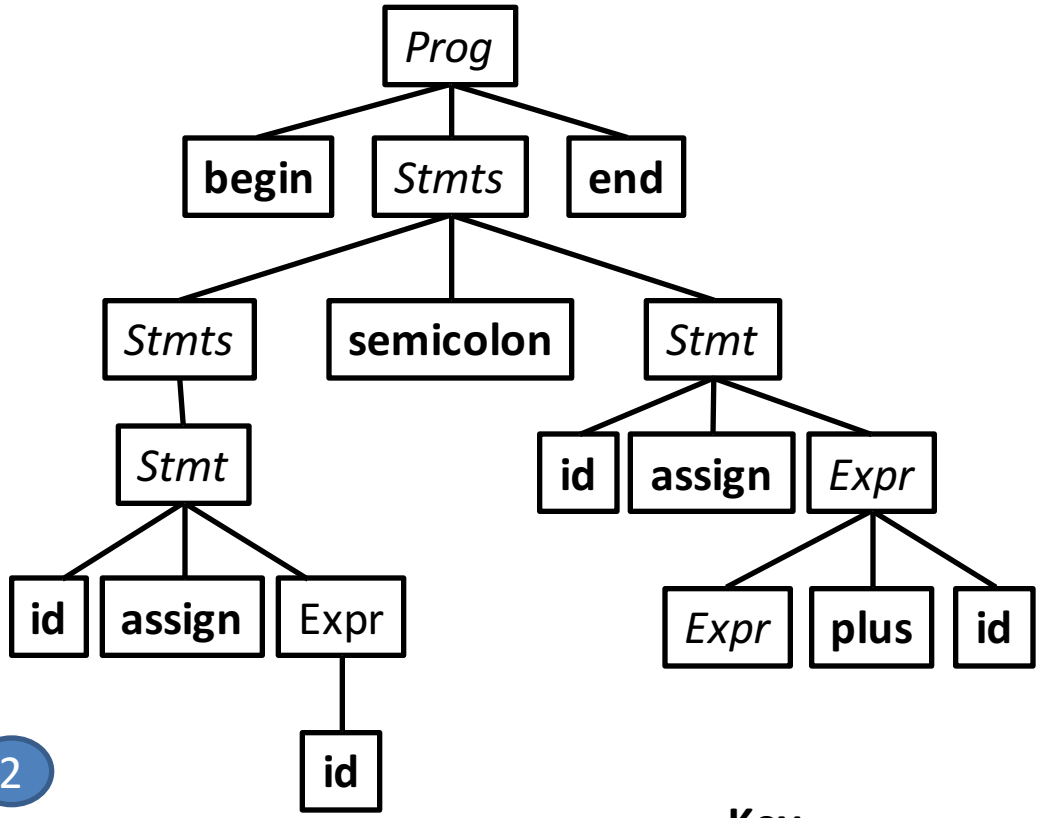
Nonterminal

Rule used

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)

Key

terminal

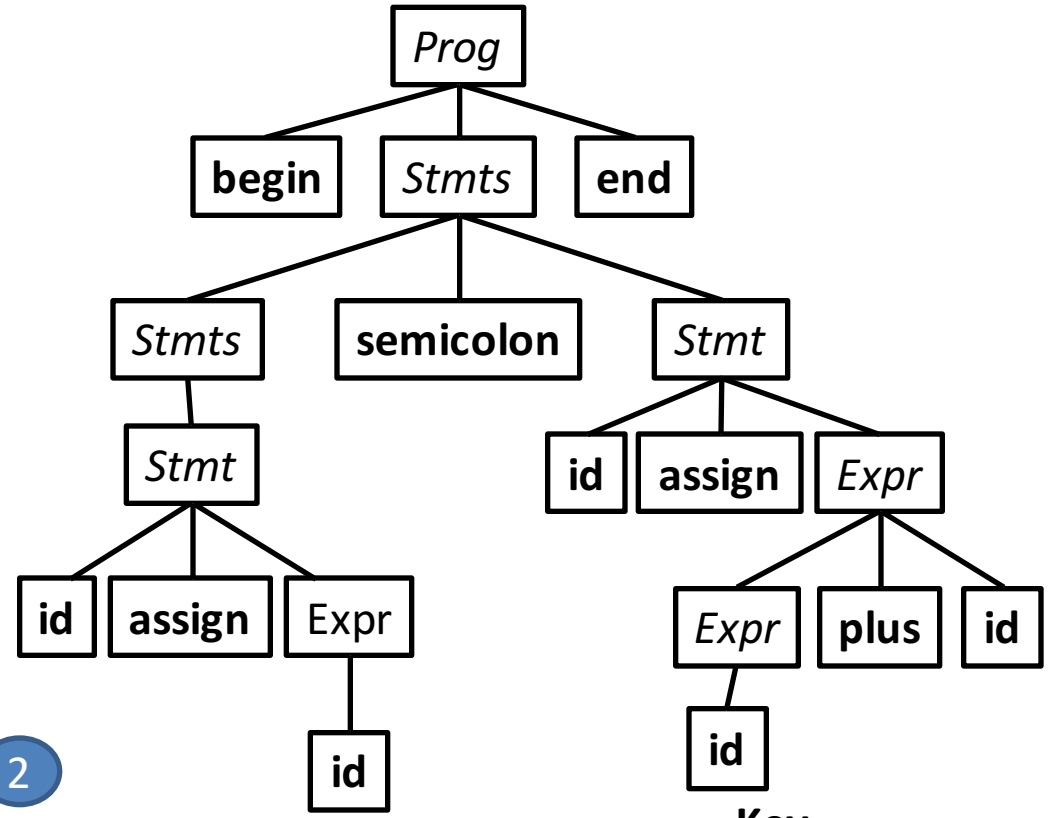
Nonterminal

Rule used

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>

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: Basic Structure

<target>: <dependency list>

(tab) <command to satisfy target>

Example

Example.class depends on example.java and IO.class

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

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

Makefiles: Basic Structure

<target>: <dependency list>

(tab) <command to satisfy target>

Example

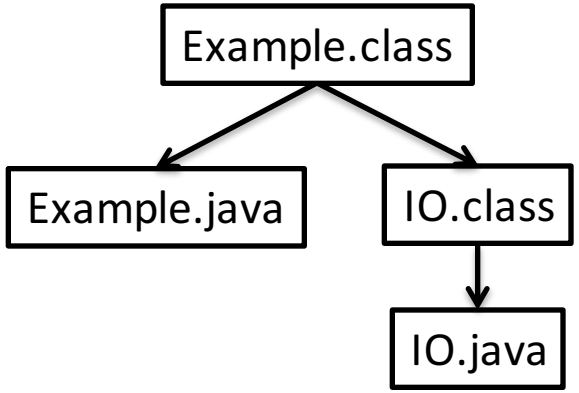
Example.class depends on example.java and IO.class

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

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

Example.class is generated by javac Example.java

Makefiles: Dependencies



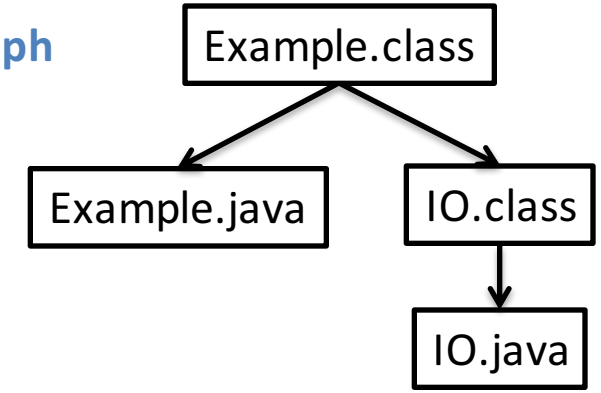
<u>Example</u>

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

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

Makefiles: Dependencies

Internal Dependency graph



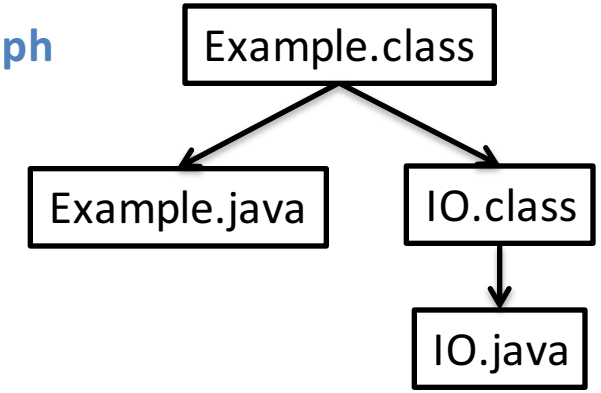
<u>Example</u>

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

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

Makefiles: Dependencies

Internal Dependency graph



A file is rebuilt if one of it's dependencies changes

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

Makefiles: Variables

You can thread common configuration values through your makefile

<u>Example</u>

JC = /s/std/bin/javac

JFLAGS = -g

Makefiles: Variables

You can thread common configuration values through your makefile

Example

JC = /s/std/bin/javac

JFLAGS = -g **Build for debug**

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

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
```

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
```

Recap

- We've defined context-free grammars
  - More powerful than regular grammars
- Submit P1
- P2 will come out tonight
- Next time we'll look at grammars in more detail