Announcements

- P2 posted
- HW2 posted

Context-free grammars (CFGs)

Roadmap

Last time

– Regex == DFA

– JLex: a tool for generating (Java code for) Lexers/ Scanners

This time

– CFGs, the underlying abstraction for Parsers

Next week

- Java CUP: A tool for generating (Java code for) parser

regular expression ==> scanner, JLex

CFG ==> parser, Java CUP

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

RegExs are great for scanner and tokenization, but not sufficient for parsing.

Limitations of RegExps

Regexp and DFAs are equivalent.
For every regexp, there's an equivalent DFA that represents the same language, and vice versa.

Cannot handle “matching”

E.g., language of balanced parentheses

$L_{()} = \{ (^n)^n \text{ where } n > 0\}$

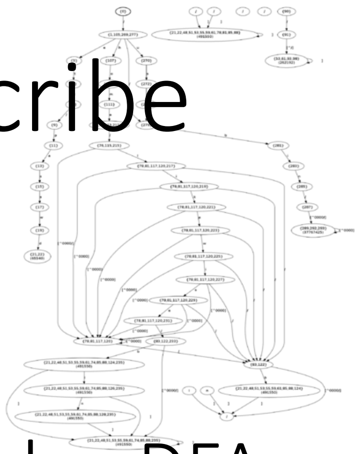
No DFA exists for this language

Intuition: A given FSM only has a fixed, finite amount of memory

- For an FSM, memory = the states
- With a fixed, finite amount of memory, how could an FSM remember how many “(“ characters it has seen?

Theorem: No RegEx/DFA can describe the language $L_{()}$

Can be asked in the exam.



By contradiction:

- Let's say there exists a DFA A for $L_{()}$ and such a DFA has **N** states
- A has to accept the string **$(^N\ )^N$** with some path **$q_0q_1 \ldots q_N \ldots q_{2N}$**

q0 is the start state, so q1...q2N takes 2N characters in total.

- By *pigeonhole principle* some state has repeated: $q_i = q_j$ for some i<j<=N
- Therefore the run **$q_0q_1 \ldots q_iq_{j+1} \ldots q_N \ldots q_{2N}$** is also accepting
- A accepts the string **$(^{N-(j-i)}\ )^N$** **not in** $L_{()}$ → contradiction!

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

Regular expression cannot indicate the precedence or associativity of the operators.

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

To build a tree, you need recursion.

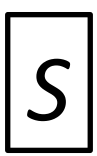
Can envision a "parse tree" that keeps structure

CFG: Intuition

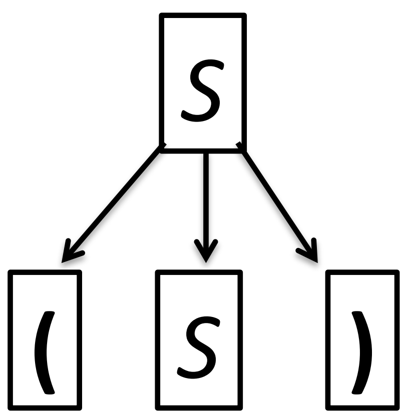
$$S \rightarrow \text{‘(‘}\ S\ \text{‘)’}$$

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

Before applying rule



After applying rule



Context Free Grammars (CFGs)

A CFG is a 4-tuple (N,Σ,P,S)

- N is a set of non-terminals, e.g., A, B, S…
- Σ is the set of terminals
- P is a set of production rules
- S (in N) is the initial non-terminal symbol

Context Free Grammars (CFGs)

A CFG is a 4-tuple (N,Σ,P,S)

- N is a set of non-terminals, e.g., A, B, S…
- Σ is the set of terminals
- P is a set of production rules
- S (in N) is the initial non-terminal symbol

Placeholder / interior nodes in the parse tree

Leaf of the 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

Compare math representation of FSM and CFG:
FSM = (Q, Σ, δ, q, F)
Q: finite set of states, Σ: alphabets, δ: transition functions, q: start state, F: final states.

CFG = (N, Σ, P, S)
N: non-terminals, Σ: terminals, P: production rules, S: initial non-terminal symbol.

Production Syntax

LHS → RHS

Single nonterminal symbol

Expression: Sequence of terminals and nonterminals

Examples:

S → '(' S ')'

S → ε

Production Shorthand

Nonterm → expression

Nonterm→ ε

equivalently:

Nonterm → expression

| ε

equivalently:

Nonterm → expression | ε

S → '(' S ')'

S → ε

S → '(' S ')'

| ε

S → '(' S ')' | ε

Derivations

Now we are generating the syntax tree by randomly picking a production rule.

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 non-terminals
- This process derives a string of terminal symbols

Derivation Syntax

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

<u>**Terminals**</u>

begin Program
end boundary
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

<u>**Terminals**</u>

begin
end

Program boundary

semicolon

Represents “;”
Separates statements

assign

Represents “=“ statement

id
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

Program boundary

Represents “;”
Separates statements

Represents “=“ statement

Identifier / variable name

An Example Grammar

For readability, bold and lowercase

Terminals

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

An Example Grammar

For readability, bold and lowercase

Terminals

begin

end

semicolon

assign

id

plus

Nonterminals

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

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

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**

For readability, Italics and UpperCamelCase

Nonterminals
Prog
Stmts
Stmt
Expr

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

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**

For readability, Italics and UpperCamelCase

Nonterminals

Prog — Root of the parse tree

Stmts — List of statements

Stmt — A single statement

Expr — A mathematical expression

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**

Parse Tree

Prog

Derivation Sequence

Prog ⇒⊥ **begin** Stmts **end** 1

Key

terminal

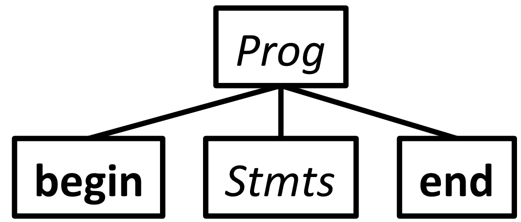
Nonterminal

Rule used

Productions

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

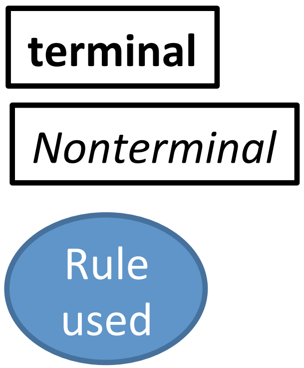
Parse Tree



Derivation Sequence

Prog $\Rightarrow^{\perp}$ **begin** Stmts **end** 1

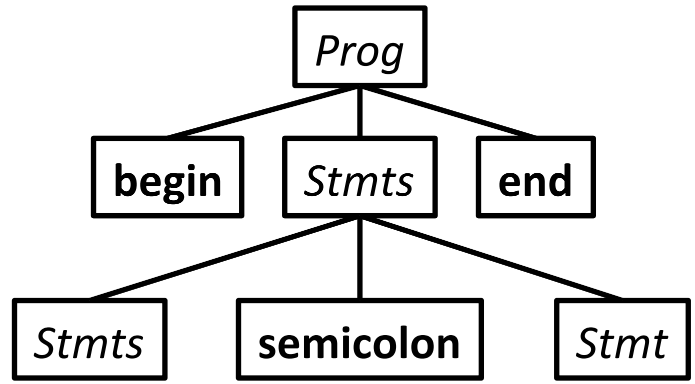
Key



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

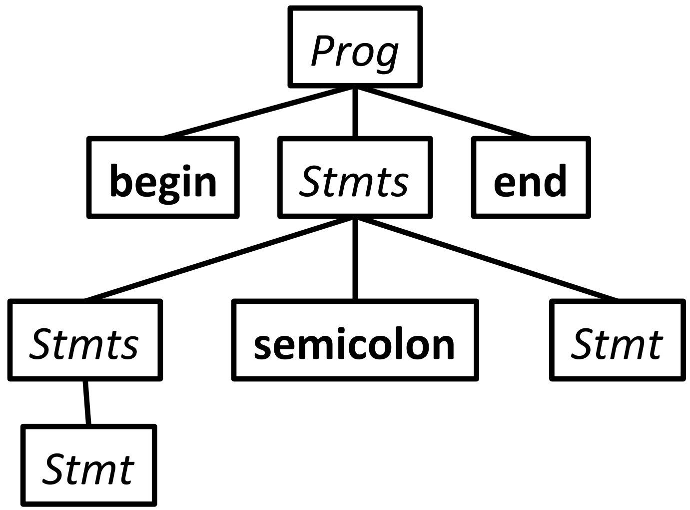
Nonterminal

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



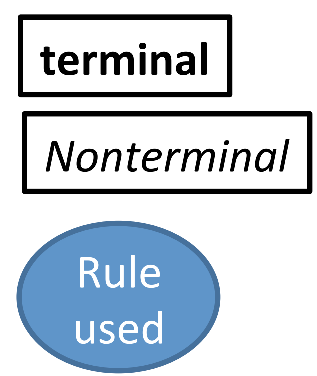
Derivation Sequence

Prog ⇒⊥ **begin** Stmts **end** 1

⇒⊥ **begin** Stmts **semicolon** Stmt **end** 2

⇒⊥ **begin** Stmt **semicolon** Stmt **end** 3

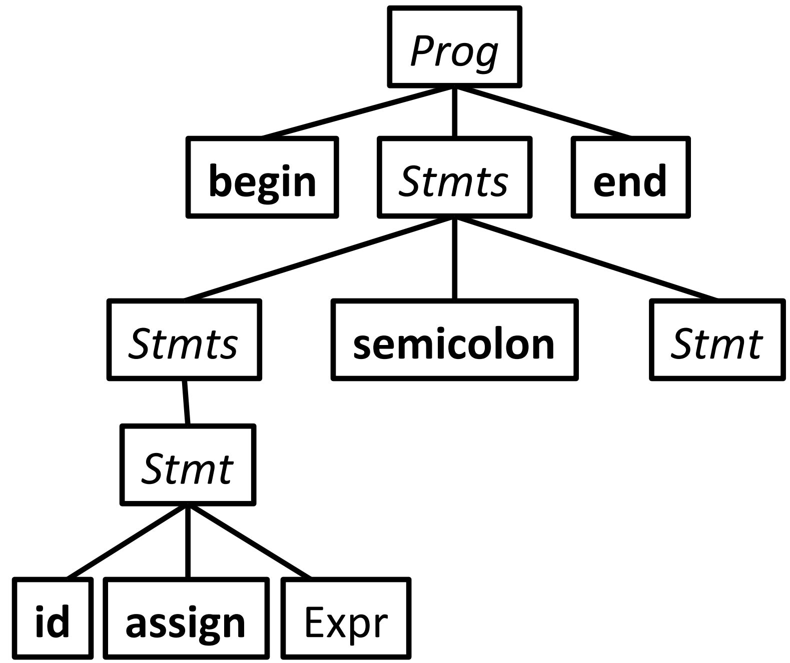
Key



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

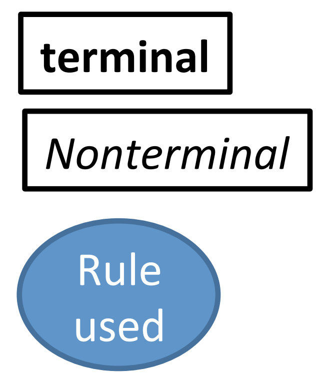
Prog ⇒⊥ **begin** Stmts **end** 1

⇒⊥ **begin** Stmts **semicolon** Stmt **end** 2

⇒⊥ **begin** Stmt **semicolon** Stmt **end** 3

⇒⊥ **begin id assign** Expr **semicolon** Stmt **end** 4

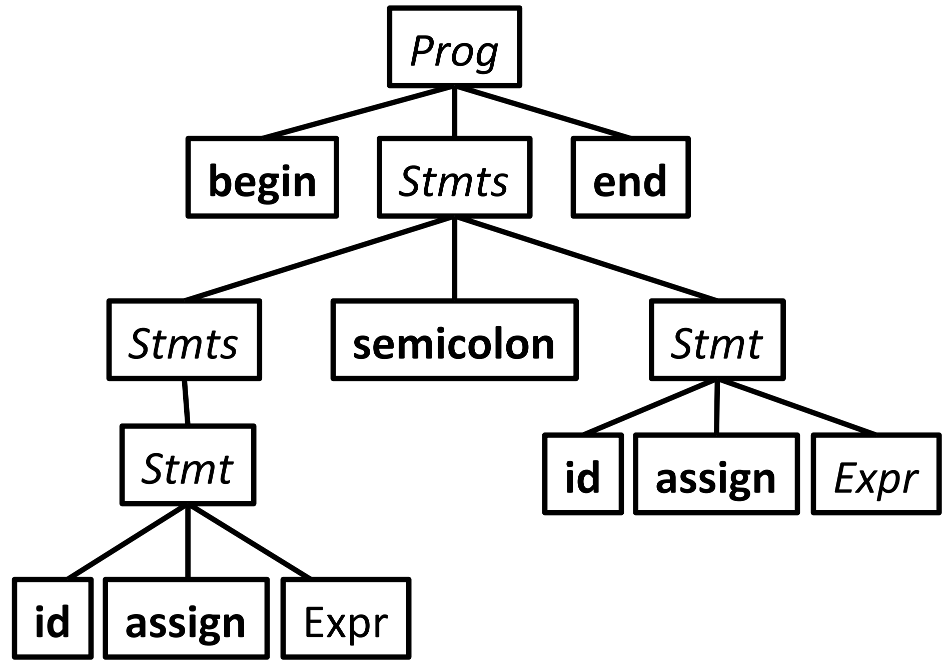
Key



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 $\Rightarrow\perp$ **begin** Stmts **end** 1

$\Rightarrow\perp$ **begin** Stmts **semicolon** Stmt **end** 2

$\Rightarrow\perp$ **begin** Stmt **semicolon** Stmt **end** 3

$\Rightarrow\perp$ **begin id assign** Expr **semicolon** Stmt **end** 4

$\Rightarrow\perp$ **begin id assign** Expr **semicolon id assign** Expr **end** 4

Key

terminal

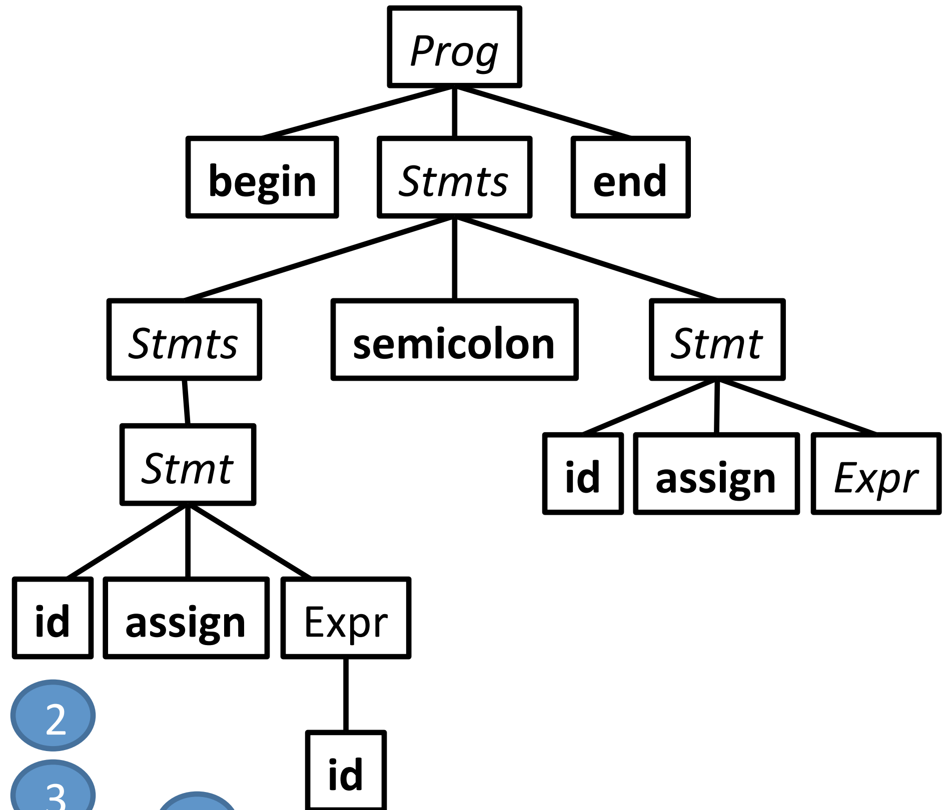
Nonterminal

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

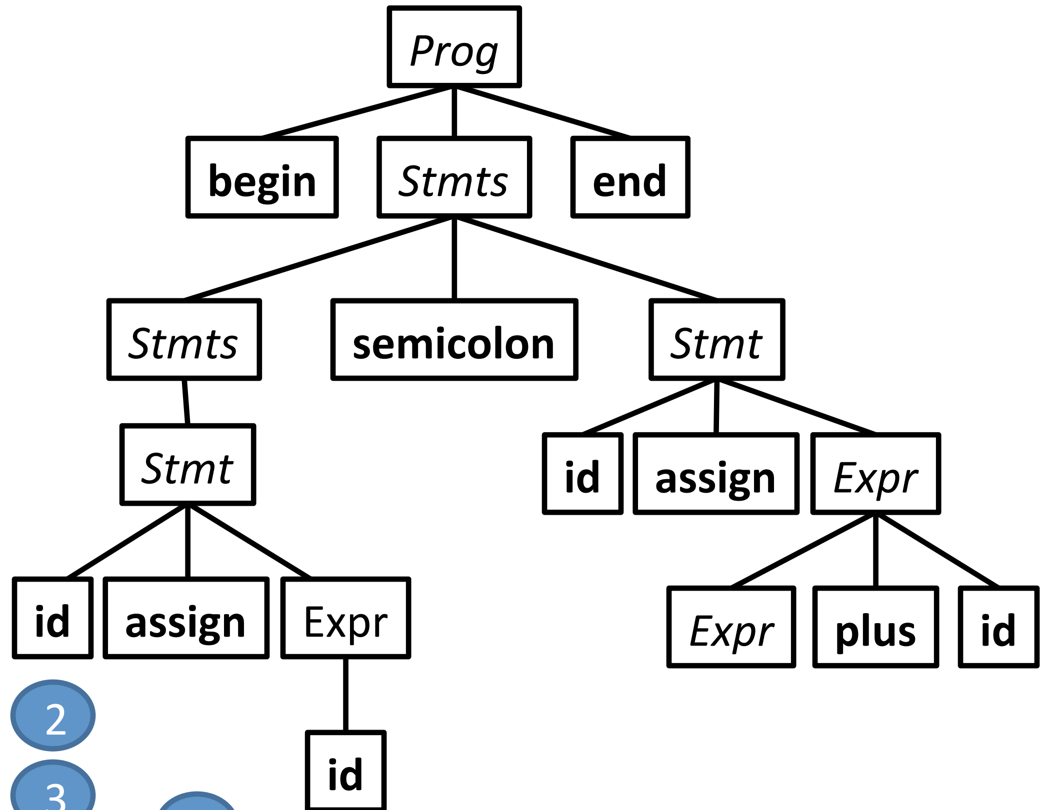
Nonterminal

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

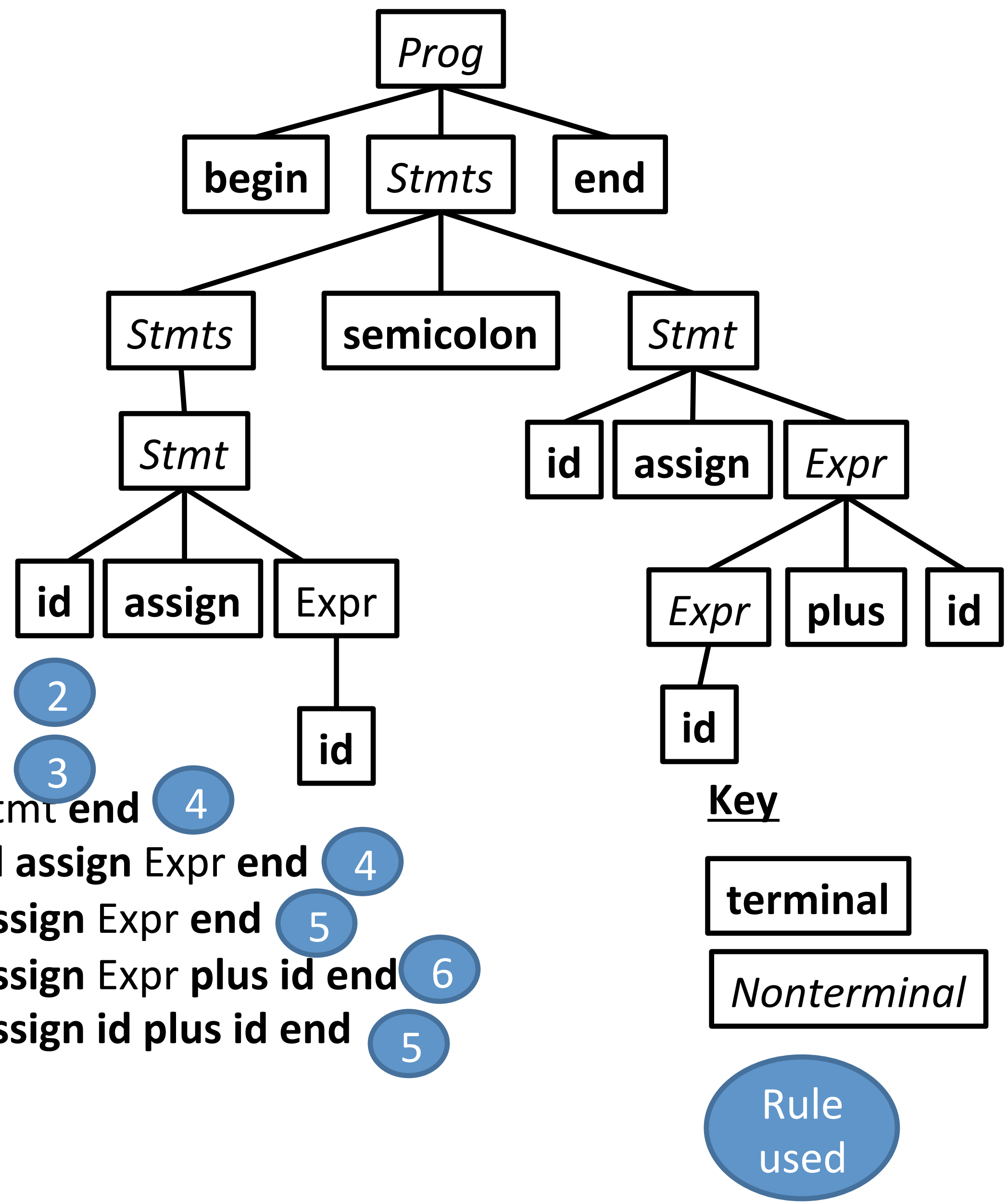
Nonterminal

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

One of the sequences accepted by the CFG.

A five minute introduction

MAKEFILE

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>

<u>Example</u>

```
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>

<u>Example</u>

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>

<u>Example</u>

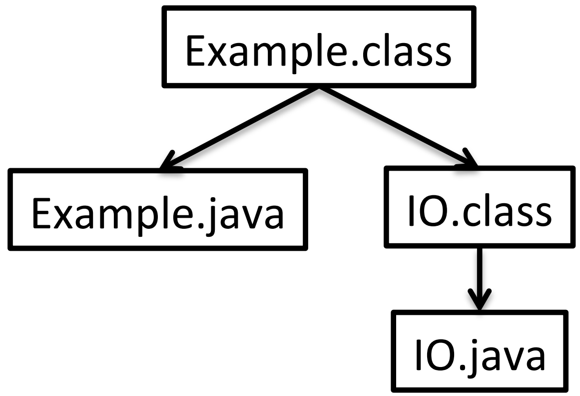
Example.class depends on example.java and IO.class

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

Example.class is generated by javac Example.java

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

Makefiles: Dependencies



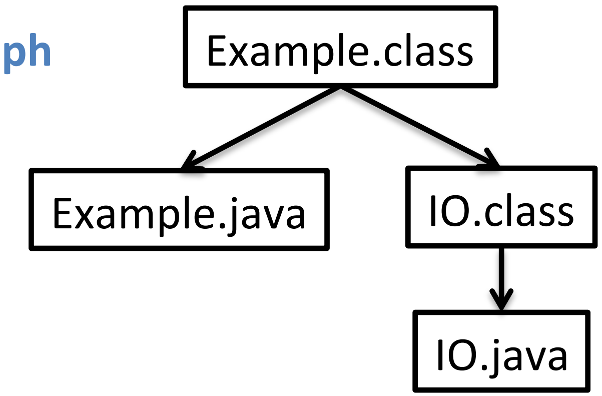
Example

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

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

Makefiles: Dependencies

Internal Dependency graph



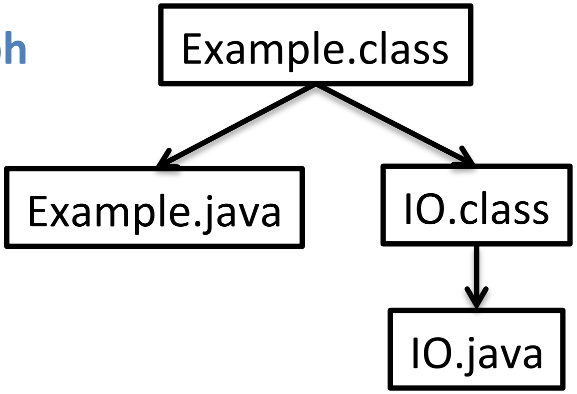
<u>Example</u>

```
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

<u>Example</u>

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 expressions
- Learned a bit about makefile
- Next time we'll look at grammars in more detail