Context-free grammars (CFGs)

Roadmap

Last time
- RegExp == DFA
- Jlex: a tool for generating (Java code for) a lexer/scanner
  - Mainly a collection of ⟨regexp, action⟩ pairs

This time
- CFGs, the underlying abstraction for parsers

Next week
- Java CUP: a tool for generating (Java code for) a parser
  - Mainly a collection of ⟨CFG-rule, action⟩ pairs

regexp : JLex :: CFG : Java CUP

RegExps Are Great!

Perfect for tokenizing a language

However, they have some limitations

- Can only define a limited family of languages
  - Cannot use a RegExp to specify all the programming constructs we need
- No notion of structure

Let's explore both of these issues

Limitations of RegExps

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?

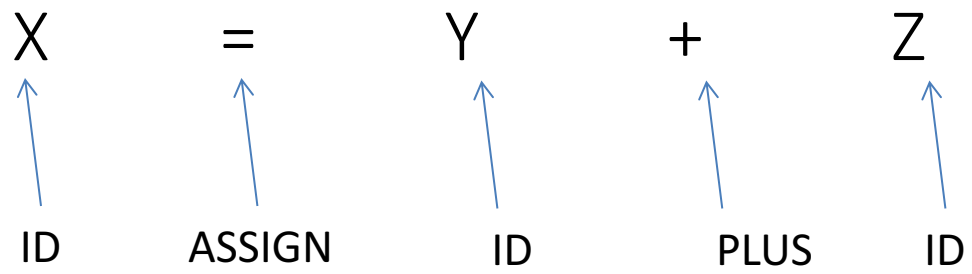
Theorem: No RegExp/DFA can describe the language $L_{()}$

Proof by contradiction:

- Suppose that there exists a DFA A for $L_{()}$ and A has **N** states
- A has to accept the string $(^N)^N$ with some path $q_0q_1…q_N…q_{2N+1}$
- By the *pigeonhole principle* some state has to repeat: $q_i = q_j$ for some $i<j<N$
- Therefore the run $q_0q_1…q_iq_{j+1}…q_N…q_{2N+1}$ is also accepting
- A accepts the string $(^{N-(j-i)})^N \notin L_{()}$, which is a contradiction!

Limitations of RegExps: No Structure

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



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

The Chomsky Hierarchy

Turing machine

LANGUAGE CLASS:

Recursively enumerable

Context-Sensitive

Context-Free

Regular

power

efficiency

Happy medium?

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

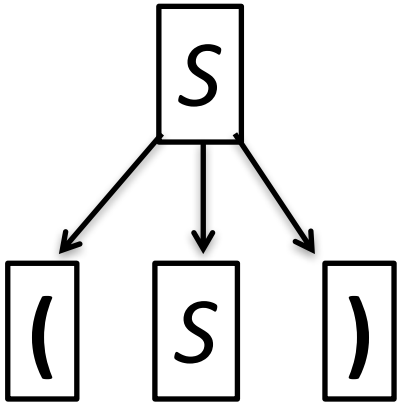
CFG: Intuition

$S \rightarrow$ '(' S ')'

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∈N is the initial non-terminal symbol (“start 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

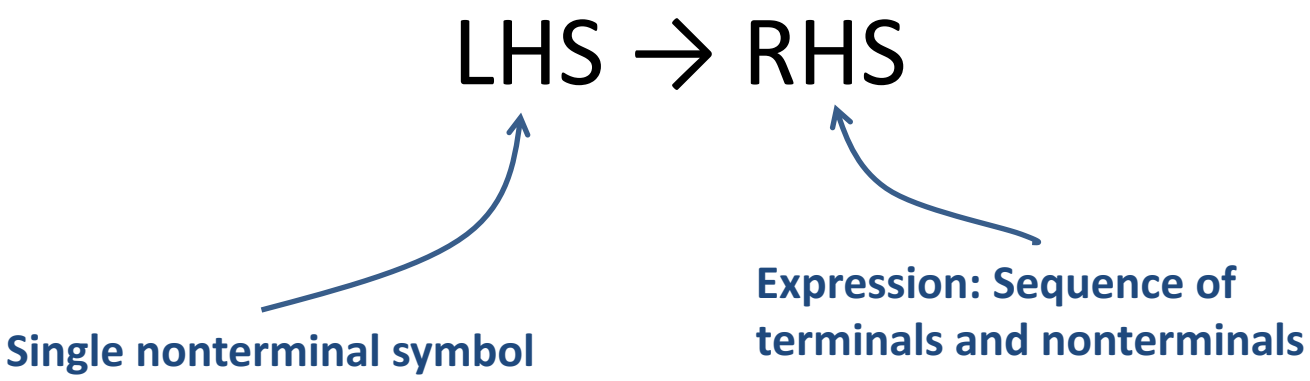
Placeholder / interior nodes in the parse tree

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



Examples:

S → ‘(‘ S ‘)’

S → ε

Production Shorthand

Nonterm → expression

Nonterm→ ε

S → ‘(‘ S ‘)’

S → ε

equivalently:

Nonterm → expression

| ε

S → ‘(‘ S ‘)’

| ε

equivalently:

Nonterm → expression | ε

S → ‘(‘ S ‘)’ | ε

Derivations

To derive a string:

- Start by setting "*Current Sequence*" to the start symbol
- Repeatedly,
  - 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 "$\overset{+}{\Rightarrow}$" for "*derives in one or more steps*" (also written as "$\Rightarrow^{+}$")
- We'll use the symbol "$\overset{*}{\Rightarrow}$" for "*derives in zero or more steps*" (also written as "$\Rightarrow^{*}$")

An Example Grammar

An Example Grammar

Terminals
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
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

Terminals

begin } Program
end } boundary

semicolon — Represents ";" Separates statements

assign — Represents "=" in an assignment statement

id

plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin } Program
end } boundary

semicolon — Represents ";" Separates statements

assign — Represents "=" in an assignment 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 "=" in an assignment statement

id

Identifier / variable name

plus

Represents "+" operator in an 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

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
end } boundary

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 — An 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**

Derivation Sequence

Parse Tree

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

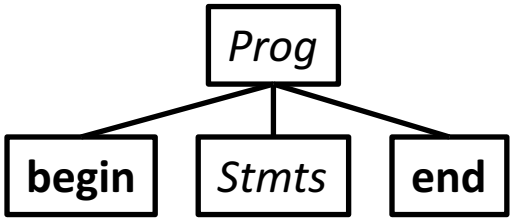
Nonterminal

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

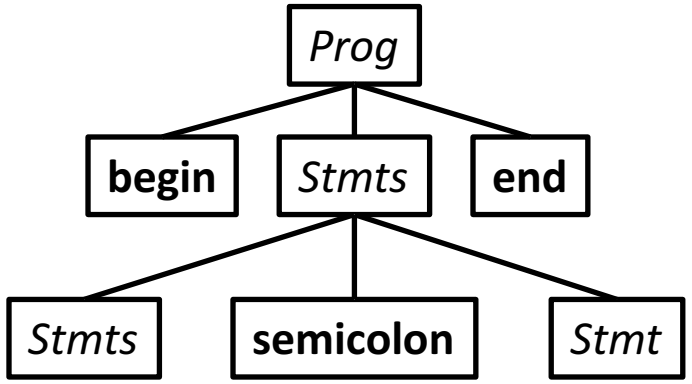
Nonterminal

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

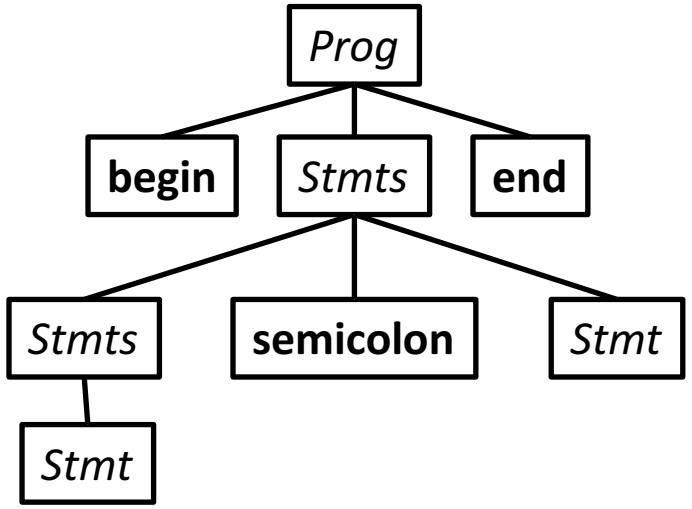
Nonterminal

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

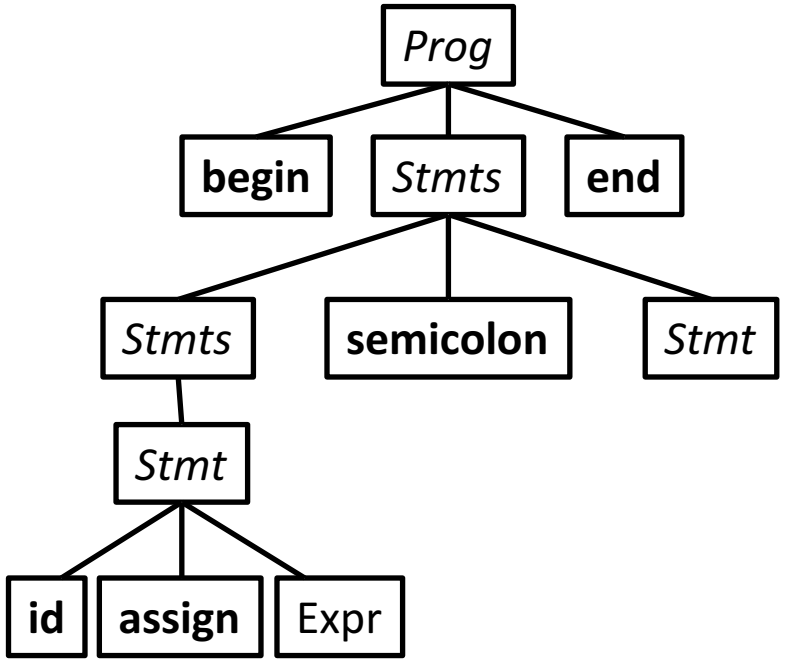
Nonterminal

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

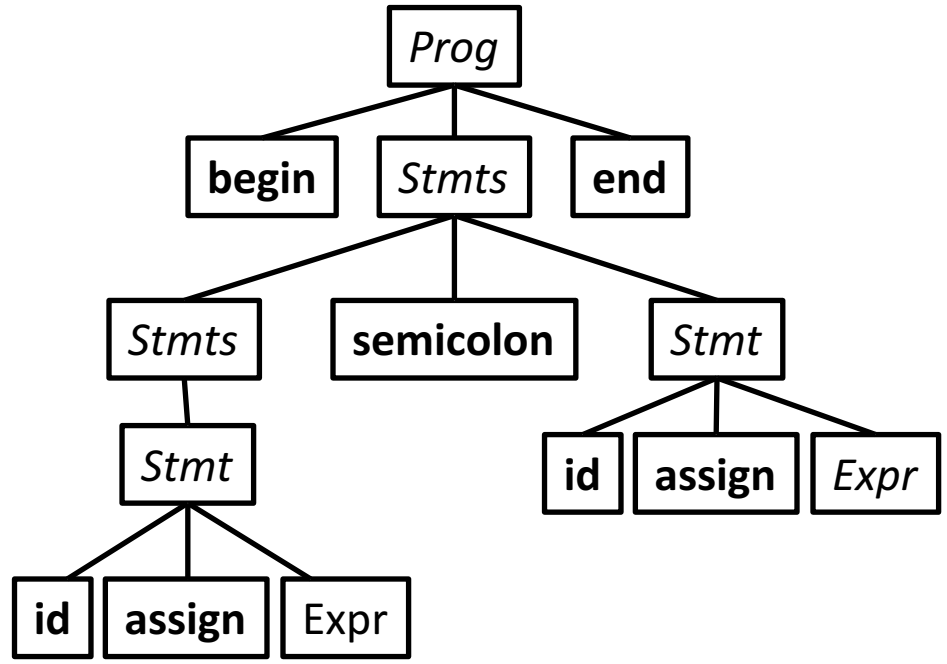
Nonterminal

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

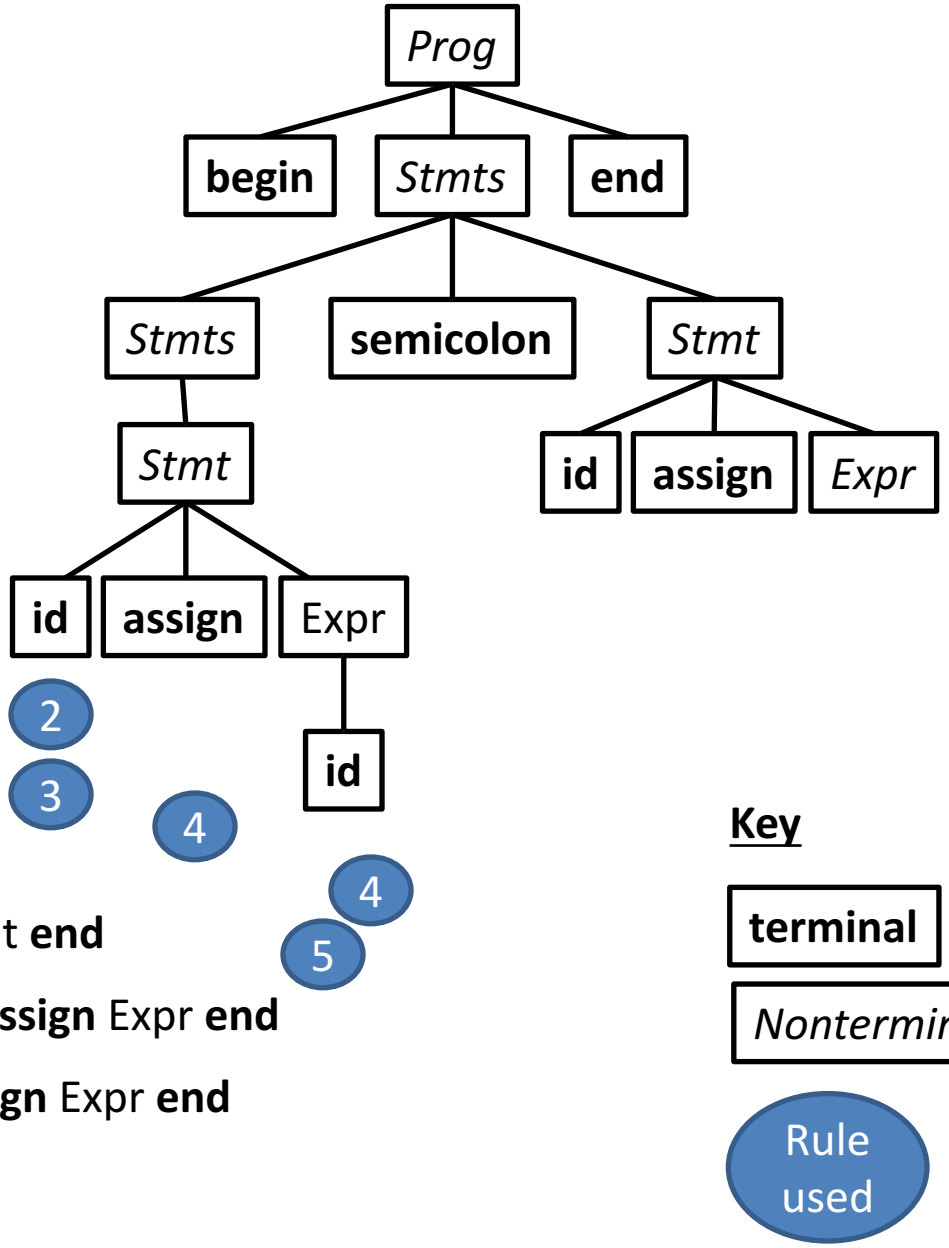
Nonterminal

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**

⇒ **begin** Stmts **semicolon** Stmt **end**

⇒ **begin** Stmt **semicolon** Stmt **end**

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

⇒ **begin id assign** Expr **semicolon id assign** Expr **end**

⇒ **begin id assign id semicolon id assign** Expr **end**

Key

terminal

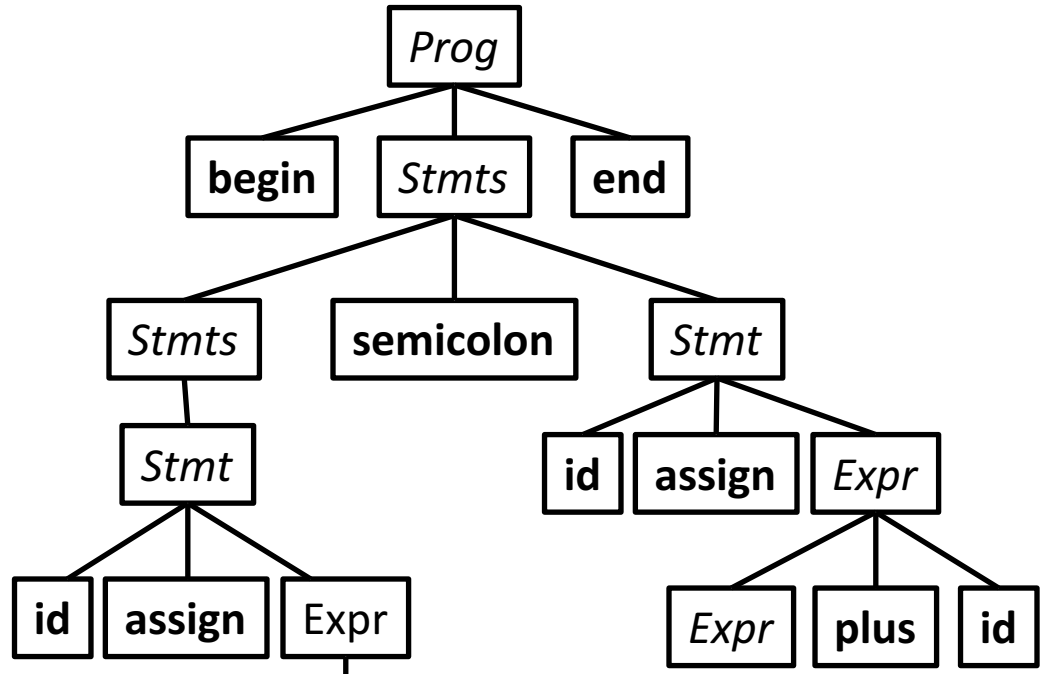
Nonterminal

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, 5)

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

⇒ **begin id assign id semicolon id assign** Expr **plus id end**

Key

terminal

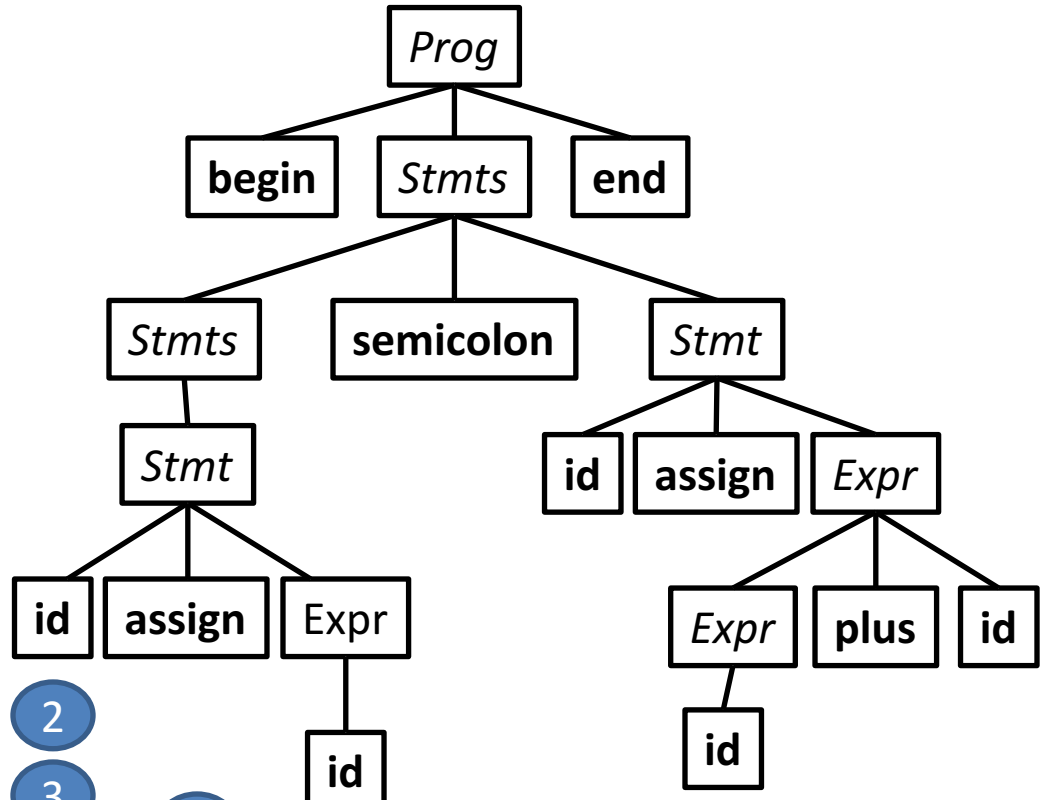
Nonterminal

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**

⇒ **begin** Stmts **semicolon** Stmt **end**

⇒ **begin** Stmt **semicolon** Stmt **end**

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

⇒ **begin id assign** Expr **semicolon id assign** Expr **end**

⇒ **begin id assign id semicolon id assign** Expr **end**

⇒ **begin id assign id semicolon id assign** Expr **plus id end**

⇒ **begin id assign id semicolon id assign id plus id end**

Key

terminal

Nonterminal

Rule used

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>

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

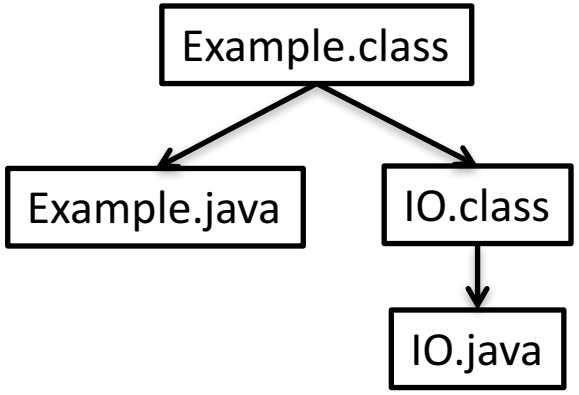
Example.class depends on example.java and IO.class

```
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



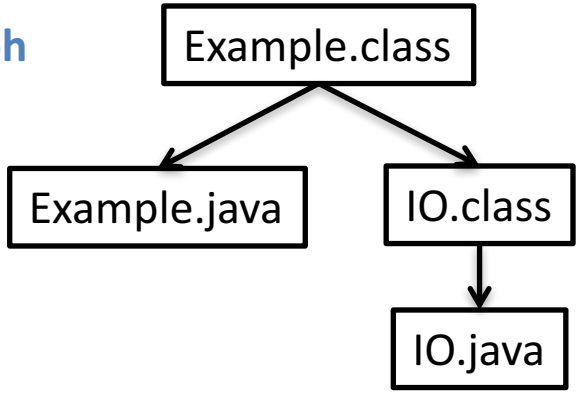
Example

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

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

Makefiles: Dependencies

Internal Dependency graph



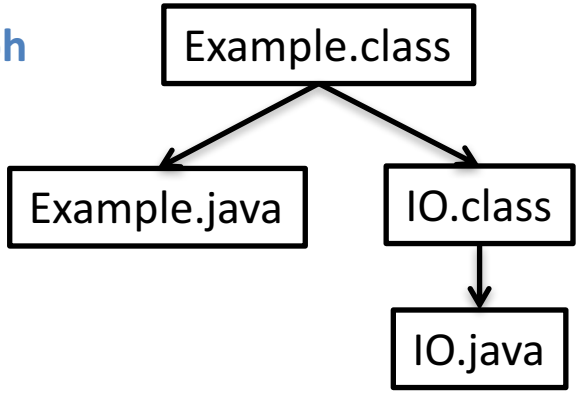
Example

```
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

Example

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 via 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 via 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 via 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 makefiles
- Next time: we'll look at grammars in more detail