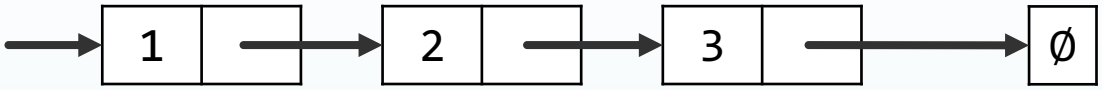
EECS 390 – Lecture 4

Grammars

Review: Scheme Lists

- A list is a sequence of pairs terminated by an empty list



- An empty list is denoted by '()

```
> (define y (cons 1 (cons 2 (cons 3 '()))))
> y
(1 2 3)
> (define y (list 1 2 3))
> y
(1 2 3)
> (car y)
1
> (cdr y)
(2 3)
> (cdr (cdr (cdr y)))
()
```

Also (cdddr y) in standard Scheme

Symbolic Data

- In Scheme, both code and data share the same representation

- Quotation specifies that what follows should be treated as data and not evaluated

```
> (define x 3)
> x
3
> 'x
x
> '(hello world)
(hello world)
```

Equivalent to (quote x)

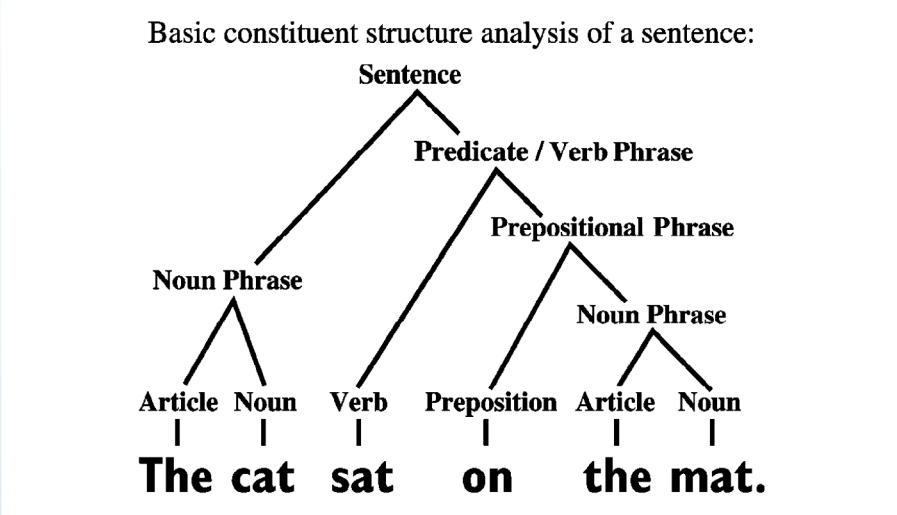
The `eval` procedure can be used to evaluate symbolic data

Review: Levels of Description

- ***Grammar***: what phrases are correct
  - ***Lexical structure***: what sequences of symbols represent correct words
  - ***Syntax***: what sequences of words represent correct phrases
- ***Semantics***: what does a correct phrase mean
- ***Pragmatics***: how do we use a meaningful phrase
- ***Implementation***: how are the actions specified by a meaningful phrase accomplished

Syntactic Structure

- In English, a sentence is just a string of characters
  - Example: "The cat sat on the mat."
- But much of what makes a sentence meaningful is that it has structure



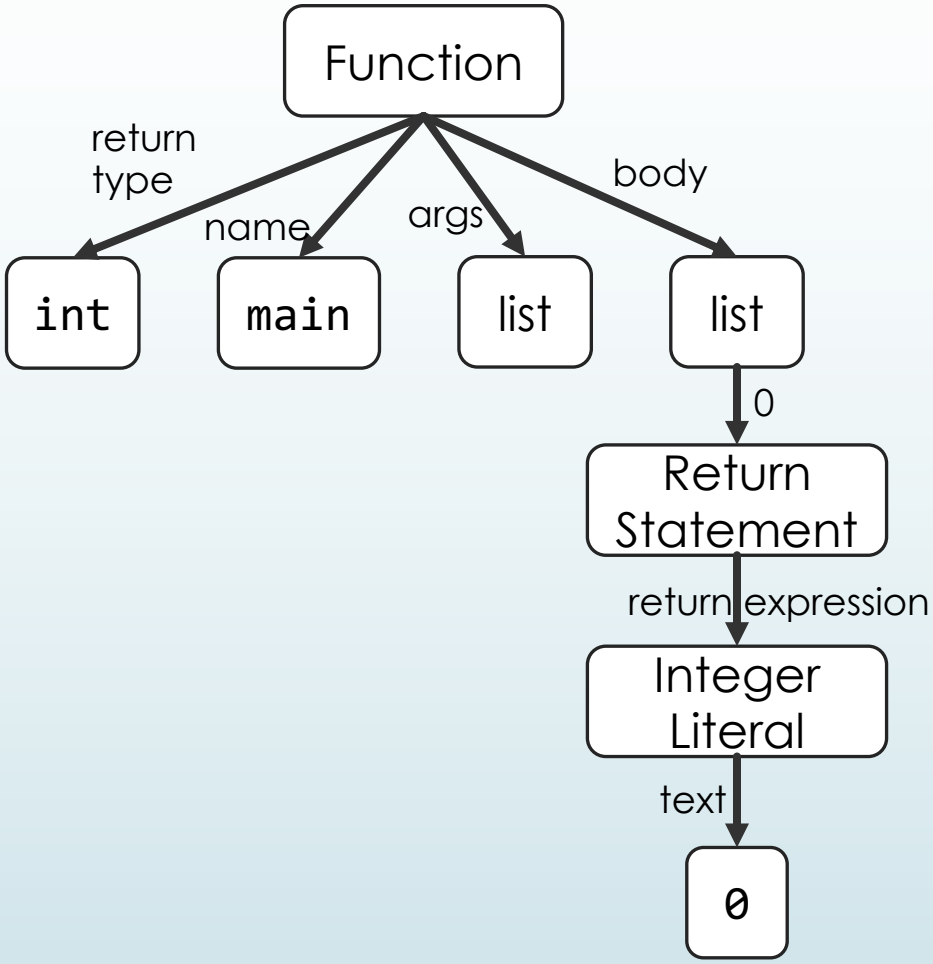
- The same goes for code

Example from https://mytext.cnm.edu/lesson/gb4-structure-of-a-sentence/

Code Structure

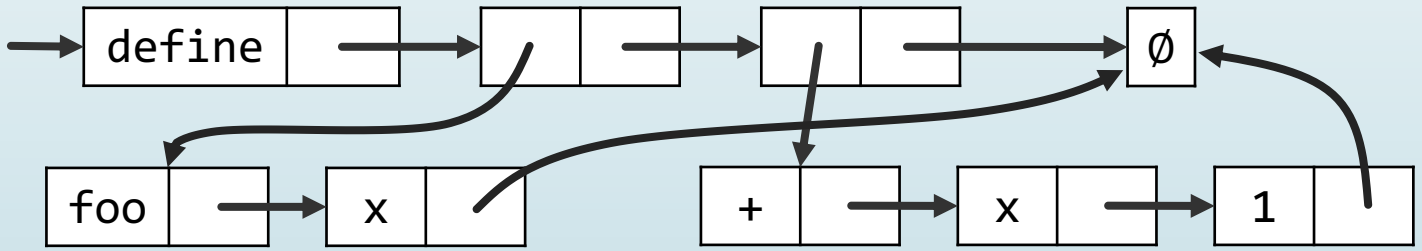
- Code example:

```
int main() {
  return 0;
}
```



Grammars and Parsing

- The ***grammar*** specifies what constitutes structurally correct phrases/sentences/code fragments
- A ***parser*** converts the linear sequence of characters that make up a phrase into a structured representation based on the grammar
- Scheme terminology (relevant to P2):
  - ***External representation***: the character sequence that represents a piece of code or data in source code
    - Example: `"(define (foo x) (+ x 1))"`
  - ***Internal representation***: the actual Scheme-level, runtime representation of a piece of code or data



Agenda

- Regular Expressions
- Context-Free Grammars

Regular Expression

- Sequence of characters that define a pattern for matching strings
- Components:
  - Empty string: ε
  - Individual characters from an alphabet: a, b
  - Concatenation: ab
  - Alternation or choice: $a|b$
  - Kleene star, zero or more occurrences of an element: a^*
- Precedence: Kleene star > concatenation > alternation
- Parentheses used for disambiguation

RegEx Examples

- $a|b$ — matches only the strings a and b
- a^*b — matches any number of a's followed by a b
  - $b,\ ab,\ aab$
- $(a|b)^*$ — any number of a's and b's
  - $\varepsilon,\ a,\ b,\ aa,\ ab,\ ba,\ bb,\ aaa$
- $ab^*(c|\varepsilon)$ — an a, followed by any number of b's, followed by an optional c
  - $a,\ ac,\ ab,\ abc,\ abb,\ abbc$

Shorthands

- Many systems provide shorthands for common cases
- Question mark: zero or one occurrence
  - $ab^*(c|\varepsilon) == ab^*c?$
- Plus sign: one or more occurrences
  - a^+b matches ab, aab, but not b
- Square brackets: set of characters
  - $[abc] == (a|b)|c$
- Character ranges
  - $[a-d] == [abcd]$

Identifiers

- RegEx to match identifiers and keywords in C-like language

$$[a-zA-Z\_][a-zA-Z\_0-9]^*$$

- An identifier or keyword starts with a letter or underscore, followed by any number of letters, digits, and underscores

- Examples: $\_,\ x,\ int,\ static\_cast,\ L337$

RegEx Limitations

- Regular expressions are powerful, but cannot express many syntax rules
- Example: $a^n b^n$, i.e. any number of a's followed by the same number of b's
  - $\varepsilon,\ ab,\ aabb,\ aaabbb$
- Example: matching parentheses
  - $(),\ (()),\ (()),\ ((()))$

Context-Free Grammar (CFG)

- Defines a recursive process for matching a string
- ***Terminals***: symbols from a language
  - Example: $\varepsilon,\ a,\ b$
- ***Variables***: items that can be replaced with other variables or terminals
  - Also called ***nonterminals***
  - Example: S
- ***Production rules***: legal ways to replace variables with other variables or terminals
  - Example: $S \rightarrow \varepsilon,\ S \rightarrow a\ S\ b$
- ***Start variable***: where to start the replacement process
  - Example: S

Derivations

- Sequence of rule applications, starting with the start variable and ending with a string of terminals
- CFG:

  1) $S \rightarrow \varepsilon$

  2) $S \rightarrow a\,S\,b$

- String that can be derived from CFG:

$S \rightarrow a\,S\,b$	by application of rule (2)
$\rightarrow a\,a\,S\,b\,b$	by application of rule (2)
$\rightarrow a\,a\,b\,b$	by application of rule (1)

- The CFG matches strings containing any number of a's, followed by the same number of b's

Matching Parentheses

- CFG:

  1) $P \rightarrow \varepsilon$

  2) $P \rightarrow (P)$

  3) $P \rightarrow P\ P$

- Derivation of (()):

$P \rightarrow (P)$ by application of rule (2)

$\rightarrow ((P))$ by application of rule (2)

$\rightarrow (( ))$ by application of rule (1)

Alternate Derivations

1) $P \rightarrow \varepsilon$
2) $P \rightarrow (P)$
3) $P \rightarrow P\ P$

- Derivations of ()()

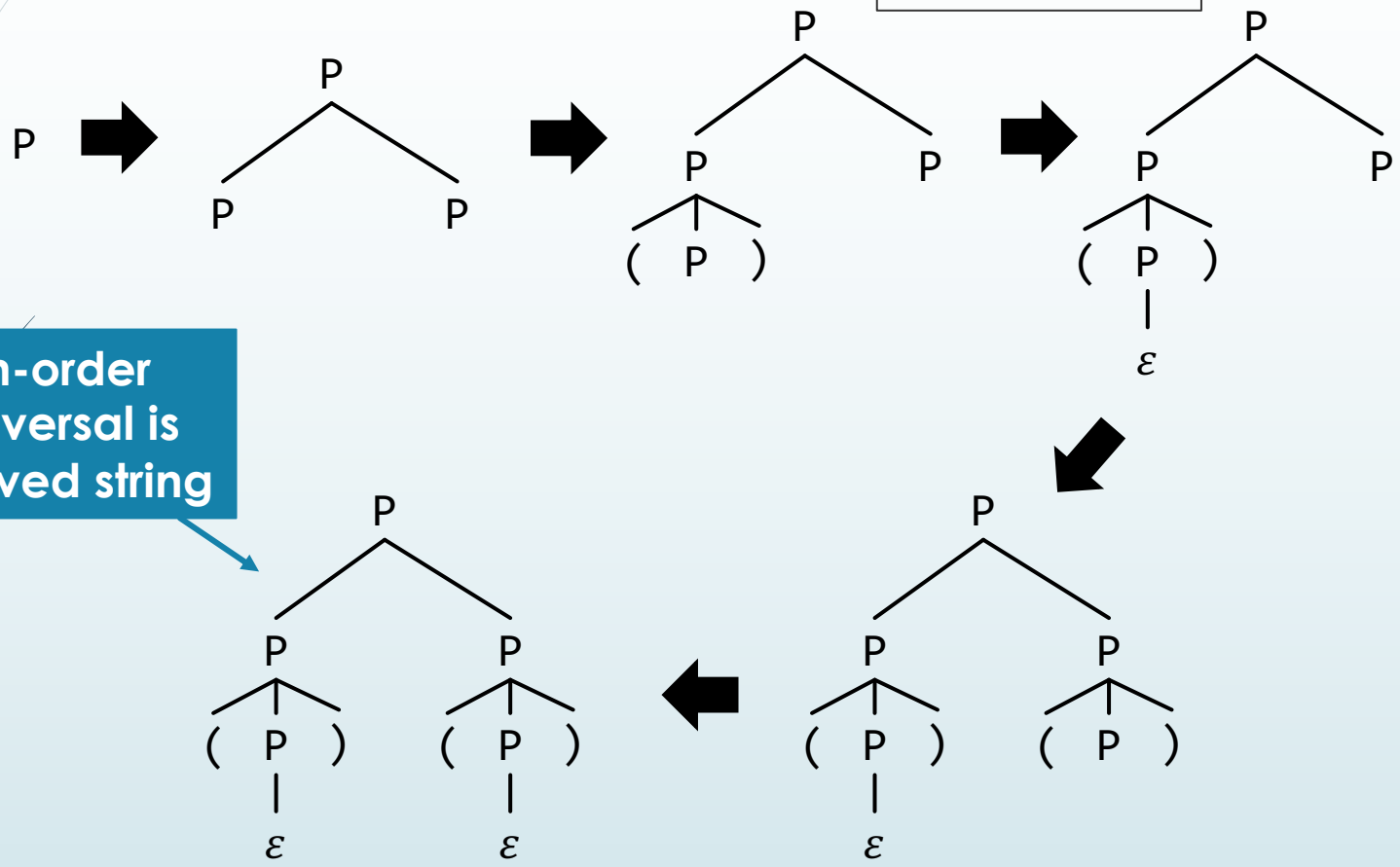
$P \rightarrow P\ P$	by application of rule (3)
$\rightarrow (P)\ P$	by application of rule (2) on 1st P
$\rightarrow ( )\ P$	by application of rule (1)
$\rightarrow ( )\ (P)$	by application of rule (2)
$\rightarrow ( )\ ( )$	by application of rule (1)

$P \rightarrow P\ P$	by application of rule (3)
$\rightarrow P\ (P)$	by application of rule (2) on 2nd P
$\rightarrow P\ ( )$	by application of rule (1)
$\rightarrow (P)\ ( )$	by application of rule (2)
$\rightarrow ( )\ ( )$	by application of rule (1)

- Not a problem if derivation trees are identical

Derivation Trees

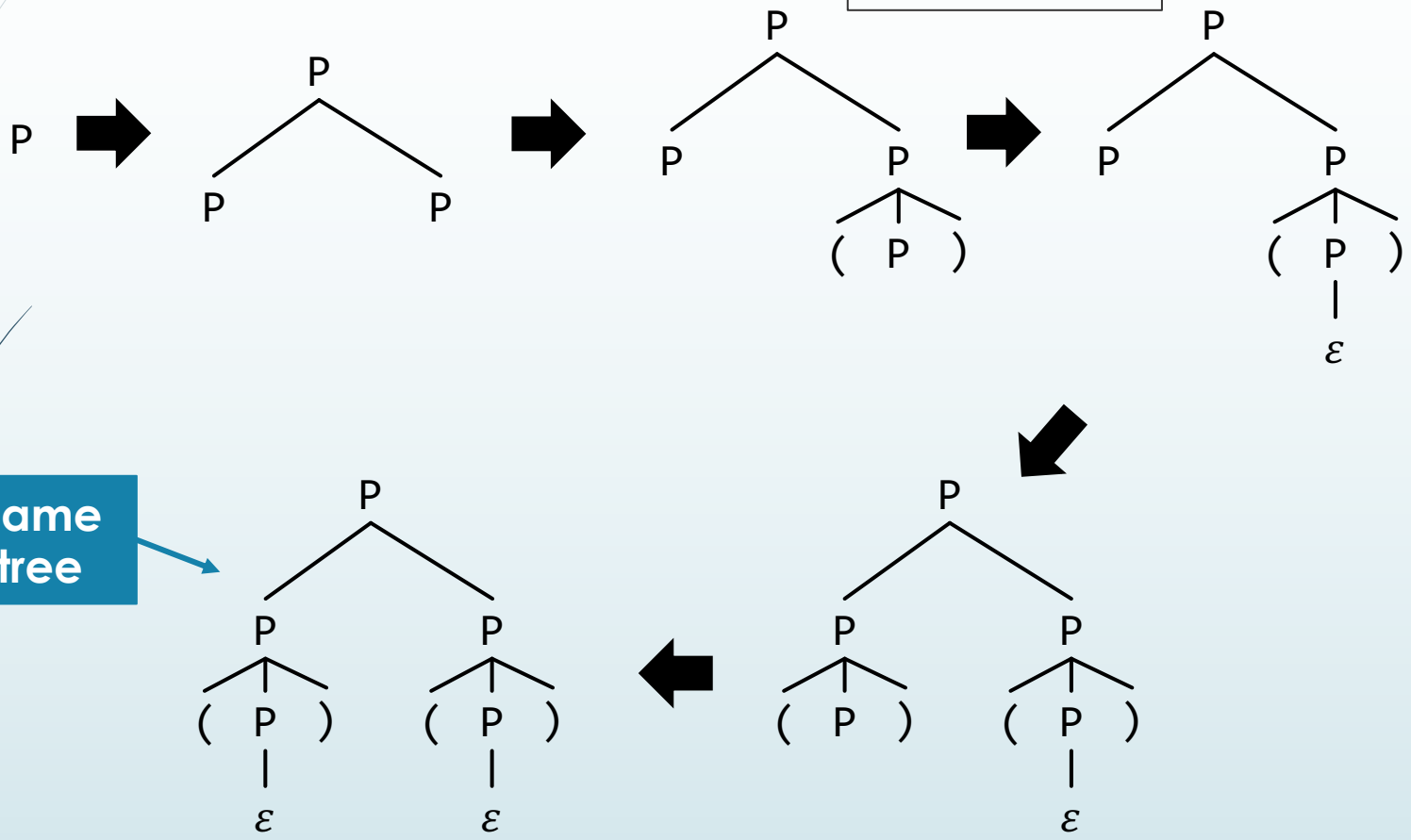
$$
\begin{aligned}
P &\rightarrow P\ P \\
&\rightarrow (\ P\ )\ P \\
&\rightarrow (\ )\ P \\
&\rightarrow (\ )\ (\ P\ ) \\
&\rightarrow (\ )\ (\ )
\end{aligned}
$$



In-order traversal is derived string

Derivation Trees

$$
\begin{aligned}
P &\rightarrow P\ P \\
&\rightarrow (\ P\ )\ P \\
&\rightarrow (\ )\ P \\
&\rightarrow (\ )\ (\ P\ ) \\
&\rightarrow (\ )\ (\ )
\end{aligned}
$$



Break Time!


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Arithmetic Grammar

1) $E \rightarrow E + E$
2) $E \rightarrow E * E$
3) $E \rightarrow a$
4) $E \rightarrow b$

- Derivations of $a + b * a$

$E \rightarrow E + E$	by rule (1)
$\rightarrow E + E * E$	by rule (2) on 2nd E
$\rightarrow a + E * E$	by rule (3) on 1st E
$\rightarrow a + b * E$	by rule (4) on 1st E
$\rightarrow a + b * a$	by rule (3)

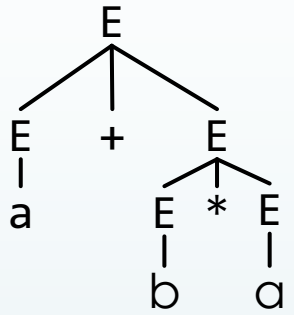
$E \rightarrow E * E$	by rule (2)
$\rightarrow E + E * E$	by rule (1) on 1st E
$\rightarrow a + E * E$	by rule (3) on 1st E
$\rightarrow a + b * E$	by rule (4) on 1st E
$\rightarrow a + b * a$	by rule (3)

Ambiguity

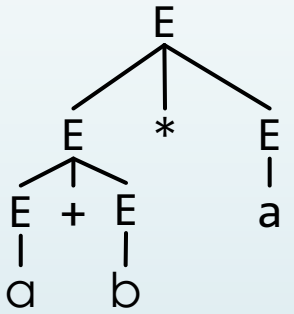
$$
\begin{array}{l}
1)\ E \rightarrow E + E \\
2)\ E \rightarrow E * E \\
3)\ E \rightarrow a \\
4)\ E \rightarrow b
\end{array}
$$

- Grammar is ***ambiguous*** since the different derivations result in different trees

$$
\begin{array}{lll}
E \rightarrow E + E & & \text{by rule (1)} \\
\quad \rightarrow E + E * E & & \text{by rule (2) on 2nd } E \\
\quad \rightarrow a + E * E & & \text{by rule (3) on 1st } E \\
\quad \rightarrow a + b * E & & \text{by rule (4) on 1st } E \\
\quad \rightarrow a + b * a & & \text{by rule (3)}
\end{array}
$$



$$
\begin{array}{lll}
E \rightarrow E * E & & \text{by rule (2)} \\
\quad \rightarrow E + E * E & & \text{by rule (1) on 1st } E \\
\quad \rightarrow a + E * E & & \text{by rule (3) on 1st } E \\
\quad \rightarrow a + b * E & & \text{by rule (4) on 1st } E \\
\quad \rightarrow a + b * a & & \text{by rule (3)}
\end{array}
$$



- First tree corresponds to $*$ having higher precedence
- Usually resolved by specifying precedence rules

Extended Backus-Naur Form

- Grammars for programming languages are generally written in an ***extended Backus-Naur form (EBNF)***
- Includes representation of production rules in a more limited character set
  - e.g. *E* := *E* + *E* instead of $E \rightarrow E + E$
- Adds shorthands like in regular expressions
  - e.g. Kleene star, alternation with | rather than separate production rules
- Language-specific extensions
  - e.g. "except", "one of" in Java grammar

Scheme Lists

- From R5RS spec:

  $\langle list \rangle \rightarrow (\langle datum \rangle^*) \mid (\langle datum \rangle^+ \,.\, \langle datum \rangle) \mid \langle abbreviation \rangle$

  $\langle abbreviation \rangle \rightarrow \langle abbrev\ prefix \rangle \langle datum \rangle$

  $\langle abbrev\ prefix \rangle \rightarrow \text{'} \mid \text{`} \mid \text{,} \mid \text{,@}$

- List can be
  - Zero or more datums in parentheses
  - Parentheses containing one or more datums, a period, and a single datum
  - A quotation character followed by a datum

Vexing Parse

- In languages with complex syntax, such as C++, ambiguity cannot be avoided in the grammar
  - External rules are specified to disambiguate fragments

```
struct foo {
  foo() {
    cout << "foo::foo()" << endl;
  }
  foo(int x) {
    cout << "foo::foo(" << x << ")" << endl;
  }
  void operator=(int x) {
    cout << "foo::operator=(" << x << ")" << endl;
  }
};

int a = 3, b = 4;

int main() {
  foo(a);       // equivalent to foo a;
  foo(b) = 3;   // equivalent to foo b = 3;
}
```

C++ disambiguates in favor of declarations

Names can be parenthesized in declarations

```
foo::foo()
foo::foo(3)
```

Most Vexing Parse

- A most vexing example:

```
struct bar {
  bar(foo f) {
    cout << "bar::bar(foo)" << endl;
  }
};

bar c(foo()); // equivalent to bar c(foo);
```

Nothing printed

C++ disambiguates in favor of function declarations

- Clang warning:

```
foo.cpp:28:8: warning: parentheses were disambiguated as a function
declaration
      [-Wvexing-parse]
  bar c(foo()); // equivalent to bar c(foo);
       ^~~~~~
foo.cpp:28:9: note: add a pair of parentheses to declare a variable
  bar c(foo()); // equivalent to bar c(foo);
       ^
       (   )
```