EECS 490 – Lecture 5

Grammars

Announcements

- Project 1 due **Thursday** 9/21 at 8pm

- Homework 2 due Friday 9/29 at 8pm

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

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
  - *ε, a, b, aa, ab, ba, bb, aaa*
- *ab*(c* | *ε)* — 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 | ε) == 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
  - *ε, ab, aabb, aaabbb*
- Example: matching parentheses
  - *(), ()(), (()), (()())*

Agenda

- Regular Expressions
- **Context-Free Grammars**

Context-Free Grammar

- Defines a recursive process for matching a string
- *Terminals*: symbols from a language
  - Example: *ε, a, b*
- *Variables*: items that can be replaced with other variables or terminals
  - 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 *()()*

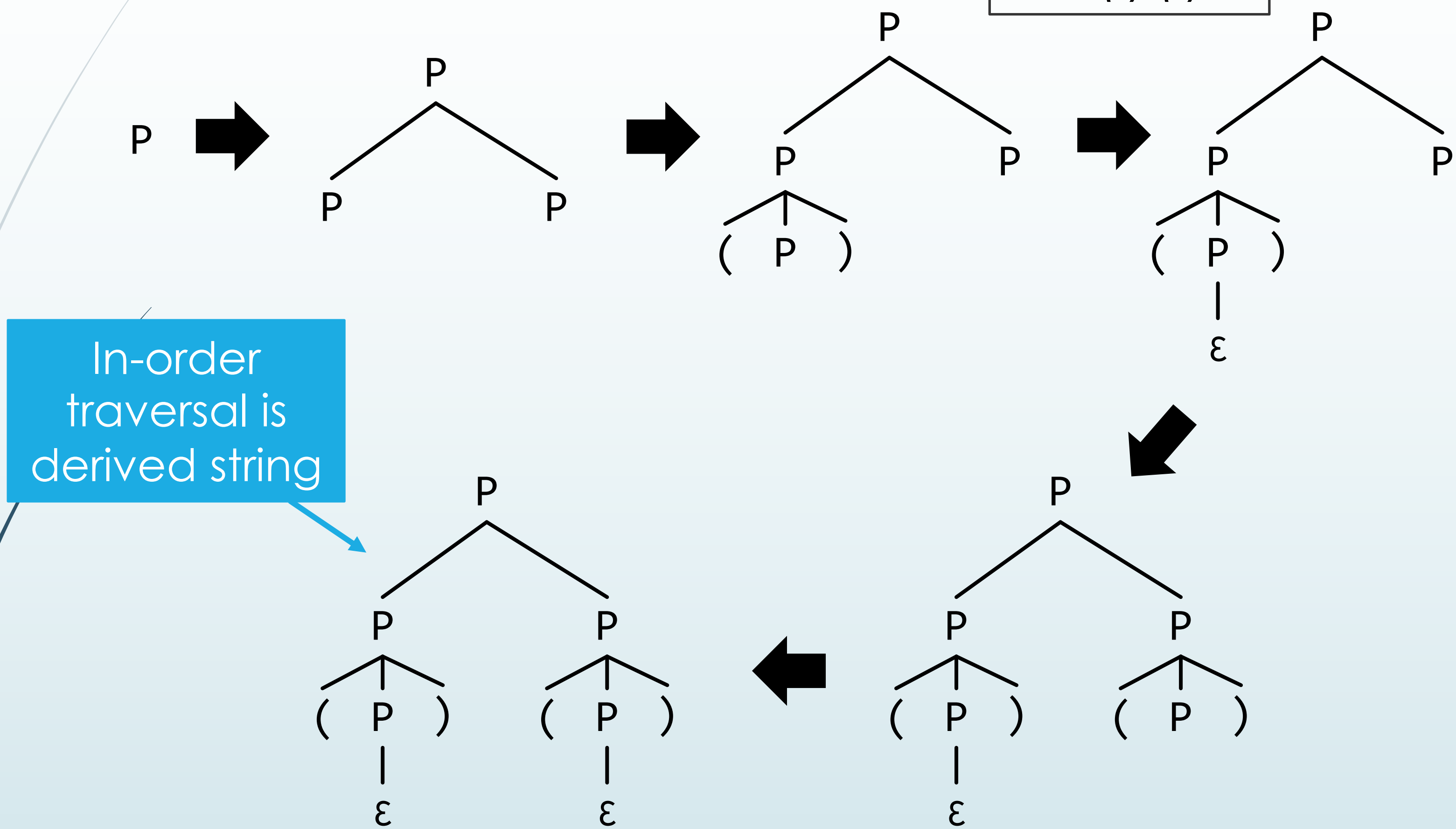
$P \rightarrow P\ P$	by application of rule (3)
$\rightarrow (P)\ P$	by application of rule (2) on 1st *P*
$\rightarrow (\,)\ P$	by application of rule (1)
$\rightarrow (\,)\ (P)$	by application of rule (2)
$\rightarrow (\,)\ (\,)$	by application of rule (1)

$P \rightarrow P\ P$	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

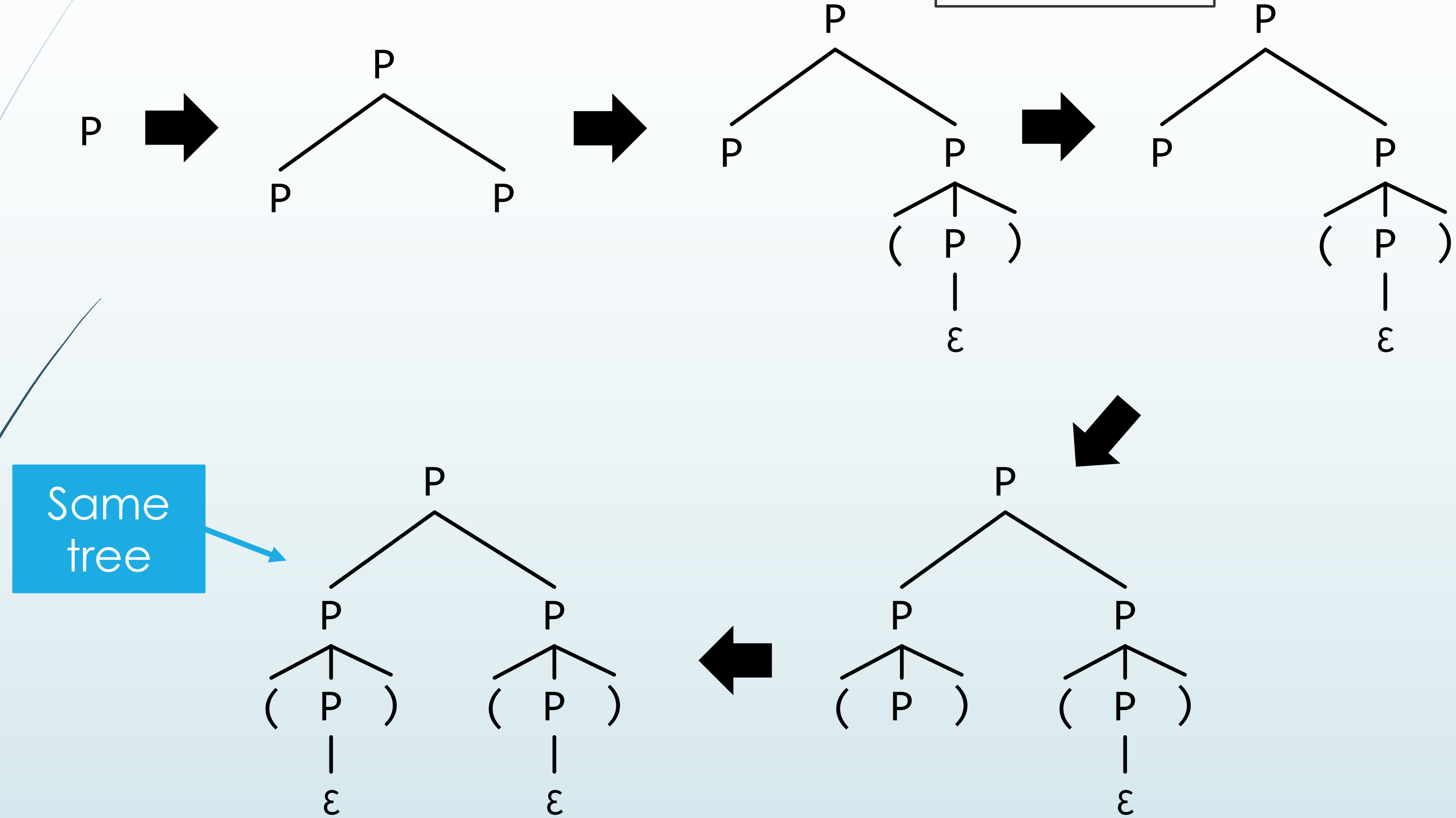
Derivation Trees

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



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

Break Time!

- We’ll start again in one minute.

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)

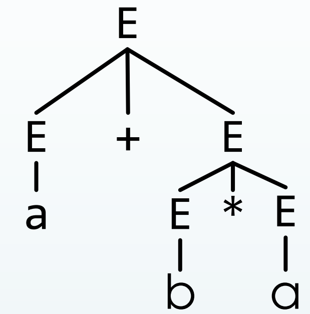
$E \rightarrow E * E$	by rule (2)
$\rightarrow E + E * E$	by rule (1) on 1st E
$\rightarrow a + E * E$	by rule (3) on 1st E
$\rightarrow a + b * E$	by rule (4) on 1st E
$\rightarrow a + b * a$	by rule (3)

Ambiguity

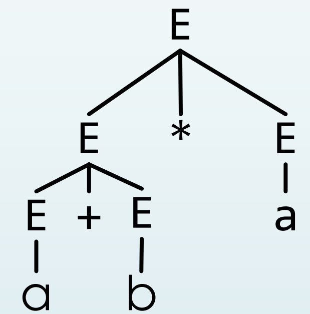
1) $E \rightarrow E + E$
2) $E \rightarrow E * E$
3) $E \rightarrow a$
4) $E \rightarrow b$

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

$E \rightarrow E + E$	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)



- 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

Identifiers

- Identifiers in a C-like language described using Java's EBNF

Alternation on separate lines

```
Identifier: except Keyword and BooleanLiteral
    IdentifierStartCharacter
    IdentifierStartCharacter IdentifierCharacters

IdentifierStartCharacter:
    _
    LowerCaseLetter
    UpperCaseLetter

IdentifierCharacters:
    IdentifierCharacter
    IdentifierCharacters IdentifierCharacter

IdentifierCharacter:
    IdentifierStartCharacter
    Digit

LowerCaseLetter: one of
    a b c d e f g h i j k l m n o p q r s t u v w x y z

UpperCaseLetter: one of
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Digit: one of
    0 1 2 3 4 5 6 7 8 9
```

Also alternation

C-Style Comments in Java

- Need to ensure that */ ends a comment, as well as **/, ***/, etc.

```
TraditionalComment:
    / * CommentTail

CommentTail:
    * CommentTailStar
    NotStar CommentTail

CommentTailStar:
    /
    * CommentTailStar
    NotStarNotSlash CommentTail

NotStar:
    InputCharacter but not *
    LineTerminator

NotStarNotSlash:
    InputCharacter but not * or /
    LineTerminator
```

Scheme Lists

- From R5RS spec:

  ⟨list⟩ → (⟨datum⟩*) | (⟨datum⟩+ . ⟨datum⟩) | ⟨abbreviation⟩

  ⟨abbreviation⟩ → ⟨abbrev prefix⟩ ⟨datum⟩

  ⟨abbrevprefix⟩ → ’ | ‘ | , | ,@

- 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);
        ^
        (    )
```