Languages: Predictive parsing 2

Glenn Bruns
CSUMB

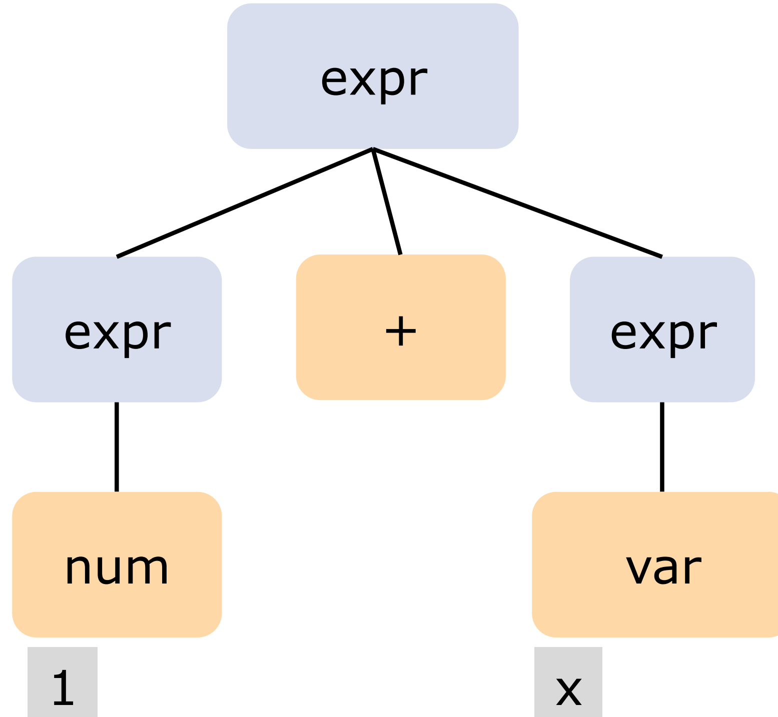
Learning outcomes

After this lecture, you should be able to:

- derive parse trees from a BNF grammar
- define what it means for a grammar to be "ambiguous"
- use two additional rules for transforming a grammar

Parse trees

```
expr = num
     | var
     | expr + expr
```



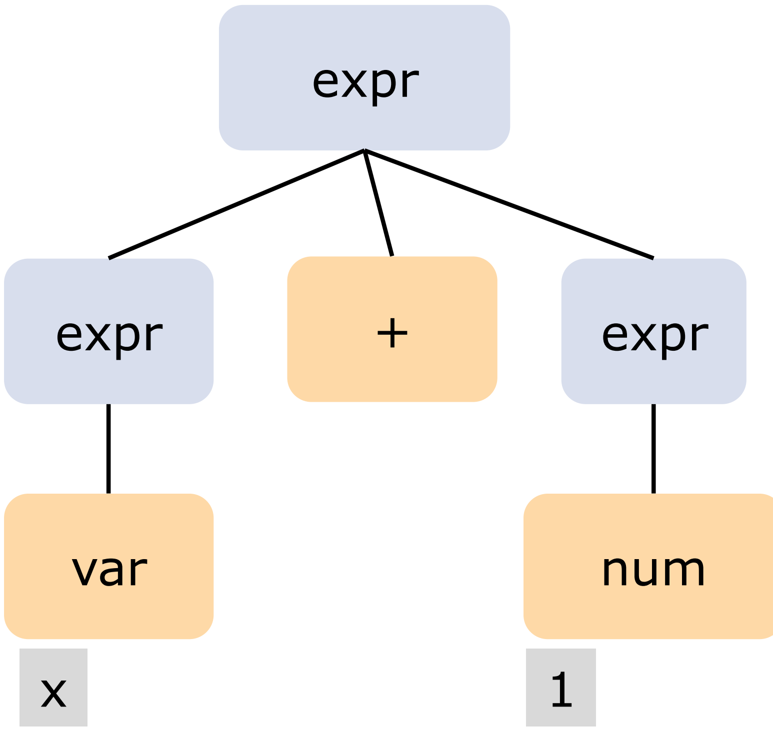
Rules:

1. root is labeled with the start symbol
2. the symbols in one of its productions become child nodes
3. continue until all leaf nodes are terminal symbols

Exercise

```
expr = num
     | var
     | expr + expr
```

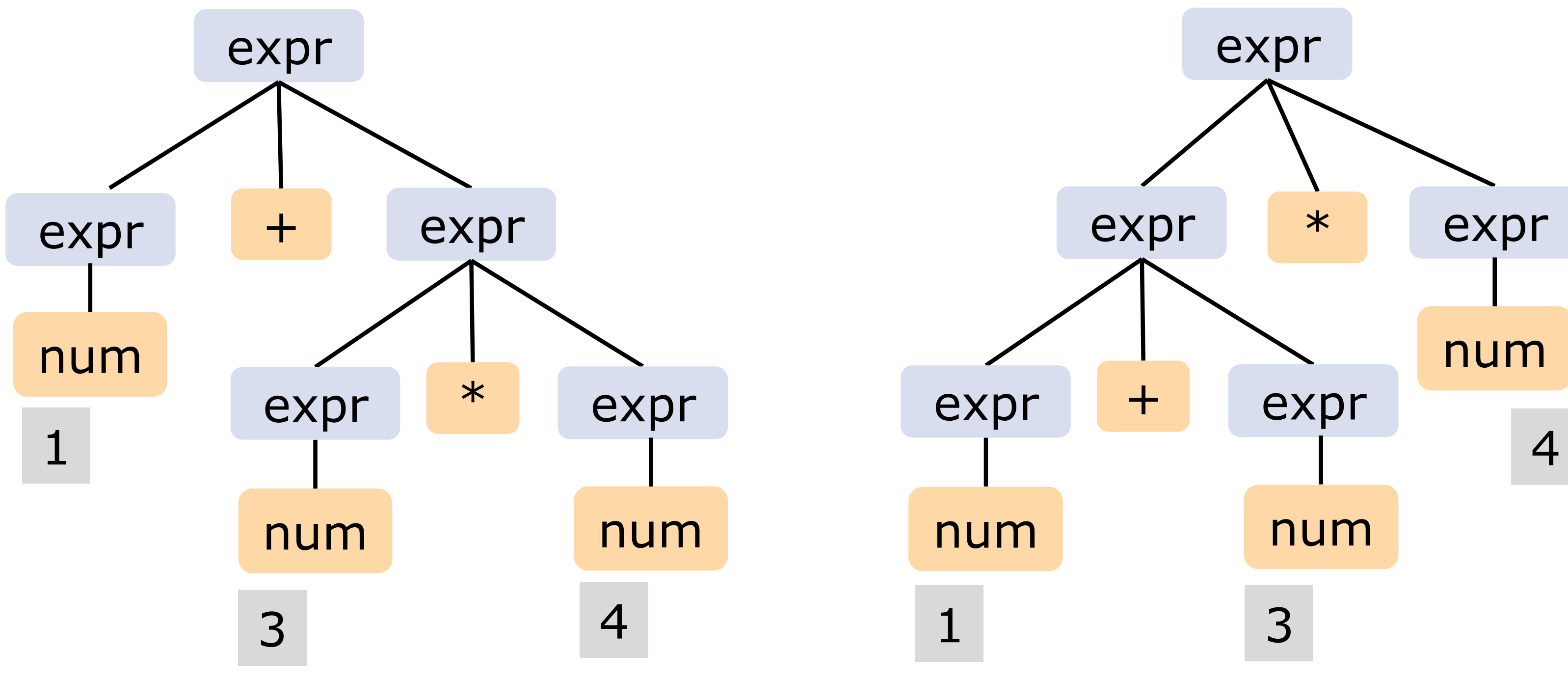
derive "x + 1" using a parse tree



Ambiguous grammar

```
expr = num | expr + expr | expr * expr
```

Create a parse tree for "1 + 3 * 4"



1 + (3 * 4)

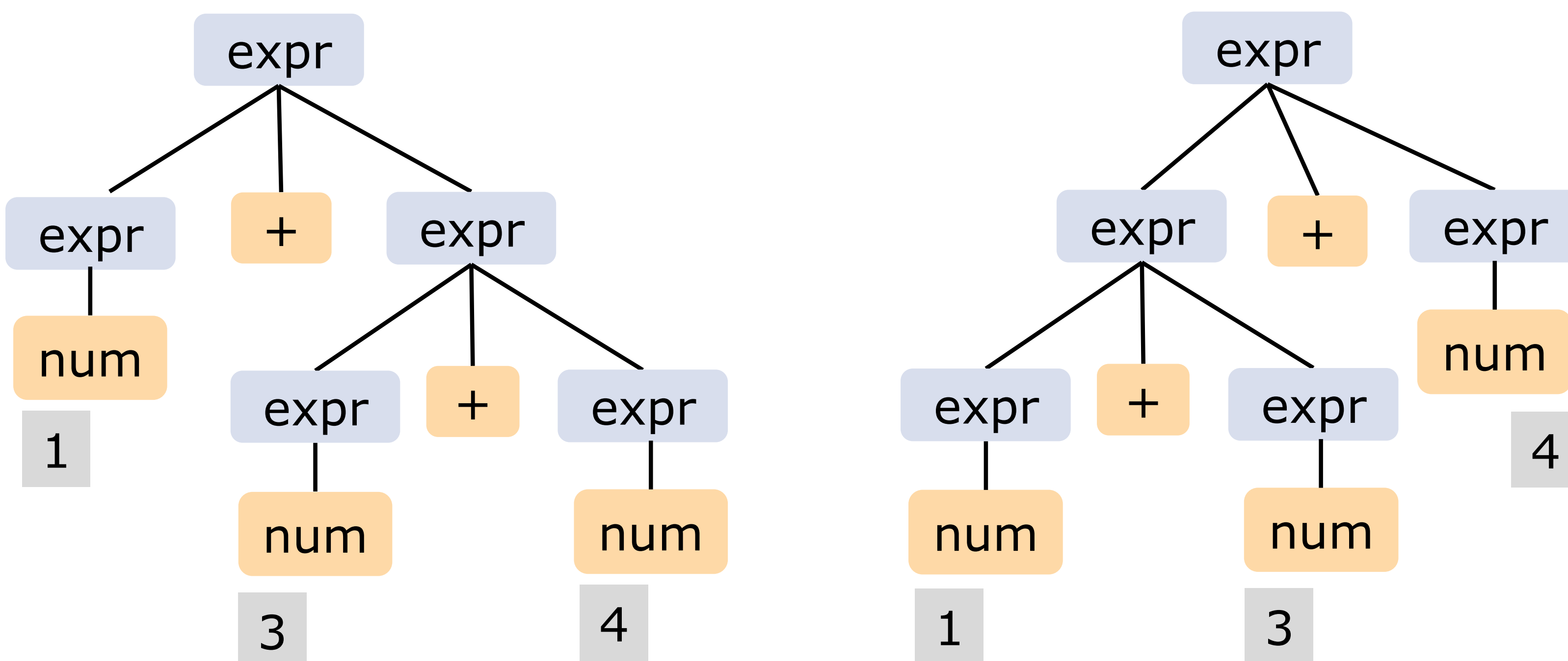
(1 + 3) * 4

Ambiguous grammar

A grammar is ambiguous is, for some string that can be derived from the grammar, there is more than one parse tree.

Different parse trees usually suggest different meanings.

So: we usually want unambiguous grammars.



Review: predictive parsing

```
stmt ::= print ( expr )
       | id = expr
expr ::= id | num
```

```
void stmt() {
switch (lookahead) {
      case PRINT:
         match(PRINT); match("("); expr(); match(")";
         break;
      case ID:
         match(ID); match("="); expr()
         break;
      default:
         error("syntax error");
   }
}
```

Transforming a BNF grammar

In the last lecture we learned how to deal with:

- left recursion
- empty productions

Two more ways to transform BNF:

- left factoring
- expanding a non-terminal

Left factoring

Any problems with a predictive parser here?

```
A ::= a B | a C | b D
```

Can we transform the BNF?

```
A ::= a E | b D
E ::= B | C
```

Exercise

Suppose an expression can be a variable or function call.
Examples:

```
tot          (a variable)
max(y)       (a function call)
```

BNF:

```
expr ::= var ( expr )           // a function call
       | var                    // a variable
```

Exercise: Apply left factoring

a solution:

```
expr  ::= var expr1
expr1 ::= ( expr ) | ""
```

Expanding a non-terminal

Any problems with a predictive parser here?

```
A ::= a B | E
E ::= c C | d D
```

Can we transform the BNF into a better form?

```
A ::= a B | c C | d D
```

For predictive parsers, we want that, for every non-terminal in the grammar:

- there's only one production for it, or
- each production starts with a token (maybe "") and each of the tokens are different

Exercise

```
prog ::= stmt | stmt ; prog
stmt ::= ID = expr | ID ( expr )
```

Exercise: There are two problems. What are they?

Fix using left factoring and removing left recursion.

```
prog ::= stmt ; prog1
prog1 ::= "" | stmt ; prog1
stmt ::= ID stmt1
stmt1 ::= = expr | ( expr )
```

Exercise: Expand a non-terminal.

```
prog ::= ID stmt1 ; prog1
prog1 ::= "" | ID stmt1 ; prog1
stmt1 ::= = expr | ( expr )
```

Note: there's no longer a 'stmt' non-terminal

Summary

- A parse tree shows how a string can be derived from a BNF grammar
- To use predictive parsing, our BNF syntax must be in a certain form. Here are some tools to get BNF in that form:
  - eliminate left recursion
  - left-factoring
  - expanding non-terminals