**CS109B Notes for Lecture 5/3/95**

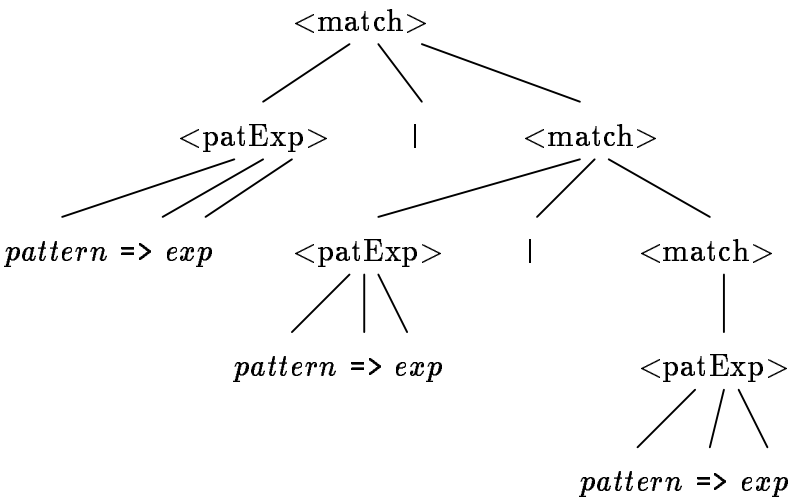
### Parse Trees

- Leaves = terminals or $\epsilon$.
- Interior nodes = SC's.
- Children of a node labeled $< A >$ form (left-to-right) the body of a production for $< A >$.

**Example:** Let us use the grammar for ML matches:

(1) <match> → <patExp> | <match>
(2) <match> → <patExp>
(3) <patExp> → *pattern* => *exp*

Here is one possible parse tree.




### Yield

The *yield* of a parse tree is the labels of the leaves in order from the left.

**Example:** The tree above has yield

*pattern* = *exp* | *pattern* = *exp* | *pattern* = *exp*

### Yields and Languages

There is a parse tree with root $< A >$ and yield $w$ iff $w$ is in $L(<A>)$.

- In "if" direction, proof is an induction on the number of "rounds" needed to demonstrate that $w$ is in $L(<A>)$.
- In "only if" direction, proof is induction on the height of a tree.
- Note the statement applies to, and must be proved simultaneously for, every SC, not just the "start" SC (e.g., $< match >$ that we view as representing our goal language.
- See details pp. 607–8, FCS.

### Ambiguous Grammars

A grammar is *ambiguous* iff it has two parse trees with the same yield.

**Example:** Here is a simpler grammar for ML matches.

(1) <match> → <match> | <match>
(2) <match> → *pattern* => *exp*

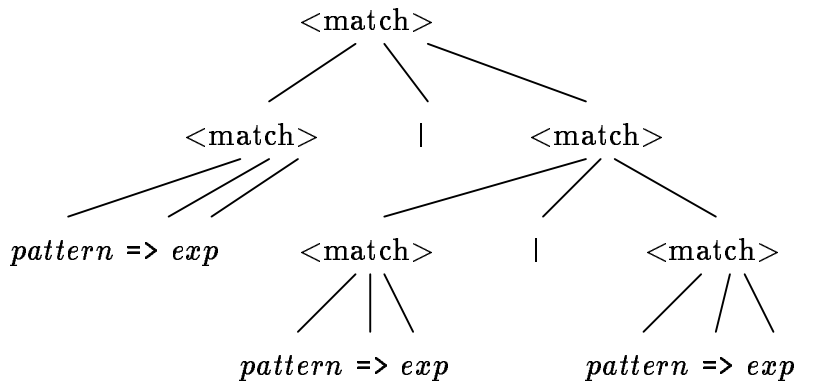
Unfortunately, it has two parse trees for the 3-rule match of our previous example.

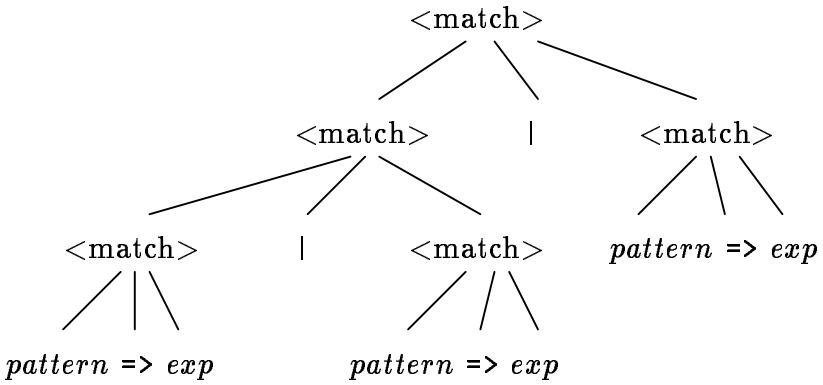
### Why Parse Trees, Ambiguity?

- Provide essential structure that enables compilers to understand the "meaning" of programs and produce the correct machine code.
- Reasonable parser algorithms (that find a parse tree for a string of terminals) require unambiguous grammars.
  - Intuitively, if a string of terminals has two different parse trees, how can the compiler know the structure of the program it is trying to compile?
  - Typical example: the grammar must disambiguate $a + b * c$ (is it $a + (b * c)$ or $(a + b) * c$?) or the compiler cannot guess the correct machine code.

### Class Problem

Here is an ambiguous grammar for nested tuples




as in ML.

- A "tuple" is a parenthesized list of "elements," separated by commas.
- An $< element >$ can be either a tuple or an *atom*; the later is a terminal standing for any non-tuple value, e.g., an `int`.

  $< tuple > \rightarrow$ ( $< elList >$ )

  $< elList > \rightarrow < elList >$ , $< elList >$
  $< elList > \rightarrow < element >$

  $< element > \rightarrow < tuple >$
  $< element > \rightarrow atom$

- First, can you find a tuple that has two parse trees?
- Then, can you fix up the grammar to make it unambiguous?
- Finally, lists in ML are almost the same, with

square rather than round brackets. However:

1. `[]` is a legal list, while `()` is the unit, not a tuple.
2. List elements must have the same type, e.g., `((atom),atom)` is a legal tuple, but `[[atom],atom]` is not a legal list.

Can you find an unambiguous grammar for lists?