Attribute Grammars
and
Abstract Syntax Trees

Attribute Grammars

Attribute grammar: Context-free grammar extended with *attributes* associated to the symbols of the grammar. The attributes provide context-sensitive information that can be transported across the syntax tree.

Attribute grammars were proposed by Donald Knuth to formalize the semantics of a context-free language. They are useful for:

- semantic analysis (e.g., type checking)
- intermediate-code generation
- language interpretation

Attribute grammars can produce *decorated* syntax trees.

Attribute Grammars

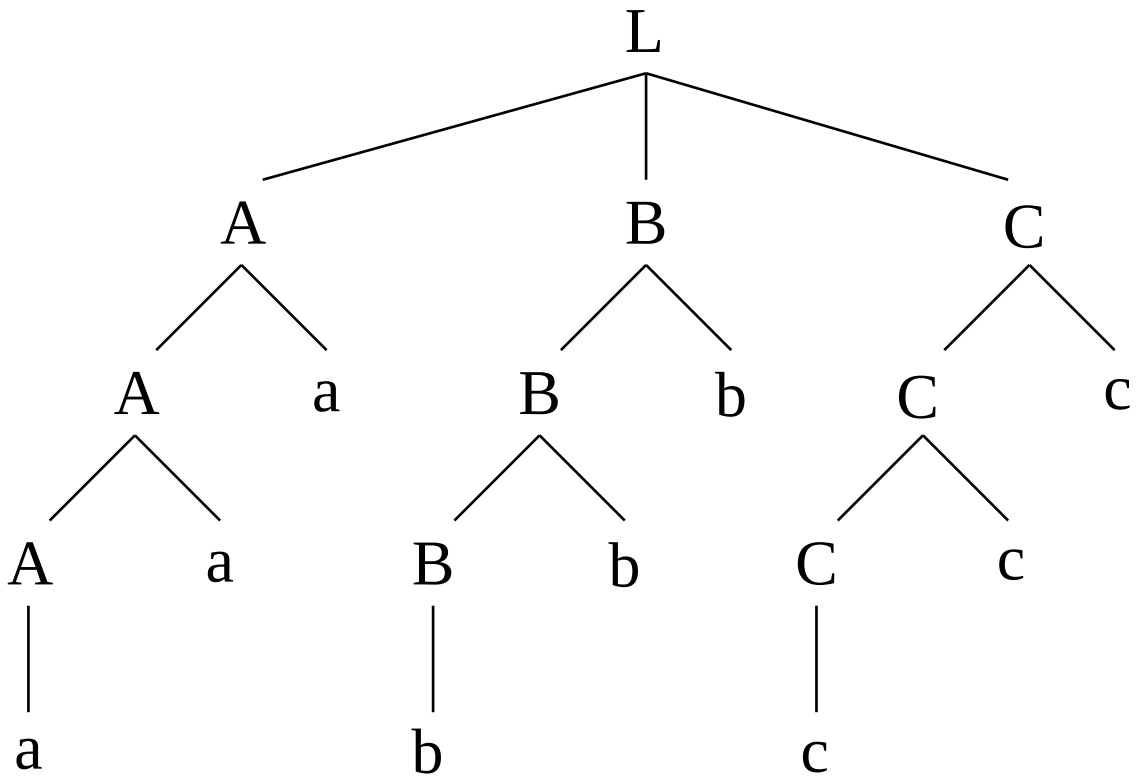
Example: write a grammar for $L = \{a^n b^n c^n \mid n \geq 1\}$.
(it is known that no context-free grammar exists)

First attempt:

```
L : A B C
A : a | A a
B : b | B b
C : c | C c
```

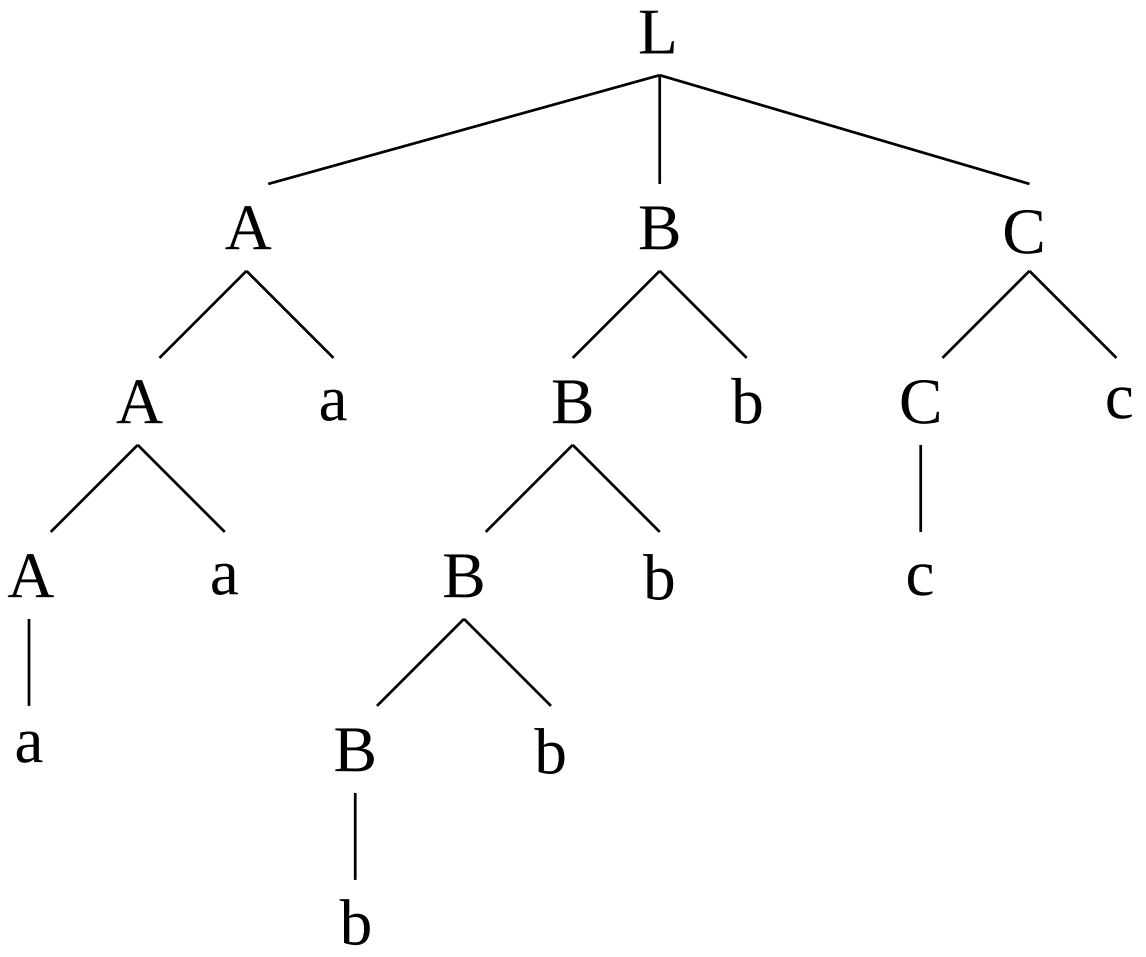
It recognizes $\{a^n b^m c^p \mid n \geq 1, m \geq 1, p \geq 1\}$.

Attribute Grammars



OK!

Attribute Grammars



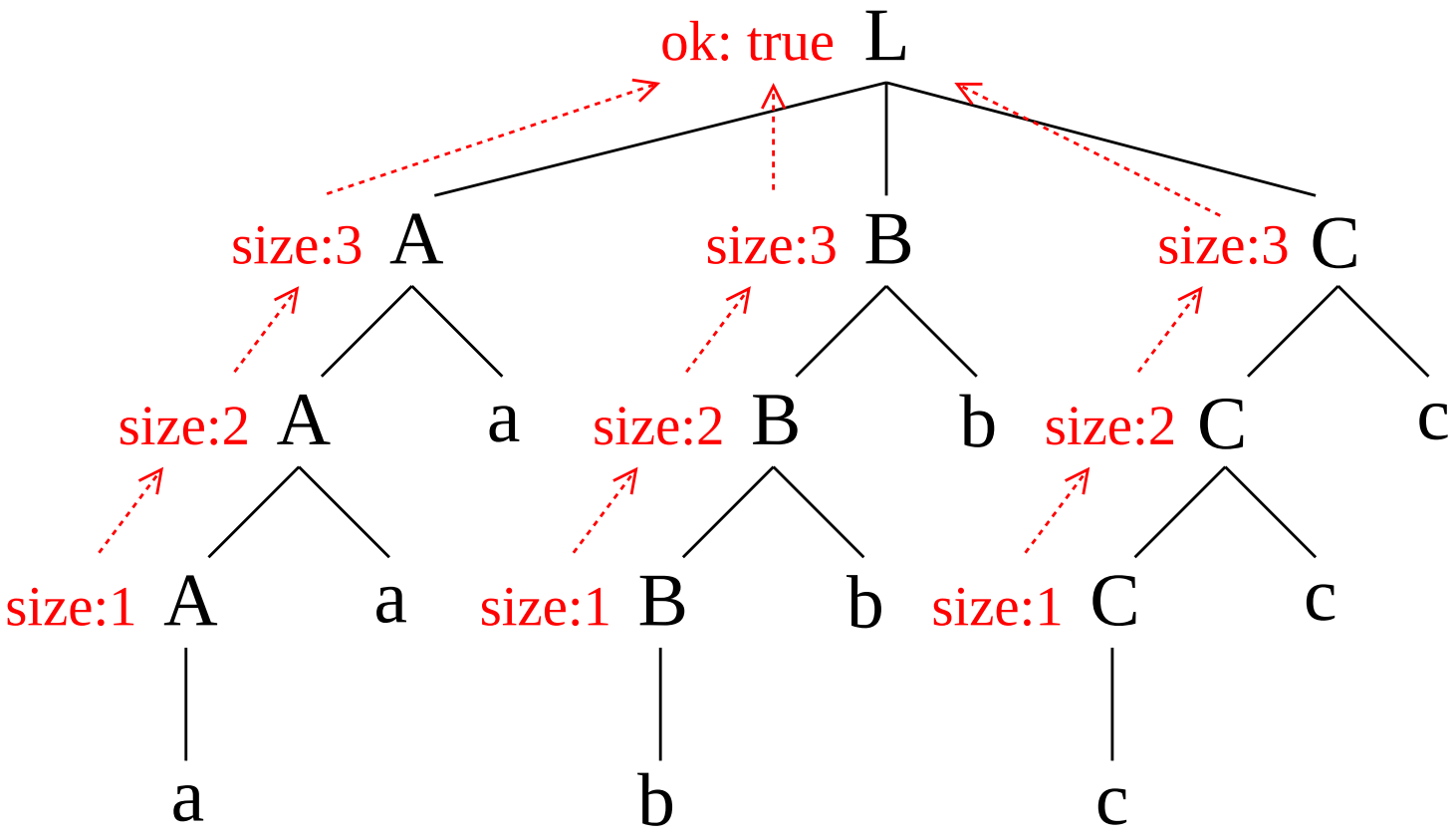
KO

Synthesized Attributes

Augment the grammar with a *synthesized* attribute associated to **A**, **B** and **C** that indicates the length of the sequence. Another synthesized attribute (**ok**) is associated to **L** to indicate that the sequence is correct.

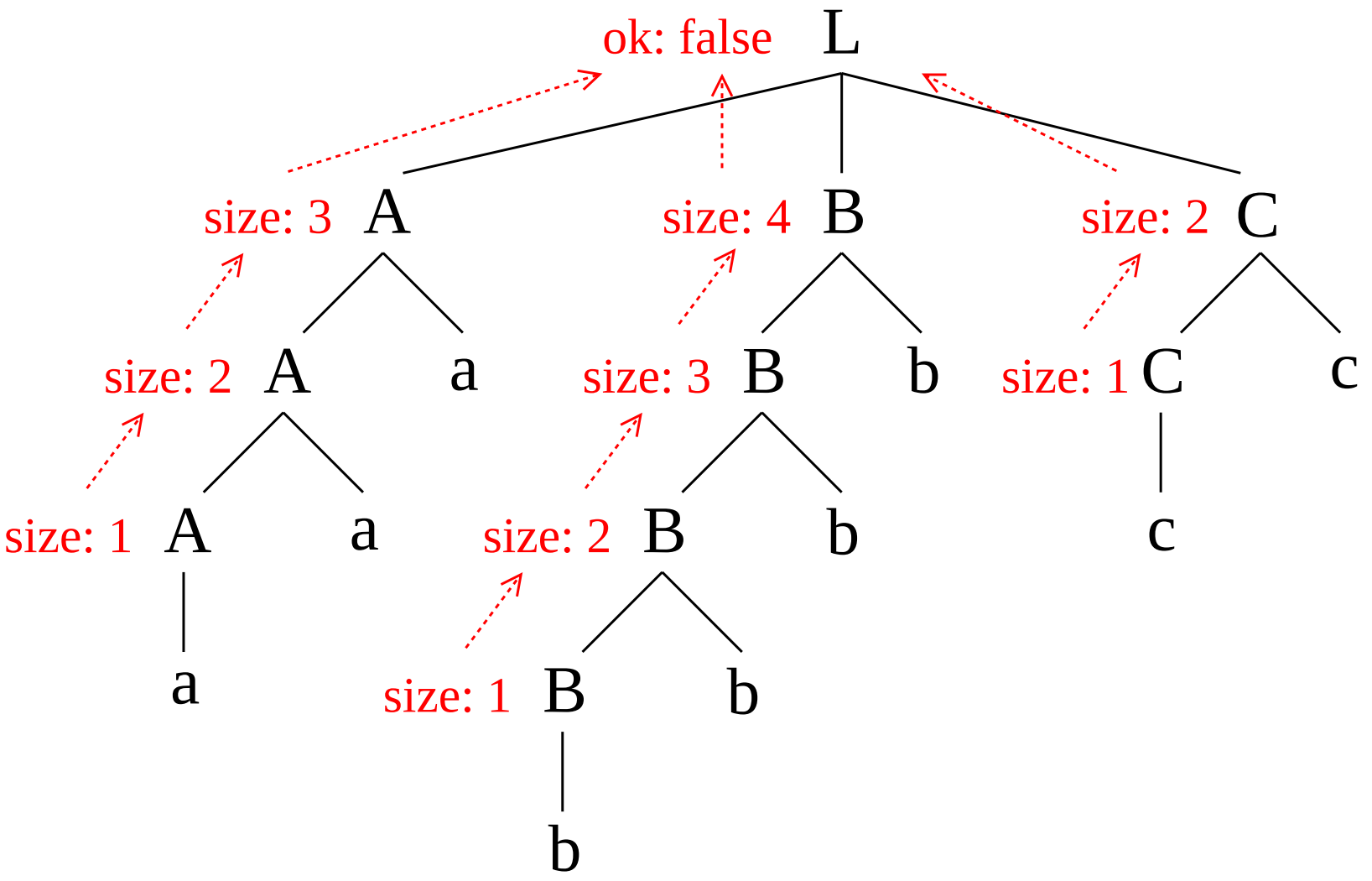
```
L : A B C { L.ok = (A.size == B.size)
                  and (A.size == C.size)}
A : a     { A.size = 1}
  | A' a  { A.size = A'.size + 1}
B : b     { B.size = 1}
  | B' b  { B.size = B'.size + 1}
C : c     { C.size = 1}
  | C' c  { C.size = C'.size + 1}
```

Synthesized Attributes



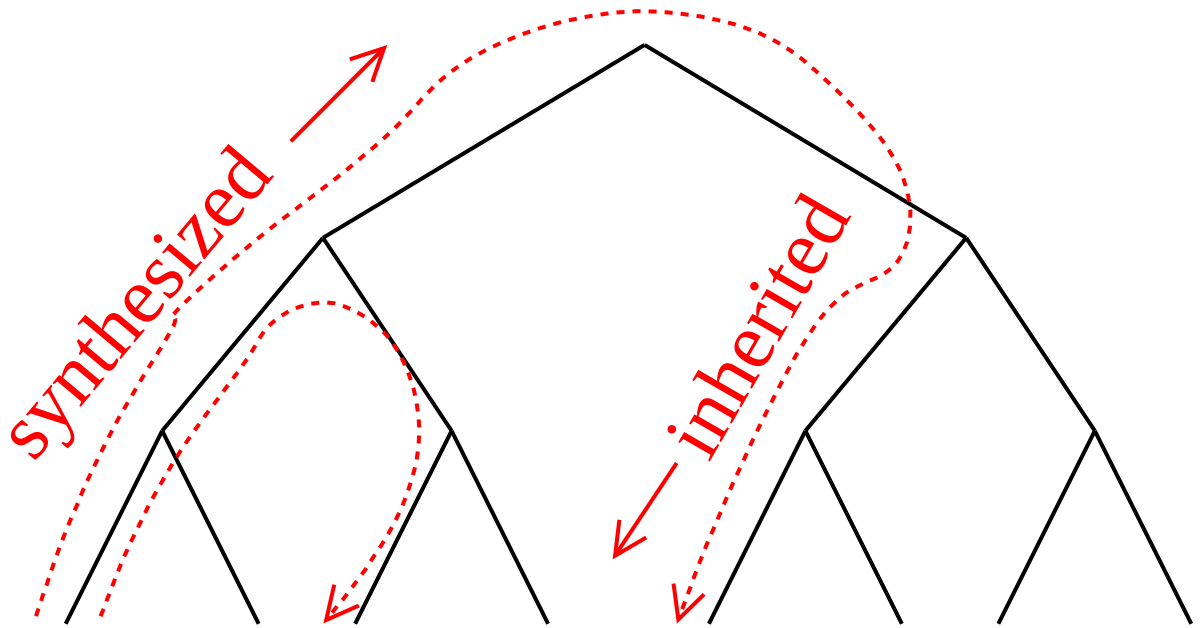
Synthesized attributes are calculated from RHS to LHS of the production rules (bottom-up in the parse tree).

Synthesized Attributes



Inherited Attributes

Sometimes it is necessary to move some information up to some node in the tree and transfer it down to some other part of the tree.



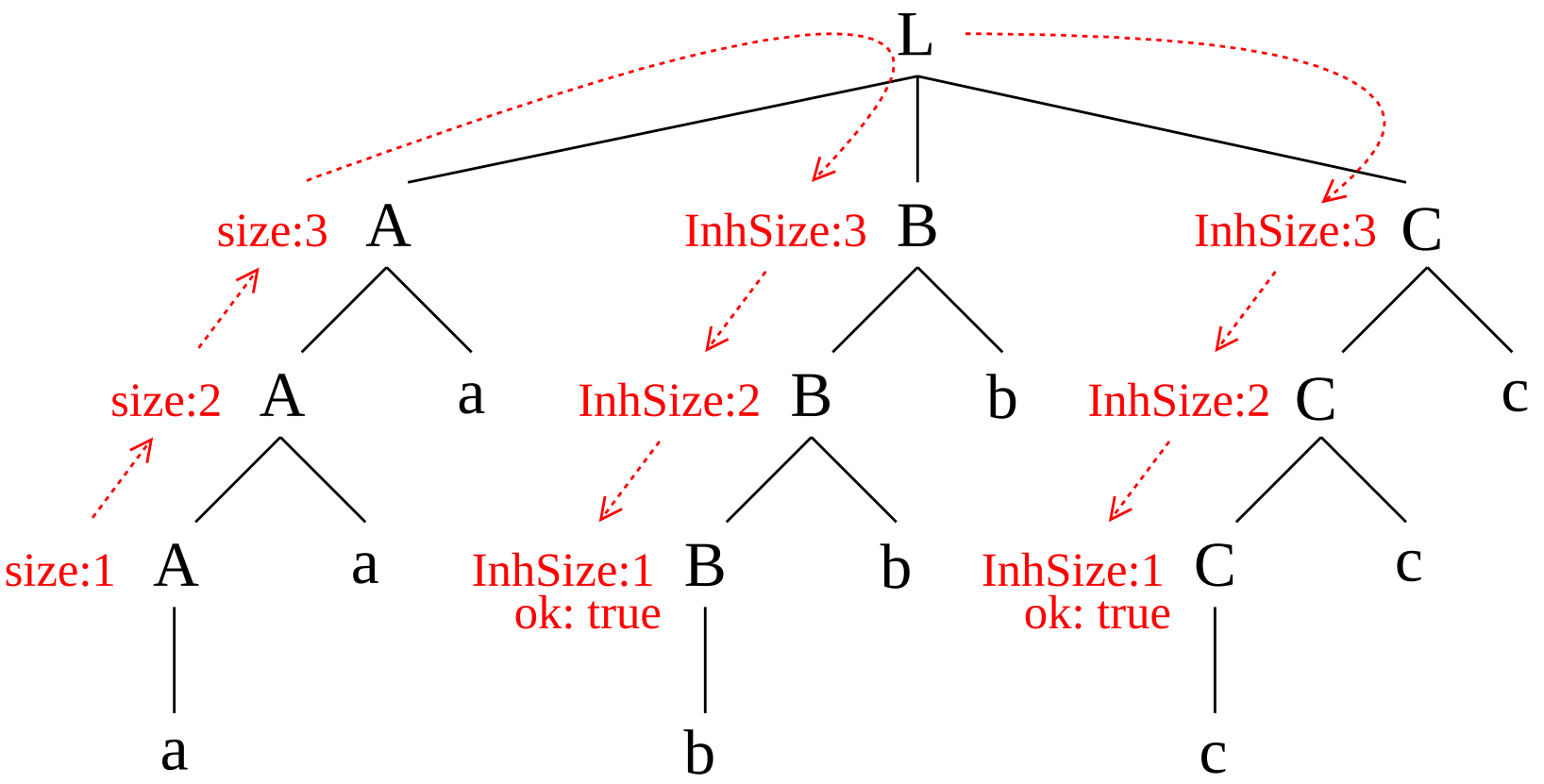
Inherited Attributes

Inherited attributes are propagated to the symbols of the RHS of the production rules (top-down or left-to-right in the parse tree).

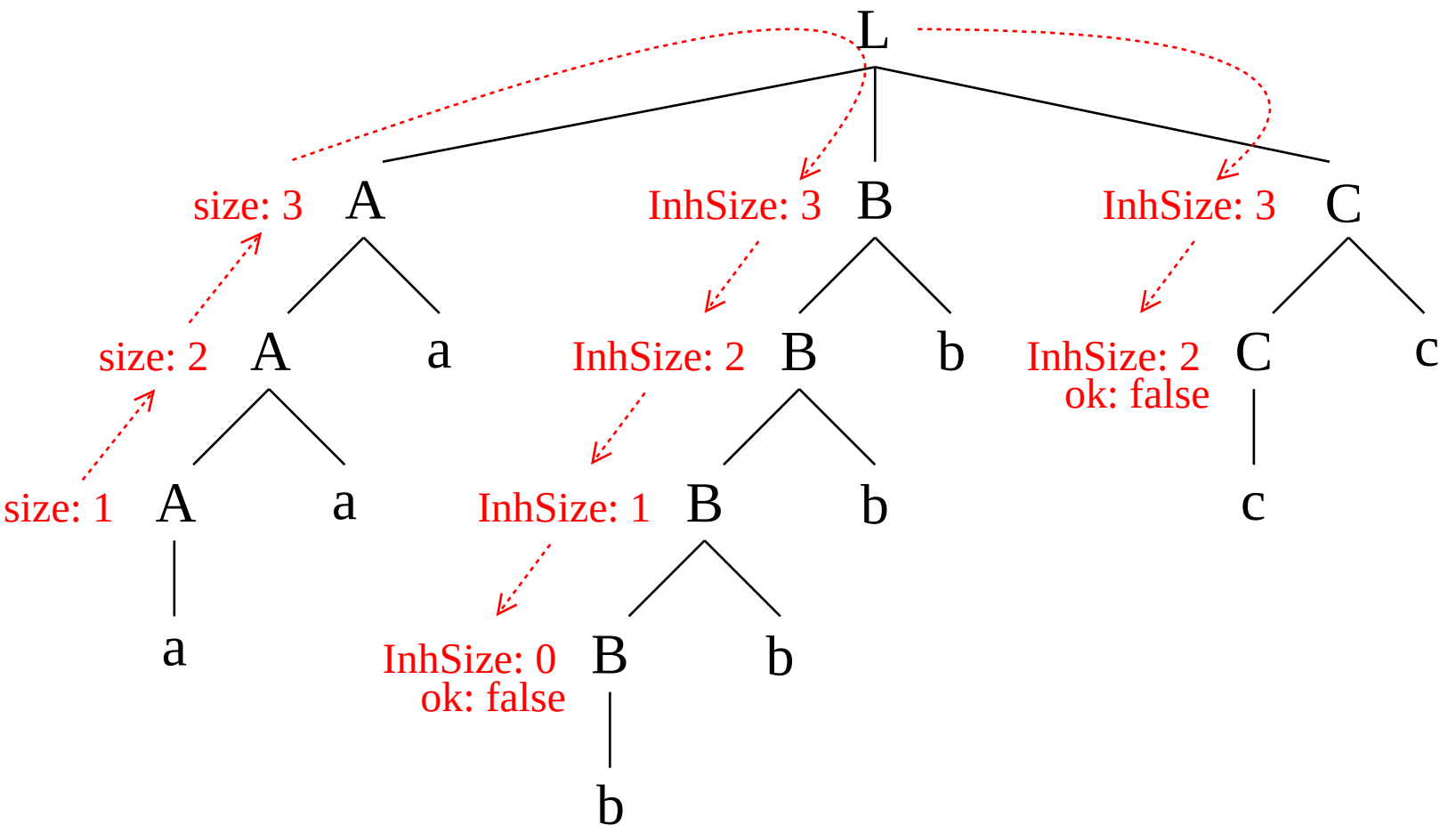
The grammar can be augmented with an inherited attribute (**InhSize**) for the symbols **B** and **C**.

```
L : A B C { B.InhSize = A.size
            C.InhSize = A.size}
A : a     { A.size = 1}
  | A' a  { A.size = A'.size + 1}
B : b     { B.ok = (B.InhSize == 1) }
  | B' b  { B'.InhSize = B.InhSize - 1}
C : c     { C.ok = (C.InhSize == 1)}
  | C' c  { C'.InhSize = C.Inhsize - 1}
```

Inherited Attributes



Inherited Attributes



Attributes in Recursive Descent Parsers

Recursive descent parsers generate a function for each non-terminal symbol.

A *synthesized* attribute is an attribute of the non-terminal symbol at the LHS of the production rule.

An *inherited* attribute is an attribute of a non-terminal symbol at the RHS of the production rule.

- Synthesized attributes are values *returned* by the functions.
- Inherited attributes are *paramaters* passed to the functions.

A simple calculator

PRODUCTION	SEMANTIC RULES
$L \to E$	$L.val = E.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to$ **digit**	$F.val =$ **digit**.*lexval*

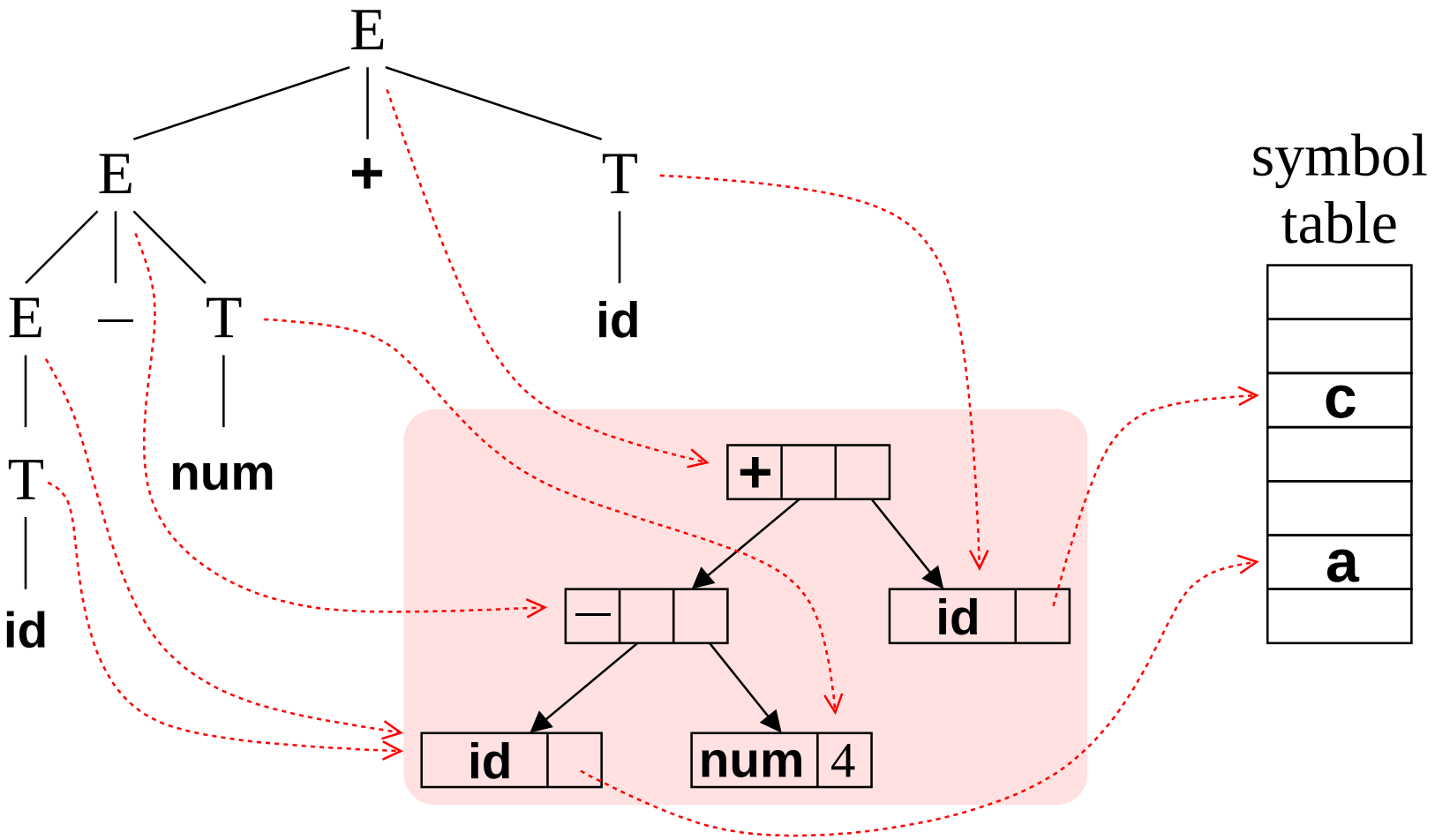
Lexical analyzers provide synthesized attributes (*lexval*) with the values of the terminal symbols.

Constructing syntax trees

PRODUCTION	SEMANTIC RULES
$E \to E_1 + T$	$E.n =$ **new** *Node*$('+', E_1.n, T.n)$
$E \to E_1 - T$	$E.n =$ **new** *Node*$('-', E_1.n, T.n)$
$E \to T$	$E.n = T.n$
$T \to (E)$	$T.n = E.n$
$T \to$ **id**	$T.n =$ **new** *Leaf*(**id**, **id**.*entry*)
$T \to$ **num**	$T.n =$ **new** *Leaf*(**num**, **num**.*val*)

Constructing syntax trees

Parse and syntax tree for: **a - 4 + c**



Abstract Syntax Trees

Parsing and Syntax Trees

Parsing decides if the program is part of the language.

Not that useful: we want more than a yes/no answer.

Like most, ANTLR parsers can include *actions*: pieces of code that run when a rule is matched.

Top-down parsers: actions executed during parsing rules.

Bottom-up parsers: actions executed when rule is "reduced."

Actions

In a top-down parser, actions are executed during the matching routines.

Actions can appear anywhere within a rule: before, during, or after a match.

```
rule { /* before */ }
  : A { /* during */ } B
  | C D { /* after */ } ;
```

Bottom-up parsers restricted to running actions only after a rule has matched.

Implementing Actions

Nice thing about top-down parsing: grammar is essentially imperative.

Action code simply interleaved with rule-matching.

Easy to understand what happens when.

Actions in ANTLR

Simple languages can be interpreted with parser actions.

```
expr : mexpr ( '+' mexpr )* EOF ;

mexpr : atom ( '*' atom )* ;

atom : INT ;
```

Actions in ANTLR

Simple languages can be interpreted with parser actions.

```
class CalcParser : public antlr4::Parser

expr returns [int r]
   : m=mexpr {$r = $m.r;}
     ('+' m=mexpr {$r += $m.r;})* EOF ;

mexpr returns [int r]
   : a=atom {$r = $a.r;}
     ('*' a=atom {$r += $a.r;})* ;

atom returns [int r]
  : i:INT {$r = stoi($i.text);} ;
```

Implementing Actions in ANTLR

```
expr returns [int r]
   : m=mexpr {$r = $m.r;}
     ('+' m=mexpr {$r += $m.r;})* EOF ;
```

```
int ExprParser::expr() {
   int r;
   int m = mexpr();
   r = m;
   while (Token == '+') {
     Match('+'); m = mexpr();
     r += m;
   }
   Match(EOF);
   return r;
}
```

Actions to build the AST

Usually, actions build a data structure that represents the program.

Separates parsing from translation.

Makes modification easier by minimizing interactions.

Allows parts of the program to be analyzed in different orders.

Actions

Bottom-up parsers can only build bottom-up data structures.

Children known first, parents later.

$\rightarrow$ Constructor for any object can require knowledge of children, but not of parent.

Context of an object only established later.

Top-down parsers can build both kinds of data structures.

What To Build?

Typically, an Abstract Syntax Tree that represents the program.

Represents the syntax of the program almost exactly, but easier for later passes to deal with.

Punctuation, whitespace, other irrelevant details omitted.

Abstract vs. Concrete Trees

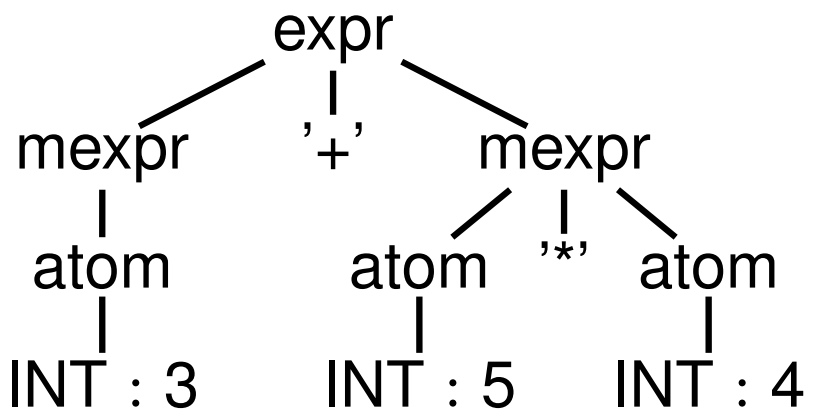
Like scanning and parsing, objective is to discard irrelevant details.

E.g., comma-separated lists are nice syntactically, but later stages probably just want lists.

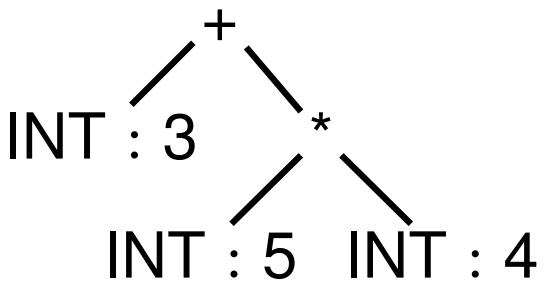
AST structure almost a direct translation of the grammar.

Abstract vs. Concrete Trees

```
expr : mexpr ('+' mexpr )* ;
mexpr : atom ('*' atom )* ;
atom : INT ;

3 + 5 * 4
```



Concrete Parse Tree

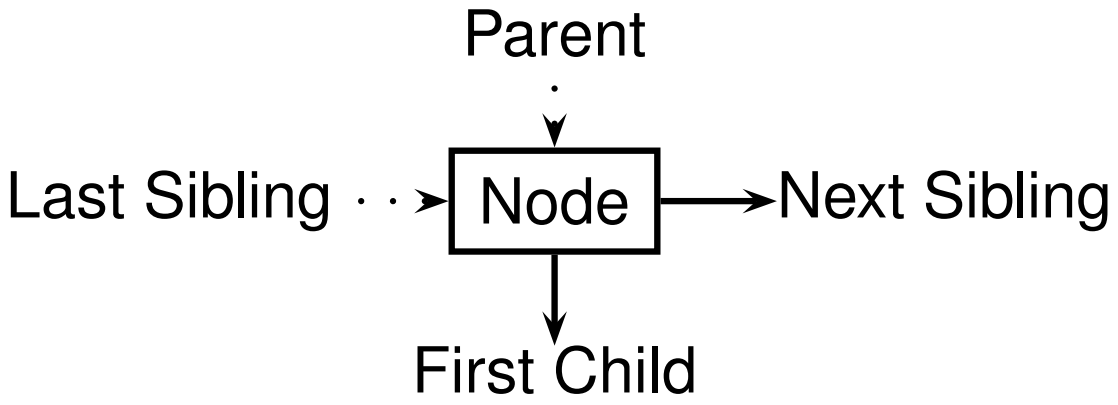


Abstract Syntax Tree

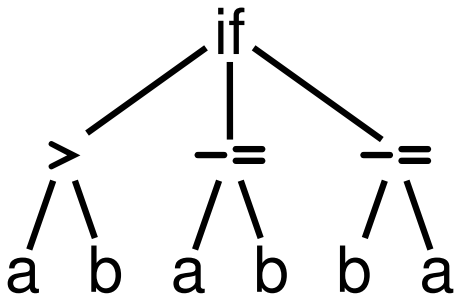
Implementing ASTs

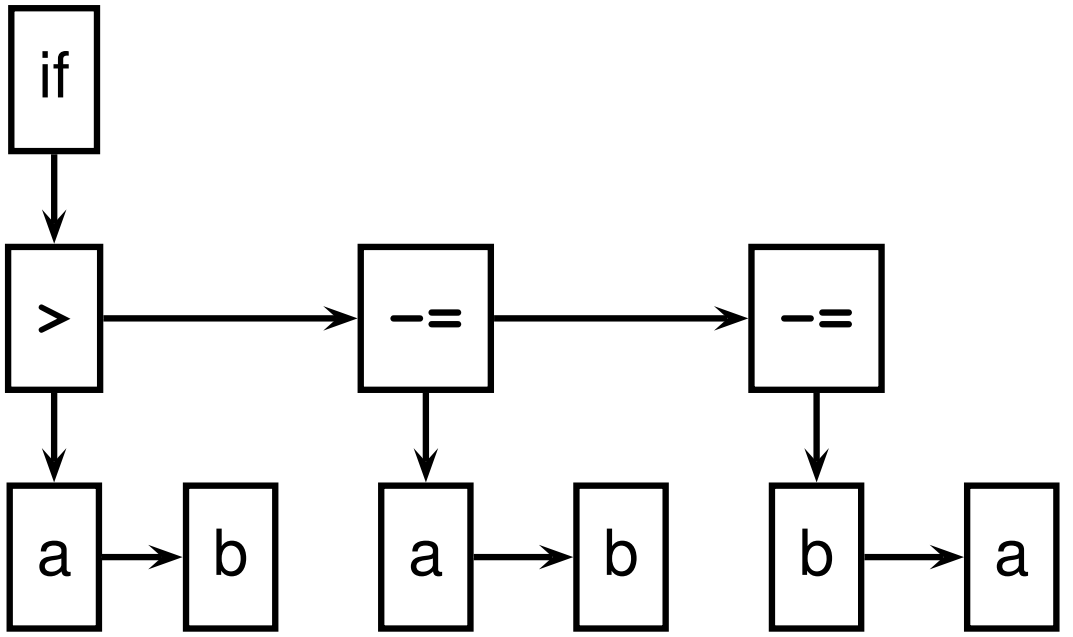
Most general implementation: ASTs are n-ary trees.

Each node holds a token and pointers to its first child and next sibling:



Example of AST structure





Comment on Generic ASTs

Is this general-purpose structure too general?

Not very object-oriented: whole program represented with one type.

Alternative: Heterogeneous ASTs: one class per object.

```
class BinOp {
  int operator; Expr left, right;
};
class IfThen {
  Expr predicate; Stmt thenPart, elsePart;
};
```

Heterogeneous ASTs

Advantage: avoid switch statements when walking tree.

Disadvantage: each analysis requires another method.

```
class BinOp {
  int operator; Expr left, right;
  void typeCheck() { ... };
  void constantProp() { ... };
  void buildThreeAddr() { ... };
};
```

Analyses spread out across class files.

Classes become littered with analysis code, additional annotations.

Comment on Generic ASTs

ANTLR offers a compromise:

It can automatically generate tree-walking code.

$\rightarrow$ It generates the big switch statement.

Each analysis can have its own file.

Still have to modify each analysis if the AST changes.

$\rightarrow$ Choose the AST structure carefully.

Building ASTs

The Obvious Way to Build ASTs

```
class ASTNode {
  ASTNode( Token t ) { ... }
  void appendChild( ASTNode c ) { ... }
  void appendSibling( ASTNode s ) { ... }
}

stmt returns [ASTNode n]
  : 'if' p=expr 'then' t=stmt 'else' e=stmt
    { n = new ASTNode(new Token("IF"));
      n.appendChild(p);
      n.appendChild(t);
      n.appendChild(e); }
  ;
```

The Obvious Way

Putting code in actions that builds ASTs is traditional and works just fine.

But it's tedious.

Designing an AST Structure

Removing unnecessary punctuation

Sequences of things

Additional grouping

How many token types?

Removing Unnecessary Punctuation

Punctuation makes the syntax readable, unambiguous.

Information represented by structure of the AST

Things typically omitted from an AST

- Parentheses

  Grouping and precedence/associativity overrides

- Separators (commas, semicolons)

  Mark divisions between phrases

- Extra keywords

  while-do, if-then-else (one is enough)

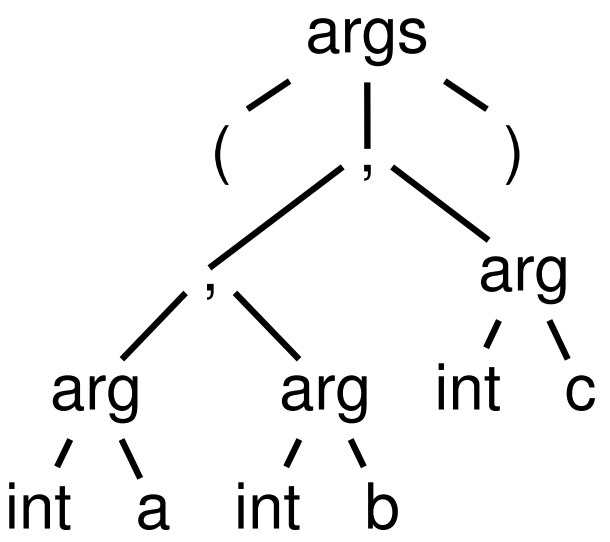
Sequences of Things

Comma-separated lists are common

```
int gcd(int a, int b, int c)
```

```
args : '(' ( arg (',' arg)* )? ')' ;
```

A concrete parse tree:



Drawbacks:

Many unnecessary nodes

Branching suggests recursion

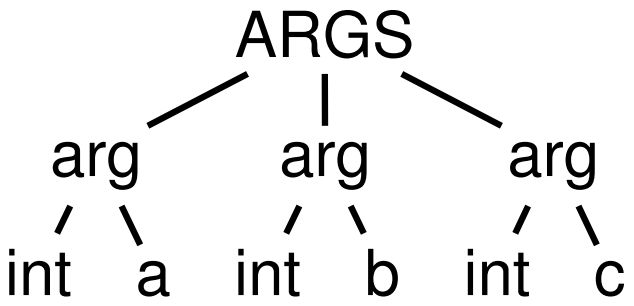
Harder for later routines to get the data they want

Sequences of Things

Better to choose a simpler structure for the tree.

Punctuation irrelevant; build a simple list.

```
int gcd(int a, int b, int c)
```



Additional Grouping

The Tiger language from Appel's book allows mutually recursive definitions only in uninterrupted sequences:

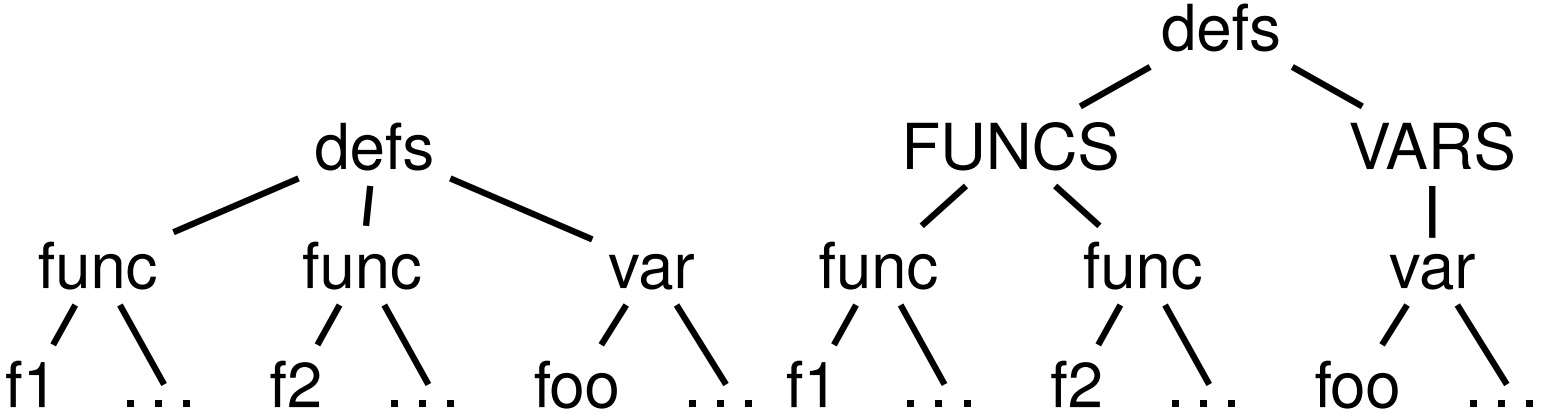
```
let
  function f1() = ( f2() )  /* OK */
  function f2() = ( ... )
in ... end

let
  function f1() = ( f2() )  /* Error */
  var foo := 42             /* splits group */
  function f2() = ( ... )
in ... end
```

Grouping

Convenient to group sequences of definitions in the AST.

Simplifies later static semantic checks.

```
let
  function f1() = ( ... )
  function f2() = ( ... )
  var foo := 42
in ... end
```



Grouping

Identifying and building sequences of definitions a little tricky in ANTLR.

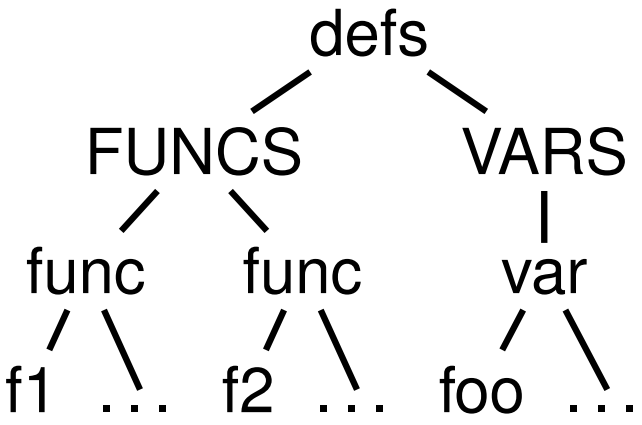
Obvious rules

```
defs  : ( funcs | vars | types )* ;
funcs : ( func )+ ;
vars  : ( var  )+ ;
types : ( type )+ ;
```

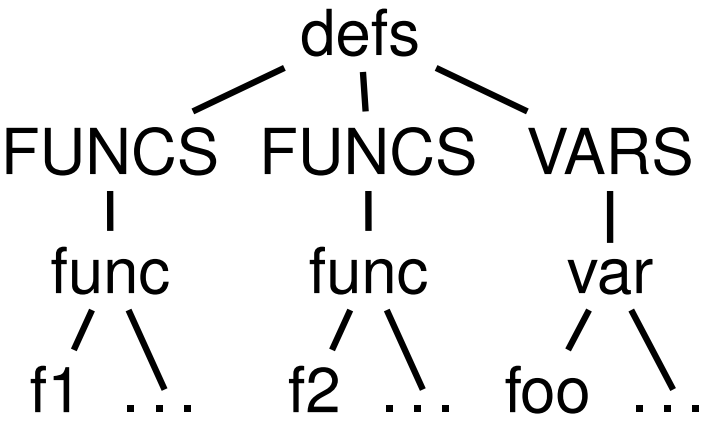
are ambiguous: Maximum-length sequences or minimum-length sequences?

Grouping

Two parse trees are possible:



Max-length sequences



Min-length sequences

Grouping

Hint: Use ANTLR's **`greedy`** option to disambiguate this.

The greedy flag decides whether repeating a rule takes precedence when an outer rule could also work.

```
string : (dots)* ;
dots : (".")+ ;
```

When faced with a period, the second rule can repeat itself or exit.

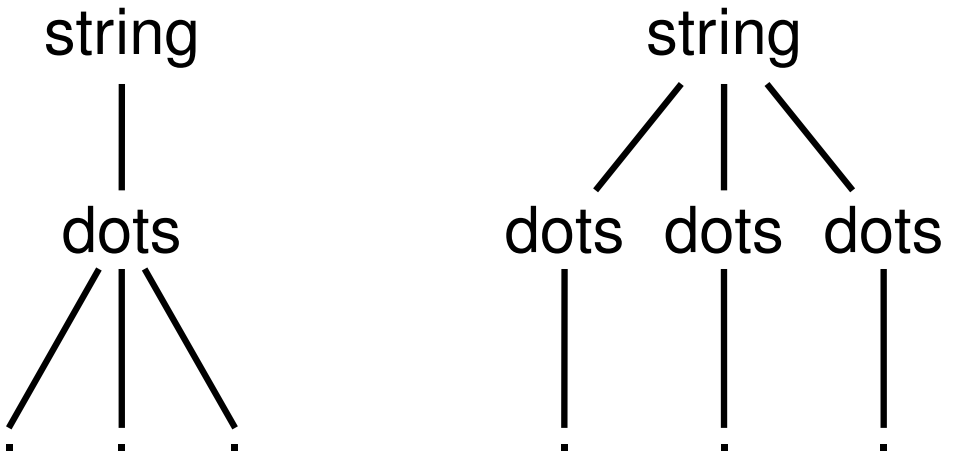
The Greedy Option

Setting greedy true makes “dots” as long as possible

```
string : (dots)* ;
dots : ( options greedy=true; : ".")+ ;
```

Setting greedy false makes each “dots” a single period

```
string : (dots)* ;
dots : ( options greedy=false; : ".")+ ;
```



How Many Types of Tokens?

Since each token is a type plus some text, there is some choice.

Generally, want each “different” construct to have a different token type.

Different types make sense when each needs different analysis.

Arithmetic operators usually not that different.

For the assignment, you need to build a node of type “BINOP” for every binary operator. The text indicates the actual operator.