- Syntax-directed translation/interpretation and translation schemes are not transparent ways of translation; they depend either on the parsing strategy and/or the order of attibute evaluation.

  Order of attribute evaluation can be different than the order of rule use in derivations (e.g., inherited attributes in a bottom-up parser or synthesized attributes in a top-down parser)

  If an AST is explicitly built, we can decouple parsing and translation. Order of semantic actions can be defined as a traversal of an AST.

- AntLR has an embedded tool for AST-based translation and/or interpretation: It is called **tree parsing**.*

*from Terrence Parr's *An Introduction to antLR.*

Formally speaking, a tree parser relates strings and trees.

```
class ExprParser extends Parser;

options {
        buildAST=true;
}

expr:   mexpr ((PLUS^|MINUS^) mexpr)*
    ;

mexpr
    :   atom (STAR^ atom)*
    ;

atom:   INT
    |   LPAREN! expr RPAREN!
    ;
```

- This grammar builds ASTs as it recognizes strings. The root of the sub-tree constructed by each rule is indicated by a suffix. ^ indicates

the root, and ! indicates that the pattern is not to be included in the sub-tree.

antLR code below produces the string and its AST:

```
import antlr.*;
import antlr.collections.*;

public class Main {
    public static void main(String[] args) throws Exception {
        ExprLexer lexer = new ExprLexer(System.in);
        ExprParser parser = new ExprParser(lexer);
        parser.expr();
        AST t = parser.getAST();
        System.out.println(t.toStringTree());
    }
}

$ java Main
3+4
 ( + 3 4 )
$ java Main
3+4*5
 ( + 3 ( * 4 5 ) )
$ java Main
(3+4)*5
```

```
 ( * ( + 3 4 ) 5 )
$
```

- The lexical analyzer and the parser are put together by the compiler designer. For example, the lexer can be:

```
class ExprLexer extends Lexer;

options {
    k=2; // needed for newline junk
    charVocabulary='\u0000'..'\u007F'; // allow ascii
}

LPAREN: '(' ;
RPAREN: ')' ;
PLUS  : '+' ;
MINUS : '-' ;
STAR  : '*' ;
INT   : ('0'..'9')+ ;
WS    : ( ' '
        | '\r' '\n'
        | '\n'
        | '\t'
        )
        {$setType(Token.SKIP);}
      ;
```

- antLR can evaluate the expression off the AST as well:

```
class ExprTreeParser extends TreeParser;

options {
    importVocab=ExprParser;
}

expr returns [int r=0]
{ int a,b; }
    :   #(PLUS  a=expr b=expr)  {r = a+b;}
    |   #(MINUS a=expr b=expr)  {r = a-b;}
    |   #(STAR  a=expr b=expr)  {r = a*b;}
    |   i:INT                   {r = (int)Integer.parseInt(i.getText(
    ;
```

- The tree constructed above is traversed by the **tree walker**. In general, rules of antLR have the template:

```
rule: alternative1
    | alternative2
   ...
    | alternativen
    ;
```

where each alternative is an antLR rule which may contain recursive specification of a tree:

```
#( root-token child1 child2 ... childn )
```

- Note that, now we have a **string grammar** *and* a **tree grammar**. These are put together in the antLR specification:

`ExprParser` works with a string grammar

`ExprTreeParser` works with a tree grammar.

```
import antlr.*;
import antlr.collections.*;

public class Main {
    public static void main(String[] args) throws Exception {
        ExprLexer lexer = new ExprLexer(System.in);
        ExprParser parser = new ExprParser(lexer);
        parser.expr();
        AST t = parser.getAST();
        System.out.println(t.toStringTree());
        ExprTreeParser treeParser = new ExprTreeParser();
        int x = treeParser.expr(t);
        System.out.println(x);
    }
}
```

Now, parsing and interpretation works together:

```
$ java Main
3+4
 ( + 3 4 )
7
$ java Main
3+(4*5)+10
 ( + ( + 3 ( * 4 5 ) ) 10 )
33
$
```

- Notice that no precedence specification is necessary when computing the result of an expression–the structure of the tree encodes this infor-

mation. That is why intermediate trees are much more than copies of the input in tree form.

The input symbols are indeed stored as nodes in the tree, but the structure of the input is encoded as the relationship of those nodes.

- Which means, if you have structural representation of expressions, you don’t need operator precedence parsing.

- AntLR can do more with trees. It can transform a tree to another tree. So, tree grammars can relate strings to trees, and trees to trees.