ANTLR 4

- Homepage: http://www.antlr4.org
MEET ANTLR

ANTLR is written in Java - so installing it is a matter of downloading the latest jar, such as antlr-4.0-complete.jar

For syntax diagrams of grammar rules, syntax highlighting of ANTLR 4 grammars, etc, use the ANTLRWorks 2 or ANTRLWorks 2 Netbeans plugin. http://tunnelvisionlabs.com/products/demo/antlrworks
What is ANTLR?

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for translating structured text.

- It's widely used to build language related tools.
- From a grammar, ANTLR generates a parser that can build and walk parse trees.
Terence Parr is the maniac behind ANTLR and has been working on language tools since 1989. He is a professor of computer science at the University of San Francisco.

Twitter search uses for example ANTLR for query parsing.
Create aliases for the ANTLR Tool, and TestRig.

$ alias antlr4='java -jar /usr/local/lib/antlr-4.0-complete.jar'
$ alias grun='java org.antlr.v4.runtime.misc.TestRig'
A First Example

Hello.g4

// Define a grammar called Hello
grammar Hello;

r : 'hello' ID ;       // match keyword hello followed by an identifier
ID : [a-z]+ ;          // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines

Then run ANTLR the tool on it:

$ antlr4 Hello.g4
$ javac Hello*.java

Now test it:

$ grun Hello r -tree
hello brink
^D
(r hello brink)
$ grun Hello r -gui
hello brink
^D

This pops up a dialog box showing that rule r matched keyword hello followed by the identifier brink.
Tokenizing:

- The process of grouping characters into words or symbols (tokens) is called lexical analysis or simply tokenizing.
- We call a program that tokenizes the input a lexer.
- The lexer group related tokens into token classes, or token types, such as INT, ID, FLOAT, etc.
- Tokens consist of at least two pieces of information: the token type and the text matched for that token by the lexer.
The parser feeds off of the tokens to recognize the sentence structure.

By default, ANTLR-generated parsers build a parse tree that records how the parse recognized the input sentence.
By producing a parse tree, a parser delivers a handy data structure to be used by the rest of the language translation application.

Trees are easy to process in subsequent steps and are well understood by programmers.

Better yet, the parser can generate parse trees automatically.
The ANTLR tool generates recursive-descent parsers from grammar rules.

Recursive-descent parsers are really just a collection of recursive methods, one per rule.

The descent term refers to the fact that parsing begins at the root of a parse tree and proceeds toward the leaves (tokens).
To get an idea of what recursive-descent parsers look like, next some (slightly cleaned up) methods that ANTLR generates from grammar rules:

```
assign : ID '=' expr ';' ; // match an assignment statement like "sp = 100;"

// assign : ID '=' expr ';' ;
void assign() { // method generated from rule assign
    match(ID); // compare ID to current input symbol then consume
    match('=');
    expr(); // match an expression by calling expr()
    match(';');
}
```
/** Match any kind of statement starting at the current input position */

stat: assign // First alternative ('|' is alternative separator)
  | ifstat // Second alternative
  | whilestat
...
;

The parser will generate:

```c
void stat() {
    switch ( «current input token» ) {
    CASE ID : assign(); break;
    CASE IF : ifstat(); break; // IF is token type for keyword 'if'
    CASE WHILE : whilestat(); break;
    ...
    default : «raise no viable alternative exception»
    }
}
```
In the example on the previous slide:

- Method stat() has to make a parsing decision or prediction by examining the next input token.
- Parsing decisions predict which alternative will be successful.
- In this case, seeing a WHILE keyword predicts the third alternative of rule().
- A lookahead token is any token that the parser sniffs before matching and consuming it.
- Sometimes, the parser needs lots of lookahead tokens to predict which alternative will succeed. ANTLR silently handles all of this for you.
Ambiguous Grammars

stat: ID '=' expr ';' // match an assignment; can match "f();"
    | ID '=' expr ';' // oops! an exact duplicate of previous alternative
    ;
expr: INT ;

stat: expr ';' // expression statement
    | ID '(' ')' ';' // function call statement
    ;
expr: ID '(' ')'
    | INT
    ;

ANTLR resolves the ambiguity by choosing the first alternative involved in the decision.
Ambiguities can occur in the lexer as well as the parser.

ANTLR resolves lexical ambiguities by matching the input string to the rule specified first in the grammar.

BEGIN : 'begin' ; // match b-e-g-i-n sequence; ambiguity resolves to BEGIN

ID : [a-z]+ ; // match one or more of any lowercase letter
Lexers process characters and pass tokens to the parser, which in turn checks syntax and creates a parse tree.

The corresponding ANTLR classes are CharStream, Lexer, Token, Parser, and ParseTree. The “pipe” connecting the lexer and parser is called a TokenStream.
ANTLR uses context objects which knows the start and stop tokens for a recognized phrase and provides access to all of the elements of that phrase.

For example, AssignContext provides methods ID() and expr() to access the identifier node and expression subtree.
Parse-Tree Listeners and Visitors

- By default, ANTLR generates a parse-tree listener interface that responds to events triggered by the built-in tree walker.

- To walk a tree and trigger calls into a listener, ANTLR’s runtime provides the class ParseTreeWalker.

- To make a language application, we write a ParseTreeListener.

- The beauty of the listener mechanism is that it’s all automatic. We don’t have to write a parse-tree walker, and our listener methods don’t have to explicitly visit their children.
ParseTreeWalker call sequence
Parse-Tree Visitors

There are situations, however, where we want to control the walk itself, explicitly calling methods to visit children.

Option -visitor asks ANTLR to generate a visitor interface from a grammar with a visit method per rule.

ParseTree tree = ... ; // tree is result of parsing

MyVisitor v = new MyVisitor();
v.visit(tree);
Let's build a grammar to recognize integers in, possibly nested, curly braces like 
{1, 2, 3} and {1, {2, 3}, 4}.

```plaintext
grammar ArrayInit;

/** A rule called init that matches comma-separated values between {...}. */
init : '{' value (',' value)* '}' ; // must match at least one value

/** A value can be either a nested array/struct or a simple integer (INT) */
value : init | INT;

// parser rules start with lowercase letters, lexer rules with uppercase
INT : [0-9]+ ; // Define token INT as one or more digits
WS : [ \t\r\n]+ -> skip ; // Define whitespace rule, toss it out
```
From the grammar ArrayInit.g4, ANTLR generates the following files:
ArrayInitParser.java This file contains the parser class definition specific to grammar ArrayInit that recognizes our array language syntax.

ArrayInitLexer.java ANTLR automatically extracts a separate parser and lexer specification from our grammar.

ArrayInit.tokens ANTLR assigns a token type number to each token we define and stores these values in this file.

ArrayInitListener.java, ArrayInitBaseListener.java By default, ANTLR parsers build a tree from the input. By walking that tree, a tree walker can fire “events” (callbacks) to a listener object that we provide.

ArrayInitListener is the interface that describes the callbacks we can implement.

ArrayInitBaseListener is a set of empty default implementations.
Brink-Van-der-Merwes-MacBook-Pro:antlr_crap brink$ grun ArrayInit init -gui
{1,{2,3},4}
Integrating a Generated Parser into a Java Program

```java
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class Test {
    public static void main(String[] args) throws Exception {
        // create a CharStream that reads from standard input
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        // create a lexer that feeds off of input CharStream
        ArrayInitLexer lexer = new ArrayInitLexer(input);
        // create a buffer of tokens pulled from the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // create a parser that feeds off the tokens buffer
        ArrayInitParser parser = new ArrayInitParser(tokens);
        ParseTree tree = parser.init(); // begin parsing at init rule
        System.out.println(tree.toStringTree(parser)); // print LISP-style tree
    }
}
```
Building a Language Application using `ArrayInitBaseListener`

```java
/** Convert short array inits like {1,2,3} to "\u0001\u0002\u0003" */
public class ShortToUnicodeString extends ArrayInitBaseListener {

    /** Translate { to " */
    @Override
    public void enterInit(ArrayInitParser.InitContext ctx) {System.out.print('"');}

    /** Translate } to " */
    @Override
    public void exitInit(ArrayInitParser.InitContext ctx) {System.out.print('"');}

    /** Convert short array inits like {1,2,3} to "\u0001\u0002\u0003" */
    @Override
    public void enterValue(ArrayInitParser.ValueContext ctx) {
        // Assumes no nested array initializers
        int value = Integer.valueOf(ctx.INT().getText());
        System.out.printf("\u%04x", value); }
}
```
Driver program for ArrayInitBaseListener

public class Translate {
    public static void main(String[] args) throws Exception {
        // create a CharStream that reads from standard input
        ANTLRInputStream input = new ANTLRInputStream(System.in);

        // create a lexer that feeds off of input CharStream
        ArrayInitLexer lexer = new ArrayInitLexer(input);

        // create a buffer of tokens pulled from the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        // create a parser that feeds off the tokens buffer
        ArrayInitParser parser = new ArrayInitParser(tokens);
        ParseTree tree = parser.init(); // begin parsing at init rule

        // Create a generic parse tree walker that can trigger callbacks
        ParseTreeWalker walker = new ParseTreeWalker();

        // Walk the tree created during the parse, trigger callbacks
        walker.walk(new ShortToUnicodeString(), tree);

        System.out.println(); // print a \n after translation
    }
}
Building a Calculator Using a Visitor

grammar LabeledExpr;

prog: stat+ ;

stat: expr NEWLINE # printExpr
| ID '=' expr NEWLINE # assign
| NEWLINE # blank
;

expr: expr op=('*'|'/') expr # MulDiv
| expr op=('+'|'-') expr # AddSub
| INT # int
| ID # id
| '(' expr ')' # parens
;

MUL : '*' ; // assigns token name to '*' used above in grammar
DIV : '/' ;
ADD : '+' ;
SUB : '-' ;
ID : [a-zA-Z]+ ; // match identifiers
INT : [0-9]+ ; // match integers
Driver class: Calc.java

LabeledExprLexer lexer = new LabeledExprLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
LabeledExprParser parser = new LabeledExprParser(tokens);
ParseTree tree = parser.prog(); // parse
EvalVisitor eval = new EvalVisitor();
eval.visit(tree);
Use ANTLR to generate a visitor interface with a method for each labeled

$ antlr4 -no-listener -visitor LabeledExpr.g4

```java
public interface LabeledExprVisitor<T> {
    T visitId(LabeledExprParser.IdContext ctx);
    T visitAssign(LabeledExprParser.AssignContext ctx);
    T visitMulDiv(LabeledExprParser.MulDivContext ctx);
    ...
}
```

To implement the calculator, we override the methods associated with statement and expression alternatives.
```java
import java.util.HashMap;
import java.util.Map;

public class EvalVisitor extends LabeledExprBaseVisitor<Integer> {
    /** "memory" for our calculator; variable/value pairs go here */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    @Override
    public Integer visitAssign(LabeledExprParser.AssignContext ctx) {
        String id = ctx.ID().getText(); // id is left-hand side of '='
        int value = visit(ctx.expr());
        memory.put(id, value);
        return value;
    }

    // etc.
}
Imagine you want to build a tool that generates a Java interface file from the methods in a Java class definition.

**Sample Input:**

```java
import java.util.List;
import java.util.Map;

public class Demo {
    void f(int x, String y) {...}
    int[ ] g( ) { return null; }
    List<Map<String, Integer>>[ ] h( ) { return null; }
}
```

**Output:**

```java
interface IDemo {
    void f(int x, String y);
    int[ ] g( );
    List<Map<String, Integer>>[ ] h( );
}
```
The key “interface” between the grammar and our listener object is called `JavaListener`, and ANTLR automatically generates it for us.

It defines all of the methods that the class `ParseTreeWalker` from ANTLR’s runtime can trigger as it traverses the parse tree.

Here are the relevant methods from the generated listener interface:

```java
public interface JavaListener extends ParseTreeListener {
    void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx);
    ...
}
```
Listener methods are called by the ANTLR-provided walker object, whereas visitor methods must walk their children with explicit visit calls.

Forgetting to invoke visit() on a node’s children means those subtrees don’t get visited.

ANTLR generates a default implementation called JavaBaseListener. Our interface extractor can then subclass JavaBaseListener and override the methods of interest.
public class ExtractInterfaceListener extends JavaBaseListener {
    JavaParser parser;
    public ExtractInterfaceListener(JavaParser parser) {this.parser = parser;}
    /** Listen to matches of classDeclaration */
    @Override
    public void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx){
        System.out.println("interface I"+ctx.Identifier()+" {");
    }
    @Override
    public void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx) {
        System.out.println("}");
    }
    /** Listen to matches of methodDeclaration */
    @Override
    public void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx)
    {
        // need parser to get tokens
        TokenStream tokens = parser.getTokenStream();
        String type = "void";
        if ( ctx.type()!=null ) {
            type = tokens.getText(ctx.type());
        } else {
            type = "void";
        }
        String args = tokens.getText(ctx.formalParameters());
        System.out.println("\t"+type+" "+ctx.Identifier()+args+";";
    }
}
Homework 6

Will be added on homepage tomorrow.

Due: 31 May