Using ANTLR In Cybersecurity

Stuart Maclean

Applied Physics Laboratory
University of Washington
stuart@apl.uw.edu

Seattle Java User Group, June 2017

Outline

Motivation, Goals

- My goal for this *work* was to auto-generate C code to be used in program execution analysis, via a technique called API-Hooking.
- My goal for this *talk* is to show how useful the ANTLR tool was in achieving my work goal, but also how useful it could be in solving problems in your domain.

What This Talk Is And Is Not

- Original idea was to present work on disk imaging infrastructure, written largely in Java. See www.osdfcon.org/2016-event. Suited to digital forensics audience.
- Recent idea was to compare/contrast my auto-generated API hook code with e.g. Cuckoo Sandbox (www.cuckoosandbox.org) and to introduce and discuss Microsoft Detours. Suited to C, program analysis audience.
- Ended up focusing on ANTLR, suited to a Java audience!

What is ANTLR?

From www.antlr.org:

> *ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.*

> *Terence Parr (University of San Francisco) is the maniac behind ANTLR and has been working on language tools since 1989.*

A Computer Science Book With Humor!

From the Definitive ANTLR Reference preface (Parr)...

> *My office mate was an astrophysicist named Kevin, who told me on multiple occasions that only physicists do real work and that programmers merely support physicists. Because all I do is build language tools to support programmers, I am at least two levels of indirection away from doing anything useful.*

and Parr's guiding principle...

> *Why program by hand in five days what you can spend five years of your life automating?*

Why Use ANTLR?

- You are learning evaluators and interpreters (REPL)
- You are building a compiler for a new language (!)
- You need to translate XML into JSON (!)
- You need to translate English into Finnish (???)

The first step of any text-processing system is recognizing the input. Instead of writing a recognizer, or *parser*, by hand, let a parser generator do the heavy lifting. ANTLR is such a tool.

ANTLR, Other Resources

Before I forget or run out of time...

- The Definitive ANTLR Reference (from which our Expr grammar was adapted).
- Enforcing Strict Model-View Separation in Template Engines (Parr).
- www.antlr.org.
- Detours: Binary Interception of Win32 Functions (Hunt,Brubacher).
- Java code for Expr parsers presented here available in a code bundle.
- For release: github.com/UW-APL-EIS/wicajo

ANTLR In Action

Create a grammar file $T.g$ for some language T. Then, run ANTLR. It produces Java source code for parsing language T:

```
$ java -cp /path/to/antlr-3.0.jar org.antlr.Tool T.g
$ ls
T.g TLexer.java TParser.java
```

Then build a test rig in e.g. `RunT.java`. Uses ANTLR-generated `TParser` class:

```
$ javac -cp /path/to/antlr-3.0.jar   RunT.java
$ java  -cp /path/to/antlr-3.0.jar:. RunT
```

ANTLR and Maven

Maven and ANTLR play great. Just reference the ANTLR plugin and runtime:

```
$ cat pom.xml

<dependency>
 <groupId>org.antlr</groupId>
 <artifactId>antlr-runtime</artifactId>
 <version>3.4</version>
</dependency>

<plugin>
 <groupId>org.antlr</groupId>
 <artifactId>antlr3-maven-plugin</artifactId>
 <version>3.4</version>
</plugin>
```

Maven's Standardized FileSystem

```
$ ls
pom.xml
src/main/antlr3/T.g
src/main/java/RunT.java

$ mvn compile
target/generated-sources/antlr3/TLexer.java
target/generated-sources/antlr3/TParser.java
target/classes/*
```

ANTLR's parser generator binds to the process-sources lifecycle phase.

A Toy Expression Language

Define a mini/toy programming language. It will contain identifiers, numbers, assignments. In our language, statements end in ';'. Valid 'programs' in this language include:

```
a = 45;
copyOfA = a;
```

To make things more interesting, we'll add four binary operators '+', '-', '*', '/', and parenthesized expressions, e.g. '(a + 4)'. Some more valid programs:

```
a = 45; sum = 4;
someCalc = 45 + (4 - 3) / ( 3 + (5 + 2) * b);
myVar = c + a * b;
```

How then to write a program to recognize all programs in this language?

A Toy Language Expressed in ANTLR (13 Lines)

```
$ cat Expr.g
grammar Expr;

// Lexer part, three token types: whitespace, numbers, identifiers
WS       : (' '|'\t'|'\n'|'\r')+ { skip(); } ;
INT      : '0' .. '9'+ ;
ID       : ('a' .. 'z' | 'A' .. 'Z')+ ;

// Parser part, five rules
prog     : stat+ ;
stat     : expr ';' | ID '=' expr ';' | ';' ;
expr     : multExpr (( '+' | '-' ) multExpr)* ;
multExpr : atom     (( '*' | '/' ) atom)* ;
atom     : INT | ID | '(' expr ')' ;
```

A Toy Language Expressed in ANTLR (13 Lines)

```
$ cat Expr.g
grammar Expr;

// Lexer part, three token types: whitespace, numbers, identifiers
WS       : (' '|'\t'|'\n'|'\r')+ { skip(); } ;
INT      : '0' .. '9'+ ;
ID       : ('a' .. 'z' | 'A' .. 'Z')+ ;

// Parser part, five rules
prog     : stat+ ;
stat     : expr ';' | ID '=' expr ';' | ';' ;
expr     : multExpr (( '+' | '-' ) multExpr)* ;
multExpr : atom     (( '*' | '/' ) atom)* ;
atom     : INT | ID | '(' expr ')' ;
```

A Toy Language Expressed in ANTLR (13 Lines)

```
$ cat Expr.g
grammar Expr;

// Lexer part, three token types: whitespace, numbers, identifiers
WS       : (' '|'\t'|'\n'|'\r')+ { skip(); } ;
INT      : '0' .. '9'+ ;
ID       : ('a' .. 'z' | 'A' .. 'Z')+ ;

// Parser part, five rules
prog     : stat+ ;
stat     : expr ';' | ID '=' expr ';' | ';' ;
expr     : multExpr (( '+' | '-' ) multExpr)* ;
multExpr : atom     (( '*' | '/' ) atom)* ;
atom     : INT | ID | '(' expr ')' ;
```

A Toy Language Expressed in ANTLR (13 Lines)

```
$ cat Expr.g
grammar Expr;

// Lexer part, three token types: whitespace, numbers, identifiers
WS       : (' '|'\t'|'\n'|'\r')+ { skip(); } ;
INT      : '0' .. '9'+ ;
ID       : ('a' .. 'z' | 'A' .. 'Z')+ ;

// Parser part, five rules
prog     : stat+ ;
stat     : expr ';' | ID '=' expr ';' | ';' ;
expr     : multExpr (( '+' | '-' ) multExpr)* ;
multExpr : atom     (( '*' | '/' ) atom)* ;
atom     : INT | ID | '(' expr ')' ;
```

A Toy Language Expressed in ANTLR (13 Lines)

```
$ cat Expr.g
grammar Expr;

// Lexer part, three token types: whitespace, numbers, identifiers
WS       : (' '|'\t'|'\n'|'\r')+ { skip(); } ;
INT      : '0' .. '9'+ ;
ID       : ('a' .. 'z' | 'A' .. 'Z')+ ;

// Parser part, five rules
prog     : stat+ ;
stat     : expr ';' | ID '=' expr ';' | ';' ;
expr     : multExpr (( '+' | '-' ) multExpr)* ;
multExpr : atom     (( '*' | '/' ) atom)* ;
atom     : INT | ID | '(' expr ')' ;
```

A Toy Language Expressed in ANTLR (13 Lines)

```
$ cat Expr.g
grammar Expr;

// Lexer part, three token types: whitespace, numbers, identifiers
WS       : (' '|'\t'|'\n'|'\r')+ { skip(); } ;
INT      : '0' .. '9'+ ;
ID       : ('a' .. 'z' | 'A' .. 'Z')+ ;

// Parser part, five rules
prog     : stat+ ;
stat     : expr ';' | ID '=' expr ';' | ';' ;
expr     : multExpr (( '+' | '-' ) multExpr)* ;
multExpr : atom     (( '*' | '/' ) atom)* ;
atom     : INT | ID | '(' expr ')' ;
```

A Toy Language Expressed in ANTLR (13 Lines)

```
$ cat Expr.g
grammar Expr;

// Lexer part, three token types: whitespace, numbers, identifiers
WS       : (' '|'\t'|'\n'|'\r')+ { skip(); } ;
INT      : '0' .. '9'+ ;
ID       : ('a' .. 'z' | 'A' .. 'Z')+ ;

// Parser part, five rules
prog     : stat+ ;
stat     : expr ';' | ID '=' expr ';' | ';' ;
expr     : multExpr (( '+' | '-' ) multExpr)* ;
multExpr : atom     (( '*' | '/' ) atom)* ;
atom     : INT | ID | '(' expr ')' ;
```

A Toy Language Expressed in ANTLR (13 Lines)

```
$ cat Expr.g
grammar Expr;

// Lexer part, three token types: whitespace, numbers, identifiers
WS       : (' '|'\t'|'\n'|'\r')+ { skip(); } ;
INT      : '0' .. '9'+ ;
ID       : ('a' .. 'z' | 'A' .. 'Z')+ ;

// Parser part, five rules
prog     : stat+ ;
stat     : expr ';' | ID '=' expr ';' | ';' ;
expr     : multExpr (( '+' | '-' ) multExpr)* ;
multExpr : atom     (( '*' | '/' ) atom)* ;
atom     : INT | ID | '(' expr ')' ;
```

ANTLR-Generated Java Code

Given Expr.g, ANTLR produces ExprLexer.java and ExprParser.java. Rules in the grammar become methods in the parser, making ANTLR a *recursive-descent* parser:

```
public class ExprParser extends DebugParser {
 // $ANTLR start "prog"
 // expr/Expr.g:37:1: prog : ( stat )+ ;
 public final void prog() throws RecognitionException {
 ... }
 // $ANTLR start "stat"
 // expr/Expr.g:40:1: stat : ( expr ';' | ID '=' expr ';' | ';' );
 public final void stat() throws RecognitionException {
 ... }
}
```

ANTLR-Generated Java Code II

Perform a quick line count:

```
$ wc -l src/main/antlr3/Expr.g
13

$ wc -l target/generated-sources/antlr3/*.java
781 ExprLexer.java
647 ExprParser.java
```

We wrote 13 lines, and ANTLR wrote 1428. That's my kind of job-share!

Alternative Output — Python

As well as Java (the default), ANTLR can produce parsers in other target languages! Thus your evaluator/compiler/translator could be in e.g. Python or C:

```
$ cat ExprPy.g

grammar ExprPy;
options {
 language=Python;
}
// rest of grammar identical to original

$ ls
target/generated-sources/antlr3/ExprPyLexer.py
target/generated-sources/antlr3/ExprPyParser.py
```

Alternative Output — C

```
$ cat ExprC.g

grammar ExprC;
options {
 language=C;
}

$ ls
target/generated-sources/antlr3/ExprCLexer.[ch]
target/generated-sources/antlr3/ExprCParser.[ch]
```

All done with a templating engine called StringTemplate. One template for each output language. Core generator logic unchanged! Other languages too, see ANTLR docs.

Testing The Expr Grammar

```
import org.antlr.runtime.*;

public class ExprRunner {
 static void parse( String input ) {
  ParseTreeBuilder ptb = new ParseTreeBuilder( "prog" );
  CharStream cs        = new ANTLRStringStream( input );
  Lexer lex            = new ExprLexer( cs );
  TokenStream tokens   = new CommonTokenStream ( lex );
  ExprParser parser    = new ExprParser( tokens, ptb );
  parser.prog();
  System.out.println( ptb.getTree().toStringTree() );
 }
}
```

Testing The Expr Grammar

```
import org.antlr.runtime.*;

public class ExprRunner {
 static void parse( String input ) {
  ParseTreeBuilder ptb = new ParseTreeBuilder( "prog" );
  CharStream cs        = new ANTLRStringStream( input );
  Lexer lex            = new ExprLexer( cs );
  TokenStream tokens   = new CommonTokenStream ( lex );
  ExprParser parser    = new ExprParser( tokens, ptb );
  parser.prog();
  System.out.println( ptb.getTree().toStringTree() );
 }
}
```

Testing The Expr Grammar

```
import org.antlr.runtime.*;

public class ExprRunner {
 static void parse( String input ) {
  ParseTreeBuilder ptb = new ParseTreeBuilder( "prog" );
  CharStream cs        = new ANTLRStringStream( input );
  Lexer lex            = new ExprLexer( cs );
  TokenStream tokens   = new CommonTokenStream ( lex );
  ExprParser parser    = new ExprParser( tokens, ptb );
  parser.prog();
  System.out.println( ptb.getTree().toStringTree() );
 }
}
```

Testing The Expr Grammar

```
import org.antlr.runtime.*;

public class ExprRunner {
 static void parse( String input ) {
  ParseTreeBuilder ptb = new ParseTreeBuilder( "prog" );
  CharStream cs        = new ANTLRStringStream( input );
  Lexer lex            = new ExprLexer( cs );
  TokenStream tokens   = new CommonTokenStream ( lex );
  ExprParser parser    = new ExprParser( tokens, ptb );
  parser.prog();
  System.out.println( ptb.getTree().toStringTree() );
 }
}
```

Testing The Expr Grammar

```
import org.antlr.runtime.*;

public class ExprRunner {
 static void parse( String input ) {
  ParseTreeBuilder ptb = new ParseTreeBuilder( "prog" );
  CharStream cs        = new ANTLRStringStream( input );
  Lexer lex            = new ExprLexer( cs );
  TokenStream tokens   = new CommonTokenStream ( lex );
  ExprParser parser    = new ExprParser( tokens, ptb );
  parser.prog();
  System.out.println( ptb.getTree().toStringTree() );
 }
}
```

Testing The Expr Grammar

```
import org.antlr.runtime.*;

public class ExprRunner {
 static void parse( String input ) {
  ParseTreeBuilder ptb = new ParseTreeBuilder( "prog" );
  CharStream cs        = new ANTLRStringStream( input );
  Lexer lex            = new ExprLexer( cs );
  TokenStream tokens   = new CommonTokenStream ( lex );
  ExprParser parser    = new ExprParser( tokens, ptb );
  parser.prog();
  System.out.println( ptb.getTree().toStringTree() );
 }
}
```

Testing The Expr Grammar

```
import org.antlr.runtime.*;

public class ExprRunner {
 static void parse( String input ) {
  ParseTreeBuilder ptb = new ParseTreeBuilder( "prog" );
  CharStream cs        = new ANTLRStringStream( input );
  Lexer lex            = new ExprLexer( cs );
  TokenStream tokens   = new CommonTokenStream ( lex );
  ExprParser parser    = new ExprParser( tokens, ptb );
  parser.prog();
  System.out.println( ptb.getTree().toStringTree() );
 }
}
```

ExprRunner In Action

```
$ java -cp myJar:antlrJar ExprRunner
1;
(<grammar prog> (prog (stat (expr (multExpr (atom 1))) ;)))

pi  = 3;
rad = 89;
dia = 2 * rad;
x   = a * (5 - (3 / 2 - 6 * z) + 27);
1   = 2;
a   = ; b = 7;
```

Running the code, we note ANTLR's great error handling. It continues after errors.

The test rig works, but nothing really happens. We want a calculator!

ExprRunner In Action

```
$ java -cp myJar:antlrJar ExprRunner
1;
(<grammar prog> (prog (stat (expr (multExpr (atom 1))) ;)))

pi  = 3;
rad = 89;
dia = 2 * rad;
x   = a * (5 - (3 / 2 - 6 * z) + 27);
1   = 2;
a   = ; b = 7;
```

Running the code, we note ANTLR's great error handling. It continues after errors.

The test rig works, but nothing really happens. We want a calculator!

ExprRunner In Action

```
$ java -cp myJar:antlrJar ExprRunner
1;
(<grammar prog> (prog (stat (expr (multExpr (atom 1))) ;)))

pi  = 3;
rad = 89;
dia = 2 * rad;
x   = a * (5 - (3 / 2 - 6 * z) + 27);
1   = 2;
a   = ; b = 7;
```

Running the code, we note ANTLR's great error handling. It continues after errors.

The test rig works, but nothing really happens. We want a calculator!

The Expr Grammar With Embedded Actions I

For the generated parser to *do* something, we add actions. These go right in the grammar file:

```
$ cat ExprActions.g

@parser::header {
import java.util.HashMap;
import java.util.Map;
}
@members {
Map<String,Integer> memory = new HashMap<>();
}

prog: stat+ ;
```

The Expr Grammar With Embedded Actions II

```
stat: expr ';'       { System.out.println( $expr.value );   }
  | ID '=' expr ';' { memory.put( $ID.text, $expr.value ); }
  | ';'
  ;

expr returns [int value]
  : e=multExpr { $value = $e.value; }
      (
        '+' e=multExpr { $value += $e.value; }
      | '-' e=multExpr { $value -= $e.value; }
      )*
  ;
```

The Expr Grammar With Embedded Actions III

```
multExpr returns [int value]
    : e=atom { $value = $e.value; }
        ( '*' e=atom { $value *= $e.value; }
        | '/' e=atom { $value /= $e.value; }
        )* ;

atom returns [int value]
    : INT { $value = Integer.parseInt( $INT.text ); }
    | ID  { Integer v = memory.get( $ID.text );
            if( v == null ) { printErr } else $value = v;
        }
    | '(' expr ')' { $value = $expr.value; } ;
```

Testing The Expr Grammar With Embedded Actions

```
import org.antlr.runtime.*;

public class ExprWithActionsRunner {
 static void parse( String input ) {
  CharStream cs             = new ANTLRStringStream( input );
  Lexer lex                 = new ExprActionsLexer( cs );
  TokenStream tokens        = new CommonTokenStream ( lex );
  ParseTreeBuilder ptb      = new ParseTreeBuilder( "prog" );
  ExprActionsParser parser = new ExprActionsParser( tokens, ptb );
  parser.prog();
 } }
```

Almost same as before. Only lexer, parser class names different. All the actions are in the ANTLR-generated code.

ExprActionsRunner In Action I

```
$ java -cp myJar:antlrJar ExprActionsRunner

> a = 5;  b = 4 * a; a; b;
5
20

> l geometry.exp
2
4
8
12
4
```

Run the demo for a clearer picture!

ExprActionsRunner In Action II

```
$ cat circle.exp
pi   = 3;
rad  = 4;
dia  = 2 * rad;
area = pi * rad * rad;
vol  = 4 / 3 * pi * rad * rad * rad;
pi; rad; dia; area; vol;

$ java -cp myJar:antlrJar ExprActionsRunner circle.exp
3
4
8
48
192
```

Tree Generation

- The embedded actions in the previous example can only go so far.
- For any moderately complex input, e.g. programming language source code, evaluating the input as you read it is infeasible.
- An intermediate form called an *abstract syntax tree* is needed. Great time to learn recursion.
- ANTLR produces these trees automagically!

The Expr Grammar With Tree Construction I

```
$ cat ExprTree.g
grammar ExprTree;

options { output=AST; }

tokens { // Dummy tokens needed for source-source translations
 PROG;
 STAT;
 PARENS;
}

prog: stat+ -> ^(PROG stat+) ;
```

The Expr Grammar With Tree Construction I

```
$ cat ExprTree.g
grammar ExprTree;

options { output=AST; }

tokens { // Dummy tokens needed for source-source translations
 PROG;
 STAT;
 PARENS;
}

prog: stat+ -> ^(PROG stat+) ;
```

The Expr Grammar With Tree Construction I

```
$ cat ExprTree.g
grammar ExprTree;

options { output=AST; }

tokens { // Dummy tokens needed for source-source translations
 PROG;
 STAT;
 PARENS;
}

prog: stat+ -> ^(PROG stat+) ;
```

The Expr Grammar With Tree Construction II

Subtree generation for rules stat, expr and multExpr:

```
// STAT dummy tokens at subtree roots. Can thus discard the ';'
stat: expr ';'           -> ^(STAT expr)
    | ID '=' expr ';'    -> ^(STAT ID '=' expr)
    | ';'                ;

expr: multExpr (( '+'^ | '-'^) multExpr)* ;

multExpr: atom (('*'^ | '/'^) atom)* ;
```

The Expr Grammar With Tree Construction II

Subtree generation for rules stat, expr and multExpr:

```
// STAT dummy tokens at subtree roots. Can thus discard the ';'
stat: expr ';'           -> ^(STAT expr)
    | ID '=' expr ';'    -> ^(STAT ID '=' expr)
    | ';'                ;

expr: multExpr (( '+'^ | '-'^) multExpr)* ;

multExpr: atom (('*'^ | '/'^) atom)* ;
```

The Expr Grammar With Tree Construction II

Subtree generation for rules stat, expr and multExpr:

```
// STAT dummy tokens at subtree roots. Can thus discard the ';'
stat: expr ';'           -> ^(STAT expr)
    | ID '=' expr ';'    -> ^(STAT ID '=' expr)
    | ';'                ;

expr: multExpr (( '+'^ | '-'^) multExpr)* ;

multExpr: atom (('*'^ | '/'^) atom)* ;
```

The Expr Grammar With Tree Construction II

Subtree generation for rules stat, expr and multExpr:

```
// STAT dummy tokens at subtree roots. Can thus discard the ';'
stat: expr ';'           -> ^(STAT expr)
    | ID '=' expr ';'    -> ^(STAT ID '=' expr)
    | ';'                ;

expr: multExpr (( '+'^ | '-'^) multExpr)* ;

multExpr: atom (('*'^ | '/'^) atom)* ;
```

The Expr Grammar With Tree Construction III

Subtree generation for atoms:

```
atom: INT
    | ID
        /*
          Discard any parenthesis source token, but root the
          new subtree with a PARENS dummy token (in our case)
        */
    | '(' expr ')' -> ^(PARENS expr)
    ;
```

Testing The Expr Grammar With Tree Construction

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

public class ExprWithTreesRunner {
 static void parse( String input ) {
  CharStream cs             = new ANTLRStringStream( input );
  Lexer lex                 = new ExprTreeLexer( cs );
  TokenStream tokens        = new CommonTokenStream ( lex );
  ParseTreeBuilder ptb      = new ParseTreeBuilder( "prog" );
  ExprTreeParser parser     = new ExprTreeParser( tokens, ptb );
  ExprTreeParser.prog_return r = parser.prog();
  Tree t = (Tree)r.getTree();
  process(t);
 }
}
```

ExprTreeRunner In Action

```
$ java -cp myJar:antlrJar ExprTreeRunner

> a = 5;  b = 4 * a;
[1]
> foo = bar + 3 * baz;
[2]
> d 2               display program 2 (via dot,png)
> e 2               emit program 2 back out as source
> load circle.exp   load a program file
[3]
> ps                list loaded programs
> w                 emit all loaded programs as source
```

ExprTreeRunner In Action

```
$ java -cp myJar:antlrJar ExprTreeRunner

> a = 5;  b = 4 * a;
[1]
> foo = bar + 3 * baz;
[2]
> d 2               display program 2 (via dot,png)
> e 2               emit program 2 back out as source
> load circle.exp   load a program file
[3]
> ps                list loaded programs
> w                 emit all loaded programs as source
```

ExprTreeRunner In Action

```
$ java -cp myJar:antlrJar ExprTreeRunner

> a = 5;  b = 4 * a;
[1]
> foo = bar + 3 * baz;
[2]
> d 2               display program 2 (via dot,png)
> e 2               emit program 2 back out as source
> load circle.exp   load a program file
[3]
> ps                list loaded programs
> w                 emit all loaded programs as source
```

ExprTreeRunner In Action

```
$ java -cp myJar:antlrJar ExprTreeRunner

> a = 5;  b = 4 * a;
[1]
> foo = bar + 3 * baz;
[2]
> d 2               display program 2 (via dot,png)
> e 2               emit program 2 back out as source
> load circle.exp   load a program file
[3]
> ps                list loaded programs
> w                 emit all loaded programs as source
```

ExprTreeRunner In Action

```
$ java -cp myJar:antlrJar ExprTreeRunner

> a = 5;  b = 4 * a;
[1]
> foo = bar + 3 * baz;
[2]
> d 2               display program 2 (via dot,png)
> e 2               emit program 2 back out as source
> load circle.exp   load a program file
[3]
> ps                list loaded programs
> w                 emit all loaded programs as source
```

ExprTreeRunner In Action

```
$ java -cp myJar:antlrJar ExprTreeRunner

> a = 5;  b = 4 * a;
[1]
> foo = bar + 3 * baz;
[2]
> d 2               display program 2 (via dot,png)
> e 2               emit program 2 back out as source
> load circle.exp   load a program file
[3]
> ps                list loaded programs
> w                 emit all loaded programs as source
```

Tree Visualization – DOT

Graphviz contains a tool called dot, which takes files in the dot format and can produce graphics, e.g. PNGs. ANTLR runtime includes a class to convert a Tree into a dot file:

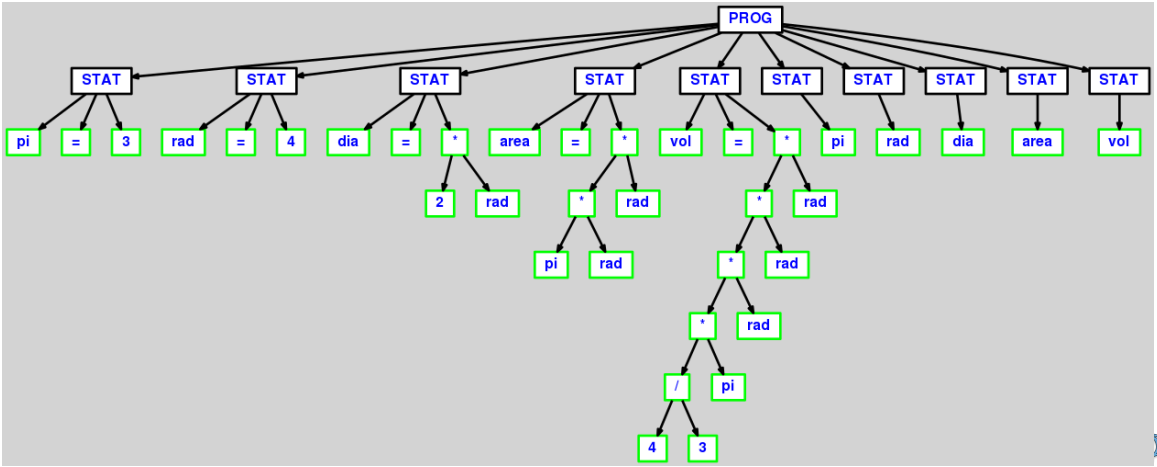
```
DOTTreeGenerator dtg = new DOTTreeGenerator();
StringTemplate st    = dtg.toDOT( someTree );
File dotFile         = new File( "someTree.dot" );
FileWriter fw        = new FileWriter( dotFile );
PrintWriter pw       = new PrintWriter( fw );
pw.println( st );
```

```
# apt-get install graphviz
$ dot someTree.dot -Tpng > someTree.png
$ display someTree.png
```

See www.graphviz.org/content/dot-language

ANTLR-Derived Tree For The Circle Expr Program: Tree-to-Dot-to-PNG

Green-colored nodes are tokens from the input stream:



ANTLR Runtime API — Trees

ANTLR grammars of the output=AST variety produce tree objects. We can then manipulate those trees, and have fun with recursion:

```
package org.antlr.runtime.tree;
public interface Tree {
 void   addChild( Tree t );
 void   deleteChild( int index );
 int    getChildCount();
 Tree   getChild( int indx );
 Tree   locateFirstChild( int type );
 int    getType();
 String getText();
 void   replaceChildren( int i, int j, Tree t );
 Tree   dupNode();
}
```

ANTLR Runtime API — Tokens

Character sequences from the input are captured in tokens. Each tree node holds a token, which has a type (NUMBER, IDENTIFIER, etc) as well as its text:

```
package org.antlr.runtime;

public interface Token {
 int    getType();
 String getText();
 void   setText( String s ); !

 int    getTokenIndex();
 int    getLine();
}
```

Tree Mutations — Some Fun With ExprTreeRunner

```
> load circle.exp
[1]
> load geometry.exp
[2]
> ps
> e 2                 see what we have
> ri  3 12            replace any number 3 with 12
> rid height seajug   rename an ID
> rp  5               replace any parened expression with 5 (sed?)
> slr                 swap leftmost, rightmost
> e 2                 see what we have now
```

Tree Mutations — Some Fun With ExprTreeRunner

```
> load circle.exp
[1]
> load geometry.exp
[2]
> ps
> e 2                 see what we have
> ri  3 12            replace any number 3 with 12
> rid height seajug   rename an ID
> rp  5               replace any parened expression with 5 (sed?)
> slr                 swap leftmost, rightmost
> e 2                 see what we have now
```

Tree Mutations — Some Fun With ExprTreeRunner

```
> load circle.exp
[1]
> load geometry.exp
[2]
> ps
> e 2                 see what we have
> ri  3 12            replace any number 3 with 12
> rid height seajug   rename an ID
> rp  5               replace any parened expression with 5 (sed?)
> slr                 swap leftmost, rightmost
> e 2                 see what we have now
```

Tree Mutations — Some Fun With ExprTreeRunner

```
> load circle.exp
[1]
> load geometry.exp
[2]
> ps
> e 2                 see what we have
> ri  3 12            replace any number 3 with 12
> rid height seajug   rename an ID
> rp  5               replace any parened expression with 5 (sed?)
> slr                 swap leftmost, rightmost
> e 2                 see what we have now
```

Tree Mutations — Some Fun With ExprTreeRunner

```
> load circle.exp
[1]
> load geometry.exp
[2]
> ps
> e 2                 see what we have
> ri  3 12            replace any number 3 with 12
> rid height seajug   rename an ID
> rp  5               replace any parened expression with 5 (sed?)
> slr                 swap leftmost, rightmost
> e 2                 see what we have now
```

Tree Mutations — Some Fun With ExprTreeRunner

```
> load circle.exp
[1]
> load geometry.exp
[2]
> ps
> e 2                 see what we have
> ri  3 12            replace any number 3 with 12
> rid height seajug   rename an ID
> rp  5               replace any parened expression with 5 (sed?)
> slr                 swap leftmost, rightmost
> e 2                 see what we have now
```

Tree Mutations — Some Fun With ExprTreeRunner

```
> load circle.exp
[1]
> load geometry.exp
[2]
> ps
> e 2                 see what we have
> ri  3 12            replace any number 3 with 12
> rid height seajug   rename an ID
> rp  5               replace any parened expression with 5 (sed?)
> slr                 swap leftmost, rightmost
> e 2                 see what we have now
```

Tree Mutations — Some Fun With ExprTreeRunner

```
> load circle.exp
[1]
> load geometry.exp
[2]
> ps
> e 2                 see what we have
> ri  3 12            replace any number 3 with 12
> rid height seajug   rename an ID
> rp  5               replace any parened expression with 5 (sed?)
> slr                 swap leftmost, rightmost
> e 2                 see what we have now
```

ANTLR Versions

- ANTLR constructs shown here apply to version 3 (quite old now).
- Other ANTLR 3 features are tree grammars and text generation via templates.
- ANTLR 4 is current version. Tree grammars (even ASTs?) deprecated in favor of parse tree listeners (??)
- I still use v3 since the C grammar I started with was a v3 document (C.g).
- New users will go with v4.

The C Programmer's Interview

So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

The C Programmer's Interview

So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

```
signal
```

The C Programmer's Interview

So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

```
signal
signal(                    )
```

The C Programmer's Interview

So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

```
signal
signal(                    )
signal(        ,           )
```

The C Programmer's Interview

So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

```
signal
signal(                 )
signal(       ,         )
signal(int sig,         )
```

The C Programmer's Interview

So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

```
signal
signal(                  )
signal(        ,         )
signal(int sig,          )
signal(int sig, void (*H)(int) )
```

The C Programmer's Interview

So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

```
        signal
        signal(                 )
        signal(       ,         )
        signal(int sig,         )
        signal(int sig, void (*H)(int) )
void (*signal(int sig, void (*H)(int) ))(int)
```

The C Programmer's Interview

So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

```
        signal
        signal(                 )
        signal(       ,         )
        signal(int sig,         )
        signal(int sig, void (*H)(int) )
void (*signal(int sig, void (*H)(int) ))(int)
void (*signal(int sig, void (*H)(int) ))(int);
```

The C Programmer's Interview

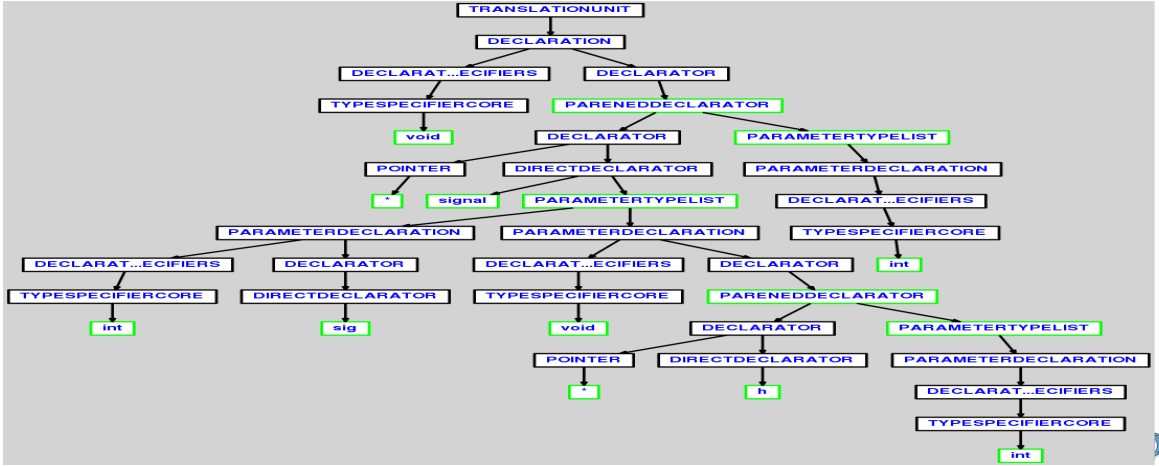
So far, have seen that we can manipulate programs in the simple Expr language using ANTLR trees. If it can be done for one language, why not others? Like C:

```
        signal
        signal(                 )
        signal(       ,         )
        signal(int sig,         )
        signal(int sig, void (*H)(int) )
void (*signal(int sig, void (*H)(int) ))(int)
void (*signal(int sig, void (*H)(int) ))(int);
```

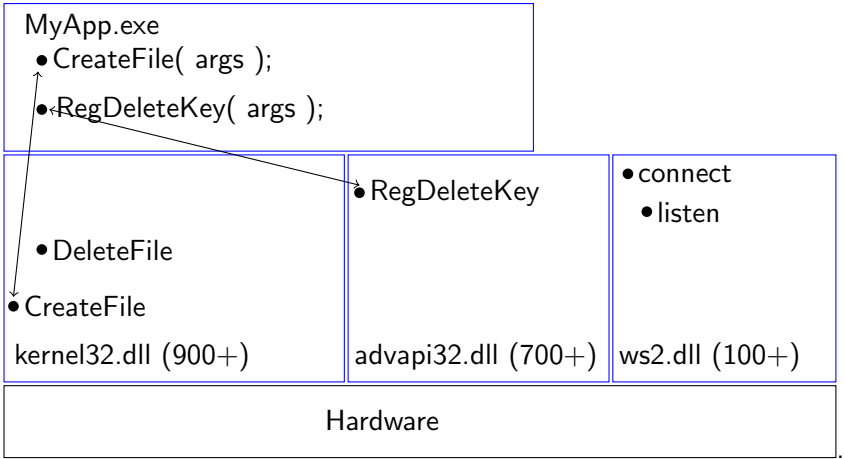
What about the tree a C compiler would build when parsing that code? Visualize that too? Moral? Grammar for C way more complex than that of Expr!

ANTLR-Derived Tree For Signal: Tree-to-Dot-to-PNG

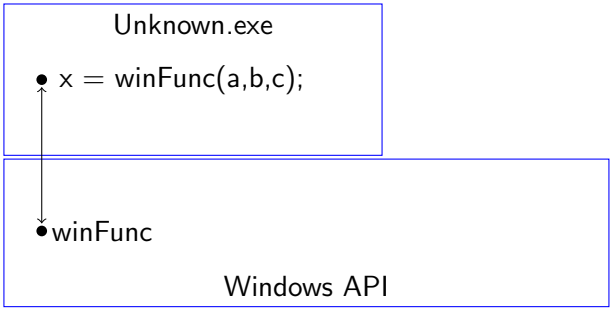
Green-colored nodes would produce output in any source-source translation:



Windows Program Execution

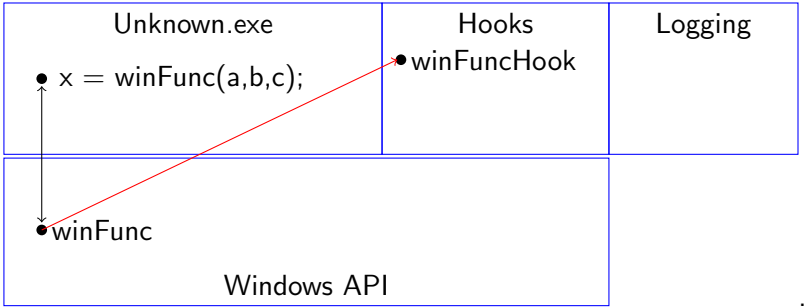


API Hooking Permits Program Monitoring



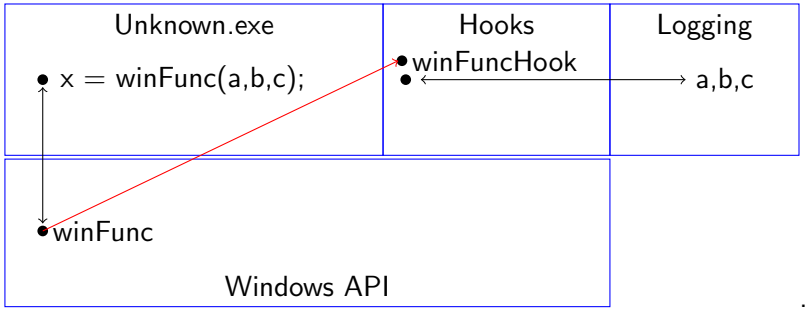
WinAPI CALL made.

API Hooking Permits Program Monitoring



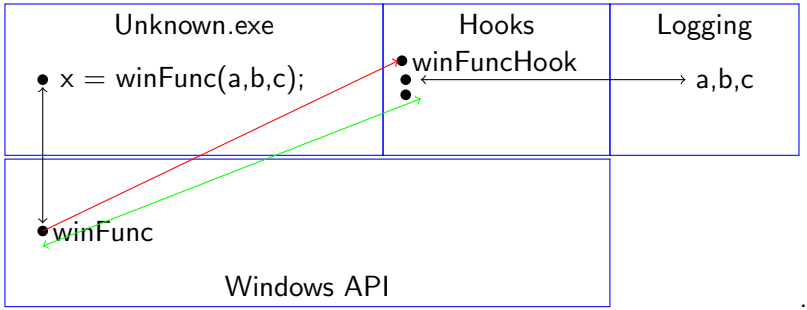
WinAPI CALL made. Hooked function JUMPs to our installed hook.

API Hooking Permits Program Monitoring



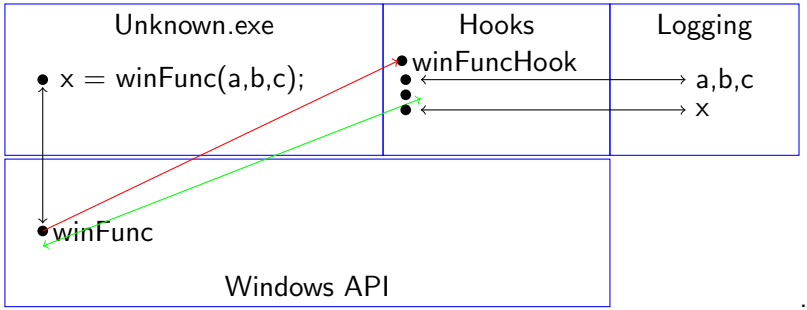
WinAPI CALL made. Hooked function JUMPs to our installed hook. The hook logs the original parameters.

API Hooking Permits Program Monitoring



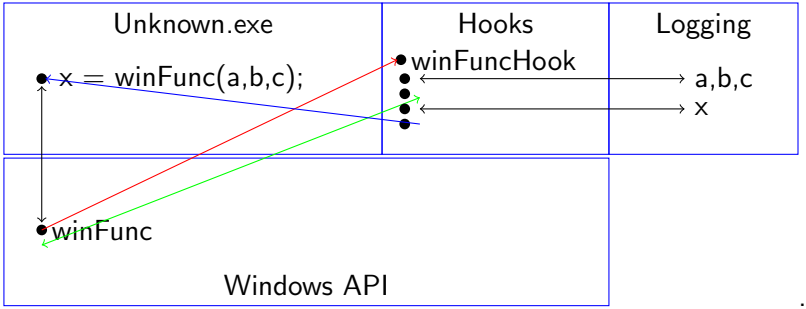
WinAPI CALL made. Hooked function JUMPs to our installed hook. The hook logs the original parameters. The hook CALLs the real function (skipping over the JUMP).

API Hooking Permits Program Monitoring



WinAPI CALL made. Hooked function JUMPs to our installed hook. The hook logs the original parameters. The hook CALLs the real function (skipping over the JUMP). The hook logs the real function's result.

API Hooking Permits Program Monitoring



WinAPI CALL made. Hooked function JUMPs to our installed hook. The hook logs the original parameters. The hook CALLs the real function (skipping over the JUMP). The hook logs the real function's result. The hook RETURNs. Due to the CALL+JUMP+RETURN, instruction pointer now back at original call site.

API Hooking Problem Statement I

Want to monitor all calls to some Windows function, say CreateFileA, taking note of the file name, access mode, etc passed in. Given this API, from windows.h:

```
HANDLE WINAPI CreateFileA(
  _In_     LPCTSTR                lpFileName,
  _In_     DWORD                  dwDesiredAccess,
  _In_     DWORD                  dwShareMode,
  _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  _In_     DWORD                  dwCreationDisposition,
  _In_     DWORD                  dwFlagsAndAttributes,
  _In_opt_ HANDLE                 hTemplateFile
);
```

API Hooking Problem Statement II

to hook that call, we have to write this new code:

```
HANDLE (WINAPI * CreateFileA_VAR)( LPCTSTR lpFileName,
                                   otherArgs ) = CreateFileA;
HANDLE WINAPI CreateFileA_HOOK( LPCTSTR lpFileName,
                                otherArgs ) {
 LOG( "In CreateFileA", lpFileName, otherArgs );
 HANDLE result = CreateFileA_VAR( lpFileName, otherArgs );
 LOG( "Result is X" );
 return result;
}
DetoursAttach( &CreateFileA_VAR, CreateFileA_HOOK ); // Only Detours call
```

Oh, and same for other 2000 functions in the Windows API!

API Hooking Problem Statement II

to hook that call, we have to write this new code:

```
HANDLE (WINAPI * CreateFileA_VAR)( LPCTSTR lpFileName,
                                   otherArgs ) = CreateFileA;
HANDLE WINAPI CreateFileA_HOOK( LPCTSTR lpFileName,
                                otherArgs ) {
 LOG( "In CreateFileA", lpFileName, otherArgs );
 HANDLE result = CreateFileA_VAR( lpFileName, otherArgs );
 LOG( "Result is X" );
 return result;
}
DetoursAttach( &CreateFileA_VAR, CreateFileA_HOOK ); // Only Detours call
```

Oh, and same for other 2000 functions in the Windows API!

API Hooking Problem Statement II

to hook that call, we have to write this new code:

```
HANDLE (WINAPI * CreateFileA_VAR)( LPCTSTR lpFileName,
                                   otherArgs ) = CreateFileA;
HANDLE WINAPI CreateFileA_HOOK( LPCTSTR lpFileName,
                                otherArgs ) {
 LOG( "In CreateFileA", lpFileName, otherArgs );
 HANDLE result = CreateFileA_VAR( lpFileName, otherArgs );
 LOG( "Result is X" );
 return result;
}
DetoursAttach( &CreateFileA_VAR, CreateFileA_HOOK ); // Only Detours call
```

Oh, and same for other 2000 functions in the Windows API!

API Hooking Problem Statement II

to hook that call, we have to write this new code:

```
HANDLE (WINAPI * CreateFileA_VAR)( LPCTSTR lpFileName,
                                   otherArgs ) = CreateFileA;
HANDLE WINAPI CreateFileA_HOOK( LPCTSTR lpFileName,
                                otherArgs ) {
 LOG( "In CreateFileA", lpFileName, otherArgs );
 HANDLE result = CreateFileA_VAR( lpFileName, otherArgs );
 LOG( "Result is X" );
 return result;
}
DetoursAttach( &CreateFileA_VAR, CreateFileA_HOOK ); // Only Detours call
```

Oh, and same for other 2000 functions in the Windows API!

API Hooking Problem Statement II

to hook that call, we have to write this new code:

```
HANDLE (WINAPI * CreateFileA_VAR)( LPCTSTR lpFileName,
                                   otherArgs ) = CreateFileA;
HANDLE WINAPI CreateFileA_HOOK( LPCTSTR lpFileName,
                                otherArgs ) {
 LOG( "In CreateFileA", lpFileName, otherArgs );
 HANDLE result = CreateFileA_VAR( lpFileName, otherArgs );
 LOG( "Result is X" );
 return result;
}
DetoursAttach( &CreateFileA_VAR, CreateFileA_HOOK ); // Only Detours call
```

Oh, and same for other 2000 functions in the Windows API!

API Hooking Problem Statement II

to hook that call, we have to write this new code:

```
HANDLE (WINAPI * CreateFileA_VAR)( LPCTSTR lpFileName,
                                   otherArgs ) = CreateFileA;
HANDLE WINAPI CreateFileA_HOOK( LPCTSTR lpFileName,
                                otherArgs ) {
 LOG( "In CreateFileA", lpFileName, otherArgs );
 HANDLE result = CreateFileA_VAR( lpFileName, otherArgs );
 LOG( "Result is X" );
 return result;
}
DetoursAttach( &CreateFileA_VAR, CreateFileA_HOOK ); // Only Detours call
```

Oh, and same for other 2000 functions in the Windows API!

API Hooking Problem Statement II

to hook that call, we have to write this new code:

```
HANDLE (WINAPI * CreateFileA_VAR)( LPCTSTR lpFileName,
                                   otherArgs ) = CreateFileA;
HANDLE WINAPI CreateFileA_HOOK( LPCTSTR lpFileName,
                                otherArgs ) {
 LOG( "In CreateFileA", lpFileName, otherArgs );
 HANDLE result = CreateFileA_VAR( lpFileName, otherArgs );
 LOG( "Result is X" );
 return result;
}
DetoursAttach( &CreateFileA_VAR, CreateFileA_HOOK ); // Only Detours call
```

Oh, and same for other 2000 functions in the Windows API!

API Hooking Solution, Partially At Least

- Adapt ANTLR's C.g grammar to do tree construction (like ExprTree.g).
- Load windows/*.h to produce (monster) trees.
- Via ANTLR's Tree API, mutate those trees to produce the new C code we need.
- Solve the LOG signature problem.

C Code Manipulation — Preparation

How to gather all the functions describing the Windows API? Do what any C programmer would do, inspect the header files. Run the preprocessor on some one-line C program, will deliver tons:

```
windows9> type grabFuncDecls.c
#include <windows.h>

windows9> cl /P /C grabFuncDecls.c
```

Now take this data to an ANTLR C parser, read grabFuncDecls.c in and transform it via tree manipulations!

Windows C as Java Objects (WICAJO)

- Idea: Use ANTLR to convert C function declarations from Windows C header files into Java objects, specifically ANTLR trees.
- Mutate those trees as needed to compose new C code with functions which are able to monitor and log program behavior.
- Compile the new functions and inject them into other programs using API hooking technologies, e.g. Microsoft Detours.
- Collect the logs to infer program execution patterns.

Also applicable to e.g. Linux but not sure if a Detours equivalent exists?

C Code Manipulation — WICAJO Shell

As per our interactive ExprTree runner, only applied to C programs, not Expr programs:

```
$ wicajosh -d grabFuncDecls.i          -d = no dot files, too many funcs?
> fs          list loaded functions
> ts          list loaded typedefs
> df F        display tree for func F
> dt T        display tree for typedef T
> pf F        a function pointer for F
> rv F        a return variable  for F
> rvr F       a resolved return variable for F
> pd N F      info on Nth argument to F
> pdr N F     resolved info on Nth argument to F
```

C Code Manipulation — WICAJO API I

The WICAJO shell makes use of the WICAJO API, Java classes representing C function declarations. The API, with example return values for the CreateFileA Windows function:

```
public class FunctionDeclaration {
 FunctionDeclaration( org.antlr.runtime.tree.Tree t );

 String pointer( String s ); -> "HANDLE (WINAPI * CreateFileA" + s + ")
                                (LPCTSTR lpFileName,
                                 otherArgs )"
 String text( String s );    -> "HANDLE WINAPI " +s+ " (LPCTSTR lpFileName,
                                                        otherArgs )"
 String result( String s );  -> "HANDLE " + s
 String args();              -> "lpFileName, otherArgs"
}
```

C Code Manipulation — WICAJO API II

We also need extraction of info from each function parameter:

```
public class ParameterDeclaration {
 ParameterDeclaration( org.antlr.runtime.tree.Tree t );
 String type()   -> "LPCTSTR"
 String name()   -> "lpFileName"

 String start()  -> "lpFileName"
 String length() -> "strlen(lpFileName)"
}
```

The last two calls return the C code snippets we would pass to a logging call — where to log from (start) and how many bytes (length). Note how WICAJO has inferred, via (recursive!) typedef resolution, that lpFileName is really a string, though its given type was LPCTSTR.

C Code Manipulation — Typedefs, Arghhh!

Windows headers make frequent use of typedefs:

```
typedef char CHAR;
typedef CHAR* LPCSTR;

int someFunc( int a, LPCSTR b, float c );
```

In order to log argument b properly in the hook for someFunc, we'd have to *resolve* all argument types to their native C types. In the case above, the byte count should be strlen(b).

C Code Manipulation — Typedefs, Done!

The typedef resolution of parameter declarations (and of any return value) can be visualized in the WICAJO interactive shell, e.g.

```
$ wicajosh
> l signal.c
> l signaltd.c
> ef 1        vanilla signal function,     named signal1
> ef 2        signal defined via typedefs, named signal2
> pd  2 1     info on 2nd arg to signal1, OK
> pd  2 2     info on 2nd arg to signal2, how big?
> pdr 2 2     info on 2nd arg to signal2, now OK
```

See the earlier ParameterDeclaration class API for details.

Conclusions

Parr's guiding principle revisited...

> *Why program by hand in five days what you can spend five years of your life automating?*

I have spent five-plus years dabbling with this automated generation of C code for the purposes of API hooking and the overall goal of black-box program analysis. Likely could have done it all by hand in three (but certainly not five days!). Parsing C code? Just say no.

Seriously, ANTLR is an amazing tool and library and is great fun too.