Language Aware Development Tools

Matthew Burke/Senior Architect, LogicNets

License

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit

http://creativecommons.org/licenses/by/4.0/.

The Point

There's more than just lexical structure in the text we manipulate. We are now seeing tools acknowledge that fact.

I hope to inspire you to use these tools, improve these tools, and invent new, more sophisticated tools.

lexical — syntax of tokens

Where We're Going

- Screen Scraping 101
- JSON Diversion
- Miscellaneous Tools
- Tops
- Diff and Merge
- DIY
- Antlr
- Clang

Screen Scraping 101

```
<class name="Aves">
 <order name="Apdiformes">
  <family name="Apodidae">
   <genus name="Cypseloides">
    <species name="rothschildi">Rothschild's swift</species>
    <species name="cryptus">White-chinned swift</species>
   </genus>
   <genus name="Streptoprocne">
    <species name="rutila">Chestnut-collared swift</spcies>
   </genus>
   <genus name="Malus">
    <species name="lattneri"><![CDATA[Lattner's swift]]></species>
   </genus>
  </family>
 </order>
</class>
```

Let’s print out all the specific epithets and the common names in this data set. Often times we attack this sort of problem with regexes. (See last slide for info on scientific names.)

- What's wrong with this regex?

  <species name="(.*)">(.*)</species>

Kleene star is greedy.

```
<class name="Aves">
 <order name="Apdiformes">
  <family name="Apodidae">
   <genus name="Cypseloides">
    <species name="rothschildi">Rothschild's swift</species>
    <species name="cryptus">White-chinned swift</species>
   </genus>
   <genus name="Streptoprocne">
    <species name="rutila">Chestnut-collared swift</spcies>
   </genus>
   <genus name="Malus">
    <species name="lattneri"><![CDATA[Lattner's swift]]></species>
   </genus>
  </family>
 </order>
</class>
```

The blue text is what the previous regex matches. The darker blue is the first capture. The second capture is empty.

- Ok, let's try this one:

  <species name="(.*?)">(.*?)</species>

*? is non-greedy. Is this better?

```
<class name="Aves">
 <order name="Apdiformes">
  <family name="Apodidae">
   <genus name="Cypseloides">
    <species name="rothschildi">Rothschild's swift</species>
    <species name="cryptus">White-chinned swift</species>
   </genus>
   <genus name="Streptoprocne">
    <species name="rutila">Chestnut-collared swift</spcies>
   </genus>
   <genus name="Malus">
    <species name="lattneri"><![CDATA[Lattner's swift]]></species>
   </genus>
  </family>
 </order>
</class>
```

Well, there are still 2 problems. (*At least*.)

```
<class name="Aves">
  <order name="Apdiformes">
    <family name="Apodidae">
      <genus name="Cypseloides">
        <species name="rothschildi">Rothschild's swift</species>
        <species name="cryptus"><![CDATA[White-chinned swift]]></species>
      </genus>
      <genus name="Streptoprocne">
        <species name="rutila">Chestnut-collared swift</species>
      </genus>
      <!-- <genus name="Malus">
        <species name="lattneri">Lattner's swift</species>
      </genus> -->
    </family>
  </order>
</class>
```

Doesn’t handle CDATA, doesn’t realize that Lattner’s Swift is commented out. Fragile. What happens if we don’t control the data source and somebody adds another attribute to the species tag (or worse, adds it intermittently).

- xmllint -xpath

  xmllint -xpath “//species/text()” document.xml

- xmlstarlet

  xmlstarlet sel -t -m ‘//species/text()’ -v ‘.’ -n document.xml

I had been using xmllint -xpath, but recently found xmlstarlet—-much more powerful/flexible.

```
rothschildi Rothschild's swift
cryptus White-chinned swift
rutila Chestnut-collared swift
```

Just to be clear, here's the output we're after. No *Malus lattneri*.

JSON Diversion

```
[
  {
    "sha": "f7a2af70526ab632008bb646d2b0e82f5c1724c9",
    "commit": {
      "author": {
        "name": "Some Body",
        "email": "somebody@gmail.com",
        "date": "2014-09-23T00:11:07Z"
      },
      "committer": {
        "name": "Some Body",
        "email": "somebody@gmail.com",
        "date": "2014-09-23T00:14:46Z"
      },
      "message": "Properly handle when objects cannot be folded\n\nFix #579.",
      "tree": {
        "sha": "45538f39af22cb0b8fd86074731f59fa2a7ff342",
        "url": "https://api.github.com/repos/stedolan/jq/git/trees/
45538f39af22cb0b8fd86074731f59fa2a7ff342"
      },
      "url": "https://api.github.com/repos/stedolan/jq/git/commits/
f7a2af70526ab632008bb646d2b0e82f5c1724c9",
      "comment_count": 0
    },
    "url": "https://api.github.com/repos/stedolan/jq/commits/
f7a2af70526ab632008bb646d2b0e82f5c1724c9",
    "html_url": "https://github.com/stedolan/jq/commit/
f7a2af70526ab632008bb646d2b0e82f5c1724c9",
```


Of course, XML isn't the only data format.

```
jq '.[] | {message: .commit.message,
name: .commit.committer.name}'
```

jq — sed for JSON data

```
{
  "message": "Properly handle when objects cannot be folded\n\nFix #579.",
  "name": "Some Body"
}
{
  "message": "Drop the jq version directory from search path",
  "name": "Any Body"
}
{
  "message": "Never close stdin; allow multiple `-` arguments",
  "name": "Any Body"
}
{
  "message": "Handle invalid inputs in argument files (fix #562)",
  "name": "Some Body"
}
{
  "message": "Properly handle incomplete json when input is file\n\nFix #562",
  "name": "Some Body"
}
```


By using a tool that “groks” JSON, we can easily filter and reformat.

Miscellaneous Tools

- Ack, Ag, PlistBuddy
- general purpose tools restricted to development context
- e.g. ignore subversion, git and other VCS directories/non-source files

Ok, so maybe PlistBuddy isn’t all that general, but it is another good example of leveraging understanding of the data format.

```
ack -type=lua —nogroup -h -o "\w*db:\w*"

    | awk -F: '{ print $2}' | sort | uniq
```

For the curious: *Ack defines Lua files as files either whose name ends in* .lua *or having a first line that matches* /^#!.*\blua(jit)?/

ack's -o option is particularly handy, it limits the output to just the matched text. Using the displayed command line, I was able to easily get a list of all distinct db:* functions called in a set of several hundred files.

BTW, ack recognizes 71 types of files (actionscript to yaml) and you can add your own

New tricks for old tools:

1. highlighting: unique colors for identifiers
2. better feedback: playgrounds, light table, jsfiddle.net, Xcode plugins, etc.

Question the status quo. Why do we display data the way we do? E.g. syntax highlighting, clearly (I think) coloring the text is useful, but are we displaying the most important/highest priority information? Check out the color identifiers mode for Emacs (link at the end) and the article its README references for a different take on how we might color source code.

```c
#include "lua.h"
#include "lauxlib.h"
#include "blink1-lib.h"
#include "lblink.h"

#define BLINK1_ERR (-1)
#define PATTERNPLAY_START 1
#define PATTERNPLAY_STOP 0

// @fixme why are these #defines and the others are static const8?
#define BADDEVSPEC_MSG "Must be either an integer in [0, n-1] (where n is the number of attached devices) or a valid seria
#define BADDEVID_MSG "ID must be in [0,n-1] where n is the number of attached devices."
#define NODEV_MSG "No blink(1) devices attached."
#define IDOPENERR_MSG "Could not open blink(1) with id %d."
#define SERIALOPENERR_MSG "Could not open blink(1) with serial #%s."

static const char *BLINK_TYPENAME = "net.bluedino.Blink1";
static const char *VID_KEY = "VID";
static const char *PID_KEY = "PID";
static const char *VERSION_KEY = "_VERSION";
static const char *DEVID_KEY = "devid";
static const char *SERIALNUM_KEY = "serial";
static const char *MARK_KEY = "mark";
static const int DEFAULT_PAUSE = 100;

const char *LUABLINK_VERSION = "0.6.1";

typedef struct blinker {
  blink1_device *device;
} blinker;
```

identifiers get their own color (*more or less*)

Tops

Comes with every Mac.

- in-place substitutions on Obj-C
- supports the following:
    1. replace
    2. replacemethod

```objc
- (void)doSomethingTo:(NSObject *)obj andExplode:(BOOL)val
{
  [self doSomething:obj];
  if (val) {
    [obj explode];
  }
}


- (void)selfdestruct
{
  [self doSomethingTo:self andExplode:YES];
}


- (void)explodeLater
{
  SEL aSEL = @selector(doSomethingTo:andExplode:);
}
```

This example is slightly frivolous. The thing to focus on is how easily we can manipulate information in multiple contexts.

```
replacemethod "doSomethingTo:<o> andExplode:<flag>"
         with "doSomethingTo:<o> andPersevere:<flag>"
{
   replace "<flag_arg>" with "!<flag_arg>"
   replace "<o_type>" with "(id<Explodable>)"
   replace "<receiver>" with "[<receiver> mutableCopy]"
}
within ("<implementation>")
{
   replace "<flag_param>" with "!<flag_param>"
}
```

Let’s change the selector, and adjust parameters at the call site (i.e. we need to negate the flag argument because the logic has changed), and change argument types, and adjust the implementation.

```
- (void)doSomethingTo:(id<Explodable>)obj andPersevere:(BOOL)val
{
  [self doSomething:obj];
  if (!val) {
    [obj explode];
  }
}


- (void)selfdestruct
{
  [[self mutableCopy] doSomethingTo:self andPersevere:!YES];
}


- (void)explodeLater
{
  SEL aSEL = @selector(doSomethingTo:andPersevere:);
}
```

And notice not only do we pick up the method definition and call, but also where we just reference the selector.

Diff and Merge

```java
public class BaseSample {

  public String method1()
  {
    return "method1";
  }

  public String method2()
  {
    return "method2";
  }
}
```

```java
public class Sample {

   public String method1()
   {
      return "method1";
   }

   public String inBetween()
   {
      return "inbetween";
   }

   public String method2()
   {
      return "method2";
   }
}
```

```java
public class ModSample {

   public String method2()
   {
      String result = "method2";
      return result;
   }


   public String inBetween()
   {
      return "inbetween";
   }


   public String method1()
   {
      return "method1";
   }

}
```

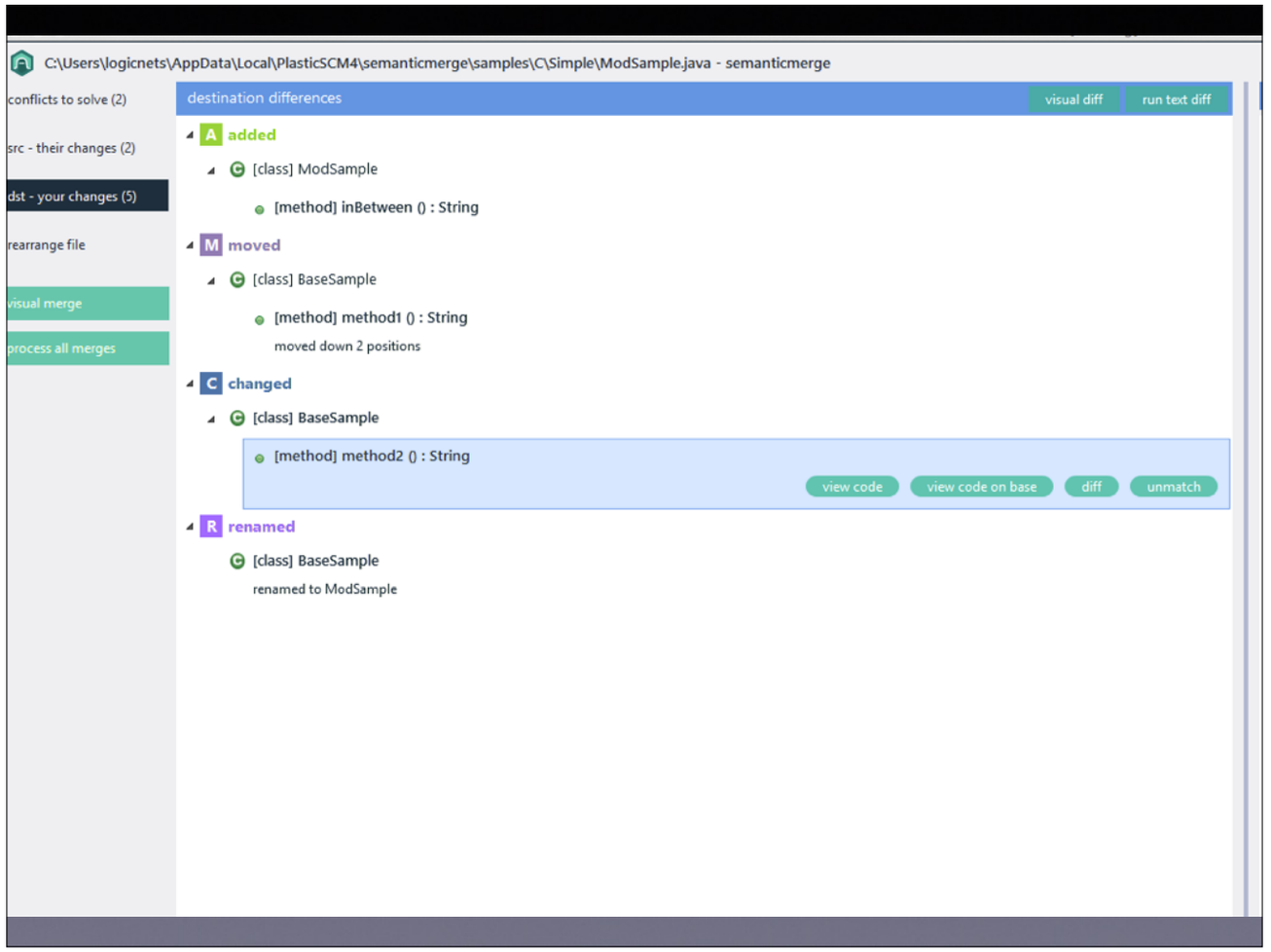
```
2c2
< public class ModSample {
---
> public class Sample {
4c4
<   public String method2()
---
>   public String method1()
6,7c6
<       String result = "method2";
<       return result;
---
>       return "method1";
9,11d7
<
<
<
18,23c14
<
<
<
<
<
<   public String method1()
---
>   public String method2()
25c16
<       return "method1";
---
>       return "method2";
27d17
<
```

Raise your hand if this has happened to you: a few simple changes and you get a wildly complicated diff listing that’s dozens, if not hundreds of lines long.

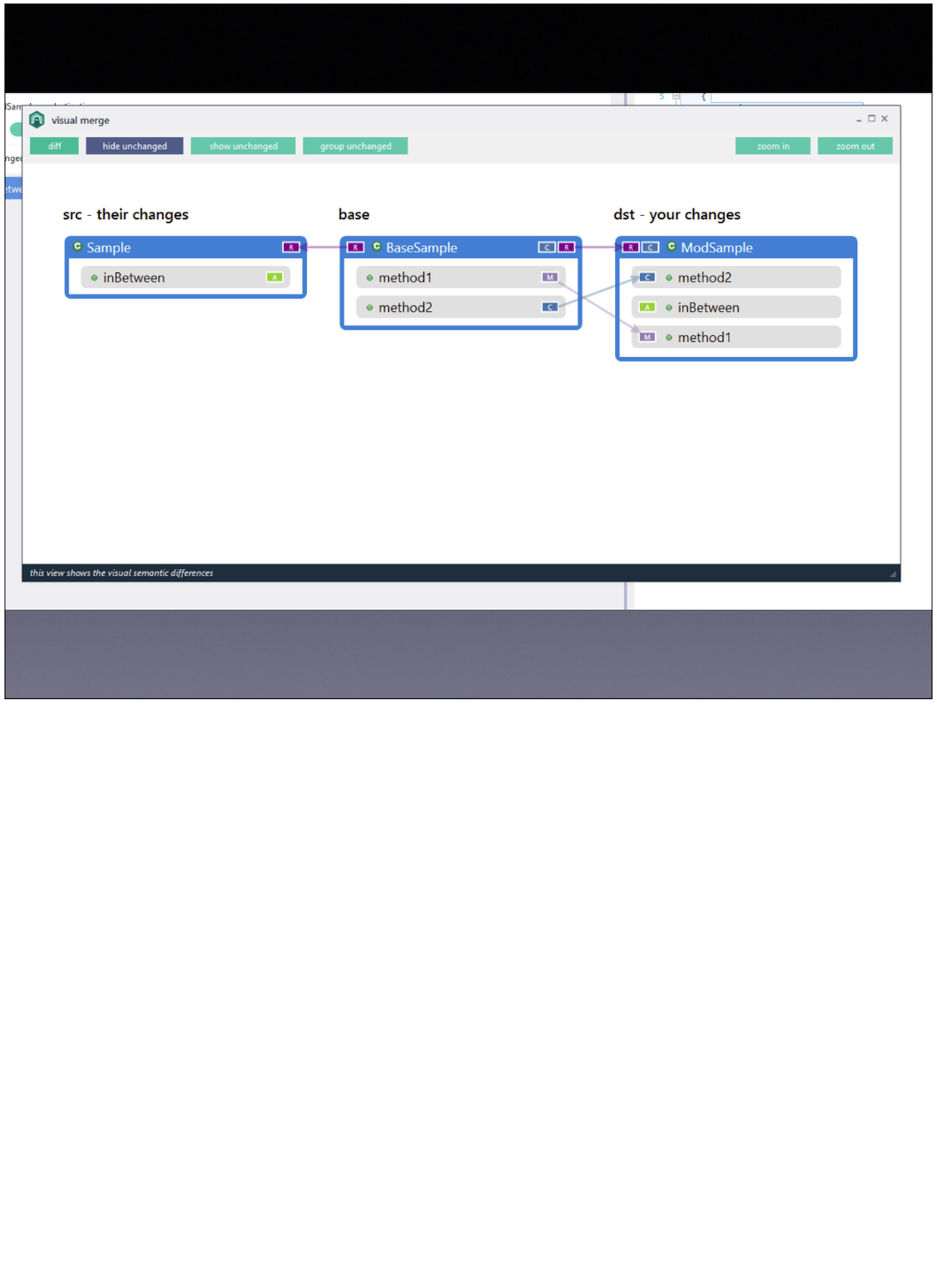
Selected existing tools:

1. Eclipse - Structure Compare
2. semanticmerge.com
3. smart differencer by SemanticDesigns

Diff should understand that all we've done is change the location of a method definition, not changed definitions.



Here's some screen shots from Semantic Merge. I've not used it a lot, but the pictures illustrate well what we should be aiming for.


visual merge
diff
hide unchanged
show unchanged
group unchanged
zoom in
zoom out
src - their changes
base
dst - your changes
Sample
inBetween
BaseSample
method1
method2
ModSample
method2
inBetween
method1
this view shows the visual semantic differences

DIY

Let’s talk about creating your own tools.

- Two main approaches:
  1. jet assist: lex, flex, yacc, bison, antlr
  2. by hand, typically recursive descent

```c
int accept(Symbol s)
{
  if (sym == s) {
    getsym();
    return 1;
  }
  return 0;
}


int expect(Symbol s)
{
  if (accept(s)) {
    return 1;
  }
  error("expect: unexpected symbol");
  return 0;
}
```

Writing a recursive descent parser is pretty straight-forward.

```
// block ::= { statement } [ returnStatement ]
// returnStatement ::= 'return' [ expressionList ] [ ';' ]

void block(void)
{
  do {
    statement();
  } while (statementStarter(sym));

  if (sym == RETURN_TOKEN) {
    returnStatement();
  }
}


void returnStatement(void)
{
  expect(RETURN_TOKEN);
  if (expressionStarter(sym)) {
    expressionList();
  }
  accept(SEMICOLON_TOKEN);
}
```

The missing piece is that as we parse structure in the source, we can insert statements in our parsing routines that build up a new data structure that we can then process with some algorithm to actually execute code, or pretty-print it, or gather stats, etc.

Antlr

- **ANother Tool for Language Recognition**
- Java development with Java, C#, Python runtime libraries (C++, JavaScript coming “soon”)
- Can handle any grammar you throw at it

Newest version of Antlr gives us a SAX-like way of processing source.

```
grammar ObjC;

translation_unit: external_declaration+ EOF;

external_declaration:
COMMENT | LINE_COMMENT | preprocessor_declaration
| function_definition
| declaration
| class_interface
| class_implementation
| category_interface
| category_implementation
| protocol_declaration
| protocol_declaration_list
| class_declaration_list;

preprocessor_declaration:
IMPORT
| INCLUDE;


class_interface:
    '@interface'
    class_name (':' superclass_name)?
    protocol_reference_list?
    instance_variables?
    interface_declaration_list?
    '@end';
```

Just a small piece of a grammar for Objective-C.

```
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.misc.Interval;

public class ProtocolMakerListener extends ObjCBaseListener
{
    ObjCParser parser;

    public ProtocolMakerListener(ObjCParser parser) { this.parser = parser; }

    @Override
    public void enterClass_interface(ObjCParser.Class_interfaceContext ctx) {
        ObjCParser.Class_nameContext cnCtx = ctx.class_name();
        System.out.println("@protocol proto_" + cnCtx.IDENTIFIER());
    }

    @Override
    public void exitClass_interface(ObjCParser.Class_interfaceContext ctx) {
        System.out.println("@end");
    }

    @Override
 public void enterMethod_declaration(ObjCParser.Method_declarationContext ctx) {
        TokenStream tokens = parser.getTokenStream();
        System.out.println("\t" + tokens.getText(ctx.getSourceInterval()));
    }
}
```

Let’s write some code to “reverse engineer” a protocol given a class.

```
// Input

@interface Sample : NSObject

- (void)method1:(NSObject *)o1 withParams:(NSObject *)o2;
- (NSInteger)method2: /* with an in-line comment */
                    (NSArray *)anArray; /* ignored comment */
- (IBAction)method3:(id)sender;

@property (nonatomic, weak) IBOutlet someUIThing;
@end


// Output

@protocol proto_Sample
    (void)method1:(NSObject *)o1 withParams:(NSObject *)o2;
    (NSInteger)method2: /* with an in-line comment */ (NSArray *)anArray;
    (IBAction)method3:(id)sender;
@end
```

Clang

```
// From http://kevinaboos.wordpress.com/2013/07/23/clang-tutorial-part-ii-libtooling-example/

class ExampleVisitor : public RecursiveASTVisitor {
private:
    ASTContext *astContext; // used for getting additional AST info

public:
    explicit ExampleVisitor(CompilerInstance *CI)
        : astContext(&(CI->getASTContext())) // initialize private members
    {
        rewriter.setSourceMgr(astContext->getSourceManager(),
            astContext->getLangOpts());
    }

    virtual bool VisitFunctionDecl(FunctionDecl *func) {
        numFunctions++;
        string funcName = func->getNameInfo().getName().getAsString();
        if (funcName == "do_math") {
            rewriter.ReplaceText(func->getLocation(), funcName.length(), "add5");
            errs() << "** Rewrote function def: " << funcName << "\n";
        }
        return true;
    }

    virtual bool VisitStmt(Stmt *st) {
        if (ReturnStmt *ret = dyn_cast(st)) {
            rewriter.ReplaceText(ret->getRetValue()->getLocStart(), 6, "val");
            errs() << "** Rewrote ReturnStmt\n";
        }
        if (CallExpr *call = dyn_cast(st)) {
            rewriter.ReplaceText(call->getLocStart(), 7, "add5");
            errs() << "** Rewrote function call\n";
        }
        return true;
    }
};
```

Clang also allows us to do “SAX-ish” processing of source. And we get a base class with a high degree of granularity.

```
// From http://kevinaboos.wordpress.com/2013/07/23/clang-
tutorial-part-ii-libtooling-example/

// this replaces the VisitStmt function above
virtual bool VisitReturnStmt(ReturnStmt *ret) {
    rewriter.ReplaceText(ret->getLocStart(), 6, "val");
    errs() << "** Rewrote ReturnStmt\n";
    return true;
}
virtual bool VisitCallExpr(CallExpr *call) {
    rewriter.ReplaceText(call->getLocStart(), 7, "add5");
    errs() << "** Rewrote function call\n";
    return true;
}
```

So we can choose to override at very specific spots.

The Point Revisited

We've looked at a range of tools to help you more effectively wrangle text. We've also seen (*hopefully*) how easy it is to develop your own tools. As you develop software, always keep Larry Wall's three cardinal virtues in mind: laziness, impatience, and hubris.

Links

- xmlstarlet: http://xmlstar.sourceforge.net/
- jq: http://stedolan.github.io/jq/
- ack: http://beyondgrep.com/
- ag: https://github.com/ggreer/the_silver_searcher

Links (cont)

- Emacs mode for variable coloring: https://github.com/ankurdave/color-identifiers-mode
- Light Table: http://lighttable.com/
- Brett Victor on Learnable Programming: http://worrydream.com/LearnableProgramming/
- Tops: open up terminal and enter ‘man tops’

Links (cont)

- Semantic Merge: http://semanticmerge.com/
- Antlr: http://www.antlr.org/
- Clang: http://clang.llvm.org/
- Clang Tutorial: http://kevinaboos.wordpress.com/2013/07/23/clang-tutorial-part-i-introduction/

Binomial nomenclature

- A scientific name (or *binomen*) is composed of two parts. The first part identifies the species. The second part (known as the *specific epithet*) identifies the species within the genus.
- The specific epithet can be the Latinized version of a surname to honor a particular person.
- *Malus,* a genus of 30-50 species which includes *M. domestica*, the orchard apple.

Some jokes aren’t all that funny to begin with.