Automating Code Magnet Generation

Julia Dana

Contents

1 Introduction **4**

1.1 The Overview: Easier Creation of Code Magnet Microlabs 4
1.2 The Context: What is WAGS? 4
1.3 The Problem: Brittle Input 5
1.4 The Solution: Parsing by Grammar 6

2 Development results and future extensions **7**

2.1 What It Does: Internal Functions 7
2.1.1 Parses by Grammar 7
2.1.2 Alternative Magnets and Other Instructor Directives . . . 11
2.1.3 Improved Representation of Magnets 11
2.2 What It Does: External Functions 12
2.2.1 Command Line Tool 12
2.2.2 Automated Interaction With WAGS Website 12
2.3 What It Could Do: Future Extensions 13
2.3.1 Automatic Generation of Alternative/Distractor Magnets 13
2.3.2 GUI Tool 13
2.3.3 Further Automated Interaction With WAGS Website . . 13
2.3.4 Supporting New Languages 14
2.3.5 Additional Serialization Formats 14

3 User Guide **14**

3.1 Quick Start - CLI Tool 14
3.2 Installation for Use 14
3.3 Explanation of Magnet Creation 15
3.3.1 Java 15
3.3.2 Python 3 15
3.4 Modifying the Magnet Creation: Instructor Directives 15
3.4.1 ALT and ENDALT: Creating Alternative Magnets 15
3.4.2 NODROP: Suppressing Drop Zones 16
3.4.3 EXTRAMAG: Creating an Extra Magnet 16

4 Developer Notes **16**

4.1 The Environment: Languages and Libraries Used 16
4.1.1 Installation for Development 17
4.2 The Design 17
4.2.1 Directory Structure 17
4.2.2 Key Classes 18
4.3 The Configuration: Specifying Magnet Sections 18
4.4 The Expansion: Adding a New Language 18
4.4.1 New Grammar Specification 18
4.4.2 Rakefile: Useful Actions for Adding a New Language . . . 19
4.5 The Packaging: Using the Rakefile 19

References **20**

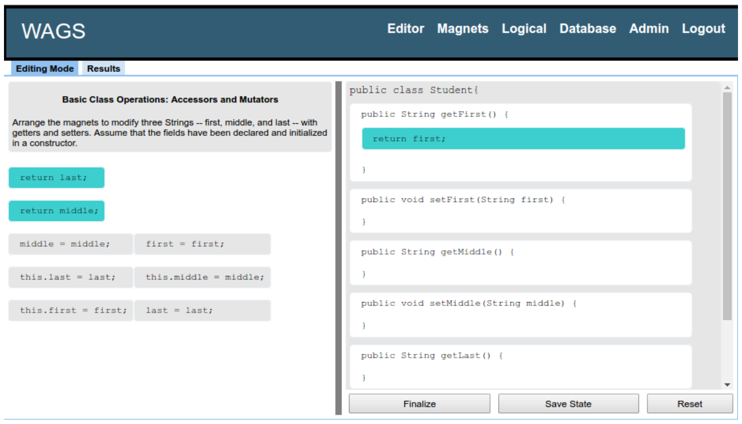
1 Introduction

1.1 The Overview: Easier Creation of Code Magnet Microlabs

The purpose of this project is to assist in the creation of code magnet microlab assignments for WAGS by creating magnets from a completed solution file. This is accomplished in a manner that supports multiple programming languages, and allows additional languages to be added with minimal configuration. Additional tools that support this idea of easier creation of assignments are also included, such as an automated interaction with the WAGS website to create assignments. Also, this project defines new formats for representing magnets, both as objects, and serialized to JSON or YAML.

1.2 The Context: What is WAGS?

WAGS (Web Automated Grading System) is an ongoing project of Appalachian State University. It is an online tool for microlabs. Microlabs are short, 5-10 minutes hands-on activities that are intended to be done as a part of a regular (i.e. not lab) class session to reinforce the concepts that are currently being covered. There are multiple types of microlabs provided by WAGS, but the one that this project is interested in is code magnet microlabs. These are microlabs where the student is given code magnets (pieces of code). Then the student must choose the correct magnets to use and drag and drop them into the correct order to "write" a solution to the microlab.



Basic Class Operations: Accessors and Mutators

Arrange the magnets to modify three Strings -- first, middle, and last -- with getters and setters. Assume that the fields have been declared and initialized in a constructor.

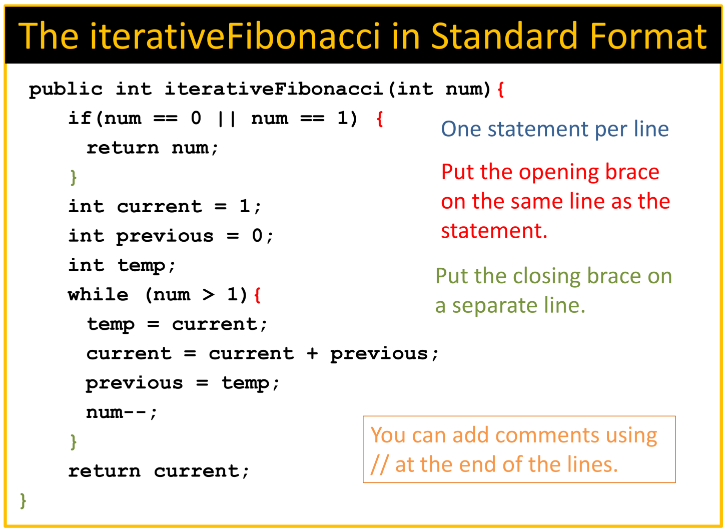
```
return last;
return middle;
middle = middle;       first = first;
this.last = last;      this.middle = middle;
this.first = first;    last = last;
```

```
public class Student{
  public String getFirst() {
    return first;
  }
  public void setFirst(String first) {
  }
  public String getMiddle() {
  }
  public void setMiddle(String middle) {
  }
  public String getLast() {
  }
```

Figure 1: An in-progress magnet microlab

1.3 The Problem: Brittle Input

The problem is that creating these magnet microlab assignments on the WAGS website is a somewhat painful process. The current parser creates magnets based on whitespace (line breaks and indentation). This limits the style of code that can be used. For example, opening curly braces in Java must be on the same line as the class/method/loop they are associated with. Also, statements have to be on a single line, regardless of length. See Figure 2 for an example [4]. These stylistic limitations, while limiting, do not force your input to the microlab creation to become an invalid solution file. However, to create alternative magnets for the existing parser, one adds extra lines to the input file. This does change the input file to no longer be a correct solution, and can even change it to no longer be a valid file for its language. This means that the instructor now has to maintain and keep in sync both a proper solution and the WAGS input file. See Figure 3 to see how adding alternative magnets can change an input file.


The iterativeFibonacci in Standard Format

```
public int iterativeFibonacci(int num){
    if(num == 0 || num == 1) {
      return num;
    }
    int current = 1;
    int previous = 0;
    int temp;
    while (num > 1){
      temp = current;
      current = current + previous;
      previous = temp;
      num--;
    }
    return current;
}
```

One statement per line

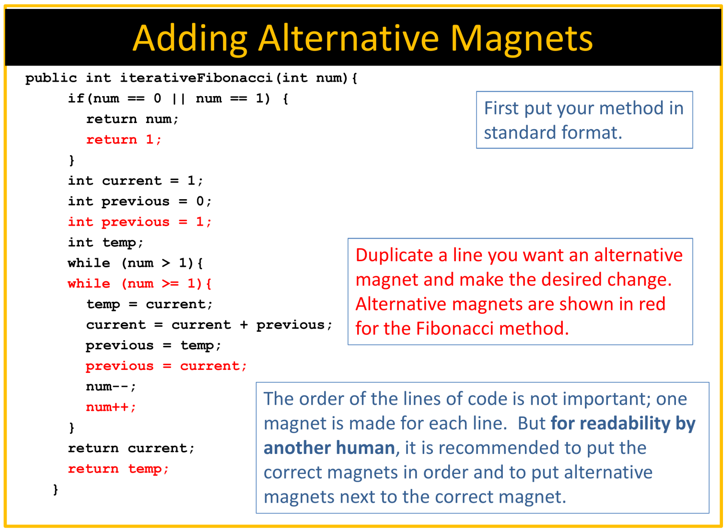
Put the opening brace on the same line as the statement.

Put the closing brace on a separate line.

You can add comments using // at the end of the lines.

Figure 2: Putting an Input File In Standard Format [4]

If your input cannot be handled by this brittle parser. WAGS does provide a manual input for magnets. However, using this manual input requires magnets to either be entered one at a time to the magnet creation wizard, or for the user to directly type the final magnet (including HTML escape sequences) to the parsed output section.



Adding Alternative Magnets

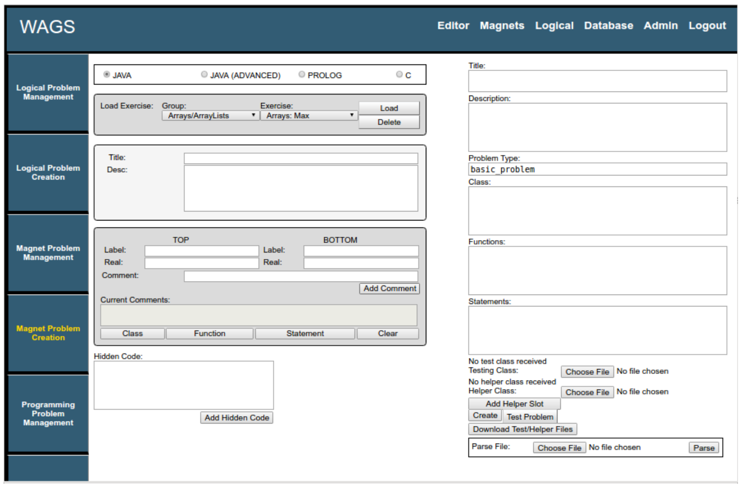
```
public int iterativeFibonacci(int num){
     if(num == 0 || num == 1) {
       return num;
       return 1;
     }
     int current = 1;
     int previous = 0;
     int previous = 1;
     int temp;
     while (num > 1){
     while (num >= 1){
       temp = current;
       current = current + previous;
       previous = temp;
       previous = current;
       num--;
       num++;
     }
     return current;
     return temp;
   }
```

First put your method in standard format.

Duplicate a line you want an alternative magnet and make the desired change. Alternative magnets are shown in red for the Fibonacci method.

The order of the lines of code is not important; one magnet is made for each line. But **for readability by another human**, it is recommended to put the correct magnets in order and to put alternative magnets next to the correct magnet.

Figure 3: Adding Alternative Magnets [4]



[✓ JAVA]

Load Exercise: Group: Arrays/ArrayLists Exercise: Arrays: Max

Problem Type: basic_problem

No test class received
Testing Class: No file chosen
No helper class received
Helper Class: No file chosen
Parse File: No file chosen

Figure 4: WAGS Magnet Assignment Creation Page

1.4 The Solution: Parsing by Grammar

The solution to the problems raised by the old, brittle parser is to use a more robust parser that is based on the grammar of the language, rather than the formatting of the input file. This project does this by using the ANTLR4 parser generator [6] and freely available grammars for common languages [2].

A grammar is a formal specification of a language. It is a series of rules (called productions) that define valid phrases of a language. A single production could define the structure of an entire file, or just be a single statement or expression. A production is the valid components of a statement (tokens and other rules). A rule can match one or more productions.

There are many parallels between a grammar of a programming language, and the grammar of a natural language (such as English). As a rough comparison, using a grammar-based parser to break code into magnets is like breaking apart a paragraph based on sentences or phrases, rather than on every period (Which commonly indicates the end of a sentence, but also has other uses, such as abbreviations). This type of parser will create tokens from the input, which are loosely the equivalent of words, and then break them into (sometimes nested) phrases and sentences according to rules called productions. Rules are named, and will sometimes contain more than one production as alternatives that provide the same function in a language.

A parser uses these rules to build a parse tree. The root of the parse tree is the start rule. The interior nodes of the parse tree are rules from the grammar. The leaf nodes of the parse tree are tokens from the lexer.

This project specifies which rules in the grammar should create their own magnets. If a rule is nested in another rule only the magnet of the nested rule gets the text. The magnet for the outer rule replaces the text of the nested rule with a drop zone (an area that accepts magnets)

Using a parsed file gives us a high level of control over how and when to create magnets. However, writing a robust parser is a non-trivial task. Today, most parsers are created by parser generators. These take the description of the language given in a grammar, and creates the parser for that language. There are many such parser generators, including YACC [7], Bison [3], and CUP [5]. This project uses the ANTLR4 [1] parser generator, which generates parsers in Java.

2 Development results and future extensions

2.1 What It Does: Internal Functions

2.1.1 Parses by Grammar

The magnetizer takes an input file, and parses it with the parser generated by ANTLR from the grammar specification. The result of this parsing is a parse tree, which is then used to create magnets. ANTLR can show us a graphical representation of this tree, as follows.

For this simple Java file, Hello.java

```java
/**
 * Sample Javadoc comment
 * @author Julia Dana
 */
public class Hello {
  public String helloWorld() {
    return "Hello World";
  }

  /* Sample multiline-style comment. */
  public static void main(String[] args) {
    // Sample singleline-style comment
    System.out.println("Hello");
  }
}
```

Listing 1: Hello.java

The resulting parse tree is:

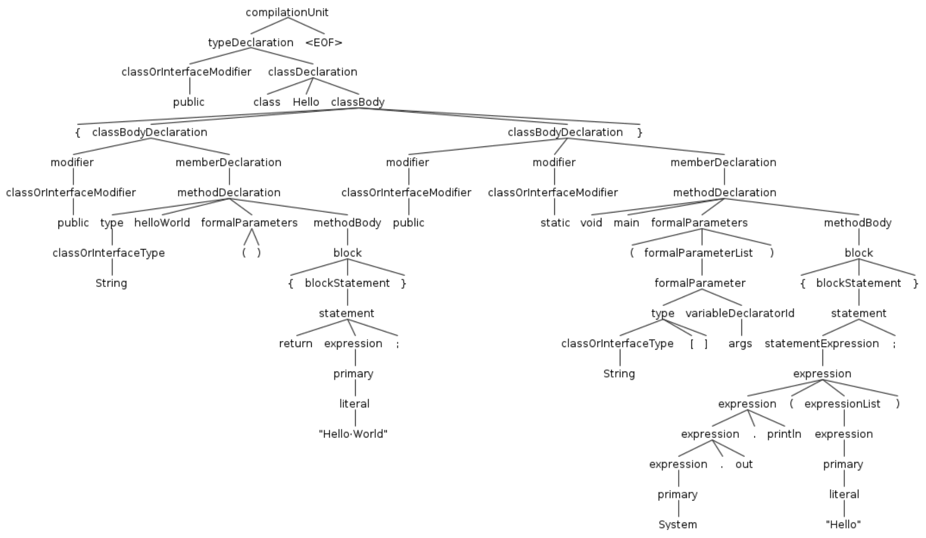


Figure 5: The parse tree from Hello.java

ANTLR provides two mechanisms for traversing this tree once it is created. One is a walker, which triggers enter and exit methods in a listener for each type of internal node, and the other is a visitor, which has methods for visiting each type of internal node. For both mechanisms, ANTLR provides a Java class which is a base version of the listener or visitor, with stubs of the methods containing minimal functionality. The programmer can then subclass these base versions to perform their project-specific actions at certain nodes. This project initially used the listener mechanism, but later switched to using the visitor to make it easier to implement the NODROP instructor directive. One key difference between the mechanisms is that the walker/listener method will always visit every node in the tree. With the visitor method, the base (ANTLR-generated) visitor visits every node by default, but the specific visitor can choose to not visit some or all of the children of a node, because the visit method is responsible for choosing when to visit children.

The visitors for this project are dynamically created JRuby subclasses of the ANTLR-generated base visitors. There is an ANTLR-generated base visitor per supported language. There is a single JRuby generator that creates a subclass of the Java base visitor for any language. The generation of the subclass is controlled by an entry in a configuration file which specifies the name of the grammar used (language name), and which rules (which become internal nodes in the parse tree) trigger the creation of a magnet. This dynamic generation prevents repeated code because we want to perform the same types of actions at every node that triggers a magnet, but the structure that ANTLR expects requires separate methods.

```
- !customObject/LanguageConfig
  name: Java
  start_rule: compilationUnit
  allow_file_statements: false
  directives:
    start: '/*#'
    end: '*/'
  preamble_type:
    - PackageDeclaration
    - ImportDeclaration
  class_type:
    - TypeDeclaration
  in_block_type:
    - ClassBodyDeclaration
    - BlockStatement
  overrides:
    - name: ClassBodyDeclaration
      child_node_type: MethodDeclaration
      child_level: 2
      location: method
```

Listing 2: The section of the configuration file that sets up Java magnetization

This configuration is loaded into a LanguageConfig object that is used to create the subclass, as shown in the following snippet.

```
# Dynamically get the (ANTLR-generated) class that
# we need to subclass.
parent_class = eval "Java::#{language.downcase}_parser.#{
    language}BaseVisitor"
# Dynamically get the ANTLR-generated parser we will
# be using.
parser_class = eval  "Java::#{language.downcase}_parser
    .#{language}Parser"

```

```
# Dynamically create the name of the class we will
# be creating.
generating_class_name = "Generated#{language}
    MagnetEmitterVisitor"

# Dynamically create our new class that subclasses the
# ANTLR-generated class
generated_class = Class.new(parent_class) do

  # Reference to the configuration for this language
  # from the YAML file
  language_info = LOADED_LANGUAGES[language]
  magnet_types = language_info.magnet_nodes

  # Omitted - creating methods with names that do
  # not change based on the configuration.

  # Create the visit methods for each of the types of
  # nodes that trigger the creation of a magnet
  magnet_types.each do |node|
    define_method "visit#{node.name}" do |ctx|
      # Omitted - the body of the visit methods
    end
  end
end

# Connect the class we just created to the name
# we want it to have.
Object.const_set generating_class_name, emitter_class
```

Listing 3: Dynamically subclassing a Java class in Ruby

One difficulties encountered in creating magnets from the parse tree is that the ANTLR-generated parser (like most parsers) discards whitespace once the parse tree has been built. By default, only the tokens themselves are kept. So, although the nodes in the tree provide a getText() method, the result of this is something like `publicclassMyClass` rather than the expected `public class MyClass`. However, ANTLR provides a way to keep extra information such as whitespace and comments around for later use if they are needed called hidden channels 6. This splits the incoming token stream into a channel for use by the parser, and hidden channels that can later be accessed to recreate the original text (with formatting) for the magnet. This hidden channel mechanism is also used to implement instructor directives.

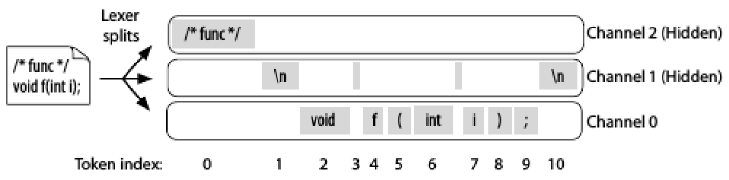


Figure 6: Diagram of tokens being split into channel [6, p. 207]

2.1.2 Alternative Magnets and Other Instructor Directives

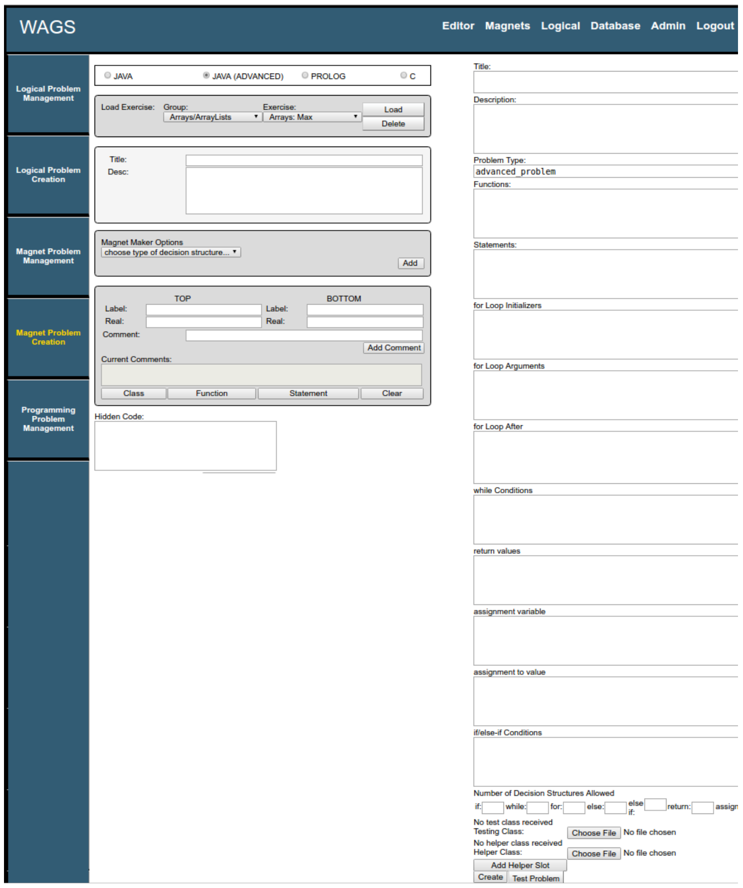
Instructor directives allow instructors to control how magnets are created. They are implemented as special comments that go to a special token channel. The content of directives are the same across all languages, but the triggering comment syntax is specific to each input language. Current directives implemented all instructors to suppress drop zones (NODROP), to indicate that a duplicate magnet with a section of alternate text should be created (ALT and ENDALT), and to create an unrelated magnet (EXTRAMAG).

All the directives currently work with Java input file. However, there are some limitations with directives in Python. Because comments in Python always go to the end of a line, creating a magnet with alternate text does not work, since the original text cannot be surrounded by comments. Also because of line-based comments, unrelated magnets must be created in a single line. There are also some cases where directives do not properly get connected to the expected node in the tree because indentation is not considered discardable whitespace in Python.

2.1.3 Improved Representation of Magnets

The old format for describing magnets is difficult to read and worse to try to manually type. Some of its quirks are that magnets are separated by an unusual separator (`.:|:.`), areas that accept other magnets (aka drop zones) are indicated by a seemingly random set of HTML tags (`<br><!-- panel --><br>`), and any special characters used in the code magnet must be escaped for HTML (so something like `1 < 2` would need to be entered as `1 &lt; 2`). This format also is limited when trying to create more advanced types of magnets. The WAGS system addresses this problem by adding more form fields when adding "advanced Java magnets", but this does not work well for adding additional languages.

This project creates an underlying object based data structure for magnets, as shown in figure 8. This structure can then be serialized as desired. Currently, serialization to the old magnet format is supported for backward compatibility, as well as JSON and YAML for more robust usages.



JAVA (ADVANCED)

Load Exercise: Group: Arrays/ArrayLists Exercise: Arrays: Max

Problem Type: advanced_problem

No test class received
Testing Class: No file chosen
No helper class received
Helper Class: No file chosen

Figure 7: Creating a magnet assignment with advanced magnets

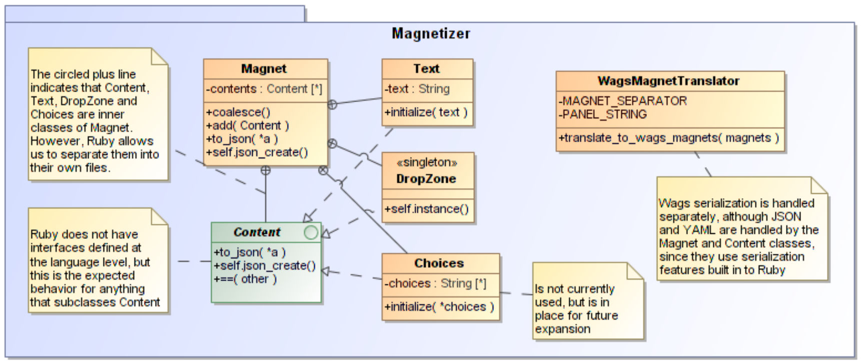
2.2 What It Does: External Functions

2.2.1 Command Line Tool

This project provides a command line tool, which is a thin wrapper exposing the internal API functions to the command line.

2.2.2 Automated Interaction With WAGS Website

This project contains a proof-of-concept automated interaction with the WAGS website, which is capable of logging in to the site, and creating an assignment with magnets from a specified input file. The assignment that is created has the magnets loaded, but it does not have any of the other files that the WAGS system needs to automatically provide feedback to the students. The automated interaction is provided using libraries commonly used in Ruby for automated



The circled plus line indicates that Content, Text, DropZone and Choices are inner classes of Magnet. However, Ruby allows us to separate them into their own files.

Ruby does not have interfaces defined at the language level, but this is the expected behavior for anything that subclasses Content

Is not currently used, but is in place for future expansion

Wags serialization is handled separately, although JSON and YAML are handled by the Magnet and Content classes, since they use serialization features built in to Ruby

Figure 8: The classes of the magnet data structure

testing of websites. It uses the Capybara acceptance test framework (which provides a convenient, cross web driver DSL for interacting with websites), the Poltergeist web driver, and the PhantomJS headless browser. Support for Capybara, the Selenium web driver, and Firefox browser is coded, but Selenium will not trigger the log in button on the current web site.

2.3 What It Could Do: Future Extensions

2.3.1 Automatic Generation of Alternative/Distractor Magnets

The next step in easily creating code magnet assignments is to have the magnet creation tool not simply create magnets needed for the correct solution and any additional magnets specified by the instructor, but also to automatically create appropriate alternative "distractor" magnets for common student misconceptions and errors. This level of manipulation is available because we have the whole parse tree to work with during magnet creation, but would require significant work per language to define.

2.3.2 GUI Tool

A GUI tool that can open an input file, perform the actions of the CLI tool on it, and also has an editor to assist in the addition of any instructor directives desired would be a nice addition to this project.

2.3.3 Further Automated Interaction With WAGS Website

It would be nice if this project could completely and reliably upload new magnet assignments to the WAGS website. The current interaction with the website does not upload the test file or any information other than the magnets. It also does not support the advanced Java magnet upload page. It also only works with the headless web driver (Poltergeist), not the one that shows the user what is happening (Selenium). However the limiting factor is that the functionality of the WAGS site is split between two versions of the website, and that the site has not been designed with programmatic interaction in mind.

2.3.4 Supporting New Languages

This project currently can create magnets for Python3 and Java, however it is set up to easily allow the addition of new languages. Basically, this is done by finding and adding an ANTLR4 grammar file for the desired language, making sure it conforms to a handful of guidelines, writing a short configuration file, and running a setup action on the project. Full details of this process are in section 4.4 The Expansion: Adding a New Language.

2.3.5 Additional Serialization Formats

Because magnets are now objects, additional serialization formats (such as XML) could easily be added.

3 User Guide

3.1 Quick Start - CLI Tool

The primary interface to this project is the magnetizer command line tool. This tool is packaged as a stand-alone executable JAR file.

It takes as input a file name, and outputs the magnets created from the input file. By default it assumes the input file is in Java. If it is in Python, the language options needs to be specified as Python3.

By default, it prints the magnets in WAGS format to the console. It can also or instead print the magnets to JSON or YAML. It also can send the magnets to a specified output file, rather than to the console.

```
Usage: java -jar wags-magnetizer.jar [options] file
    -l, --language LANGUAGE          Specify a language (default: Java,
                                       other: Python3)
        --json                       Print the JSON output
        --yaml                       Print the YAML output
        --[no-]wags                  Print the output as WAGS magnets
(default)
    -o, --output-file BASE_FILENAME  Output to a file
    -h, --help                       Show this message
```

3.2 Installation for Use

This tool is packaged as a stand-alone JAR file. It does not need to be installed before running. Its only dependency is that Java needs to be installed, and it has been tested with Java 7.

3.3 Explanation of Magnet Creation

3.3.1 Java

The current configuration for Java creates magnets for

- Package declarations
- Import declarations
- Type declarations
- Class body declarations - anything that can be directly in the class body
- Block statements - anything that can be directly inside a block.

3.3.2 Python 3

The current configuration for Python creates magnets for

- simple statements
- compound statements

3.4 Modifying the Magnet Creation: Instructor Directives

Instructor directives are information that can be put in the input file that change how the magnets are created. They are a special version of comments. The triggering syntax varies per language, but the directives are the same for all languages.

Java `/*# <directive> */`

Python3 `## <directive>`

Note that all the directives are functioning in Java, but not all are currently supported in Python. See the instructions for each directive to see if it is supported.

3.4.1 ALT and ENDALT: Creating Alternative Magnets

The ALT and ENDALT directives are used to create alternatives with different text for a magnet. The ALT directive also takes the desired alternative text, and the ENDALT directive is used to indicate where the end of the alternative text is. These directives should always be used as a pair, and encompass the text that should be replaced. They cannot go around anything that would trigger the creation of a new magnet or drop zone. This directive has only been tested in Java.

Example: To provide an alternative condition for a loop in Java. This snippet will create 3 magnets: one of the `while` statement with `altCondition` as the condition, one of the `while` statement with `realCondition` as the condition, and one for the `System.out.println()` statement.

```
while(/*# ALT altCondition */ realCondition /*# ENDALT */
    )
{
  System.out.println("Hello");
}
```

3.4.2 NODROP: Suppressing Drop Zones

The NODROP directive suppresses the creation of drop zones for the following magnet. An example of when this would be used is to put an entire loop on a single magnet, rather than having the individual statements in the body be on their own magnets. This directive has only been tested in Java.

Example: To create a loop whose body is contained on the main magnet in Java. Used when creating the body of a loop/method/etc. is not the focus of the microlab. This snippet will create one magnet for the entire for loop, rather than creating a separate one for the `System.out.println()` statement.

```
/*# NODROP */
for (int i = 0; i < 10; i++) {
  System.out.println(i);
}
```

3.4.3 EXTRAMAG: Creating an Extra Magnet

The EXTRAMAG directive is used to add a magnet that is unrelated to any of the magnets in the input file. This directive is tested in Java and Python.

Example: To see if students understand the purpose of a setter method in Java. This snippet will create three magnets: one for the method, one for assigning name, and one with a return statement.

```
public void setName(String name) {
  this.name = name;
  /*# EXTRAMAG return this.name; */
}
```

4 Developer Notes

4.1 The Environment: Languages and Libraries Used

This project is written in Ruby for the JRuby interpreter. JRuby is able to access Java classes from Ruby. For this project, this means we can use to use a parser generated by ANTLR from Ruby.

Automated interaction with the WAGS site is provided by using Capybara to interact with Selenium (drives Firefox) or Poltergeist/PhantomJS (headless).

Rake (Ruby make) is used to process new grammar files and other build tool functionality.

Bundler is used to handle dependency installation, and provides some helper tasks to Rake.

RSpec is used to handle testing.

Git is used for source control, and the projects are hosted on GitHub

4.1.1 Installation for Development

This project is stored in two repositories, `https://github.com/JuliaDana/wags-magnetizer` and `https://github.com/JuliaDana/wags-site-driver`. The majority of the project is in the wags-magnetizer repository, but the automated interaction with the WAGS website has been moved into the separate wags-site-driver repository, which uses the wags-magnetizer.

Once either repository has been cloned, run `bundle install` to install the dependencies for the project.

The wags-site-driver project uses (and includes) the PhantomJS headless browser. However, there are different versions for different operating systems, and it is currently manually set up for Linux. If you are using a different operating system, you will need to unzip the appropriate file in the etc/ folder, and re-create the phantomjs symbolic link.

4.2 The Design

4.2.1 Directory Structure

/**data** Contains grammar files and language-specific configuration information. This is the only location that has language-specific information.

/**doc** Contains documentation for the project.

/**etc** Contains any external projects and libraries needed for this project that are not available as Ruby gems.

/**exe** Contains the Ruby script that has set up as an executable and is the command line interface.

/**java** Contains Java code generated by ANTLR and the compiled class files from that generated Java code.

/**lib** The primary directory of the project. This contains the Ruby code for the magnetizer.

/**spec** Contains the RSpec tests for the project.

4.2.2 Key Classes

MagnetEmitterBase A mix-in that contains methods that have the same implementation across all the ruby-generated Visitors. In Ruby, a mix-in is a module that provides similar functionality to an abstract class in Java. When a class "include"s a module as a mix-in, that class gets all the methods defined in that module. Unlike abstract classes in Java, a class can have multiple modules mixed in, and a mix-in is unable to define abstract methods (Ruby does not have an equivalent to abstract methods).

MagnetEmitterVisitorGenerator Creates the correct subclasses of the ANTLR BaseVisitor. This class is where the bridging magic of JRuby is the most obvious, because we are creating subclasses of a Java class in Ruby.

ANTLR provides two primary patterns for traversing a parser tree: listener and visitor. This project uses the visitor pattern so that we do not have to visit child nodes in the parse tree for nodes that we know are terminal to our magnet generation.

4.3 The Configuration: Specifying Magnet Sections

The configuration that handles specifying how the parse tree for a language should be converted to magnets. The configuration is loaded from a YAML file in the `/data` directory. A language configuration contains the name (which corresponds to the name of the grammar file), the name of the rule that is the start rule for the parsing, whether or not the language should have a file-level drop zone, information about the special comment that indicates directives for that language. The names of the rules that indicate are listed by which WAGS section the magnets will need to go into. There is also a section to specify overrides for cases where the name of the rule is not sufficient information, and there needs to be additional information used to determine which list a magnet should go in.

This YAML file describes instances of LanguageConfig class, and the magnetizer then uses the LanguageConfig objects to create the magnets.

4.4 The Expansion: Adding a New Language

To add a new language to be magnetized, a new grammar needs to be added, the project rebuilt to include that grammar, and a new configuration as mentioned in section 4.3 The Configuration: Specifying Magnet Sections needs to be added. The build file (Rakefile) also provides some additional tools that are helpful to visualize the new grammar when writing the configuration.

4.4.1 New Grammar Specification

A new G4 file can be added to the system. However, whitespace is require to go on channel 1, or your magnets will not have any whitespace, and you might get things like "publicclassMyClass". Also, special comments for the directives need to be defined, and they need to go on channel 2. The entire comment for a directive should be recognized as a single token.

4.4.2 Rakefile: Useful Actions for Adding a New Language

When you have added a new grammar file to the system, run `rake generate` to use ANTLR to generate the new Java files for the parser, base visitor, etc. Then run `rake compile` to compile those files and package them for use.

4.5 The Packaging: Using the Rakefile

To run all tests prior to packaging, use `rake spec`.

To package the project as a gem (for use on systems that have a functioning JRuby environment), use `rake build`.

To package the project as a jar (which packages a JRuby-complete environment in with the project, use `rake jar`.

References

[1] *Antlr Homepage*. URL: http://www.antlr.org/.

[2] *antlr/grammars-v4*. URL: https://github.com/antlr/grammars-v4.

[3] *Bison Homepage*. URL: https://www.gnu.org/software/bison/.

[4] *Creating Code Magnet Lab*. URL: http://www.cs.appstate.edu/microlabs/ICER15/4-CreatingCodeMagnetLab.pptx%20http://www.cs.appstate.edu/microlabs/ICER15/index.html.

[5] *Cup Homepage*. URL: http://www2.cs.tum.edu/projects/cup/.

[6] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2012. ISBN: 978-1-93435-699-9.

[7] *YACC Homepage*. URL: http://dinosaur.compilertools.net/yacc/.