Automating Code Magnet Generation

Julia Dana

Contents

1 Introduction

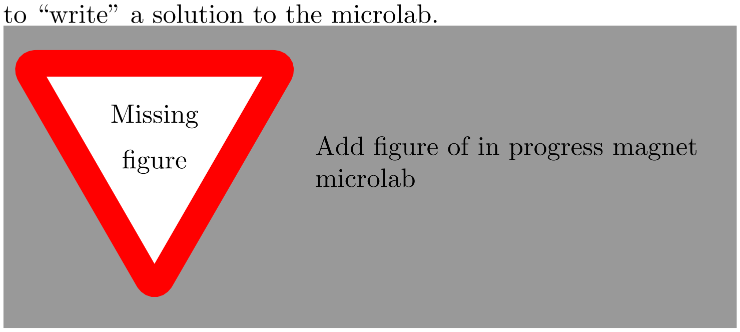
Defense of the problem definition

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.



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 1.3 for

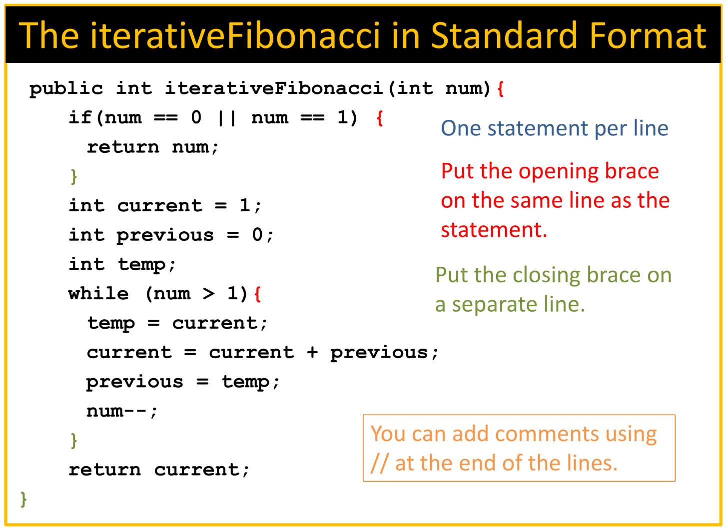
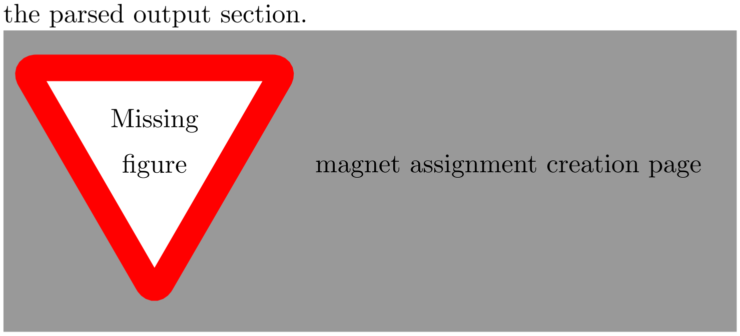


Figure 1: Putting an Input File In Standard Format

an example [2]. These sylistic 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. See Figure 1.3 to see how adding alternative magnets can change an input file[2].

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.



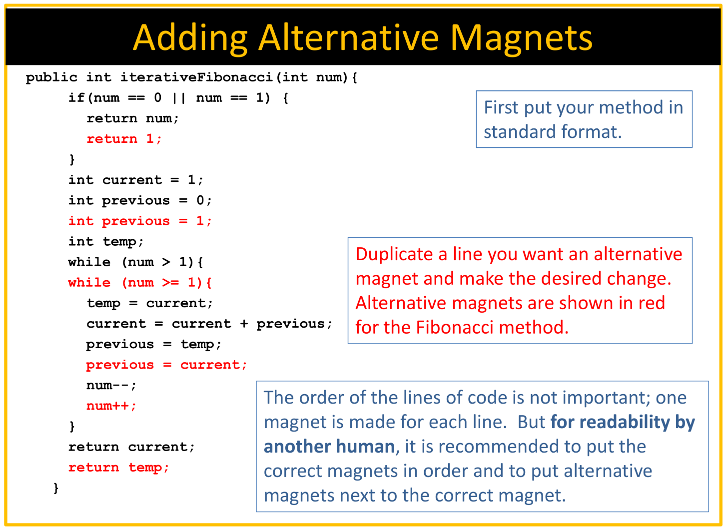


Figure 2: Adding Alternative Magnets

1.4 The Solution: Parsing by Grammar

The solution to this is to use a more robust parser that is based on the grammar of a language, rather than simply splitting on line breaks. This project does this by using the ANTLR4 parser generator[3] and freely available grammars for common languages[1].

Roughly speaking, 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.

Create magnets from certain types of "phrases"

One of the outputs available from a parser is a parse tree. This is similar to a sentence diagram.

configure nodes in the tree to trigger magnet creation

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 . This project uses ANTLR4, which generates parsers in Java.

list and cite parser generators

expand on ANTLR - LL(star)

2 Development results and future extensions

Defense of solution

2.1 What It Does: Internal Functions

2.1.1 Parses by Grammar

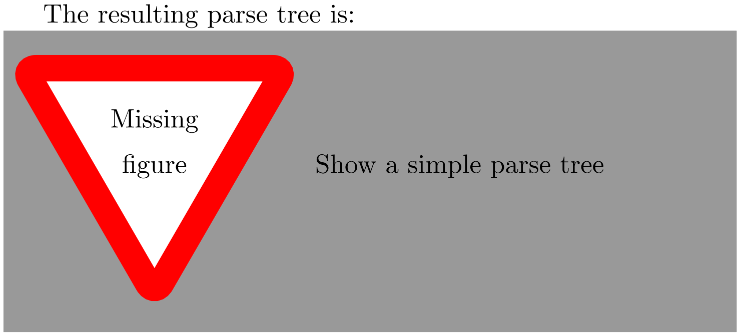
The magnetizer takes an input file, and parses it with the parser generated by ANTLR from the grammar specification. The result of this parsing is a parse tree, which can be represented graphically. An example follows.

Clarify this entire section

magnetizer find a better word

For this simple Java file, Hello.java

The resulting parse tree is:



Each language supported by the magnetizer has an entry in a configuration file that specifies which nodes in the parse tree should trigger the creation of a magnet. Whenever a child node triggers its own magnet, the parent is given a drop zone for that section, and that section of text is not directly included in the parent magnet.

ANTLR visitor vs. listener

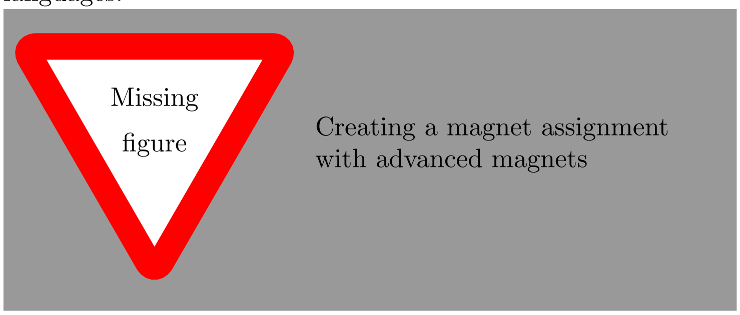
2.1.2 Alternative Magnets and Other Instructor Directives

Instructor directives allow instructors to control how magnets are created. They are implemented as special comments. 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, to indicate that a duplicate magnet with a section of alternate text should be created, and to create an unrelated magnet.

Discuss limitations in Python due to Indent/Dedent tokens

2.1.3 Improved Representation of Magnets

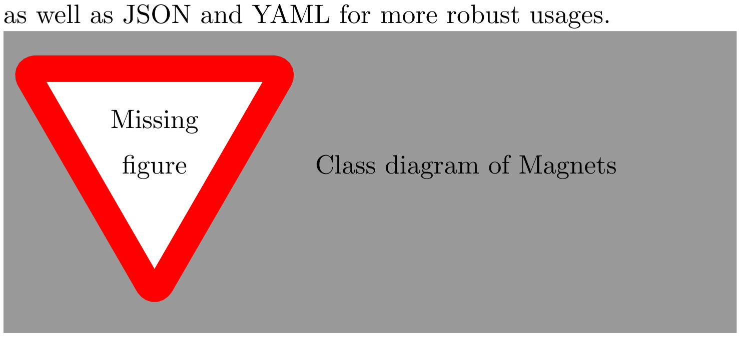
The old format for describing magnets is difficult to read and worse to try to manually type. Some of its quirks are that magnets are separated by an unusual separator (`.:|:.`), areas that accept other magnets (aka drop zones) are indicated by a seemingly random set of HTML tags (`<br><!-- panel --><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 < 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.



Creating a magnet assignment with advanced magnets

This project creates an underlying object based data structure for magnets, as shown in figure . 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.

link figure of class diagram



Class diagram of 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

Get information about Selenium using the wrong JS trigger from Sam

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 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 4.5 The Expansion: Adding a New Language.

2.3.4 Additional Serialization Formats

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

3 User Guide

Targeted to a CIS 1 instructor

3.1 Quick Start - CLI Tool

The primary interface to this project is the magnetizer command line tool. This tool is capable of outputting to WAGS-style magnets that can be copy-pasted into the parsed sections of the WAGS website or to a JSON or YAML representation of the magnets.

Make sure usage is the latest version.

```
Usage: magnetizer [options] file
    -l, --language LANGUAGE          Specify a language (default: Java)
        --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

3.3 Explanation of Magnet Creation

explain these in a way that doesn't require deep understanding of the ANTLR grammar.

3.3.1 Java

The current configuration for Java creates magnets for

- Package declarations
- Import declarations
- Type declarations
- Class body declarations - this is the nonterminal that includes anything that can be directly in the class body
- Block statements - this is the nonterminal that includes 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 directives in python are somewhat limited because comments cannot occur in a line.

Verify that there are no inline comments in python

explain which directives work in python

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.

Concrete example of ALT

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.

Concrete example of NODROP

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.

Concrete example of EXTRAMAG

3.5 Automatic Upload to WAGS Website

4 Developer Notes

Targeted to someone expanding on this project

4.1 The Environment: Languages and Libraries Used

Section needs more details

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.

RSpec is used to handle testing.

4.1.1 Installation for Development

Consider if this section should be on its own

4.2 The Design

4.2.1 Directory Structure

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

/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.

/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 The second of the two folders used for testing. This is for RSpec spec tests.

4.2.2 JRuby / Java Integration

4.2.3 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 Testing

4.4 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.

Link in the source code for the Java language specification

Class diagram of LanguageConfig, maybe?

4.5 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 4.4 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.

4.5.1 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.

4.5.2 Rakefile: Useful Actions for Adding a New Language

References

[1] antlr/grammars-v4.

[2] Creating Code Magnet Lab.

[3] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2012.

Final export of bibliography from Mendeley and link it

Make sure citations and bibliography are styled correctly