EFragment, a tool set for generating error fragments from YACC/Bison .y files

John Goettsche

August 26, 2014

Abstract

To reduce the burden of the author of compilers to generate meaningful error messages, a set of tools for generating error fragments from a YACC/Bison (.y) file was created. The generated fragments can used by a tool called Merr to generate a table driven yyerror() function that outputs a different syntax error message for each fragment. The EFragment tool set proved to be effective in generating a set of unique error Fragments that where tested against an existing set error fragments for the Unicon language.

1 Introduction

One of the tasks of an author or implementer of a programming language is the production of meaningful error messages for the developers who use the language. Anyone who has spent time developing software is all too aware that they are capable of making their fair share syntactic errors. When a syntax error exists in some code, it is helpful to have a description of the error so that it can be easily identified and corrected. This is especially true for the novice or intermediate programmer.

The Merr tool is a means by which the author of a compiler to provide meaningful messages for syntax errors and diagnostic messages. [2] With Merr the messages are stored in a message table that associates the error messages with state numbers. Merr reduces the difficulty of producing error messages for the author, by associating each error message with a string of tokens which the cause the error. The error messages and token strings are stored in a formatted file that is read by Merr.

This paper describes a software tool that reads in YACC/Bison (.y) files, constructs a data structure representing the context free grammar, and uses it to generate a set of example syntax error fragments in Merr format that exercise the grammar. The tool achieves most of its goals, which are described in Section

2. The implementation is described in Section 3. The tool has been evaluated by using it on a large grammar for a real programming language; the results and evaluation are given in Section 4, followed by conclusions.

2 Requirements

The EFragment tool set performs the following tasks:

1. Read, parse, and analyze its contents of a YACC/Bison (.y) file. Identify the terminal and non-terminal tokens, the context free grammar and the production rules in a YACC/Bison (.y) file.
2. Create the states and transitions which define a deterministic push down automata from the grammar.
3. Determine the path or chain of states required to reach each state.
4. Determine the acceptable and unacceptable inputs for each state.
5. Determine a chain of inputs from the chain of states that can result in reaching each state.
6. Determine an input to be added to each state that can result in an error fragment.
7. Provide a report listing the error fragments for all the states.
8. Provide a file that can be edited containing all the error fragments in Merr format.
9. Do an analysis of the generated error Fragments and comparing them to the error fragments that have already been identified.
10. Generate a report on the overlap of the generated and existing error fragments.
11. Generate an editable file containing all unique error fragments to date for the grammar.

The first item deals with extracting the data from the YACC/Bison (.y) file which EFragment tool set uses to generate the error fragments. The next five items (2 through 6) are where the analysis of the data takes place and the error fragments are generated. The remaining items (7 through 11) report the results and tests the effectiveness of the error fragments generated and reduces the redundant fragments.

3 Project Development

This project was developed using the Unicon language. One of the advantages of goal directed languages like Unicon is the simplicity of code in performing complex tasks. It is also an Object Oriented programming language which is advantageous for creating a data structure representing a push down automata.

In developing the tool, the project was divided into three sub sub-projects

1. **Reading:** Reading the YACC/Bison (.y) file, scanning it for tokens, identifying terminals and non terminal tokens and extracting the raw grammar for subsequent analysis.

2. **Analyzing and Generating Error Fragments:** Use the raw grammar to construct SLR items or states and an SLR parsing table. Identify the inputs that generate an error in the grammar for each of the states, which is used to generate a Fragment for the error fragment.

3. **Testing:** Each of the error Fragments was tested to see if the current error messaging system has identified the error, or if the error is being caught on the erroneous token. The acceptable new errors are saved to a file in the Merr format and a report of the results is generated.

3.1 Reading Source Files

The reading of the YACC/Bison file and retrieving the grammar of the YACC/Bison (.y) file was designed to be performed by a tool called 'yscanner'. This tool reads the file and extracts those parts that define the grammar and its terminal and non terminal tokens. To accomplish this a Data Access Object or Package was developed containing procedures for reading and writing data files. The Unicon language makes the retrieval of information from files very simple. The following code was used to access the data from a YACC/Bison (.y) file:

```
procedure getLinesFromFile(fileName)
   local infile
   inFile := open(fileName, "r") | stop("DAO ERROR: failure to read " || fileName)
   every line := !inFile do suspend line
   close(inFile)
end
```

After reading each line yscanner broke each line down into chucks of characters or words. Consider the structure of YACC/Bison (.y) files; [3] [4] they are divided into three sections and divided by '%%' having the following structure:

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

The definitions section is primarily used to define variables and tokens. This section will not be used in analyzing the grammar. All the words from this section are removed from the list. The rules section is where the grammar is located. The only portion that will be retained are the words that define the grammar, while the C code that defines how each production rule is to implemented, is removed. All the words found in the auxiliary routines section are removed if there are any.

This leaves a list of words that made up the raw grammar which was saved in a .raw file. From the raw grammar yscanner identifies all the non terminal tokens. This is done by identifying each production rule in the grammar and placing the word on the left side of every production rule into a non terminals list that has not already been added. To identify the terminals, it reads every token, ignoring '||' and ';', and adding to the terminals list each word that is not in the non terminals list and not already included. These lists are saved with the terminals being placed into a .t file and the non terminals being placed into .nt file.

It will be necessary during the analysis of this data for the system to identify lexical patterns that will be used when writing a program. The user may type in the lexeme '(' for example which is represented in the grammar with the token LPAREN; to generate accurate error Fragments, the program has to be able to identify the lexeme of each terminal token. Therefore yscanner generates a .tk file much like this one:

```
PLUS          #PLUS
TIMES         #TIMES
LPAREN        #LPAREN
RPAREN        #RPAREN
ID            #ID
```

The user is then able to edit this file replacing each of the #tokens with the appropriate lexeme.

3.2 Analyzing and Generating Error Fragments

The analysis of the raw grammar and its tokens is performed by the tool called 'EFragSLR'. This tool reads each of the files created in yscanner and stores that data into the following classes:

1. A terminal class containing its token and its lexeme.

2. A non terminal class containing its token and related production rules.

3. A list of production rules with lists of terminal and non terminals making up the production.

From this information, EFragSLR generates an SLR parser is produced accordance with the *closure* function in figures 4.33 and the *items* procedure in figure 4.34 in section 4.7 of Compilers: principles, techniques, and tools by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, or popularly known as the Dragon Book. Some grammars contain an *error* token which is part of the error recovery mechanism. Therefore, production rules containing an *error* token are ignored when EFragSLR generates an SLR parser for the context free grammar. Then an SLR parsing table was constructed by using Algorithm 4.8 from the same section of the same book. [1]

In generating each unique state, EFragSLR uses the *closure* function and *items* procedure described for 'Constructing SLR Parsing Tables in Section 4.7 of *'Comilers'* or the Dragon Book. It is from these states, an SLR parsing table is generated using Algorith 4.8 of *'Comilers'* This algorithm identifies whether to use a shift, reduce, or accept goto transition according to the input for that state. Inputs that do not result in a goto transition are considered to be an error.

Having identified the chain of states that brings it to the current state and the possible error inputs, an error token string can be identified by placing any error token at the end of the chain of states. Replacing the tokens in the error token string with their appropriate lexeme should result in a fragment that generates an error.

3.3 Testing

In order to test how effective EFragSLR was at producing unique error Fragments, errSubset was created. It takes arguments for the compiler being used, a string of characters to identify where an existing Fragment is returned, the name of the existing Merr error Fragment file, and the error Fragment file to be tested.

Each error Fragment in the test file is placed into a test file and compiled. It reads the error message and determines whether it is included in the existing file and whether it is breaking at the appropriate input value. If the message proves to be unique it is added to a list of unique error Fragments; if it has already been identified it is recorded as existing; if it breaks in an unexpected location, then it is identified as a malformed Fragment.

When the analysis is done, a new file is generated containing all the existing error Fragments along with the unique error Fragments. The unique error Fragments are given the message 'syntax error'. The user will have to edit them to have the desired message for each particular error.

4 Execution and Results

These tools were applied to the Unicon grammar with the following results:

• yscanner was able to identify 133 tokens, two being: EOF and error, all of which were included in the token and the terminal files. It identified 71 non terminals that were included in the non terminal file. I was also able to identify a raw grammar that was saved in the raw grammar file.

• EFragSLR was able to generate a list of production rules and produce a closure for the grammar with 522 states and their associated actions. It was able to identify a direct chain of states from the starting state to the current state for every state, along with the appropriate input tokens taking it to each subsequent state in the chain. It was able to identify every acceptable and every unacceptable input for each state.

Finally it was able to produce 331 unique error Fragments. Not all states produce an error message. [2]

• errSubset examined the new and existing error Fragments and compared them. It generated the following report for their coverage:

```
Error Fragments:                                                  165

Existing Error Fragments:                                         70
        included in New Error Fragments:                          0 or 0%
        Existing Error Fragments included in New Error Fragments: 37 or 52%

New Error Fragments:                                              331
        Matching Fragments (Not Added):                           0 or 0%
        Fragments Accounted for (Not Added):                      144 or 43%
        Erroneous Fragments (Not Added):                          92 or 27%
        New unique Error Fragments Added:                         95 or 28%

unigram-ES.err created.
```

This report says that a unigram-ES.err file was created with 165 unique error Fragments. From the existing Fragment file, there were 70 error Fragments, none that were identical to the new error Fragments and 37 of the existing error Fragments were triggered while compiling all the new error Fragments, resulting in a coverage of 52%. From the new error Fragment file there were 331 error Fragments, none match any of the Fragments in the existing Fragment file. 144 of the new Fragments where accounted for by existing error Fragments, for a cover age of 43%. There were 92 error Fragments that broke on the wrong terminal and are therefore counted as malformed or erroneous Fragments. In all there were 95 unique error Fragments that were added to the existing 70 for a total of 165 error Fragments.

5 Conclusions

yscanner was successful in reading a YACC/Bison (.y) file, identifying the grammar and its terminal and non terminal tokens, and storing that information in a set of files that were to be used by EFragSLR. EFragSLR was able to read the files generated by yscanner to create production rules, states and their associated items and parsing table, and was able to generate a set of error Fragments in the Merr format. When the errSubset tool was run on the existing set of error Fragments with the error Fragments generated by EFragSLR, the success was not as significant as I hoped, but it did manage to generate an additional 95 error Fragments that had not yet been identified.

This set of tools can be used to reduce the burden of the author of a compiler to identify syntax errors and provide meaningful messages to programmers.

The success of EFragSLR shows that future work with error message generators is warranted. Tools to generate error Fragments in the Merr format should be produced using the LR(1) and/or LALR parsing methods in order to create a more complete set of error Fragments.

References

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools¡/TITLE¿*. Addison Wesley¡/publisher¿, 1985.

[2] Clinton L Jeffery. Generating lr syntax error messages from examples. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):631–640, 2003.

[3] John Levine. *Flex & bison*. " O'Reilly Media, Inc.", 2009.

[4] Kenneth C Louden. *Compiler construction*. PWS Publ., 1997.