Moderation Project: JSON to XML using Flex and Bison

Sammy Furr

December 2, 2019

1 Introduction

For the fifth lab in Design of Programming Languages, we were asked to write a program to translate JSON to XML. To do this, we used a scanning and parsing system called SLLGEN which was created by one of the two books for the class, Essentials of Programming Languages.[4, pg.379] Written in Scheme, SLLGEN uses macros to turn regular expressions and grammars represented in Scheme into lexers and parsers. Though SLLGEN works, the error reporting leaves a lot to be desired, and I was interested in writing a lexer and parser in more industry-standard tools. I've written a lexer, parser, and interpreter using Flex and Bison, two open-source and industry-standard tools. This paper aims to explore the differences between writing a scanner and parser in SLLGEN vs. in Flex and Bison. I will explore the choices languages and tools make when writing metaprograms.

2 Comparison

2.1 Language Differences

Since JSON is a relatively simple language to lex and parse, the largest difference between using SLLGEN and Flex+Bison was the language *they* are implemented in. SLLGEN is written in Scheme, whereas Flex and Bison are written in C. This gives a significant advantage to SLLGEN for those looking to program quickly, as the time necessary to write a JSON to XML translator after the lexer and parser are finished will be lower in Scheme than in C. In addition, those who don't wish to worry about memory management will be better served using SLLGEN–even simply lexing JSON in C requires memory management to handle JSON strings.

2.2 SLLGEN vs. Flex: Lexing

The implementation of the lexer in Flex and SLLGEN is relatively similar—both take a list of regular expressions, along with what token corresponds to the appropriate regular expression. In Flex, this matches a JSON number:

```
-?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
{ yylval.NUM = atof (yytext); return NUM; }
```

And this is part of the code to do the same using SLLGEN:

```
(jnumber ((or (concat "-" digit)
              digit)
          (arbno digit)
          ... number)
```

Though regular expressions are represented using Scheme functions, they work fundamentally the same way. Flex gives a bit more power when creating tokens–we can decide what, if any, value we want the token to have (yylval). The real advantage to using Flex over SLLGEN came when parsing JSON strings. The specification for JSON strings states that a JSON string is a quote, followed by anything until another quote is reached, with control characters escaped.[1] The notion of "any non-quote character" turns out to be impossible to represent using SLLGEN. Flex solves problems like this by using the concept of *start conditions.*[2, 10] A start condition causes the lexer to enter a state when it matches a particular rule, and only exit the state when an end rule is matched. Using an exclusive start condition, the lexer will not attempt to match any rules except for those intended to be matched inside the start condition—for example, control characters in a JSON string.[2]

Flex is more powerful than the SLLGEN equivalent, though this power comes at the expense of complexity. Flex can ultimately lex more complexly and with more options than SLLGEN. The trade-off is worth it in the case of parsing JSON, since JSON strings cannot be parsed correctly with SLLGEN.

2.3 SLLGEN vs. Bison: Parsing

Bison and the SLLGEN parser work in fundamentally different ways. SLLGEN generates a parser from an *LL(1)* grammar, specified using a scheme macro that approximates Backus-Naur form (BNF). In contrast, Bison generates a parser using *LR* grammars, specified in a format closer to written BNF.

SLLGEN generates a top-down LL parser, while Bison generates a shift-and-reduce LR parser. This leads to significant differences when specifying grammars for the parser generators. Since the SLLGEN parser is LL, it cannot utilize left-recursion to define rules such as those for JSON objects[3,

[1] Appendix A contains the JSON string specification.

[2] See json_xml.l, lines 23-84.

pg.67]. In contrast, Bison prefers left-recursion, since it requires less stack space to parse.[1, 3.3.3] The rule for parsing JSON objects in SLLGEN is generated using the SLLGEN special command for a seperated list:

```
(json ("{" (separated-list jstring ":" json ",") "}") jobj)
```

In contrast, the rule in Bison is specified using left recursion:

```
jobj:   jobjpair
      | jobj LS jobjpair
      ;
```

The code necessary to generate the Bison scanner ends up being longer than SLLGEN's, but the Bison grammar is far more clear. It more closely follows BNF, and it is clear from simply looking at the grammar how tokens will be shifted and reduced. In contrast, the SLLGEN grammar is frustratingly abstracted, it is difficult to tell in the above example how exactly separated-list works, and macros like this hide some of the actually grammar specification. In addition, Bison allows for code to be executed while the parse-tree is being generated. This leads to a far more powerful possibilities than simply generating a parse tree.

2.4 JSON to XML

The Bison parser can have a *semantic action* for every syntactic rule.[1, 3.4.6] This is what allows the parser to do more than run and say if the input is valid or not–it allows you to construct the parse tree for use in a compiler or interpreter, or in the case of the JSON to XML translator, simply translate as the parse tree is constructed.[3] To do this, the JSON to XML parser includes *midrule actions*[1, 3.4.8] as well as end of rule actions. For example, these midrule and normal actions print the XML tags for every item in a list:

[3]This merging of the JSON to XML interpreter and the parser is not typical in Bison. A parse tree could have been constructed and processed later, but this was unnecessary since most of JSON can translate directly to XML.

```
jlist:
        json
      |
        jlist
        { printf("</%s>", list_names[list_depth]); }
        LS
        { printf("<%s>", list_names[list_depth]); }
        json
        { printf("</%s>", list_names[list_depth]); }
      ;
```

Writing the JSON to XML interpreter in C is more time consuming than writing it in Scheme, though it is made easier by combining it with the Bison generated parser. Even with this time reduction, it is still far faster to write the interpreter in a language that so naturally handles recursion and has automatic memory management.

3 Conclusion

The languages and tools we use to write programs have a direct impact on our efficacy as programmers, and on the possible properties of our programs. This is more true than normal when writing metaprograms.

The complexity of metaprograms makes having tools that are easy to understand and which promote correct code essential. After implementing a JSON to XML interpreter in both Scheme using SLLGEN and in C using Flex+Bison, it is clear that Flex+Bison are the better tools for larger projects. Flex's scanner simply allows the programmer to do more than SLLGEN's does. Bison is far more explicit and powerful than SLLGEN's parser generator.

Despite these advantages for larger projects, the implementation language of our metaprogramming contestants comes into question for a small project such as this. Even while being comfortable in both C and Scheme, writing an interpreter in C simply takes more time, and can produce far more errors. When a programmer wants to spend a couple hours writing a simple interpreter, SLLGEN is a better fit, simply because it comes with Scheme.

4 Tests

Testing for the program was composed of two main parts: testing correctness of JSON to XML translation, and test memory management. The program correctly translates JSON to XML, with a couple of exceptions that I did not want to implement:

- There is no Unicode support.
- JSON object names are not camel cased when translated to XML tags.

Tests can be found in the file jxml_tests.

To test, I ran a number of increasingly complex working and broken JSON examples. The parser and lexer successfully catch when JSON is incorrectly formatted. To test memory, I used valgrind to check for leaks under circumstances when the program terminates. The parser generated by Bison does not garbage collect memory required by the stack or some other internal functions on exit, so if 3 blocks are not freed on exit this is acceptable. Any more, and there is a problem with memory management. The program does not leak additional memory, under any circumstances.

References

[1] Free Software Foundation. *GNU Bison - The Yacc-compatible Parser Generator*. Free Software Foundation, 2019.

[2] Vern Paxson Will Estes John Millaway. *Lexical Analysis With Flex, for Flex 2.6.3*. 2017.

[3] Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2007.

[4] Daniel P. Friedman Mitchell Wand. *Essentials of Programming Languages*. MIT Press, 2008.

A JSON String Specification