Parsing mathematical expressions

31 March 2019

Ever wondered how you can implement a simple calculator in C++? You probably haven't, but it is interesting to look at how the problem can be solved very elegantly using code generation. This post looks at how we can use two code generation tools, Ragel and Lemon, to generate a parser for simple mathematical expressions like '(1.2 + 1) * 2.5'.

Origins

One of my more ambitious past projects was an attempt to build something that, at the time, I described as an object-oriented spreadsheet. In my design, a class was essentially a set of constraints and type definitions that would apply to a range within a spreadsheet (e.g. one row, where each column follows a particular set of constraints, followed by *n* rows, where the columns follow another set of constraints). A programmer could then augment that 'class' with functions that were type-safe and well-defined, assuming all of those constraints were satisfied.

A class could then be instantiated on a particular range of a spreadsheet, after which all of those instance methods would be guaranteed to be well-behaved. Modifications to the data would fail if constraints would be violated. Instances of classes could even be overlapped. In many ways, this is Excel tables on steroids.

One way I thought this could be useful was to automatically generate constraints for a small, well-defined datasets, and to later apply those constraints to larger datasets as part of a data cleansing process. Constraint violations or type-mismatches could be detected, localised, and corrected, allowing the dataset to be used as an input to potentially larger applications. In retrospect, this could be more accurately described as applying type-safety to a spreadsheet, which is an interesting idea in itself, but not the focus of this post.

Formulas

One of the first steps in this project was constructing a formula parser. As many C or C++ programmers would do, I wrote an initial implementation using Flex and Bison (successors of Lex and Yacc, respectively). However, I was disappointed with their support for C++. Another sore point was that I had hoped to support multi-threading, and at the time, this was either impossible (or very poorly documented).

After looking at alternatives (included Boost Spirit), I settled upon *Ragel* and *Lemon*. It was Zed Shaw's post on Ragel State Charts that really piqued my interest in this option. His post painted Ragel+Lemon out to be a much less intimidating option than Boost Spirit, which is a parser library based on C++ templates. While powerful, Spirit has a reputation for long compile times, and being somewhat difficult to read.

Now if you've never heard of Ragel or Lemon (or Flex/Bison, or even Lex/Yacc) it'll be helpful to understand what they actually do.

Lexical analysis

THe first step in parsing a formula (or any programming language) is tokenisation, or lexical analysis. This is how we take a formula (as a string) and extract the meaingful substrings. For example, when parsing a formula in a spreadsheet, we care about things like cell references, numbers, and such, but we usually do not care about whitespace.

The example we'll come back to a few times in this post is the string '(1.2 + 1) * 2.5'. If we tokenised this, ignoring whitespace, we might come up with the following representation:

```
LPAREN
LITERAL(1.2)
ADD
LITERAL(1)
RPAREN
MUL
LITERAL(2.5)
```

This representation allows a program to perform useful tranformations on the string, although for convenience, I'll continue to write the tokens as '(', '1.2', '+', etc.

Lexical analysis is usually performed by taking a list of regexes. When a part of the input matches a particular regex, an appropriate token will be emitted. Ragel can generate code that does this efficiently.

Parser

Once we have a stream of tokens, how do we evaluate them, to arrive at a final value?

Say we had the sequence of tokens '(1.2 + 1) * 2.5'. We could write a function that sees a '(', and knowing that a ')' must follow at some point, looks for that token. Once ')' is found, the function could call itself recursively on the tokens in between '(' and ')'. This might work, but it would be inefficient for larger inputs.

Another approach is to work left to right, pushing tokens onto a stack, until we find that the tokens on the top of the stack can be reduced in some way. When those tokens are reduced, they are popped off the stack, and the reduced version is pushed onto the stack in their place. Thus, the sequence '(1.2 + 1) * 2.5' could be computed like so (with the stack shown on the left, and reductions on the right):

```
(
( 1.2
( 1.2 +
( 1.2 + 1   ->   ( 2.2      #  The tokens '1.2', '+', '1' are reduced to '2.2'
( 2.2 )     ->   2.2        #  The tokens '(', '2', ')' are reduced to '2'
2.2 *
2.2 * 2.5   ->   5.5        #  The tokens '2.2', '*', '2.5' are reduced to '5.5
```

And we have our answer.

This is, in fact, the basis of an LR Parser - or a left-to-right (L), rightmost derivation (R) parser. Amazingly, an LR parser can be automatically generated from a set rules (or reductions), which together form the grammar for a language. Lemon is one such parser generator.

Abstract Syntax Trees

The last thing I wanted to mention before moving on to the more practical aspects of this post is the idea of an Abstract Syntax Tree.

An Abstract Syntax Tree (AST) allows us to represent a computation as a tree of simpler operations. Each node can represent an operation, value, or some other concept in a program, and can branch out as necessary. For example, when evaluating a 'addition' node. To evaluate that node, we would recursively evaluate all of its children, before calculating the sum of their evaluations.

Compilers typically use Abstract Syntax Trees to perform program optimisation, before finally converting the tree into byte-code or some other linear form. This is not something we explore in this post, but I think it is useful to be aware that the purpose of many parsers is to construct an AST, rather than evaluate an expression.

Now we can get to the practical details…

Ragel

Let's start with Ragel. Borrowing from the Ragel website:

> Ragel compiles executable finite state machines from regular languages. Ragel targets C, C++ and ASM. Ragel state machines can not only recognize byte sequences as regular expression machines do, but can also execute code at arbitrary points in the recognition of a regular language. Code embedding is done using inline operators that do not disrupt the regular language syntax.

Anyway, here's a Ragel program to match the expressions in our mini-language:

```
%%{

machine tokeniser;

main := |*

('-'?[0-9]+('.'[0-9]+)?) { std::cout << "LITERAL(" << std::atof(ts) << ")" <<
'+'                      { std::cout << "ADD" << std::endl; };
'-'                      { std::cout << "SUB" << std::endl; };
'*'                      { std::cout << "MUL" << std::endl; };
'/'                      { std::cout << "DIV" << std::endl; };
'('                      { std::cout << "LPARENS" << std::endl; };
")"                      { std::cout << "RPARENS" << std::endl; };
space                    { /* ignore whitespace */ };
any                      { throw std::runtime_error("Unexpected character"); }

*|;

}%%

#include <iostream>
#include <stdexcept>

int main() {
    const std::string input("(1.2 + 1) * 2.5");

    // Setup constants used in generated code
    const char * p = input.c_str();
    const char * pe = input.c_str() + input.size();
    const char * eof = pe;

    int cs;
    const char * ts;
    const char * te;
    int act;

    // Directives to embed tokeniser
    %% write data;
    %% write init;
    %% write exec;

    return 0;
}
```

If you put this in a file called 'tokeniser.rl', you can run it through Ragel to generate C++ code:

```
ragel -o tokeniser.cpp tokeniser.rl
```

You can then compile and run it:

```
g++ -o tokeniser tokeniser.cpp
./tokeniser
```

Hopefully, you'll get some output like this:

```
LPARENS
LITERAL(1.2)
ADD
LITERAL(1)
RPARENS
MUL
LITERAL(2.5)
```

Next, we'll do something useful with these tokens.

Lemon

This is where Lemon comes in. Lemon is a parser generator that is maintained as part of the SQLite project. It can be used to generate an LALR parser from a context-free grammar. This is what we will use to match sequences of tokens, and replace them with simpler derivations (e.g. `LITERAL(1.2) ADD LITERAL(1)` becomes `LITERAL(2.2)`).

LALR parsers?

At this point, I should step back for a moment, and talk about LALR parsers…

When I introduced LR parsers, I neglected to mention that LR parsers come in different 'strengths'. An LR parser is often denoted LR(k), where 'k' is the number of look-ahead tokens that the parser uses to determine which reduction to use next, or whether to push another token onto the stack.

So an LR(1) parser uses one token for look-ahead. This doesn't seem like a big deal, but it can have significant effect on the size of the parser's internal state machine. *Okay, we'll just use an LR(0) parser - what's the difference?*

The problem is that some languages cannot be unambiguously parsed with an LR(0) parser, so we need something bigger. LALR parsers were invented as a compromise between LR(0) and LR(1). They allow for more powerful languages (not quite LR(1), though often close enough) while having a smaller internal state machine.

Fun fact: C++ cannot be parsed by an LR(1) parser.

Back to Lemon

So Lemon takes as input a source file (typically with a '.y' extension) and a template (lempar.c), which is provided as part of the Lemon distribution.

First, put the following into a file called 'context.h' (the reason why will become clear shortly):

```
#pragma once

struct Context {
    double result;
    bool error;
};
```

Then, put the following into a file called parser.y:

```
%include {
#include <assert.h>
#include <stdbool.h>
#include "context.h"
#include "parser.h"
}

%token_type { double }
%extra_argument { struct Context * context }

formula ::= expr(A).
    { context->result = A; }

expr(A) ::= expr(B) ADD expr(C).
    { A = B + C; }

expr(A) ::= expr(B) SUB expr(C).
    { A = B - C; }

expr(A) ::= expr(B) MUL expr(C).
    { A = B * C; }

expr(A) ::= expr(B) DIV expr(C).
    { A = B / C; }

expr(A) ::= LPAREN expr(B) RPAREN.
    { A = B; }

expr(A) ::= LITERAL(B).
    { A = B; }

%parse_failure
    { context->error = true; }
```

Lets take a look at what is happening in this file. We start with the `%include` directive, which gives Lemon some code that needs to be included before the generated parser (in this case, header files). We then use `%token_type` to assign a type to the value that will be attached to each token. For our calculator, `double` is an appropriate choice. `%extra_argument` allows us to specify an argument for the parser, that will be used each time the parser is called.

The next section is the grammar itself. These are the reductions that make it possible to take a stream of tokens, and to evaluate them.

Finally, we have a `%parse_failure` directive, which says what to do if none the tokens could not be matched to the grammar. This highlights the importance of the 'context.h' file. The struct defined in that file keeps track of both the final result, and the current state of the parser.

Okay. Let's turn this into C code, by running it through Lemon:

```
lemon parser.y
```

What you get back is a file (parser.c) which, when compiled into your program, provides functions that will look like this:

```
void Parse(
    void * pParser,               /** The parser */
    int kind,                     /** The major token code number */
    double value,                 /** Value associated with a token (%token_ty
    Context * context             /** Optional %extra_argument parameter */
);

void *ParseAlloc(
    void * (*mallocProc)(size_t)  /** Function used to allocate memory */
);

void ParseFree(
    void * pParser,               /** The parser to be deleted */
    void (*freeProc)(void*)       /** Function used to reclaim memory */
);
```

You will also get a header file (parser.h), which contains numeric `#define`s for each kind of token used in the grammar.

`ParseAlloc` and `ParseFree` are used to initialise and destroy a parser, respectively. And once a parser has been initialised, you can pass in tokens, one at a time, using `Parse`. When passing in a token, we have to tell the parser what kind of token it is, using one of the constants defined in the parser.h.

When we ran Lemon with this grammar, we also got the following output:

```
4 parsing conflicts.
```

This is because we haven't told lemon what the precedence of the MUL/DIV and ADD/SUB tokens should be. We can do that by adding the following lines above `%token_type`:

```
%left ADD SUB.
%left MUL DIV.
```

This gives `MUL` and `DIV` higher precedence than `ADD` and `SUB`, and defines them as being left-associative. Left-associative means that in seeing an expression like this:

```
1 * 2 + 1
```

Lemon will attempt to parse that as:

```
(1 * 2) + 1
```

Running Lemon again will now ensure that we have an unambiguous parser.

Putting it all together

Now we have to link our Ragel tokeniser and our Lemon parser into one program. Although it would be nice to expose our Lemon-generated parser through C++ code, my experience has been that it is simpler to leave the Lemon-generated parser in its original C form. This way, we simply use `extern "C"` to call the parser functions from C++. You'll see how I've done that in the example code below.

To actually make a calculator, we'll need to update our Ragel code to know how to call the parser. This is going to involve a number of changes, so let's start a new file called 'calculator.rl':

```
%%{

machine formula;

main := |*

('-'?[0-9]+('.'[0-9]+)?) { Parse(parser, LITERAL, std::atof(ts), context); };
'+'                      { Parse(parser, ADD, 0, context); };
'-'                      { Parse(parser, SUB, 0, context); };
'*'                      { Parse(parser, MUL, 0, context); };
'*'                      { Parse(parser, DIV, 0, context); };
'('                      { Parse(parser, LPAREN, 0, context); };
")"                      { Parse(parser, RPAREN, 0, context); };
space                    { /* ignore whitespace */ };
any                      { return false; };

*|;

}%%

#include <iostream>
#include <stdexcept>

#include "context.h"
#include "parser.h"

extern "C" {
    void Parse(
        void * parser,                /** The parser */
        int kind,                     /** The major token code number */
        double value,                 /** Value associated with a token (%toke
        Context * context             /** Optional %extra_argument parameter *
    );

    void *ParseAlloc(
        void * (*mallocProc)(size_t)  /** Function used to allocate memory */
    );

    void ParseFree(
        void * pParser,               /** The parser to be deleted */
        void (*freeProc)(void*)       /** Function used to reclaim memory */
    );
}

bool calculate(void * parser, const std::string & input, Context * context) {
    int cs;
    const char * ts;
    const char * te;
    int act;

    // Setup constants for lexical analyzer
    const char * p = input.c_str();
    const char * pe = input.c_str() + input.size();
    const char * eof = pe;

    %% write data;
    %% write init;
    %% write exec;

    Parse(parser, 0, 0, context);

    return true;
}
```

We can see three important parts here: the state machine definition, parser references (parser.h and function prototypes), and a function called `calculate()`. The `%% write` directives in `calculate()` will embed the lexical analyser in that function.

Finally, we'll add a `main()` function, which prompts the user for input, and calls `calculate()`. It will loop until the input stream is terminated:

```
int main() {
    void * parser = ParseAlloc(::operator new);
    while (std::cin) {
        std::cout << "> ";
        std::string input;
        std::getline(std::cin, input);
        Context context = {0, false};
        if (calculate(parser, input, &context) && !context.error) {
            std::cout << context.result << std::endl;
        } else {
            std::cout << "Error: Invalid input." << std::endl;
        }
    }

    ParseFree(parser, ::operator delete);
    return 0;
}
```

Let's generate and compile everything from scratch, to ensure we have everything we need:

```
lemon parser.y
gcc -c parser.c
ragel -o calculator.cpp calculator.rl
g++ -o calculator calculator.cpp parser.o
```

This should produce an executable called `calculator`. Running this will present you with the calculator prompt:

```
>
```

Try it out!

```
> ( 1.2 + 1 ) * 2.5
5.5
> 1 + 1
2
> 2 * (2 + 3)
10
> Hello
Error: Invalid input.
> ((1 + 1)
Error: Invalid input
```

Inspect

What can we do with all of this?

One example is a small project that I've nicknamed *Inspect*. You can find the code here.

Inspect is a toy spreadsheet REPL that includes a formula parser, and allows cells to be defined using either literals, or formulas that reference other cells. At any time, the entire sheet can be recalculated. And to keep things simple, references to undefined cells will result in an empty value being retrieved. While simple, the project actually uses Ragel and Lemon in several interesting ways.

What also makes this project interesting to look at is that, instead of parsing formulas every time a sheet is recalculated, they are instead converted into an Abstract Syntax Tree at the time the formula is entered.

Note that when I say 'toy implementation', I mean it! Inspect has a dead simple command line interface, and is really only useful as a reference for using Ragel and Lemon to generate an AST.

Benchmarks?

I had originally intended to include some benchmarks and other metrics in this post, but I think it is long enough as it is. What I would like to do instead is write a follow up post that includes a more in-depth comparison with Boost Spirit and ANTLR-based parsers.

I'm also interested in explore how the same problem would be approached in other languages, such as Rust. Haskell would also be an interesting journey into the land of parser combinators.

In the mean time, you can find all of the code for this post on Github, in the microcalc repo. This includes a handy Makefile for compiling the calculator. It also supports for a few more basics operations, such as subtraction and division, and performs floating point arithmetic instead of integer arithmetic.

Useful resources

- Lemon page on SQLite website
- Lemon documentation
- Ragel website
- Ragel documentation
- Ragel State Charts (by Zed Shaw)

About

This is a blog about Software Engineering and the pursuit of Performance, Reliability and Scalability - with an occasional detour into the human and business side of technology.

Elsewhere

If you enjoy the content on this site, you should follow me on Twitter, or explore my code on Github.

Recent Posts

- Preprocessing JSON... using JSON
- Getting started with CUDA on AWS
- Parsing mathematical expressions
- Why I came back to Australia
- Reading list progress report
- A reading list for the year ahead...
- Tail Call Optimisation and C / C++
- Extending Ruby with C++
- Writing a Gem with native extensions
- Build your own workstation
- Porting an Asteroids clone to JavaScript
- Sparsehash Internals
- Fixing the authorship of a Git commit
- WordPress and Docker
- Implementing a Distributed Hash Table with Scala and Akka
- Installing R and RStudio on Mac OS X