Generating parsers using Ragel and Lemon

Tristan Penman

Melbourne C++ Meetup, April 2019

Parsers, and parser generation

Some computational theory

- Language
  - a formal language consists of words whose letters are taken from an alphabet and are well-formed according to a specific set of rules
- Alphabets (or characters)
  - Collection of valid symbols in a language e.g. $\Sigma = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, = \}$:
- Words (or tokens)
  - Valid concatenations of symbols that carry meaning. e.g. 1024
- Grammars
  - Rules that describe well-formed sentences in a particular language

Lexical analysis (1)

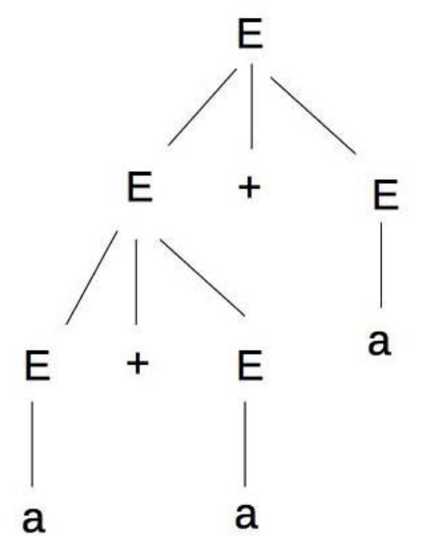
- a.k.a. Tokenisation, is the first step
- Applies a series of regular expressions in order to identify tokens, the fundamental units of meaning in a language
- Examples of tokens include:
  - Number: [0-9]+
  - Identifier: [_a-zA-Z][_0-9a-zA-Z]*
  - Plus: '+'
- Regular expressions ⇔ finite state machines
- Finite state machines can be combined to efficiently apply a set of regular expressions to an input
- Software/function that does this is often called a 'lexer'

Lexical analysis (2)

- Let's take an input: (1.2 + 1) * 2.5
- Tokenising this with some fairly intuitive rules, could yield the following stream of tokens:
  - LPARENS
  - LITERAL(1.2)
  - ADD
  - LITERAL(1)
  - RPARENS
  - MUL
  - LITERAL(2.5)
- This is useful, but it does not unambiguously describe how we should interpret those tokens

Syntax analysis (1)

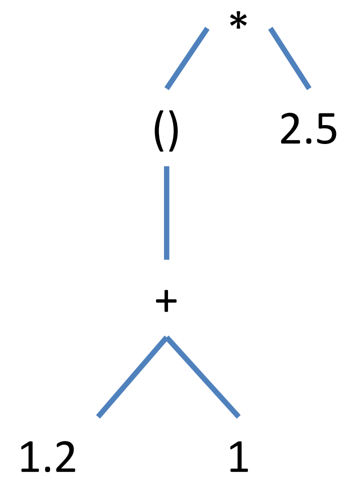
- a.k.a. *Parsing*, is the next step
- We have a stream a tokens representing an input, so we now apply the rules of a grammar to extract meaning, which is encoded within the relative position of tokens, etc
- The result of this is a *parse tree*, which may be evaluated as it is generated, or used to build other structures, such as Abstract Syntax Trees
- Some parser generators also include support for lexical analysis (e.g. ANTLR)



Syntax analysis (2)

- At a high level, this the transformation we want to achieve:

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



Syntax analysis (1)

- Using simple rules, we find structure in the stream of tokens
  - Terminals => UPPERCASE, non-terminals => lowercase
- The rules we'll use are:
  1. expr(B) ⬄ LPARENS expr(A) RPARENS, where B = A
     - Tells us that whatever appears between parentheses has highest precedence
  2. expr(C) ⬄ expr(A) MUL expr(B), where C = A * B
     - After that, multiplication has highest precedence
  3. expr(C) ⬄ expr(A) ADD expr(B), where C = A + B
     - Finally, addition can be applied
  4. expr(B) ⬄ LITERAL(A), where B = A
     - And this shows how terminals can be converted to non-terminals
- How can we apply these rules, efficiently?

LR parsers

- One way to do this is to use an LR parser
  - L -> left to right
  - R -> right-most derivation
- Push symbol onto stack, and look for right-most matches
- An example, that evaluates an expression in-place:

```
(                      #  Push '(' onto stack
( 1.2                  #  Push '1.2' onto stack
( 1.2 +                #  Push '+' on to stack
( 1.2 + 1   ->  ( 2.2  #  Push '1' on to stack; '1.2', '+', '1' are reduced to '2.2'
( 2.2 )     ->  2.2    #  Push '2.2'; Tokens '(', '2.2', ')' are reduced to '2.2'
2.2 *                  #  Push '*'
2.2 * 2.5   ->  5.5    #  Push '2.5'; Tokens '2.2', '*', '2.5' are reduced to '5.5'
```

LR(k) parsers

- This may not be enough...
- Some languages are complex enough that the parser needs to *peek into the future* to unambiguously reduce the input
    - *(C++ is just one of those languages)*
- An LR(k) parser can look ahead up to k future tokens to decide how to behave in the present
- Unfortunately LR(k) parsers can cause exponential growth in the size of the state machine used by the parser
    - So we tend favour parsers that have are LR(1), or smaller
    - We'll see how a parser can exist between LR(0) and LR(1), shortly
- First, we'll see how we can implement a tokeniser and parser in practice

Ragel

- Ragel... is a state machine generator
- A Ragel source file combines a state machine definition, that matches symbols in an input stream, with regular C or C++ source code
- Each regular expressions matched by the FSM can be associated with an action, which is a snippet of C++ code
- Actions are executed in the context in which the FSM was embedded
- Output is a C or C++ file containing code and data that implement a finite state machine

Ragel example

tokeniser.rl (1/3)

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

#include <iostream>
#include <stdexcept>

using namespace std;
```

Ragel example

tokeniser.rl (2/3)

```
void tokenise(const string & input)
{
    // Pointers to configure input stream
    const char * p = input.c_str();
    const char * pe = input.c_str() + input.size();
    const char * eof = pe;

    // Local variables that we can access in actions
    int cs;
    const char * ts;
    const char * te;
    int act;

    // Embed finite state machine
    %% write data;
    %% write init;
    %% write exec;
}
```

Ragel example

tokeniser.rl (3/3)

```
int main()
{
    while (cin) {
        cout << "> ";
        string input;
        getline(cin, input);
        try {
          tokenise(input);
        } catch (const exception & e) {
          cout << "Error: " << e.what() << endl;
        }
    }

    return 0;
}
```

Ragel example

- Now we can compile it:

```
# ragel tokeniser.rl –o tokeniser.cpp
# g++ -o tokeniser tokeniser.cpp
# ./tokeniser
```

```
> 1
LITERAL(1)
> (1.2 + 1) * 2.5
LPARENS
LITERAL(1.2)
ADD
LITERAL(1)
RPARENS
MUL
LITERAL(2.5)
> Wat?
Error: Unexpected character
```

Lemon

- Lemon... is a parser generator, maintained as part of SQLite
- A Lemon source file combines a grammar, that matches token in an input stream, with regular C or C++ source code
- Output is a C file containing code to implement a parser
- Basic algorithm is as follows:
  - Tokens are consumed one-by-one, and added to a stack
  - Rules can be made up from terminals (tokens) and non-terminals, which are simply different kinds of internal nodes in the parse tree
  - When a grammar rule can be unambiguously applied to a sequence of one or more terminals/non-terminals at the top of the stack, a *reduction* will be performed
  - Goal is to reduce input to a single non-terminal

LALR parsers

- Lemon generates an *LALR parser*
  - LA = Look-Ahead
  - LR = Left-to-right, right-most derivation
- An LALR parser can parse most of the grammars supported by an LR(1) parser, without the overhead of an LR(1) parser
- Some grammars are not supported, but in practice, this is not an issue

Lemon example

parser.y (1/2)

```
%include {

// Headers that might be needed for code in parser actions
#include <assert.h>
#include <stdbool.h>

// We use a struct called 'Context' to pass data between invocations of the parser
#include "context.h"

// This file is generated by Lemon, and includes #defines for kind of terminal,
// or token, that will be required by the grammar. Our final lexer will use
// these definitions when generating tokens.
#include "parser.h"

}
```

Lemon example

parser.y (2/2)

```
// Controls operator precedence
%left ADD SUB.
%left MUL DIV.

// Data associated with a node in the parse tree; represented as A, B, C below
%token_type { double }

// Data passed between invocations of the Parse function
%extra_argument { struct Context * context }
%parse_failure { context->error = true; }

// The grammar
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; }
```

Lemon example

context.h

- Defines a simple struct to pass data out of the Parse function
  - We care about the final result
  - But we also care about syntax errors

```
#pragma once

struct Context {
  double result;    // formula ::= expr(A). { context->result = A; }
  bool error;       // %parse_failure { context->error = true; }
};
```

Lemon example

API

```
// These are the C functions that Lemon will generate for us:

void Parse(
    void * parser,      /** The parser */
    int kind,           /** The major token code number */
    double value,       /** The value associated with the token (%token_type) */
    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 */
);
```

Lemon example

Building

- Let's compile it...

```
# lemon parser.y
# g++ -c parser.c
```

- It compiles, but it doesn't do much right now
- We need some code that uses the parser...

Calculator

calculator.rl (1/4)

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

#include <iostream>
#include <stdexcept>
#include "context.h"
#include "parser.h"
```

Calculator

calculator.rl (2/4)

```
extern "C"
{
    void Parse(
        void * parser,      /** The parser */
        int kind,           /** The major token code number */
        double value,       /** The value associated with the token (%token_type) */
        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 */
    );
}
```

Calculator

calculator.rl (3/4)

```
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;
}
```

Calculator

calculator.rl (4/4)

```
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;
}
```

Calculator

Build it

- Let's put it all togeher...

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

Calculator

Try it out

- Start the calculator

```
# ./calculator
```

- Enter some expressions

```
> 1
1
> 1 + 2
3
> (1.2 + 1) * 2.5
5.5
> ((1+ 2) * 1.1 * (-1 * 2))
-6.6
> Wat?
Error: Invalid input.
> ((1+ 2) * 1.1
Error: Invalid input.
>
```

Conclusions

- Once you become familiar with their conventions, parser generators make it easy to implement non-trivial functionality
- With an example like this, you can see how easy it would be to support features such as computed input fields in a GUI app – a useful feature for power users
- However, be warned: once you become familiar with parser generators, everything begins to look like a problem that can be solved with them
  - No one will judge you for writing a handful of functions and switch statements, if that is right thing to do

Resources

- *‘Parsing mathematical expressions’* blog post: http://tristanpenman.com/blog/posts/2019/03/31/parsing-mathematical-expressions/
- Microcalc: https://github.com/tristanpenman/microcalc
- Ragel website: http://www.colm.net/open-source/ragel/
- Lemon website: https://www.sqlite.org/lemon.html
- Zed Shaw’s ‘*Ragel State Charts*’ post: https://zedshaw.com/archive/ragel-state-charts/

Thanks for listening