Parsing with lex/yacc

Simon Richter
@GyrosGeier@mastodon.social

Motivation

- Programs operate on data structures in memory
- How does data get into memory?

History

- In the old days, data was organized on disks in records.
- Programming languages defined how to interpret records.
- This is okay for tabular data, but requires you to define data types for everything, and stick to them.

Idea

- What if we make the on-disk format a bit more flexible and create the records only in memory?
  - Data files become version independent, except if they use newer functions.
  - We can work with things that are not necessarily tables.

Problem

- Data comes from disk as a stream of bytes
  - The meaning of individual bytes depends on *context*.
    ⇒ we need a *state machine* to determine what to do with the data
  - Data comes in chunks that don’t correspond to interpretation boundaries
    ⇒ we need to keep buffers

Recursive Descent

- State machine is implicit
  - The program counter represents the current state
  - The program stack represents the state stack
  - Every token is read by dedicated code
  - Every piece of code that accepts multiple bytes needs to deal with buffer underflows

Recursive Descent

```
read_file() {
   read_header();
   read_body();
}

read_header() {
   read_version();
   read_type();
   read_name();
}

read_version() {
   version = read_int();
   check_version(version);
}

read_type() {
   type = read_int();
   check_type(type);
}

read_name() {
   int length = read_int();
   name = read_string(length);
}

…
```

Recursive Descent

```
int read_int() {
    return
        (get_byte() << 24) |
        (get_byte() << 16) |
        (get_byte() << 8) |
        (get_byte() << 0);
}

string read_string(int length) {
    string ret;
    for(i = 0..length) {
        ret += get_byte();
    }
    return ret;
}
```

Limitations

```
read_variable_declaration() {
    read_var_keyword();
    string var_name = read_word();
    bool has_value = check_equals_sign();
    if(has_value)
        expression value = read_expression();
    …
}

expression read_expression() {
    expression current_expression;
    if(check_int())
        current_expression = read_int();
    else if(check_string())
        current_expression = read_string();
    …
```

Limitations

```
expression read_expression() {
    expression current_expression;
    if(check_int())
        current_expression = read_int();
    else if(check_string())
        current_expression = read_string();
    else
        current_expression = make_variable_reference(read_name());

    if(check_end())
        return current_expression;

    if(check_plus()) {
        skip_plus();
        current_expression = make_plus_expr(
                current_expression, read_expression());
```

Limitations

- So, lots of “check” functions, where we have to leave stuff in the buffer
- What about “a + b * c”?

Tokenizing

- Idea: Stack two state machines
- One of them reads the stream and tells the other what is next

Tokenizing

- When we see digits, we read all of them, and say “that’s an integer”
- When we see an opening quote, we read all the characters until the closing quote and say “that’s a string”
- When we see a plus sign, we say “that’s a plus sign”
- Anything else is just a name

Tokenizing

```
read_expression() {
    switch(get_token_type()) {
        case INT: current_expression = get_int(); break;
        case STRING: current_expression = get_string(); break;
        case NAME: current_expression = make_variable_reference(
                get_name()); break;
        default: ERROR();
    }

    switch(get_token_type()) {
        case SEMICOLON: return current_expression;
        case PLUS: current_expression = make_plus_expr(
                current_expression, read_expression());
        default: ERROR();
    }
}
```

Tokenizing

- Better
- Still doesn’t handle precedence
- Still lots of typing

Precedence

- Idea: Hierarchy of states
- We always try to use the highest precedence operator
- If that doesn't work, we try a lower precedence

a + b * c

- Wrong: (a + b) * c
- Right: a + (b * c)
- Treat “a” as multiplication until it’s clear that there is no “*” there.
- Treat “b * c” as another multiplication
- Then sum up the products

Generating a Parser

- Still, lots of typing
- All the functions kind of look the same
- We can generate them from an easier description

Backus-Naur-Form (BNF)

```
expression: sum

sum: multiplication | sum ‘+‘ multiplication

multiplication: parenthesis | multiplication ‘*‘ parenthesis

parenthesis: term | ‘(‘ expression ‘)‘

term: LITERAL | VARIABLE
```

Generating a Parser

- Lots of generated code
- Lots of states where the next token collapses the state stack quite a bit
- So generated recursive descent is not optimal
- Also, we still need to tokenize

Generating a Lexer

```
LITERAL: /[0-9]+/

VARIABLE: /[a-z][a-z0-9]*/
```

Fitting It All Together

- So, we can now generate a state machine that reads a byte stream and produces a token stream
- We can also generate a state machine that reads a token stream and recognizes BNF
- All we need is to execute some code when we've recognized something

Fitting It All Together

- Tokens can have optional values, like integers or strings
- BNF rules are also tokens, and have values (like the return values in the recursive descent parser)

BNF + Actions

```
expression: sum { $$ = $1 }

sum: multiplication { $$ = $1 } |
      sum ‘+‘ multiplication { $$ = make_sum($1, $3); }

multiplication: parenthesis { $$ = $1 } |
      multiplication ‘*‘ parenthesis { $$ = make_mult($1, $3); }

parenthesis: term { $$ = $1 } | ‘(‘ expression ‘)‘ { $$ = $2; }

term: LITERAL { $$ = $1 } | VARIABLE { $$ = $1 }
```

BNF + Actions

```
expression: sum

sum: multiplication |
       sum ‘+‘ multiplication { $$ = make_sum($1, $3); }

multiplication: parenthesis |
       multiplication ‘*‘ parenthesis { $$ = make_mult($1, $3); }

parenthesis: term | ‘(‘ expression ‘)‘ { $$ = $2; }

term: LITERAL | VARIABLE
```

Lex and Yacc

- Lex generates a Lexer
  - Regular expressions
  - Code fragment when recognized
- Yacc generates a Parser
  - BNF rules
  - Code fragment when recognized

CODE

Advanced: Locations

- The lexer counts characters
- When you recognize a newline, you can just reset the column count and increment the line count, and you know where it is
- This is really useful for error messages

Advanced: Error Handling

- When an unexpected token appears, parsing fails
- You can define rules that have “error” tokens in them that match whenever something cannot be understood
- The action should record the error and try to make sense of the rest

Advanced: Memory Management

- When a parsing error occurs, the current token is discarded, and we go up the stack until we find a rule that has an “error” token here.
- If the current token value is something we allocated, we need to free it
- There are special declarations for that

Advanced: Precedence

```
%prec ‘(‘
%prec ’*‘ ‘/‘
%prec ‘+‘ ‘-‘

expr: ‘(‘ expr ‘)‘ |
       expr ‘+‘ expr |
       expr ‘-‘ expr |
       expr ‘*‘ expr |
       expr ‘/‘ expr |
       term
```

Advanced: Pure Parsers

- You really don’t want global variables in your program
- Especially if you have multiple parsers