pog

Release 0.2.0

Marek Milkovic

Aug 12, 2019

CONTENTS

1 Installation **1**
- 1.1 Getting source code . . . 1
- 1.2 Requirements . . . 1
- 1.3 Compilation . . . 2
- 1.4 Usage . . . 2

2 Writing parser **3**
- 2.1 Principles . . . 3
- 2.2 Tokenization . . . 3
- 2.3 Grammar rules . . . 5
- 2.4 Parsing . . . 6
- 2.5 Examples . . . 7

3 Advanced features **9**
- 3.1 Operator & rule precedence . . . 9
- 3.2 Mid-rule actions . . . 10
- 3.3 Tokenizer states . . . 11
- 3.4 Input stream stack . . . 11

4 Debugging **13**
- 4.1 HTML report . . . 13
- 4.2 LR automaton . . . 13

CHAPTER

ONE

INSTALLATION

This page describes installation of *pog* and how you can integrate it into your own project.

1.1 Getting source code

At first, you'll need to obtain source code. If you want the latest released version, which should always be stable, you can download one from GitHub releases page. If you want to get the latest development version or you are intersted in the development of the library, you can also get the source code using `git`. To clone the repository run:

```
git clone https://github.com/metthal/pog.git
```

1.2 Requirements

In order to use *pog*, you will need:

- C++ compiler with C++17 support
- CMake 3.8+
- re2
- fmt

You can install them from your distribution repositories. For Ubuntu based distributions use:

```
apt-get install libfmt-dev libre2-dev
```

For Red Hat based distributions use:

```
dnf install fmt-devel re2-devel
```

For macOS use:

```
brew install fmt re2
```

There is also an option to download `re2` or `fmt` while building the project. See *Compilation* for more information regarding this.

1.3 Compilation

pog itself is header-only library but it has dependencies which are not header-only. To compile it run:

```
cmake -DCMAKE_BUILD_TYPE=Release [OPTIONS] ..
cmake --build . --target install
```

Other options you can use:

- `POG_DOWNLOAD_RE2` - `re2` will be downloaded during build-time. It will be compiled and installed as `libpog_re2.a` (or `pog_re2.lib` on Windows) together with the library. (Default: `OFF`)
- `POG_DOWNLOAD_FMT` - `fmt` will be downloaded during build-time. It will be compiled and installed as `libpog_fmt.a` (or `pog_fmt.lib` on Windows) together with the library. (Default: `OFF`)
- `POG_TESTS` - Build tests located in `tests/` folder. (Default: `OFF`)
- `POG_EXAMPLES` - Build examples located in `examples/` folder. (Default: `OFF`)

1.4 Usage

`pog` will be installed together with CMake configuration files which make integration into other CMake projects much more easier. If you use CMake in your project put following lines in your `CMakeLists.txt` file and that should be it.

```
find_package(pog REQUIRED)
target_link_libraries(<YOUR_TARGET> pog::pog)
```

For projects which use other build systems, you can use pkgconfig files which are installed too. To obtain which compilation flags are needed run following commands in your shell or integrate it directly into your build system.

```
pkg-config --cflags pog
pkg-config --libs pog
```

To use *pog* from your source code, include file `<pog/pog.h>`. Everything in *pog* is located inside `pog` namespace. Example:

```
#include <pog/pog.h>

int main()
{
    pog::Parser<Value> parser;

    // your parser definition
}
```

CHAPTER

TWO

WRITING PARSER

In this section, we will describe basics on how to write your own parser using *pog*. It is assumed that you have already integrated *pog* into your project. If not, see *Installation*. If you already know how to write parser in *pog* and want to know about more advanced features, see *Advanced features*.

2.1 Principles

Before you start reading this section, make sure that you are familiar with *grammars* in a sense of formal languages. Even though we have tried to write this documentation in a way so that even person without extensive knowledge of formal languages will understand it, it is still expected that reader is somehow familiar with the concepts and at least is aware of them and know what they represent.

When you want to parse something, you need to know the grammar of the language you are trying to parse out and that is also what *pog* expects from you. You need to give it a grammar so it knows the syntax of what it is parsing. Every grammar is made out of rules. While parsing, these rules will be applied to the parts of the input you are parsing. In *pog*, you will be able to specify so called *semantic actions* which are tied to the rules of your grammar. Each time some rule of your grammar will be applied, its semantic action will be performed. That is where your code will come and it will give a meaning to the rules of your grammar. Your actions will however need some data to operate on. Since rules are made out of symbols, there will be data tied the symbols of the right-hand side of the rule and you will be able to tie the result to the symbol on the left-hand side of the rule.

pog splits the process of parsing into two separate procedures in a similar way like flex and bison. So at first, the input is tokenized into tokens using set of regular expressions. Some of the tokens can be simply skipped and never reach the parser at all (like whitespaces or comments) but some may be turned into symbols of your grammar and represented in your grammar rules. This makes the grammar less cluttered. Since it is expected that you might want to perform actions even when a token is found, you will be able to specify actions for tokens and tie some data to the symbol they represent.

The base class on which the whole *pog* stands is `Parser<ValueT>`. `ValueT` here is data type that can be tied to *all* symbols of your grammar and when we say *all* we really mean it. This data type will be tied to each symbol you have in your grammar and you will operate on it in your semantic actions. It will be also expected that each semantic actions will return value of `ValueT` data type. The only constraints are that `ValueT` needs to be default constructible and copyable or movable (move is preferred if it is possible, if not then copy is performed). We do now want to force you to use any specific data type but we recommend using `std::variant`.

2.2 Tokenization

You start defining your parser with tokens. You need to provide regular expression that describes how the recognize the token on the input. The syntax for regular expression is Perl-like. Here is an example on how to define tokens for recognizing boolean, integer literals and string literals (without escape sequences).

```
using Value = std::variant<int, bool, std::string>;
pog::Parser<Value> parser;

parser.token("\\s+");                    // Token for skipping whitespaces
parser.token("=");                       // Token for single '='
parser.token("(true|false)");            // Token for boolean value
parser.token("[0-9]+");                  // Token for integer value
parser.token(R"("[^"]*")");              // Token for string value (you might find raw␣
↪string literals useful)
parser.token("[a-zA-Z_][a-zA-Z0-9_]*"); // Token for identifiers
```

This will allow the parser to recognize what is on the input. If there is anything that it is not able to tokenize it is reported as syntax error. Tokenization also takes place only at the very start of the input so there is no automatic skipping of unknown characters. Once the token is read, the characters that were matched with the regular expression are consumed and the new start of the input starts at the end of the matched token.

There might be a situation when multiple regular expression match the same input. For example, consider regular expressions `(true|false)` and `[a-zA-Z_][a-zA-Z0-9_]*`. They both would match on `true` or `false` but they would also match on `trueaaa`. We certainly know that `true` and `false` should be tied to the first regular expression while `trueaaa` the second one but tokenizer does not know that. To resolve these issues, tokenizer always prefers the longest possible match. If there are multiple matches of the same length then the token which is specified earlier in the source code is chosen. There is also an another option to resolve this issue but it requires your interaction. If you specify token as `fullword` then the regular expression you provide will be replaced by `<YOUR_REGEXP>(\b|$)`. In our example, `(true|false)` would not match on `trueaaa` in that case.

```
parser.token("(true|false)").fullword();
```

By specifying these regular expressions, our input is now recognized but we should be able to specify what symbol in our grammar will these tokens represent. We do that with `symbol` method.

```
using Value = std::variant<int, bool, std::string>;
pog::Parser<Value> parser;

parser.token("\\s+");
parser.token("=").symbol("=");
parser.token("(true|false)").symbol("bool");
parser.token("[0-9]+").symbol("int");
parser.token(R"("[^"]*")").symbol("string");
parser.token("[a-zA-Z_][a-zA-Z0-9_]*").symbol("id");
```

Attention: Do not use symbol names prefix with either `@` or `_`. Those are reseved for internal purposes of the parser. Proper working of the parser is not guaranteed in such case.

As you can see, we haven't specified any symbol for the first token since we are not interested in whitespaces in our grammar. This way, all whitespaces will be automatically skipped and we can only focus on our 3 symbols which we have specified - `bool`, `int` and `string`. Once we have symbols, we would also like to take the actual boolean value, digits of a number or characters of a string and tie it to our symbol. You specify the action with `action` method which expects callable object that accepts `std::string_view` containing the part of the input that was matched with that specific regular expression and returns value of `Value` type.

```
using Value = std::variant<int, bool, std::string>;
pog::Parser<Value> parser;

parser.token("\\s+");
```

(continues on next page)

(continued from previous page)

```
parser.token("=").symbol("=");
parser.token("(true|false)")
  .symbol("bool")
  .action([](std::string_view str) -> Value {
    return str == "true";
  });
parser.token("[0-9]+")
  .symbol("int")
  .action([](std::string_view str) -> Value {
    return std::stoi(std::string{str});
  });
parser.token(R"("[^"]*")")
  .symbol("string")
  .action([](std::string_view str) -> Value {
    return std::string{str.begin() + 1, str.end() - 1};
  });
parser.token("[a-zA-Z_][a-zA-Z0-9_]*")
  .symbol("id")
  .action([](std::string_view str) -> Value {
    return std::string{str};
  });
```

Token for = does not need any action because it itself doesn't bear any value. We only need an information that it is located on the input. You might also need to perform some action whenever end of an input is reached. In that case you can use `end_token` method.

```
parser.end_token().action([](std::string_view str) -> Value {
  // some action
  return {};
});
```

2.3 Grammar rules

Once your input is tokenized, you may start with specfying grammar rules. Let's define grammar rule for assignment of boolean, integer or string literal value to an variable.

```
// var_init -> id = literal
parser.rule("var_init")
  .production("id", "=", "literal");
// literal -> bool | int | string
parser.rule("literal")
  .production("bool")
  .production("int")
  .production("string");
```

Se `rule()` method expects symbol on the left-hand side of the rule and can have multiple productions with multiple symbols on the right-hand side (even none). If we wanted to write rules for languages which would represents 0 or more variable initializations, we could write that as (is usual way to denote empty string in formal languages):

```
// var_init_list -> var_init_list var_init |
parser.rule("var_init_list")
  .production("var_init_list", "var_init")
  .production();
```

To specify action that should be performed when the rule is applied, you put it directly into the production at the very end.

```
// var_init -> id = literal
parser.rule("var_init")
  .production("id", "=", "literal", (auto&& args) -> Value {
    // args[0] == value tied to 'id'
    // args[1] == value tied to '='
    // args[2] == value tied to 'literal'

    // Make sure that 'id' is declared
    // Assign args[2] to variable specified by args[0]
    return {};
  });
// literal -> bool | int | string
parser.rule("literal")
  .production("bool", [](auto&& args) -> Value { return std::move(args[0]); })
  .production("int", [](auto&& args) -> Value { return std::move(args[0]); })
  .production("string", [](auto&& args) -> Value { return std::move(args[0]); });
```

Action accepts `std::vector<Value>` as its parameters that contains values tied to the symbols of right-hand side of the rule. Returned value from the action is tied to the symbol on the left-hand side of the rule.

Every grammar also needs to have some symbol which is used as a starting point for the whole parsing. That would be the symbol from which you are able to generate the whole language you are able to parse. You need to specify this symbol explicitly using `set_start_symbol()` method.

```
parser.set_start_symbol("var_init");
```

2.4 Parsing

Once you've specified all tokens and grammar rules, you are almost ready to start parsing. But before, you need to *prepare* your parser. This action will take your tokens and grammar rules and will build parsing table. It will return report that will either mean success or some kind of problem with the grammar. To check the result, simply treat is as boolean value. You may also iterate over it with range-based for loop to obtain issues found in the grammar. These issues can be:

- *Shift-reduce* conflict
- *Reduce-reduce* conflict

These conflicts are specific to shift-reduce parser which *pog* also generates. Those are types of parser which use stack to which they either shift values they see on the input or they *reduce* the stack by popping out multiple values out of stack and shift another value (this represents application of a grammar rule). Whenever parser is not able to decide whether to shift some value to the stack or to reduce using certain rule, it means *shift-reduce* conflict. If the parser is not able to decide whether to reduce by rule *A* or rule *B*, it is *reduce-reduce* conflict. There isn't a simple cookbook on how to fix these conflicts in your grammar (and sometimes it is not even possible) so if you run into one, you might want to learn more about parsers and how these conflicts occur to resolve them. The fact that these issues are in your grammar does not mean that you cannot use your parser. You may still be able to parse out your language but you might not be able to parse certain constructs of your language and will receive syntax errors. There are however cases in which these conflicts can be completely ignored.

After preparing your parser, you are ready to parse the input using method `parse()`. It accepts input stream (such as `std::istream`) and returns `std::optional<ValueT>`. In case of a successful parsing, the returned value will contain what was tied to the starting symbol of the grammar. When syntax error occurrs, `parse()` raises an

exception of type `SyntaxError`. In some corner cases which are not covered by syntax errors but might represent internal failure of the parser, the returned optional value will be empty.

```
auto report = parser.prepare();
if (!report)
{
  for (const auto& issue : report)
    std::cerr << issue.to_string() << std::endl;
}

std::stringstream input(/* your input */);
try
{
  auto result = parser.parse(input);
  if (!result)
  {
    std::cerr << "Error" << std::endl;
    return;
  }

  std::cout << "Parsed value " << result.value() << std::endl;
}
catch (const SyntaxError& err)
{
  std::cerr << err.what() << std::endl;
}
```

2.5 Examples

Check examples folder to see some simple examples of existing parsers.

CHAPTER

THREE

ADVANCED FEATURES

3.1 Operator & rule precedence

Typical features of many languages are operators such as arithmetic operators, bitwise operators, relational operators and many others. These operators have certain precedence and associativity. That also complicates parsing process because if we for example take `2 + 3 * 4 - 5`, the order of evaluation is important here. The usual way to model this in grammars is to specify all of this directly in grammar rules but this might seriously complicate the whole grammar. This is a grammar you would usually write for addition and multiplication expressions with support for parentheses.

$$E ß E + E$$
$$E ß E * E$$
$$E ß (E)$$
$$E ß number$$

This doesn't however incorporate any precedence whatsoever. Here is a grammar with resolved precedences.

$$E ß E + T$$
$$E ß T$$
$$T ß T * F$$
$$T ß F$$
$$F ß (E)$$
$$F ß number$$

Now imagine this situation with much more operators and much more precedence levels. *pog* shifts the responsibility of handling precedences from the user to itself. At first, you need to define precedence level and associativity of the token you consider as operators.

```
parser.token("\\+").symbol("+").precedence(1, Associativity::Left);
parser.token("-").symbol("-").precedence(1, Associativity::Left);
parser.token("\\*").symbol("*").precedence(2, Associativity::Left);
parser.token("/").symbol("/").precedence(2, Associativity::Left);
```

Level of the precedence is just an unsigned integer. The higher the number, the greater the precedence. If two operators have the same precedence then associativity comes in to resolve this situation. During parsing, precedence is resolved at the moment of operator symbol being the next token on the input. To resolve which symbol to compare against, the right-most terminal symbol of the rule which would be currently reduced is considered as operator.

Let's imagine it on an example. You are currently in the state of the parser where you are deciding whether to reduce by rule $E ß E + E$ or shift the following symbol. The next symbol on the input is $*$. Right-most terminal in $E + E$ is $+$ so we should rather shift than reduce because multiplication needs to be evaluated before addition.

Precedence cannot only be assigned to tokens (or more presicely symbols tied to tokens) but also rules. If the rule has priority assigned then this priority is considered rather than priority of the right-most terminal. The case when this can be useful is for example for unary minus. Unary minus uses the same symbol as subtraction but unary minus has greater precedence than for example multiplication or division.

```
parser.token("\\+").symbol("+").precedence(1, Associativity::Left);
parser.token("-").symbol("-").precedence(1, Associativity::Left);
parser.token("\\*").symbol("*").precedence(2, Associativity::Left);
parser.token("/").symbol("/").precedence(2, Associativity::Left);
parser.token("[0-9]+").symbol("number").action(/* action */);

parser.rule("E")
  .production("E", "+", "E")
  .production("E", "-", "E")
  .production("E", "*", "E")
  .production("E", "/", "E")
  .production("-", "E")
    .precedence(3, Associativity::Right)
  .production("number");
```

Whenever you specify precedence like this, it is always tied to the last production you have specified.

3.2 Mid-rule actions

Most of the times, it is enough to have a rule action being preformed after the whole rule is applied (or rather *reduced* since we are in advanced section). There might be some cases when there is a need for so called *mid-rule action*. That is action performed after only certain part of the rule has been parsed out. This comes as a problematic since the parser cannot know whether it is in the middle of parsing certain rule. It will know which rule to reduce after it has parsed its whole right-hand side (and also depending on the next input symbol).

pog tries to solve this by internally splitting your rule into more smaller rules to achieve this. Let's take for example rule `func -> function id ( args ) { body }`. Usually you would write it as:

```
parser.rule("func")
  .production("function", "id", "(", "args", ")", "{", "body", "}",
      [](auto&& args) { /* action */ }
  );
```

But you might want to check whether name of the function does not collide with some other already defined function before you start parsing the function body to exit early. You can write it as following:

```
parser.rule("func")
  .production(
      "function", "id",
          [](auto&& args) { /* action 1 */ },
      "(", "args", ")", "{", "body", "}",
          [](auto&& args) { /* action 2 */ }
  );
```

Internally it would look like this:

```
parser.rule("func")
  .production("_func#0.0", "_func#0.1",
      [](auto&& args) { /* passthrough last value */ }
  );
```

(continues on next page)

(continued from previous page)

```
parser.rule("_func#0.0")
  .production(
      "function", "id",
          [](auto&& args) { /* action 1 */ }
  );
parser.rule("_func#0.1")
  .production(
      "(", "args", ")", "{", "body", "}",
          [](auto&& args) { /* action 2 */ }
  );
```

This comes with some disadvantages. Since internally, rule is being split, it can introduce *shift-reduce* conflicts which weren't there before. You also loose access to the values of symbols that were covered by previous mid-rule actions, so for example *action 2* in example above wouldn't have access to `function` nor `id` since they are covered by *action 1*. Also keep in mind that value from all mid-rule actions is lost and cannot be recovered. The left-hand side symbol will always be assigned value from the end-rule action.

3.3 Tokenizer states

Tokenizer has set of regular expressions and matches them all against the start of the input. However, it might be sometimes unncessary to match every single regular expression or even impossible to design such that it doesn't collide with other regular expressions and always returns you the right token you want. For this purpose, you can define tokenizer states and transition between those states as you want. Tokenizer can be in a single state at the time and can transition to any other state. Regular expression of token can be in active in multiple states at once. States are represented using string literals. Default state is called `@default`. This is for example useful for tokenizing string literals with escape sequences. Upon reading `"` from input, you can enter special state which reads characters one by one and whenever runs into escape sequences like `\n`, `\t` or any other, then it appends corrent escaped character to the string. Upon reaching ending `"`, we enter default state. While we are tokenizing this string literal, there is no reason to match all other regular expressions for other tokens because we know we are in a specific context in which characters have other special meaning.

```
p.token("a"); // only active in default state
p.token("b")
  .states("state1", "state2"); // only active in states state1 and state2
p.token("c") // only active in default state
  .enter_state("state1"); // causes transition to state1
p.token("d")
  .states("state1") // only active in state1
  .enter_state("@default"); // causes transition to default state
```

3.4 Input stream stack

Parser in *pog* is capable of working with multiple inputs. When you call `parse()` method with some input stream, what actually happens is that this stream is pushed onto input stream stack. You are able to control this input stream stack with methods `push_input_stream()` and `pop_input_stream()`. Whenever parser asks tokenizer for the next token, it will be always returned from the top-most input stream on the stack. End token actions are still performed when we reach the end of the top-most input stream but end symbol is not passed to the parser until we reach the very last input stream on the stack. So end symbol is passed down to parser only if the input stream stack is empty or we reach the end of the top-most input stream without anyone popping it. This can be useful for implementing things like `include` of another file.

```
static std::vector<std::string> input_streams = {
  "<stream 1>",
  "<stream 2>",
  "<stream 3>",
  "<stream 4>"
};

// You need to ensure that lifetime of stream is longer than its use in parser
std::vector<std::unique_ptr<std::stringstream>> inputs;

p.token("include [0-9]+").action([&](std::string_view str) {
  auto stream_idx = std::stoi(std::string{str.data() + 8, str.end()}); // skip
↪'include '
  inputs.emplace_back(std::make_unique<std::stringstream>(input_streams[stream_idx]));
↪ // create stream
  p.push_input_stream(*inputs.back().get()); // push it onto input stack
  return 0;
});
p.end_token().action([&](std::string_view) {
  p.pop_input_stream();
  return 0;
});
```

In the example above you can see how to implement basic include-like functionality that allows you to include one of `input_streams` by their index. It also works recursively out of the box. Be aware that you need to ensure the lifetime of your input stream is longer than its use in parser because parser does not take ownership of your streams.

CHAPTER

FOUR

DEBUGGING

Debugging errors in parser can be a hard process since its whole complexity and the amount of data and code you need to go through to even reach the source of your problems. This sections describes some options you have to debug the problems in your parser.

4.1 HTML report

You are able to generate HTML report out of your parser. The output is HTML file which contains full parsing table, contains information about LR automaton and it even includes Graphviz representation of LR automaton. In order to generate HTML report, initialize it with your `Parser` after you've prepared it.

```
Parser<Value> parser;

// tokens & rules

// Don't forget to first call prepare()
parser.prepare();

HtmlReport html(parser);
html.save("parser.html");
```

You can now open `parser.html` in your current working directory and inspect the inner structure of your parser.

4.2 LR automaton

Generated HTML report will contain Graphviz representation of LR automaton at the very bottom. Copy it over to some other file and run following command to turn it into PNG image.

```
dot -Tpng -o automaton.png <INPUT_FILE>
```