Pegpp, a C++ PEG library

Table of Contents

The PEG library

PEG (Parsing Expression Grammar) is a formalism for describing recursive descent parsers with unlimited lookahead and backtracking. The theory of PEGs can be found in Wikipedia or in Bryan Ford's original paper:

https://en.wikipedia.org/wiki/Parsing_expression_grammar
http://bford.info/pub/lang/peg.pdf.

Like YACC, which is a compiler of context-free grammars, there are compilers of PEG grammars. The LEG tool by Ian Piumarta is a good example. LEG takes a PEG grammar with embedded actions and generates a parser in C.

Pegpp is inspired by LEG and tries to mimic its syntax, but it is not a compiler. It is a library that allows embedding PEG grammars in C++ code and parsing them directly without intermediate code generation.

The way of handling the value stack was inspired by YACC.

The library is a single header file (peg.h) and requires C++17.

PEG grammars

A PEG grammar is a set of rules (non-terminals). Each rule is assigned a parsing expression, or just "expression", for short:

```
rule1 = expression1;
rule2 = expression2;
...
```

A PEG parser tries to match its input against one of the rules, selected as the *start rule*. Parsing a rule means parsing the expression assigned to it. The parser reads input and advances its input pointer as long as the input matches the parsed expression. During this process the parser may schedule actions to be executed after a successful parse. If parsing does not succeed, the parser backtracks by moving the input pointer backwards to where it was before the unsuccessful attempt and canceling the scheduled actions.

Basic parsing expressions:

- Lexical primitives
- Rules
- Embedded actions
- Semantic predicates

Lexical primitives match some simple input patterns. If they parse successfully they advance the input pointer over the matched input.

Any() matches any character. It only fails when no character can be read, i.e. at end of file.

Lit(c) matches a single character c.

Lit(s) matches the character sequence in string s.

Ccl(s) matches any character contained in the character class defined by string s. Character – is special. If it appears between two other characters in s it defines a character range, otherwise, if it is the first or last character in s, it just represents itself. Ccl(" \t\r\n\f") may be a definition of white space. Ccl("0-9") is the class of decimal digits. If s begins with ^, the class is complemented or negated. The ^ is removed from s and the class contains all characters not included in the rest of s. Thus Ccl("^") is a class that contains all characters and matches any. It is equivalent to Any().

Rules are named expressions.

Embedded actions do not consume input and always succeed. They schedule actions to be executed after a successful parse. Actions scheduled during an unsuccessful parse are cancelled.

Do(f) schedules an action f. It takes an argument of type std::function<void()>, i.e. a function, function object or lambda that takes no arguments and returns nothing.

Semantic predicates do not consume input. They execute actions during parsing, which may cause the predicate to fail or succeed depending on certain semantic conditions.

Pred(f) takes an argument of type std::function<void(bool &)>, i.e. a function, function object or lambda that receives a reference to a boolean variable and returns nothing. The function passed to Pred() is executed when the predicate is parsed, and parsing succeeds or fails, depending on the value of the boolean variable when the function returns. The variable is initialized to true (success) before calling the function, so that Pred(f) succeeds by default if f does not modify its argument.

If e, e1 and e2 are expressions, other expressions may be obtained by using some operators:

- Text capture --e e--
- Repetition e[{n1, n2}] e[{n}] e[n] ~e *e +e
- Syntactic predicate &e !e
- Sequence e1 >> e2
- Attachment e1 (e2)
- Choice e1 | e2

Text capture

The expressions --e and e-- succeed or fail as e does. If they succeed they capture the text consumed by e. The captured text may be used by Do() or Pred() actions following the capture. Captures may be nested (which is useful for debugging).

Repetition

The general repetition expression e[{n1, n2}] tries to parse e no less than n1 and no more than n2 times. It succeeds if e is parsed at least n1 times. After the first n1 times, the parser keeps trying to parse e until a maximum of n2 times.

The special case n2 == 0 actually means n2 == ∞. There is no upper limit and the parser keeps trying to parse e indefinitely.

e[{n}] and e[n] are equivalent to e[{n, n}]. If n != 0 they try to parse e exactly n times. For n == 0 they try to parse e as many times as possible and always succeed.

~e is equivalent to e[{0, 1}]. It tries to parse e once and always succeeds.

*e is equivalent to e[{0, 0}]. It tries to parse e as many times as possible and always succeeds.

+e is equivalent to e[{1, 0}]. It tries to parse e as many times as possible and succeeds if e is parsed at least once.

Syntactic predicate

Syntactic predicates are lookahead mechanisms that check if some expression e can be parsed at the current point. This is done by actually trying to parse e and backtracking if parsing succeeds, so that input eventually consumed by e is returned to the input stream and actions eventually scheduled by e are canceled.

&e ("and-predicate") succeeds if e can be parsed at the current point.

!e ("not-predicate") succeeds if parsing e at the current point would fail. This kind of predicate is most frequently used as a stop condition. !Any() means end of file.

Sequence

In the original LEG syntax sequences are built by concatenation. C++ requires an operator, so >> was chosen. Parsing e1 >> e2 succeeds if both e1 and e2 are parsed successfully and in that order. If e1 fails, e2 is not parsed.

Attachment

An attachment e1 (e2) parses like the sequence e1 >> e2, but e2 does not occupy any slots in the value stack. This will be explained later.

Choice

Parsing the ordered choice e1 | e2 starts by parsing e1. If e1 succeeds, the expression succeeds without parsing e2. If e1 fails, the result is obtained by parsing e2. This short-circuit evaluation makes PEG grammars deterministic, since alternatives are always tried in fixed left-to-right order.

This table lists the available operators grouped by decreasing order of priority:

- e-- e1(e2) e[{n1, n2}] e[{n}] e[n]
- --e &e !e ~e *e +e
- e1 >> e2
- e1 | e2

Parentheses may be used for grouping in order to override priorities.

Note that unary operator & has been overridden, so that &e means "e can be parsed at this point" and not "the address of e". In the unlikely case that the address of an expression is needed, use std::addressof.

A simple example

This little example is a direct translation of a LEG parser taken from the LEG distribution. It copies its input to its output, replacing occurrences of the string "username" by the current user's login name.

The original LEG source:

```
%{
    #include <unistd.h>
    #include <stdio.h>
%}

start   = "username"      { printf("%s", getlogin()); }
        | < . >           { putchar(yytext[0]); }

%%

int main()
{
    while ( yyparse() )
        ;
    return 0;
}
```

The same parser using pegpp:

```
#include <unistd.h>
#include <iostream>

#include "peg.h"

using namespace std;
using namespace peg;

class parser : public Parser<>
{
    Rule start;

public:

    parser(istream &in = cin) : Parser(start, in)
    {
        start   = "username"_lit        do_( cout << getlogin(); )
                | Any()--               do_( cout << text(); )
                ;
    }
};

int main()
{
    parser p;
    while ( p.parse() )
        p.accept();
}
```

This parser has only one rule (start). The empty parameter list in Parser<> indicates that it does not use a value stack; if it did, the types of the values returned by the rules would be listed between the angle brackets.

The constructor receives an argument with the input stream to read characters from, by default std::cin (standard input). The base class (Parser) needs to be initialized with the grammar's start rule (start) and the input stream, which also defaults to std::cin if not passed.

The "username"_lit construct is a user-defined literal returning Lit("username"). It matches the string "username" literally. Characters not matching "username" are matched individually by Any().

Embedded actions are placed in the grammar using the do_(...) construct, which replaces the pair of braces { ... } used in the original LEG syntax. This construct is a macro, which expands to (peg::Do([&]{ ... })), i.e. it schedules a capture-all-by-reference lambda. Note that the Do() expression is surrounded by parentheses. This makes it attach to the preceding expression, making an intermediate >> operator unnecessary.

Operator -- captures the text matched by Any(), which is accessed by calling the text() method inherited from the Parser class.

The parser is executed by calling its parse() method. When parse() returns, accept() is called to execute the actions scheduled while parsing and discard the consumed input. It is not necessary to call accept() every time parse() returns, as is done here. Matched input and scheduled actions just queue up in the parser until accept() is called.

A calculator

This simple integer calculator supports the four basic operations and grouping with parentheses:

```
#include <iostream>
#include <string>

#include "peg.h"

using namespace std;
using namespace peg;

class calculator : public Parser<int>
{
    Rule WS, SIGN, DIGIT, NUMBER, LPAR, RPAR, ADD, SUB, MUL, DIV;
    Rule calc, expression, term, factor;

public:

    calculator(istream &in = cin) : Parser(calc, in)
    {
       // Lexical rules

       WS             = *" \t\f\r\n"_ccl;
       SIGN           = "+-"_ccl;
       DIGIT          = "0-9"_ccl;
       NUMBER         = (~SIGN >> +DIGIT)-- >> WS   do_( val(0) = stoi(text()); );
       LPAR           = '(' >> WS;
       RPAR           = ')' >> WS;
       ADD            = '+' >> WS;
       SUB            = '-' >> WS;
       MUL            = '*' >> WS;
       DIV            = '/' >> WS;

       // Calculator

       calc           = WS >> expression            do_( cout << val<int>(1) << endl; )
                      ;
       expression     = term >> *(
                                 ADD >> term         do_( val<int>(0) += val<int>(2); )
                               | SUB >> term         do_( val<int>(0) -= val<int>(2); )
                               )
                      ;
       term           = factor >> *(
                                 MUL >> factor       do_( val<int>(0) *= val<int>(2); )
                               | DIV >> factor       do_( val<int>(0) /= val<int>(2); )
                               )
                      ;
       factor         = NUMBER
                      | LPAR >> expression >> RPAR  do_( val(0) = val(1); )
                      ;
    }
};

int main()
{
    calculator c;
    while ( c.parse() )
        c.accept();
}
```

The rules

Rules are instances of the Rule class. Assigning an expression to a rule builds a syntax tree, dynamically allocating memory for the structures in the nodes of the tree. Smart pointers of the type std::shared_ptr take care of releasing memory when the rules are no longer used.

When a grammar is built, rules may be assigned to in any arbitrary order. Any rule can refer to itself or to any other rule, whether it has already been initialized or not. This is necessary to build recursive grammars.

When used as a parsing expression, a rule behaves as a reference to itself. For this reason no Rule temporaries can be created, since they would produce expressions with dangling references. The Rule class has no constructors except the default, and the copy constructor has been deleted.

Rules must be defined as uninitialized variables. The grammar is built by assigning parsing expressions to the rules.

Left recursion

The copy assignment operator of rules is non-standard. Normally, the statement r = r is supposed to do nothing. However, if r is a rule, it actually makes r left-recursive, meaning that the recursive-descent parser for r calls itself directly without reading input. Parsing such a rule enters an infinite recursive loop and overflows the stack. This is an example of direct left recursion, where a rule is assigned an expression that starts by trying to parse the same rule without reading any input.

Left recursion can also be indirect, originating in cyclic references. The calculator has the following reference cycle:

expression → term → factor → expression

The first two references (expression → term → factor) are direct. The third one (factor → expression) closes the cycle but only after reading a LPAR (left parenthesis) token. Without this LPAR the grammar would be left recursive. This is an inherent limitation of PEG grammars: reference cycles that may close without consuming at least one character from the input are forbidden.

Parsing the grammar

Once built, a grammar is used by calling method parse() of the parser. It returns true if the input matches the grammar, and false otherwise. When parse() returns the parser holds in an internal buffer all input read so far, and an input pointer pointing to the first character not matched. For example, let’s assume that the calculator is fed with this input:

(1 + 2) * 3 Hello

The first call to parse() returns true. Now the parser has advanced its input pointer to the H in Hello. The parser also holds a vector of actions scheduled by Do() during parsing. These actions must be executed in order to evaluate the calculation. Calling accept() executes the scheduled actions, resets the vector that holds them and discards the consumed part of the input. Input now starts at the H in Hello. The next call to parse() fails, since Hello does not match the grammar, and the input pointer does not advance.

The value stack

Expressions in a rule may return a value in a slot of the value stack, according to their position in the sequence. Rule actions (and semantic predicates) may return a value by assigning val(0). If it does not assign val(0), a rule returns the value of its first expression.

The calculator inherits from Parser<int>, meaning that the value stack will hold elements of type int. Reading a value in the stack requires the val<T>(n) syntax, although in this case T can be only int.

The index of each expression in a rule is easily calculated by counting how many >>’s separate it from the start of the rule. For example, in rule term of the calculator, the first factor is in position 0 and the second one in position 2. If an expression of the rule has alternatives, it uses as many stack slots as its longest alternative. For example:

```
r = e0 >> (
             a1 >> a2
           | b1 >> b2 >> b3 >> b4
           )                        >> e5;
```

The expression between parentheses uses four stack slots. The result of e5 always appears in val(5), no matter which alternative matches when parsing (a1 ... a2 or b1 ... b4).

This way of handling the value stack is similar to YACC’s. Our val(0) is equivalent to YACC’s $1 and $$, val(1) is YACC’s $2, etc.

Implicit lexical primitives

Many of the operators used to build the rules are overloaded so that they can accept values of the types taken by the lexical primitives, and automatically promote them to the respective primitive. The assignment operator of class Rule is similarly overloaded.

std::string s	converts to Lit(s)
const char *s	converts to Lit(std::string(s))
char c	converts to Lit(c)
std::function<void()> f	converts to Do(f)
std::function<void(bool &)> f	converts to Pred(f)

In many cases these automatic conversions allow replacing explicit calls to the primitives by their arguments. In the calculator we have eliminated explicit calls to Lit(), for example. It is not possible to eliminate calls to Ccl(s), since a standalone string s converts to Lit(s), not to Ccl(s).

Binary operators require that at least one of the arguments be an explicit expression; the other may convert automatically. Prefix and postfix operators have to be applied to an explicit expression.

Calls to Lit(s), Lit(c) and Ccl(s) with literal arguments may also be replaced by user-defined literals:

"ab"_lit	is Lit(std::string("ab", 2))
'c'_lit	is Lit('c');
"a-z"_ccl	is Ccl(std::string("a-z", 3))

User-defined string literals allow embedded null characters, since the compiler passes the length of the literal to the operator. The literal operators are defined in inline namespace peg::literals.

Action and semantic predicate macros

Peg.h defines some macros for using lambdas as embedded actions and semantic predicates with a compact notation:

```
#define do_(...)     (peg::Do([&]{ __VA_ARGS__ }))
#define pa_(...)     (peg::Pred([&](bool &){ __VA_ARGS__ }))
#define pr_(...)     (peg::Pred([&](bool &__r){ __r = [&]()->bool{ __VA_ARGS__ }(); }))
#define if_(...)     (peg::Pred([&](bool &__r){ __r = (__VA_ARGS__); }))
```

The do_(...) macro defines a scheduled action. The argument is the body of a void() function.

The other macros define semantic predicates, which execute at parsing time.

> The argument of pa_(...) is the body of a void() function. This predicate always succeeds.
>
> The argument of pr_(...) is the body of a bool() function. This predicate succeeds if the function returns true, and fails otherwise.
>
> The argument of if_(...) is a boolean expression. This predicate succeeds if the value of the expression is true, and fails otherwise.

The following rules are equivalent and always fail:

```
Rule fail1, fail2;
fail1 = if_( false );
fail2 = pr_( return false; );
```

These macros expand to parenthesized expressions. If they are placed immediately following another expression, they attach to it, because the parentheses are interpreted as operator(). Otherwise, the parentheses are redundant and have no effect.

Attaching actions and semantic predicates is very useful because it allows adding or removing them anywhere in the rules without shifting the positions of the rule's expressions in the value stack.

These macros are meant to streamline notation and make grammars more readable, but they are optional. Do() and Pred() expressions can always be written, either directly or implicitly:

```
Rule fail, succeed;
fail = [](bool &r){ r = false; };
succeed = [](bool &){ };
```

Note that a void(bool &) lambda is implicitly converted to Pred(f) when assigned to a rule, and that the predicate succeeds if f ignores its argument.

Consider this simple palindrome recognizer as a use case for semantic predicates and parsing time actions:

```
#include <string>
#include <iostream>

#include "peg.h"

using namespace std;
using namespace peg;

class parser : public Parser<string>
{
    Rule start, pal, chr;

public:

    parser(istream &in = cin) : Parser(start, in)
    {
        start   = pal--                 do_( cout << text() << endl; )
                ;
        pal     = chr >> pal >> chr     if_( val(0) == val(2) )
                | chr >> chr            if_( val(0) == val(1) )
                | chr
                ;
        chr     = Any()--               pa_( val(0) = text(); )
                ;
   }
};

int main()
{
    parser p;
    while ( p.parse() )
        p.accept();
}
```

The pal rule defines a palindrome as a symmetric string, including single characters. For lengths > 1 the symmetry is enforced by two if_(...) macros. Characters read from the input are placed in the value stack by the pa_(...) macro. In this parser the value stack is only used during parsing. At execution time, the do_(...) macro in rule start outputs a direct text capture of each recognized palindrome, followed by a newline. Recognized palindromes are not necessarily the longest possible, especially when the input contains sequences of repeated characters.

Checking a grammar

If the macro PEG_DEBUG is defined before including peg.h, the rules are compiled with a method that allows checking the grammar by calling method check() after the grammar is built.

Check() visits all grammar paths originating in the starting rule and throws an exception of type Rule::bad_rule if it detects uninitialized rules or potential left recursion, i.e. closed paths that may be traversed without consuming input.

It cannot be called more than once on the same grammar, which would be pointless anyway. Once the grammar is checked, both the macro and the call to check() can be removed.

In debug mode each rule may be assigned a name (a const char *). For example,

```
calc.name = “calc”;
```

And peg.h defines the macro peg_debug() as follows:

```
#define peg_debug(rule) rule.name = #rule
```

Rules that have debugging names assigned will print messages to standard error when visited by check(). For example, we define PEG_DEBUG in the calculator. In the constructor, after the grammar is built, we write:

```
peg_debug(NUMBER);
peg_debug(LPAR);
peg_debug(RPAR);
peg_debug(ADD);
peg_debug(SUB);
peg_debug(MUL);
peg_debug(DIV);
peg_debug(calc);
peg_debug(expression);
peg_debug(term);
peg_debug(factor);

check();
```

When the parser is instantiated, we get the following output on standard error:

```
calc
| expression
| | term
| | | factor
| | | | NUMBER
| | | | LPAR
| | | | expression (r)
| | | | RPAR
| | | MUL
| | | factor (v)
| | | DIV
| | | factor (v)
| | ADD
| | term (v)
| | SUB
| | term (v)
calc: check OK
```

The name of each rule is printed when visited, with an indentation that reflects the level of nesting. The syntax tree of each rule is visited only once. Rules that have already been visited are marked (v). Recursive calls are marked (r).

Now we introduce a small change in the grammar: in rule factor we make LPAR optional (~LPAR). This change makes the grammar left-recursive, as the cycle expression → term → factor → expression can now close without consuming input. Check() detects the problem:

```
calc
| expression
| | term
| | | factor
| | | | NUMBER
| | | | LPAR
| | | | expression (r)
terminate called after throwing an instance of 'peg::Rule::bad_rule'
  what():  Left-recursive rule
```

Some low-level lexical rules like WS have not been debugged, because they would clutter the output of the debugger. Note that check() always checks the whole grammar, independently of which rules are debugged, if any. Normally check() is first called without debugging. If the grammar checks OK, nothing else need be done. If check() throws, some rules may be gradually added to the debugger, until the cause of the problem is clear.

Unicode support

Pegpp supports Unicode.

Strings and input streams should be encoded in UTF8. This is the default in Linux. In other environments, it may be necessary to prefix string literals containing non-ascii characters with the u8 prefix to enforce UTF8 encoding. The u (16 bit), U (32 bit) and L (wide char) prefixes cannot be used on string literals, since wide strings are not supported. This will generate a compilation error. Examples:

```
"aa"                          (ok, ascii string)
u8"áá"                        (ok, explicit UTF8: 0xc3, 0xa1, 0xc3, 0xa1, 0)
u8"\U000000e1\U000000e1"      (same, using code points)
"áá"                          (ok in Linux)

U"áá"                         (compilation error, wide strings not supported)
```

Valid string literals may be suffixed with _lit or _ccl as necessary.

Character values are 32 bits wide (char32_t) and can hold any Unicode code point. Ascii (7-bit) literals like 'a' are automatically promoted to 32 bits.

Non-ascii character literals must be explicitly defined as char32_t by prepending them with the U prefix. Omitting the prefix is always incorrect. Depending on the context, it may generate compilation errors or warnings about using multichar constants, or just go undetected.

```
'a'                           (ok, ascii value implicitly promoted to char32_t)
U'á'                          (ok, explicit char32_t)
U'\xe1'                       (same, using code point)
U'á'_lit                      (ok, explicit char32_t)

'á'                           (incorrect, possible compiler warning)
'á'_lit                       (compilation error, operator""_lit(int) not defined)
```

Valid character literals accept the _lit suffix.

The Parser class

The Parser class encapsulates a grammar, an input matcher and a value stack:

Parser<T...>

The value stack holds variants supporting the set of types possibly returned by the grammar's rules (T…) and is implemented by default with an auto-resizing vector. If there is a possibility that the stack would resize and the copy constructor of a stack element is very costly, an alternative implementation using a map may be more efficient. To try this option, #define PEG_USE_MAP before including peg.h.

This is the interface of the Parser<T...> class:

```
// Constructor
Parser(Rule &r, std::istream &in = std::cin);

// Parsing methods
bool parse();
void accept();
void clear();
std::string text() const;

// Grammar check
#ifdef PEG_DEBUG
void check() const;
#endif

// Accessing the value stack
element_type &val(std::size_t idx);
template <typename U> U &val(std::size_t idx);
```

The constructor takes one or two arguments. The first argument is a reference to the grammar's start rule. The second (optional) argument is a reference to the input stream, by default standard input.

- parse() parses the starting rule.
- accept() executes the scheduled actions and discards matched input.
- clear() discards all input and all scheduled actions.
- text() returns a string with the contents of the most recently closed text capture.
- If PEG_DEBUG is defined, check() checks the grammar from the start rule.
- val(n) and val<T>(n) access the value stack.

Method val(n) returns a reference to slot n of the stack, of type element_type = std::variant<std::monostate, T...>. Rules return their values by assigning val(0). For example, in a Parser<int, std::string> a rule could execute any of the following:

```
val(0) = val(2);      // assign type and value of slot 2 to slot 0
val(0) = 33;          // assign slot 0 type int and value 33
val(0) = “hello”;     // assign slot 0 type string and value “hello”
```

When reading the value contained in a slot, it is necessary to use the second form of val(), with an explicit type qualification: val<T>(n) is equivalent to std::get<T>(val(n)). It returns a reference to the value of type T contained in slot n of the value stack, or throws std::bad_variant_access if the slot does not currently hold a value of type T.

Knowing that slot 3 contains a string and slot 2 contains an int, these are legal:

```
std::cout << val<std::string>(3) << std::endl;
val<int>(2) += 100;
```

Certain operations that involve reading values, like comparing two slots for equality, can be done without type qualification. For example,

```
if ( val(n1) == val(n2) )
        ...
```

checks that slots n1 and n2 hold values of the same type and equal magnitude.

A multi-typed value stack parser

This parser copies its input to its output, except when it finds embedded integers or sums of integers, which are replaced by their values. It is an artificial example, just for illustration.

```
#include <iostream>
#include <string>
#include "peg.h"

using namespace std;
using namespace peg;

class numsum : public Parser<int, string>
{
    Rule start, sum, other, number;

public:

    numsum(istream &in = cin) : Parser(start, in)
    {
        start   = sum                   do_( cout << val<int>(0); )
                | other                 do_( cout << val<string>(0); )
                ;
        sum     = number >> *(
                        '+' >> number   do_( val<int>(0) += val<int>(2); )
                    )
                ;
        number  = (+"0-9"_ccl)--        do_( val(0) = stoi(text()); );    // return int
        other   = Any()--               do_( val(0) = text(); );          // return string
    }
};

int main()
{
    numsum ns;
    while ( ns.parse() )
        ns.accept();
}
```

Input: aaa123+001+02bbb00044cc
Output: aaa126bbb44cc