

Pegplusplus, a C++ PEG library

What is PEG

PEG (Parsing Expression Grammar) is a formalism for specifying recursive descent parsers with unlimited backtracking. The theory can be consulted in Wikipedia or in Bryan Ford's original paper <http://bford.info/pub/lang/peg.pdf>.

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

Here we introduce a C++ library inspired by LEG. Using this library PEG grammars can be embedded in C++ code and parsed directly, without intermediate code generation.

The library

The peg library is a single header file (peg.h) and requires at least C++11. It is conceptually similar to boost::spirit, although much smaller, simpler and less efficient.

The syntax has been made as similar to LEG's as possible, within the limitations imposed by the available C++ operators, their associativities and precedence.

Primary expressions:

Lit(c) matches a single character c.

Lit(s) matches the string s literally.

Ccl(s) matches any character 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("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. Ccl("^") is a class that contains all characters and matches any character. It is equivalent to Any().

Any() matches any character. In the original PEG syntax this primary is represented by a dot (.). It fails only when the parser has reached the end of the input and cannot read more (end-of-file).

Do() succeeds without consuming input. It schedules an action to be executed after a successful parse if the branch of the grammar where the Do() is placed is matched. It takes an argument of type std::function<void()>, i.e. a free function, function object or lambda that takes no arguments and returns nothing.

Pred() is a semantic predicate. It takes an argument of type std::function<void(bool &)>, i.e. a free function, function object or lambda that receives a reference to a boolean variable and returns nothing. The function passed to Pred() is not scheduled for later, it is executed immediately when the predicate is parsed. Pred() succeeds without consuming input 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() succeeds by default if the function does not modify its argument.

Sequence and choice:

Sequences, built by concatenation in the original PEG syntax, are implemented with the binary operator >> in C++. Parsing e1 >> e2 starts by parsing e1. If e1 fails, the whole expression fails without parsing e2. If it succeeds, the result is determined by parsing e2.

The prioritized choice operator is |. Parsing e1 | e2 starts by parsing e1. If e1 succeeds, the whole expression succeeds without parsing e2. If it fails, the result is determined by parsing e2.

Repetition:

~exp means parse exp zero or one time (i.e. optionally). It always succeeds.

*exp means parse exp zero or more times. It always succeeds.

+exp means parse exp one or more times. It succeeds if exp is parsed at least once.

Syntactic predicates:

&exp ("and-predicate") succeeds if exp can be parsed at this point. It does not consume input and does not schedule any actions that exp might contain.

!exp ("not-predicate") succeeds if exp fails and vice-versa, without consuming input or scheduling actions. This kind of predicate is most frequently used. For example, !Any() means end-of-file.

Text capture:

--exp or exp-- capture the text consumed by exp. The captured text is available in Do() or Pred() code following the capture. Captures may be nested (which is useful for debugging).

As an example, consider this first version of a simple calculator implemented with the library:

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

#define PEG_USE_SHARED_PTR

#include "peg.h"

using namespace std;
using namespace peg;

int main(int argc, char *argv[])
{
    matcher m;
    vect<int> val;

    // Lexical rules

    Rule WS, SIGN, DIGIT, NUMBER, LPAR, RPAR, ADD, SUB, MUL, DIV;

    WS          = *Ccl(" \t\f\r\n");
    SIGN        = Ccl("+ -");
    DIGIT       = Ccl("0-9");
    NUMBER      = (~SIGN >> +DIGIT)-- >> WS >> Do([&]{ val.push(stoi(m.text())); });
    LPAR        = Lit('(') >> WS;
    RPAR        = Lit(')') >> WS;
    ADD         = Lit('+') >> WS;
    SUB         = Lit('-') >> WS;
    MUL         = Lit('*') >> WS;
    DIV         = Lit('/') >> WS;

    // Calculator

    Rule calc, expression, term, factor;

    calc        = WS >> expression >> Do([&]{ cout << val.top() << endl; val.pop(); });

    expression  = term >> *(
        ADD >> term >> Do([&]{ val.top(-1) += val.top(); val.pop(); })
        | SUB >> term >> Do([&]{ val.top(-1) -= val.top(); val.pop(); })
    );

    term        = factor >> *(
        MUL >> factor >> Do([&]{ val.top(-1) *= val.top(); val.pop(); })
        | DIV >> factor >> Do([&]{ val.top(-1) /= val.top(); val.pop(); })
    );

    factor      = NUMBER
        | LPAR >> expression >> RPAR;

    while ( calc.parse(m) )
        m.accept();
}
```

We will now explain and simplify this code.

The rules

Rules are instances of the Rule class. Defining a rule builds a syntax tree, dynamically allocating memory for the structures in the nodes of the tree. If a grammar is defined just once, it is probably not worth worrying about the allocated memory. In this case, each node of the tree points to its children using normal pointers. If the macro PEG_USE_SHARED_PTR is defined before including peg.h, these pointers are replaced by smart pointers of the type std::shared_ptr. These take care of releasing memory when the rules are no longer used. In this example we have defined the macro just to check that the smart pointer version works, although it is not necessary since the grammar is built only once, in the main function.

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

The Rule class has been designed in such a way that, when used as an expression, a Rule becomes a reference to itself. For this reason its use is very restricted: rules can only be default constructed and assigned from another rule, an expression or a type implicitly converting to an expression. No temporaries can be created, since they would

produce expressions with dangling references. For that reason the class has no constructors except the default, and the copy constructor has been deleted. Rule() is the only possible temporary, avoid it. It is useless, anyway.

The copy assignment operator is non-standard. One would expect that the statement `r = r` does nothing. However, if `r` is a rule, it actually makes `r` left-recursive, since `r` calls itself directly without reading input. Such a rule overflows the stack if parsed.

Left recursion can also arise from cyclic references. The calculator has the following 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. Otherwise the grammar would be left-recursive and the parser would enter an infinite loop. This is an inherent limitation of PEG grammars: there cannot be reference cycles that may close without reading at least one character from the input.

Once built, a grammar is used by calling method `parse()` of the starting rule (`calc` in our case). `Parse()` takes an argument of type `matcher` (see below). It returns `true` if the input matches the grammar, and `false` otherwise. Either way, when `parse()` returns the matcher holds the whole input that was read during parsing, both the part that matches the grammar and the one that does not. For example, let's assume that the parser is fed with this input:

(1 + 2) * 3 Hello

The call to `calc.parse(m)` returns `true`. The matcher has advanced its input pointer to the `H` in `Hello` (the first character that does not match the grammar). This is the limit of the part of the input consumed by the parser. The matcher also holds a vector of actions (scheduled by `Do()` during parsing). These actions must be executed in order to evaluate the expression. Calling `m.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 `calc.parse()` fails, since `Hello` does not match the grammar.

The matcher class

A PEG parser uses a grammar (set of rules) and an instance of the matcher class. This class handles input and provides services to the parser. Most of its methods are for internal library use, only a few are public and directly callable by the user:

- The constructor takes an optional argument of type `std::istream` with a default value of `std::cin`. Passing an adequate `std::istream` object to the matcher the parser can be made to get its input from anywhere.
- Method `accept()` must be called after a successful parse, in order to execute the scheduled actions and discard consumed input.
- Method `clear()` discards all input and all scheduled actions.
- Both scheduled actions and semantic predicates may call method `text()` to get a string with the contents of the most recently closed text capture.

The vect class

Class `vect` is a simple wrapper around `std::vector` with some handy features:

Operator `[]` automatically resizes the vector when it is indexed beyond its current limit.

These methods are specially useful for using the vector as a stack:

- Method `push()` adds an element at the end of the vector.
- Method `top()` allows addressing relative to the top of the stack. `Top()` or `top(0)` accesses the last element of the vector, `top(-1)` the previous one, etc.
- Method `pop()` removes elements at the end of the vector. `Pop()` or `pop(1)` removes the last element, `pop(2)` removes the last two elements, etc.

Methods `reserve()`, `size()`, `resize()` and `clear()` are the same as in `std::vector`.

Automatic handling of the value stack – classes `value_stack` and `value_map`

Manual handling of the value stack may be easy in a simple case like the calculator, but in more complicated cases it is better to handle the stack automatically. This is easily accomplished using special classes that handle stack indexing with the help of the matcher.

This version of the calculator handles the value stack automatically using the `value_stack` class. It also eliminates all explicit calls to the primaries except `Ccl()` by using an overload mechanism that will be explained later.

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

#define PEG_USE_SHARED_PTR

#include "peg.h"

using namespace std;
using namespace peg;

int main(int argc, char *argv[])
{
    matcher m;
    value_stack<int> val(m);

    // Lexical rules

    Rule WS, SIGN, DIGIT, NUMBER, LPAR, RPAR, ADD, SUB, MUL, DIV;

    WS          = *Ccl(" \t\f\r\n");
    SIGN        = Ccl("+ -");
    DIGIT       = Ccl("0-9");
    NUMBER      = (~SIGN >> +DIGIT)-- >> WS >> [&]{ val[0] = stoi(m.text()); };
    LPAR        = '(' >> WS;
    RPAR        = ')' >> WS;
    ADD         = '+' >> WS;
    SUB         = '-' >> WS;
    MUL         = '*' >> WS;
    DIV         = '/' >> WS;

    // Calculator

    Rule calc, expression, term, factor;

    calc        = WS >> expression >> [&]{ cout << val[1] << endl; };

    expression  = term >> *(
        ADD >> term >> [&]{ val[0] += val[2]; }
        | SUB >> term >> [&]{ val[0] -= val[2]; }
    );

    term        = factor >> *(
        MUL >> factor >> [&]{ val[0] *= val[2]; }
        | DIV >> factor >> [&]{ val[0] /= val[2]; }
    );

    factor      = NUMBER
        | LPAR >> expression >> RPAR >> [&]{ val[0] = val[1]; };

    while ( calc.parse(m) )
        m.accept();
}
```

A `value_stack` is constructed with a reference to the matcher (`m`). The constructor, operator `[]` and method `clear()` are its only public interface.

Expressions in a rule may return a value in a slot of the value stack, according to their position in the sequence. Rule actions may access these values using indices relative to the start of the rule. Rules return values by assigning `val[0]` (assuming `val` is the value stack). If a rule's actions do not assign `val[0]`, the rule returns the value of its first expression.

The index of each expression in a rule can be easily calculated by counting how many `>>`'s separate it from the start of the rule. For example, in rule `term`, 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 (`a1 ... a2` or `b1 ... b4`) matches when parsing.

This way of handling the value stack is similar to YACC's. Our `val[0]` is equivalent to YACC's `$$` and `$1`, `val[1]` is YACC's `$2`, etc.

Handling the stack automatically with `value_stack` is easier than doing it manually, but it can be less efficient because of the unused positions. `Value_stack` is implemented with `vect<T>`. The size of the vector grows automatically to include the biggest index used. When the vector grows, unused slots are filled with the default value `T()`. It is also possible that the vector needs to re-allocate memory and copy the values in the previous buffer to the new one. This can be mitigated or avoided by configuring the stack with a sufficiently large initial number of slots. `Value_stack`'s constructor accepts an optional second argument specifying the initial capacity, by default 128.

Class `value_map` is implemented using `std::map<int, T>` and has the same public interface as `value_stack`, except it does not accept the second argument in the constructor since it does not need to reserve memory. Maps accept any index without filling the unused slots, and they do not need to re-allocate memory or copy values from one buffer to another. `Value_map` is probably more efficient than `value_stack` if the stack is very sparse and either the default constructor or copy assignment of `T` are costly.

It is possible to use more than one value stack with the same matcher; they just run in parallel.

Elimination of primaries

The binary operators used to build the rules are overloaded so that they can accept in one of their arguments (but not in both) values of the types taken by the primaries, and automatically convert them to the respective primary. The assignment operator of class `Rule` is similarly overloaded.

<code>std::string s</code>	becomes <code>Lit(s)</code>
<code>const char *s</code>	becomes <code>Lit(std::string(s))</code>
<code>char c</code>	becomes <code>Lit(c)</code>
<code>std::function<void()> f</code>	becomes <code>Do(f)</code>
<code>std::function<void(bool &);> f</code>	becomes <code>Pred(f)</code>

In many cases these automatic conversions allow replacing explicit calls to the primaries by their arguments. In the second version of the calculator we have eliminated all the `Lit()` and `Do()` primaries. It is not possible to eliminate calls to `Ccl(s)`, since a standalone `s` becomes `Lit(s)`.

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

<code>"ab"_lit</code>	becomes <code>Lit(std::string("ab", 2))</code>
<code>'c'_lit</code>	becomes <code>Lit('c');</code>
<code>"a-z"_ccl</code>	becomes <code>Ccl(std::string("a-z", 3))</code>

Using these literals is mostly a matter of aesthetic preference. User-defined string literals allow null characters to be included in the basic literal, since the compiler passes the length of the literal to the literal operator. The operators are defined in inline namespace `peg::literals`.

Attached expressions

An expression can be attached to another using the syntax `e1 (e2)`. Expression `e2` is parsed immediately after successfully parsing `e1`. The behaviour is similar to `e1 >> e2`, except `e2` does not use any value stack slots. This is very useful for debugging when automatic handling of the value stack is used, since debugging actions can be inserted anywhere in the rules without changing the position of any expression in the stack. For example, we could modify the calculator to make it show everytime a `+` sign is matched in an arithmetic expression:

```
expression    = term >> *(
                    ADD                ([&]{ cout << "add\n"; })
                    >> term            >> [&]{ val[0] += val[2]; }
                    | SUB >> term      >> [&]{ val[0] -= val[2]; }
                    );
```

We have attached an action to expression `ADD`, without disturbing the placement of the second term in position 2 of the stack. If we now feed the calculator with the arithmetic expression `((1+2)*3+4)*5+6`, we get the following output:

```
add
add
add
71
```

Attached expressions may be arbitrarily complex. Operator `()` has been overloaded for the same types as the binary operators, saving an explicit call to `Do()` in the previous example.

Attaching actions saves unused slots in value stacks (although this is irrelevant if `value_map` is used instead of `value_stack`). The previous rule can be re-written with all actions attached without altering its behavior:

```
expression    = term >> *(
                    ADD                ([&]{ cout << "add\n"; })
                    >> term            ([&]{ val[0] += val[2]; })
                    | SUB >> term      ([&]{ val[0] -= val[2]; })
                    );
```

Action and semantic predicate macros

`Peg.h` defines three macros for using capture-by-reference lambdas in actions and semantic predicates with a very concise notation.

This is the macro for an action:

```
#define _(code...)    (peg::Do([&]{ code }))
```

The argument of the macro is the body of a function of type `void()`. Re-writing the previous example using this macro makes the code more concise:

```
expression    = term >> *(
                    ADD                _( cout << "add\n"; )
                    >> term            _( val[0] += val[2]; )
                    | SUB >> term      _( val[0] -= val[2]; )
                    );
```

The other two macros are for semantic predicates:

```
#define _if(cond...)  (peg::Pred([&](bool &__r){ __r = (cond); })))
#define _pr(code...)  (peg::Pred([&](bool &__r){ __r = [&]()->bool{ code }(); })))
```

The argument of `_if(...)` is any `bool` C++ expression (which may include commas). The predicate succeeds if the expression evaluates to true, and fails otherwise.

The argument of `_pr(...)` is the body of a function of type `bool()`. The predicate succeeds if this function returns true, and fails otherwise. This macro accepts full C++ code and requires an explicit return statement.

As an example, the following rules are equivalent and always fail:

```
Rule fail1, fail2;
fail1 = _if( false );
fail2 = _pr( return false; );
```

The three macros expand to parenthesized expressions and may be placed anywhere in a rule. If they are placed immediately following another expression, they attach to it, because the parentheses are interpreted as a function call. Otherwise, the parentheses are redundant and have no effect.

Final version of the calculator

The final version of the calculator replaces calls to `Ccl()` by user-defined literals and attaches all actions using the `_(...)` macro. This is the closest we can get to LEG's syntax.

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

#define PEG_USE_SHARED_PTR

#include "peg.h"

using namespace std;
using namespace peg;

int main(int argc, char *argv[])
{
    matcher m;
    value_stack<int> val(m);

    // Lexical rules

    Rule WS, SIGN, DIGIT, NUMBER, LPAR, RPAR, ADD, SUB, MUL, DIV;

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

    // Calculator

    Rule calc, expression, term, factor;

    calc      = WS >> expression          _( cout << val[1] << endl; );

    expression = term >> *(
        ADD >> term          _( val[0] += val[2]; )
        | SUB >> term        _( val[0] -= val[2]; )
    );

    term       = factor >> *(
        MUL >> factor        _( val[0] *= val[2]; )
        | DIV >> factor      _( val[0] /= val[2]; )
    );

    factor     = NUMBER
        | LPAR >> expression >> RPAR  _( val[0] = val[1]; );

    while ( calc.parse(m) )
        m.accept();
}
```

Checking a grammar

If the macro `PEG_DEBUG` is defined before including `peg.h`, the rules are compiled with a method that allows checking a grammar. This method must be applied to the starting rule, in our example:

```
calc.check();
```

`Check()` visits all grammar paths originating in the starting rule and throws an exception of type `Rule::bad_rule` whenever 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.

Additionally, each rule may be assigned a name (a `const char *`). For example,

```
calc.name = "calc";
```

In debug mode `peg.h` defines the macro `peg_debug()` as follows:

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

So in order to assign a debugging name to `calc`, this is the easiest way:

```
peg_debug(calc);
```

Rules that have names assigned will print debugging messages to standard error when visited by `check()`. This is useful for understanding the paths that `check()` follows through the grammar and determining the reason for eventual failures.

For example, we define `PEG_DEBUG` in the calculator. After the grammar is built and before parsing 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);

calc.check();
```

When the program runs, 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 make a small change to the grammar: in the last rule (factor) we make `LPAR` optional (`~LPAR`). This change makes the grammar left-recursive, because now the cycle `expression` \rightarrow `term` \rightarrow `factor` \rightarrow `expression` can 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` (white space) 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 can be gradually added to the debugger, until the cause of the problem is clear.

Unicode support

The library 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 force 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 produce a compilation error.

<code>"aa"</code>	(ok, ascii string)
<code>u8"áá"</code>	(ok, explicit UTF8: 0xc3, 0xa1, 0xc3, 0xa1, 0)
<code>u8"\U000000e1\U000000e1"</code>	(same, using code points)
<code>"áá"</code>	(ok in Linux)
<code>U"áá"</code>	(compilation error, wide strings not supported)

Valid string literals can 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.

<code>'a'</code>	(ok, ascii value implicitly promoted to <code>char32_t</code>)
<code>U'á'</code>	(ok, explicit <code>char32_t</code>)
<code>U'\xe1'</code>	(same, using code point)
<code>U'á'_lit</code>	(ok, explicit <code>char32_t</code>)
<code>'á'</code>	(incorrect, possible compiler warning)
<code>'á'_lit</code>	(compilation error, operator <code>"'_lit(int)"</code> not defined)

Valid character literals accept the `_lit` suffix.