# Pegpp, a C++ PEG library

## Table of Contents

## The PEG library

PEG (Parsing Expression Grammar) is a formalism for specifying recursive descent parsers with unlimited backtracking. The theory of PEGs can be found 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 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, but it is not a compiler. Using this library PEG grammars can be embedded in C++ code and parsed without intermediate code generation.

The basic library is a single header file (peg.h) and requires C++11. A second optional header (pegparser.h) includes peg.h and adds parser classes. This header requires C++17, since it uses variants.

## PEG grammars

PEG grammars are written using a syntax similar to Extended Backus-Naur Form (EBNF).

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

```
rule 1 = expression 1;
rule 2 = expression 2;
...
rule n = expression n;
```

Once assigned, a rule allows referring to an expression by name.

These are the most basic expressions:

- Primaries
- Rules
- Embedded actions
- Semantic predicates

*Primaries* match some basic input patterns. If they parse successfully they consume the matched input. Otherwise, no input is consumed. Input read during an unsuccessful parse is returned to the input stream.

> 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().

*A rule* is a reference to the expression assigned to it.

*Embedded actions* do not consume input and always succeed. They schedule actions to be executed after a successful parse, if the branch that contains them is part of the match. 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 are executed during parsing and may be used as a guard in order to make parsing 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 is an expression, other expressions may be obtained by applying some unary operators:

- A text-capturing expression: --e e--
- A repetitive expression: ~e *e +e
- A syntactic predicate: &e !e

If e1 and e2 are expressions, other expressions may be obtained by composing them with some binary operators:

- A sequence: e1 >> e2
- An attachment: e1(e2)
- An ordered choice: e1 | e2

*Text capture*

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

*Repetitions*

~e means parse e 0 or 1 times. It tries to parse e once and succeeds.
*e means parse e 0 or more times. It tries to parse e as many times as possible and succeeds.
+e means parse e 1 or more times. It tries to parse e as many times as possible and succeeds if e is successfully parsed at least once.

*Syntactic predicates*

These are lookahead mechanisms that check if some expression e can be parsed at the current point. If e parses successfully, input eventually consumed by e is returned to the input stream and actions eventually scheduled by e are cancelled.

&e ("and-predicate") succeeds if e can be parsed at this point.
!e ("not-predicate") succeeds if parsing e fails and vice-versa. This kind of predicate is most frequently used as a stop condition. !Any() means end of file.

*Sequence*

Sequences, built by concatenation in the original LEG 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 e1 succeeds, the result is obtained by parsing e2.

*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*

The prioritized choice operator is |. Parsing e1 | e2 starts by parsing e1. If e1 succeeds, the whole 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.

These are the available operators in decreasing order of priority:

- e-- e1(e2)
- --e &e !e ~e *e +e
- e1 >> e2
- e1 | e2

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

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.

## Example: a simple calculator

Consider this first version of a simple integer calculator:

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

#define PEG_USE_SHARED_PTR

#include "peg.h"

using namespace std;
using namespace peg;

int main()
{
	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();
}
```

This code will be explained and simplified below.

## 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. 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 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, which are very frequent.

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. Rule() is the only possible and useless temporary, please avoid it.

A grammar is a set of rules which must be defined as global, local or instance 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 r calls itself directly without reading input. Such a rule overflows the stack if parsed. 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 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 and the parser would enter an infinite loop. 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 a grammar

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. When parse() returns the matcher holds in an internal buffer all input 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 first call to calc.parse(m) returns true. Now 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, and the input pointer does not advance.

## 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. 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 (standard input). 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 automatic resizing and stack operations.

- Operator [] automatically resizes the vector when it is indexed beyond its current limit.
- Method push() adds an element at the end of the vector when used as a stack.
- 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 top of the stack. Pop() or pop(1) removes the last element of the vector, pop(2) removes the last two, etc.

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

## Calculator revisited: automatic value stack

Manual handling of the value stack may be easy in a simple case like the calculator, but in more complicated cases it may be better to handle the stack automatically. This is easily accomplished using special classes that handle stack indexing relative to the start of each rule, as YACC does with its $$, and $n pseudo-variables. Classes value_stack and value_map have been designed for this purpose, and handle relative stack indexing with the help of the matcher. As an example, 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 operator 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()
{
        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), which handles indexing relative to the start of each rule. 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 (and semantic predicates) may return values by assigning val[0] (assuming val is the value stack). If it does not assign val[0], a 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 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 $$ and $1, val[1] is YACC's $2, etc.

Although handling the stack automatically with value_stack is easier than doing it manually, it may be less efficient because of the unused positions. Value_stack is implemented around 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<T> is implemented using std::map<int, T>. It 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 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 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.

## Eliminating primaries

The operators used to build the rules are overloaded so that they can accept 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.

| | |
|---|---|
| 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 primaries by their arguments. In the second version of the calculator we have eliminated all calls to Lit() and Do(). It is not possible to eliminate calls to Ccl(s), since a standalone string s converts to Lit(s), not to Ccl(s).

In the case of the binary operators, one of the arguments has to be an explicit expression; the other may convert automatically.

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)) |

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.

## Attaching expressions

An expression can be attached to another using the syntax e1(e2). Expression e2 is parsed immediately after successfully parsing e1. The behavior is similar to e1 >> e2, except e2 does not occupy any value stack slots. This is very useful when automatic handling of the value stack is used, since debugging actions can be attached 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 every time 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 affecting the placement of the second term in position 2 of the stack. It is not necessary to call Do() explicitly, since operator() is overloaded. 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
```

Actions and semantic predicates may always be attached. This saves unused slots in value stacks (although this is irrelevant if value_map is used instead of value_stack) and the resulting notation is more compact. The previous rule can be re-written will 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 some macros for using capture-all-by-reference lambdas in embedded actions and semantic predicates with a very compact notation:

```
#define   _(...)     (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 _(...) macro defines a scheduled action. The argument is the body of a void() function. Re-writing the previous example using this macro:

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

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.

For example, the following rules are equivalent and always fail:

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

These 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.

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;

int main()
{
    matcher m;
    value_stack<string> val(m);
    Rule start, pal, chr;

    start   = pal--                                   _( cout << m.text() << endl; )
            ;
    pal     = chr >> pal >> chr                     if_( val[0] == val[2] )
            | chr >> chr                            if_( val[0] == val[1] )
            | chr
            ;
    chr     = Any()--                               pa_( val[0] = m.text(); )
            ;

    while ( start.parse(m) )
        m.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 _(...) 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.

## 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.

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

#define PEG_USE_SHARED_PTR

#include "peg.h"

using namespace std;
using namespace peg;

int main()
{
        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 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.

Additionally, in debug mode 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 introduce a small change in the grammar: in the last rule (factor) we make LPAR optional (~LPAR). This change makes the grammar left-recursive, because now the cycle expression → term → factor → expression may 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 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 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. 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 classes

In order to use the parser classes, pegparser.h should be included instead of peg.h. The parser classes are:

| | |
|---|---|
| BasicParser | encapsulates a matcher and a starting rule, no value stack. |
| Parser<T...> | extends BasicParser adding a value stack with variant type elements. |
| Parser<T> | extends BasicParser adding a value stack with elements of type T. |
| Parser<> | same as BasicParser, no value stack. Provided for uniformity. |

Value stacks are implemented by default with the value_stack class. If value_map is preferred instead, #define PEGPARSER_USE_MAP before including pegparser.h.

This is the interface of the BasicParser class:

```
// Constructor
BasicParser(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
```

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

Method parse() parses the starting rule. Methods accept(), clear() and text() call the respective methods of the matcher. If compiled with PEG_DEBUG, check() checks the grammar from the starting rule.

The Parser<T...> class adds the following interface for accessing the value stack:

```
element_type &val(std::size_t idx);
template <typename U> U &val(std::size_t idx);
```

where element_type is the type of the elements of the value stack, defined as std::variant<std::monostate, T...>. The inclusion of std::monostate guarantees that the stack is default constructible.

Method val(n) returns a reference to the variant in slot n of the stack. Rules should 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 accessing the value contained in a slot, it is necessary to use the second form of val(), with an explicit type qualification: val<T>(n) returns a reference to the value of type T contained in the variant 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.

For example, 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;
```

The Parser<T> class has a value stack with elements of a single type T. It has the same interface as Parser<T...>, but now element_type is T, not a variant.

The val<T>(n) method is redundant and returns the same reference as val(n). However, its use is encouraged for compatibility with the variant case. Incorrect invocations, like calling val<std::string>(n) on a Parser<int> do not throw, since no variants are used, but generate compilation errors.

The Parser<> class has no value stack and is just a BasicParser. When writing a parser it is handy to derive it from Parser<> and populate the initially empty types list as necessary when the grammar is developed.

## An example using the Parser class

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 to illustrate the use of a variant value stack parser.

```
#include <iostream>
#include <string>
#include "pegparser.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                   _( cout << val<int>(0); )
                | other                 _( cout << val<string>(0); )
                ;
        sum     = number >> *(
                        '+' >> number   _( val<int>(0) += val<int>(2); )
                    )
                ;
        number  = (+"0-9"_ccl)--        _( val(0) = stoi(text()); );    // return int
        other   = Any()--               _( val(0) = text(); );          // return string
    }
};

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

Input:   aaa123+001+02bbb00044cc
Output:  aaa126bbb44cc