PGPP, a C++ PEG library

Table of Contents

Introduction

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.

Although they might look superficially similar, PEGs are different from CFGs (context-free grammars). CFGs are declarative and frequently ambiguous, while PEGs are imperative and unambigous.

YACC is a popular CFG compiler that takes a CFG written in BNF (Backus-Naur form) with embedded actions and generates a parser in C.

LEG, by Ian Piumarta, is a similar tool that takes a PEG written in a formalism similar to EBNF (extended Backus-Naur form) with embedded actions and generates a parser in C.

PEGPP is inspired by LEG, but it is a library, not a compiler. PEGPP grammars are C++ code and are parsed without intermediate compilation. PEGPP uses overloaded C++ operators to emulate LEG’s formalism. The resulting code resembles LEG closely enough that LEG grammars can be translated to PEGPP almost mechanically. 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: it moves the input pointer backwards to where it was before the unsuccessful attempt and cancels scheduled actions.

Basic parsing expressions

These are the basic parsing expressions:

- Lexical primitives
- Rules
- Embedded actions
- Semantic predicates

Lexical primitives

Lexical primitives match some simple input patterns. They are generated by calling some functions. When parsing, they read input and try to match one character or a sequence of characters. If they parse successfully they advance the input pointer over the matched input. Otherwise they leave the input pointer untouched.

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 of the characters in string s. Ccl(“ \t\r\n\f”) may be a definition of white space. When placed between two other characters, – defines a range. Ccl(“0-9”) is the class of decimal digits. If s begins with ^, the class is complemented or negated and contains the characters *not* included in the rest of s. Thus Ccl(“^”) matches any character and is equivalent to Any().

User-defined literals

Lit(s), Lit(c) and Ccl(s) with literal arguments may 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. They are defined in inline namespace peg::literals.

Rules

Rules represent the expressions assigned to them. When a rule is parsed it parses its expression, setting the origin of indices into the value stack relative to the first term in the expression. More on the value stack later.

Embedded actions

Embedded actions succeed without consuming input. They schedule actions together with their respective contexts. If parsing succeeds, the actions may be executed in the same order that they were scheduled and with the same contexts. Actions scheduled during an unsuccessful parse are canceled when the parser backtracks.

> *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

Semantic predicates do not consume input. They execute actions during parsing, and fail or succeed depending on semantic conditions verified by these actions.

> *Pred(f)* takes an argument of type std::function<void(bool &)>, i.e. a function or function object that receives a reference to a boolean variable and returns nothing. When Pred(f) is parsed, a boolean variable v is set to true and then f(v) is called. When f(v) returns Pred(f) succeeds if v is still true, or fails otherwise. Pred(f) succeeds by default if f ignores its argument.

Compound expressions

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)
• Ordered choice	e1 | e2

Text capture

The expressions --e and e-- parse e, and if this succeeds they capture the text matched by e. Otherwise, they fail without capturing. The last captured text may be accessed from Do() or Pred() actions following the capture. Captures may be nested.

Repetition

These expressions try to parse e some number of times.

~e Optional e, tries to parse e once and succeeds. Same as e[{0, 1}].

*e Zero or more times e, tries to parse e as many times as possible and succeeds. Same as e[{0, 0}].

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

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. The parser keeps trying to parse e until a maximum of n2 times. If n2 is 0 there is no upper limit and the parser keeps trying to parse e as many times as possible.

e[{n}] or e[n]
Shorthands for e[{n, n}]. If n is 0, same as *e. Otherwise it tries to parse e exactly n times.

Syntactic predicates

These 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 buffer and actions eventually scheduled by e are canceled.

> &e ("and-predicate") succeeds if e can be parsed at the current point. Note that this syntax overloads operator &, 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.

> !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++ syntax requires an operator to build a sequence, 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 tried.

Attachment

An attachment e1 (e2) parses like the sequence e1 >> e2, but e2 does not occupy any slots in the value stack. More on the value stack later.

Ordered 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.

Operator priorities

These are 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.

A simple example

This simple parser is a direct translation of a sample from the LEG. It reads from standard input (std:cin) and copies all input to the output, replacing occurrences of the string "username" by the name of the currently logged in user.

This is the LEG original:

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

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

%%

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

And this is the PEGPP version:

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

#include "peg.h"

using namespace std;
using namespace peg;

int main()
{
    matcher m;

    Rule start;

    start   = "username"_lit        do_( cout << getlogin(); )
            | Any()--               do_( cout << m.text(); )
            ;

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

The two versions look very similar. This was a design goal of PEGPP.

The matcher reads input, recognizes basic lexical patterns and keeps the parser's state, including matched and unmatched input, text capture and scheduled actions. Matchers can be initialized with a reference to the input stream to read from, which defaults to standard input (std::cin).

This grammar has only one rule (start) with a choice of two branches.

The first branch matches the string "username" literally.

The second branch matches any single character and captures it by means of the -- operator. It will fail if Any() fails, which only happens at end of file.

Embedded actions are placed in the grammar using the do_(...) construct, which replaces the pair of braces { ... } used by both YACC and LEG. This construct is a macro that 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.

The grammar is parsed by calling method parse() of the start rule, which returns true if the input matches. In this grammar, parse() succeeds after reading and matching either 8 bytes ("username") or a single byte (Any()).

Upon a successful parse(), calling method accept() of the matcher executes the scheduled actions, erases them and discards the matched part of the input. Input now starts at the character following the last one matched. For more details, see "Parsing a grammar".

A calculator

This is a simple integer calculator that supports the four basic operations and grouping with parentheses. This example demonstrates a more complex set of parsing expressions and uses a value stack.

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

#include "peg.h"

using namespace std;
using namespace peg;

int main()
{
    matcher m;
    value_stack<int> v(m);

    Rule WS, SIGN, DIGIT, NUMBER, LPAR, RPAR, ADD, SUB, MUL, DIV;
    Rule calc, expression, term, factor;

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

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

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

The following rules illustrate several important points:

```
WS         = *" \t\f\r\n"_ccl;
SIGN       = "+-"_ccl;
DIGIT      = "0-9"_ccl;
NUMBER     = (~SIGN >> +DIGIT)-- >> WS    do_( v[0]= stoi(text()); );
```

WS is "white space", defined as zero or more occurrences (*) of any character from the character class " \t\f\r\n".

In the NUMBER rule the expression (~SIGN >> +DIGIT) describes an integer number as "an optional (~) sign followed by one or more (+) digits". The text matched by this expression is captured by the -- operator. The rule consumes any white space that may follow the number. Finally, the scheduled action retrieves the captured text from the matcher, converts it to an integer and returns this integer from the rule by assigning it to v[0], the first slot in this rule's view of the value stack.

In a YACC parser this action would have been written { $$ = atoi(yytext()); }. In PEGPP slot 0 of the value stack is equivalent to YACC's $$ and $1, slot 1 is $2, and so on. For details on the value stack, see "The value stack" and "The parser class" below.

The value stack is initialized with a reference to the matcher, which keeps track of the relative offset of each rule's view into the underlying vector.

Implicit basic expressions

Whenever possible, operators have been overloaded so that they can accept strings, characters and functions and automatically promote them to basic parsing expressions.

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 the calculator we have eliminated all explicit calls to Lit(). It is not possible to eliminate calls to Ccl(), since a standalone string s converts to Lit(s).

Unary operators can only be applied to expressions.

Binary operators >> and | require that at least one of the arguments be an expression; the other may convert implicitly.

For example, in the calculator, the expression

```
'+' >> WS
```

is implicitly converted to

```
Lit('+') >> WS
```

since WS is an expression.

In the case of e1 (e2), e1 must be an expression and e2 may convert implicitly. Most of the time e2 is a function or functional object f and promotes to Do(f) or Pred(f).

The rules

Rules are instances of the Rule class. A rule is assigned either a basic expression or the root of an expression tree built by composing basic expressions with operators and parentheses. Memory is dynamically allocated for the structures in the nodes of the tree. The tree is built using smart pointers of the type std::shared_ptr, so that this memory is released when the rules go out of scope and are destroyed.

Rules may be assigned to in any arbitrary order. The expression tree assigned to a rule may use the same rule or other rules as basic expressions, whether they have already been initialized or not. This is necessary to support grammars with cyclic references. A *rule expression* is a proxy that holds a reference to the rule. This is a plain C++ reference, not a smart pointer, otherwise cyclic grammars would leak memory.

This behavior imposes some limitations on the Rule class. Rules have to be defined as uninitialized variables and assigned an expression. They cannot be directly initialized or copied. The statement r = r is normally supposed to do nothing. But if r is a rule the two r's are different things: the r on the left is a rule, and the r on the right is a rule expression. This statement makes r left-recursive.

Left recursion

A PEG is left recursive if it contains at least one rule that may try to parse itself without consuming any input. Parsing such a rule goes into infinite recursion and overflows the stack.

The statement r = r is a case of direct left recursion: 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, if the grammar has cyclic references. The calculator has the following cycle:

expression → term → factor → expression

The first two references (expression → term and term → factor) are direct: expression parses term and term parses factor before reading any input. The third reference (factor → expression) closes the cycle, but this time factor parses expression only after parsing a left parenthesis token (LPAR), which consumes at least one ‘(‘ character. 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 are forbidden.

Parsing a grammar

Once built, a grammar is parsed by calling method parse() of the start rule. It returns true if the grammar is parsed successfully, and false otherwise. When parse() suceeds, the matcher holds in an internal buffer all input read so far, and an input pointer pointing to the first character after the last one matched. For example, let's assume that the calculator of the previous example 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 are stored together with a context that includes the position and length of the last captured text in the input buffer and the current offset of indices into the value stack vector, so that the text() method returns the correct string and indexing the value stack works properly.

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.

Usually parse() is called in a loop until it fails, and accept() is called every time parse() suceeds:

```
while ( start.parse(m) )
    m.accept();
```

Calling accept() may be arbitrarily delayed; in this case matched input and scheduled actions just queue up in the matcher until accept() is called.

The value stack

Rule actions (and semantic predicates) may return a value from their rule by assigning slot 0 of the value stack. The value stack is based on a self-resizing vector and is indexed like a vector, but the indices are relative to a certain offset that is set when each rule starts parsing.

When a rule r1 is used as a basic expression in a rule r2, the value assigned to slot 0 by r1 appears to r2 in slot i, where i is the distance of r1 to the root of r2. This distance is measured by counting the number of >>’s necessary to reach r1 from the root of r2.

In rule term of the calculator, for example, counting the >>’s tells that the first factor is in slot 0 and the second one is in slot 2:

```
term           = factor >> *(
                         MUL >> factor       do_( v[0] *= v[2]; )
                       | DIV >> factor       do_( v[0] /= v[2]; )
                       )
               ;
```

If an expression has alternatives, it uses as many stack slots as the longest one. For example:

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

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

The value_stack class

The value_stack<T> class is based on a auto-resizing vector that holds elements of type T. This is the public interface:

```
value_stack(const matcher &m, std::size_t capacity = 128);
T &operator[](std::size_t idx);
```

The constructor takes a reference to the matcher, that keeps the indexing offset, and an optional initial capacity that defaults to 128. If parsing calls are too nested and the value stack fills up, it reallocates memory automatically. This involves copying and destroying all current elements of the vector. Depending on T, this might be a costly operation. To avoid reallocations, set a bigger initial capacity.

If the parser needs to pass values of different types on the stack, it is possible to use more than one value stack, one for each type. These stacks are initialized with the same matcher and run in parallel. However, a variant value stack is usually a better solution.

Variant value stacks - the var class

Sometimes rules need to return values of different types on the value stack. In YACC this is done with unions (the %UNION directive). The same approach can be used in C++, but std::variant is a more robust alternative.

The var<T...> class is a trivial extension of std::variant<std::monostate, T...>. Including std::monostate as a first default type ensures that a vector of vars is default constructible. A var behaves like a standard variant and adds a method val<U>() for type-qualified access. For example:

```
var<int, double, char *> v;
v = 2.23;
double &d = std::get<double>(v);
double &e = v.val<double>();          // same
```

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 to a grammar does not affect the positions of the rule's expressions in the value stack. This is very useful when it is necessary to add new actions or predicates to an existing grammar, either to add functionality or for debugging, without changing references to the value stack in previously existing actions and predicates.

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

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

Note how a void(bool &) lambda is implicitly converted to Pred() when assigned to a rule, and that the predicate succeeds if the lambda 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;

int main()
{
    matcher m;
    value_stack<string> v(m);

    Rule start, pal, chr;

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

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

The pal rule defines a palindrome as a symmetric string, including strings of lengh 1. For lengths > 1 the symmetry is enforced by the 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 the text capture of each recognized palindrome, followed by a newline. Recognized palindromes are not always the longest possible, especially if the input contains sequences of repeated characters.

The Parser class

The Parser<T...> class encapsulates a grammar, an input matcher and a variant value stack. This is the public interface:

```
// Constructor
Parser(Rule &start, std::istream &in = std::cin);
Parser(Rule &start, std::size_t capacity, 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
var<T...> &val(std::size_t idx);
template <typename U> U &val(std::size_t idx);
```

The constructor takes one mandatory and two optional arguments: a reference to the grammar's start rule, the initial capacity of the value stack (by default 128) and a reference to the input stream (by default standard input). Usually the grammar's rules are instance variables of the parser and the grammar is built in the parser's constructor.

- parse() parses the start 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 var<T...>. Rules return their values by assigning to val(0). For example, in a Parser<int, std::string> a rule could execute any of the following assignments:

```
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<U>(n) is equivalent to std::get<U>(val(n)). It returns a reference to the value of type U 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 U.

Knowing that slot 3 of the value stack contains a string and slot 0 contains an int, these are correct:

```
std::cout << val<std::string>(3) << std::endl;
val<int>(0) += 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 the same type and value.

The Parser class does not add any new functionality to its components: matcher, rules and value stack. It just provides an OOP approach for writing a parser and a handy notation for accessing the value stack. If a parser does not use a value stack, it may be declared to extend Parser<> with an empty type list.

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 numeric 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

Checking a grammar

If the macro PEG_DEBUG is defined before including peg.h, the rules are compiled with a method check() that visits all grammar paths originating in the starting rule and throws an exception of type Rule::bad_rule if it detects either 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, since it writes some accounting information in the rules. It would be pointless anyway. Once the grammar is checked, both the PEG_DEBUG 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.set_name(“calc”);
```

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

```
#define peg_debug(rule) rule.set_name(#rule)
```

Rules that have debugging names assigned will print messages on standard error when visited by check(). For example, we define PEG_DEBUG in the calculator and, after the grammar is built, we enable some rules for debugging and check the grammar:

```
        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();
```

Check() prints the following 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 reading 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

PGPP 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, UTF8 is default)
        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.