PEGPP, a C++ PEG library

Table of Contents

Introduction....2
PEG grammars....2
Basic parsing expressions....2
    Lexical primitives....2
    User-defined literals....2
    Rules....3
    Scheduled actions....3
    Semantic predicates....3
Compound expressions....3
    Text capture....3
    Repetition....3
    Syntactic predicates....3
    Sequence....4
    Attachment....4
    Ordered choice....4
    Operator priorities....4
A simple example....4
A calculator....5
Implicit basic expressions....6
The rules....6
Left recursion....7
Parsing a grammar....7
Using the value stack....7
Scheduled action and semantic predicate macros....8
The Parser class....9
    An example with a variant value stack....10
Error reporting....10
    An example using labeled rules....11
Memoization....12
    An example of memoization....13
Checking a grammar....14
Unicode support....15

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 may 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 parser generator 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. See https://www.piumarta.com/software/peg.

PEGPP is inspired by LEG, but it is a library, not a compiler. PEGPP grammars are C++ code and do not need an intermediate compilation. Overloaded C++ operators 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:

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

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
- User-defined literals
- Rules
- Scheduled actions
- Semantic predicates

Lexical primitives

Lexical primitives match some simple input patterns. 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 at its original position. These are the lexical primitives:

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

Lit(c) matches the single character c.

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

Ccl(s) matches any character contained in string s. Ccl(" \t\r\n\f") is a common 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 matches any of the characters *not* included in the rest of s. Thus Ccl("^") matches any character and is equivalent to Any().

User-defined literals

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

User-defined string literals allow embedded null characters, since the size of the literal is passed to the string constructor. 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.

Scheduled actions

Scheduled actions succeed without consuming input. They store actions and their respective contexts to be executed later. If parsing succeeds, scheduled actions are executed in the order that they were scheduled and with their contexts. Actions scheduled during an unsuccessful parse are canceled when the parser backtracks.

> Do(f) schedules an action f. The type of f should be convertible to std::function<void()>, i.e. f must be 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 checked by these actions.

> Pred(f) takes an argument of a type convertible to std::function<void(bool &)>, i.e. f must be a function, function object or lambda 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. Note that Pred(f) succeeds by default if f just ignores its argument.

Compound expressions

If e, e1 and e2 are expressions, other expressions may be obtained by composing them with 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 by calling method text() of the parser. 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. During lookahead no actions are scheduled.

> &e ("and-predicate") succeeds if e can be parsed at the current point. Note that this syntax overloads unary operator &, so that &e means "e can be parsed at this point" and not "the address of e".
>
> !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 PEG syntax sequences are built by concatenation. C++ 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. It is evaluated in short-circuit, i.e. 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 is what 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 LEG. It reads 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>
    #include <stdio.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()
{
    Rule start;

    Parser p(start);

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

    while ( p.parse() )
        p.accept();
}
```

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

The parser is initialized with the start rule and reads input, recognizes basic lexical patterns and keeps an internal state, including matched and unmatched input, text captures and scheduled actions. Parsers can also 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 alternative 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.

Scheduled actions are placed in the grammar using a do_(...) block, which replaces the pair of braces { ... } used by both YACC and LEG. This 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(), which returns true if the input matches. In this grammar, parse() succeeds after reading and matching either 8 characters (“username”) or a single character (Any()).

Upon a successful parse(), calling method accept() 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 <iostream>
#include <string>

#include "peg.h"

using namespace std;
using namespace peg;

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

public:

    intcalc(istream &in = cin ) : Parser(calc, in)
    {
        // Lexical rules
        WS          = *" \t\f\r\n"_ccl;
        SIGN        = "+-"_ccl;
        DIGIT       = "0-9"_ccl;
        NUMBER      = (~SIGN >> +DIGIT)-- >> WS     do_( val(0) = stoi(text()); );
        LPAR        = '(' >> WS;
        RPAR        = ')' >> WS;
        ADD         = '+' >> WS;
        SUB         = '-' >> WS;
        MUL         = '*' >> WS;
        DIV         = '/' >> WS;

        // Calculator
        calc        = WS >> expression              do_( cout << val(1) << endl; )
                    ;

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

        term        = factor >> *(
                            MUL >> factor           do_( val(0) *= val(2); )
                          | DIV >> factor           do_( val(0) /= val(2); )
                          )
                    ;

        factor      = NUMBER
                    | LPAR >> expression >> RPAR    do_( val(0) = val(1); )
                    ;
    }
};


int main()
{
    intcalc calc;

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

We have derived a parser class (intcalc) from Parser<int>. Parser<T>, where T is any type except void, has a value stack whose elements are of type T and val() methods por accessing the value stack. Parser, Parser<> or Parser<void> do not have a value stack and lack the val() methods.

Here rules are declared as instance variables of the parser, and the grammar is built in its constructor. This way actions and predicates can call inherited Parser methods like text() and val() directly. The parser reads characters from an input stream, by default std::cin (standard input).

The following rules illustrate several important points:

```
WS          = *" \t\f\r\n"_ccl;
SIGN        = "+-"_ccl;
DIGIT       = "0-9"_ccl;
NUMBER      = (~SIGN >> +DIGIT)-- >> WS     do_( val(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, but this is not part of the captured text. Finally, the scheduled action retrieves the captured text, converts it to an integer and returns this integer from the rule by assigning it to val(0), the first slot of this rule's view of the value stack.

In PEGPP val(0) is equivalent to YACC's $$ and $1, val(1) is $2, and so on. For details on the value stack, see "The parser class" below.

Implicit basic expressions

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

std::string s	promotes to Lit(s)
char *s	promotes to Lit(std::string(s))
char c	promotes to Lit(c)
std::function<void()> f	promotes to Do(f)
std::function<void(bool &)> f	promotes 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 side of the operation is already an expression; the other side may convert implicitly. For example, the expression

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

is implicitly converted to

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

since WS is already 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 either Do(f) or Pred(f) depending on its signature.

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 can be assigned a const char * label when defined. This is used for error reporting. See "Error reporting" below. They can also be assigned a boolean that enables memoization for that rule. See "Memoization" below. If both memoization and error reporting are required, the boolean argument goes first and the label second.

Rules may be assigned 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 first (either uninitialized or initialized with a boolean, a label or both) and assigned a parsing expression afterwards. They cannot be copied. A

statement like r = r is normally supposed to do nothing, but this is not the case with rules. If the r on the left is a rule, the r on the right is a *rule expression* that is assigned to the r on the left. This statement makes r left recursive.

Left recursion

A PEG is left recursive if it contains at least one reachable 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 grammar has the following reference 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 from the input. Without this LPAR the grammar would be left recursive. This is an inherent limitation of PEG grammars: reference cycles that may close without previously consuming at least one character are forbidden.

Parsing a grammar

Once built, a grammar is parsed by calling the parse() method of a parser initialized with the grammar's start rule. Parse() returns true if the grammar is parsed successfully. When parse() succeeds, the parser 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 moved 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 at the time each action was scheduled, so that the text() method returns the correct string and accessing the value stack with val() 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 move.

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

```
while ( p.parse() )
    p.accept();
```

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

Using the value stack

Scheduled actions and semantic predicates may return a value from their rule by assigning slot 0 of the value stack, val(0). The value stack is based on a self-resizing vector that is indexed relative to the current offset that each rule has when it starts parsing.

When a rule r1 is used as a basic expression in another rule r2, the value assigned to val(0) by r1 appears to r2 as val(n), where n 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 us that the first factor is in slot 0 and the second ones are in slot 2:

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

If an expression has alternative branches, 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 branch matches when parsing (either a1 ... a2 or b1 ... b4).

Scheduled action and semantic predicate macros

Peg.h defines some macros for using lambdas as scheduled 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 scheduled actions and semantic predicates to expressions in a grammar does not affect the position of each expression in the value stack. This is useful when it is necessary to add new actions or predicates to an existing grammar without shifting references to the value stack in existing actions and predicates.

These macros are useful but optional. Do() and Pred() can always be hand coded, either directly or implicitly:

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

This simple palindrome recognizer uses semantic predicates and parsing time actions:

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

#include "peg.h"

using namespace std;
using namespace peg;

int main()
{
    class pal_parser : public Parser<string>
    {
        Rule start, pal, chr;

    public:

        pal_parser(istream &in = cin) : Parser(start, in)
        {
            start   = pal--                 do_( cout << text() << endl; )
                    ;

            pal     = chr >> pal >> chr     if_( val(0) == val(2) )
                    | chr >> chr            if_( val(0) == val(1) )
                    | chr
                    ;

            chr     = Any()--               pa_( val(0) = text(); )
                    ;
        }
    };

    pal_parser p;

    while ( p.parse() )
        p.accept();
}
```

The pal rule defines a palindrome as a symmetric string, including those 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. But this is just a toy example.

The Parser class

The Parser<T> class (where T is not void) has the following 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;
std::string get_error() const;

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

// Reference to a value stack slot
T &val(std::size_t idx);
const T &val(std::size_t idx) const;

// Reference to a value contained in a variant value stack slot
template <typename U> U &val(std::size_t idx);
template <typename U> const U &val(std::size_t idx) const;
```

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). The value stack uses a self-resizing vector. Depending on T, resizing a vector of Ts may be a costly operation. In order to avoid resizing, you may define a bigger initial capacity.

- parse() parses the start rule.
- accept() executes the scheduled actions and discards matched input.
- clear() discards all input and scheduled actions.
- text() returns a string with the contents of the most recently closed text capture.
- get_error() returns a string describing a parsing error, only meaningful if labeled rules are used in the grammar and parse() fails. See "Error reporting" below.
- If PEG_DEBUG is defined, check() checks the grammar from the start rule.
- val(n) and val<U>(n) access the value stack.

A concrete parser usually extends Parser<T>, defines the rules as instance variables and builds the grammar in the constructor.

Method val(n) returns a reference to the T in slot n of the stack. Rules return their values by assigning val(0). For example, in a Parser<std::variant<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 of a variant value stack, you must use the second form of val(), with 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, or throws std::bad_variant_access if the slot does not currently hold a value of type U.

Knowing that slot 3 of a variant 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 of variant types can be done without type qualification. For example, variants may be compared directly:

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

This checks that slots n1 and n2 hold the same type and have the same value.

If you don't need a value stack, you can declare the parser as Parser<void>, Parser<> or just Parser. This class lacks the second form of the constructor and the val() methods.

An example with a variant value stack

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<variant<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

Error reporting

When a parser finds an error in its input, one would like it to report the position and nature of the error. This is difficult for PEG parsers because as a result of backtracking, the detection of an error can occur at a position far away from where the actual error is.

A good treatment of this problem can be found in this paper:

http://www.inf.puc-rio.br/~roberto/docs/sblp2013-1.pdf

PEGPP implements the “farthest failure position” heuristic described there, with the help of labeled rules.

The parser remembers the farthest position where a labeled rule has failed and the set of labels of the labeled rules that failed at that position. When a labeled rule fails, the current position is compared to the farthest failure position. If it is smaller, nothing is done. If it is the same (this means that after backtracking from a previous failure the parser fails at the same position as before, but this time attempting to parse a different rule) the label is added to the set of labels already stored. If it is greater, the farthest failure position is updated, the label set is emptied and the label of the failing expression is added to it.

When parse() fails, get_error() may be called. It returns a string describing the error as follows:

```
Line <line number>
Expecting <label 1> <label 2> ... <label N>
Found <60 characters max. of input, starting at the farthest failure position>
```

Label 1, label 2 ... label N are the labels of the labeled rules that have failed at the farthest failure position. The parser keeps line numbering and remembers the positions of newlines in the input. Line numbering is not reset by accept(), so that parsing can be done by calling parse() and accept() in a loop until parse() fails. Line numbering is only reset by clear().

This is how a labeled rule might be defined:

```
Rule Plus{“PLUS”};
Plus = ‘+’ >> WhiteSpace;
```

An example using labeled rules

We add labels to some rules of our integer calculator:

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

#include "peg.h"

using namespace std;
using namespace peg;

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

    bool eof{false};

public:

    bool at_eof() { return eof; }

    intcalc(istream &in = cin ) : Parser(calc, in)
    {
        // Lexical rules
        WS          = *" \t\f\r\n"_ccl;
        SIGN        = "+-"_ccl;
        DIGIT       = "0-9"_ccl;
        NUMBER      = (~SIGN >> +DIGIT)-- >> WS     do_( val(0) = stoi(text()); );
        LPAR        = '(' >> WS;
        RPAR        = ')' >> WS;
        ADD         = '+' >> WS;
        SUB         = '-' >> WS;
        MUL         = '*' >> WS;
        DIV         = '/' >> WS;

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

int main()
{
    intcalc calc;

    while ( calc.parse() )
        if ( calc.at_eof() )
            return 0;
        else
            calc.accept();

    cerr << calc.get_error() << endl;
}
```

We have labeled those rules that represent basic lexical units: numbers (NUMBER), parentheses (LPAR, RPAR) and arithmetic operators (ADD, SUB, MUL, DIV). In a two-level parser these would be recognized by the lexer as tokens.

We have also added an end-of-file check in the calc rule, so that we can exit without error on an empty input.

Let’s try a correct input first and then a couple of erroneous inputs:

```
Input:

(1+2)*3
((2+3)*(5))

Output:

9
25
```

```
Input:

(1+2)*3
((2+3)*(5)
Hello

Output:

9
Line 3
Expecting MUL DIV ADD SUB RPAR
Found Hello

Input:

(1+2)*3
((2+3)*(5)+
Hello

Output:

9
Line 3
Expecting NUMBER LPAR
Found Hello
```

Memoization

As PEG parsers can backtrack, depending on the grammar, rules may be parsed several times with the same input context, producing identical results each time. This is a typical example:

```
Rule r, a, b, c;

r = a >> b
  | a >> c
  ;
```

Parsing r starts by parsing a in the first branch of r. Let's assume that a parses successfully but then b fails. The parser backtracks and tries the second branch, which parses a again with the input pointer at the same position as before. Parsing a will succeed again and produce the same results as the first time. This is a trivial case, which can be hand optimized by factoring a as a common prefix for both branches:

```
Rule r, a, b, c;

r = a >> ( b | c );
```

Other grammars are not so easy to optimize. The general solution to avoid unnecessary repeated parsing of a rule with the same input context is to remember the results of the first time and apply them directly on repetitions. This is called "memoizing". Memoizing all rules is called "packrat parsing". Memoization has a cost, both in terms of memory usage and execution time, so PEGPP allows enabling it on a rule by rule basis. Rules that are costly to parse and likely to be parsed more than once with the same input context are good candidates for memoization. In our example, we would memoize a:

```
Rule r, a{true}, b, c;

r = a >> b
  | a >> c
  ;
```

In PEGPP the input context includes:

- The current position of the input pointer.
- The current base of indices into the value stack, if the parser uses one (because semantic predicates can succeed or fail depending on value stack contents).
- The current lookahead mode (because lookahead disables scheduling actions).

The memoizer remembers the result of parsing (success or failure) and, in case of success:

- The position of the input pointer after parsing.
- The last text capture.
- Actions scheduled during parsing.
- The value of val(0) (if the parser uses a value stack).

Please bear in mind that the memoizer will not remember any side effects of semantic predicates except the value assigned to val(0).

An example of memoization

This is our previous palindrome recognizer, with some modifications:

```
#include <string>
#include <iostream>
#include <chrono>
#include <thread>

#include "peg.h"

using namespace std;
using namespace peg;

int main(int argc, char *argv[])
{
    static bool memoize = argc > 1;

    class pal_parser : public Parser<string>
    {
        Rule start, pal, chr{memoize};

    public:

        pal_parser(istream &in = cin) : Parser(start, in)
        {
            start   = pal                                                   do_( cout << val(0) << endl; )
                    ;

            pal     = chr >> pal >> chr     if_( val(0) == val(2) )     do_( val(0) += val(1) + val(2); )
                    | chr >> chr            if_( val(0) == val(1) )     do_( val(0) += val(1); )
                    | chr
                    ;

            chr     = Any()--               pa_( val(0) = text(); this_thread::sleep_for(100ms); )
                                                                        do_( val(0) = text(); )
                    ;
        }
    };

    pal_parser p;

    const auto start = chrono::steady_clock::now();
    while ( p.parse() )
        ;
    const auto end = chrono::steady_clock::now();
    cout << "Parsing time: " << (end - start) / 1ms << "ms\n";

    p.accept();
}
```

We have conditionally memoized the chr rule and introduced (unnecessary) scheduled actions that reference the value stack, just to check that the memoizer handles them correctly. We have also made parsing chr artificially costly by introducing a 100ms delay and we measure the total parsing time as the difference between two timestamps.

Running this parser with chr memoized shows a reduction of total parsing time as compared to the unmemoized case.

Memoized:

```
$ ./mpal memo << eof
> 12345678909876543321abcdefedcba
> eof
Parsing time: 6119ms
1234567890987654321
abcdefedcba
```

Unmemoized:

```
$ ./mpal << eof
> 12345678909876543321abcdefedcba
> eof
Parsing time: 15545ms
1234567890987654321
abcdefedcba
```

The exact ratio depends on the actual input, but memoization of chr typically reduces the total parsing time to less than one half. This means that more than half of the time the results of parsing chr are found in the memoizer and actual parsing is skipped.

Checking a grammar

If the macro PEG_DEBUG is defined before including peg.h, rules are compiled with a check() method that visits all grammar paths reachable from that 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.

Check() cannot be called more than once on a grammar, since it writes some accounting information in the rules. It would be pointless anyway. Once a grammar is checked, both the PEG_DEBUG macro and the call to check() should be removed.

In debug mode each rule may be assigned a name. 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 names assigned 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 call the parser's check() method, which in turn calls the check() method of the start rule:

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

check();
```

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. Rules are 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 (we make it ~LPAR). This change makes the grammar left-recursive, as now the cycle expression → term → factor → expression may close without reading any 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 all reachable rules, 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

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.