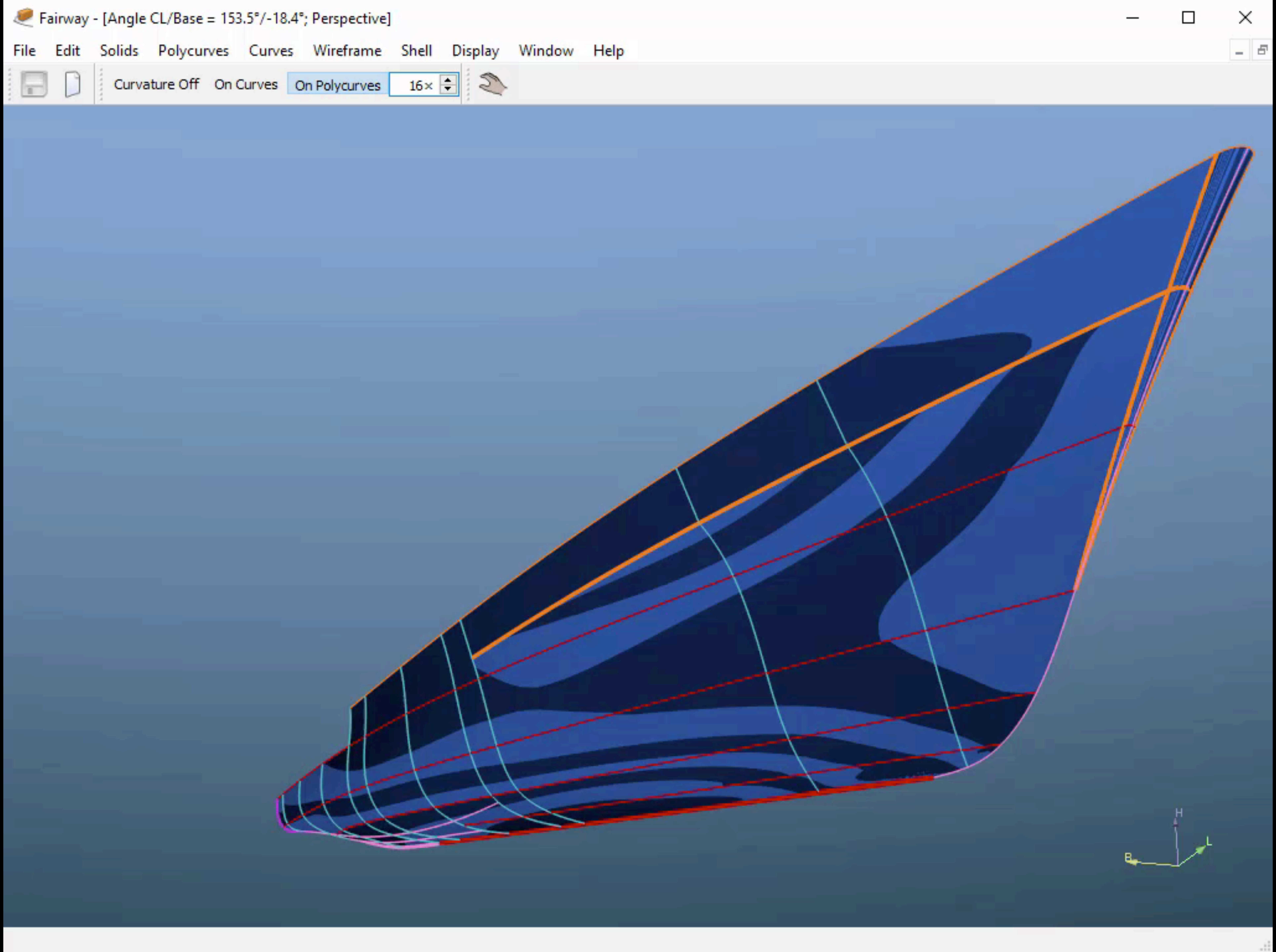
Friedhelm Lang


BAW
Bundesanstalt für Wasserbau
Kompetenz für die Wasserstraßen
Wasserspiegellagen-
änderung [m]
(bez. 71.5 m+NN)
0.50
0.30
0.10
-0.10
-0.30
-0.50
Druckverteilung
[Pa]
(abz. hydrost. Druck)
1000.0
400.0
-200.0
-800.0
-1400.0
-2000.0
Bildmaterial: FutureShip
FutureShip

Holger Weinandt

Allseas


Allseas


Fairway - [Angle CL/Base = 153.5°/-18.4°; Perspective]
File
Edit
Solids
Polycurves
Curves
Wireframe
Shell
Display
Window
Help
Curvature Off
On Curves
On Polycurves
16×
H
L
B

Extended Pascal

- Fast compilation
- Fast code
- Small language
- ANSI/ISO standard

- Nested functions
- Compilation speed
- Garbage collection

- Nested functions
- Compilation speed
- Garbage collection
- Experimentation
- Easy parallelisation

- Nested functions
- Compilation speed
- Garbage collection
- Experimentation
- Easy parallelisation
- Contract programming
- @safe
- Unit tests

- Nested functions
- Compilation speed
- Garbage collection
- Experimentation
- Easy parallelisation

- Contract programming
- @safe
- Unit tests
- Clean templates
- WysiwygString

Welcome to DUB, the D package registry. The following list shows all available packages:

Select category: Development library ▾ Development support libraries ▾ Parsers and lexers ▾

Name	Last update ↕	Registered	Description
dproto	2.1.3, *12 days ago*	2013-Oct-05	D Protocol Buffer library
pegged	0.4.2, *17 days ago*	2013-Mar-08	Parsing Expression Grammar (PEG) generator
bencode	1.0.0, *18 days ago*	2017-Apr-16	bencode parser/deserializer and serializer
coda-d	0.0.3, *2 months ago*	2017-Mar-28	Coda bindings, NLP stack for Russian and English languages
sdpc	0.1.1, *2 months ago*	2015-Jun-13	Stupid D parser combinator
yaml-d	0.0.1, *2 months ago*	2017-Mar-19	A YAML parser and emitter in D.
libdparse	0.7.1-beta.1, *2 months ago*	2014-Sep-04	Library for lexing and parsing D source code
expat-d	0.1.1, *3 months ago*	2017-Feb-26	D bindings for the Expat XML parser
pry	0.3.2, *4 months ago*	2016-Dec-15	Pry - practical parser combinators library for D
mustache-d	0.1.3, *4 months ago*	2013-Mar-20	Mustache template engine for D.
tcenal	0.0.3, *5 months ago*	2015-Jan-11	A compile-time syntax extension library.
dyaml-dlang-tour	0.6.0, *5 months ago*	2016-Nov-15	YAML parser and emitter
libdominator	1.1.2, *5 months ago*	2016-Jun-29	A HTML Parser Library
cssd	0.1.3, *5 months ago*	2015-Apr-02	CSS library for D
dateparser	3.0.0, *7 months ago*	2016-Mar-07	Library for parsing randomly formatted date strings
sdlang-d	0.10.1, *7 months ago*	2013-Mar-06	An SDL (Simple Declarative Language) library for D.
jin-tree	3.0.0, *8 months ago*	2015-Jan-26	Tree - simple fast compact user-readable binary-safe extensible structural

https://github.com/PhilippeSigaud/Pegged/wiki/Pegged-Tutorial

Pegged Tutorial

The following pages will show you how to use **Pegged** to create grammars and parse inputs with them.

1. PEG Basics
2. Declaring a Grammar
3. Using a Grammar
4. Memoization
5. Grammars as D Modules
6. Grammar Composition
7. Using the Parse Tree
8. Tree Decimation
9. Extended PEG Syntax
10. Parameterized Rules
11. Semantic Actions
12. Generating Code
13. User-Defined Parsers
14. Writing Your Own Grammar
15. Behind the Curtain: How Pegged Works
16. Optimizations
17. Grammar Debugging
18. Grammar Testing

Grammar

ISO/IEC 10206:1990(E)

6.9.3.4 If-statements

if-statement = 'if' Boolean-expression 'then' statement [else-part] .

else-part = 'else' statement .

```
`
# 6.9.3.4 If-statements
    IfStatement <- 'if' BooleanExpression 'then' Statement ( ElsePart )?
    ElsePart    <- 'else' Statement
`
```

```
import pegged.grammar;

mixin(grammar(`
Arithmetic:
    Term     < Factor (Add / Sub)*
    Add      < "+" Factor
    Sub      < "-" Factor
    Factor   < Primary (Mul / Div)*
    Mul      < "*" Primary
    Div      < "/" Primary
    Primary  < Parens / Neg / Number / Variable
    Parens   < :"(" Term :")"
    Neg      < "-" Primary
    Number   < ~([0-9]+)
    Variable <- identifier
`));

void main()
{
    enum parseTree = Arithmetic("1 + 2 - (3 * 2 - 5) * 6");
    import std.stdio;
    writeln(parseTree);
}
```

```
$ rdmd -I../../.. arithmetic.d
Arithmetic [0, 23]["1", "+", "2", "-", "3", "*", "2", "-", "5", "*", "6"]
 +-Arithmetic.Term [0, 23]["1", "+", "2", "-", "3", "*", "2", "-", "5", "*", "6"]
    +-Arithmetic.Factor [0, 2]["1"]
    |  +-Arithmetic.Primary [0, 2]["1"]
    |     +-Arithmetic.Number [0, 2]["1"]
    +-Arithmetic.Add [2, 6]["+", "2"]
    |  +-Arithmetic.Factor [4, 6]["2"]
    |     +-Arithmetic.Primary [4, 6]["2"]
    |        +-Arithmetic.Number [4, 6]["2"]
    +-Arithmetic.Sub [6, 23]["-", "3", "*", "2", "-", "5", "*", "6"]
       +-Arithmetic.Factor [8, 23]["3", "*", "2", "-", "5", "*", "6"]
          +-Arithmetic.Primary [8, 20]["3", "*", "2", "-", "5"]
          |  +-Arithmetic.Parens [8, 20]["3", "*", "2", "-", "5"]
          |     +-Arithmetic.Term [9, 18]["3", "*", "2", "-", "5"]
          |        +-Arithmetic.Factor [9, 15]["3", "*", "2"]
          |        |  +-Arithmetic.Primary [9, 11]["3"]
          |        |  |  +-Arithmetic.Number [9, 11]["3"]
          |        |  +-Arithmetic.Mul [11, 15]["*", "2"]
          |        |     +-Arithmetic.Primary [13, 15]["2"]
          |        |        +-Arithmetic.Number [13, 15]["2"]
          |        +-Arithmetic.Sub [15, 18]["-", "5"]
          |           +-Arithmetic.Factor [17, 18]["5"]
          |              +-Arithmetic.Primary [17, 18]["5"]
          |                 +-Arithmetic.Number [17, 18]["5"]
          +-Arithmetic.Mul [20, 23]["*", "6"]
             +-Arithmetic.Primary [22, 23]["6"]
                +-Arithmetic.Number [22, 23]["6"]
```

```
import pegged.grammar;

mixin(grammar(`
Arithmetic:
    Term     < Factor (Add / Sub)*
    Add      < "+" Factor
    Sub      < "-" Factor
    Factor   < Primary (Mul / Div)*
    Mul      < "*" Primary
    Div      < "/" Primary
    Primary  < Parens / Neg / Number / Variable
    Parens   < :"(" Term :")"
    Neg      < "-" Primary
    Number   < ~([0-9]+)
    Variable <- identifier
`));

void main()
{
        enum parseTree = Arithmetic("1 + 2 - (3 * 2 - 5) * 6");

        float value(ParseTree p)
        {
            switch (p.name)
            {
                case "Arithmetic", "Arithmetic.Primary", "Arithmetic.Parens", "Arithmetic.Add", "Arithmetic.Mul":
                    return value(p.children[0]);
                case "Arithmetic.Sub", "Arithmetic.Neg":
                    return -value(p.children[0]);
                case "Arithmetic.Div":
                    return 1.0/value(p.children[0]);
                case "Arithmetic.Term":
                    float v = 0.0;
                    foreach(child; p.children) v += value(child);
                    return v;
                case "Arithmetic.Factor":
                    float v = 1.0;
                    foreach(child; p.children) v *= value(child);
                    return v;
                case "Arithmetic.Number":
                    import std.conv: to;
                    return to!float(p.matches[0]);
                default:
                    return float.nan;
            }
        }

        import std.stdio;
        enum answer = value(parseTree)
        writeln(answer);
}
```

```d
import pegged.grammar;

mixin(grammar(`
Arithmetic:
    Term     < Factor (Add / Sub)*
    Add      < "+" Factor
    Sub      < "-" Factor
    Factor   < Primary (Mul / Div)*
    Mul      < "*" Primary
    Div      < "/" Primary
    Primary  < Parens / Neg / Number / Variable
    Parens   < :"(" Term :")"
    Neg      < "-" Primary
    Number   < ~([0-9]+)
    Variable <- identifier
`));

void main()
{
        enum parseTree = Arithmetic("1 + 2 - (3 * 2 - 5) * 6");

        float value(ParseTree p)
        {
            switch (p.name)
            {
                case "Arithmetic", "Arithmetic.Primary", "Arithmetic.Parens", "Arithmetic.Add", "Arithmetic.Mul":
                    return value(p.children[0]);
                case "Arithmetic.Sub", "Arithmetic.Neg":
                    return -value(p.children[0]);
                case "Arithmetic.Div":
                    return 1.0/value(p.children[0]);
                case "Arithmetic.Term":
                    float v = 0.0;
                    foreach(child; p.children) v += value(child);
                    return v;
                case "Arithmetic.Factor":
                    float v = 1.0;
                    foreach(child; p.children) v *= value(child);
                    return v;
                case "Arithmetic.Number":
                    import std.conv: to;
                    return to!float(p.matches[0]);
                default:
                    return float.nan;
            }
        }

        import std.stdio;
        enum answer = value(parseTree)
        writeln(answer);
}
```

```
$ rdmd -I../../.. arithmetic.d
-3
```

Rule ➔ D

```
A <- B C D+ E?
```

```
import pegged.peg;  // Predefined parser combinators

ParseTree A(ParseTree p)
{
    return and!(B, C, oneOrMore!(D), option!(E))(p);
}
```

PEG

- **Bryan Ford**, *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*, Symposium on Principles of Programming Languages, 2004
- Top-down recursive descent
- Prioritised choice
- Greedy * and + (A* A never succeeds)
- Linear-time (through memoization)

PEG formally describing its own syntax:

```
Grammar    <- Spacing Definition+ EndOfFile
Definition <- Identifier LEFTARROW Expression
Expression <- Sequence (SLASH Sequence)*
Sequence   <- Prefix+
Prefix     <- (AND / NOT)? Suffix
Suffix     <- Primary ( QUESTION / STAR / PLUS )?
Primary    <- Identifier !LEFTARROW / OPEN Expression CLOSE / Literal / Class / DOT
Identifier <- IdentStart IdentCont* Spacing
IdentStart <- [a-zA-Z_]
IdenCont   <- IdenStart / [0-9]
Literal    <- ['] (!['] Char)* ['] Spacing
            / ["] (!["] Char)* ["] Spacing
Class      <- '[' (!']' Range)* ']' Spacing
Range      <- Char '-' Char / Char
Char       <- '\\' [nrt'"\[\]\\]
            / '\\' [0-2][0-7][0-7]
            / '\\' [0-7][0-7]?
            / !'\\' .
# Terminals
LEFTARROW  <- "<-" Spacing
SLASH      <- '/' Spacing
AND        <- '&' Spacing
NOT        <- '!' Spacing
QUESTION   <- '?' Spacing
STAR       <- '*' Spacing
PLUS       <- '+' Spacing
OPEN       <- '(' Spacing
CLOSE      <- ')' Spacing
DOT        <- '.' Spacing
# Blanks
Spacing    <- (Space / Comment)*
Comment    <- "#" (!EOL .)* (EOL/EOI)
Space      <- ' ' / '\t' EndOfLine
EndOfLine  <- '\r\n' / '\n' / '\r'
EndOfFile  <- !.
```

Bootstrapping:

1. Want a parser for PEG
2. Write a D implementation for each rule using parser combinators
3. Use this to generate a parse tree
4. Write an evaluator that converts the parse tree into a set of parsers (automating 2)

Ordinary recursion

Parse strings like “n”, “n+n”, “n+n+n”, etc.

```
R <- "n+" R / "n"
```

```
import pegged.peg;

ParseTree R(ParseTree p)
{
    return or!(and!(literal!("n+"), R), literal!("n"))(p);
}

void main()
{
    import std.stdio;
    ParseTree p = { input : "n+n" };
    auto result = R(p);
    writeln(result);
}
```

```
$ rdmd -I../../.. ../../../libpegged.a recursion.d
or!(and!(literal, R), literal!("n")) [0, 3]["n+", "n"]
 +-and!(literal, R) [0, 3]["n+", "n"]
    +-literal!("n+") [0, 2]["n+"]
    +-or!(and!(literal, R), literal!("n")) [2, 3]["n"]
       +-literal!("n") [2, 3]["n"]
```

Left-recursion

Parse strings like “n”, “n+n”, “n+n+n”, etc.

```
R <- R "+n" / "n"
```

```
import pegged.peg;

ParseTree R(ParseTree p)
{
    return or!(and!(R, literal!("+n")), literal!("n"))(p);
}

void main()
{
    import std.stdio;
    ParseTree p = { input : "n+n" };
    auto result = R(p);
    writeln(result);
}
```

```
$ rdmd -I../../.. ../../../libpegged.a left_recursion_naive.d
Segmentation fault: 11
```

Left-recursion

```
R <- R "+n" / "n"
```

direct left-recursion

```
R <- S? R "+n" / "n"
```

hidden left-recursion

```
R <- S R "+n" / "n"
S <- A B C / D*
```

```
R <- T "+n" / "n"
T <- G H / J
J <- K / R L
```

indirect left-recursion

```
R <- T <- J <- R
```

```
R <- S T "+n" / "n"
S <- A B C / D*
T <- G H / J
J <- K / R L
```

hidden indirect left-recursion

```
R <- T <- J <- R
```

Left-recursion

input: `nlm-n+(aaa)n`

```
E <- F 'n' / 'n'
F <- E '+' I* / G '-'
G <- H 'm' / E
H <- G 'l'
I <- '(' A+ ')'
A <- 'a'
```

interlocking cycles

```
E <- F <- E
```

```
G <- H <- G
```

```
E <- F <- G <- E
```

input: `x(n)(n).x(n).x`

```
L <- P '.x' / 'x'
P <- P '(n)' / L
```

mutual left-recursion

```
L <- P <- L
```

```
P <- P
```

The Cure

- **Sergio Medeiros** et al., *Left Recursion in Parsing Expression Grammars*, Programming Languages, Volume 7554 of the series Lecture Notes in Computer Science pp 27-41, 2012.
- **Nicolas Laurent, Kim Mens**, *Parsing Expression Grammars Made Practical*, Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp 167-172.

Bounded left-recursion:

1. A way to control recursion step by step
2. A condition for when to stop recursion

Stopping condition

Parse strings like “n”, “n+n”, “n+n+n”, etc.

```
Rᵢ <- Rᵢ₋₁ "+n" / "n"
```

```
R₋₁ <- fail
```

```
R₀ <- R₋₁ "+n" / "n"
```

→

```
R₀ <- "n"
```

```
R₁ <- R₀ "+n" / "n"
```

→

```
R₁ <- "n" "+n" / "n"
```

```
R₂ <- R₁ "+n" / "n"
```

→

```
R₂ <- ("n" "+n" / "n") "+n" / "n"
```

Recursion should stop
when the length of the match stops to grow

Recursion control

Parse strings like “n”, “n+n”, “n+n+n”, etc.

```
// R <- R "+n" / "n"
ParseTree R(ParseTree p)
{
    static ParseTree prev = none;                      // No recursion has happened.
    if (prev != none)                                  // We are recursing. Don't evaluate R anew
        return prev;                                   //     return the memoized result instead.
    ParseTree current = fail;                          // R_{-1}.
    prev = current;                                    // Stop on next recursive call.
    while (true)                                       // Controlled loop, try R with increased
    {                                                  //      recursion bound.
        ParseTree result = or!(and!(R, literal!("+n")), literal!("n"))(p);
        if (result.length > current.length)            // The match length is growing, continue.
        {
            prev = result;                             // Memoize R_{n-1} for when we recurse.
            current = result;
        }
        else                                           // Optimum bound exceeded, current is
        {                                              //       the best match.
            prev = none;                               // Clean up.
            return current;                            // Done.
        }
    }
}
```

Recursion control

Parse strings like “n”, “n+n”, “n+n+n”, etc.

```
// R <- R "+n" / "n"
ParseTree R(ParseTree p)
{
    static ParseTree[size_t /*position*/] prev;
    if (auto s = p.end in prev)              // We are recursing. Don't evaluate R anew
        return *s;                           //      return the memoized result instead.
    ParseTree current = fail(p);             // R_{-1}.
    prev[p.end] = current;                   // Stop on next recursive call.
    while (true)                             // Controlled loop, try R with increased
    {                                        //       recursion bound.
        ParseTree result = or!(and!(R, literal!("+n")), literal!("n"))(p);
        if (result.end > current.end)        // The match length is growing, continue.
        {
            prev[p.end] = result;            // Memoize R_{n-1} for when we recurse.
            current = result;
        }
        else                                 // Optimum bound exceeded, current is
        {                                    //        the best match.
            prev.remove(p.end);              // Clean up.
            return current;                  // Done.
        }
    }
}
```

```d
import pegged.peg;

ParseTree R(ParseTree p)
{
    static ParseTree[size_t /*position*/] prev;
    if (auto s = p.end in prev)                 // We are recursing. Don't evaluate R anew
        return *s;                              //     return the memoized result instead.
    ParseTree current = fail(p);                // R_{-1}.
    prev[p.end] = current;                      // Stop on next recursive call.
    while (true)                                // Controlled loop, try R with increased
    {                                           //     recursion bound.
        ParseTree result = or!(and!(R, literal!("+n")), literal!("n"))(p);
        if (result.end > current.end)           // The match length is growing, continue.
        {
            prev[p.end] = result;               // Memoize R_{n-1} for when we recurse.
            current = result;
        }
        else                                    // Optimum bound exceeded, current is
        {                                       //     the best match.
            prev.remove(p.end);                 // Clean up.
            return current;                     // Done.
        }
    }
}

void main()
{
    import std.stdio;
    ParseTree p = { input : "n+n" };
    auto result = R(p);
    writeln(result);
}
```

```d
import pegged.peg;

ParseTree R(ParseTree p)
{
    static ParseTree[size_t /*position*/] prev;
    if (auto s = p.end in prev)                     // We are recursing. Don't evaluate R anew
        return *s;                                  //     return the memoized result instead.
    ParseTree current = fail(p);                    // R_{-1}.
    prev[p.end] = current;                          // Stop on next recursive call.
    while (true)                                    // Controlled loop, try R with increased
    {                                               //       recursion bound.
        Par                                                                  continue.
        if
        {
                                                                             ecurse.

        }
        els                                                                  nt is
        {                                           //       the best match.
            prev.remove(p.end);                     // Clean up.
            return current;                         // Done.
        }
    }
}

void main()
{
    import std.stdio;
    ParseTree p = { input : "n+n" };
    auto result = R(p);
    writeln(result);
}
```

```
$ rdmd -I../../.. ../../../libpegged.a left_recursion_proper.d
or!(and!(R, literal), literal!("n")) [0, 3]["n", "+n"]
 +-and!(R, literal) [0, 3]["n", "+n"]
    +-or!(and!(R, literal), literal!("n")) [0, 1]["n"]
    |  +-literal!("n") [0, 1]["n"]
    +-literal!("+n") [1, 3]["+n"]
```

```d
import pegged.peg;

ParseTree R(ParseTree p)
{
    static ParseTree[size_t /*position*/] prev;
    if (auto s = p.end in prev)                    // We are recursing. Don't evaluate R anew
        return *s;                                 //     return the memoized result instead.
    ParseTree current = fail(p);                   // R_{-1}.
    prev[p.end] = current;                         // Stop on next recursive call.
    while (true)                                   // Controlled loop, try R with increased
    {                                              //     recursion bound.
        Par
        if                                                                          continue.
        {
                                                                                    ecurse.

        }
        els                                                                         nt is
        {                                          //     the best match.

        }
    }
}

void main()
{
    import std.stdio;
    ParseTree p = { input : "n+n" };
    auto result = R(p);
    writeln(result);
}
```

```
$ rdmd -I../../.. ../../../libpegged.a left_recursion_proper.d
or!(and!(R, literal), literal!("n")) [0, 3]["n", "+n"]
 +-and!(R, literal) [0, 3]["n", "+n"]
    +-or!(and!(R, literal), literal!("n")) [0, 1]["n"]
    |  +-literal!("n") [0, 1]["n"]
    +-literal!("+n") [1, 3]["+n"]
```

```
$ rdmd -I../../.. ../../../libpegged.a recursion.d
or!(and!(literal, R), literal!("n")) [0, 3]["n+", "n"]
 +-and!(literal, R) [0, 3]["n+", "n"]
    +-literal!("n+") [0, 2]["n+"]
    +-or!(and!(literal, R), literal!("n")) [2, 3]["n"]
       +-literal!("n") [2, 3]["n"]
```

Generic solution:
Cures all kinds of left-recursion

1. Analysis: identify left-recursive cycles
2. Instrument one rule in every cycle
3. Pause memoization while in left-recursion

Fully automatic

Generic solution:
Cures all kinds of left-recursion

```
/** Left-recursive cycles:
ComponentVariable <- IndexedVariable <- ArrayVariable <- VariableAccess
VariableAccess <- IdentifiedVariable <- PointerVariable
VariableAccess <- BufferVariable <- FileVariable
VariableAccess <- SubstringVariable <- StringVariable
FunctionAccess <- ComponentFunctionAccess <- IndexedFunctionAccess <- ArrayFunction
FunctionAccess <- ComponentFunctionAccess <- RecordFunctionAccess <- RecordFunction
FunctionAccess <- SubstringFunctionAccess <- StringFunction
ComponentVariable <- IndexedVariable <- StringVariable <- VariableAccess
ComponentVariable <- FieldDesignator <- RecordVariable <- VariableAccess
ConstantAccess <- ConstantAccessComponent <- IndexedConstant <- ArrayConstant
ConstantAccess <- ConstantAccessComponent <- IndexedConstant <- StringConstant
ConstantAccess <- ConstantAccessComponent <- FieldDesignatedConstant <- RecordConstant
ConstantAccess <- ConstantAccessComponent <- SubstringConstant <- StringConstant
*/
```

Fully automatic

Other PRs

- Interactive HTML5 output for large parse trees
- Case-insensitive matching of literals
- --config=tracer (using std.experimental.logger)
- longestOr (!=PEG)
- Pegged/pegged/examples/extended_pascal

Status

- Constructs parse trees for large EP files
- Include comments and white space
- No symbol table
- No focus on speed

Next

- Verify that EP language constructs have proper D equivalents
- Implement translation

Summary

- Ways in which D appeals to engineers
- Magic of Pegged
- Left-recursion explained
- Left-recursion cured
- Parsing Extended Pascal successfully
- Prospects are good