Pry - pragmatic parser combinators in D

Dmitry Olshansky

Dconf 2017

Setting up the stage

For me it all started out with std.regex in 2011

... aimed just to plug a hole in the ecosystem

actually got us in the top regex libraries!

The tools that got us up on that hill are:

1. Compile-time execution - *building data structures*
2. Compile-time codegen - *constructing the source code*

Earned lot of experience dealing with Unicode

crystalized in new std.uni (2012)

Has been in the regex arms race ever since

Simpleixty of regex

There is a *pure simple beautiful* subset of regex

- It's the one that actually runs fast
- Woefully underpowered though

And then there are extensions... ugly beasts

- Lookaround and backreferences kill any optimizations
- The power they add is marginal at best

All in all

- Highly overused due to popularity (use parser!)
- Have severe usability problems (100+ lines of regex)

Challenge is to create and popularize parser generators

State of things

Parser generators are generally frowned upon

- Most languages end up re-writing their parsers by hand

The general usability problems are:

- Cumbersome extra build step
- Poor error handling
- Low performance

Actually there is a number of parser generators in D

- In particular Pegged integrates nicely with the language
- But I find it idealistic and not performance minded

Ideals and Goals

Want a parser generator that

- Easy to use - less hassle than writing by hand
- Performs on par with handwritten parser
- Has simple and composable implementation
- Has sensible error handling

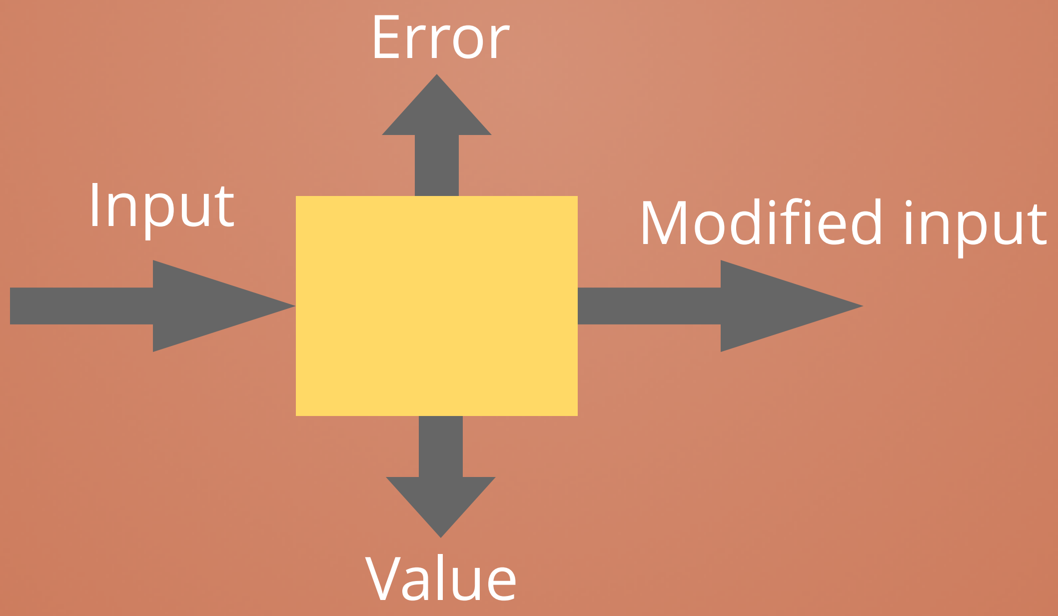
Key principle - performance first, features second

- If it's too slow nobody will use it
- Can always add features later, unlike performance

Parser combinators

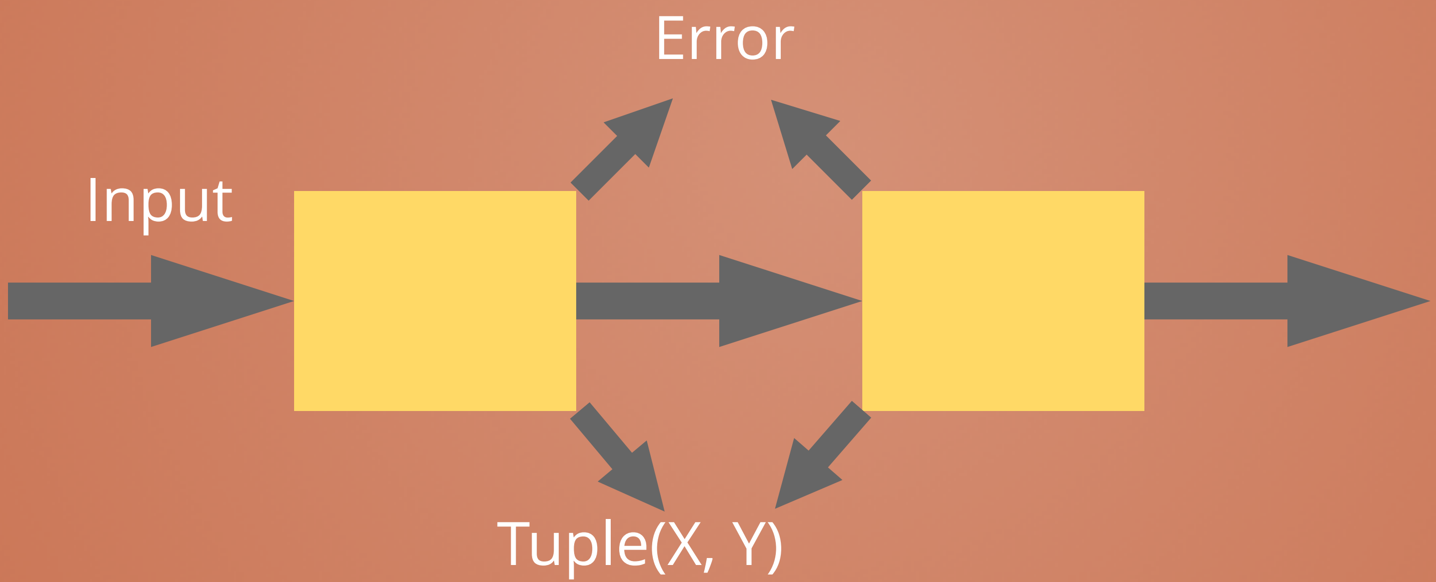
A parser is basically a function:

Input -> OneOf(Value, Error)



Parser combinators

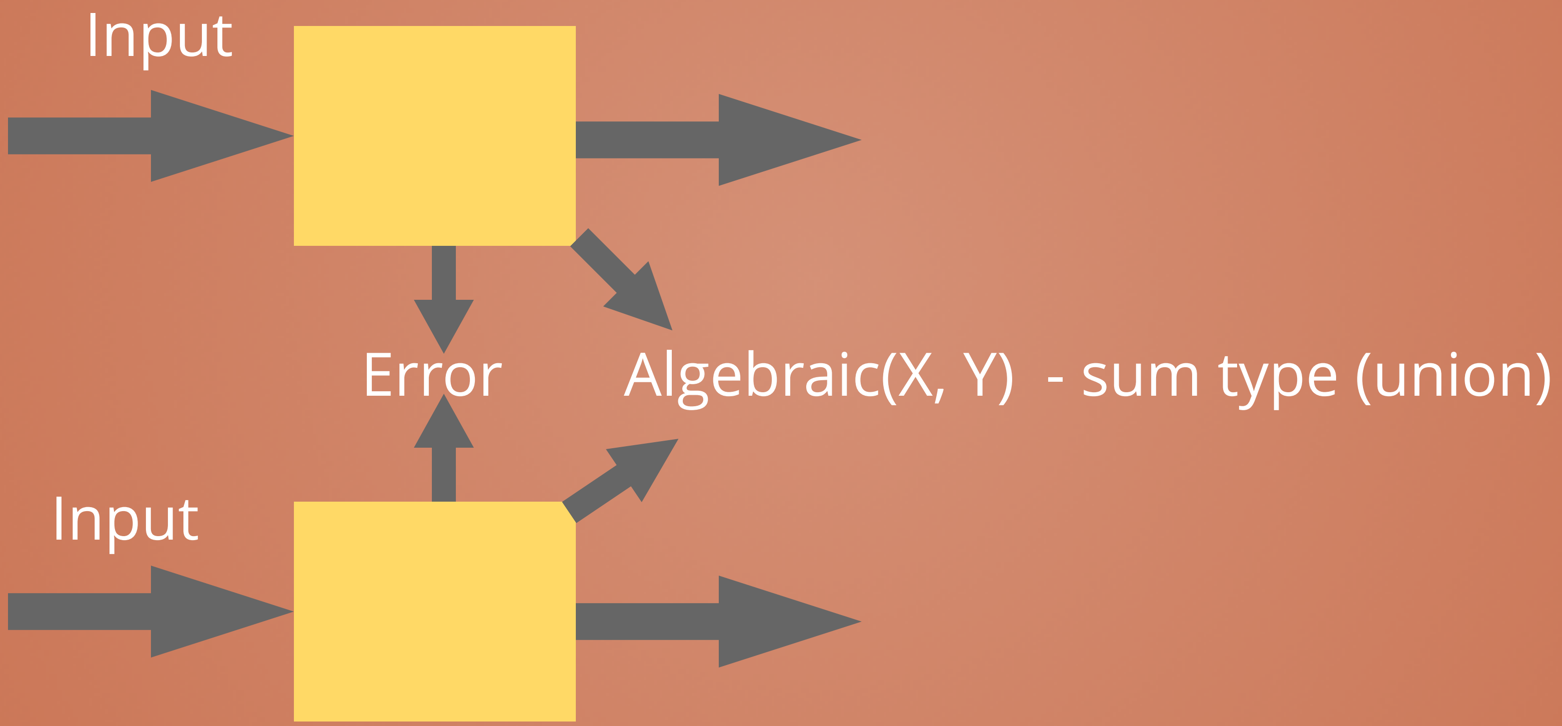
Naturally parsers can be combined as a sequence creating a new parser



If first parser succeeds the next one is applied the result is considered as a tuple of values

Parser combinators

Alternatively parsers can be combined as a choice



Only if the first parser fails the next one in the chain is tried, the result is naturally an Algebraic(X,Y)

Bits and pieces

Library generally provides

Atoms - basic building blocks:

token, literal, char class (a-la regex) , etc.

Combinators:

sequence, alternative, repetition, slice, delimited sequence, aa, lookahead, etc.

Grammar:

a module that constructs combinators from textual DSL - PEG grammar

New atoms and combinators could be easily written by user

Show me the code!

Let's consider the most basic parser - a fixed token

```
struct Tk(alias c) {
    static immutable msg = "expected '" ~ to!string(c)~"'";
    alias Value = ElementType!Stream;

    bool parse(ref Stream stream, ref Stream value, ref Stream.Error err) const {
        if(stream.empty) {
            err.location = stream.location;
            err.reason = "unexpected end of stream";
            return false;
        }
        if(stream.front == c){
            value = c;
            stream.popFront();
            return true;
        }
        else {
            err.location = stream.location;
            err.reason = msg;
            return false;
        }
    }
}

auto tk(alias c)(){ return Tk!c(); }
```

Char classes

More serious building block - test if char belongs to a set

```
struct Set(alias set) {
    import std.uni;
    enum val = set.byInterval.length;
    static if(val <= 6) {
        // Generate optimal "binary search" of if/else clauses
        mixin("static " ~ set.toSourceCode("test"));
    }
    else {
        // This actually builds multi-staged lookup table at compile-time
        static immutable matcher = CharMatcher(set);

        static bool test(dchar ch){
            return matcher[ch];
        }
    }
    ... // same as tk save for the test
}
```

This leverages the same fast lookup tables as std.regex

Sequence

Implementing a sequence with D's variadic templates

```
struct Seq(P...){
    alias Stream = ParserStream!(P[0]);
    alias Value = Tuple!(staticMap!(ParserValue, P));

    private P parsers;

    bool parse(ref Stream stream, ref Value value, ref Stream.Error err) const {
        auto save = stream.mark;
        foreach(i, ref p; parsers) {
            if(!p.parse(stream, value[i], err)){
                stream.restore(save);
                return false;
            }
        }
        return true;
    }
}
```

Alternative

Again going to use variadic template

```
struct Any(P...){
    alias Stream = ParserStream!(P[0]);
    alias Values = NoDuplicates!(staticMap!(ParserValue, P));
    alias Value = Algebraic!Values;
    private P parsers;

    bool parse(ref Stream stream, ref Value value, ref Stream.Error err) const {
        ...
    }
}
```

Alternative #2

```
bool parse(ref Stream stream, ref Value value, ref Stream.Error err) const {
        Stream.Error current;
        foreach(i, ref p; parsers) {
            ParserValue!(P[i]) tmp;
            static if(i == 0){
                if(p.parse(stream, tmp, err)){
                    value = tmp;
                    return true;
                }
            }
            else {
                if(p.parse(stream, tmp, current)){
                    value = tmp;
                    return true;
                }
                // pick the deeper error
                if(err.location < current.location){
                    err = current;
                }
            }
        }
        return false;
}
```

Array

Repeatedly apply a parser and append values to array

```
struct ArrayImpl(size_t minTimes, size_t maxTimes, Parser){
    alias Stream = ParserStream!Parser;
    alias Value = ParserValue!Parser[];
    private Parser parser;

    bool parse(ref Stream stream, ref Value value, ref Stream.Error err) const {
        auto start = stream.mark;
        ParserValue!Parser tmp;
        size_t i = 0;
        value = null;
        for(; i<minTimes; i++) {
            if(!parser.parse(stream, tmp, err)){
                stream.restore(start);
                return false;
            }
            value ~= tmp;
        }
        for(; i<maxTimes; i++){
            if(!parser.parse(stream, tmp, err)) break;
            value ~= tmp;
        }
        return true;
    }
}
```

Forward reference

Sometimes we need ability to do self-recursion

Have to reference a parser that is not fully constructed

```
interface DynamicParser(V) {
    bool parse(ref Stream stream, ref V value, ref Stream.Error err) const;
}

// Use LINE & FILE to provide unique types of dynamic.
auto dynamic(V, size_t line=__LINE__, string file=__FILE__)(){
    static class Dynamic : DynamicParser!V {
        DynamicParser!V wrapped;
    final:
        void opAssign(P)(P parser)
        if(isParser!P && !is(P : Dynamic)){
            wrapped = wrap(parser);
        }

        bool parse(ref Stream stream, ref V value, ref Stream.Error err) const {
            assert(wrapped, "Use of empty dynamic parser");
            return wrapped.parse(stream, value, err);
        }
    }
    return new Dynamic();
}
```

Forward reference

And the second bit - wrapping any parser as dynamic

```
auto wrap(Parser)(Parser parser){
    alias V = ParserValue!Parser;
    static class Wrapped: DynamicParser!V {
        Parser p;

        this(Parser p){
            this.p = p;
        }

        bool parse(ref Stream stream, ref V value, ref Stream.Error err) const {
            return p.parse(stream, value, err);
        }
    }
    return new Wrapped(parser);
}
```

May raise a valid concern about performance

Practical example

A simple arithmetic expression parser

```
auto calc(){
    with(parsers!string) {
        auto expr = dynamic!int;
        auto primary = any(
            range!('0', '9').rep.map!(x => x.to!int),
            seq(tk!'(', expr, tk!')').map!(x => x[1])
        );
        auto term = dynamic!int;
        term = any(
            seq(primary, tk!'*', term).map!(x => x[0] * x[2]),
            seq(primary, tk!'/', term).map!(x => x[0] / x[2]),
            primary
        );
        expr = any(
            seq(term, tk!'+', expr).map!(x => x[0] + x[2]),
            seq(term, tk!'-', expr).map!(x => x[0] - x[2]),
            term
        );
        return expr;
    }
}
unittest {
        assert("2+4*(2+3)".parse(calc) == 22);
}
```

Perf Consideration

A subtle problem e.g. the following parser will call 'term' 3 times on expression "42"

```
expr = any(
            seq(term, tk!'+', expr).map!(x => x[0] + x[2]),
            seq(term, tk!'-', expr).map!(x => x[0] - x[2]),
            term
        );
```

Each of those in turn calls primary 3 times

```
term = any(
            seq(primary, tk!'*', term).map!(x => x[0] * x[2]),
            seq(primary, tk!'/', term).map!(x => x[0] / x[2]),
            primary
        );
```

In total 9 times parsing the simple digit string!

Something went better then expected

Solutions

Packrat approaches problem in its full generality

- memoize each recursive call and respective position in input
- each time instead of calling smth. check the cache

Truly academical achievement:

O(n) parsing but in O(n) space

In real world not a single hand-written parser does it

...yet they don't degrade to exponential behavior

They do unthinkable - they just don't repeat the same work twice if it's the same in each alternative

Merging prefixes

The idea is to detect the following pattern:

```
auto x = any(
    seq(Prefix, Suffix1).map!(...),
    seq(Prefix, Suffix2).map!(...),
    ...
    Prefix.map!(...)  // potentially the lone prefix on its own
);
```

Conceptually transform it into:

```
auto x = seq(Prefix, any(
        Suffix1,
        Suffix2,
        ...
        Epsilon // empty parser
    )
).map!(...);
```

Can't do it litterally like that due to how map contains arbitrary code

Takes a bit of meta-programming - needs those unique types

Performance

Simple arithmetic expressions (looong ones)

Kind	Time, ms	LOCs
Handwritten	57ms	92
Pry	67ms	23

JSON parsing ~33Kb of RPC-message

Kind	Time, us	LOCs
std.json	1098	326
stdx.data.json	688	~1600*
Pry	769	86

* cutting out multi-line comments and unittests, etc.

Going to Grammar

Taking a page from Pegged project here

```
unittest {
  mixin(grammar(`
    calc:
      expr : int <-
        (term '+' expr) { return it[0] + it[2]; }
        / (term '-' expr) { return it[0] - it[2]; }
        / term ;
      term : int <-
        (primary '*' term) { return it[0] * it[2]; }
        / (primary '/' term) { return it[0] / it[2]; }
        / primary ;
      primary <-
        [0-9]+ { return to!int(it); }
        / :'(' expr :')';
    `));
  assert(" ( 2 + 4) * 2".parse(calc) == 12);
}
```

1. Need to run full parser of PEG grammar at compile-time

2. Generate appropriate sequence of calls to combinators

Parsing the PEG

Need to build a compile-time parser at compile-time

Actually works!

Same combinators API utilized

~200LOCs with tests and such

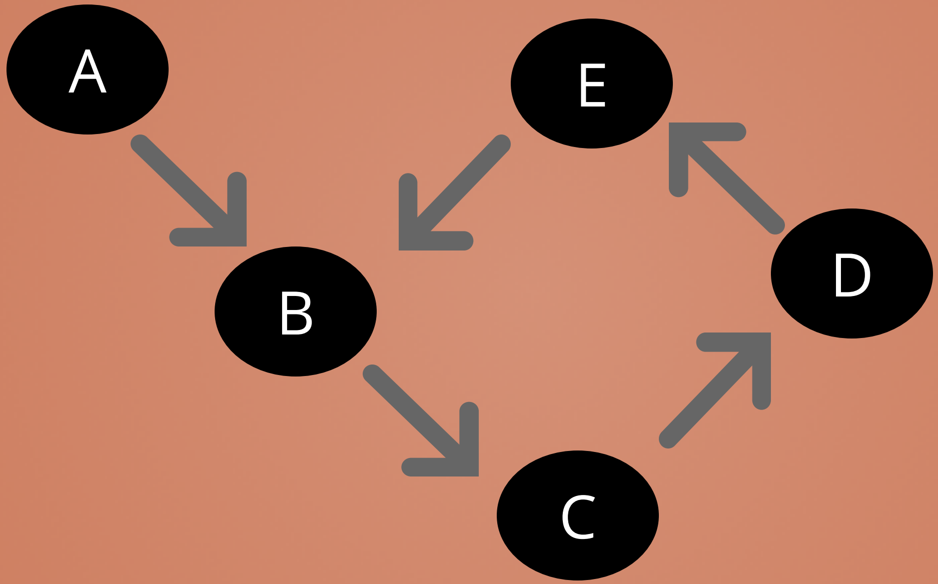
Regex character classes are reused from std.regex*

**Pull request is still hanging in the Q*

Produces AST that has to be processed at compile-time

Tracking dependencies

PEG rules basically form a directed graph of dependencies

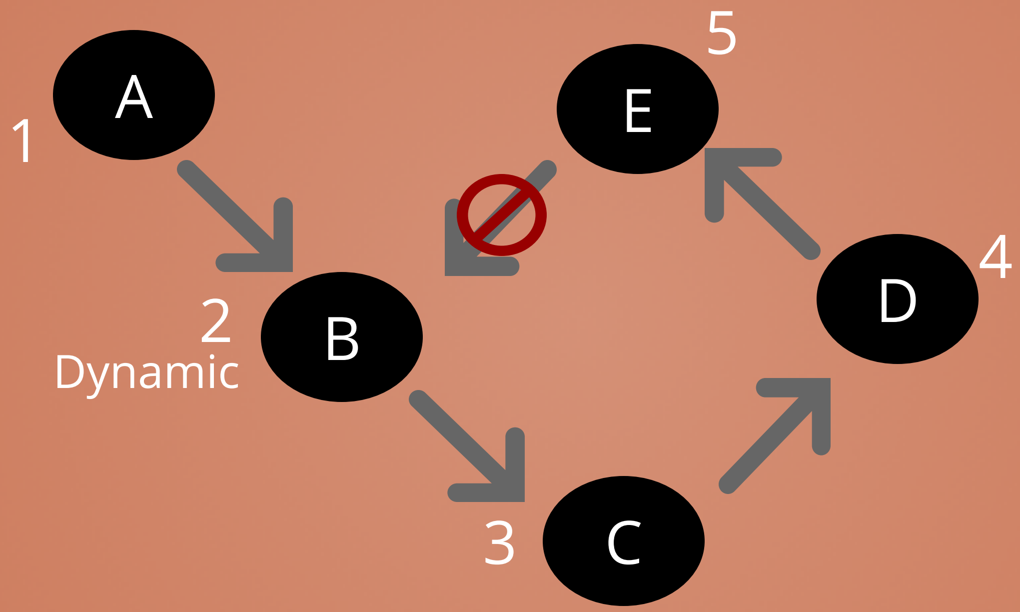


Need to establish an order of code generation

Some rules will have to be forward-referenced

Tracking dependencies #2

Do a topological sort at compile-time (simpler than it sounds)



Each detected cycle is broken, back edge will be dynamic

The order is in priority: 5-4-3-2-1

Following codegen is straight-forward

Open problems

- In case something goes wrong stack traces are unhelpful
  - Combinators tend to produce 10K+ bytes long symbols
- To skip whitespace or not to skip whitespace
  - Current approach needs more thought
- Left recursion is not detected nor supported
  - Will happily cause a stack overflow with horrible stack trace
- Error messages are not very helpful yet

Future directions

Document all things!

On the combinators API

- Want to support "parsing" binary formats in the same fashion
- Support allocators for "array" and "aa"

Grammar module is very early development, still need:

- Proper type-checking with user-friendly errors
- Detect left-recursion, supporting it(?)
- Provide **more** real-world examples!
- Ability to auto-generate sensible AST classes

That's it!

Stay pragmatic

and get involved on Github

https://github.com/DmitryOlshansky/pry