A SWIFTLY TILTING PARSER

in memory of Madeleine L'Engle

https://github.com/robrix/A-Swiftly-Tilting-Parser

rob.rix@github.com 🚀 @rob_rix

THE DERIVATIVE of PARSERS

- Might, Darais, & Spiewak's 2011 paper *Parsing with Derivatives—a Functional Pearl*
- Recognizes and parses context-free languages
    - Recognizing: "is my input valid?"
    - Parsing: "how is the input structured?"
- Validity and structure are defined by the grammar, which is made of parser combinators

PARSER COMBINATORS

(We'll use "parser" as a synonym)

- Executable LEGOs for parsing text
  - Each one is a tiny program
  - Some parse input directly
  - Some combine other parsers
- Put together, they match patterns in text

KINDS of PARSERS

- *Literal*: match a specific character
- **Alternation:** match *x* **or** *y*
- **Concatenation:** match *x* & **then** *y*
- **Repetition:** match *x* **zero or more** times
- **Reduction:** match *x* & **map** with a function
- *Null*: match the empty string; hold parse trees
- *Empty*: never ever match

TERMINAL PARSERS in OBJC

```
@interface HMRLiteral : HMRTerminal
+(instancetype)literal:(id)object;
@property (readonly) id object;
@end

@interface HMREmpty : HMRTerminal
@end

@interface HMRNull : HMRTerminal
+(instancetype)captureForest:(NSSet *)forest;
@property (readonly) NSSet *parseForest;
@end
```

NONTERMINAL PARSERS in OBJC

```
@interface HMRAlternation : HMRNonterminal
+(instancetype)alternateLeft:(HMRCombinator *)left right:(HMRCombinator *)right;
@property (readonly) HMRCombinator *left;
@property (readonly) HMRCombinator *right;
@end

@interface HMRConcatenation : HMRNonterminal
+(instancetype)concatenateFirst:(HMRCombinator *)first second:(HMRCombinator *)second;
@property (readonly) HMRCombinator *first;
@property (readonly) HMRCombinator *second;
@end

@interface HMRRepetition : HMRNonterminal
+(instancetype)repeat:(HMRCombinator *)combinator;
@property (readonly) HMRCombinator *combinator;
@end

@interface HMRReduction : HMRNonterminal
+(instancetype)reduce:(HMRCombinator *)combinator usingBlock:(HMRReductionBlock)block;
@property (readonly) HMRCombinator *combinator;
@property (readonly) HMRReductionBlock block;
@end
```

PARSERS in SWIFT

```swift
enum Language<Alphabet : Alphabet, Recur> {
  case Literal(Box<Alphabet>)

  case Alternation(Delay<Recur>, Delay<Recur>)
  case Concatenation(Delay<Recur>, Delay<Recur>)
  case Repetition(Delay<Recur>)
  case Reduction(Delay<Recur>, Alphabet -> Any)

  case Empty
  case Null(ParseTree<Alphabet>)
}
```

OPERATIONS

1. Parsing
2. Derivative
3. Nullability
4. Parse forest
5. Compaction

OPERATIONS

1. **Parsing**
2. Derivative
3. Nullability
4. Parse forest
5. Compaction

PARSING

- Go through the input character by character
- At each step, compute the derivative of the parser
- Compact it
- Use it for the next step
- Return the parsed input as a parse forest

PARSING in OBJC

```
NSSet *HMRParseCollection(HMRCombinator *parser, id sequence) {
  parser = [sequence reduce:parser combine:^(HMRCombinator *parser, id each) {
    return [parser derivative:each];
  }];
  return parser.parseForest;
}
```

PARSING in SWIFT

```
extension Combinator {
  func parse<S : Sequence where S.GeneratorType.Element == Alphabet>
    (sequence: S) -> ParseTree<Alphabet> {
    return reduce(sequence, self) { parser, term in
      derive(parser, term).compact()
    }.parseForest
  }
}
```

OPERATIONS

1. Parsing
2. **Derivative**
3. Nullability
4. Parse forest
5. Compaction

DERIVATIVE

- Returns the parser that would match *after* the current one
- Stores matched input in parse trees
- On failure, returns the empty parser
- Different definition for each kind of parser

TERMINAL DERIVATIVE in OBJC

```
// Literal
-(HMRCombinator *)derivative:(id)object {
  return [self evaluateWithObject:object]?
    [HMRCombinator captureTree:object]
  : [HMRCombinator empty];
}

// Null
-(HMRCombinator *)derivative:(id)object {
  return [HMRCombinator empty];
}

// Empty
-(HMRCombinator *)derivative:(id)object {
  return self;
}
```

NONTERMINAL DERIVATIVE in OBJC

```
// Alternation
-(HMRCombinator *)deriveWithRespectToObject:(id)object {
  return [[self.left derivative:object] or:[self.right derivative:object]];
}
// Reduction
-(HMRReduction *)deriveWithRespectToObject:(id)object {
  return [[self.combinator derivative:object] mapSet:self.block];
}
// Repetition
-(HMRCombinator *)deriveWithRespectToObject:(id)object {
  return [[self.combinator derivative:object] concat:self];
}
// Concatenation
-(HMRCombinator *)deriveWithRespectToObject:(id)object {
  return HMRCombinatorIsNullable(first)?
    [[[first derivative:object] concat:second]
      or:[[HMRCombinator capture:first.parseForest] concat:[second derivative:object]]]
  : [[first derivative:object] concat:second];
}
```

DERIVATIVE in SWIFT

```
func derive(c: Alphabet) -> Recur {
  switch self.language {
  case let .Literal(x) where x == c:
    return Combinator(parsed: ParseTree(leaf: c))

  case let .Alternation(x, y):
    return derive(x, c) | derive(y, c)

  case let .Reduction(x, f): return derive(x, c) --> f

  case let .Repetition(x): return derive(x, c) ++ self

  case let .Concatenation(x, y) where x.value.nullable:
    return
      derive(x, c) ++ y
    | Combinator(parsed: x.value.parseForest) ++ derive(y, c)
  case let .Concatenation(x, y): return derive(x, c) ++ y

  default: return Combinator(.Empty)
  }
}
```

RECURSION 🌀 & NONTERMINATION 🔄

- Context-free languages & grammars are recursive
- NB: Not just the *types*: the object graph is cyclic!
- Key to why you can’t parse arbitrary HTML with a regexp
- Regexps can be matched with a list, but context-free languages need a stack
- Naïve implementations will infinite loop 🔄

PROTECTING your PARSERS from NONTERMINATION 😎

1. **Laziness** 😴

LAZINESS 😴

- Alternations, concatenations, repetitions, & reductions use closures to delay evaluation
- Avoid nontermination in the derivative
- Necessary to even *construct* cyclic grammars!

LAZINESS 😴 in OBJC

```
@implementation HMRDelayCombinator

-(HMRCombinator *)forced {
    HMRCombinator *(^block)(void) = _block;
    _block = nil;
    if (block) _forced = block();
    return _forced;
}

-(NSString *)description {
    return [@"λ." stringByAppendingString:[self.forced description]];
}

-(id)forwardingTargetForSelector:(SEL)selector {
    return self.forced;
}

@end

#define HMRDelay(x) ((__typeof__(x))[HMRDelayCombinator delay:^{ return (x); }])
…
HMRDelay([self derivativeWithRespectToObject:c]);
```

LAZINESS 😴 in SWIFT

```swift
@final class Delay<T> {
    var _thunk: (() -> T)?

    @lazy var value: T = {
        let value = self._thunk!()
        self._thunk = nil
        return value
    }()

    init(_ thunk: () -> T) {
        _thunk = thunk
    }

    @conversion func __conversion() -> T {
        return value
    }
}
// Construct an alternation.
func | <Alphabet : Alphabet>
  (left: @auto_closure Void -> Combinator<Alphabet>,
  right: @auto_closure Void -> Combinator<Alphabet>) -> Combinator<Alphabet> {
    return Combinator(.Alternation(Delay(left), Delay(right)))
}
```

PROTECTING your PARSERS from NONTERMINATION 😎

1. Laziness 😴
2. **Memoization** 📎

MEMOIZATION 📎

- The first time you call a memoized function with a set of arguments, it stores the results
- The next time, it looks them up; memoize ≅ cache
- Can store results in a dictionary, ivar, etc.
- Allows the derivative to “tie the knot” when building a cyclic grammar *from* a cyclic grammar

MEMOIZATION 📎 in OBJC

```
#define HMRMemoize(x, start, body) ((x) ?: ((x = (start)), (x = (body))))

// HMRNonterminal.m
-(HMRCombinator *)derivative:(id<NSObject, NSCopying>)object {
  return HMRMemoize(_derivativesByElements[object],
    [HMRCombinator empty],
    [self deriveWithRespectToObject:object].compaction);
}
```

MEMOIZATION 📎 in SWIFT

```
func derive(c: Alphabet) -> Recur {
  let derive: (Recur, Alphabet) -> Recur = memoize { recur, parameters in
    let (combinator, c) = parameters
    switch combinator.language {
    case let .Literal(x) where x == c:
      return Combinator(parsed: ParseTree(leaf: c))

    case let .Alternation(x, y):
      return recur(x, c) | recur(y, c)

      …
    }
  }
  return derive(self, c)
}
```

OPERATIONS

1. Parsing
2. Derivative
3. **Nullability**
4. Parse forest
5. Compaction

NULLABILITY

- “Is this grammar nullable?” = “Will it match an empty string?”
- Equivalent: “Can it match at the end of the input?”
- Equivalent: “Can it be skipped?”

NULLABILITY in OBJC

```
bool HMRCombinatorIsNullable(HMRCombinator *combinator) {
  return [HMRMemoize(cache[combinator], @NO, HMRMatch(combinator, @[
    [[[HMRBind() concat:HMRBind()] quote] then:^(HMRCombinator *fst, HMRCombinator *snd) {
      return @(recur(fst) && recur(snd));
    }],

    [[[HMRBind() or:HMRBind()] quote] then:^(HMRCombinator *left, HMRCombinator *right) {
      return @(recur(left) || recur(right));
    }],

    [[[HMRBind() map:REDIdentityMapBlock] quote] then:^(HMRCombinator *combinator) {
      return @(recur(combinator));
    }],

    [[[HMRAny() repeat] quote] then:^{ return @YES; }],
    [[HMRNull quote] then:^{ return @YES; }],
  ])) boolValue];
}
```

NULLABILITY in SWIFT

```
var nullable: Bool {
  let nullable: Combinator<Alphabet> -> Bool = memoize { recur, combinator in
    switch combinator.language {
    case .Null: return true

    case let .Alternation(left, right):
      return recur(left) || recur(right)

    case let .Concatenation(first, second):
      return recur(first) && recur(second)

    case .Repetition: return true

    case let .Reduction(c, _): return recur(c)

    default: return false
    }
  }
  return nullable(self)
}
```

NULLABILITY and NONTERMINATION 🔄

- Nullability walks the grammar eagerly, defeating laziness 😴
- Nullability computes pass/fail, not a structure, defeating memoization 📎
- Thus: 🔄

PROTECTING your PARSERS from NONTERMINATION 😎

1. Laziness 😴

2. Memoization 📎

3.

PROTECTING your PARSERS from NONTERMINATION 😎

1. Laziness 😴
2. Memoization 📎
3. *Math* ✖️ ➗

PROTECTING your PARSERS from NONTERMINATION 😎

1. Laziness 😴
2. Memoization 📎
3. ~~*Math* ✖️ ➗~~

PROTECTING your PARSERS from NONTERMINATION 😎

1. Laziness 😴
2. Memoization 📎
3. ~~*Math* ✖️➗~~ **Fixed points** 🔨☝️

~~MATH ✖️➗~~ FIXED POINTS 🔨☝️

- If $f(x) = x$, f is fixed at x; x^2 is fixed at `0` and `1`
- If L is nullable, $\delta(L)$ is null, otherwise empty
- Any fixpoints of δ are likewise either null or empty
- Interpret $\delta(L) = \delta(L)\ \alpha \mid \epsilon$ as a fixpoint of δ
- Iterate $\delta^n(L)$ from $\delta^0(L) =$ `false` until $\delta^n(L) = \delta^{n-1}(L)$ (Kleene fixpoint theorem)

~~FIXPOINTS 🔨~~ ☝️ in OBJC

```
bool HMRCombinatorIsNullable(HMRCombinator *combinator) {
  return [HMRMemoize(cache[combinator], @NO, HMRMatch(combinator, @[
    [[[HMRBind() concat:HMRBind()] quote] then:^(HMRCombinator *fst, HMRCombinator *snd) {
      return @(recur(fst) && recur(snd));
    }],

    [[[HMRBind() or:HMRBind()] quote] then:^(HMRCombinator *left, HMRCombinator *right) {
      return @(recur(left) || recur(right));
    }],

    [[[HMRBind() map:REDIdentityMapBlock] quote] then:^(HMRCombinator *combinator) {
      return @(recur(combinator));
    }],

    [[[HMRAny() repeat] quote] then:^{ return @YES; }],
    [[HMRNull quote] then:^{ return @YES; }],
  ])) boolValue];
}
```

FIXPOINTS 🔨☝️ in OBJC

```
bool HMRCombinatorIsNullable(HMRCombinator *combinator) {
  NSMutableDictionary *cache = [NSMutableDictionary new];
  bool (^__weak __block recur)(HMRCombinator *);
  bool (^isNullable)(HMRCombinator *) = ^bool (HMRCombinator *combinator) {
    return [HMRMemoize(cache[combinator], @NO, HMRMatch(combinator, @[
      [[[HMRBind() concat:HMRBind()] quote] then:^(HMRCombinator *fst, HMRCombinator *snd) {
        return @(recur(fst) && recur(snd));
      }],

      [[[HMRBind() or:HMRBind()] quote] then:^(HMRCombinator *left, HMRCombinator *right) {
        return @(recur(left) || recur(right));
      }],

      [[[HMRBind() map:REDIdentityMapBlock] quote] then:^(HMRCombinator *combinator) {
        return @(recur(combinator));
      }],

      [[[HMRAny() repeat] quote] then:^{ return @YES; }],
      [[HMRNull quote] then:^{ return @YES; }],
    ])) boolValue];
  };
  recur = isNullable;
  return isNullable(combinator);
}
```

~~FIXPOINTS 🔨☝~~ in SWIFT

```swift
var nullable: Bool {
  let nullable: Combinator<Alphabet> -> Bool = memoize { recur, combinator in
    switch combinator.language {
    case .Null: return true

    case let .Alternation(left, right):
      return recur(left) || recur(right)

    case let .Concatenation(first, second):
      return recur(first) && recur(second)

    case .Repetition: return true

    case let .Reduction(c, _): return recur(c)

    default: return false
    }
  }
  return nullable(self)
}
```

FIXPOINTS 🔨☝️ in SWIFT

```swift
var nullable: Bool {
  let nullable: Combinator<Alphabet> -> Bool = fixpoint(false) { recur, combinator in
    switch combinator.language {
    case .Null: return true

    case let .Alternation(left, right):
      return recur(left) || recur(right)

    case let .Concatenation(first, second):
      return recur(first) && recur(second)

    case .Repetition: return true

    case let .Reduction(c, _): return recur(c)

    default: return false
    }
  }
  return nullable(self)
}
```

OPERATIONS

1. Parsing
2. Derivative
3. Nullability
4. **Parse forest**
5. Compaction

PARSE FOREST

- Construct and return any matched parse trees
- Apply reductions
  - This is how you construct *your* objects
- If > 1 parser matched the input, > 1 parse tree in the parse forest
  - This means there's ambiguity in the grammar 😨

PARSE FOREST in OBJC

```
-(NSSet *)parseForest {
  return cache[combinator] = HMRMatch(combinator, @[
    [[[HMRBind() or:HMRBind()] quote] then:^(HMRCombinator *left, HMRCombinator *right) {
      return [parseForest(left, cache) setByAddingObjectsFromSet:parseForest(right, cache)];
    }],

    [[[HMRBind() concat:HMRBind()] quote] then:^(HMRCombinator *fst, HMRCombinator *snd) {
      return [parseForest(fst, cache) product:parseForest(snd, cache)];
    }],

    [[[HMRBind() map:REDIdentityMapBlock] quote]
       then:^(HMRCombinator *c, HMRReductionBlock f) {
      return [[NSSet set] f(parseForest(c, cache))];
    }],

    [[[HMRAny() repeat] quote] then:^{
      return [NSSet setWithObject:[HMRPair null]];
    }],
    [[HMRNull quote] then:^{
      return combinator.parseForest;
    }],
  ]));
}
```

PARSE FOREST in SWIFT

```
var parseForest: ParseTree<Alphabet> {
  let parseForest: Combinator<Alphabet> -> ParseTree<Alphabet> =
      fixpoint(ParseTree.Nil) { recur, combinator in
    switch combinator.language {
    case let .Null(x): return x

    case let .Alternation(x, y): return recur(x) + recur(y)

    case let .Concatenation(x, y): return recur(x) * recur(y)

    case let .Repetition(x): return .Nil

    case let .Reduction(x, f): return map(recur(x), f)

    default: return .Nil
    }
  }
  return parseForest(self)
}
```

OPERATIONS

1. Parsing
2. Derivative
3. Nullability
4. Parse forest
5. **Compaction**

WITHOUT COMPACTION

"The implementation is brief. The code is pure. The theory is elegant. So, how does this perform in practice?"

WITHOUT COMPACTION

"The implementation is brief. The code is pure. The theory is elegant. So, how does this perform in practice? In brief, it is awful."

WITHOUT COMPACTION

"The implementation is brief. The code is pure. The theory is elegant. So, how does this perform in practice? In brief, it is awful."

- Derivative of concatenation *doubles* grammar size
- Worst case: $O(2^{2n}G^2)$: *G* = grammar size, *n* = input length 💥

COMPACTION

- Replace complex parsers with simpler equivalents
- Enables better performance
  - Worst case still terrible
  - Expected case (unambiguous grammars) is O(nG)
  - Quite reasonable in practice; *no* algorithm is fast under ambiguity

COMPACTION in OBJC

```
// HMRAlternation
-(HMRCombinator *)compact {
  HMRCombinator *left = self.left.compacted, *right = self.right.compacted;
  if ([left isEqual:[HMRCombinator empty]]) return right;
  else if ([right isEqual:[HMRCombinator empty]]) return left;

  else if ([left isKindOfClass:[HMRNull class]]
    && [right isKindOfClass:[HMRNull class]]) {
    NSSet *all = [left.parseForest setByAddingObjectsFromSet:right.parseForest];
    return [HMRCombinator capture:all];
  }
  else if ([left isKindOfClass:[HMRConcatenation class]]
    && [left.first isKindOfClass:[HMRNull class]]
    && [right isKindOfClass:[HMRConcatenation class]]
    && [left.first isEqual:right.first]) {
    HMRCombinator *innerLeft = left.second;
    HMRCombinator *innerRight = right.second;
    alternation = [innerLeft or:innerRight];
    return [left.first concat:[innerLeft or:innerRight]];
  }
  else return [left or:right];
}
```

COMPACTION in OBJC

```
// HMRConcatenation
-(HMRCombinator *)compact {
  HMRCombinator *fst = self.first.compaction, *snd = self.second.compaction;
  if ([fst isEqual:[HMRCombinator empty]] || [snd isEqual:[HMRCombinator empty]])
    return [HMRCombinator empty];
  else if ([fst isKindOfClass:[HMRNull class]] && [snd isKindOfClass:[HMRNull class]])
    return [HMRCombinator capture:[fst.parseForest product:snd.parseForest]];
  else if ([fst isKindOfClass:[HMRNull class]]) {
    NSSet *parseForest = fst.parseForest;
    if (parseForest.count == 0) return snd;
    else return [snd map:^(id each) {
        return HMRCons(parseForest.anyObject, each);
      }];
  }
  else if ([snd isKindOfClass:[HMRNull class]]) {
    NSSet *parseForest = snd.parseForest;
    if (parseForest.count == 0) concatenation = fst;
    else return [fst map:^(id each) {
        return HMRCons(each, parseForest.anyObject);
      }];
  }
  else return [fst concat:snd];
}
```

COMPACTION in OBJC

```
-(HMRCombinator *)compact {
  HMRCombinator *combinator = self.combinator.compaction;
  return [combinator isEqual:[HMRCombinator empty]]?
    [HMRCombinator captureTree:[HMRPair null]]
  : (combinator == self.combinator? self : [combinator repeat]);
}
```

COMPACTION in OBJC

```
// HMRReduction
-(HMRCombinator *)compact {
  HMRCombinator *combinator = self.combinator.compaction;
  if ([combinator isEqual:[HMRCombinator empty]])
    return [HMRCombinator empty];

  else if ([combinator isKindOfClass:[HMRReduction class]])
    return HMRComposeReduction(combinator, self.block);

  else if ([combinator isKindOfClass:[HMRNull class]])
    return [HMRCombinator capture:[self map:combinator.parseForest]];

  else return [combinator mapSet:self.block];
}
```

COMPACTION in SWIFT

```swift
func compact() -> Combinator<Alphabet> {
  let compact: Recur -> Recur = fixpoint(self) { recur, combinator in
    switch combinator.destructure(recur) {
    /// Alternations with Empty are equivalent to the other alternative.
    case let .Alternation(x, .Empty): return Combinator(x)
    case let .Alternation(.Empty, y): return Combinator(y)

    /// Concatenations with Empty are equivalent to Empty.
    case .Concatenation(.Empty, _), .Concatenation(_, .Empty):
      return Combinator.empty

    /// Repetitions of empty are equivalent to parsing the empty string.
    case .Repetition(.Empty): return Combinator(parsed: .Nil)
    case let .Repetition(x): return Combinator(x)*

    /// Reductions of reductions compose.
    case let .Reduction(.Reduction(x, f), g): return Combinator(x --> compose(g, f))

    default: return combinator
    }
  }
  return compact(self)
```

COMPACTION in the FUTURE 🚀

- Generally must compact after derivative, or else 🔄
  - Can we avoid complex parsers altogether in some cases?
- Enables better features
  - Incremental results: 🔢 vs. 1️⃣…2️⃣…3️⃣…4️⃣…
  - (Good) error reporting?
  - Disambiguation? ✨

CHALLENGES in OBJC & SWIFT

- Understanding the paper is hard 😖
- ObjC & Swift are reference counted
  - Cyclic grammars = refcycles (unless handled specially)
  - Potential solution: a refcycle-breaking combinator
- Pattern matching cyclic grammars is tricky

CHALLENGES UNIQUE to OBJC

- Language/algorithm impedance mismatch
- Verbose; dense; splits functions across many files
- Pattern matching cyclic grammars is *really* tricky
  - The language doesn’t have pattern matching 😭
  - Implemented pattern matching for parsers *using* parsers 💫🌀
- Nontermination is much harder to solve, e.g. `-isEqual:` for equal cyclic grammars

CHALLENGES UNIQUE to SWIFT

- Beta (& evolving!) compiler & IDE 😱
  - No codegen yet for some features
  - Crash-happy 😂💥 (as of Xcode 6b3)
  - Bad error reporting (ProTip™: extract nested expressions into constants to isolate issues)
- Some language design/prioritization choices need workarounds
  - Making it up as I go ✈️💺👖

BENEFITS of SWIFT vs. OBJC

- Much better tool:job match
    - enum is a better fit than classes for parsers 👍
    - Pattern matching 😍
    - Operator overloading for constructing parsers ✨
- Stronger typing → safer, better program 💪
- Solve *my* problems more, incidental ones less 🙌
- Make mistakes faster & with greater confidence 🎢

BENEFITS of OBJC vs. SWIFT

- ObjC is stable
- `clang` is stable
- Familiarity
- Unlikely to break my code on the day of the talk 😆

SUBTLETIES: OBJC > SWIFT…?

- It was initially hard describing parse trees' type in Swift
  - ObjC: sets, pairs, input, & AST are all `id`
  - However, easy ≠ *good*: 💥
- Can use macros & dynamic proxies in ObjC
  - No real equivalents in Swift 😕
  - *Had* to use macros & dynamic proxies in ObjC 😞

SUBTLETIES: SWIFT > OBJC…?

- *Much* more readable with `enum`/pattern matching
  - Wasn't sure this approach would work 1w ago 😰
  - If not, same solution as ObjC, with beta tools! 😡
- `@auto_closure` & operators are ✨ for grammars
  - Potentially masks refcycles
  - Hard to break cycles automatically *or* manually

SWIFT ❤️

- Objective-C is the wrong tool for the job
- Much more sound theoretically
  - Inheritance is holding us back
  - Better type system → more flexibility, less effort
- Much more sound practically
  - Safer & more productive
  - Types enable better optimizations → *fast!*

¿Q&A!

THANK YOU! 🙇

David Darais, Kris Markel, Matt Might,
Kelly Rix, David Smith, Daniel Spiewak,
the Swift team, @DecksetApp,
& especially you 💟

https://github.com/robrix/A-Swiftly-Tilting-Parser
rob.rix@github.com 🚀 @rob_rix