PARSER PARSER COMBINATORS FOR PROGRAM TRANSFORMATION

Rijnard van Tonder

@rvtond

Carnegie Mellon University

sourcegraph

About Me

Automated

Program {Transformation, Analysis, Repair}

Carnegie
Mellon
University

sourcegraph

About Me

Automated

Program {Transformation, Analysis, Repair}

Carnegie Mellon University

sourcegraph

About Me

Automated

Program {Transformation, Analysis, Repair}

Code {Search, Intelligence, Review}

Carnegie Mellon University

sourcegraph

About Me

I <3 developer tools

My favorite color is OCaml

About You
(and this talk)

Code changes.

About You
(and this talk)

Code changes.

All the time.

About You
(and this talk)

HERACLITVS.

change is the only constant

https://commons.wikimedia.org/wiki/File:Heraclitus_in_Thomas_Stanley_History_of_Philosophy.jpg

About You (and this talk)

HERACLITVS.

code
change is the only constant
^

https://commons.wikimedia.org/wiki/File:Heraclitus_in_Thomas_Stanley_History_of_Philosophy.jpg

See the link for demo video

https://drive.google.com/open?id=1ziCXghgdNwheDCnLy1ml5us8fGBzW3CU

Have this for any language

Have this for $\boxed{\text{any}}$ language

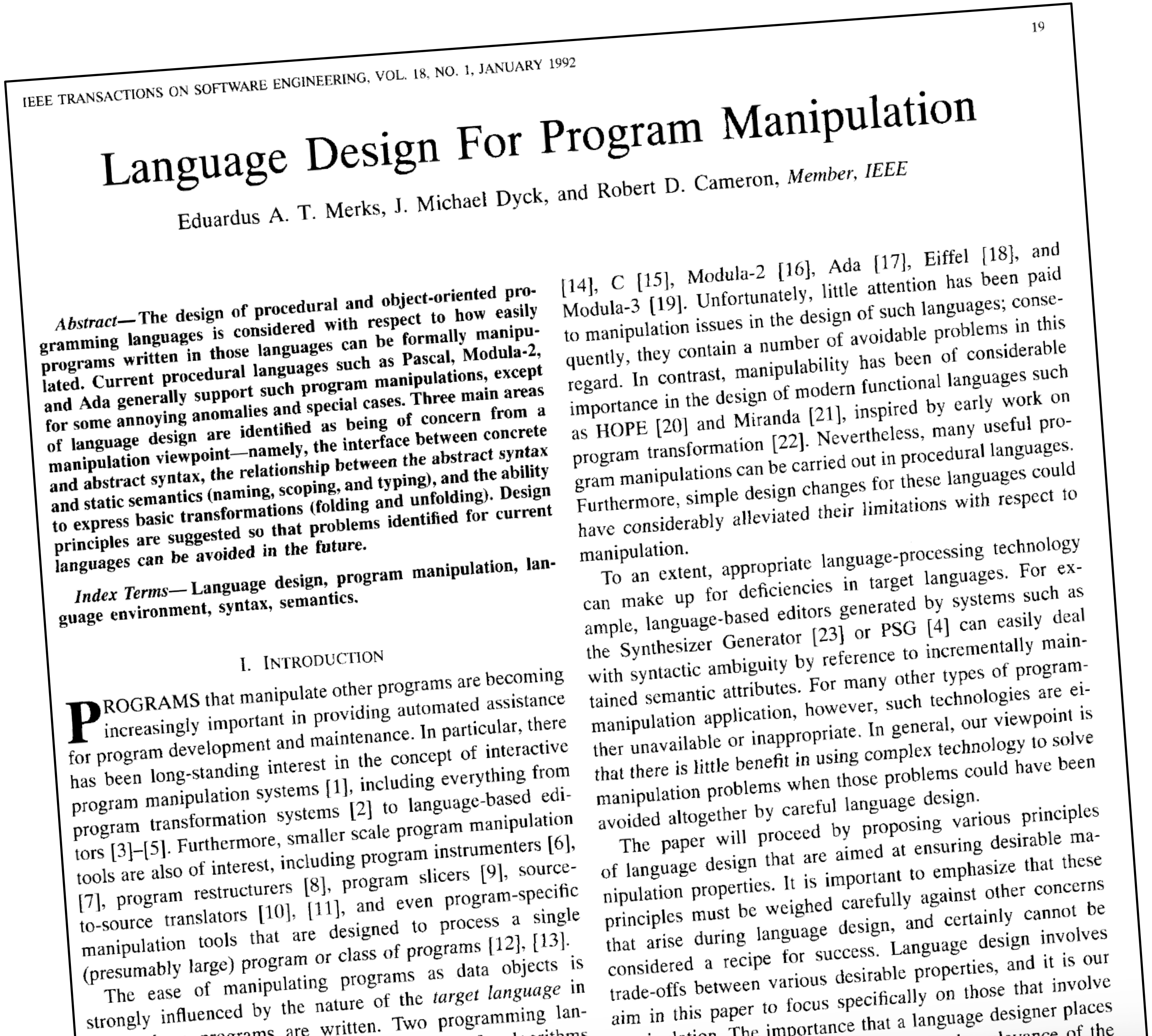
What I’d like you to take away from this talk:

A new way to change code

1. A deeper understanding of program syntax, manipulation, and challenges

2. My solution for manipulating syntax
   - Exposure to neat ideas in a functional paradigm

3. Practical tooling & demos



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 18, NO. 1, JANUARY 1992 19

Language Design For Program Manipulation

Eduardus A. T. Merks, J. Michael Dyck, and Robert D. Cameron, *Member, IEEE*

***Abstract*— The design of procedural and object-oriented programming languages is considered with respect to how easily programs written in those languages can be formally manipulated. Current procedural languages such as Pascal, Modula-2, and Ada generally support such program manipulations, except for some annoying anomalies and special cases. Three main areas of language design are identified as being of concern from a manipulation viewpoint—namely, the interface between concrete and abstract syntax, the relationship between the abstract syntax and static semantics (naming, scoping, and typing), and the ability to express basic transformations (folding and unfolding). Design principles are suggested so that problems identified for current languages can be avoided in the future.**

***Index Terms*— Language design, program manipulation, language environment, syntax, semantics.**

I. Introduction

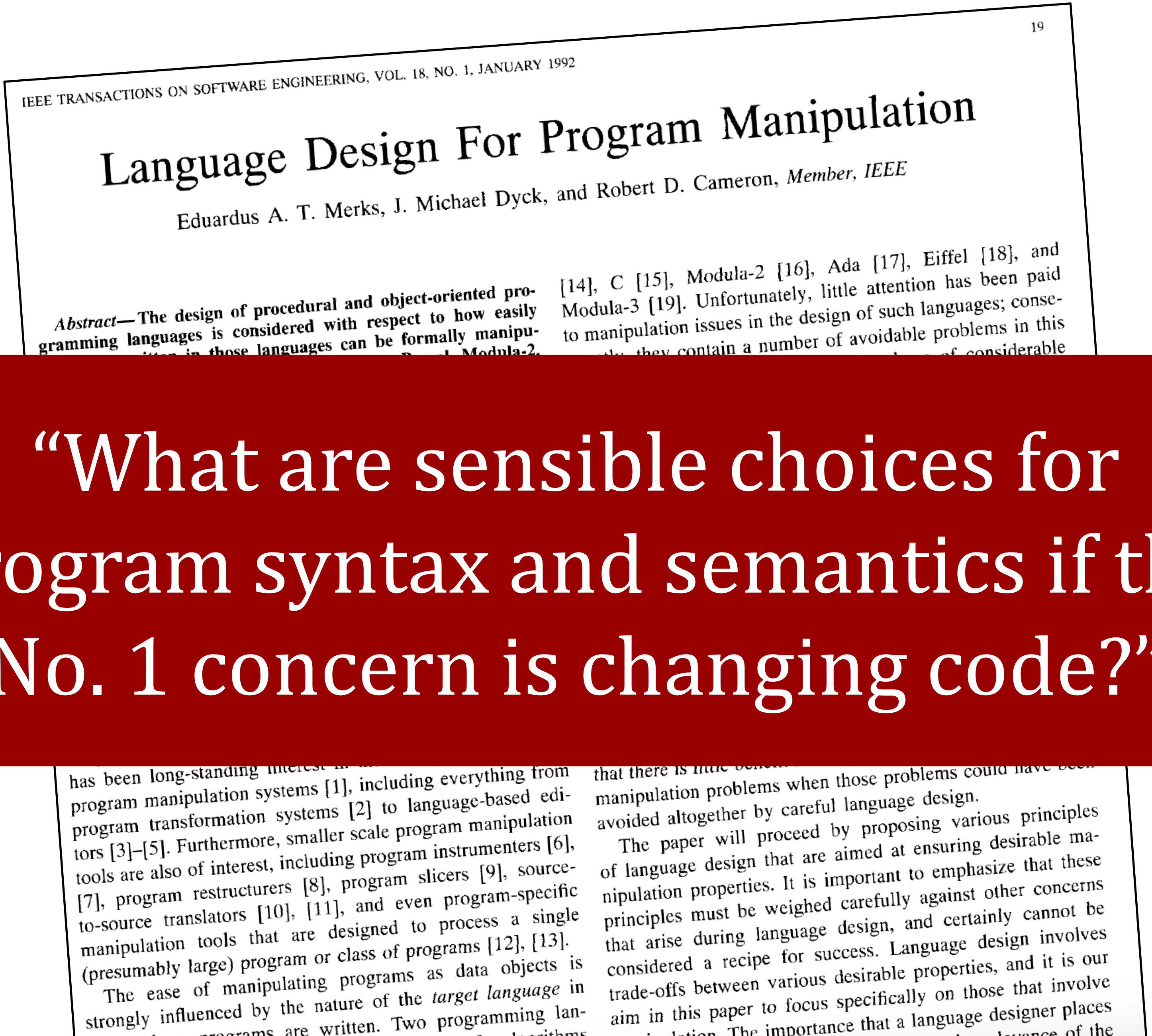
PROGRAMS that manipulate other programs are becoming increasingly important in providing automated assistance for program development and maintenance. In particular, there has been long-standing interest in the concept of interactive program manipulation systems [1], including everything from program transformation systems [2] to language-based editors [3]–[5]. Furthermore, smaller scale program manipulation tools are also of interest, including program instrumenters [6], [7], program restructurers [8], program slicers [9], source-to-source translators [10], [11], and even program-specific manipulation tools that are designed to process a single (presumably large) program or class of programs [12], [13].

The ease of manipulating programs as data objects is strongly influenced by the nature of the *target language* in which the programs are written. Two programming lan-

[14], C [15], Modula-2 [16], Ada [17], Eiffel [18], and Modula-3 [19]. Unfortunately, little attention has been paid to manipulation issues in the design of such languages; consequently, they contain a number of avoidable problems in this regard. In contrast, manipulability has been of considerable importance in the design of modern functional languages such as HOPE [20] and Miranda [21], inspired by early work on program transformation [22]. Nevertheless, many useful program manipulations can be carried out in procedural languages. Furthermore, simple design changes for these languages could have considerably alleviated their limitations with respect to manipulation.

To an extent, appropriate language-processing technology can make up for deficiencies in target languages. For example, language-based editors generated by systems such as the Synthesizer Generator [23] or PSG [4] can easily deal with syntactic ambiguity by reference to incrementally maintained semantic attributes. For many other types of program-manipulation application, however, such technologies are either unavailable or inappropriate. In general, our viewpoint is that there is little benefit in using complex technology to solve manipulation problems when those problems could have been avoided altogether by careful language design.

The paper will proceed by proposing various principles of language design that are aimed at ensuring desirable manipulation properties. It is important to emphasize that these principles must be weighed carefully against other concerns that arise during language design, and certainly cannot be considered a recipe for success. Language design involves trade-offs between various desirable properties, and it is our aim in this paper to focus specifically on those that involve manipulation. The importance that a language designer places



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 18, NO. 1, JANUARY 1992

Language Design For Program Manipulation

Eduardus A. T. Merks, J. Michael Dyck, and Robert D. Cameron, *Member, IEEE*

***Abstract*—The design of procedural and object-oriented programming languages is considered with respect to how easily programs written in those languages can be formally manipulated. ... Pascal, Modula-2,**

has been long-standing interest in program manipulation systems [1], including everything from program transformation systems [2] to language-based editors [3]–[5]. Furthermore, smaller scale program manipulation tools are also of interest, including program instrumenters [6], [7], program restructurers [8], program slicers [9], source-to-source translators [10], [11], and even program-specific manipulation tools that are designed to process a single (presumably large) program or class of programs [12], [13].

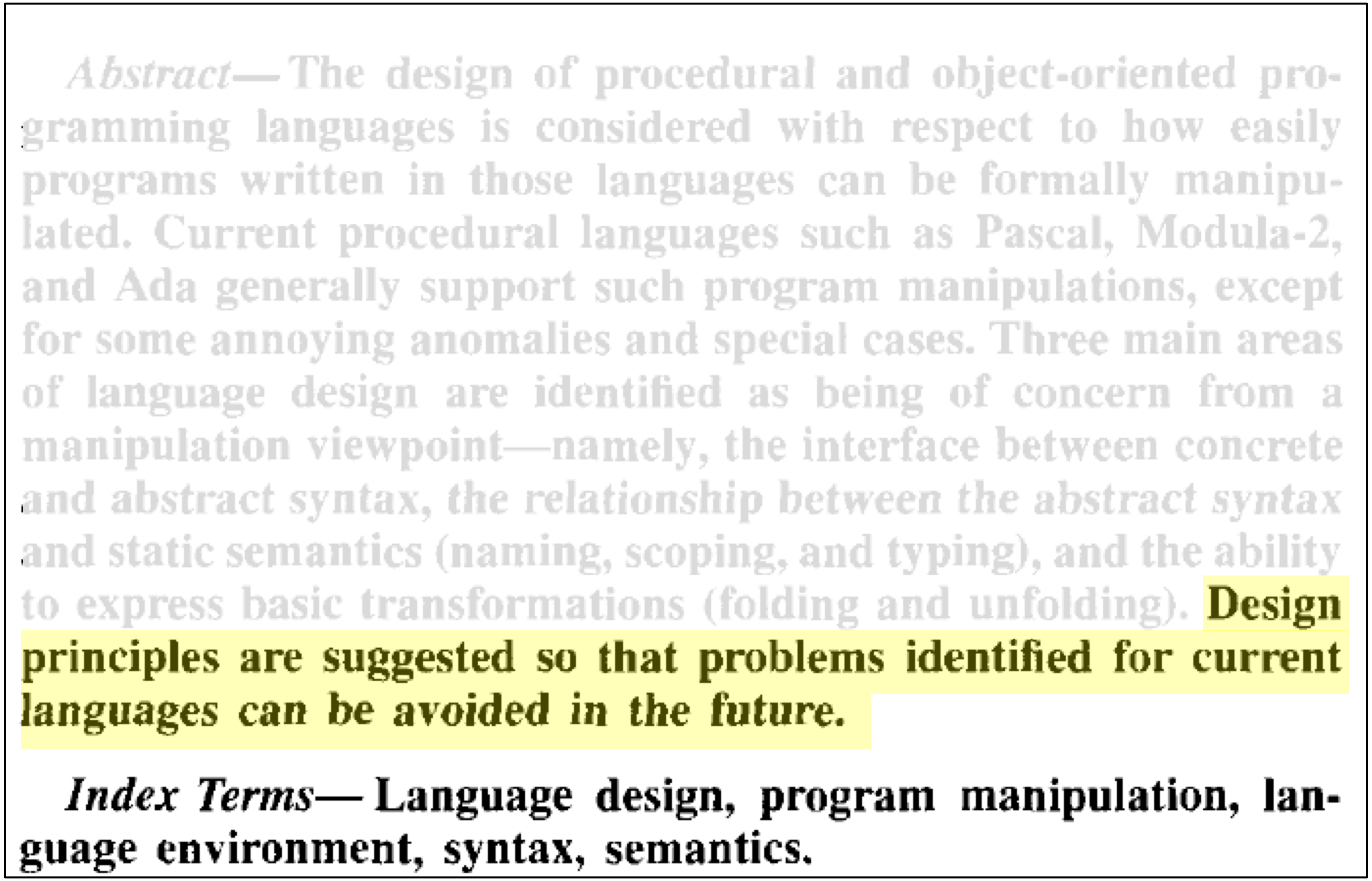
The ease of manipulating programs as data objects is strongly influenced by the nature of the *target language* in which programs are written. Two programming lan-

[14], C [15], Modula-2 [16], Ada [17], Eiffel [18], and Modula-3 [19]. Unfortunately, little attention has been paid to manipulation issues in the design of such languages; consequently, they contain a number of avoidable problems in this of considerable

that there is little benefit manipulation problems when those problems could have been avoided altogether by careful language design.

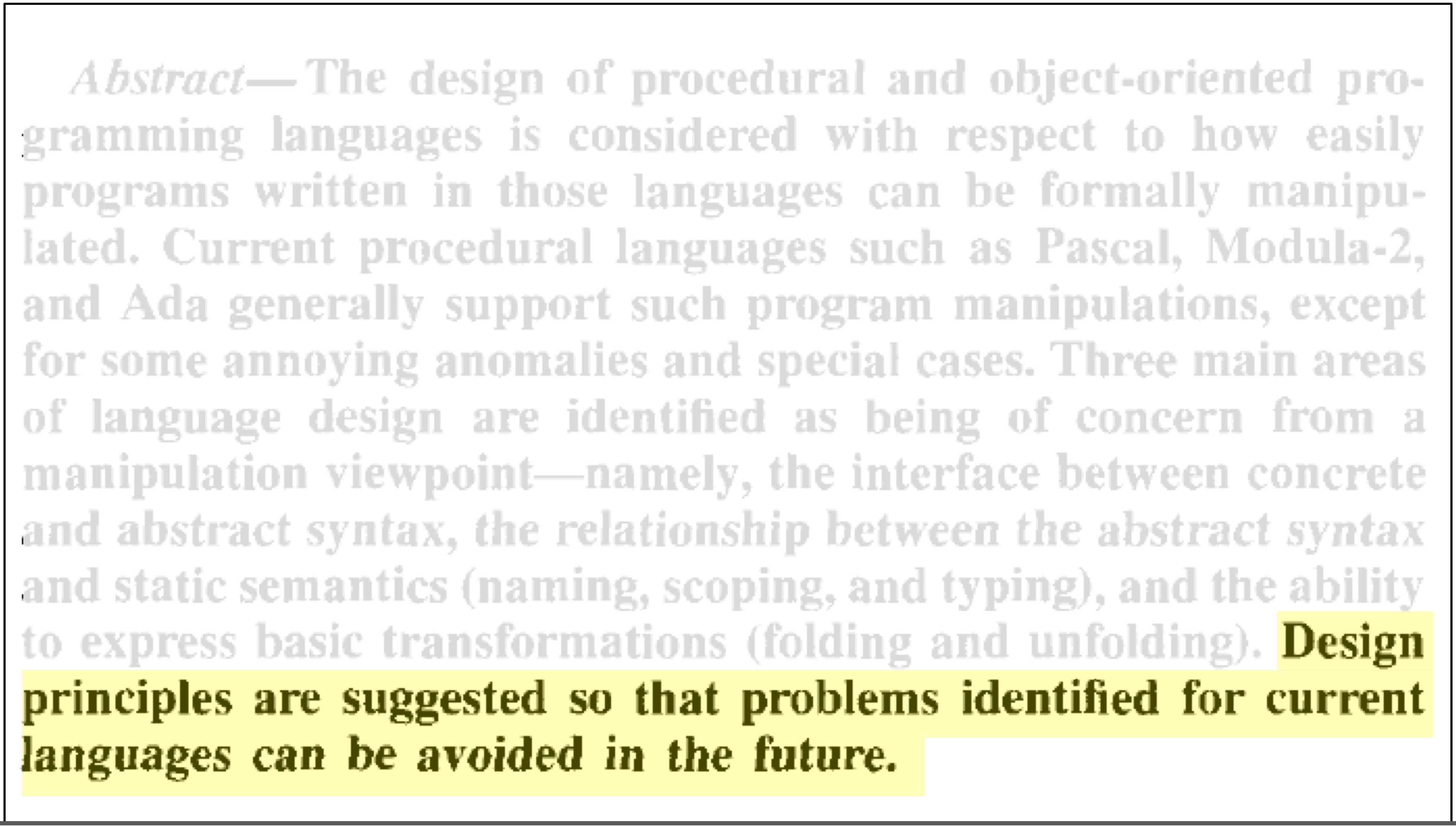
The paper will proceed by proposing various principles of language design that are aimed at ensuring desirable manipulation properties. It is important to emphasize that these principles must be weighed carefully against other concerns that arise during language design, and certainly cannot be considered a recipe for success. Language design involves trade-offs between various desirable properties, and it is our aim in this paper to focus specifically on those that involve manipulation. The importance that a language designer places

“What are sensible choices for program syntax and semantics if the No. 1 concern is changing code?”



Abstract—The design of procedural and object-oriented programming languages is considered with respect to how easily programs written in those languages can be formally manipulated. Current procedural languages such as Pascal, Modula-2, and Ada generally support such program manipulations, except for some annoying anomalies and special cases. Three main areas of language design are identified as being of concern from a manipulation viewpoint—namely, the interface between concrete and abstract syntax, the relationship between the abstract syntax and static semantics (naming, scoping, and typing), and the ability to express basic transformations (folding and unfolding). Design principles are suggested so that problems identified for current languages can be avoided in the future.

Index Terms—Language design, program manipulation, language environment, syntax, semantics.

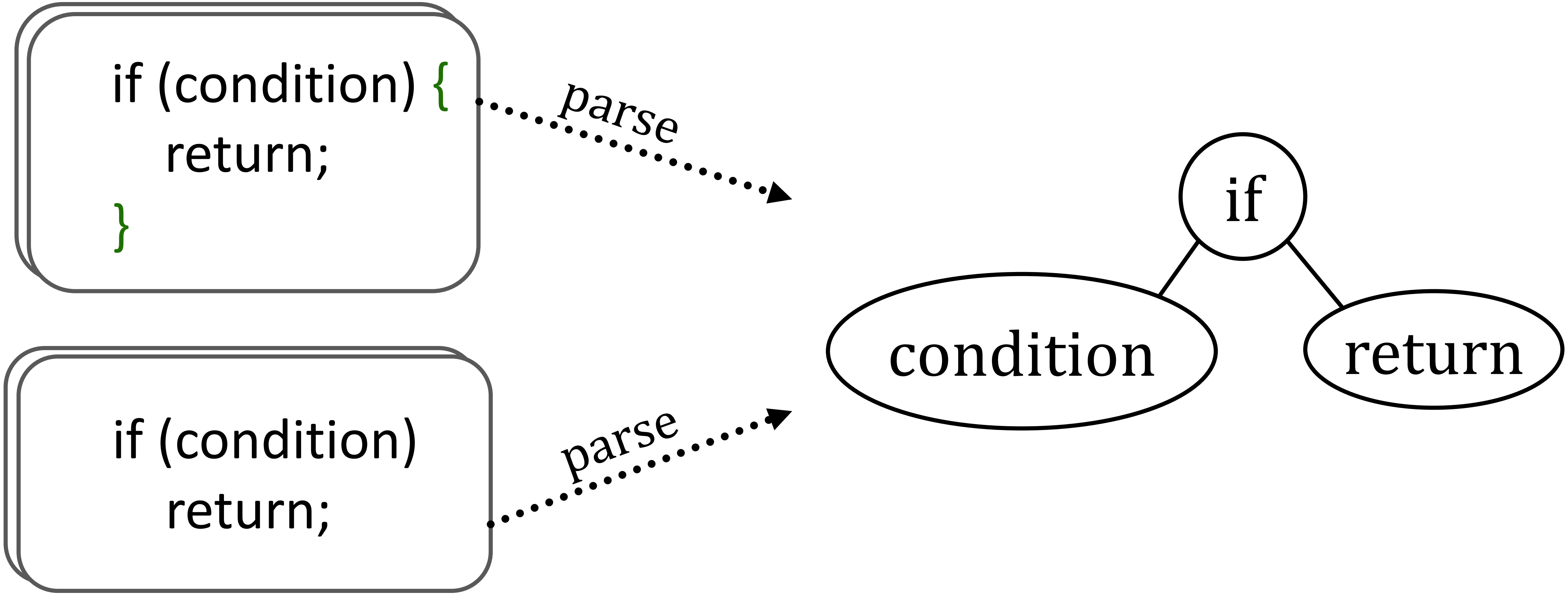


Abstract—The design of procedural and object-oriented programming languages is considered with respect to how easily programs written in those languages can be formally manipulated. Current procedural languages such as Pascal, Modula-2, and Ada generally support such program manipulations, except for some annoying anomalies and special cases. Three main areas of language design are identified as being of concern from a manipulation viewpoint—namely, the interface between concrete and abstract syntax, the relationship between the abstract syntax and static semantics (naming, scoping, and typing), and the ability to express basic transformations (folding and unfolding). **Design principles are suggested so that problems identified for current languages can be avoided in the future.**

Narrator: they were not avoided

Just one example:

concrete syntax ⟷ diverges ⟷ abstract syntax tree

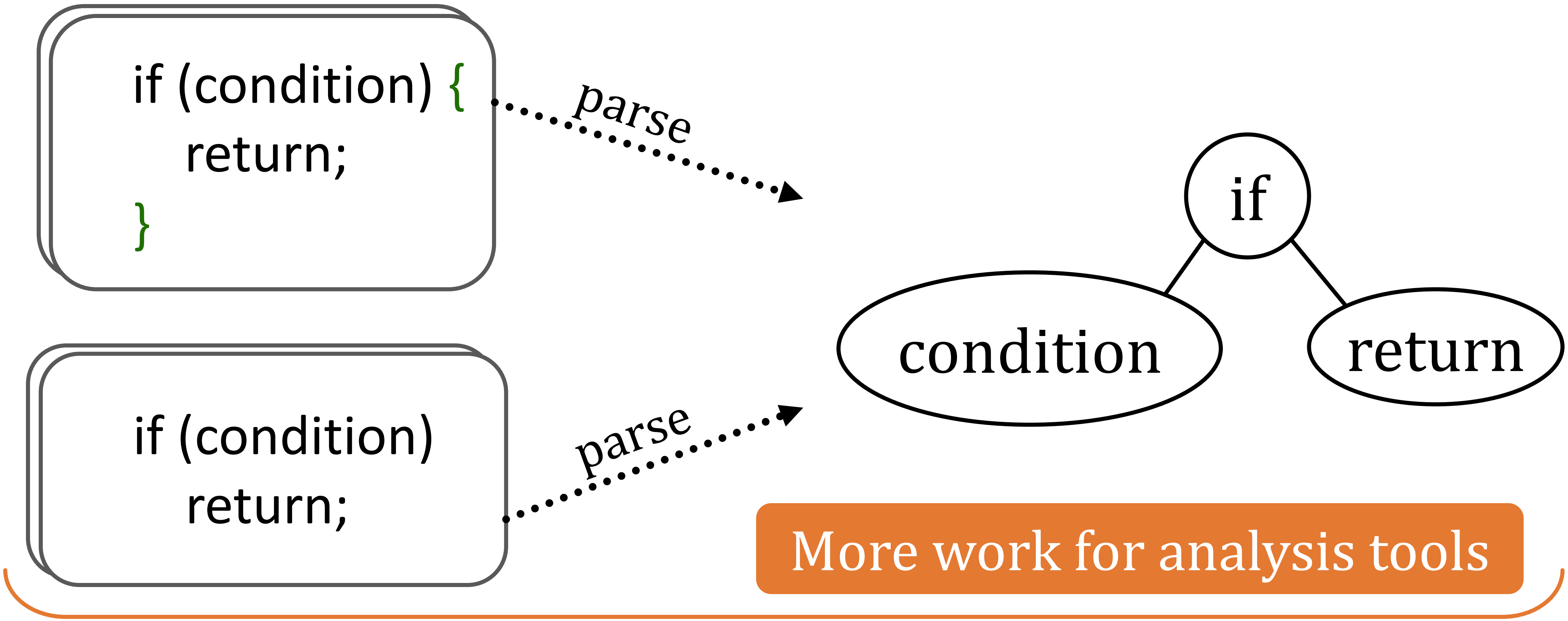


```
if (condition) {
    return;
}
```

```
if (condition)
    return;
```

Just one example:

concrete syntax ⟷ diverges ⟷ abstract syntax tree



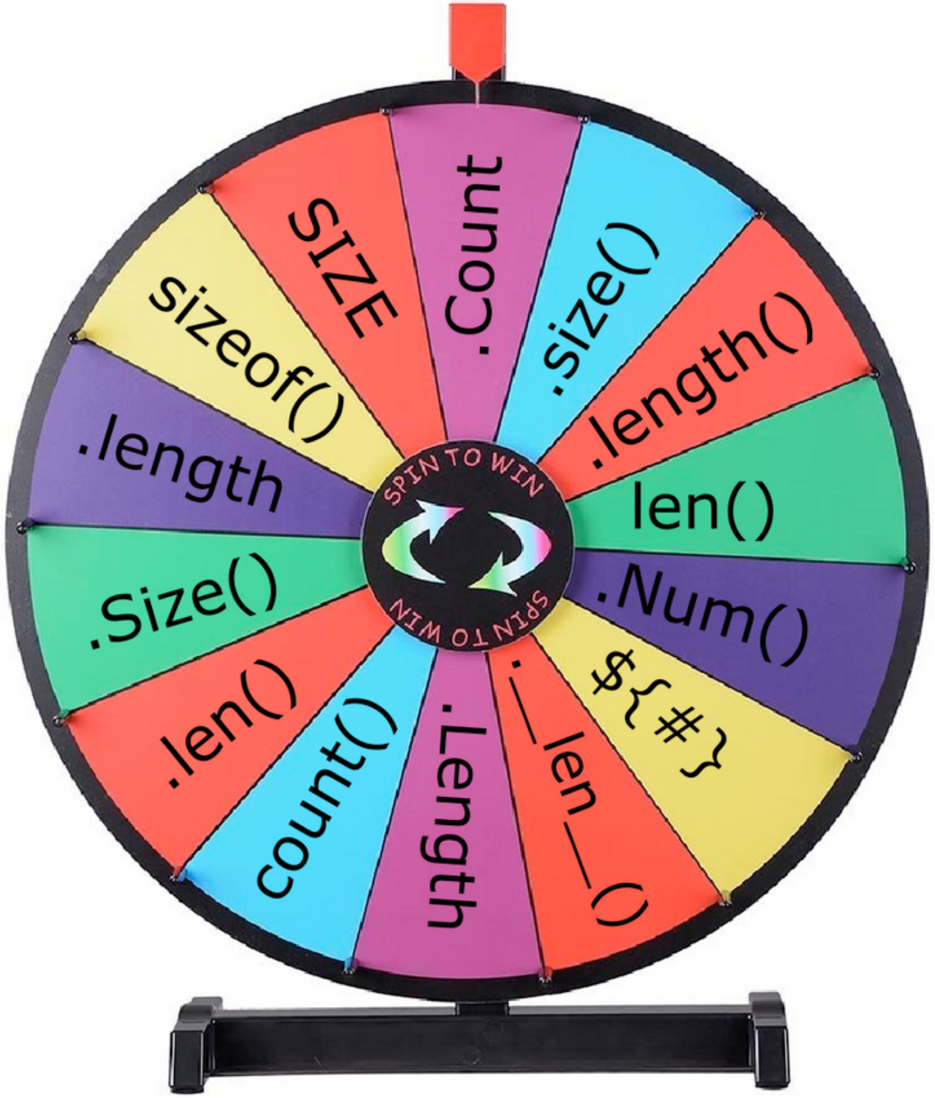
```
if (condition) {
    return;
}
```

```
if (condition)
    return;
```

More work for analysis tools

Multiple languages?

https://c-6rtwjumjzx7868x24ymtzlmyhfyfqtlx2eknqjx78x2ebtwiuwjx78x78x2eh.g00.thoughtcatalog.com/g00/3_c-6ymtzlmyhfyfqtl.htr_/c-6RTWJUMJZX68x24myyux78x3ax2fx2fymtzlmyhfyfqtl.knqjx78.btwiuwjx78x78.htrx2f7560x2f66x2fx78hwjjs-x78mty-7560-66-57-fy-67-62-70-ur.uslx3fwjx78nejx3d6468x2c6627x26vzfqnydx3d40x26x78ywnux3dfqqx26hwtux3d6x26n65h.rfwpx3dnrflj_$/$/$/$

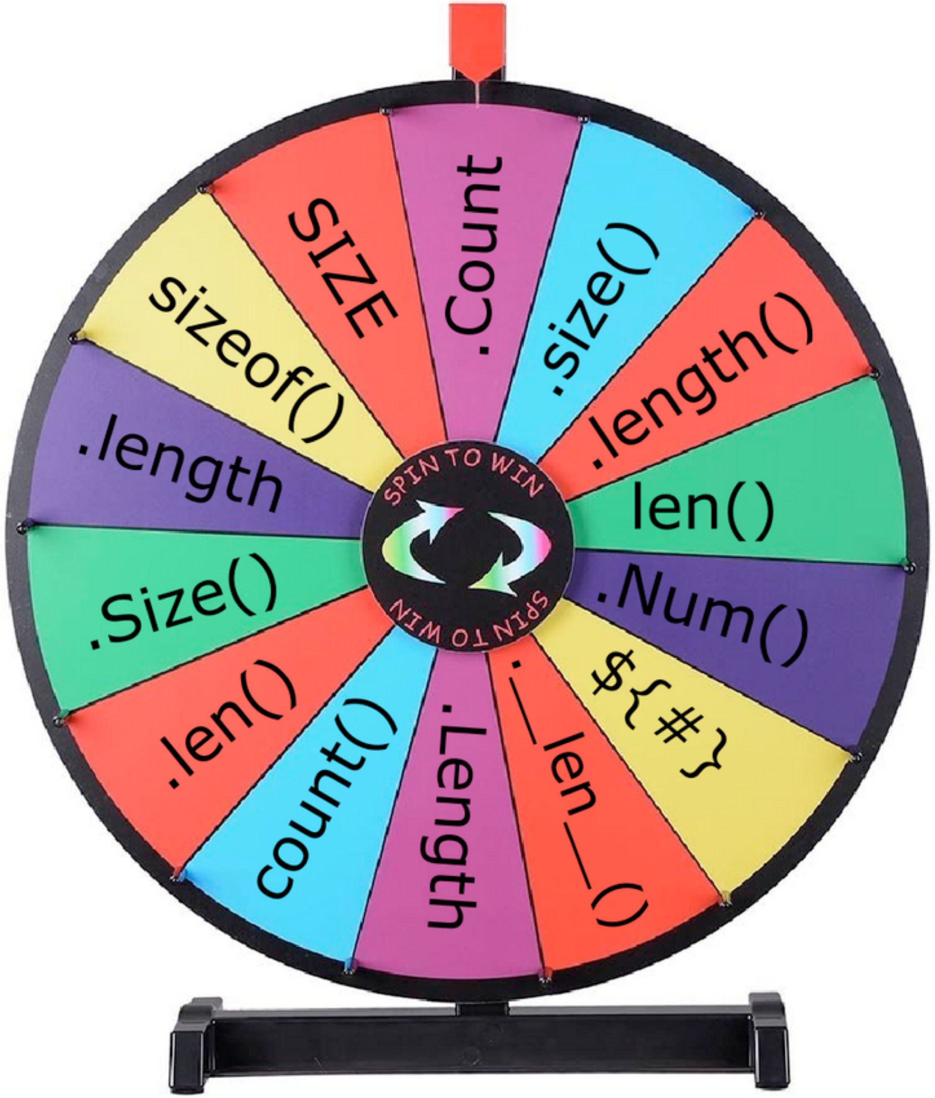

.Count
.size()
.length()
len()
.Num()
${#}
.__len__()
.Length
count()
.len()
.Size()
.length
sizeof()
SIZE
SPIN TO WIN
SPIN TO WIN

sizeof()
SIZE
.Count
.size()
length()
.len()
count()
.Length
__len__()
${#}

Well…

But wouldn’t it be nice if…

No one wrote a tool yet



See the link for demo video

https://drive.google.com/open?id=1xu0Vt_XXyY_9iVT7wmYhSWRfc0cyvltM

No one wrote a tool yet

Syntax extensions

.Count
.size()
.length()
len()
.Num()
${#}
.__len__()
.Length
count()
.len()
.Size()
.length
sizeof()
SIZE
SPIN TO WIN
SPIN TO WIN

PEP 572

```
if (match := pattern.search(data)) ...
```

No one wrote a tool yet

Syntax extensions

Inline assembly?

```
if (ZEND_CONST_COND(offset == 0, 0))
{
  __asm__ ("mul" LP_SUFF  " %3\n\t"
    "adc $0,%1"
     : "=&a"(res), "=&d" (m_overflow)
     : "%0"(res),
       "rm"(size));
}
```

sizeof()
SIZE
.Count
.size()
length()
.len()
count()
.Length
__len__()
${#}

This is why we can’t have nice things

Lightweight Syntax Transformations & Tooling

Example:
Remove redundant nil checks in Go

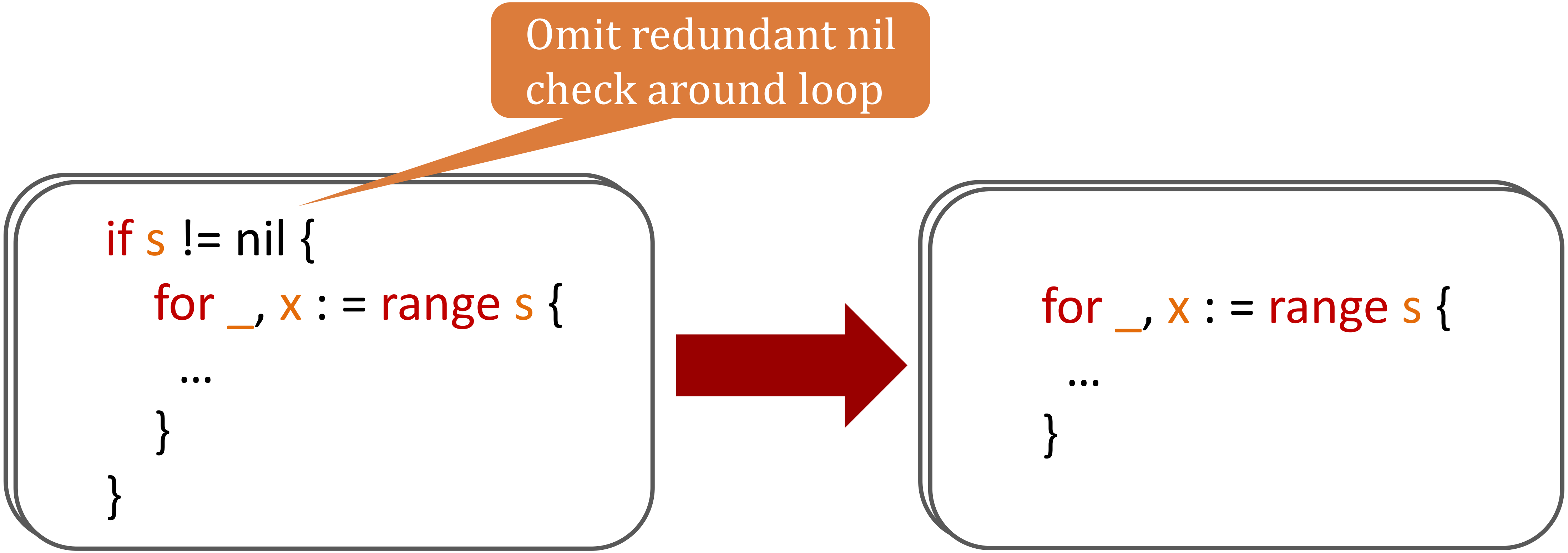
```
if s != nil {
    for _, x : = range s {
      …
    }
}
```

Example:
Remove redundant nil checks in Go

Omit redundant nil check around loop

```
if s != nil {
    for _, x : = range s {
      …
    }
}
```

Example:
Remove redundant nil checks in Go



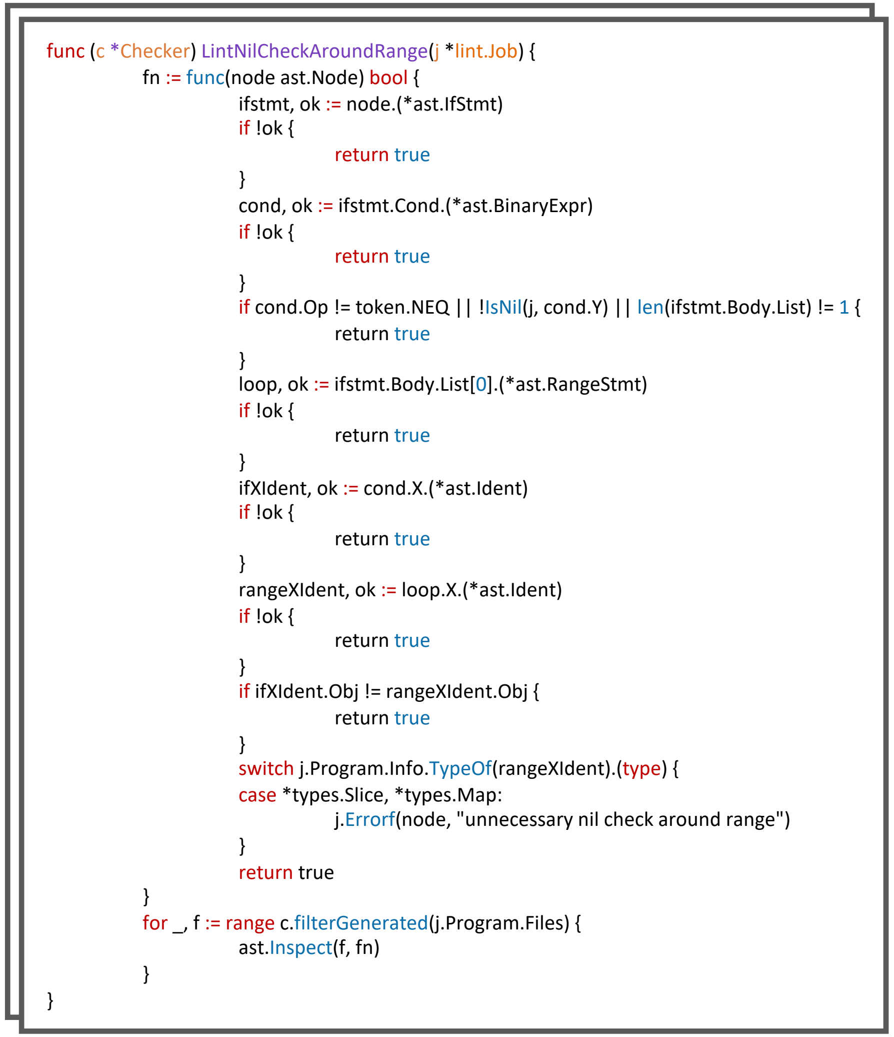
Omit redundant nil check around loop

```
if s != nil {
   for _, x : = range s {
     …
   }
}
```

```
for _, x : = range s {
  …
}
```

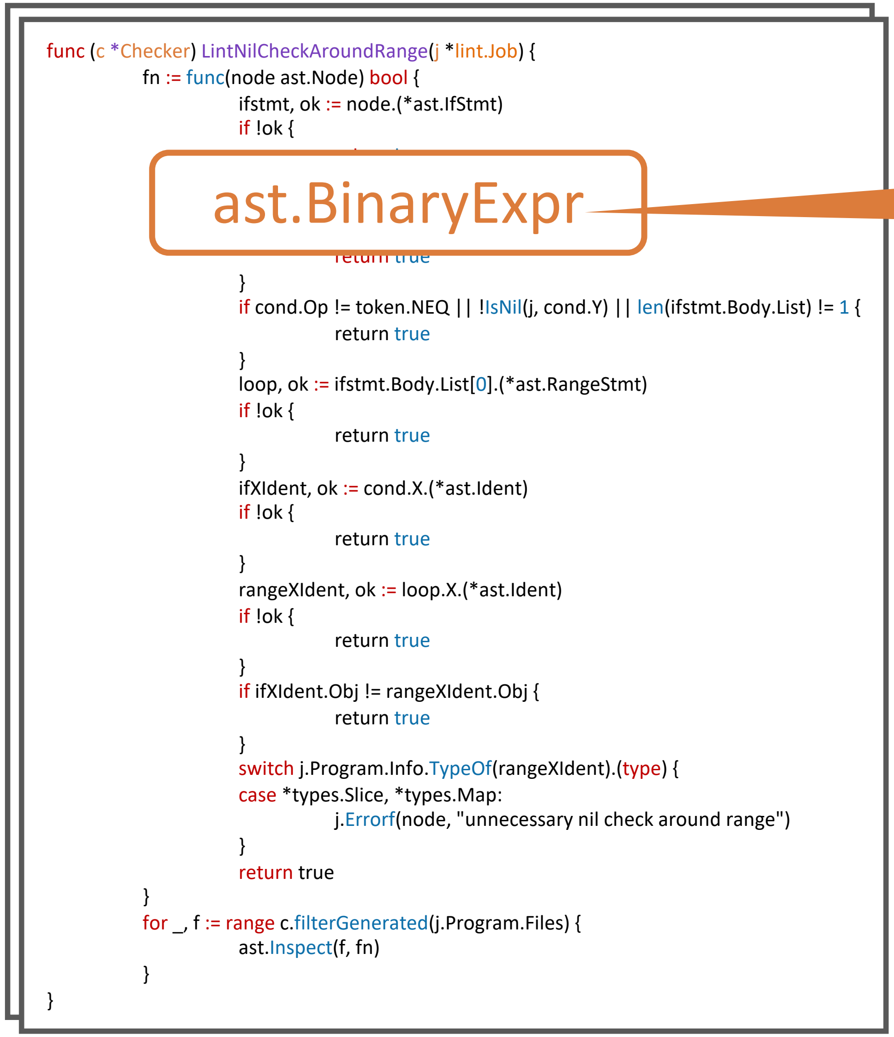
https://staticcheck.io/docs/gosimple#S1031

Implementation for redundant checks



```
func (c *Checker) LintNilCheckAroundRange(j *lint.Job) {
        fn := func(node ast.Node) bool {
                ifstmt, ok := node.(*ast.IfStmt)
                if !ok {
                        return true
                }
                cond, ok := ifstmt.Cond.(*ast.BinaryExpr)
                if !ok {
                        return true
                }
                if cond.Op != token.NEQ || !IsNil(j, cond.Y) || len(ifstmt.Body.List) != 1 {
                        return true
                }
                loop, ok := ifstmt.Body.List[0].(*ast.RangeStmt)
                if !ok {
                        return true
                }
                ifXIdent, ok := cond.X.(*ast.Ident)
                if !ok {
                        return true
                }
                rangeXIdent, ok := loop.X.(*ast.Ident)
                if !ok {
                        return true
                }
                if ifXIdent.Obj != rangeXIdent.Obj {
                        return true
                }
                switch j.Program.Info.TypeOf(rangeXIdent).(type) {
                case *types.Slice, *types.Map:
                        j.Errorf(node, "unnecessary nil check around range")
                }
                return true
        }
        for _, f := range c.filterGenerated(j.Program.Files) {
                ast.Inspect(f, fn)
        }
}
```

Implementation for redundant checks

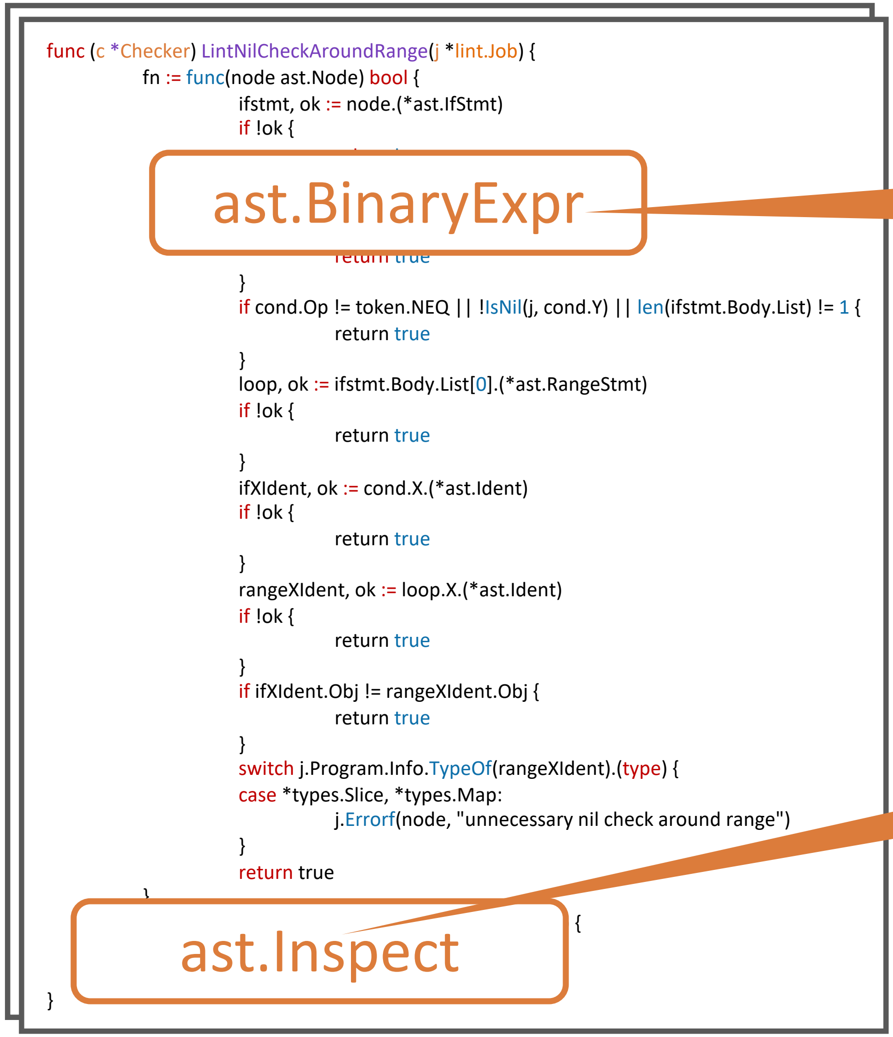


```
func (c *Checker) LintNilCheckAroundRange(j *lint.Job) {
        fn := func(node ast.Node) bool {
                ifstmt, ok := node.(*ast.IfStmt)
                if !ok {
[illegible]
                        return true
                }
                if cond.Op != token.NEQ || !IsNil(j, cond.Y) || len(ifstmt.Body.List) != 1 {
                        return true
                }
                loop, ok := ifstmt.Body.List[0].(*ast.RangeStmt)
                if !ok {
                        return true
                }
                ifXIdent, ok := cond.X.(*ast.Ident)
                if !ok {
                        return true
                }
                rangeXIdent, ok := loop.X.(*ast.Ident)
                if !ok {
                        return true
                }
                if ifXIdent.Obj != rangeXIdent.Obj {
                        return true
                }
                switch j.Program.Info.TypeOf(rangeXIdent).(type) {
                case *types.Slice, *types.Map:
                        j.Errorf(node, "unnecessary nil check around range")
                }
                return true
        }
        for _, f := range c.filterGenerated(j.Program.Files) {
                ast.Inspect(f, fn)
        }
}
```

ast.BinaryExpr

Know the AST data structure

Implementation for redundant checks



```
func (c *Checker) LintNilCheckAroundRange(j *lint.Job) {
	fn := func(node ast.Node) bool {
		ifstmt, ok := node.(*ast.IfStmt)
		if !ok {

			return true
		}
		if cond.Op != token.NEQ || !IsNil(j, cond.Y) || len(ifstmt.Body.List) != 1 {
			return true
		}
		loop, ok := ifstmt.Body.List[0].(*ast.RangeStmt)
		if !ok {
			return true
		}
		ifXIdent, ok := cond.X.(*ast.Ident)
		if !ok {
			return true
		}
		rangeXIdent, ok := loop.X.(*ast.Ident)
		if !ok {
			return true
		}
		if ifXIdent.Obj != rangeXIdent.Obj {
			return true
		}
		switch j.Program.Info.TypeOf(rangeXIdent).(type) {
		case *types.Slice, *types.Map:
			j.Errorf(node, "unnecessary nil check around range")
		}
		return true
	}
                                              {
}
```

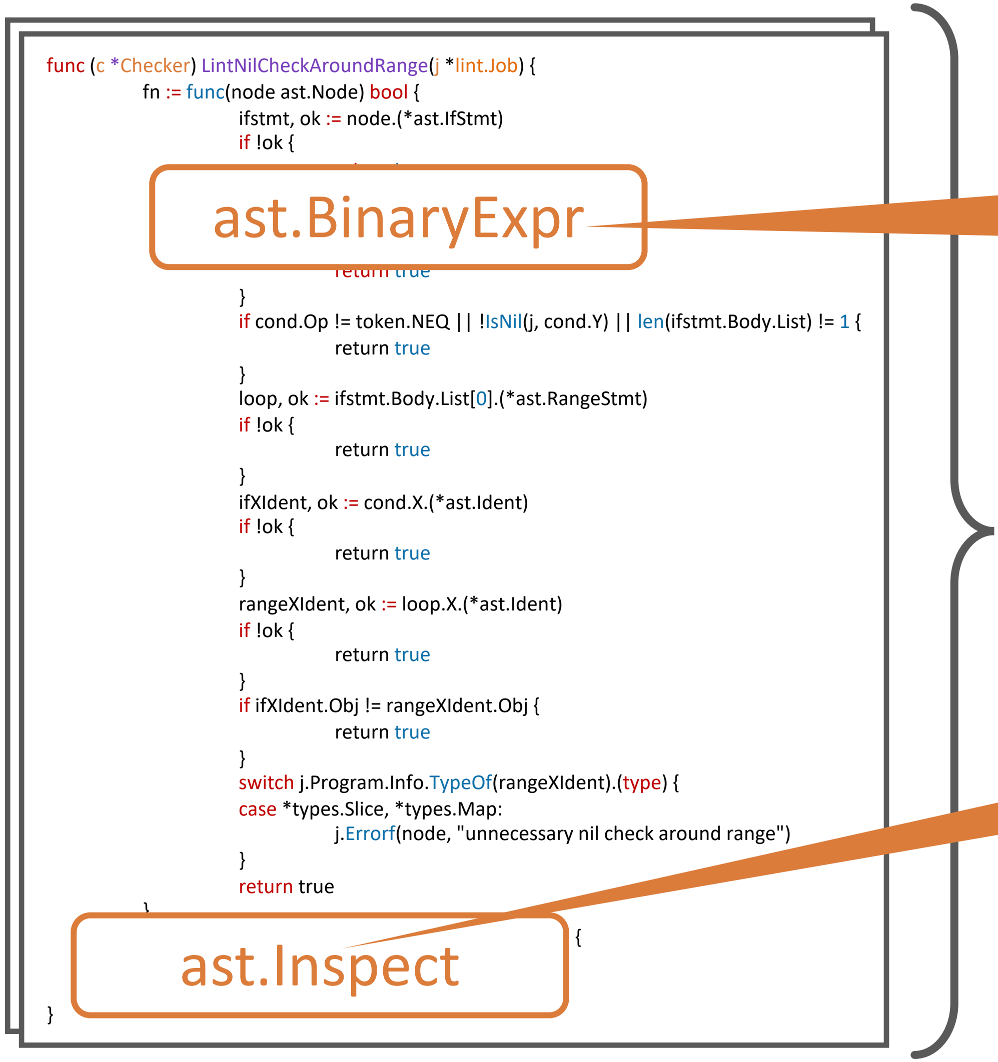
ast.BinaryExpr

ast.Inspect

Know the AST data structure

Learn the visitor API

Implementation for redundant checks



```
func (c *Checker) LintNilCheckAroundRange(j *lint.Job) {
    fn := func(node ast.Node) bool {
        ifstmt, ok := node.(*ast.IfStmt)
        if !ok {

            return true
        }
        if cond.Op != token.NEQ || !IsNil(j, cond.Y) || len(ifstmt.Body.List) != 1 {
            return true
        }
        loop, ok := ifstmt.Body.List[0].(*ast.RangeStmt)
        if !ok {
            return true
        }
        ifXIdent, ok := cond.X.(*ast.Ident)
        if !ok {
            return true
        }
        rangeXIdent, ok := loop.X.(*ast.Ident)
        if !ok {
            return true
        }
        if ifXIdent.Obj != rangeXIdent.Obj {
            return true
        }
        switch j.Program.Info.TypeOf(rangeXIdent).(type) {
        case *types.Slice, *types.Map:
            j.Errorf(node, "unnecessary nil check around range")
        }
        return true
    }
                                                    {
}
```

ast.BinaryExpr

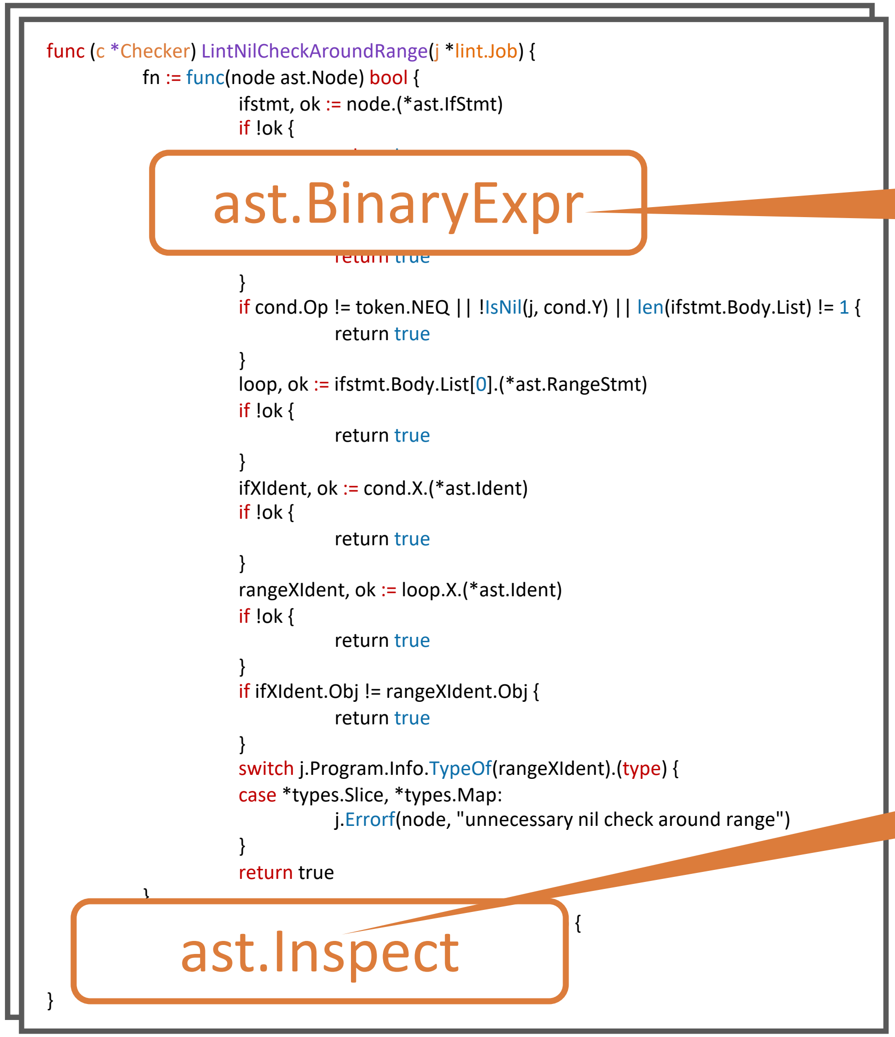
ast.Inspect

Know the AST data structure

Implement it in your language

Learn the visitor API

Implementation for redundant checks



```
func (c *Checker) LintNilCheckAroundRange(j *lint.Job) {
    fn := func(node ast.Node) bool {
        ifstmt, ok := node.(*ast.IfStmt)
        if !ok {

            return true
        }
        if cond.Op != token.NEQ || !IsNil(j, cond.Y) || len(ifstmt.Body.List) != 1 {
            return true
        }
        loop, ok := ifstmt.Body.List[0].(*ast.RangeStmt)
        if !ok {
            return true
        }
        ifXIdent, ok := cond.X.(*ast.Ident)
        if !ok {
            return true
        }
        rangeXIdent, ok := loop.X.(*ast.Ident)
        if !ok {
            return true
        }
        if ifXIdent.Obj != rangeXIdent.Obj {
            return true
        }
        switch j.Program.Info.TypeOf(rangeXIdent).(type) {
        case *types.Slice, *types.Map:
            j.Errorf(node, "unnecessary nil check around range")
        }
        return true
    }
                                                  {
}
```

ast.BinaryExpr

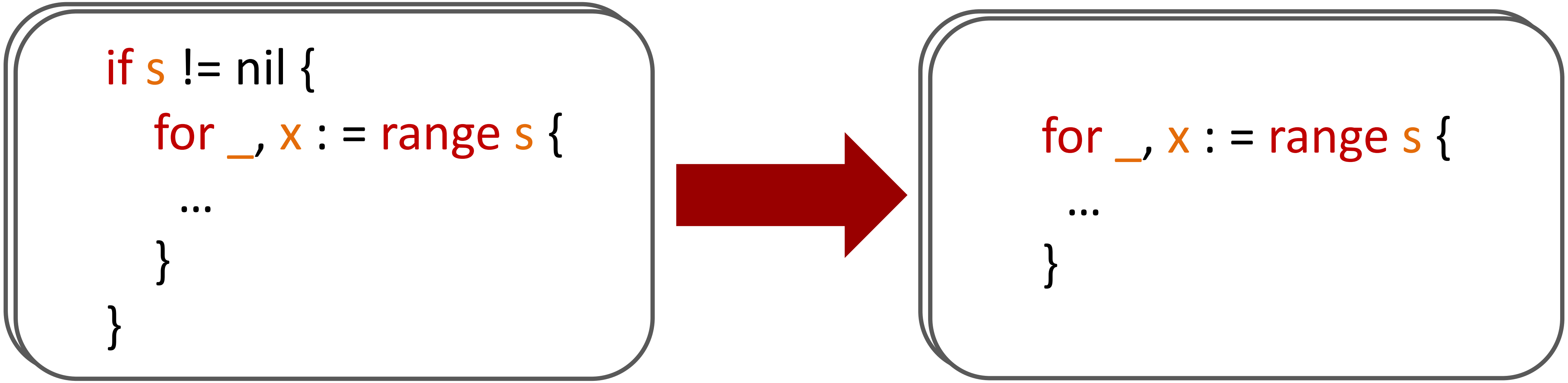
ast.Inspect

Know the AST data structure

Implement it in your language

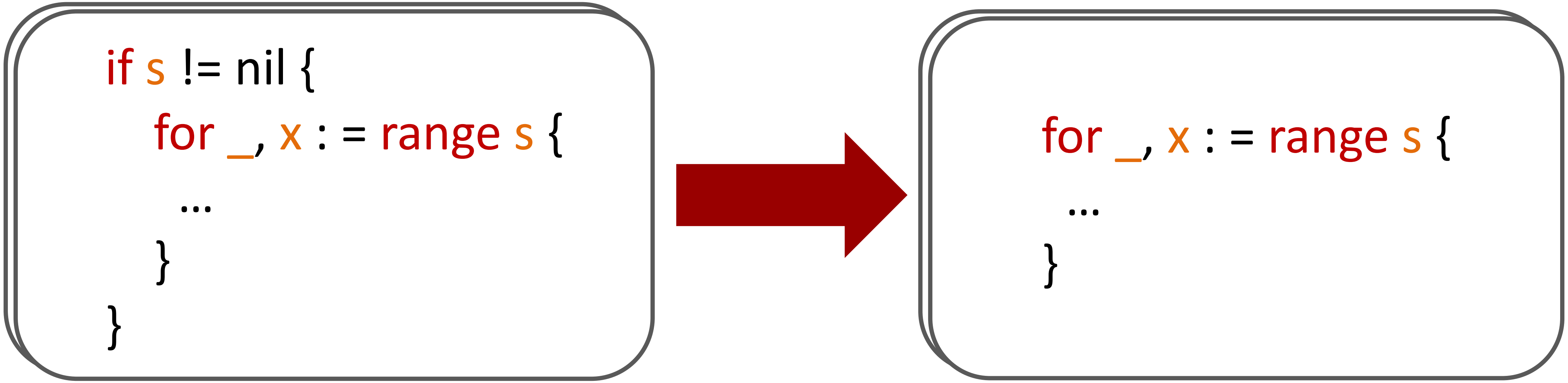
Learn the visitor API

Now do the same for Rust, C, Haskell…



```
if s != nil {
    for _, x : = range s {
      …
    }
}
```

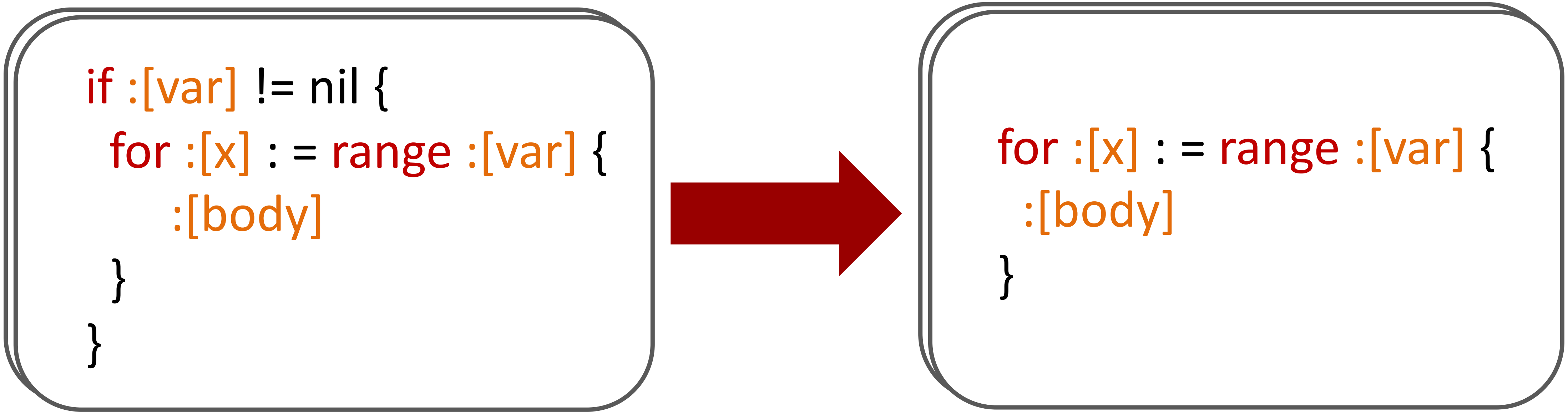
```
for _, x : = range s {
  …
}
```



```
if s != nil {
    for _, x : = range s {
      …
    }
}
```

```
for _, x : = range s {
  …
}
```

Solution: syntactically close templates



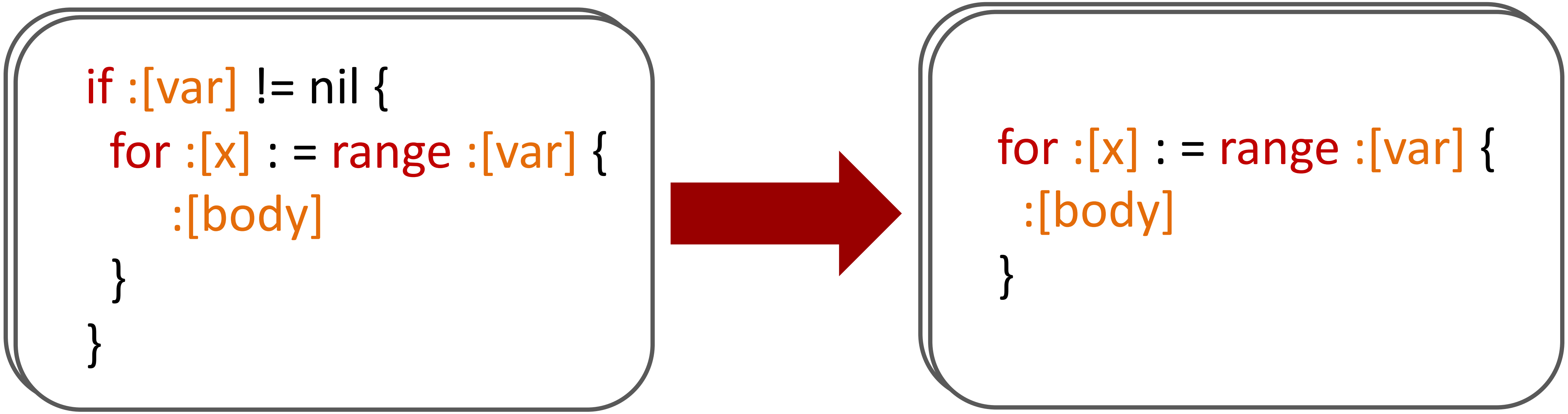
```
if :[var] != nil {
  for :[x] : = range :[var] {
       :[body]
  }
}
```

```
for :[x] : = range :[var] {
  :[body]
}
```

See the link for demo video

https://drive.google.com/open?id=19X9YL2tZmfOCvK8GxL8OEnUkUB88SC3n

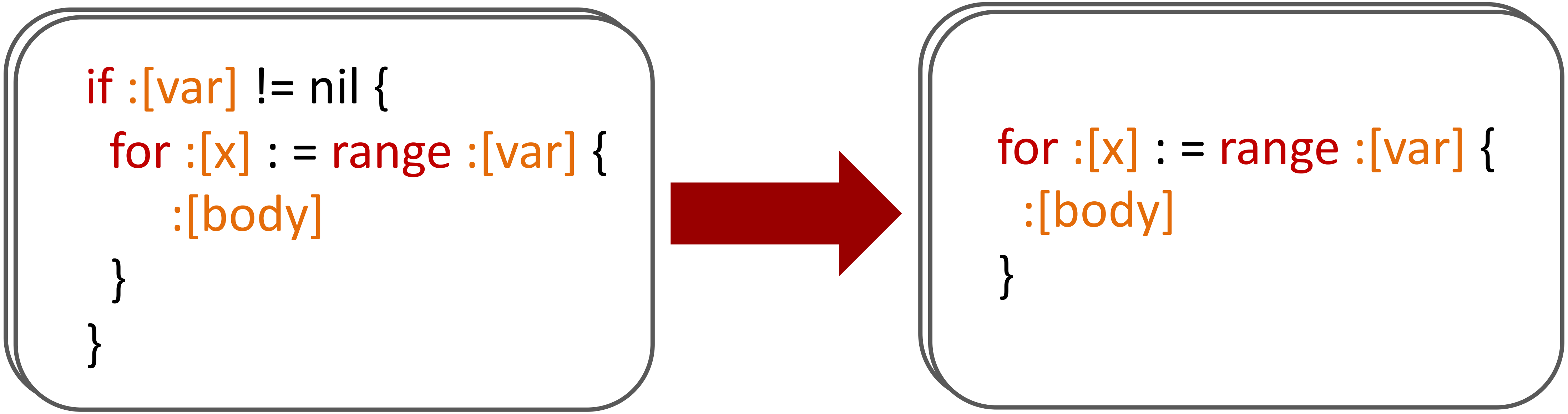
Syntax only



```
if :[var] != nil {
 for :[x] : = range :[var] {
     :[body]
 }
}
```

```
for :[x] : = range :[var] {
  :[body]
}
```

Nothing about this is Go specific

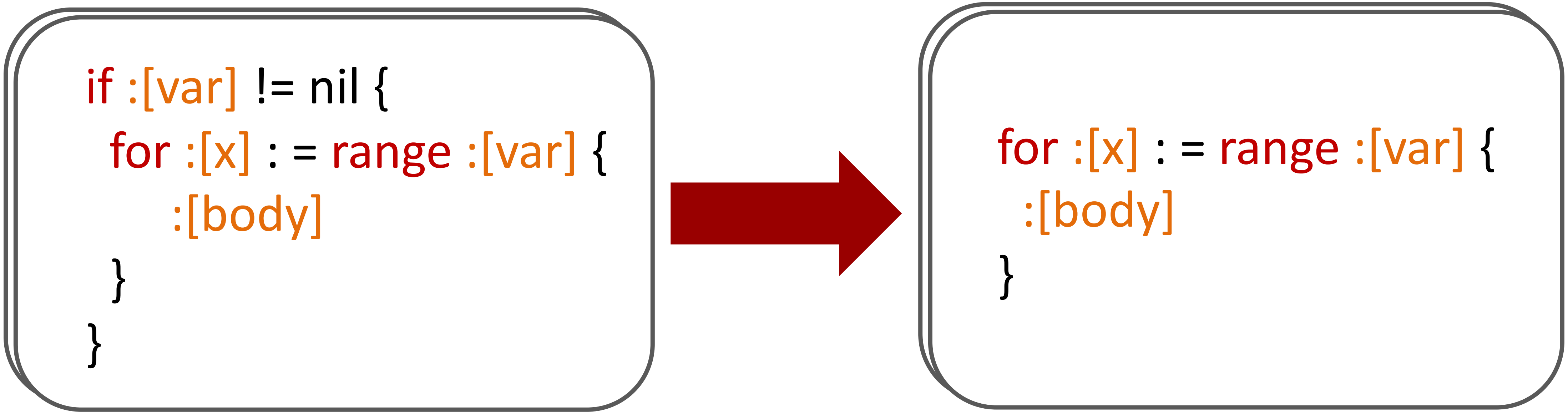


```
if :[var] != nil {
  for :[x] : = range :[var] {
       :[body]
  }
}
```

```
for :[x] : = range :[var] {
  :[body]
}
```

Nothing about this is Go specific

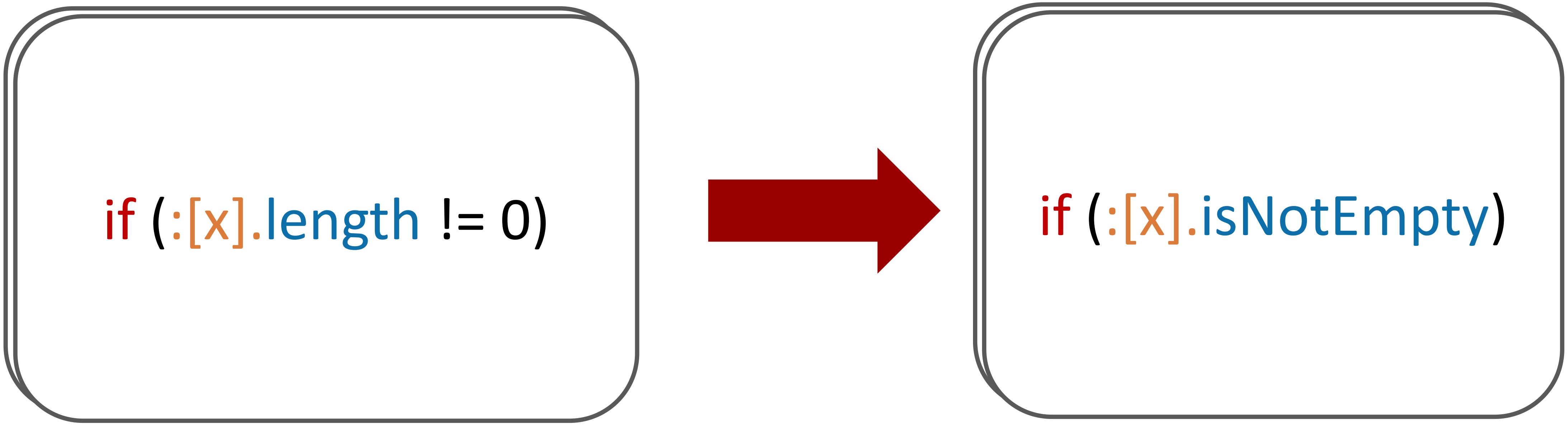
(syntactically)



```
if :[var] != nil {
 for :[x] : = range :[var] {
     :[body]
 }
}
```

```
for :[x] : = range :[var] {
 :[body]
}
```

Dart

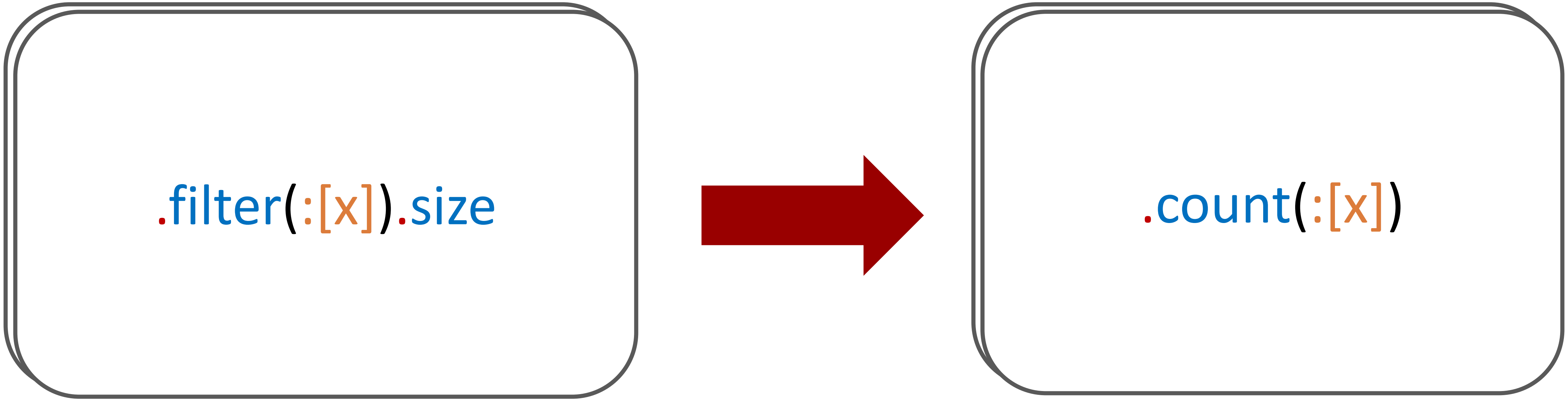


```
if (:[x].length != 0)
```

```
if (:[x].isNotEmpty)
```

`(= :[x] nil)` → `(?nil :[x])`

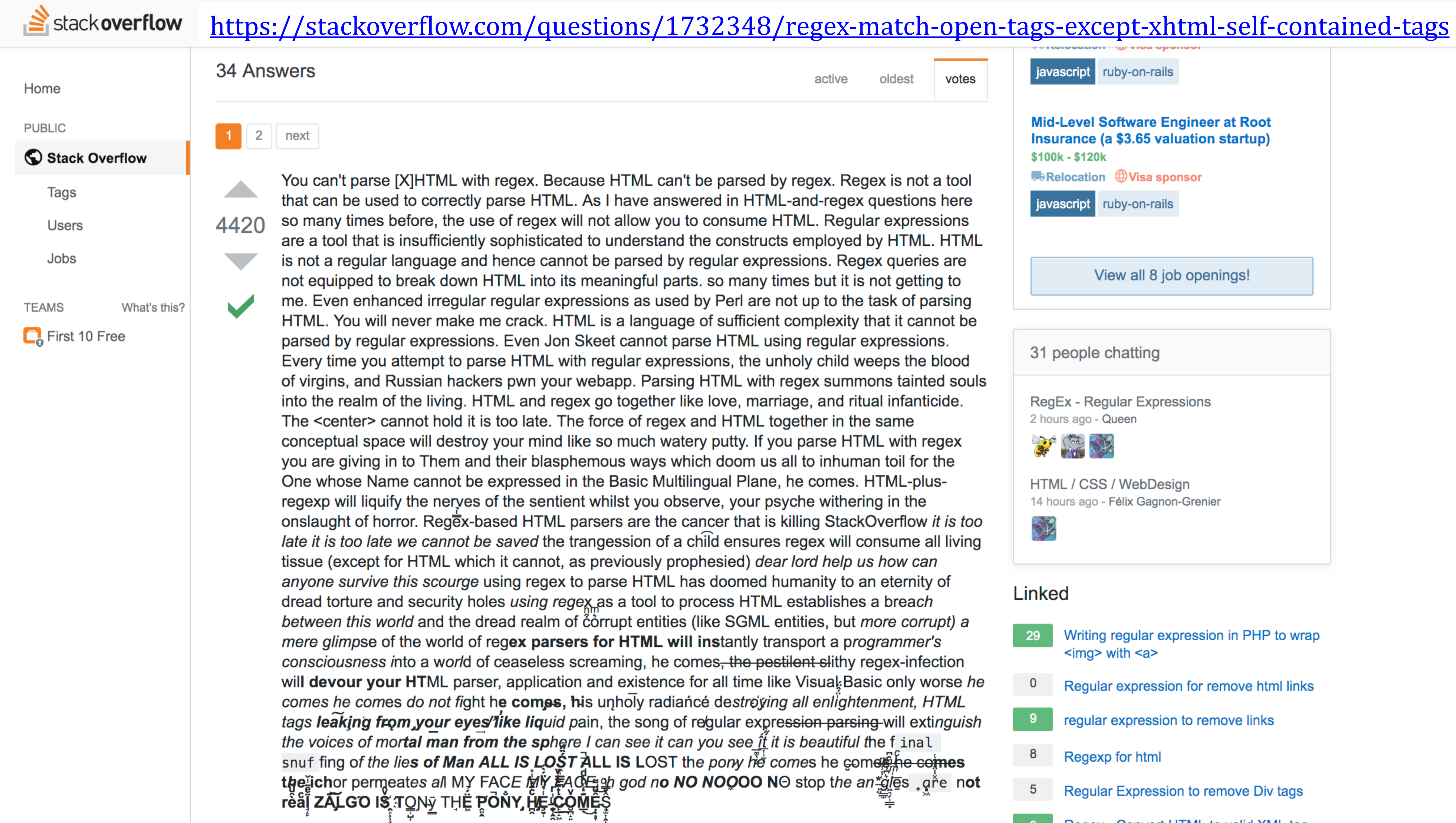
Scala



.filter(:[x]).size

.count(:[x])

https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags



34 Answers

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach *between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimp*se of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a wor*l*d of ceaseless screaming, he comes~~, the pestilent sl~~ithy regex-infection wil**l devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, hi**s unholy radiancé de*stroying all enlightenment, HTML tags* ***leaking from your eyes like liq****uid* pain, the song of regular expr~~ession parsing~~ will exti*nguish the voices of mor****tal man from the sp****here I can see it can you see it it is beautiful t*he f `inal snuf` fing o*f the lie****s of Man ALL IS LOST ALL IS** LOST th*e pony he comes* he c~~omes he comes~~ **the ich**or permeates al*l MY FACE *MY FACE ᵒh god n**o NO NOOOO N**Θ stop th*e an*gles `are` **not real ZALGO IS TONY THE PONY HE COMES**

Mid-Level Software Engineer at Root Insurance (a $3.65 valuation startup)
$100k - $120k
Relocation Visa sponsor

31 people chatting

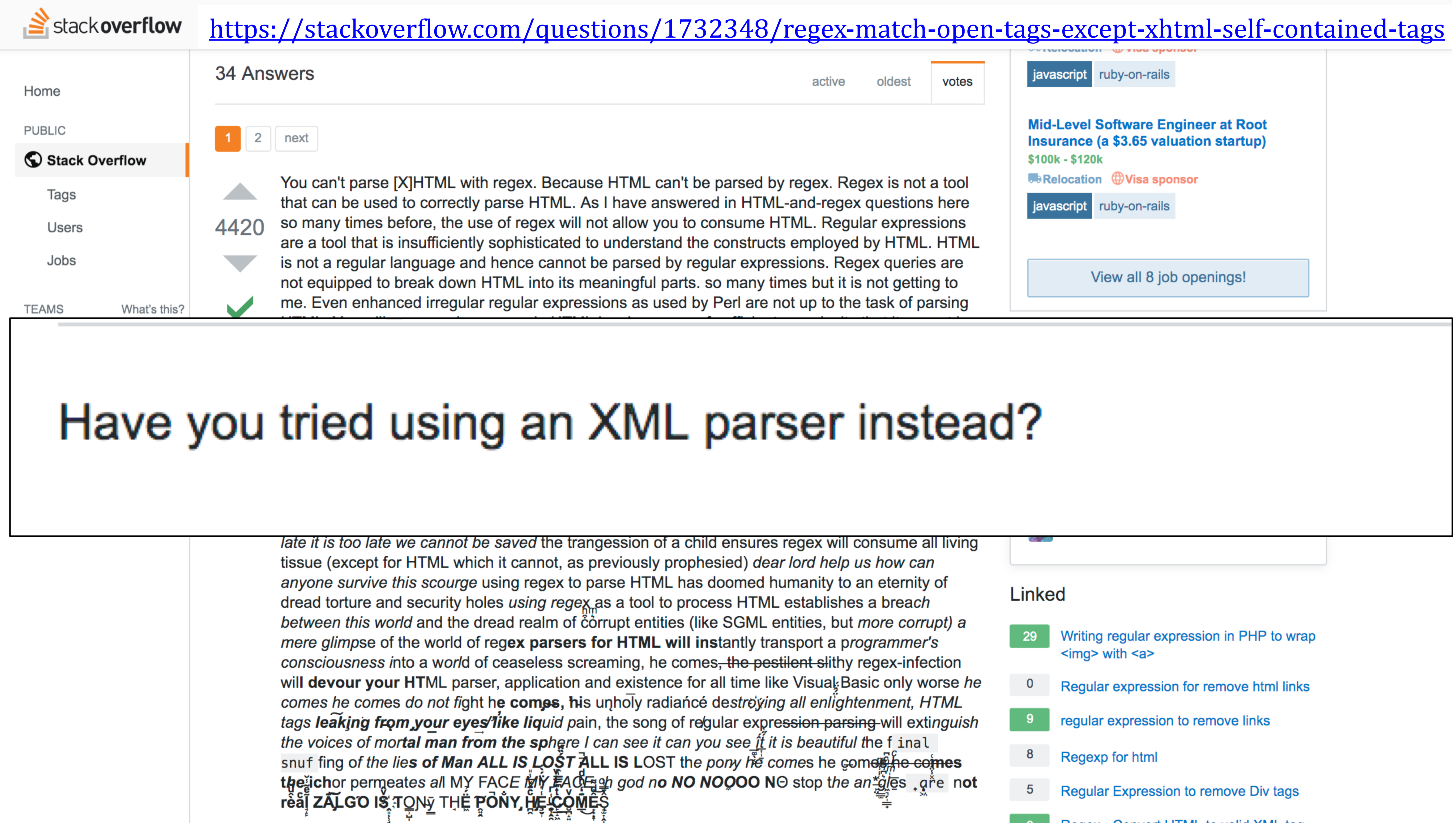
RegEx - Regular Expressions
2 hours ago - Queen

HTML / CSS / WebDesign
14 hours ago - Félix Gagnon-Grenier

Linked

29	Writing regular expression in PHP to wrap <img> with <a>
0	Regular expression for remove html links
9	regular expression to remove links
8	Regexp for html
5	Regular Expression to remove Div tags

https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags



34 Answers

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing

Have you tried using an XML parser instead?

late it is too late we cannot be saved the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach *between this world* and the dread realm of c̒ͪo͛ͫrrupt entities (like SGML entities, but *more corrupt) a mere glimpse* of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a wor*l*d of ceaseless screaming, he comes~~, the pestilent sl~~ithy regex-infection wil**l devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e com̡e̶s, ̕h̵i**s un̨ho͞ly radiańcé destro҉ying all enli̍ghtenment, HTML tags **lea͠ki̧n͘g fr̶ǫm ̡yo͟ur eye͢s̸ ̛l̕ik͏e liq**uid pain, the song of re̸gular expre~~ssion parsing~~ will ext*ingu*ish *the voices of mor***tal man from the sp**here I can see it can you see ̲͚̖͔̙î̩́t̲͎̩̱͔́̋̀ it is beautiful t*he f `inal snuf` fing o*f the lie***s of Man ALL IS LOŚ͖̩͇̗̪̏̈́T ALL I**S LOST the pon̷y he comes he c̶̮om~~es he co~~mes **t**h**e ich**or permeates al*l MY FACE MY FACE ᵒh god no **NO NOO̼OO N**Θ stop the an*̶͑̾̾̅ͫ͏̙̤g͛͆̾ͫ̑͆l͖͉̗̩̳̟̍ͫͥͨe̠̅s `ͣ͒a̧re` not rè̑ͧ̌aͨl̘̝̙̃ͤ͂̾̆ **ZA̡͊͠͝LGΌ ISͮ̂҉̯͈͕̹̘̱ TO͇̹̺ͅƝ̴ȳ̳ TH̘Ë͖́̉ ͠P̯͍̭O̚N̐Y̡ H̸̡̪̯ͨ͊̽̅̾̎Ȩ̬̩̾͛ͪ̈́̀́͘ ̶̧̨̱̹̭̯ͧ̾ͬC̷̙̲̝͖ͭ̏ͥͮ͟Oͮ͏̮̪̝͍M̲̖͊̒ͪͩͬ̚̚͜Ȇ̴̟̟͙̞ͩ͌͝S̨̥̫͎̭ͯ̿̔̀ͅ**

Linked

29	Writing regular expression in PHP to wrap <img> with <a>
0	Regular expression for remove html links
9	regular expression to remove links
8	Regexp for html
5	Regular Expression to remove Div tags

A Parser for Multiple Languages

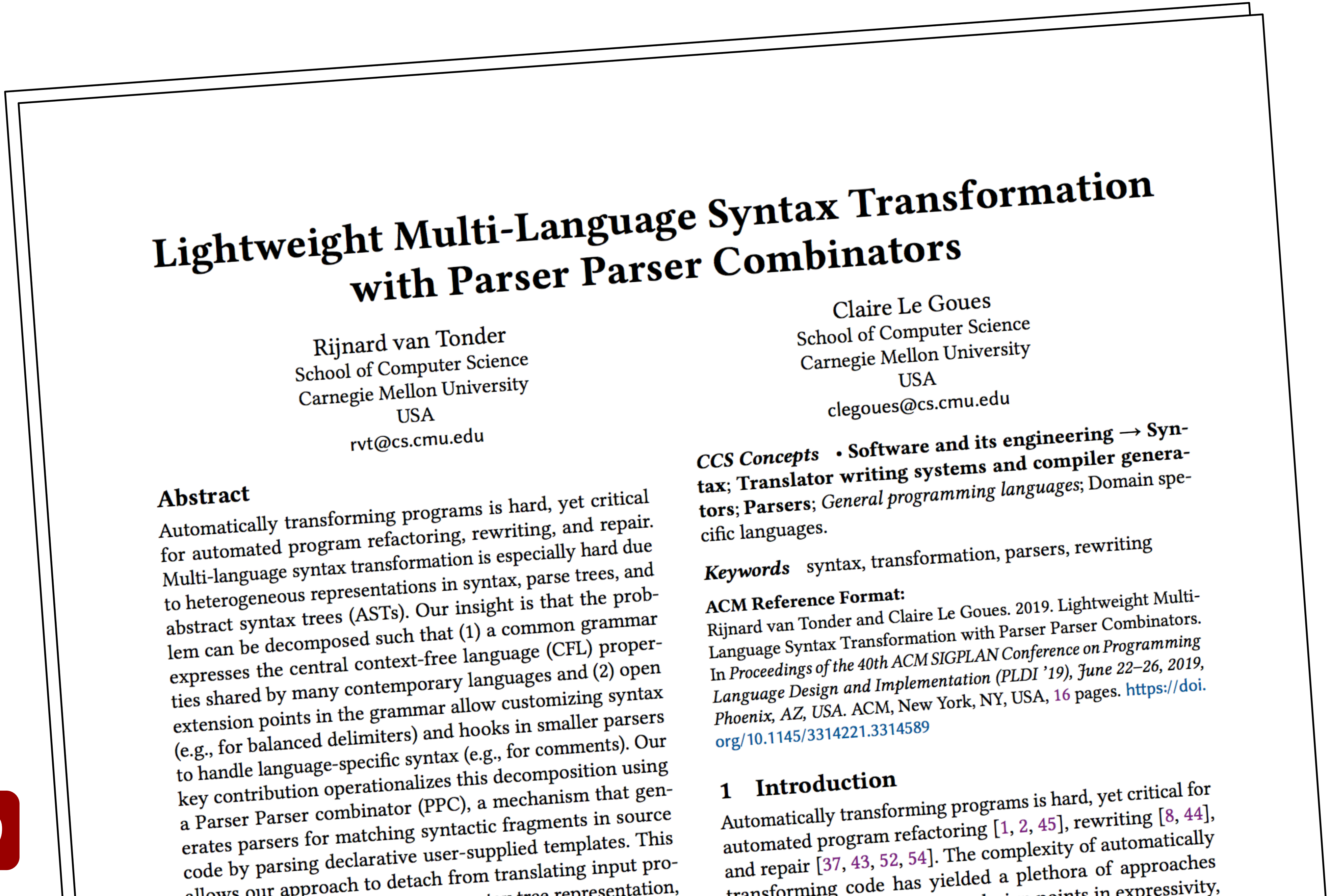
A parser for multiple languages

- Shared context-free language properties
  - Balanced delimiters
  - Delineate trees

A parser for multiple languages

- Shared context-free language properties
  - Balanced delimiters
  - Delineate trees
- Take a parenthesis language ➡ extend it

$$S \rightarrow \epsilon \mid SS \mid (\ S \ )$$



Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators

Rijnard van Tonder
School of Computer Science
Carnegie Mellon University
USA
rvt@cs.cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
USA
clegoues@cs.cmu.edu

Abstract

Automatically transforming programs is hard, yet critical for automated program refactoring, rewriting, and repair. Multi-language syntax transformation is especially hard due to heterogeneous representations in syntax, parse trees, and abstract syntax trees (ASTs). Our insight is that the problem can be decomposed such that (1) a common grammar expresses the central context-free language (CFL) properties shared by many contemporary languages and (2) open extension points in the grammar allow customizing syntax (e.g., for balanced delimiters) and hooks in smaller parsers to handle language-specific syntax (e.g., for comments). Our key contribution operationalizes this decomposition using a Parser Parser combinator (PPC), a mechanism that generates parsers for matching syntactic fragments in source code by parsing declarative user-supplied templates. This allows our approach to detach from translating input pro- … ion tree representation,

CCS Concepts • **Software and its engineering → Syntax; Translator writing systems and compiler generators; Parsers**; *General programming languages*; Domain specific languages.

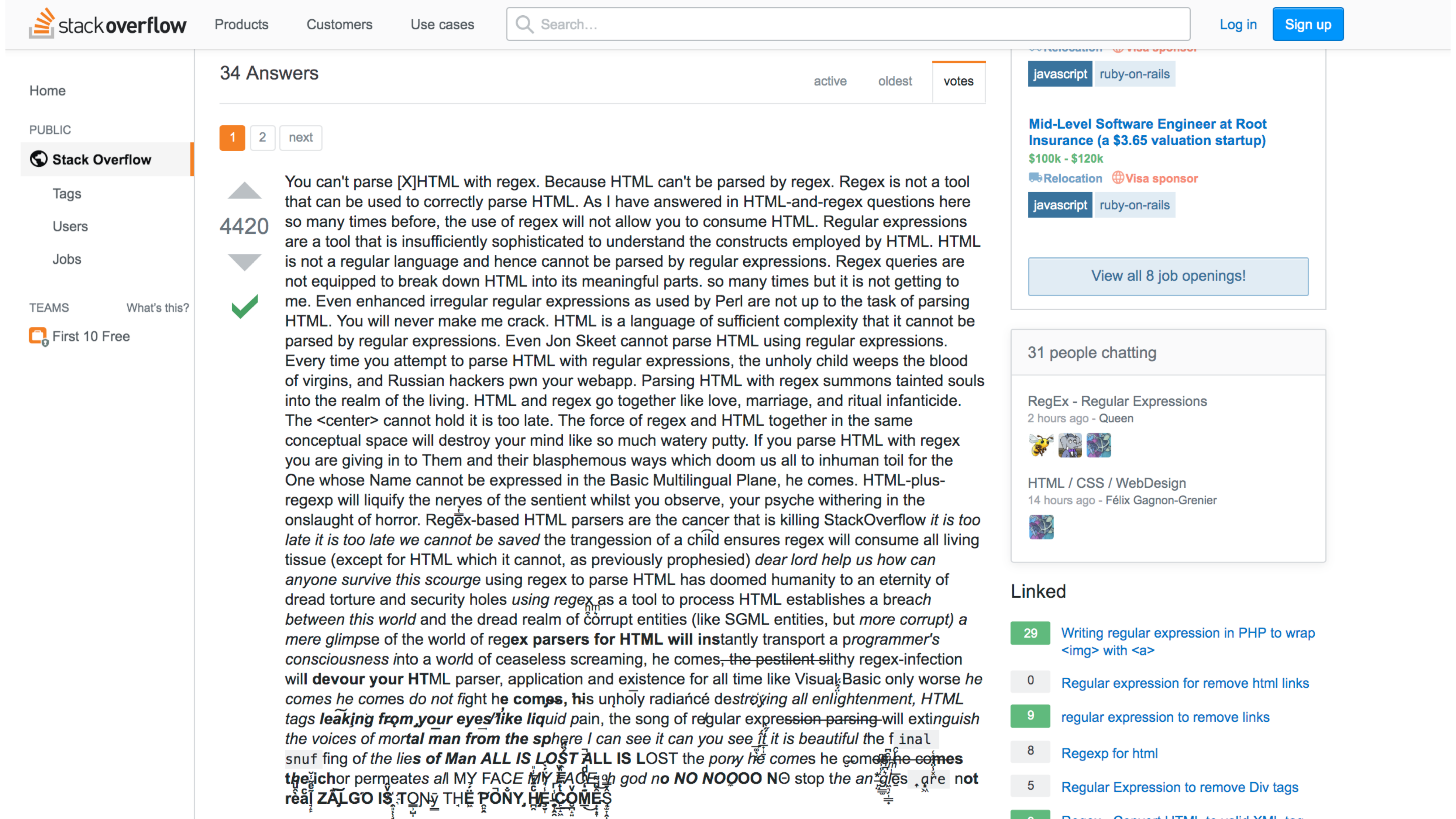
Keywords syntax, transformation, parsers, rewriting

ACM Reference Format:
Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314589

1 Introduction

Automatically transforming programs is hard, yet critical for automated program refactoring [1, 2, 45], rewriting [8, 44], and repair [37, 43, 52, 54]. The complexity of automatically transforming code has yielded a plethora of approaches … points in expressivity,

PLDI '19



34 Answers

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach *between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimp*se of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a wor*l*d of ceaseless screaming, he comes~~, the pestilent sl~~ithy regex-infection wil**l devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, hi**s unholy radiancé de*stroying all enlightenment, HTML tags* ***leaking from your eyes like liq****uid* pain, the song of regular expr~~ession parsing~~ will exti*nguish the voices of mor****tal man from the sp****here I can see it can you see it it is beautiful t*he f `inal snuf` fing o*f the lie****s of Man ALL IS LOST ALL IS L**OST the pon*y he comes* he com~~es he comes~~ **the ich**or permeat*es al*l MY FACE *MY FACE ᵒh god n***o NO NOOOO N**Θ stop *the an*gles* `are` not **real ZALGO IS TONY THE PONY HE COMES**

Relocation
Visa sponsor
javascript
ruby-on-rails
Mid-Level Software Engineer at Root Insurance (a $3.65 valuation startup)
$100k - $120k
Relocation
Visa sponsor
javascript
ruby-on-rails
View all 8 job openings!
31 people chatting
RegEx - Regular Expressions
2 hours ago - Queen
HTML / CSS / WebDesign
14 hours ago - Félix Gagnon-Grenier
Linked
29
Writing regular expression in PHP to wrap <img> with <a>
0
Regular expression for remove html links
9
regular expression to remove links
8
Regexp for html
5
Regular Expression to remove Div tags

A simple grammar

```
grammar ::= term* EOF
term ::= '(' term ')' | '{' term '}' | '[' term ']’ | term term | token
token ::= ...
```

A simple grammar

grammar ::= term* EOF

term ::= '(' term ')' | '{' term '}' | '[' term ']’ | term term | token

token ::= anything_else

Example parse tree for Go code

```
func main() { fmt.Println("hello world") }
```

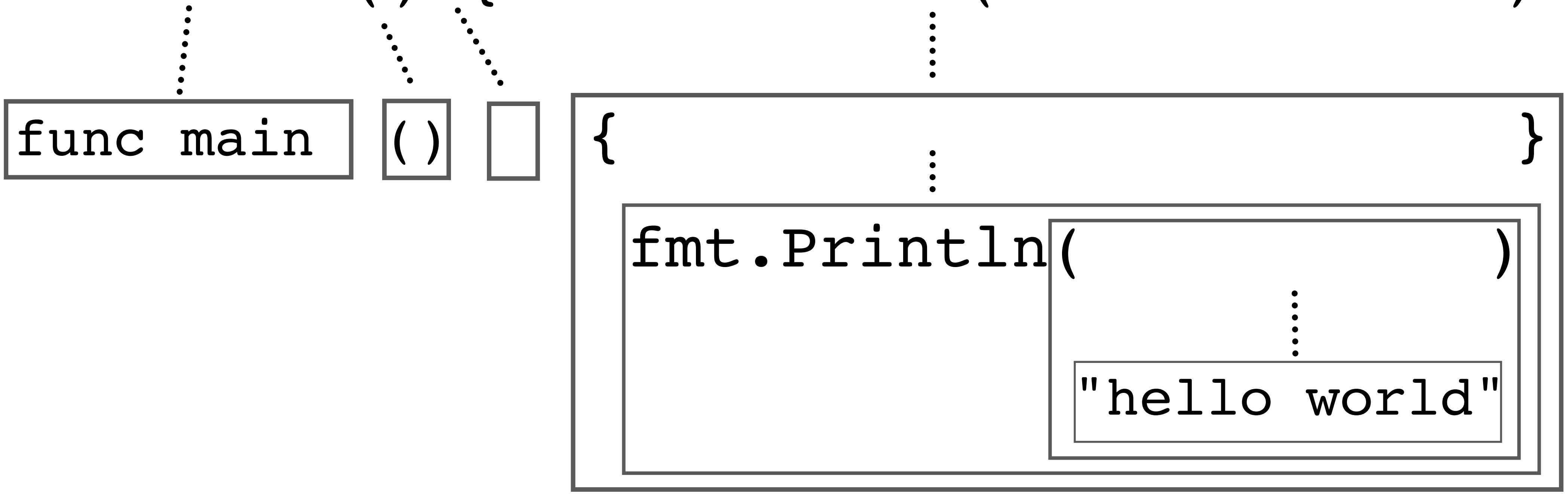
Example parse tree for Go code

```
func main() { fmt.Println("hello world") }
```

Decompose with respect to delimiters

Example parse tree for Go code

```
func main() { fmt.Println("hello world") }
```



func main

()

{

}

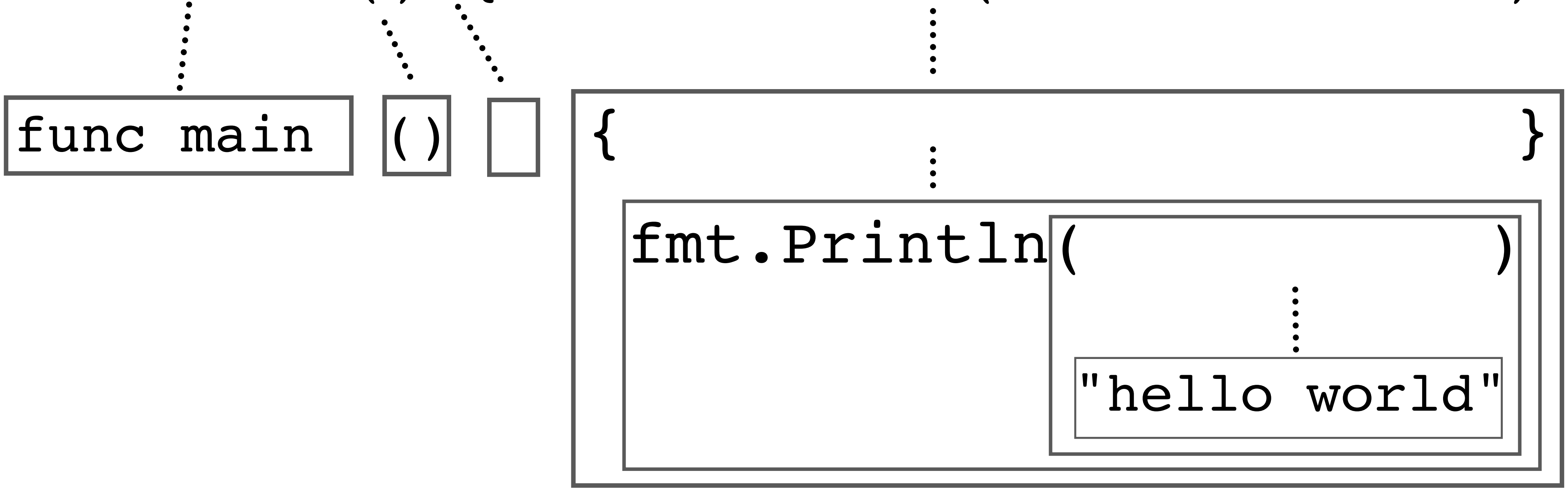
fmt.Println(

)

"hello world"

Example parse tree for Go code

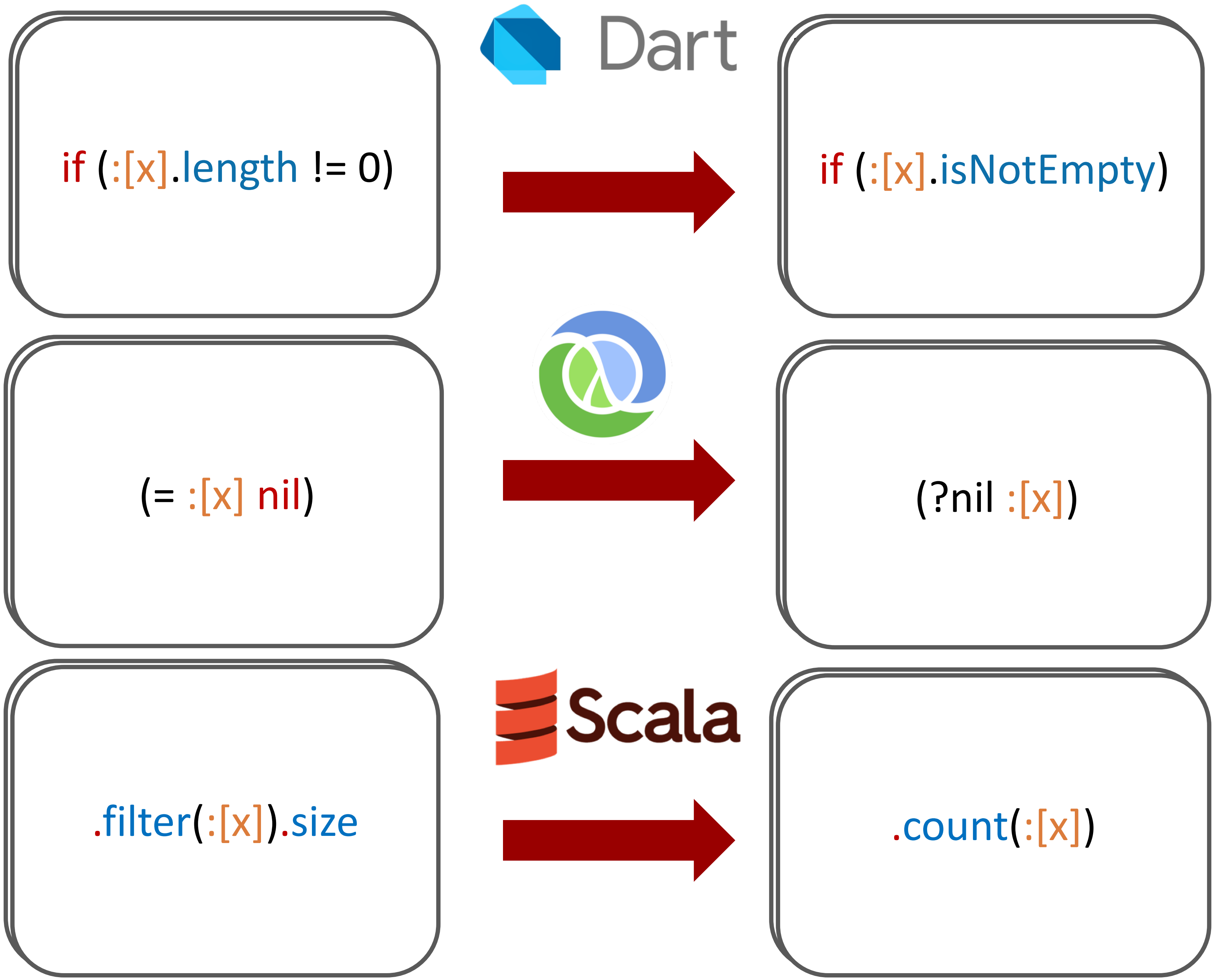
```
func main() { fmt.Println("hello world") }
```



grammar ::= term EOF

term ::= '(' term ')' | '{' term '}' | '[' term ']’ | term term | token

token ::= anything_else



Dart

```
if (:[x].length != 0)
```

```
if (:[x].isNotEmpty)
```

```
(= :[x] nil)
```

```
(?nil :[x])
```

Scala

```
.filter(:[x]).size
```

```
.count(:[x])
```

How to match
declarative templates ⟷ source code?

```
if :[var] != nil {
 for :[x] : = range :[var] {
      :[body]
 }
}
```

Parser Combinators

Model parsers as functions that can be composed using higher-order functions (combinators) to implement grammar constructions.

Parser Combinators

A parser for an int…

Parser Combinators

A parser for a string…

Parser Combinators

A parser for an expression…

Parser Combinators are polymorphic in their production

A parser for an expression…

Parser Combinators are polymorphic in their production

A parser for parsers

How to match declarative templates ⟷ source code?

```
if :[var] != nil {
  for :[x] : = range :[var] {
      :[body]
  }
}
```

How to match
declarative templates ⟷ source code?

```
if :[var] != nil {
  for :[x] : = range :[var] {
        :[body]
  }
}
```

This defines a parser

How to match declarative templates ⟷ source code?

Parse an if

```
if :[var] != nil {
 for :[x] : = range :[var] {
     :[body]
  }
}
```

This defines a parser

How to match declarative templates ⟷ source code?

Parse an if

```
if :[var] != nil {
 for :[x] : = range :[var] {
      :[body]
  }
}
```

Produce a parser for “if”

This defines a parser

How to match declarative templates ⟷ source code?

Parse whitespace

```
if :[var] != nil {
  for :[x] : = range :[var] {
      :[body]
  }
}
```

This defines a parser

How to match
declarative templates ⟷ source code?

Parse whitespace

```
if :[var] != nil {
  for :[x] : = range :[var] {
      :[body]
  }
}
```

➡ Produce a parser for whitespace

This defines a parser

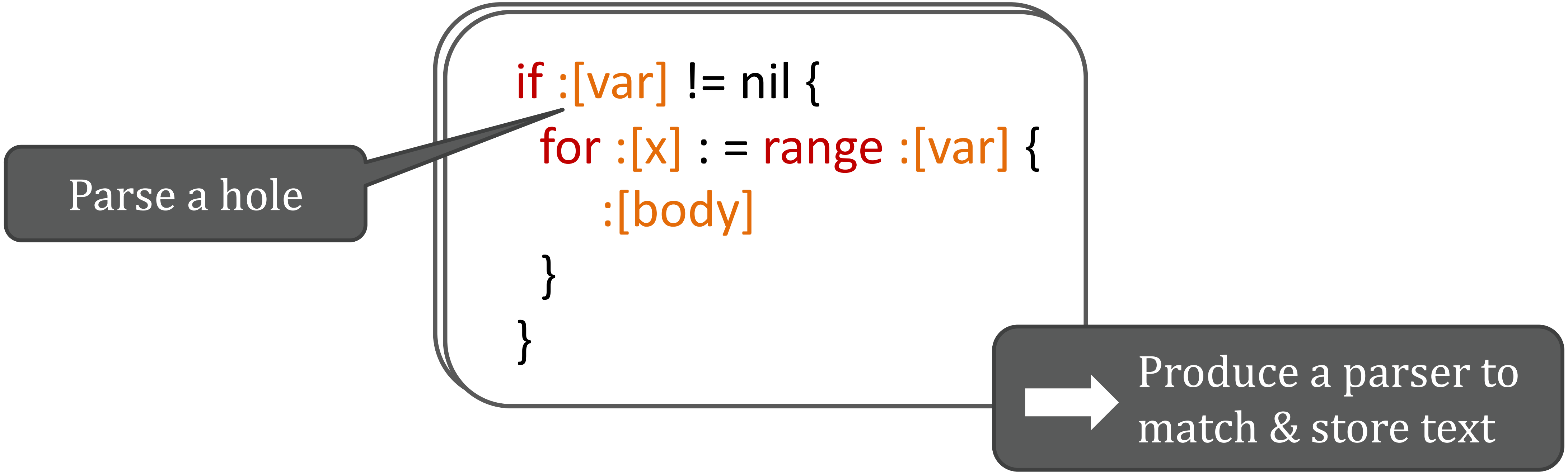
How to match declarative templates ⟷ source code?

Parse a hole

```
if :[var] != nil {
 for :[x] : = range :[var] {
      :[body]
 }
}
```

This defines a parser

How to match declarative templates ⟷ source code?



```
if :[var] != nil {
 for :[x] : = range :[var] {
     :[body]
 }
}
```

Parse a hole

Produce a parser to match & store text

This defines a parser

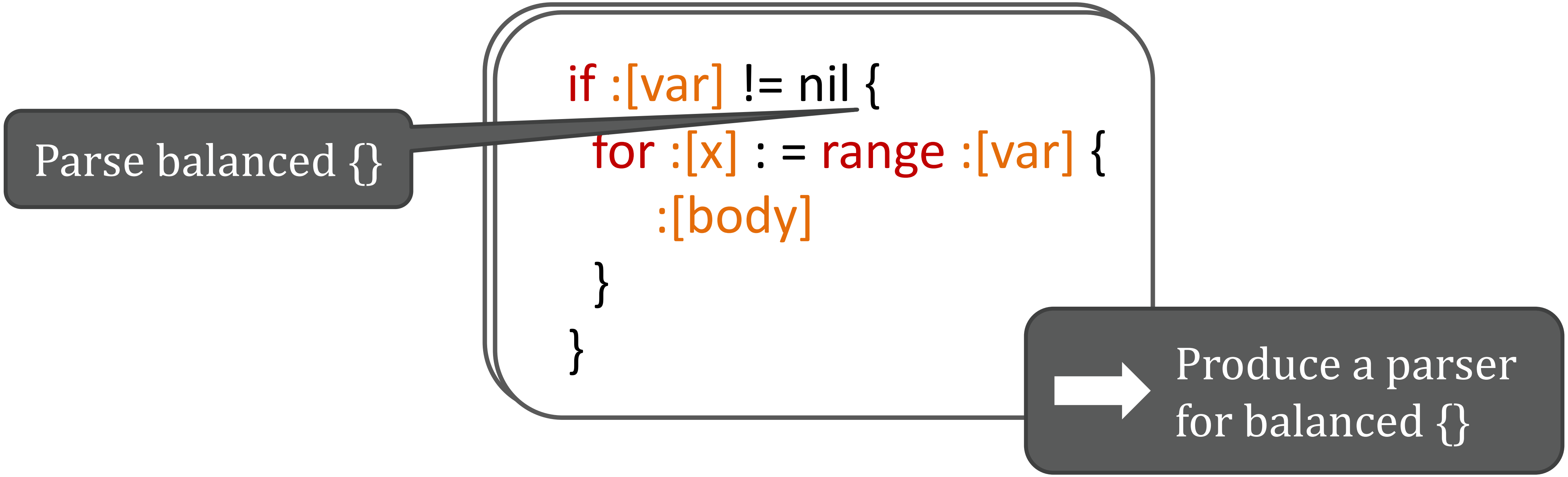
How to match declarative templates ⟷ source code?

Parse balanced {}

```
if :[var] != nil {
  for :[x] : = range :[var] {
      :[body]
  }
}
```

This defines a parser

How to match declarative templates ⟷ source code?



```
if :[var] != nil {
  for :[x] : = range :[var] {
      :[body]
  }
}
```

Parse balanced {}

→ Produce a parser for balanced {}

This defines a parser

How to match declarative templates ⟷ source code?

```
if :[var] != nil {
 for :[x] : = range :[var] {
     :[body]
 }
}
```

Chain all the parsers…

This defines a parser

How to match declarative templates ⟷ source code?

```
if :[var] != nil {
  for :[x] : = range :[var] {
      :[body]
  }
}
```

Chain all the parsers…

Parser Parser Combinators

Where is the complexity?

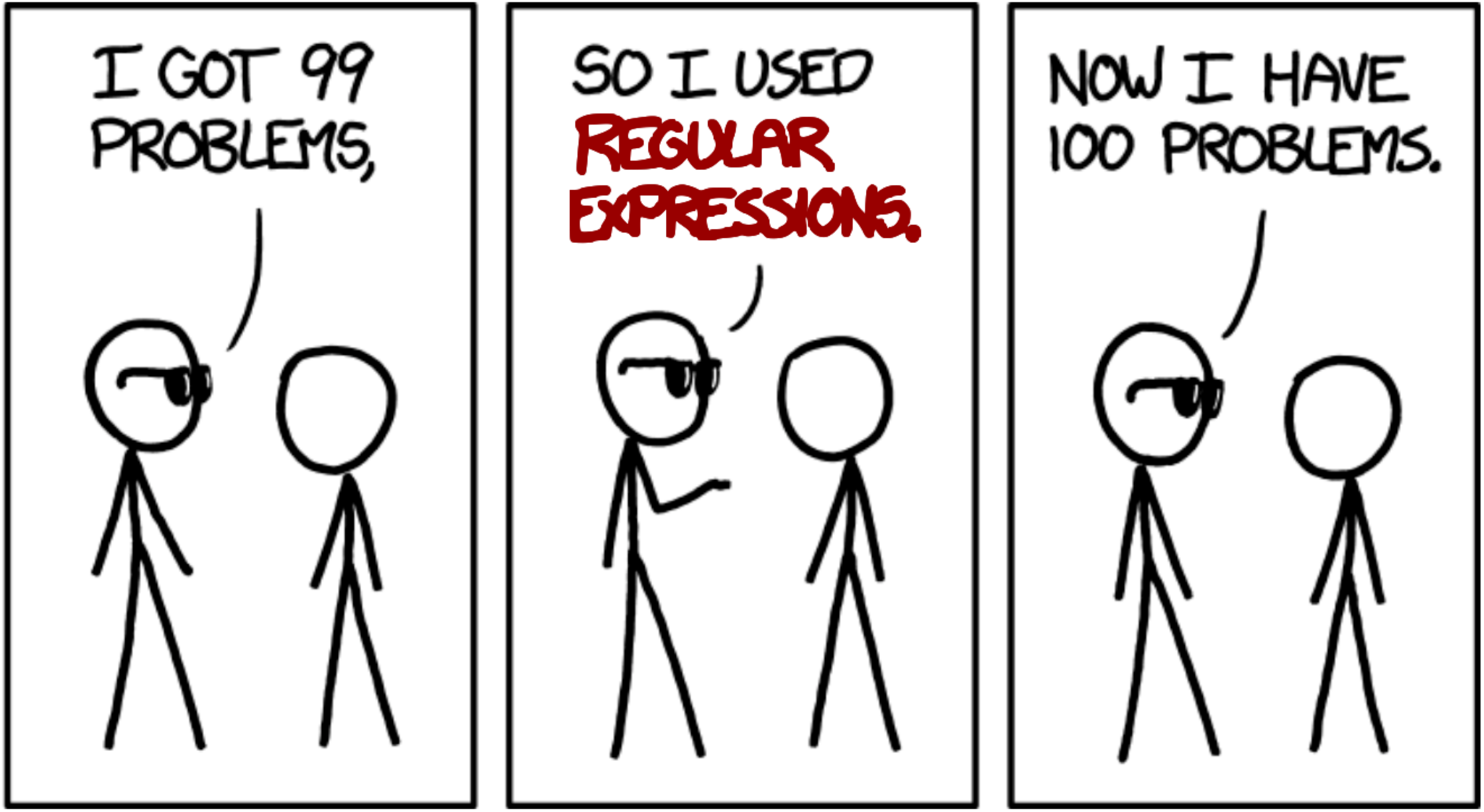
```
// 1) More comments more problems
printf(/* arg 1) */ "1) unbalanced \"parens\" (");
```

```
// 1) More comments more problems
printf(/* arg 1) */ "1) unbalanced \"parens\" (");
```

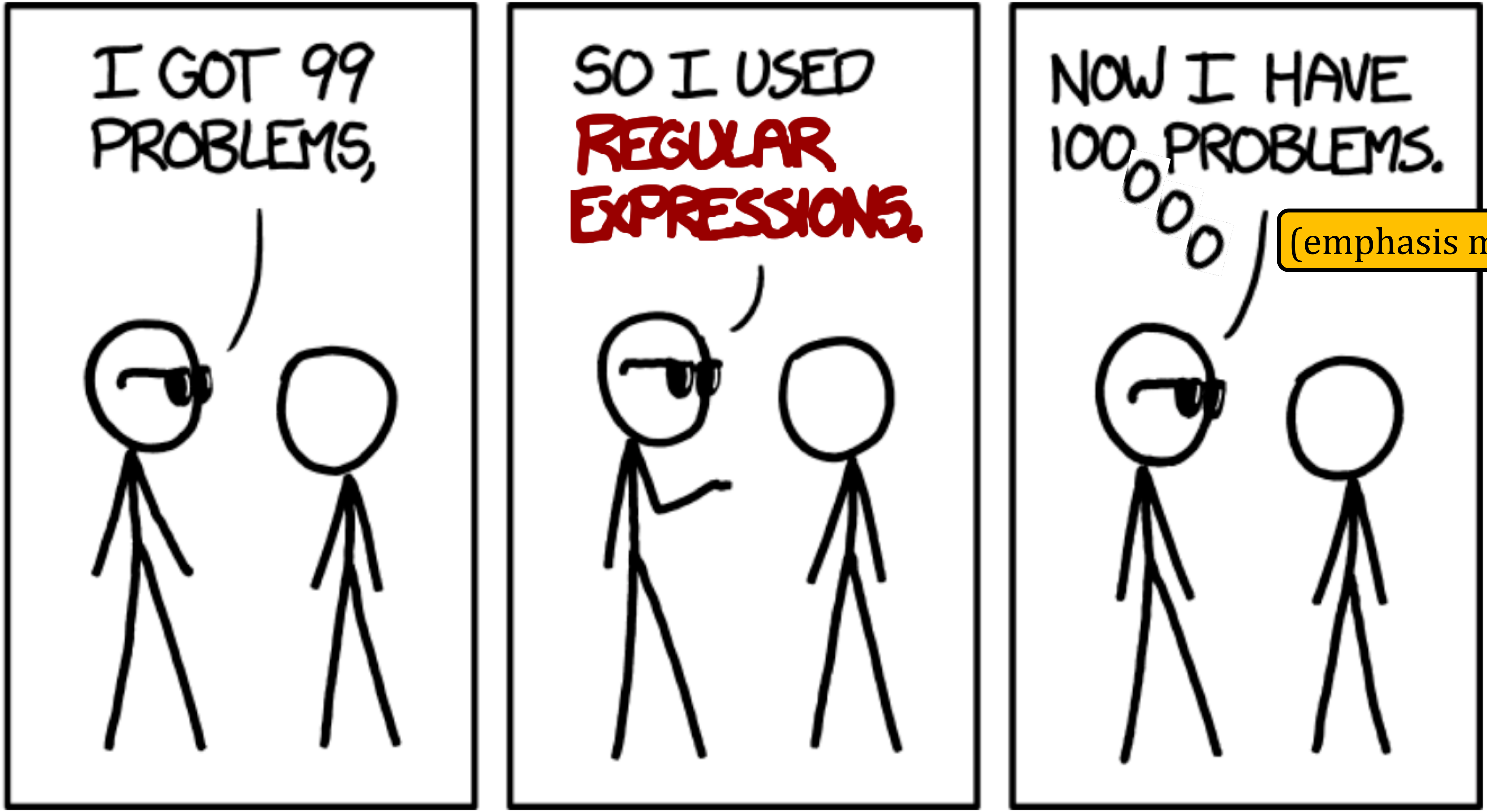
# in python

```
// 1) More comments more problems
printf(/* arg 1) */ "1) unbalanced \"parens\" (");
```

```
// 1) More comments more problems
printf(/* arg 1) */ "1) unbalanced \"parens\" (");
```



https://xkcd.com/1171/



(emphasis mine)

https://xkcd.com/1171/

See the link for demo video

https://drive.google.com/open?id=1EGhrBfxw_GQqeH5aWEFLfHSzYBKCjRyz

Small parsers handle syntax idiosyncracies across languages

Small parsers handle syntax idiosyncracies across languages

```
let user_defined_delimiters = [ "(" , ")"; "{" , "}"; "[" , "]"]

let string_literals = ["\""; "'"]

let raw_string_literals = ["`", "`"]

let comment_parser = [ Multiline ("/*" , "*/") ; Until_newline "//"]
```

Small parsers handle syntax idiosyncracies across languages

```
let user_defined_delimiters = [ "(" , ")"; "{" , "}"; "[" , "]"]

let string_literals = ["\""; "'"]

let raw_string_literals = ["`", "`"]

let comment_parser = [ Multiline ("/*" , "*/") ; Until_newline "//"]
```

And embed into a parser skeleton

Real world application

Large scale application

- Top 100 GitHub repos for 12 languages
  - 1,200 repos

Go, Dart, Julia, JS, Rust, Scala, Elm, OCaml, C, Clojure, Erlang, Python

- One to three rewrite rules per language

- 280 million lines of code parsed

- 42 minutes (20 cores)

Large scale application

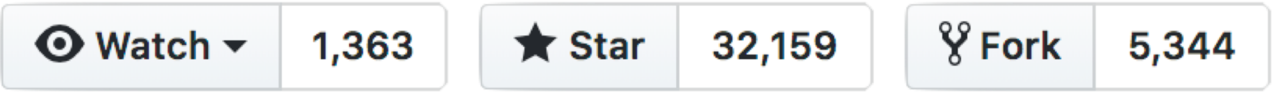
- Pull requests to 50 unique repositories
  - Merged ~40 PRs

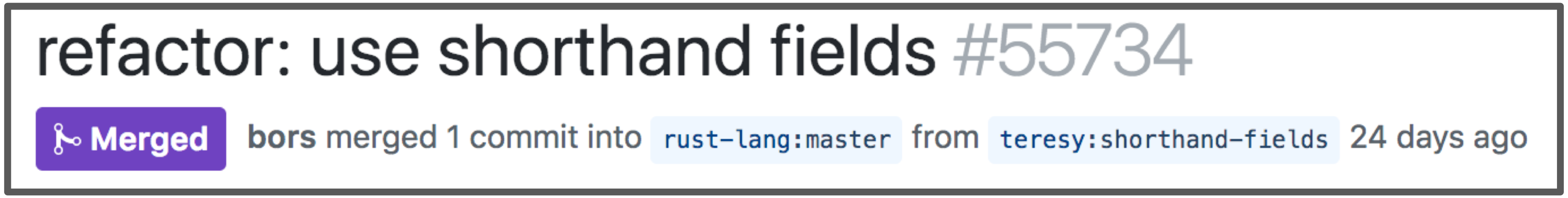
https://catalog.comby.dev/

https://github.com/squaresLab/pldi-artifact-2019/blob/master/PullRequests.md

Large scale application

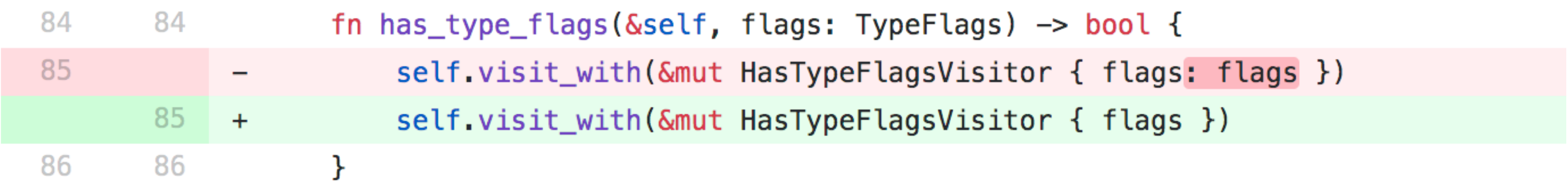
- Pull requests to 50 unique repositories
  - Merged ~40 PRs





refactor: use shorthand fields #55734

Merged bors merged 1 commit into rust-lang:master from teresy:shorthand-fields 24 days ago



```
     fn has_type_flags(&self, flags: TypeFlags) -> bool {
-        self.visit_with(&mut HasTypeFlagsVisitor { flags: flags })
+        self.visit_with(&mut HasTypeFlagsVisitor { flags })
     }
```

Demo: end-to-end with nested rewrite

See the link for demo video

https://drive.google.com/open?id=14UpdLtYA-2YD71AUDawt_zh0D_SCSZ7C

Summary

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 18, NO. 1, JANUARY 1992 19

Language Design For Program Manipulation

Eduardus A. T. Merks, J. Michael Dyck, and Robert D. Cameron, *Member, IEEE*

***Abstract*—The design of procedural and object-oriented programming languages is considered with respect to how easily programs written in those languages can be formally manipulated. Current procedural languages such as Pascal, Modula-2, and Ada generally support such program manipulations, except for some annoying anomalies and special cases. Three main areas of language design are identified as being of concern from a manipulation viewpoint—namely, the interface between concrete and abstract syntax, *the relationship between the abstract syntax* and static semantics (naming, scoping, and typing), and the ability to express basic transformations (folding and unfolding). Design principles are suggested so that problems identified for current languages can be avoided in the future.**

***Index Terms*— Language design, program manipulation, language environment, syntax, semantics.**

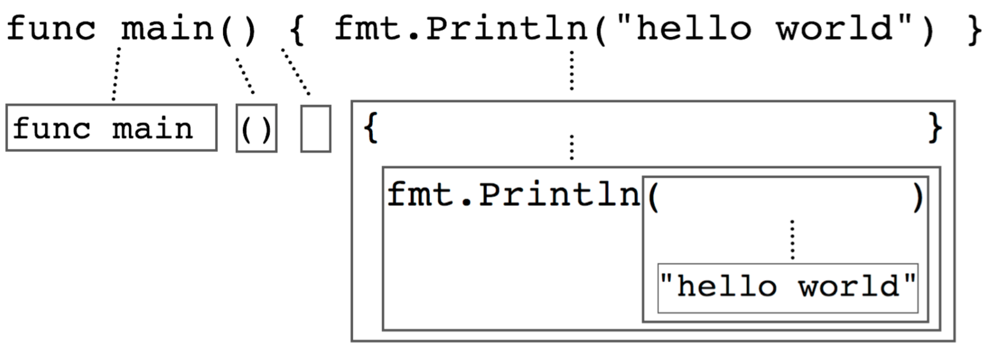
I. Introduction

PROGRAMS that manipulate other programs are becoming increasingly important in providing automated assistance for program development and maintenance. In particular, there has been long-standing interest in the concept of interactive program manipulation systems [1], including everything from program transformation systems [2] to language-based editors [3]–[5]. Furthermore, smaller scale program manipulation tools are also of interest, including program instrumenters [6], [7], program restructurers [8], program slicers [9], source-to-source translators [10], [11], and even program-specific manipulation tools that are designed to process a single (presumably large) program or class of programs [12], [13].

The ease of manipulating programs as data objects is strongly influenced by the nature of the *target language* in which those programs are written. Two programming languages could be equally expressive as notations for algorithms and data structures, but one might be much more conducive to [14], C [15], Modula-2 [16], Ada [17], Eiffel [18], and Modula-3 [19]. Unfortunately, little attention has been paid to manipulation issues in the design of such languages; consequently, they contain a number of avoidable problems in this regard. In contrast, manipulability has been of considerable importance in the design of modern functional languages such as HOPE [20] and Miranda [21], inspired by early work on program transformation [22]. Nevertheless, many useful program manipulations can be carried out in procedural languages. Furthermore, simple design changes for these languages could have considerably alleviated their limitations with respect to manipulation.

To an extent, appropriate language-processing technology can make up for deficiencies in target languages. For example, language-based editors generated by systems such as the Synthesizer Generator [23] or PSG [4] can easily deal with syntactic ambiguity by reference to incrementally maintained semantic attributes. For many other types of program-manipulation application, however, such technologies are either unavailable or inappropriate. In general, our viewpoint is *that there is little benefit in using* complex technology to solve manipulation problems when those problems could have been avoided altogether by careful language design.

The paper will proceed by proposing various principles of language design that are aimed at ensuring desirable manipulation properties. It is important to emphasize that these principles must be weighed carefully against other concerns that arise during language design, and certainly cannot be considered a recipe for success. Language design involves trade-offs between various desirable properties, and it is our aim in this paper to focus specifically on those that involve manipulation. The importance that a language designer places on the various properties will determine the relevance of the



```
func main() { fmt.Println("hello world") }
```

grammar ::= term EOF
term ::= '(' term ')' | '{' term '}' | '[' term ']' | term term | token
token ::= anything_else

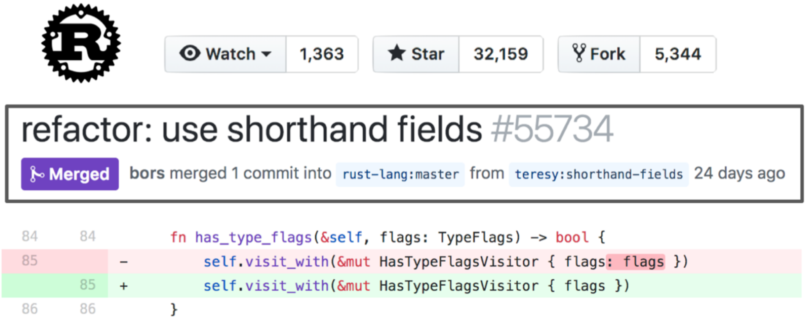
How to match declarative templates ⟷ source code?

```
if :[var] != nil {
  for :[x] : = range :[var] {
     :[body]
  }
}
```

This defines a parser

Large scale application

- Pull requests to 50 unique repositories
  - Merged ~40 PRs



refactor: use shorthand fields #55734

Merged bors merged 1 commit into `rust-lang:master` from `teresy:shorthand-fields` 24 days ago

```
        fn has_type_flags(&self, flags: TypeFlags) -> bool {
-           self.visit_with(&mut HasTypeFlagsVisitor { flags: flags })
+           self.visit_with(&mut HasTypeFlagsVisitor { flags })
        }
```

https://github.com/comby-tools/comby

@rvtond