

Semantic Analysis with Abstract Parsing

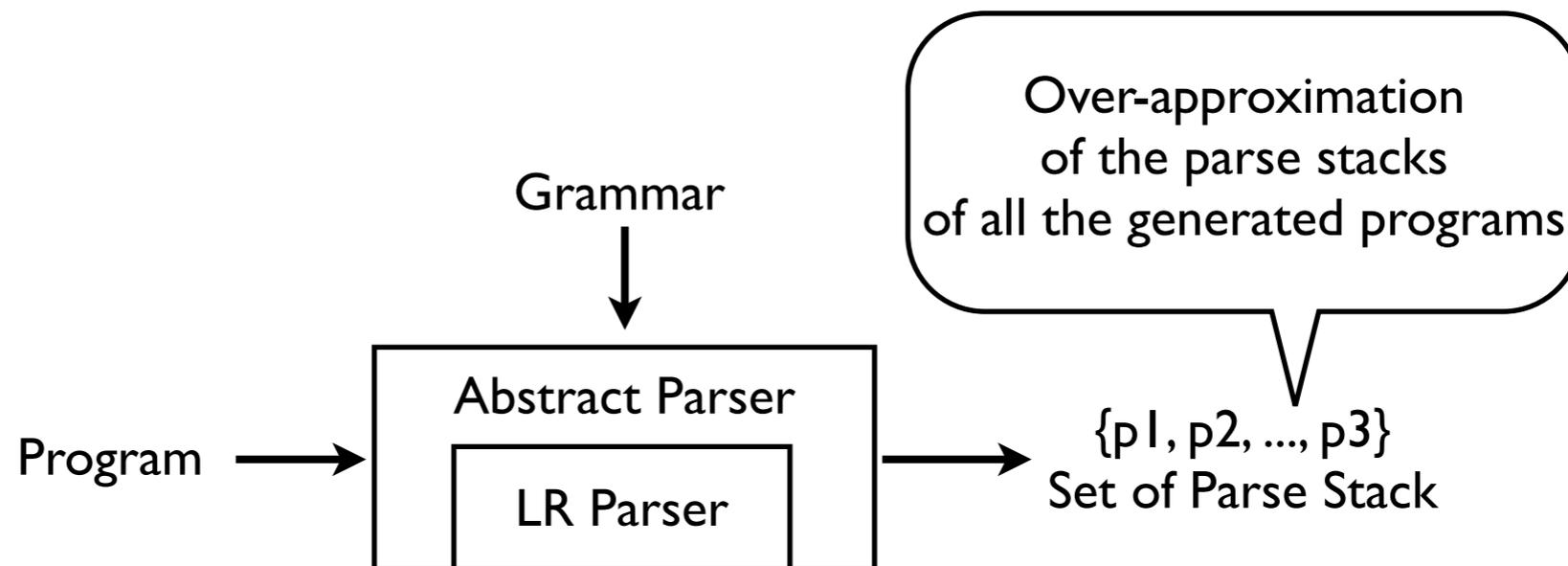
Soonho Kong
soon@ropas.snu.ac.kr

Wontae Choi
wtchoi@ropas.snu.ac.kr

2009/10/9
ROPAS Show & Tell

Abstract Parsing

- Powerful static string analysis technique by Doh, Kim, and Schmidt¹(2009)
- Syntax checking of generated strings
- Use LR parser as a component



1. Kyung-Goo Doh, Hyunha Kim, and David Schmidt. "Abstract parsing: static analysis of dynamically generated string output using LR-parsing technology." In Proceeding of the International Static Analysis Symposium, 2009.

Previous Work

GPCE'09

Abstract Parsing for Two-staged Languages with Concatenation *

Soonho Kong Wontae Choi Kwangkeun Yi
Seoul National University
{soon,wchoi,kwang}@ropas.snu.ac.kr

Abstract

This article, based on Doh, Kim, and Schmidt's "abstract parsing" technique, presents an abstract interpretation for statically checking the syntax of generated code in two-staged programs. Abstract parsing is a static analysis technique for two-staged programs of generated strings. We adopt this technique in the abstract interpretation framework and formulate it in the abstract domain of grammaring languages and analyze it as long as it satisfies the condition we provide. We also present an instance of the abstract domain, namely an abstract parse stack and its widening with k -cutting.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords Multi-staged languages, Program analysis, Abstract Interpretation, Parsing

1. Introduction

1.1 Motivation

For programs that generate and run programs during execution, statically checking the program safety is a challenge. We need to check the safety of generated programs as well as that of the immediate target program. Checking the safety must include checking the programs resulting from evaluating programs.

The semantic safety of such multi-staged programs can be achieved in part by a static type system as reported in [4, 15, 18, 25]. A sound static type system assures that program as data as well as the immediate target program will not have a type error during their executions.

In such a static type system, syntactic errors in the generated code are not an issue. The considered target language is such

*This work was supported by the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2009 and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF), (R11-2008-007-01002-0).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GPCE'09, October 4-5, 2009, Denver, Colorado, USA.
Copyright © 2009 ACM 978-1-60558-494-2/09/10...\$10.00

that primitive code fragments and their compositions are always syntactically correct.

In reality though (as in most web-programming or scripting languages such as PHP, Python, Ruby, Perl, and Javascript), if code is represented as a string and code composition is achieved by string concatenation, syntactically checking the code string value is the foremost issue in the static safety check of such multi-staged programs.

Recently, Doh, Kim, and Schmidt reported a powerful technique called "abstract parsing" [16] that statically analyzes the string values from programs. In abstract parsing, to statically check the generated string is to simulate the parsing actions for possible string values.

In this paper, we report a formalization of abstract parsing in the abstract interpretation framework [10, 11, 12] for two-staged programming languages. Our contribution is to lay a basis to expose the power and, if any, limitation of abstract parsing as static analysis for multi-staged languages.

1.2 Abstract Parsing

We here review the abstract parsing idea of [16]. Suppose we want to check that strings generated by the program in Figure 1 conform to the following grammar.

$$S \rightarrow a \mid [S]$$

Abstract parsing derives data-flow equations from the program as in Figure 1. The equation variables are treated as functions that map an input parse state to an output parse stack. They are solved using the goto controller of an LR parser for the grammar, shown in Figure 2.

Suppose we want to check that X_1 will accept strings of the target grammar. The analysis starts with $X_1(s_0)$ where s_0 is an initial parse state. To solve

$$X_1(s_0) = (L.X_0)(s_0),$$

the analysis first computes $L(s_0)$. With the state s_0 and the token "[", the goto controller returns $goto(s_0, [) = s_1$. Having $L(s_0) = ["$, the analysis computes $X_0(s_1)$. After consuming the token "a" and moving to the parse state s_2 , parser reduces with $S \rightarrow a$ and moves the parse state back to s_1 . Then $goto(s_1, S) = s_3$ yields

$$\begin{array}{l} x = 'a' \\ l = '[' \\ r = ']' \\ x = l \cdot x \end{array} \quad \begin{array}{l} X_0 = a \\ L = [\\ R =] \\ X_1 = L.X_0 \end{array}$$

Figure 1. Example program (left) and its data-flow equations (right)

1. Formalize/Generalize abstract parsing in the abstract interpretation framework
2. A correct and parameterized basis for its variants
3. For two-staged language with concatenation

Previous Work

PCC'09

PCC Framework for Program-Generators*
Soonho Kong Wontae Choi Kwangkeun Yi
Seoul National University
{soon,wtchoi,kwang}@ropas.snu.ac.kr

Abstract

In this paper, we propose a proof-carrying code framework for program-generators. The enabling technique is abstract parsing, a static string analysis technique, which is used as a component for generating and validating certificates. Our framework provides an efficient solution for certifying program-generators whose safety properties are expressed in terms of the grammar representing the generated program. The fixed-point solution of the analysis is generated and attached with the program-generator on the code producer side. The consumer receives the code with a fixed-point solution and validates that the received fixed point is indeed a fixed point of the received code. This validation can be done in a single pass.

1 Introduction

To certify the safety of a mobile program-generator, we need to ensure not only the safe execution of the generator itself but also that of the generated programs. Safety properties of the generated programs are specified efficiently in terms of the grammar representing the generated programs. For instance, the safety property "generated programs should not have nested loops" can be specified and verified by the reference grammar for the generated programs.

Recently, Doh, Kim, and Schmidt presented a powerful static string analysis technique called abstract parsing [4]. Using LR parsing as a component, abstract parsing analyzes the program and determines whether the strings generated in the program conform to the given grammar or not.

In this paper, we propose a Proof-Carrying Code (PCC) framework [8, 9] for program-generators. We adapt abstract parsing to check the generated programs, the code producer abstract-parses the program specifying the safety property of the generated programs, the code producer sends the program-generator with the computed fixed-point solution as a certificate. The code producer sends the program-generator accompanied with the fixed-point solution. The code consumer receives the program-generator and computes a fixed-point solution. The code consumer receives the program-generator with the computed fixed-point solution and validates that the received fixed point is indeed the solution for the received program-generator. Our framework can be seen as an abstraction-carrying code framework [1, 5] specialized to program-generators which is modeled by a two-staged language with concatenation.

This work is, to our knowledge, the first to present a proof-carrying code framework that certifies grammatical properties of the generated programs. Directly computing the parse stack information as a form of the fixed-point solution, abstract parsing provides an efficient way to validate the certificates on the code consumer side. In contrast to abstract parsing, the previous static string analysis techniques [3, 7, 2] approximate the possible values of a string expression of the program with a grammar and see whether the approximated grammar is included in the reference grammar. This grammar inclusion check takes too much time and makes those techniques difficult to be used as a validation component of a PCC framework.

*This work was supported by the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2009 and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), (R11-2008-007-01002-0).

Building a Proof-Carrying Code framework using abstract parsing

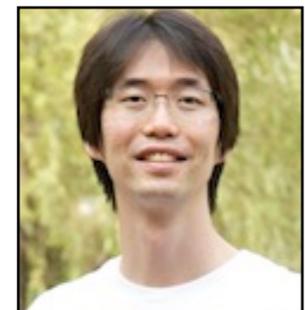
Motivation



Naoki Kobayashi

Q: Possible to check
semantic property?

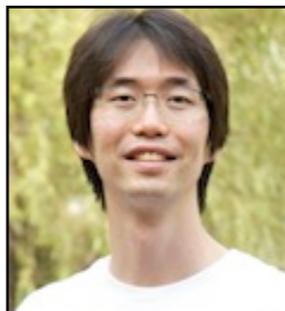
A: Not now.
Only support syntactic property.



Soonho Kong

Motivation

Can we extend **abstract parsing**
to check **semantic** property?



Soonho Kong



Wontae Choi

Language

- Two-staged language with concatenation

Syntax

$$e \in \text{Exp} ::= x \mid \text{let } x \ e_1 \ e_2 \mid \text{or } e_1 \ e_2 \mid \text{re } x \ e_1 \ e_2 \ e_3 \mid ' f$$

$$f \in \text{Frag} ::= x \mid \text{let} \mid \text{or} \mid \text{re} \mid (\mid) \mid f_1.f_2 \mid , e$$

Operational Semantics

$\sigma \vdash^0 e \Rightarrow v$			$\sigma \vdash^1 f \Rightarrow v$		
$\frac{}{\sigma \vdash^0 x \Rightarrow \sigma(x)}$	(variable)	$\frac{}{\sigma \vdash^1 x \Rightarrow x}$		$\frac{}{\sigma \vdash^1 \text{let} \Rightarrow \text{let}}$	(token)
$\frac{\sigma \vdash^0 e_1 \Rightarrow v \quad \sigma[x \mapsto v] \vdash^0 e_2 \Rightarrow v'}{\sigma \vdash^0 \text{let } x \ e_1 \ e_2 \Rightarrow v'}$	(let binding)	$\frac{}{\sigma \vdash^1 \text{or} \Rightarrow \text{or}}$		$\frac{}{\sigma \vdash^1 \text{re} \Rightarrow \text{re}}$	
$\frac{\sigma \vdash^0 e_1 \Rightarrow v}{\sigma \vdash^0 \text{or } e_1 \ e_2 \Rightarrow v} \quad \frac{\sigma \vdash^0 e_2 \Rightarrow v}{\sigma \vdash^0 \text{or } e_1 \ e_2 \Rightarrow v}$	(branch)	$\frac{}{\sigma \vdash^1 (\Rightarrow (}$		$\frac{}{\sigma \vdash^1) \Rightarrow)}$	
$\frac{\sigma \vdash^0 e_1 \Rightarrow v \quad \sigma[x \mapsto v] \vdash^0 \text{loop } x \ e_2 \ e_3 \Rightarrow v'}{\sigma \vdash^0 \text{re } x \ e_1 \ e_2 \ e_3 \Rightarrow v'}$	(loop)	$\frac{\sigma \vdash^1 f_1 \Rightarrow v_1 \quad \sigma \vdash^1 f_2 \Rightarrow v_2}{\sigma \vdash^1 f_1.f_2 \Rightarrow v_1 v_2}$			(concatenation)
$\frac{\sigma \vdash^0 e_2 \Rightarrow v \quad \sigma[x \mapsto v] \vdash^0 \text{loop } x \ e_2 \ e_3 \Rightarrow v'}{\sigma \vdash^0 \text{loop } x \ e_2 \ e_3 \Rightarrow v'}$				$\frac{\sigma \vdash^0 e \Rightarrow v}{\sigma \vdash^1 , e \Rightarrow v}$	(comma)
$\frac{\sigma \vdash^0 e_3 \Rightarrow v}{\sigma \vdash^0 \text{loop } x \ e_2 \ e_3 \Rightarrow v}$					
$\frac{\sigma \vdash^1 f \Rightarrow v}{\sigma \vdash^0 ' f \Rightarrow v}$	(back quote)				

Language

Example

x is initialized with a

Loop body is not executed.

```
re x 'a
```

```
  'or . ,x
```

```
  ',x . b
```

value is a b

=> a b

Language

Example

```
re x 'a
```

x is initialized with a

```
'or . ,x
```

x is or a

```
' ,x . b
```

value is or a b

=> or a b

Loop body is executed once

Language

Example

```
re x 'a
```

x is initialized with a

```
'or . ,x
```

x is or or a

```
' ,x . b
```

value is or or a b

Loop body is executed twice

=> or or a b

Language

Example

re x 'a

'or . ,x

' ,x . b

This program possibly generates one of the followings:

a b

or a b

or or a b

or or or a b

...

Only this one is
syntactically correct

This program is possible to generate syntactically incorrect code.

Language

Example

```
re x 'a
```

```
  'or . ,x
```

```
  ',x . b
```

This program possibly generates one of the following code:

```
a b
```

```
or a b
```

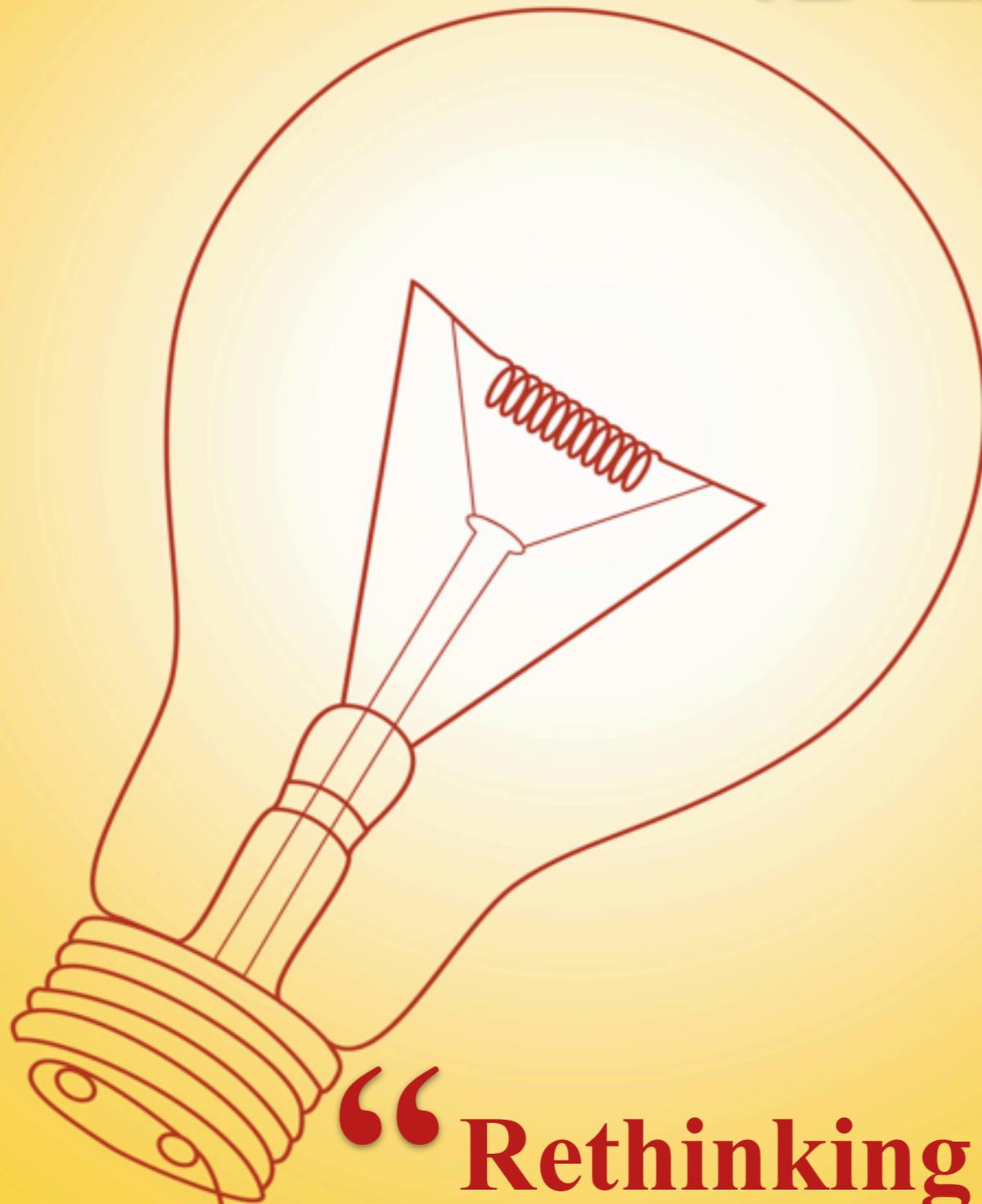
```
or or a b
```

```
or or or a b
```

```
...
```

$$\llbracket \text{re } x \text{ 'a ('or . ,x) (',x . b)} \rrbracket^0 \{ \sigma_0 \}$$
$$= \underline{\{ a b, \text{ or } a b, \text{ or or } a b, \text{ or or or } a b, \dots \}}$$

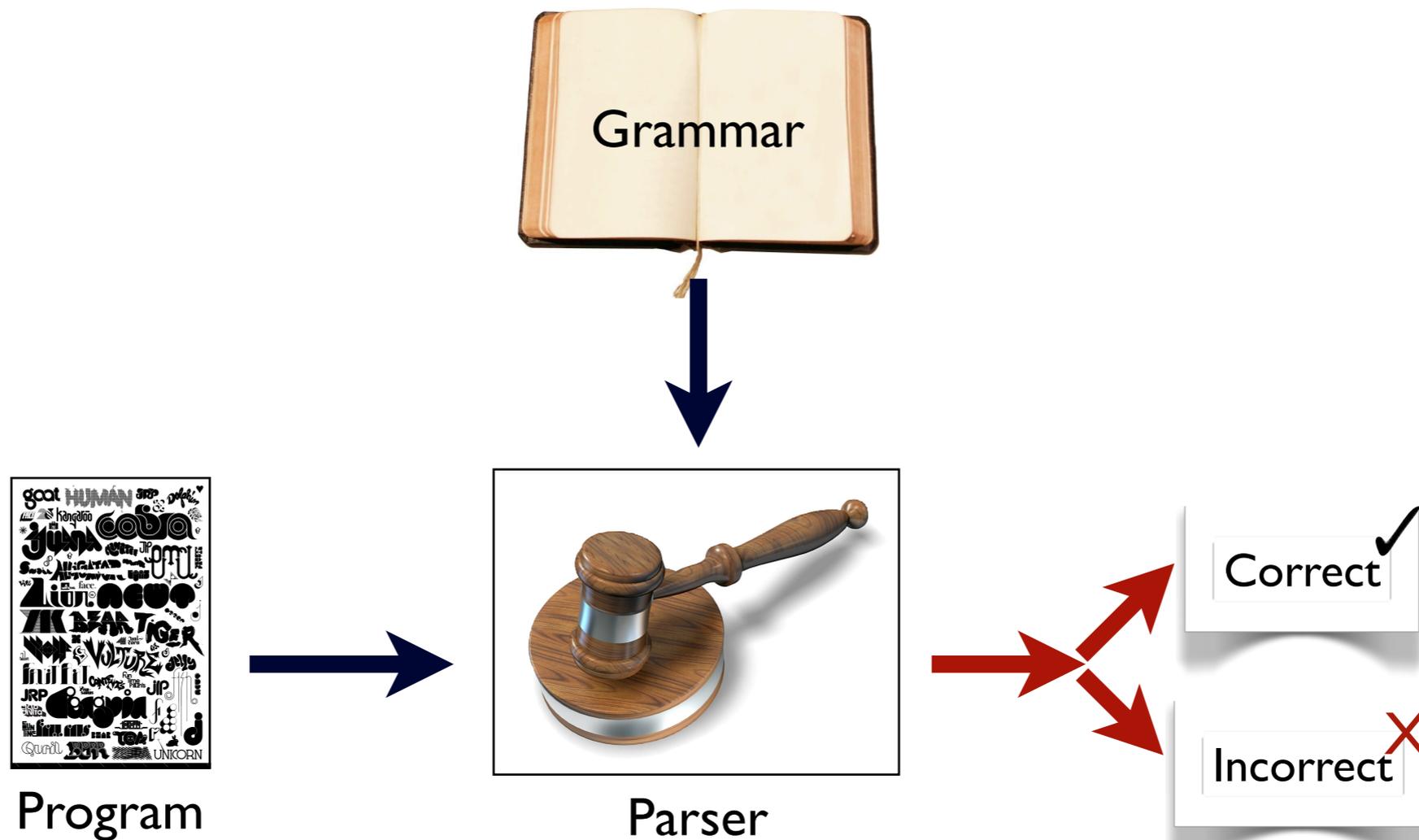
IDEA



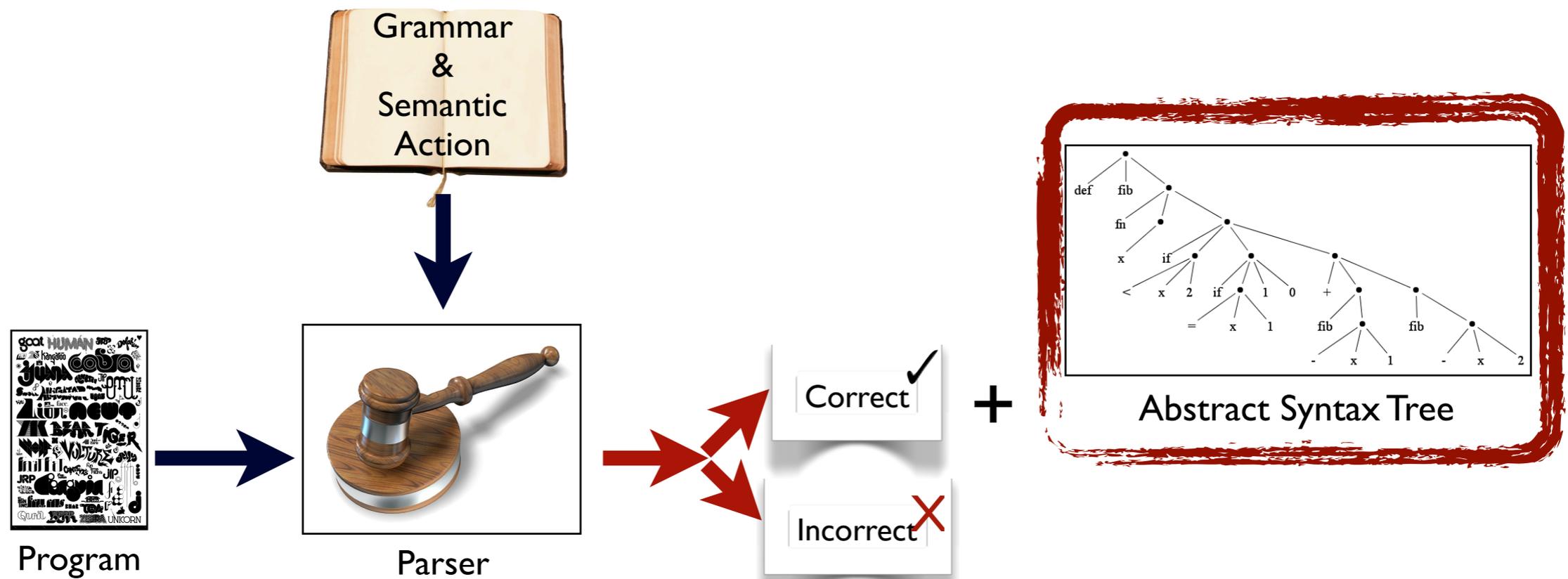
**“ Rethinking
what we get by parsing! ”**



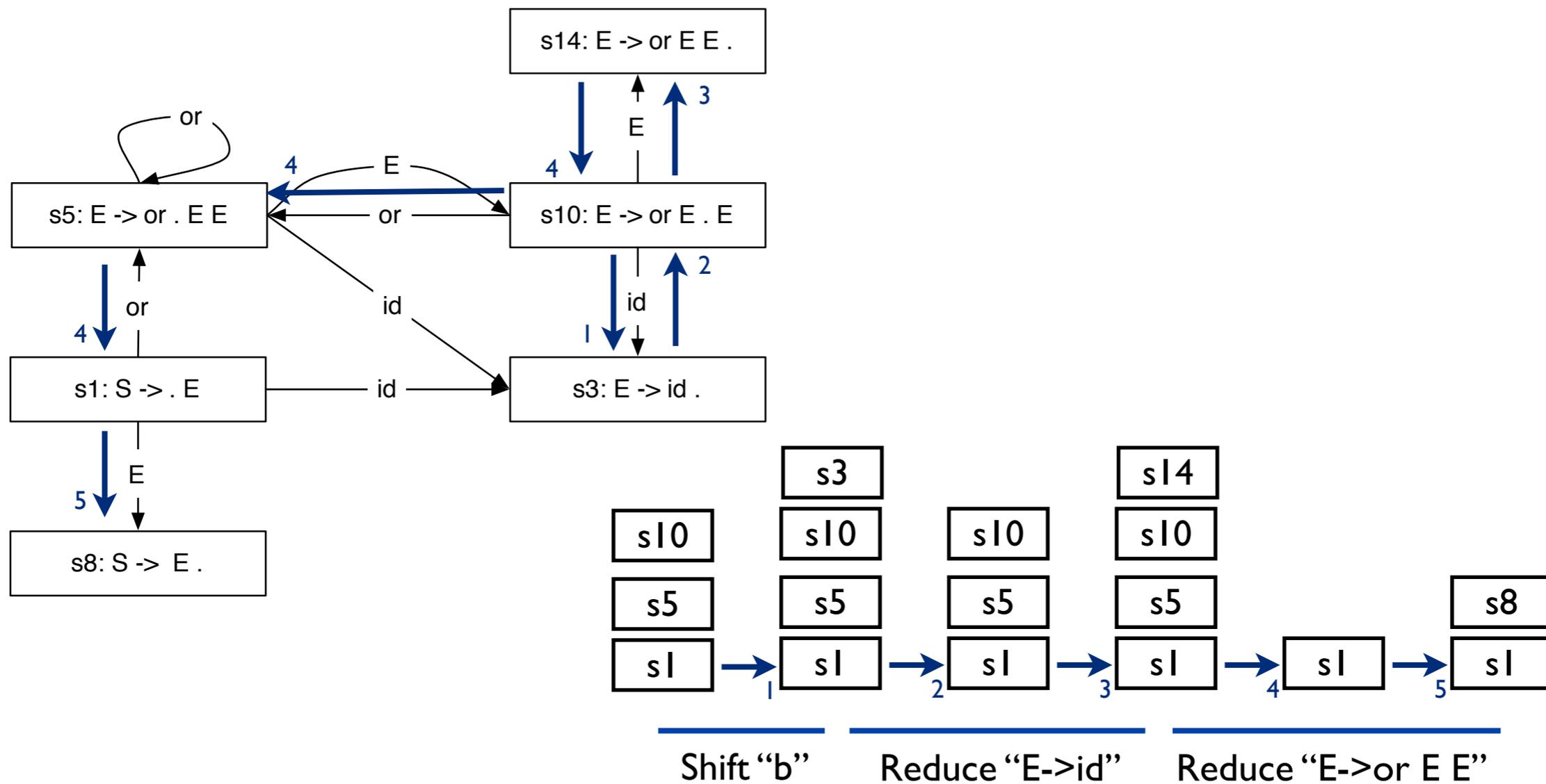
Parsing as a Decision Procedure



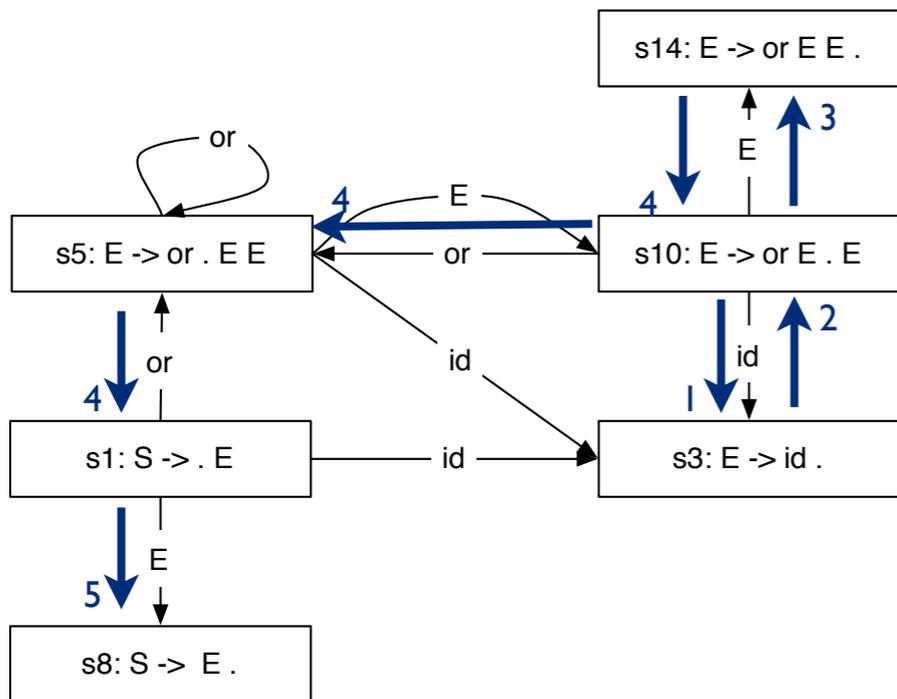
Parsing with Semantic Action



Example: Parsing "or a . b"

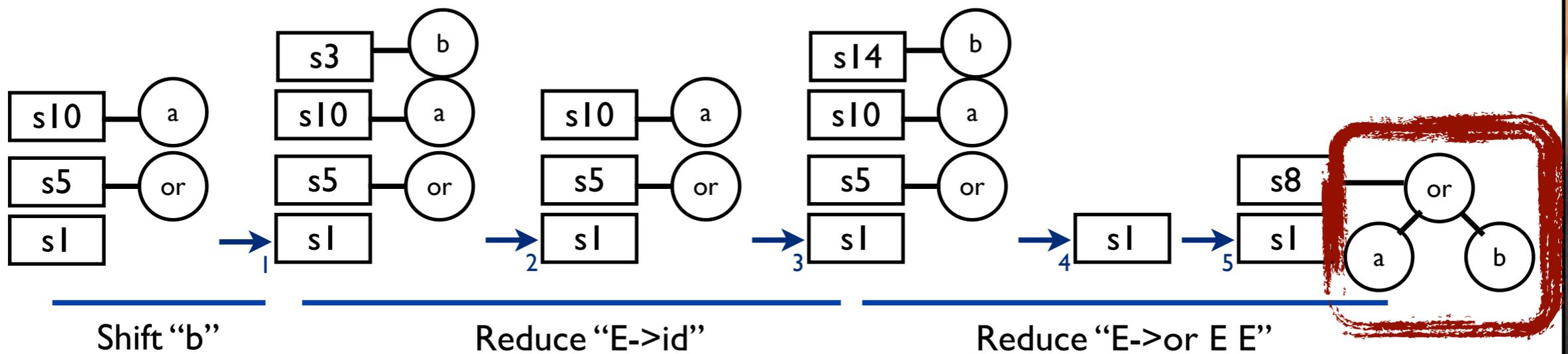


Example: Parsing "or a b"



Semantic Action

$E: OR\ e\ e\ \{ OR(\$2, \$3) \}$



Observation

- Semantic action allows us to construct AST
- Is it possible to construct other things?

Observation

- Semantic action allows us to construct AST
- Is it possible to construct other things?



Yes!

Observation

- Semantic action allows us to construct AST
- Is it possible to construct other things?



Yes!

“as long as it is **compositionally** constructive”

Observation

- Semantic action allows us to construct AST
- Is it possible to construct other things?

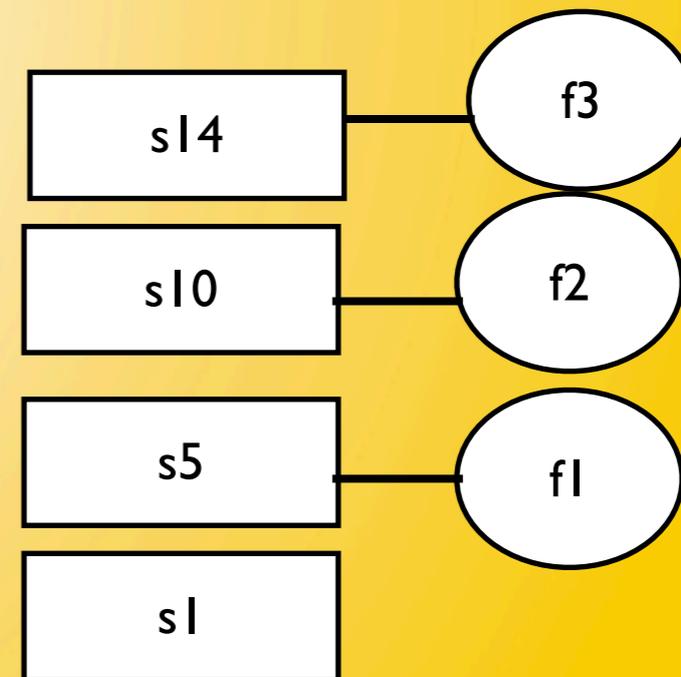
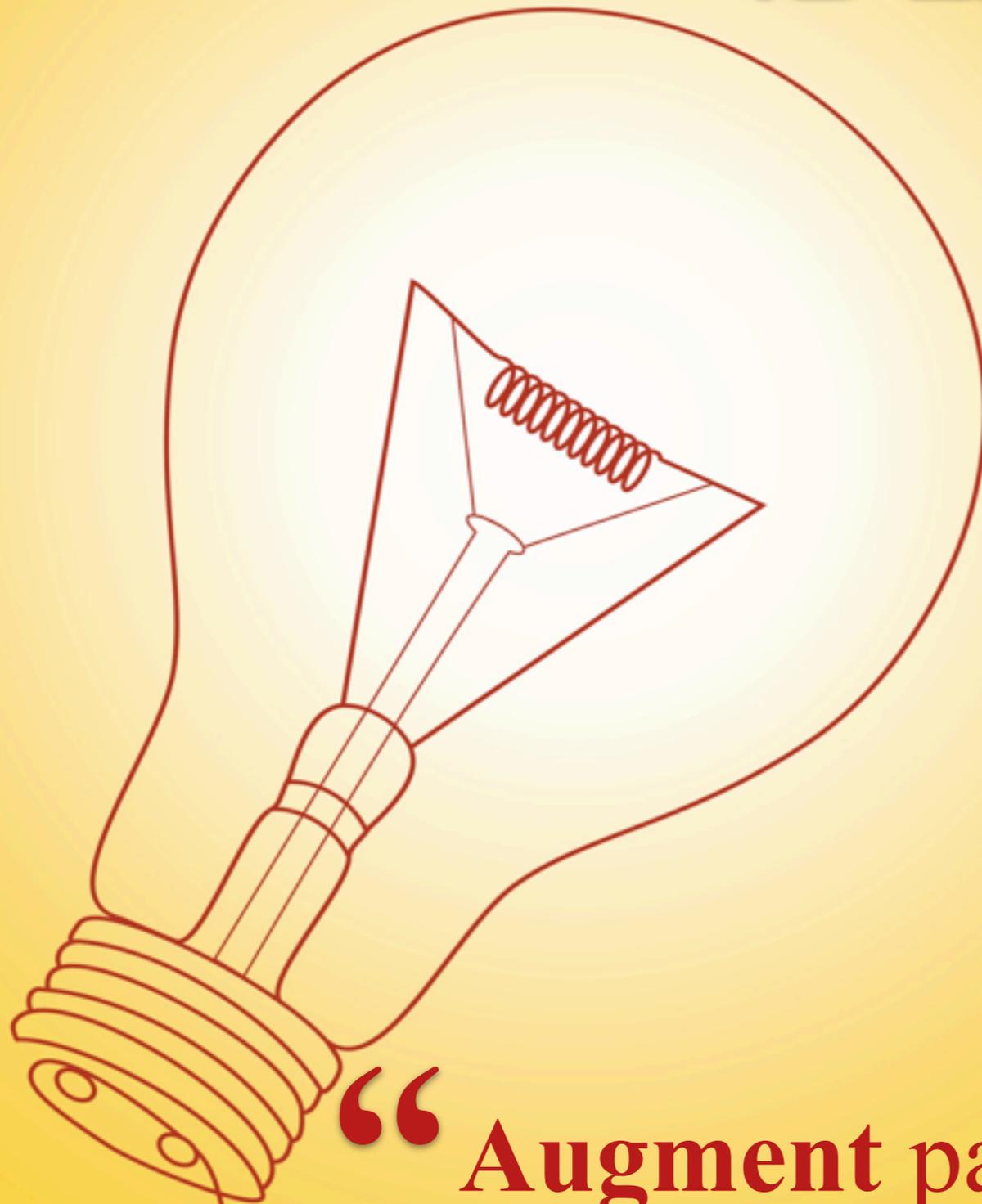
Yes!

“as long as it

Semantics!

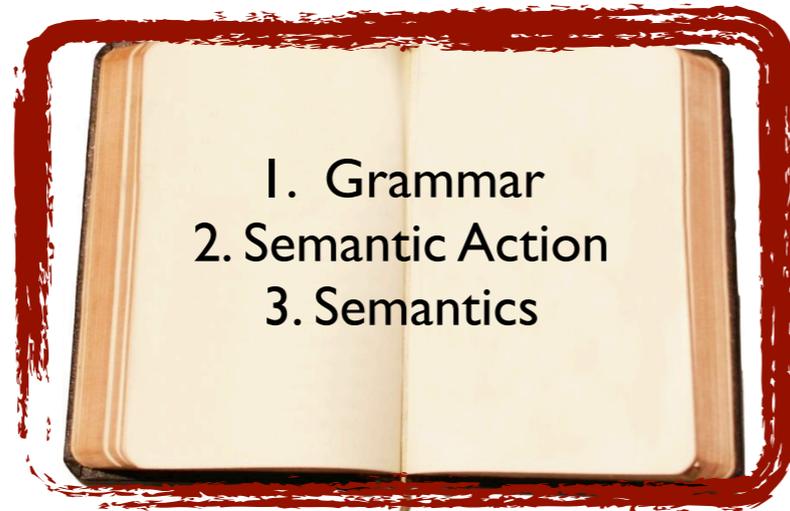
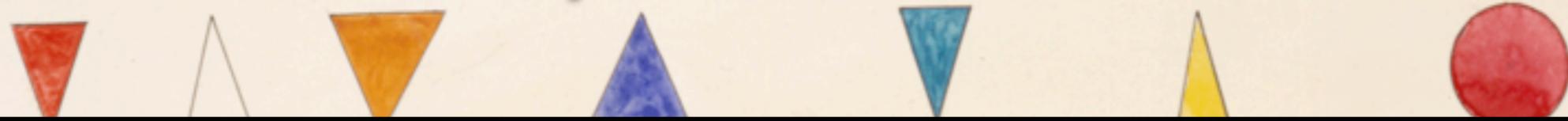
is **not** constructive”

IDEA



“Augment parse state with semantic function!”

Semantic Analysis with Abstract Parsing



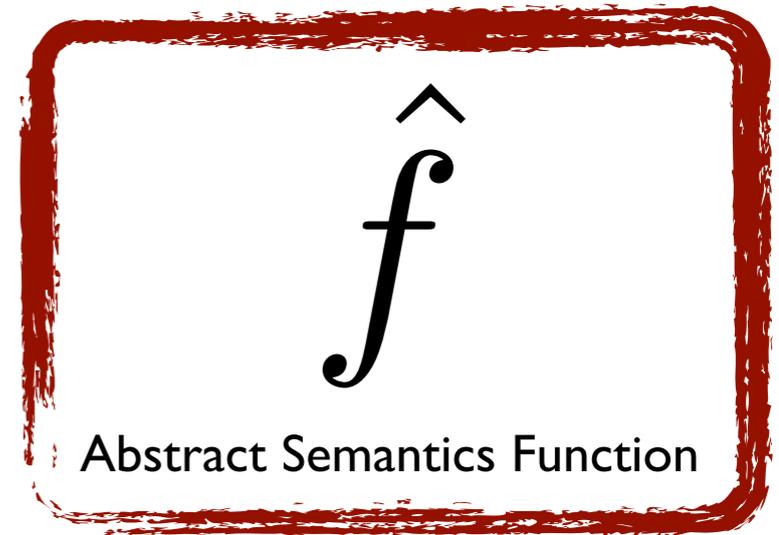
Program



Correct ✓

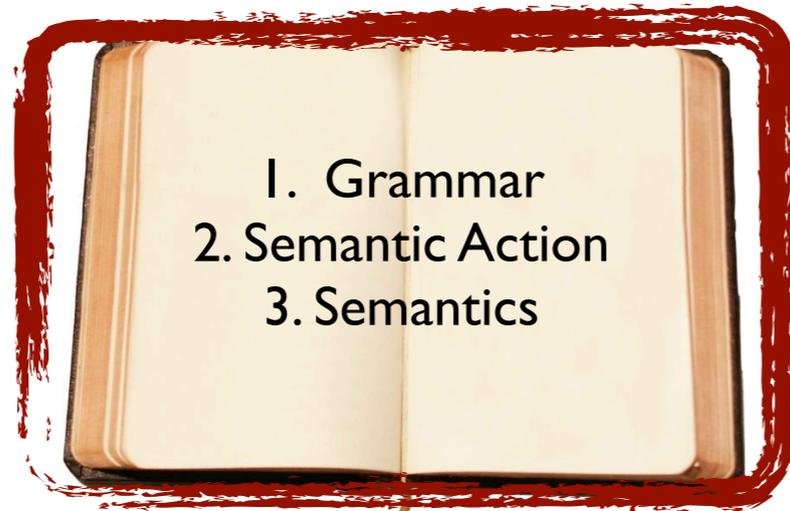
Incorrect ✗

+



Semantic Analysis with Abstract Parsing

Big Picture

- 
1. Grammar
 2. Semantic Action
 3. Semantics



Program



Correct ✓

Incorrect ✗

+



Abstract Semantics Function

Example

Unused Variable Analysis

$$\llbracket \text{or } e_1 \ e_2 \rrbracket = \llbracket e_1 \rrbracket \cap \llbracket e_2 \rrbracket$$

$$\llbracket \text{let } x \ e_1 \ e_2 \rrbracket = \llbracket e_1 \rrbracket \cap \llbracket e_2 \rrbracket$$

$$\llbracket x \rrbracket = U - \{x\}$$

$$\llbracket \text{re } x \ e_1 \ e_2 \ e_3 \rrbracket = \llbracket e_1 \rrbracket \cap \llbracket e_2 \rrbracket \cap \llbracket e_3 \rrbracket$$

Example

Unused Variable Analysis

```
let p1 'let . x . a
let p2 'let . y . b
      ',p1 . ,p2 . or . x . x

=> let x a
    let y b
    or x x
```

Example

Unused Variable Analysis

```
let p1 'let . x . a
let p2 'let . y . b
      ',p1 . ,p2 . or . x . x
```

```
=> let x a
    let y b
      or x x
```

$$U = \{x, y, a, b\}$$

$$[\text{or } e_1 e_2] = [e_1] \cap [e_2]$$

$$[\text{let } x e_1 e_2] = [e_1] \cap [e_2]$$

$$[x] = U - \{x\}$$

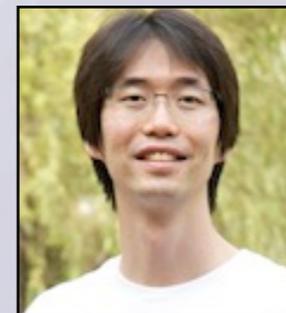
$$[\text{re } x e_1 e_2 e_3] = [e_1] \cap [e_2] \cap [e_3]$$

Conclusion



Naoki Kobayashi

Q: Possible to check
semantic property?



Soonho Kong



Wontae Choi

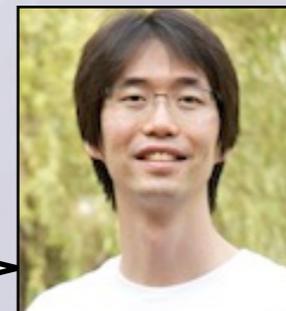
Conclusion



Naoki Kobayashi

Q: Possible to check
semantic property?

A: **Yes, we can!**



Soonho Kong



Wontae Choi

Conclusion

- **Semantic analysis** is possible with **abstract parsing**
- By augmenting parse stack
 - Semantic function for each parse state
- Work in Progress
 - Formalize the idea
 - Find out more examples

Thank you

