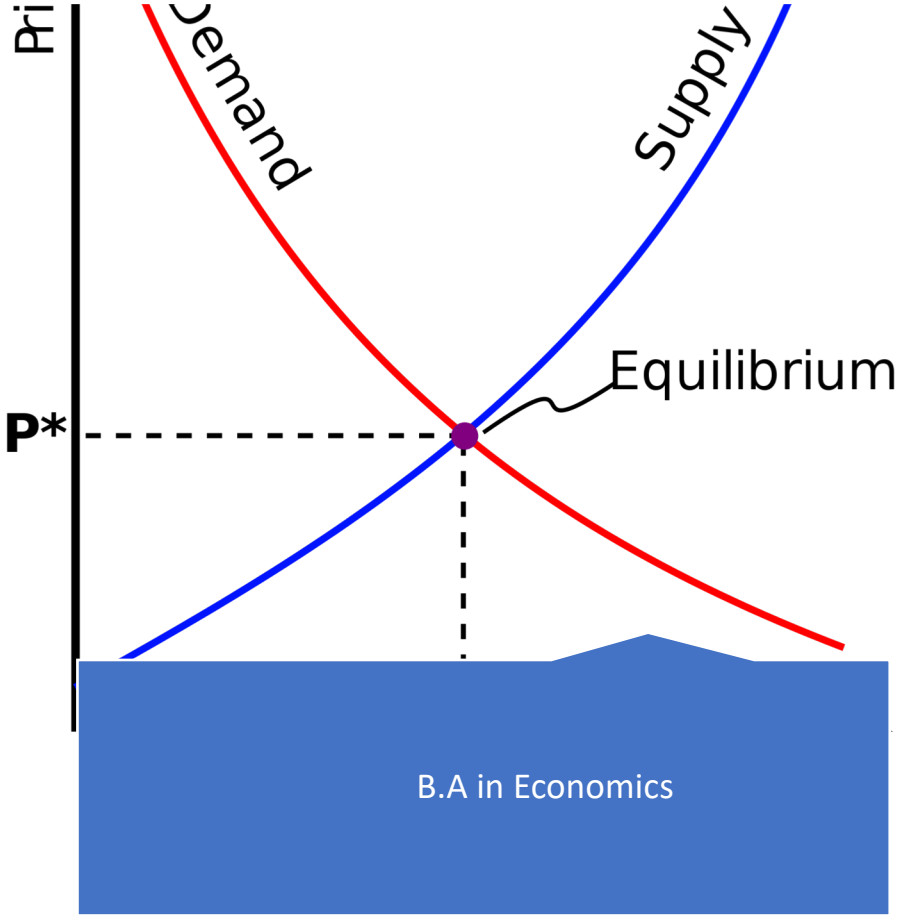
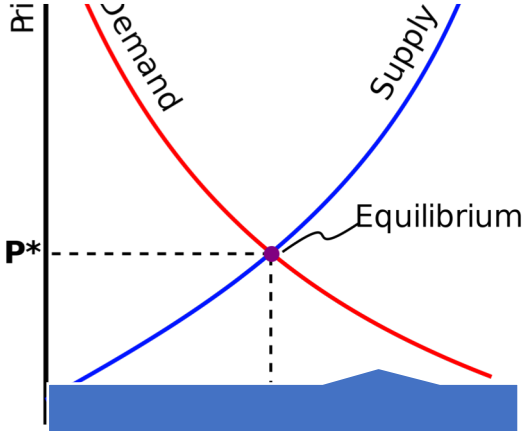
Minimizing Technical Barriers to Learning Programming

Martin Velez, Ph.D. Candidate
University of California, Davis
Galois Tech Talk
August 13, 2018

About Me



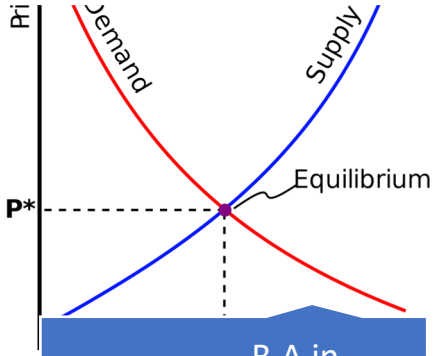
About Me



B.A in Economics

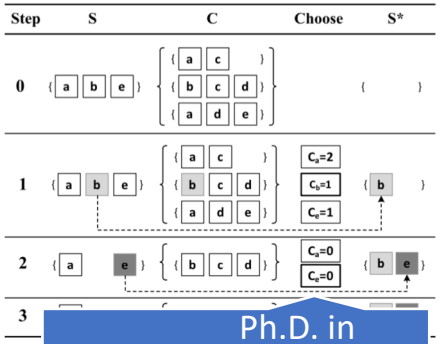
Software Engineer

About Me



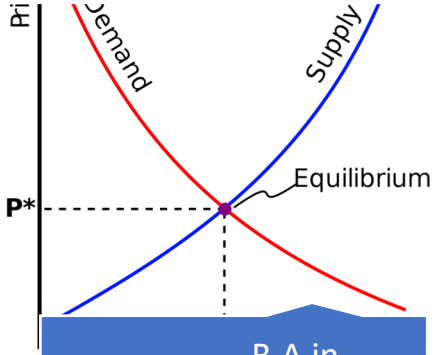
B.A in Economics

Software Engineer



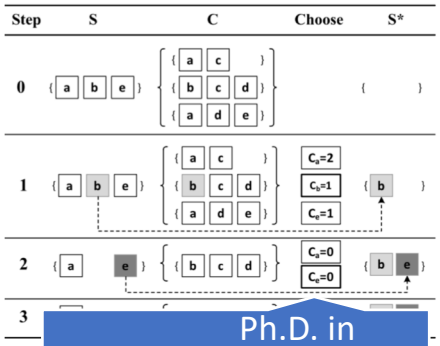
Ph.D. in Computer Science

About Me

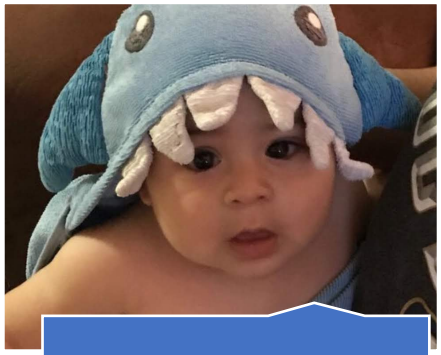


B.A in Economics

Software Engineer

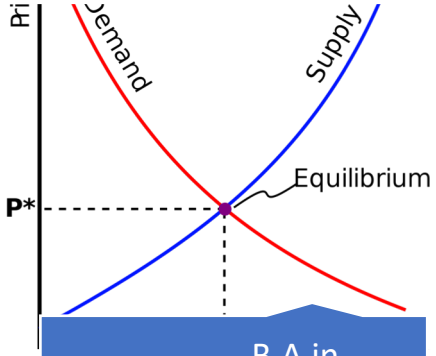


Ph.D. in Computer Science



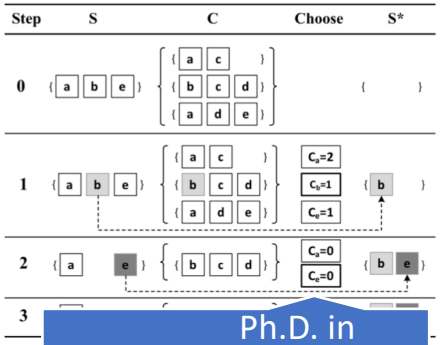
Father

About Me

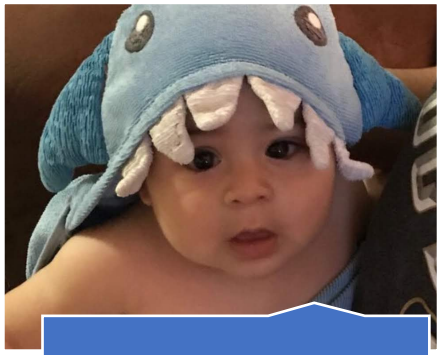


B.A in Economics

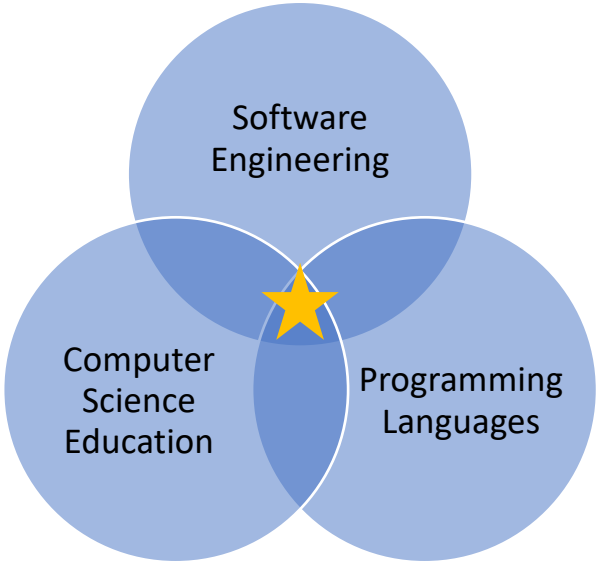
Software Engineer



Ph.D. in Computer Science

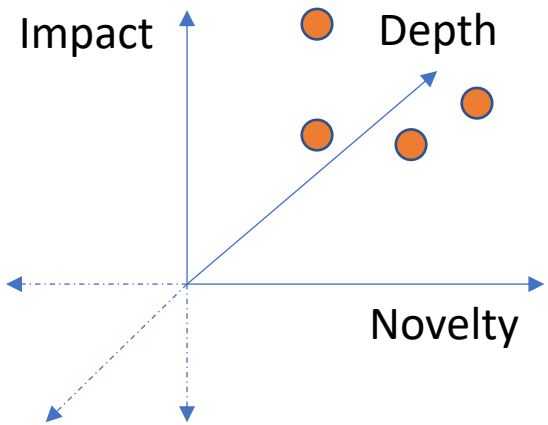


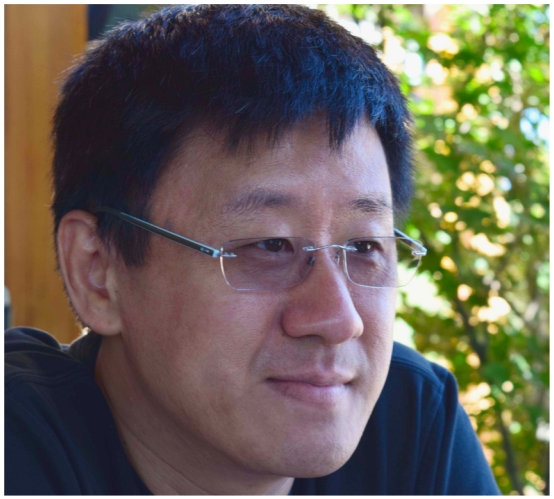
Father



Lessons

Research

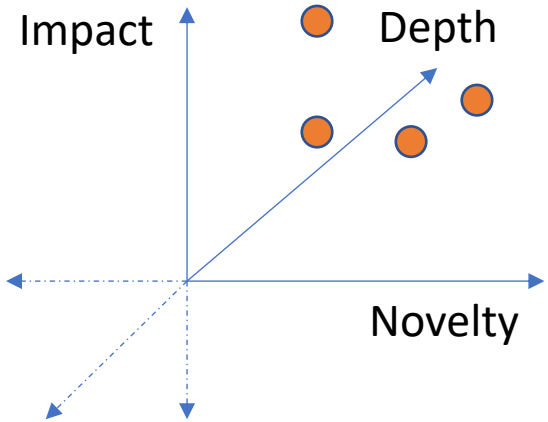




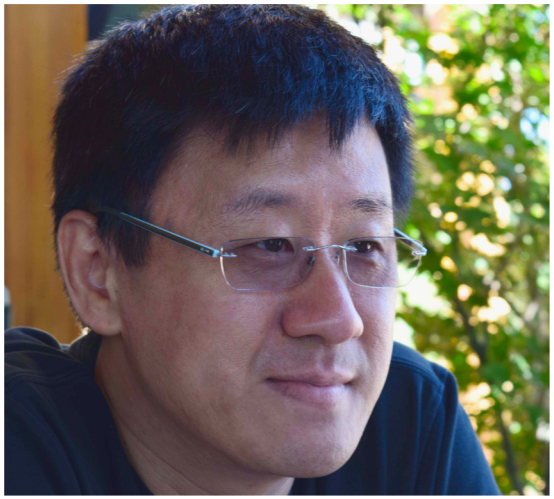
Lessons

Research

Good Research

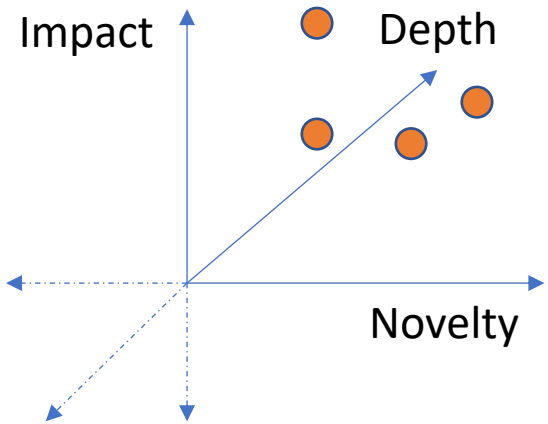


High in at least one dimension.



Lessons

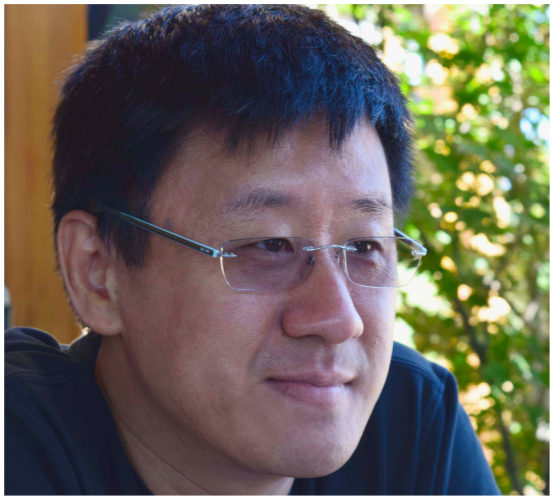
Research



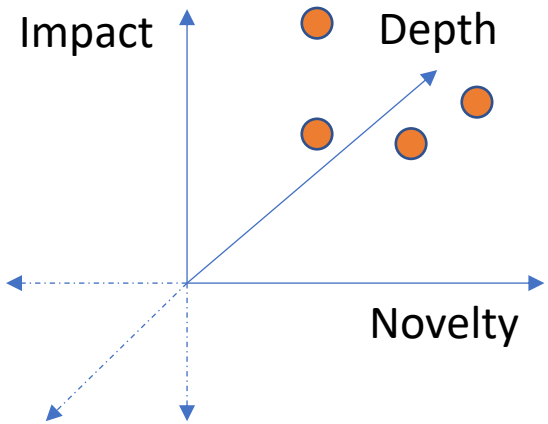
Good Research

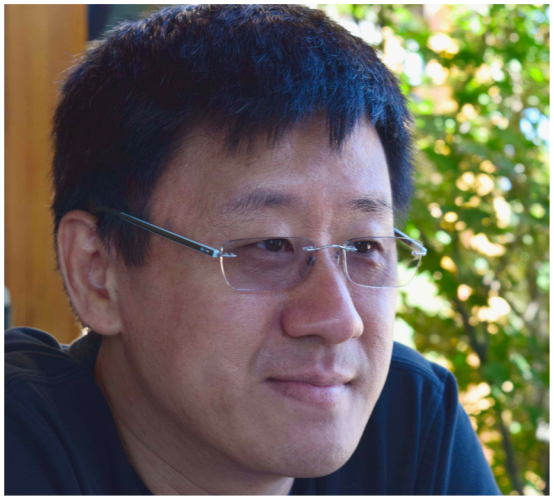
High in at least one dimension.

Alternative: It inspires you, good or bad.



Lessons

Research	Good Research	Your Own Research
	High in at least one dimension. Alternative: It inspires you, good or bad.	First, it must **matter** to you.



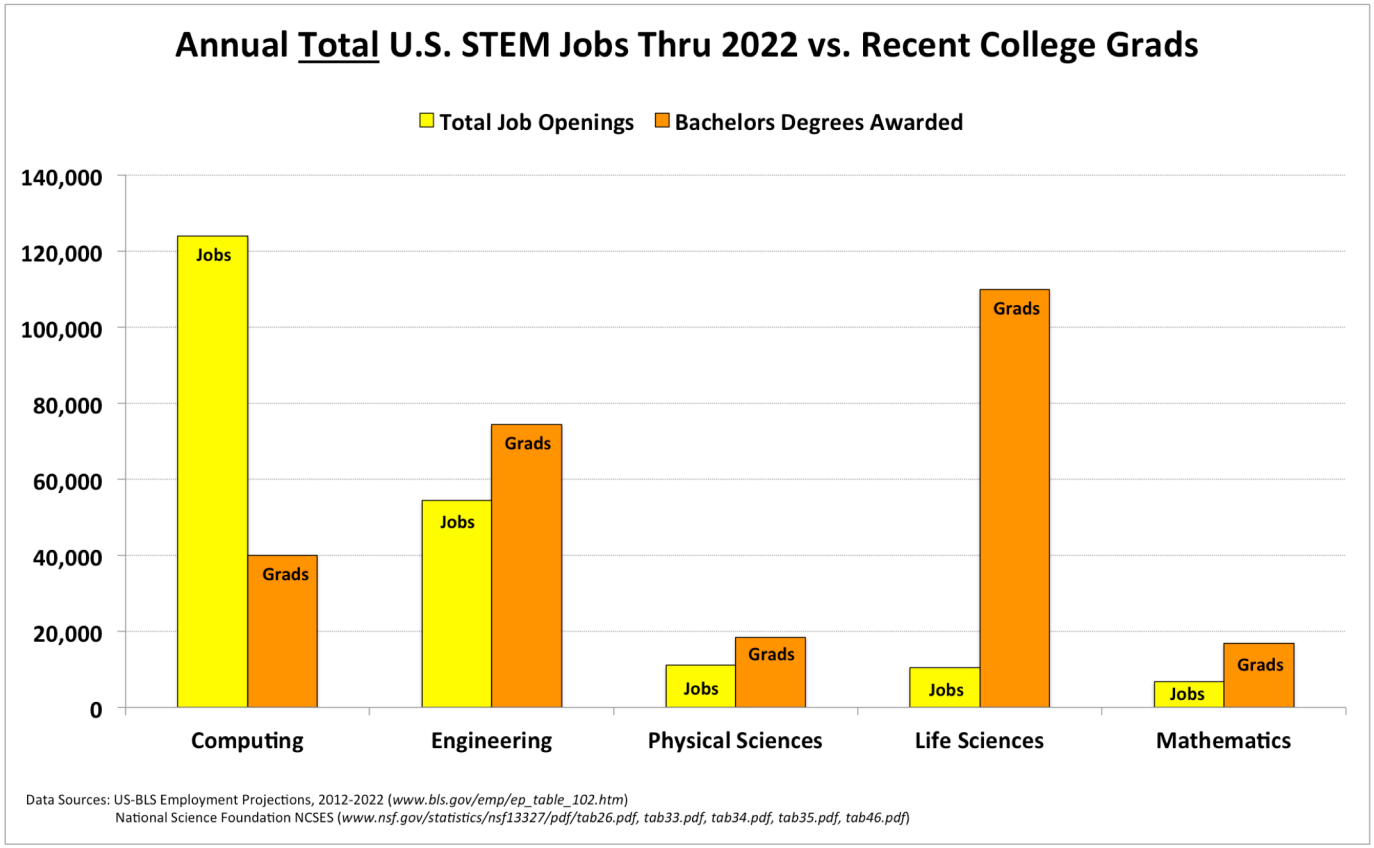
Motivation

Software is an integral part of our lives.

Quick Facts: Software Developers	
2017 Median Pay	$103,560 per year $49.79 per hour
Typical Entry-Level Education	Bachelor's degree
Work Experience in a Related Occupation	None
On-the-job Training	None
Number of Jobs, 2016	1,256,200
Job Outlook, 2016-26	24% (Much faster
Employment Change, 2016-26	302,500

Software Engineer Labor Market

Source: https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm



Software Engineer Scarcity

Source: https://cacm.acm.org/blogs/blog-cacm/180053-computing-is-the-safe-stem-career-choice-today/fulltext

Rising Demand for CS Education

UC Davis

- Class sizes tripled.
- ECS 30 (Introduction to Programming) had **400** students.

UC Berkeley

- CS 61A (Structure and Interpretation of Computer Programs) had **1762** students.

MOOCs

- Millions of students

Source: http://www.dailycal.org/2017/08/24/introductory-computer-science-course-enrollment-increases-last-year/

A lot students are failing.

- UC Davis, ECS 140A, 40 / 140
- Watson and Li, ITiCSE 2014, 33% failure rate.

A lot students are dropping out.

- Ireland, 2016, 33%.

At large scales, teaching quality is affected negatively.

- Room capacity.
- Less interaction with professor and TAs.
- Less personalized instruction.

Issues

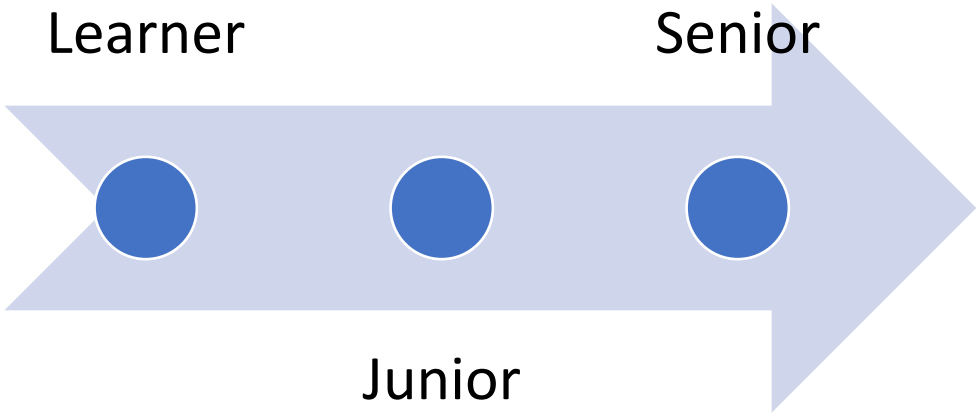
Programming is difficult.

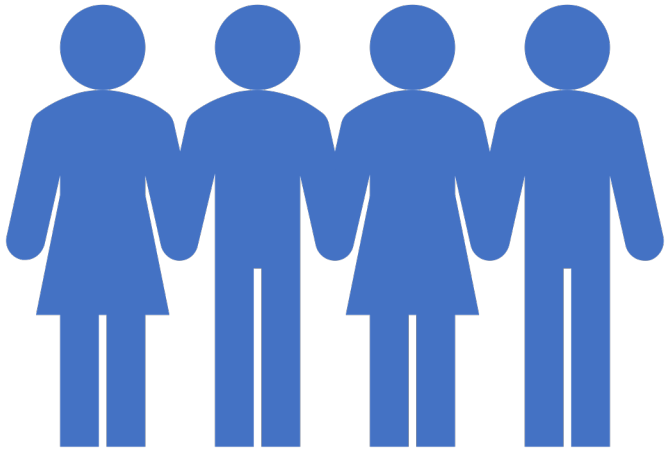
“The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.”

“Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.”

Programmer Experience

- Let’s not think in binary.
- People can be on experience spectrum.
- Experience helps.





How can we help people who are learning to program?

Learners + Junior Programmers >> Senior Programmers

High Potential Impact

Technical Barriers to Learning Programming

Def. Technical challenges that all programmers face but to learners and junior programmers:

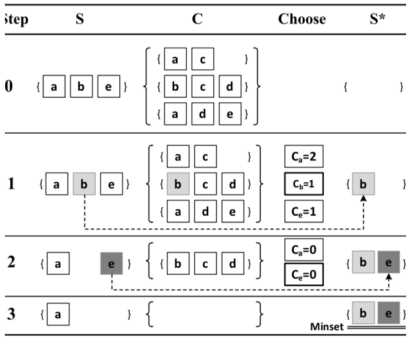
- are much more difficult to resolve,
- can cause excessive frustration,
- can cause excessive wasted time and effort,
- and may even cause learners to quit.

Examples:

- Syntax
- Developer Tools
- Compiler Errors
- Go Programming Language
- Eclipse Che

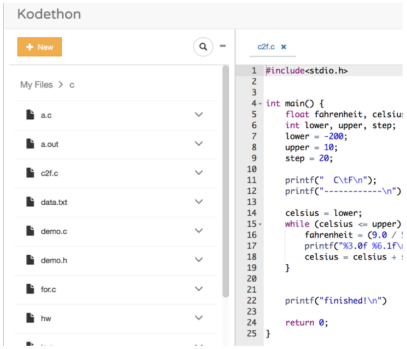
This Talk

Lexical Distinguishability



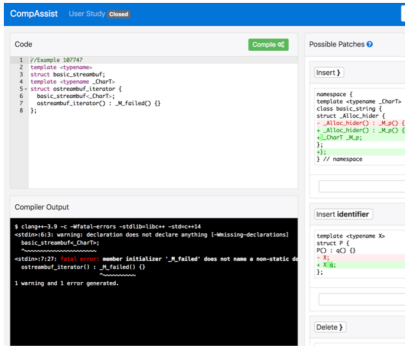
Language Syntax

Kodethon



Development Tools

CompAssist



Compiler Error Messages

UCDAVIS COMPUTER SCIENCE

UNIVERSITY of HOUSTON

Kodethon: University Student Adoption and Perceptions of a web IDE

Collaborators:

Michael Yen,

Mathew Le,

Zhendong Su,

Amin Alipour.

To be submitted to SIGCSE 2019

- Programming consists of many steps: coding, testing, debugging, compiling, and linking.
- Each step requires proper installation and configuration of tools and environment.
- This can be time-consuming and error-prone for both students and teachers.
- Web IDEs provide a uniform, ready-to-use programming platform.

Motivation

Challenges

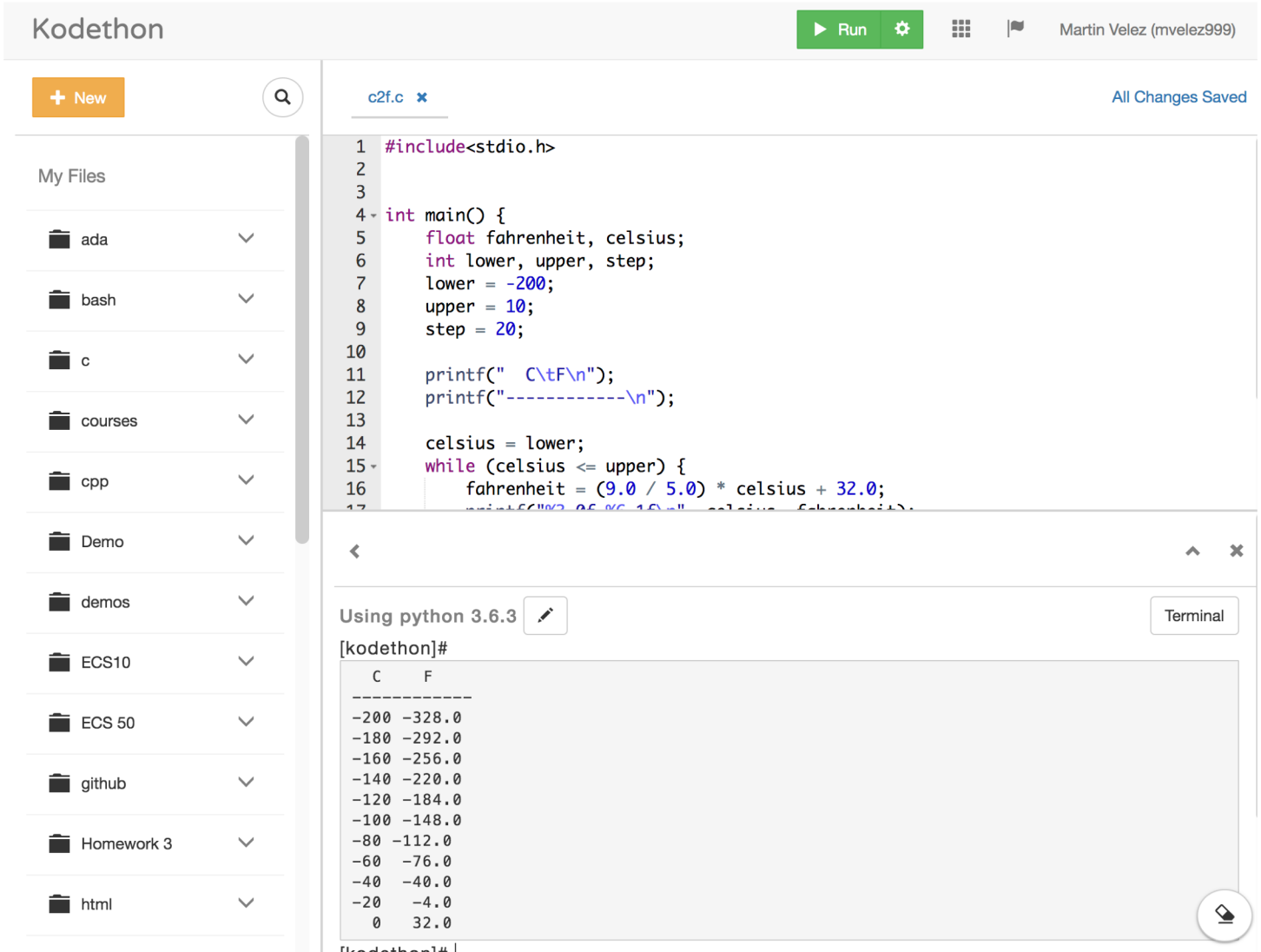
No existing suitable web IDE.

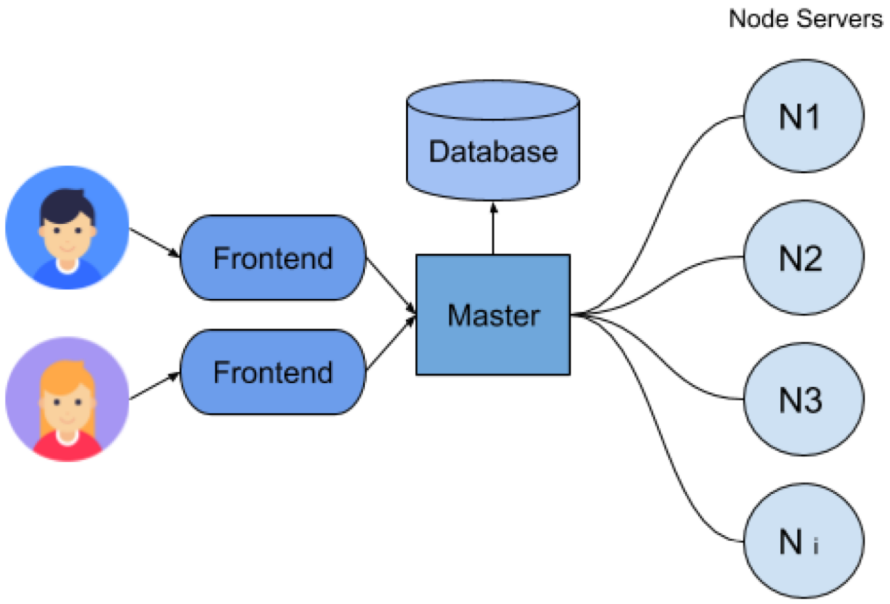
- Cloud9: Too much like desktop IDEs.
- Runnable.com: Defunct.
- Koding.com: Privoted and abandoned IDE.

Basic requirements:

- Support all current PLs.
- Scales to hundreds, thousands of students.
- High availability especially on homework due dates.
- Low latency.

User Interface





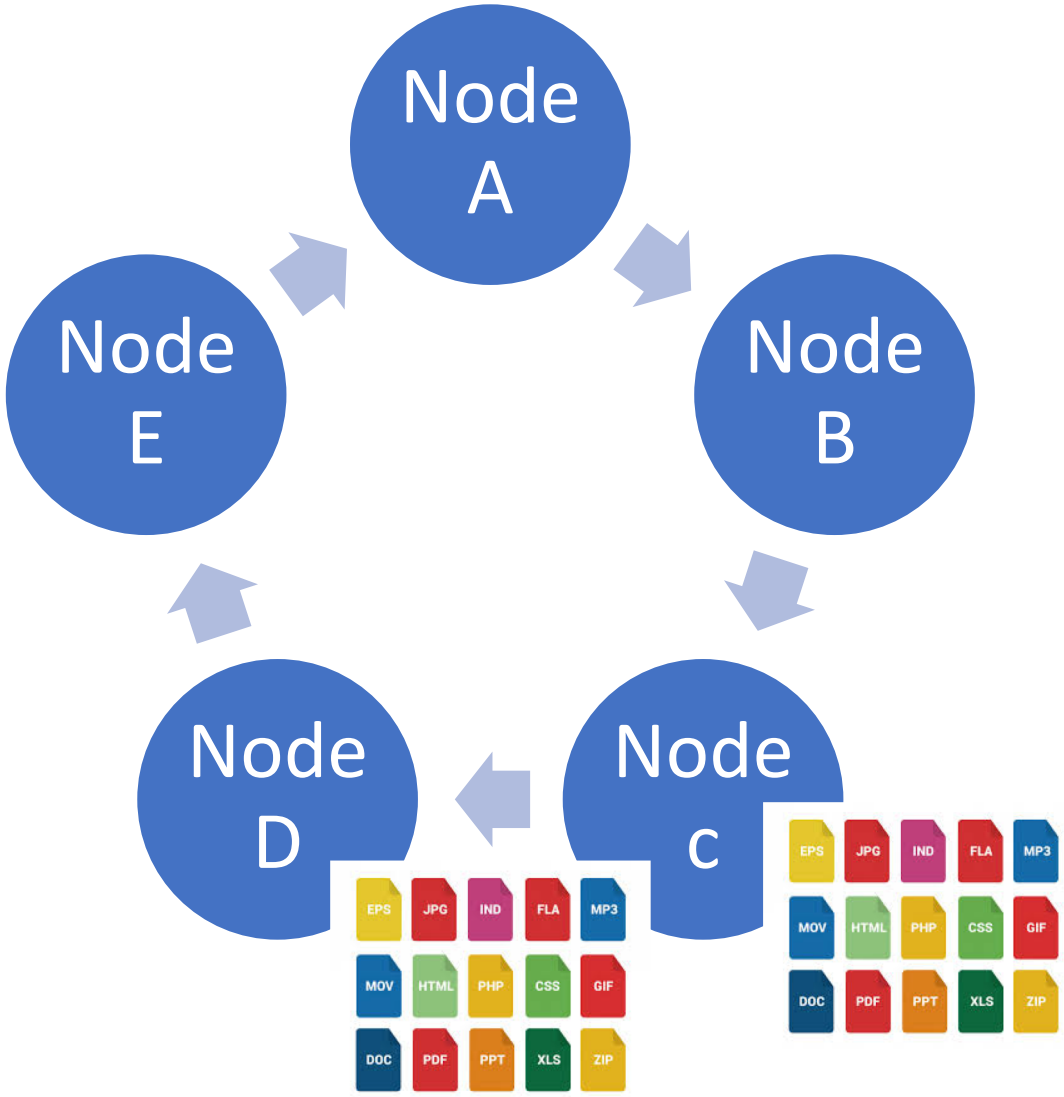
Technologies

- AngularJS
- Ruby-on-Rails
- Docker
- PostgreSQL
- Firebase
- Google API
- And more...

Architecture

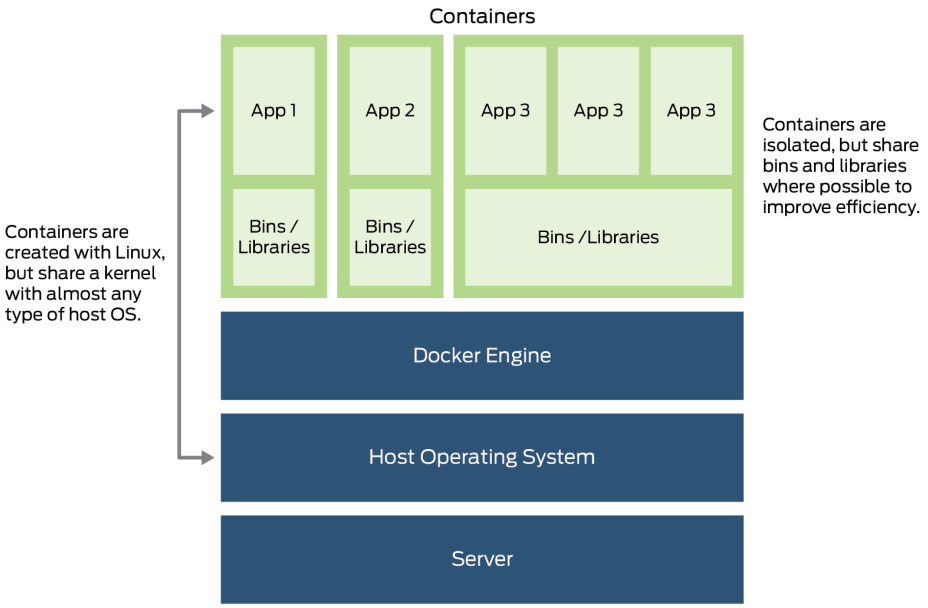
File System

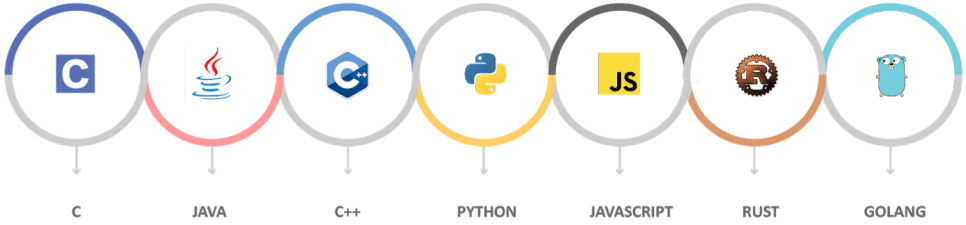
- Circular replication to k nodes.
- Your files are stored on node C.
- They are also replicated on the next k nodes: D, E, ….
- In our implementation k = 1.



Program Execution

- Student programs are compiled and executed on their assigned cluster node.
- When a user runs a program, a Docker container is started.
- Benefits:
  - Security
  - Resource limits like RAM, CPU, and disk space.
- Program output is redirected to a Linux pipe.





Feature Highlights

Projects	• Private and public
Real-time collaboration	• Pair-programming
Unix Terminal.	• Power-users
Learning Management System	• Automatic grading of programming systems

Deployment

1

Development Started in Summer 2014

2

First trial in Spring 2015

- ECS 10 – Introduction to Programming (in Python)

3

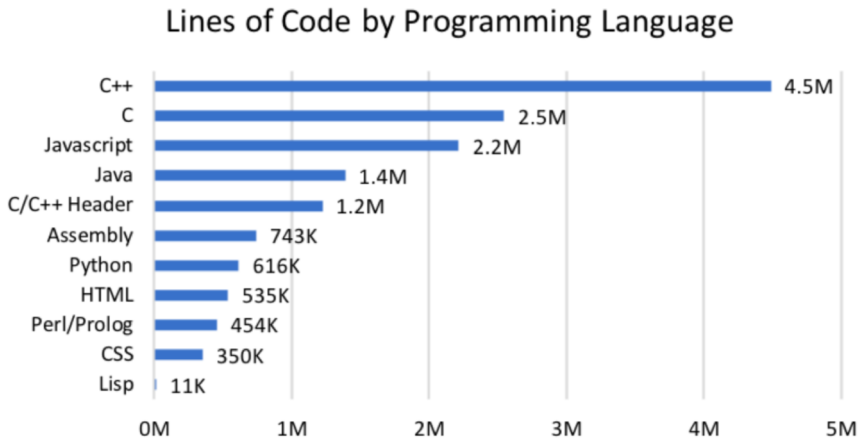
Since then, at least 10 live demonstrations.

- 5 – 15 minutes

Adoption

Title	Programming Language(s)	Live Demo	Survey	Student Use
Introduction to Programming	Python	✓		✓
Introduction to Programming	C	✓		✓
Software Development and Object-Oriented Programming	C++, Rust	✓	✓	✓
Computer Organization and Machine-Dependent Programming	Assembly, C++	✓	✓	✓
Data Structures and Programming	C, C++	✓		✓
Theory of Computation	N/A			✓
Algorithm Design & Analysis	C, C++			✓
Probability and Statistical Modeling for Computer Science	Python, R			✓
Programming Languages	Java, Lisp, Prolog	✓	✓	✓
Scripting Languages and Their Applications	Python, R			✓
Parallel Architecture	C			✓
Software Engineering	XXX			✓
Web Programming	HTML, CSS, Javascript			✓
Introduction to Artificial Intelligence	XXX			✓
Computer Graphics	XXX			✓

- Over 3000 students have signed up.
- We gave live demos in 6 courses but Kodethon is used in at least 15 different undergrad courses.
- Over 15.1M LOC.
  - 4.5M in C++
  - 2.5M in C



Survey

40 closed and open-ended questions.

USE questionnaire (Lund 2001)

- Usability, usefulness, ease-of-use, ease-of-learning, and satisfaction.

Recruitment

- Class forum of 2 ongoing courses (ECS 10 and ECS 30)
- Email students from previous quarte (ECS 140A)
- Received 140 responses

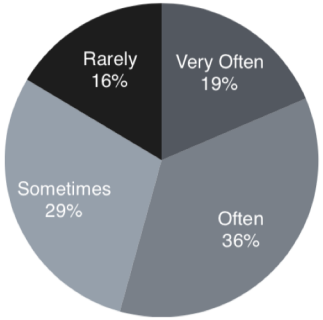
Demographics:

- Male 87, Female 51
- Asian 90, White 35, Other 17
- Freshman 39, Sophomore 40, Junior 38, Senior 23

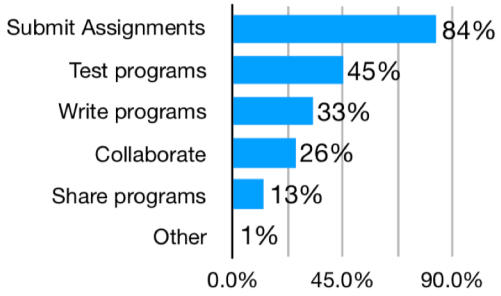
Usage Insights

54% Often or Very Often.

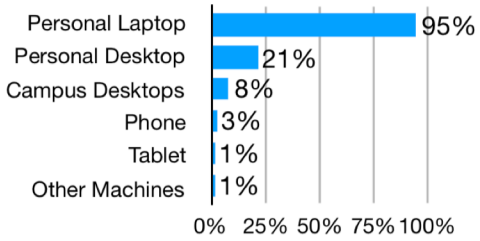
33% Write Programs.



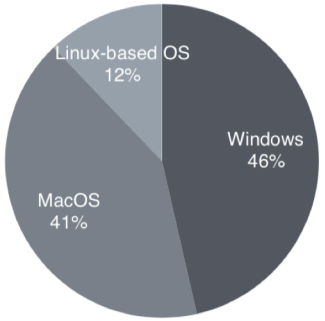
(a) "I use OurIDE..." Note: No one responded "Never".



(b) "I use OurIDE to..."



(c) "I use OurIDE on the following devices..."



(d) "I use OurIDE primarily in..." Note: No one selected "iOS", "Android", or "Other".

#	Item	Strongly agree		Agree		Neither agree nor disagree		Disagree		Strongly disagree		Total
1	It helps me be more effective.	5%	7	27%	36	33%	44	20%	27	15%	20	134
2	It helps me be more productive.	5%	7	23%	31	30%	40	26%	35	16%	22	135
3	It is useful.	5%	7	50%	68	27%	37	8%	11	10%	13	136
4	It gives me more control over the activities in my life.	5%	7	16%	20	38%	49	22%	28	19%	25	129
5	It makes the things I want to accomplish easier to get done.	4%	6	21%	28	32%	43	26%	35	17%	23	135
6	It saves me time when I use it.	4%	6	24%	33	25%	34	29%	40	18%	25	138
7	It meets my needs.	4%	6	33%	45	32%	43	18%	25	13%	17	136
8	It does everything I would expect it to do.	6%	8	24%	33	25%	35	29%	40	17%	23	139
	Total	**5%**	**54**	**27%**	**294**	**30%**	**325**	**22%**	**241**	**16%**	**168**	**1082**

Perceptions (Usefulness)

1082 responses, 32% Positive, 30% Neutral, 38% Negative

Perceptions (Features)

People like benefits resulting from a cloud IDE.

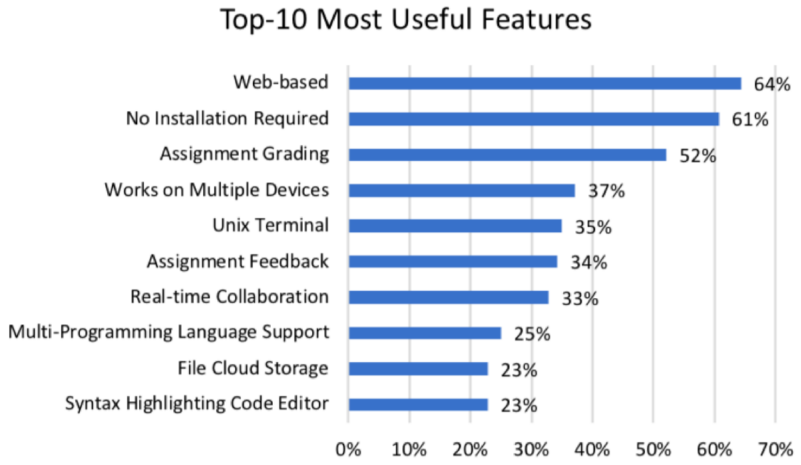


Figure 7: ***"What features of OurIDE do you find most useful?"***

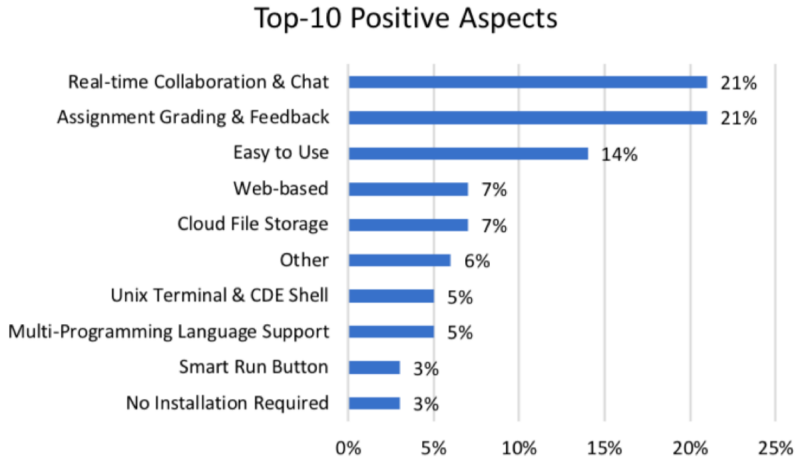
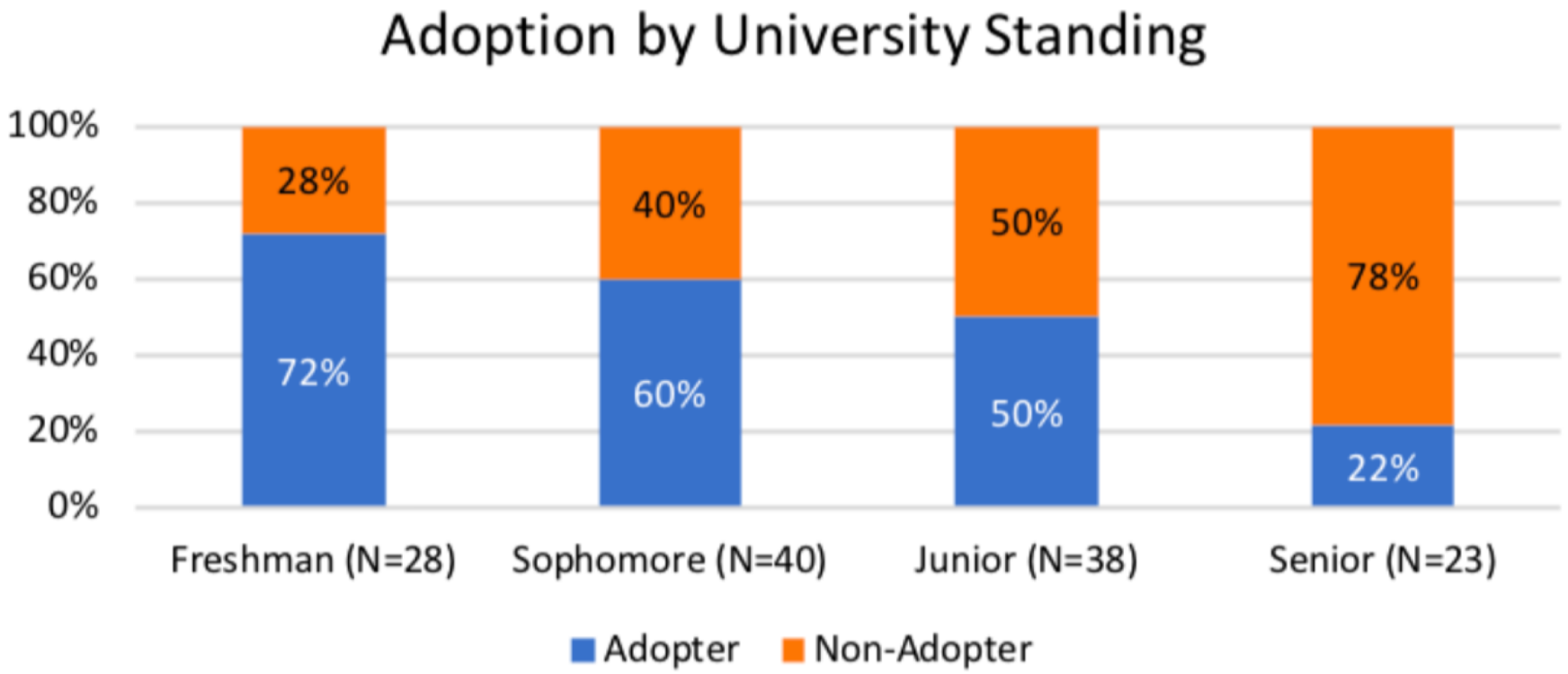


Figure 8: ***"List the most positive aspect(s) of OurIDE:"***



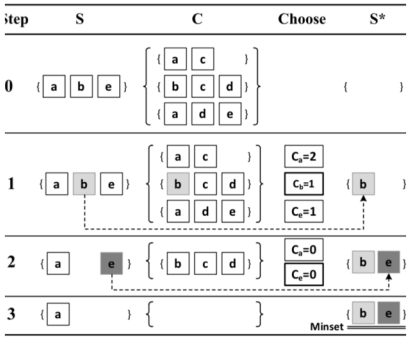
Adoption Factors

Summary and Future Work

- Everyone learns differently.
- Realistic expectations for adoption of pedagogical tools.
- Instructors should introduce students to web IDEs.
  - About a third of students found it "useful”.
- Implicit Learning Objectives
  - "I want them to learn how to set up a development environment.”
  - “I want them to use git.”
- What is “real” programming?
  - Students perceive web IDEs as not “real” tools.
  - Computational thinking vs System building.
- Intelligent Coding
  - Leverage IDE usage to provide relevant information automatically.

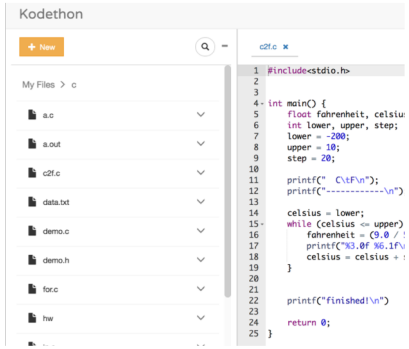
This Talk

Lexical Distinguishability



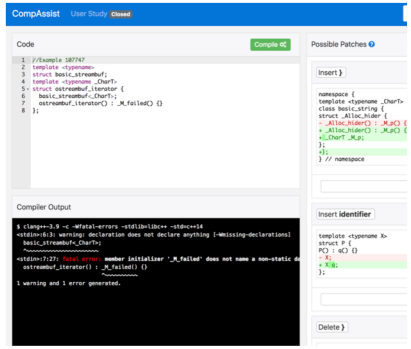
Language Syntax

Kodethon



Development Tools

CompAssist



Compiler Error Messages

Collaborators:

Chengnian Sun (Google),

Nima Joharizadeh,

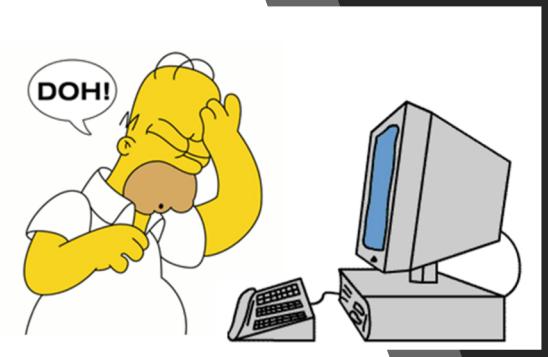
Haichuan Wang (Huawei),

Zhendong Su.

To be submitted to ICSE 2019.

CompAssist: Synthesizing Minimal Compilation Repair Examples

Motivation



All programmers, learners to senior, make compilation errors.	• Compiler errors are difficult to understand.
SIGCSE 2015	• 37M compilations, 17M failures
Google Study 2014	• C++ developers build 10 times/day and 37% fail. • Java developers build 7 times/day and 30% fail.

Example

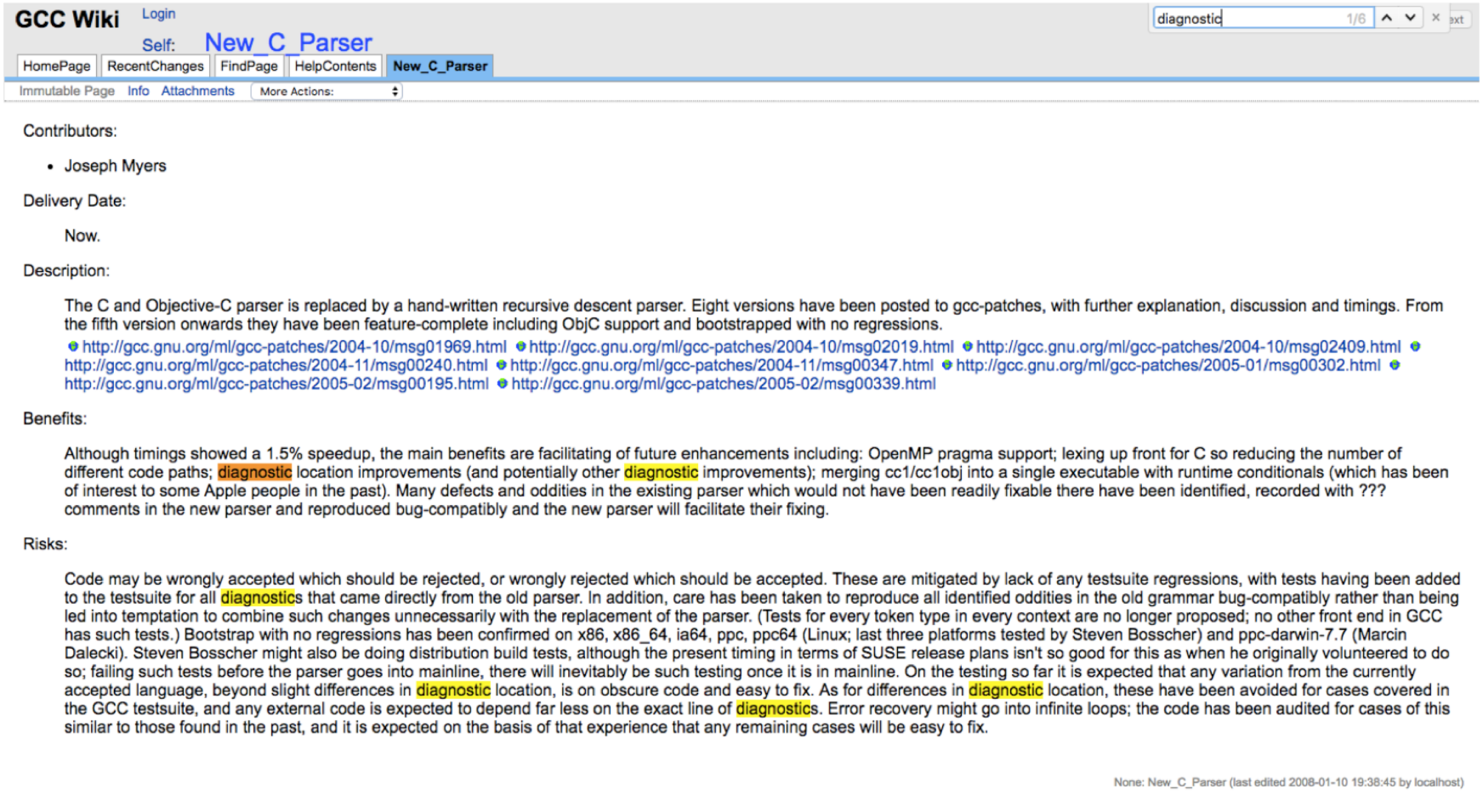
Code

Compile

```
1  int my_scanf(const char* format, ...) {}
2  int main() {
3      int x = 0;
4      my_scanf("%d" &x); // missing comma after double quote
5  }
```

Compiler Output

```
$ clang++-3.9 -c -Wfatal-errors -stdlib=libc++ -std=c++14
<stdin>:1:40: warning: control reaches end of non-void function [-Wreturn-type]
int my_scanf(const char* format, ...) {}
                                       ^
<stdin>:4:19: fatal error: invalid operands to binary expression ('const char *' and 'int')
    my_scanf("%d" &x); // missing comma after double quote
              ~~~~ ^~
1 warning and 1 error generated.
```



GCC Wiki Login

Self: New_C_Parser

diagnostic 1/6

HomePage | RecentChanges | FindPage | HelpContents | New_C_Parser

Immutable Page Info Attachments More Actions:

Contributors:

- Joseph Myers

Delivery Date:

Now.

Description:

The C and Objective-C parser is replaced by a hand-written recursive descent parser. Eight versions have been posted to gcc-patches, with further explanation, discussion and timings. From the fifth version onwards they have been feature-complete including ObjC support and bootstrapped with no regressions.

http://gcc.gnu.org/ml/gcc-patches/2004-10/msg01969.html http://gcc.gnu.org/ml/gcc-patches/2004-10/msg02019.html http://gcc.gnu.org/ml/gcc-patches/2004-10/msg02409.html http://gcc.gnu.org/ml/gcc-patches/2004-11/msg00240.html http://gcc.gnu.org/ml/gcc-patches/2004-11/msg00347.html http://gcc.gnu.org/ml/gcc-patches/2005-01/msg00302.html http://gcc.gnu.org/ml/gcc-patches/2005-02/msg00195.html http://gcc.gnu.org/ml/gcc-patches/2005-02/msg00339.html

Benefits:

Although timings showed a 1.5% speedup, the main benefits are facilitating of future enhancements including: OpenMP pragma support; lexing up front for C so reducing the number of different code paths; diagnostic location improvements (and potentially other diagnostic improvements); merging cc1/cc1obj into a single executable with runtime conditionals (which has been of interest to some Apple people in the past). Many defects and oddities in the existing parser which would not have been readily fixable there have been identified, recorded with ??? comments in the new parser and reproduced bug-compatibly and the new parser will facilitate their fixing.

Risks:

Code may be wrongly accepted which should be rejected, or wrongly rejected which should be accepted. These are mitigated by lack of any testsuite regressions, with tests having been added to the testsuite for all diagnostics that came directly from the old parser. In addition, care has been taken to reproduce all identified oddities in the old grammar bug-compatibly rather than being led into temptation to combine such changes unnecessarily with the replacement of the parser. (Tests for every token type in every context are no longer proposed; no other front end in GCC has such tests.) Bootstrap with no regressions has been confirmed on x86, x86_64, ia64, ppc, ppc64 (Linux; last three platforms tested by Steven Bosscher) and ppc-darwin-7.7 (Marcin Dalecki). Steven Bosscher might also be doing distribution build tests, although the present timing in terms of SUSE release plans isn't so good for this as when he originally volunteered to do so; failing such tests before the parser goes into mainline, there will inevitably be such testing once it is in mainline. On the testing so far it is expected that any variation from the currently accepted language, beyond slight differences in diagnostic location, is on obscure code and easy to fix. As for differences in diagnostic location, these have been avoided for cases covered in the GCC testsuite, and any external code is expected to depend far less on the exact line of diagnostics. Error recovery might go into infinite loops; the code has been audited for cases of this similar to those found in the past, and it is expected on the basis of that experience that any remaining cases will be easy to fix.

None: New_C_Parser (last edited 2008-01-10 19:38:45 by localhost)

Can’t we just write better error messages?

- Previous example from recent version of clang.
- In 2008, GCC replaced entire parser.

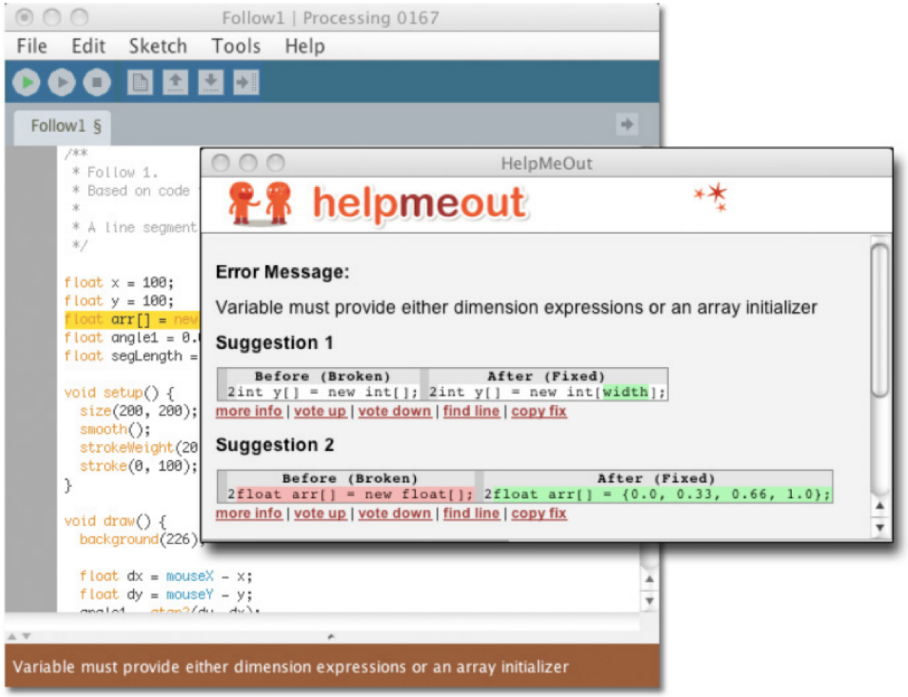


Figure 2. The HelpMeOut Suggestion Panel shows possible corrections for a reported compiler error.

HelpMeOut (CHI 2010)

- An "IDE" plugin that shows example fixes.
- Collect example fixes using **crowdsourcing**.

Limitations

- Privacy
- The Bootstrap Problem
- Extensibility
- Bias/Completeness
- Inferring Patches

Compiler Error Message Augmentation

Uncompilable Program

```
1 int my_scanf(const char* format, ...) {}
2
2 int main() {
3     int x = 0;
4     my_scanf("%d" &x);
5 }
```

```
$ clang++-3.9 -c -Wfatal-errors -stdlib=libc++ -std=c++14
<stdin>:1:40: warning: control reaches end of non-void function [-Wreturn-type]
int my_scanf(const char* format, ...) {}
                                       ^
<stdin>:4:19: fatal error: invalid operands to binary expression ('const char *' and 'int')
    my_scanf("%d" &x);
             ~~~~ ^~
1 warning and 1 error generated.
```

Compilable Program

```
1 int my_scanf(const char* format, ...) {}
2
2 int main() {
3     int x = 0;
4     my_scanf("%d", &x);
5 }
```

```
$ clang++-3.9 -c -Wfatal-errors -stdlib=libc++ -std=c++14
:1:40: warning: control reaches end of non-void function [-Wreturn-type]
int my_scanf(const char* format, ...) {}
                                       ^
1 warning generated.
```

Compilation Repair Example

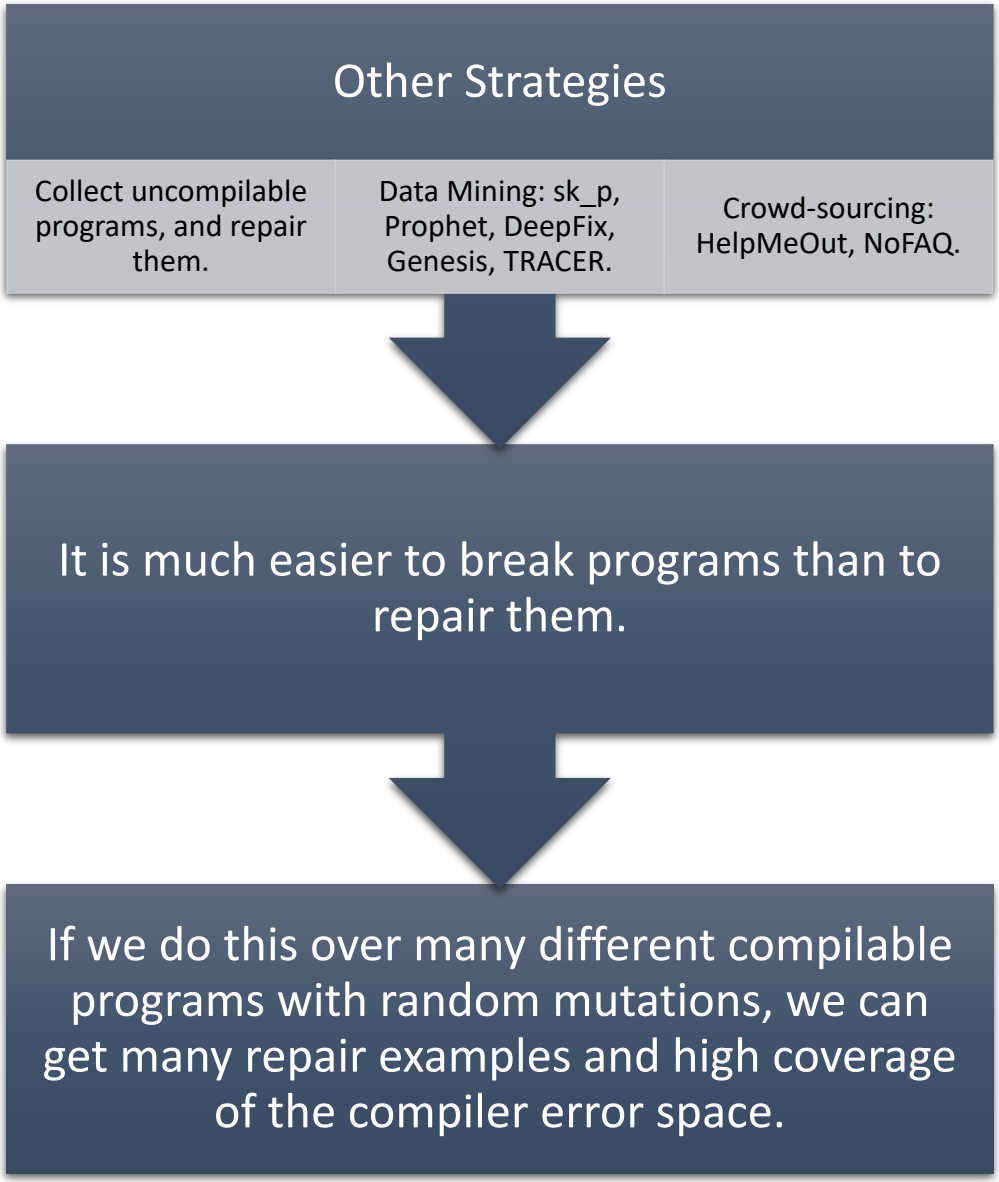
For a specific compiler, (u, c) is a compilation repair example for error message e, if u is an uncompilable program that triggers e, and c is a compilable program. The difference between u and c is a repair patch (diff) Δ.

Challenges

Compilation repair examples can help programmers resolve errors.

How do you obtain compilation repair examples?

How do you present them?


Other Strategies
Collect uncompilable programs, and repair them.
Data Mining: sk_p, Prophet, DeepFix, Genesis, TRACER.
Crowd-sourcing: HelpMeOut, NoFAQ.
It is much easier to break programs than to repair them.
If we do this over many different compilable programs with random mutations, we can get many repair examples and high coverage of the compiler error space.

Key Insight

```
1 constexpr int Apply(const int in, int (*f)(const int&)) { return
      f(in); }
2 using Foo = int;
3 static constexpr int id(const Foo& i) { return i; }
4 - static constexpr int results1 = Apply(0, &id);
5 + static constexpr int results1 = Apply(0 &id); //Delete comma
6 //Error Message: invalid operands to binary expression ('int' and
      'int (*)(const Foo &)' (aka int (*)(const int &)'))
```

Fuzz

We perform single-operation, single-token mutations.

- Tokenize source code
- Mutate
  - Delete a random token.
  - Select a random token. Insert it a random position.

Algorithm 2: Reduce a compilation error repair example.

Data: u_0 is the source code of a uncompilable program.
Data: c_0 is the source code of a compilable program.

Function Reduce(u_0, c_0)
- $u_1 \leftarrow$ Format(u_0) /* One token per line. */
- $c_1 \leftarrow$ Format(c_0) /* One token per line. */
- $(u_2, c_2) \leftarrow$ Align(u_1, c_1) /* See description. */
- $\Delta_0 \leftarrow$ Diff(u_2, c_2) /* Repair patch. */
- $\Delta_0^R \leftarrow$ Diff(c_2, u_2) /* Breaking mutation. */
- $c_3 \leftarrow$ DeltaDebug(c_2, Δ_0^R)
- $u_3 \leftarrow$ Patch(c_3, Δ_0^R)
- $u_4 \leftarrow$ PrettyPrint(u_3)
- $c_4 \leftarrow$ PrettyPrint(c_3)
- **return** (u_4, c_4) /* Reduced repair example. */

```
//Error Message: invalid operands to binary expression ('int' and
      'int (*)(const Foo &)' (aka int (*)(const int &)'))
int Apply(int, int(const int&)) {}
using Foo = int;
int id(const Foo&) {}
- int results1 = Apply(0 &id);
+ int results1 = Apply(0, &id); // Insert comma
```

Post Condition: u4 triggers same error as u0.

Reduce

Algorithm 1: Generate compilation repair examples via the FUZZ-AND-REDUCE approach.

Data: $\mathcal{P}$ is a corpus of compilable programs.

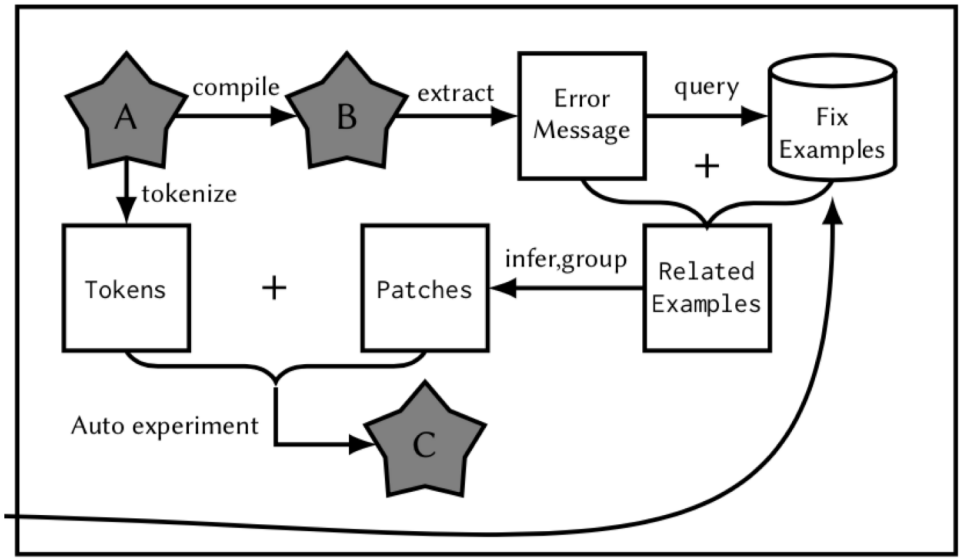
```
1 Function Generate(P)
2     E ← ∅                                   /* Repair Examples */
3     foreach c ∈ P do
4         u ← Fuzz(c)                         /* Fuzz seed program */
5         (exit_status, output) ← Compile(u)
6         if exit_status == ERROR then
7             (u′,c′) ← Reduce(u,c)
8             E ← E ∪ {(u′, c′)}
9     return E
```

Fuzz-and-Reduce

Search Engine

Given a user program that fails to compile.

1. Extract error message.
2. Get related examples.
3. Group examples by patch.
4. Auto-experiment on user program with patches.
5. Return ranked patches with successful patches first.



Evaluation

RQ1: What proportion of compiler errors are covered by the repair examples?

RQ2: Do we generate a diverse collection of repair examples for each compiler error?

RQ3: Are the generated compiler repair examples simple?

Experimental Setup

A mainstream C++ compiler

- clang++-3.9.

Compilable Programs

- GNU GCC test suite.
- 25,240 compilable programs for C and C++

Fuzz-and-Reduce

- 10 days
- Exit on first fatal error

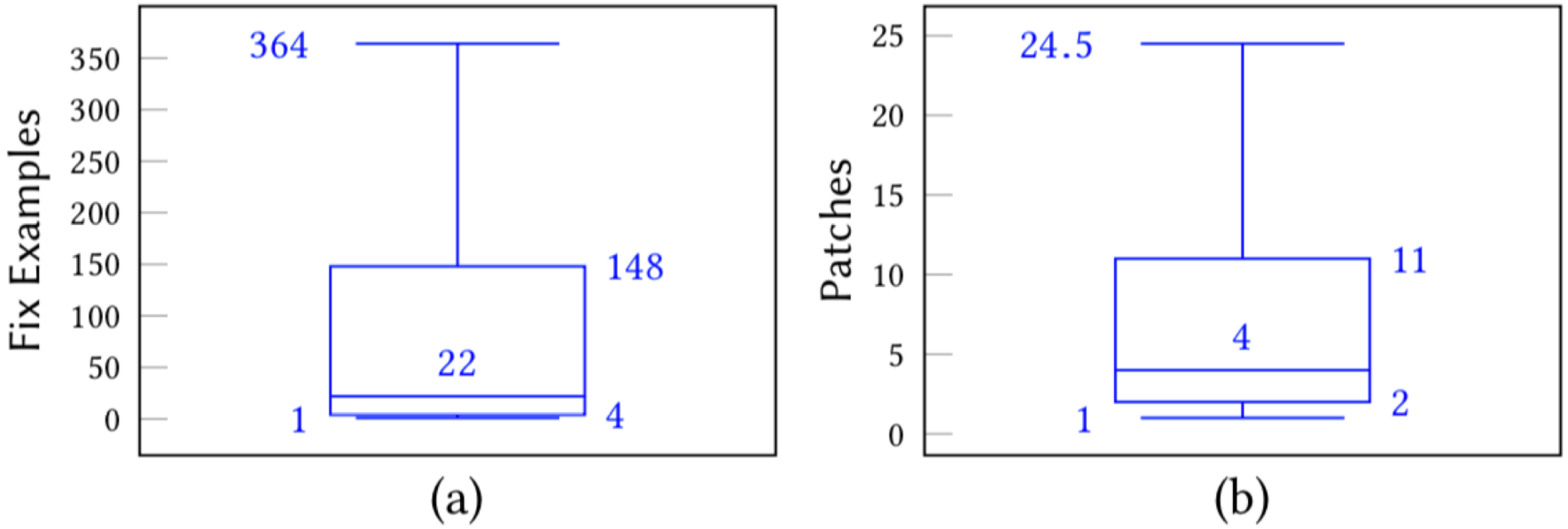
clang's source code.

- 4, 092 distinct diagnostic messages, including errors, warnings, notes, remarks, and other type of messages.
- 1, 686 kinds of error that could potentially be triggered by compiling C and C++ programs.

Component	Repair Examples	Covered Errors	Total Errors	Coverage
Lex	925	4	105	3.9%
Parse	27,505	140	216	64.8%
Sema	90,955	734	1374	53.4%
CodeGen	5	2	12	16.7%
All	116,019	867	1686	51.4%

RQ1: What proportion of compiler errors are covered by the repair examples?

Error Message Coverage (Breadth)

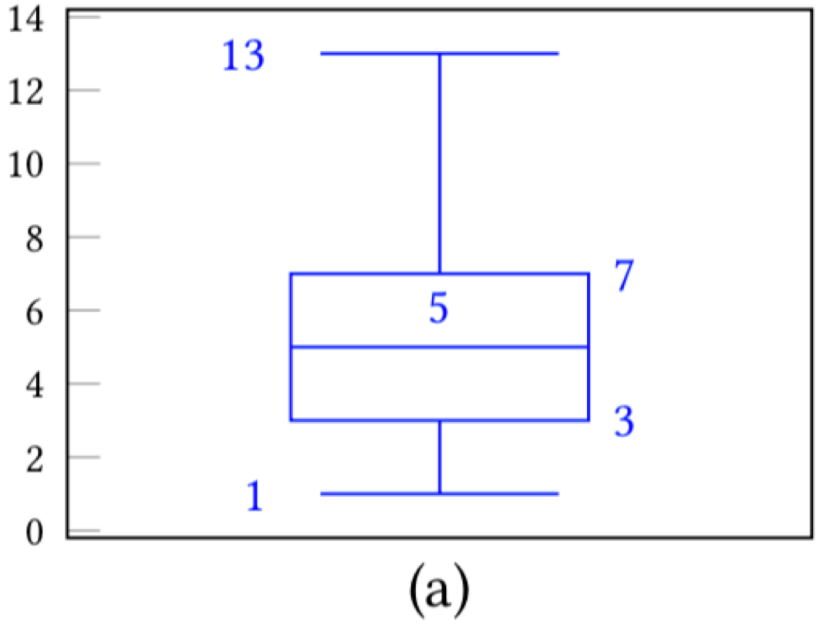


Generated at least 2 different repair examples for 89.6% of errors. Also, for 79.8% of errors, it generated at least 2 distinct patches.

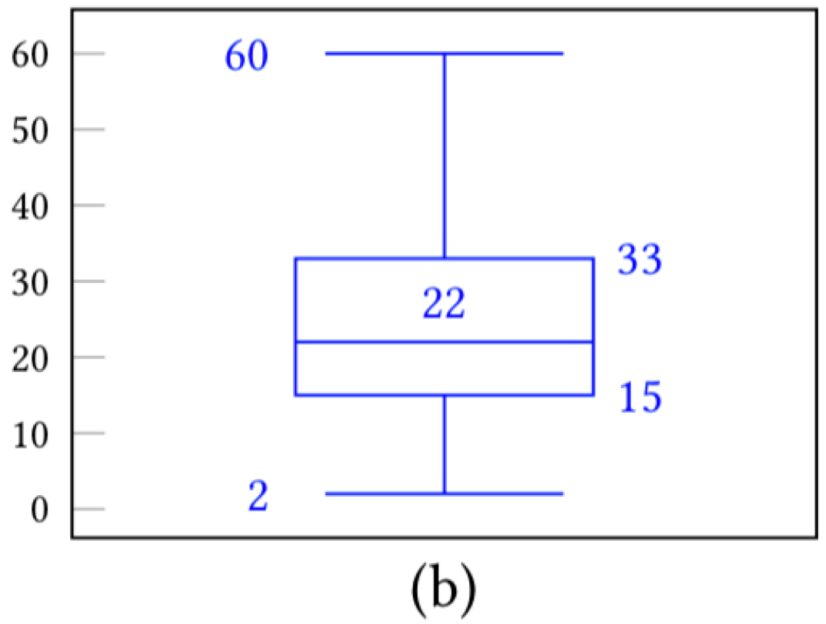
RQ2: Do we generate a diverse collection of repair examples for each compiler error?

Error Message Coverage (Depth)

LOC



Tokens



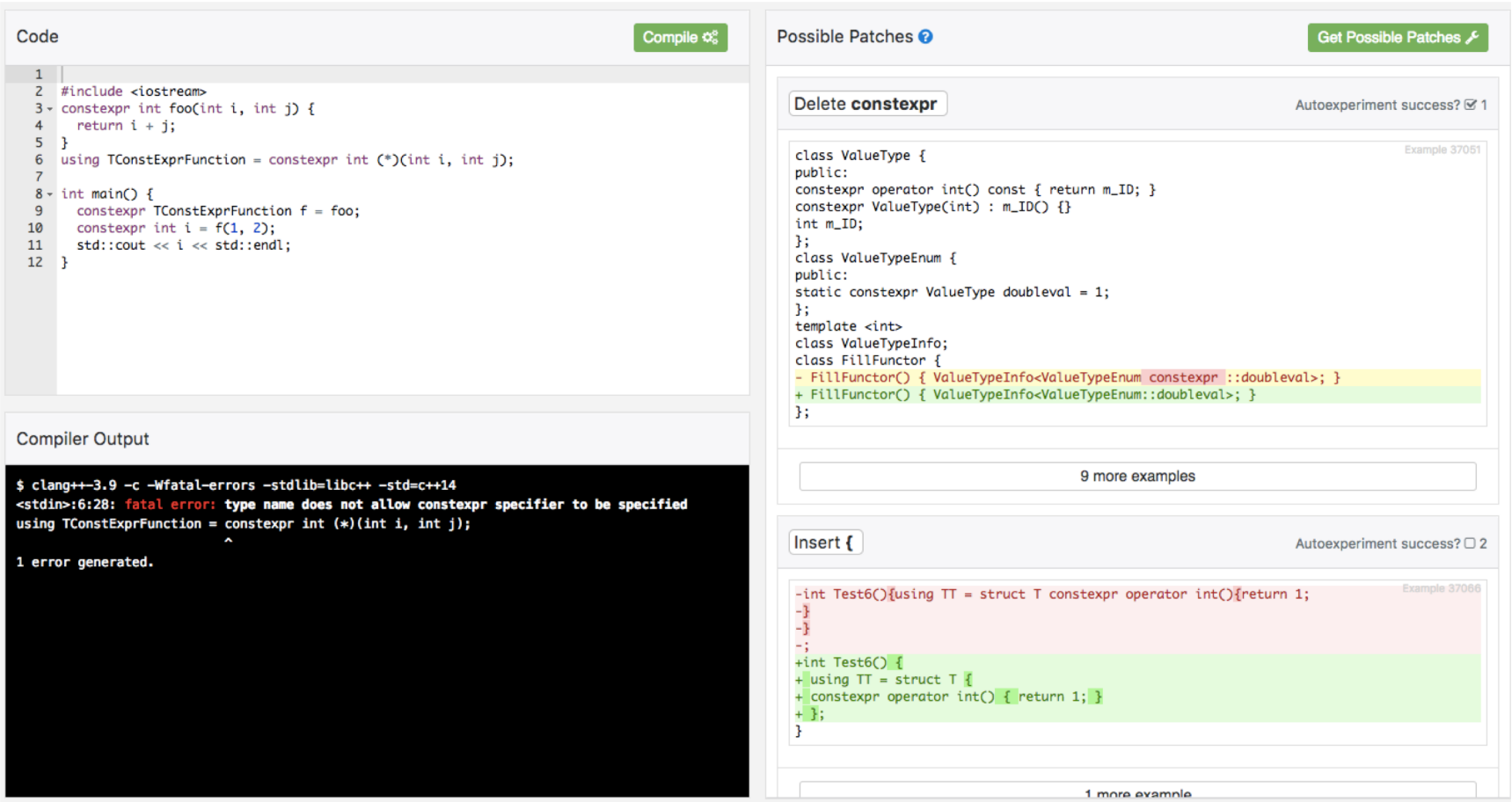
RQ3: Are the generated compiler repair examples simple?

59.6% have 5 or fewer LOC, 60.8% have 25 or fewer tokens

User Study

- Do programmers find the generated compilation repair examples helpful in resolving compiler errors of real-world C++ programs?
- 9 Tasks (real-world C++ uncompilable programs).
- 14 participants
- Find and fix the code defect. Submit solution.
- Rate helpfulness of repair examples.

Example Task



User Study Results

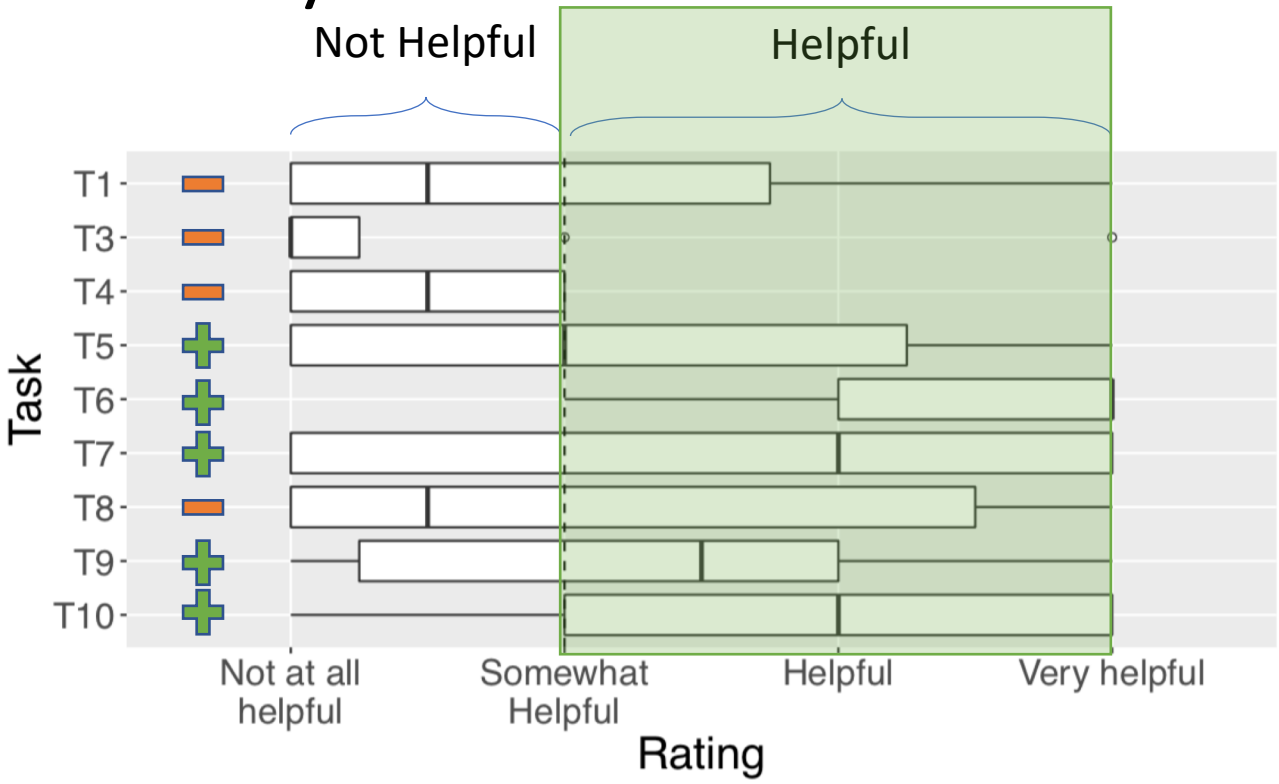


Figure 7: After each task, participants rated the helpfulness of repair examples. Participants rated examples as helpful ("Somewhat helpful" or above) in 5 out of 9 tasks.

Participant Feedback

Repair examples illustrated patches that resolved the compiler error.

- P12: “the ones they got right they were perfectly good”

Repair examples helped with unfamiliar C++ concepts.

- P1: “Suggested patches was helpful for error messages to concepts that I am unfamiliar with.”
- P8: “The suggested patches and examples were a good refresher and helped save me a lot of time.”

Repair examples suggested possible actions.

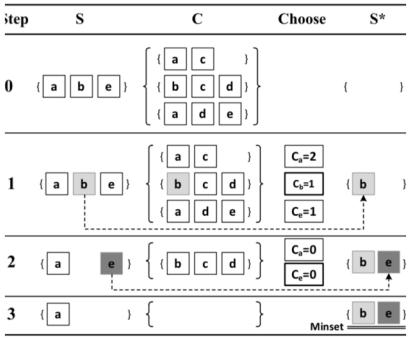
- P2: “some possible moves you can make from the current state of the program.”
- P11: “repair examples told me exactly what to do to fix the compiler error even though I had no idea what was going on.”

Summary and Future Work

- Novel technique for synthesizing compilation repair examples.
  - Offline Generation + Online Search
- Generalizability: We can adapt this technique for different compilers, e.g., javac.
- Coverage:
  - Continuous fuzz-and-reduce.
  - Additional seeds.
  - Unreachable code?
- Apply top-ranked patched automatically to user program.
- Integrate CompAssist into Kodethon, amd IDEs like Visual Studio.
- Adapt compiler error messages to user proficiency.

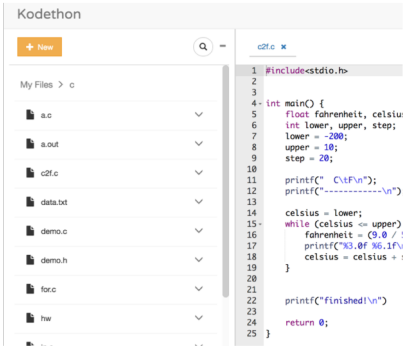
Thank you!

Lexical Distinguishability



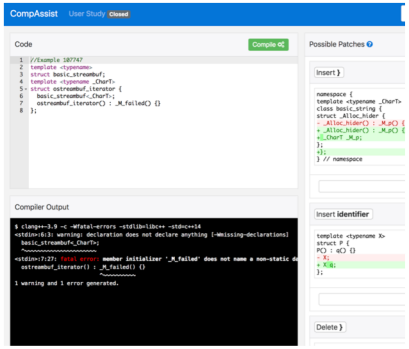
Language Syntax

Kodethon



Development Tools

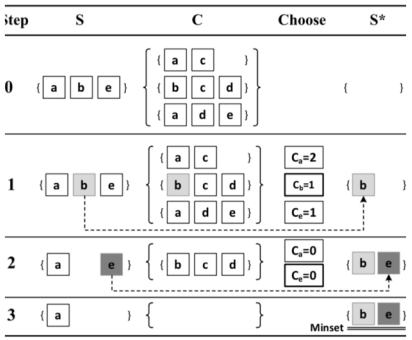
CompAssist



Compiler Error Messages

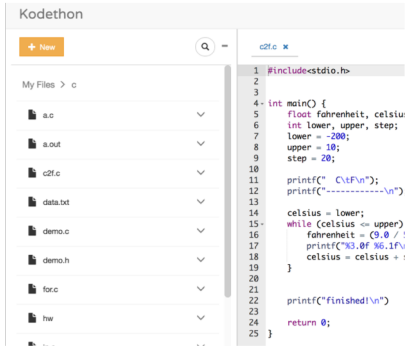
This Talk

Lexical Distinguishability



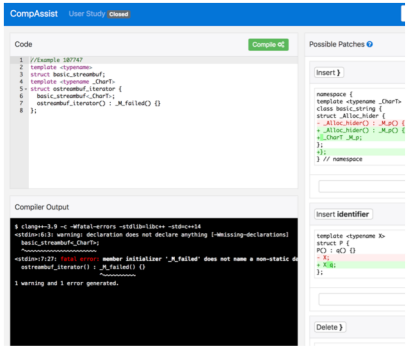
Language Syntax

Kodethon



Development Tools

CompAssist



Compiler Error Messages

Collaborators:

Dong Qiu (Huawei),

You Zhou (Facebook),

Earl Barr (UCL),

Zhendong Su (UC Davis).

Gold Medal at ACM Student Research Competition @ OOPSLA 2014.

On the Lexical Distinguishability of Source Code

Motivation

- In a natural language **sentence**, some **words** are more **important** to its meaning than others.
  - "My **backpack** is very **heavy**."
  - "There is a lot of **smoke outside**."
- From a few **distinctive words**, we can often **guess** the **meaning** of the original sentence.
  - I **enjoy rainy** days.
  - They **enjoy** the **rainy** season.

- Sloppy Programming
  - Little et al., ASE, 2007
  - Write a few key words, "list add".
  - Synthesize rest of code.
- SmartSynth
  - Le et al., MobiSys, 2013.
  - Keywords to TouchDevelop Scripts.
- Code Search
  - "*What should I type to find the code I want?*"

The Wheat and Chaff Hypothesis

Source code consists of wheat and chaff. It contains a lot of chaff.

Challenge: Can we separate the wheat from the chaff in programming language sentences?

Vision: Programmer writes wheat. Computer fills in chaff.

Problem Formulation

- In source code:
  - What is wheat?
  - What is chaff?
  - How do we separate them?

Bag-of-Words Model

Unit of Code

- Functions are natural, likely distinct, pieces of code and functionality.

Bag-of-Words

- A function is a set/multiset of words.

A **word** is a lexeme or some abstraction.

- ”hello” -> STRINGLITERAL
- 3.14 -> float

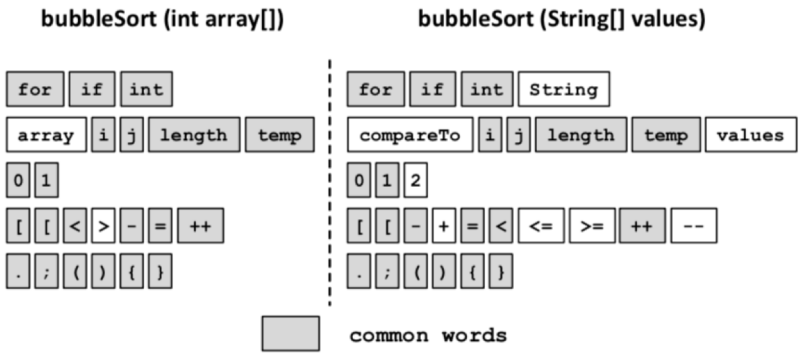
Lexicon

- A **lexicon** is a set of words.
- Varying the lexicon allows us to explore programming language-specific information.

Example

```java
/**
 * Standard Bubble Sort algorithm.
 * @param array The array to sort.
 */
private static void bubbleSort(int array[]) {
  int length = array.length;
  for (int i = 0; i < length; i++) {
    for (int j = 1; j > length - i; j++) {
      if (array[j-1] > array[j]) {
        int temp = array[j-1];
        array[j-1] = array[j];
        array[j] = temp;
      }
    }
  }
}
```

```java
static void bubblesort(String[] values) {
  // no Java sort, so ugly bubble sort
  for (int i=0; i<=values.length-2; i++) { // stop sort
       early to save time!
    for (int j=values.length-2; j>=i; j--) {
      // check that the jth value is smaller than j+1th,
      // else swap
      if (0 < values[j].compareTo(values[j+1])) {
        // swap
        String temp = values[j];
        values[j] = values[j+1];
        values[j+1] = temp;
      }
    }
  }
}
```



Distinguishable Code

Given a set S, and a finite collection of finite sets $Coll$, S^* is a **distinguishing subset** of S, if and only if:

$$S^* \subseteq S$$

$$\forall C \in Coll, S^* \notin C$$

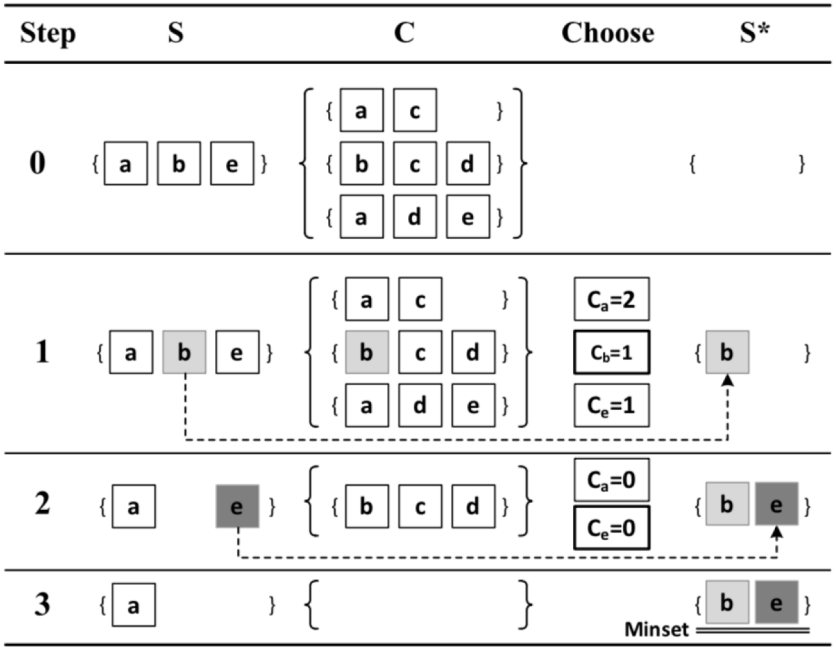
A unit of code is **lexically distinguishable** if it has a distinguishing subset

The MINSET Problem

- Given a finite set S, and a finite collection of finite sets Coll, find a minimum distinguishing subset (minset) S* of S.
- Plain English: We want to find the smallest distinguishing subset of a function.
- Example:
  - Let S1 = {a,b,c,e}, S2 = {a,c}, S3 = {b,c,d}, S4 = {a,d,e}.
  - MINSET(S1, {S2, S3, S4}) = {a,b}.
  - MINSET(S2, {S1, S3, S4}) = Ø
  - MINSET(S3, {S1, S2, S4}) = {b,c}

Computational Complexity

- Theorem:
  - MINSET is NP-hard.
- Proof Sketch:
  - Reduce HITTING-SET to MINSET.



Input: S, the set to minimize.
Input: $\mathcal{C}$, the collection of sets against which S is minimized.

1: $\mathcal{C}_e = \{C \mid C \in \mathcal{C} \wedge e \in C\}$ are those sets in $\mathcal{C}$ that contain e.
2: $S^* = \emptyset$
3: **while** $S \neq \emptyset \wedge \mathcal{C} \neq \emptyset$ **do**
// Greedily pick an element that most differentiates S.
4: $e := \text{CHOOSE}(\{x \in S \mid |\mathcal{C}_x| \leq |\mathcal{C}_y|, \forall y \in S\})$
5: **if** $\mathcal{C}_e = \emptyset \vee \mathcal{C}_e = \mathcal{C}$ **break**
6: $S^* := S^* \cup \{e\}$
7: $S := S \setminus \{e\}$
8: $\mathcal{C} := \mathcal{C}_e$
9: **return** $S^*, \mathcal{C}$

The Minset Algorithm

Research Questions

RQ1	RQ2	RQ3
• How many units of code are lexically distinguishable?	• How much of code is needed to distinguish?	• What is a natural, minimal lexicon?

Empirical Setup

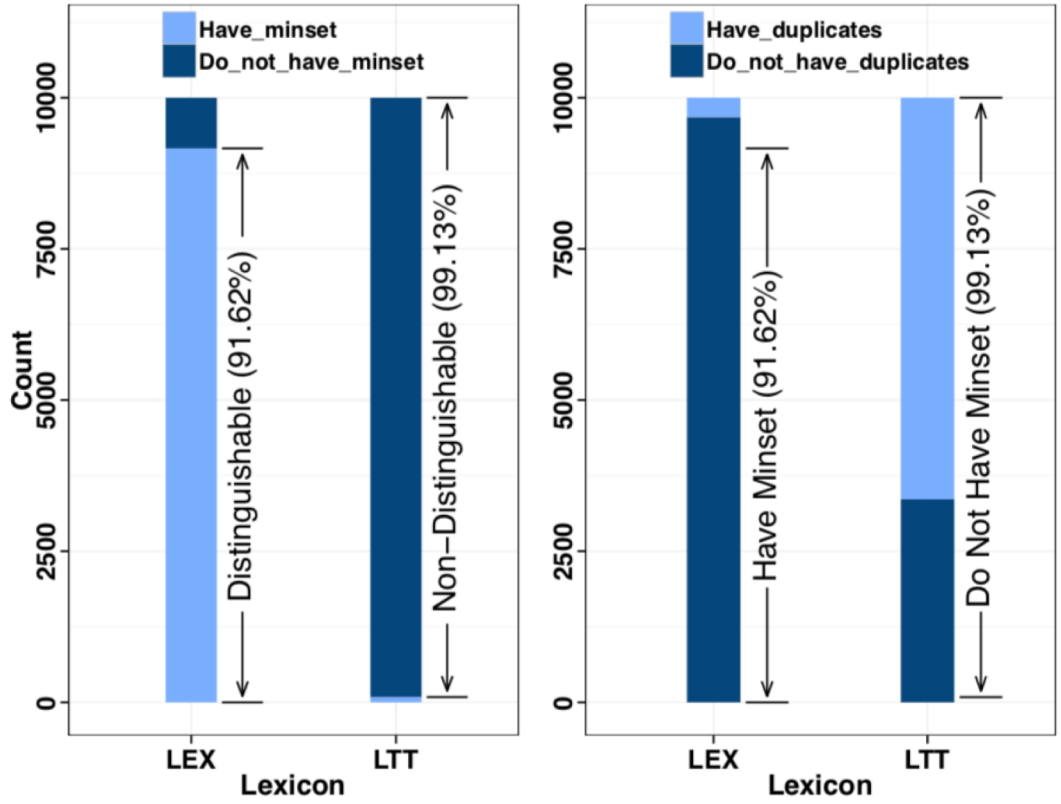
- A popular programming language, Java.
- Most popular projects in open-source repositories.
- Large Corpus (100M LOC)
- Large Universe of Methods (1.9M)

Methods	Count
Total (in corpus)	8,918,575
Unique	8,135,663
Unique (50 or more tokens)	1,870,905
Unique (50 to 562 tokens)	1,801,370

Repository	Projects	Files	Lines of Code
Apache	103	101,480	10,891,228
Eclipse	102	287,669	32,770,246
Github	170	133,793	13,752,295
Sourceforge	533	373,556	42,434,029
Total	908	896,498	99,847,798

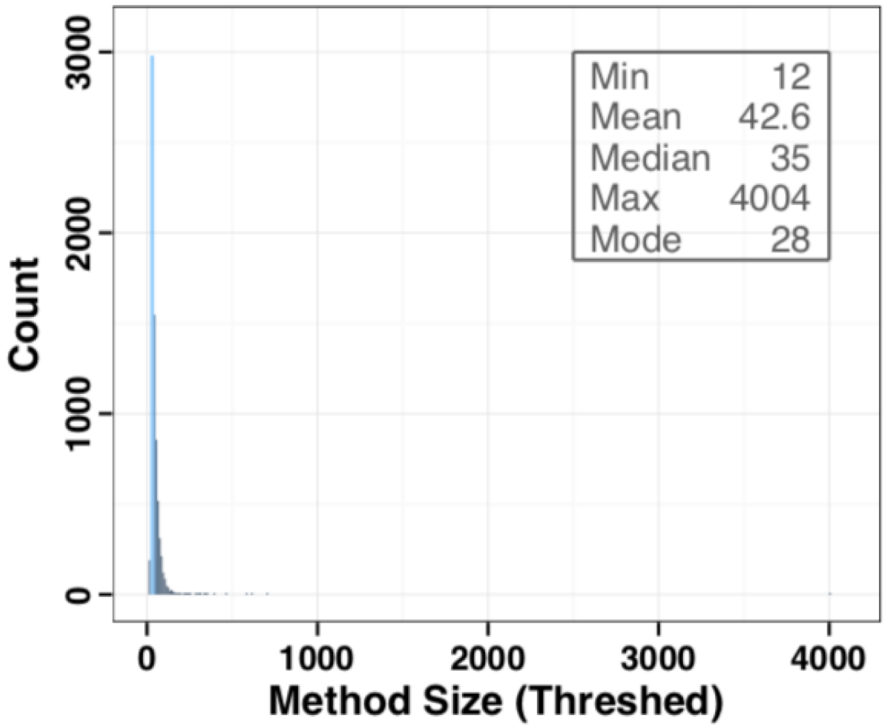
Name	Description	Size (words)
LEX	All (raw) lexemes	5,611,561
LTT	All lexer token types	101
MIN1	Fully qualified standard library method names	55,543
	and basic operators	55,543
MIN2	MIN1 plus control keywords	55,556
MIN3	MIN2 plus fully qualified public type names	91,816
MIN4	MIN3 plus additional keyword and token types	91,829

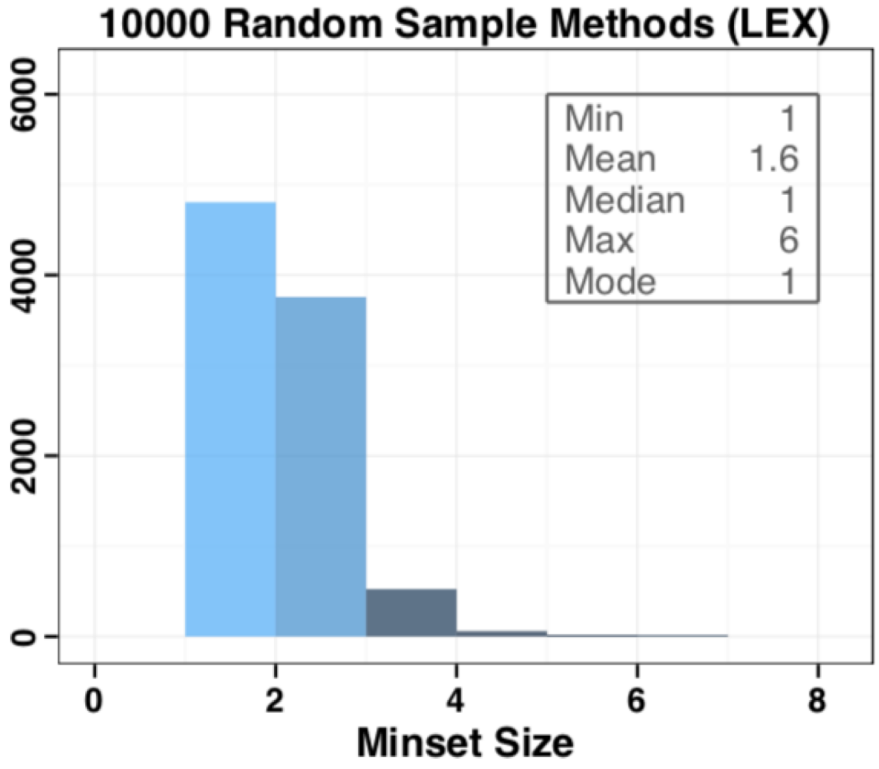
Lexicons



RQ1: How many units of code are lexically distinguishable?

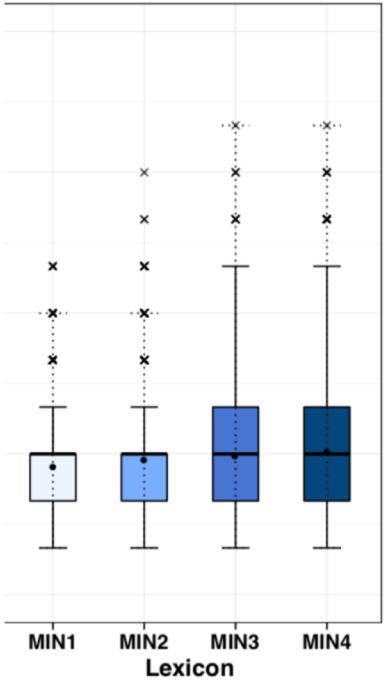
Over LEX, 91% are distinguishable. LTT is too coarse.

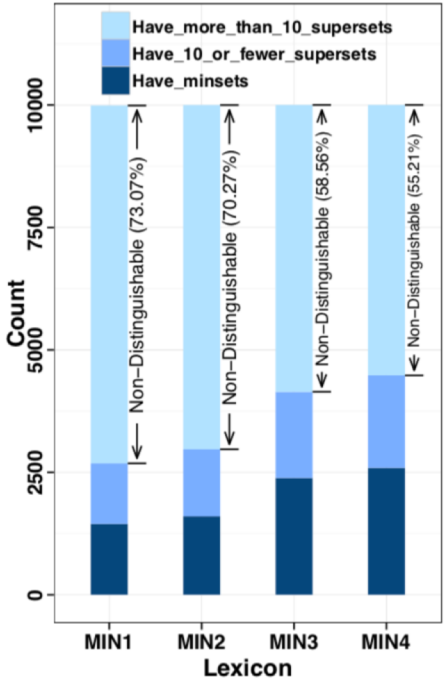




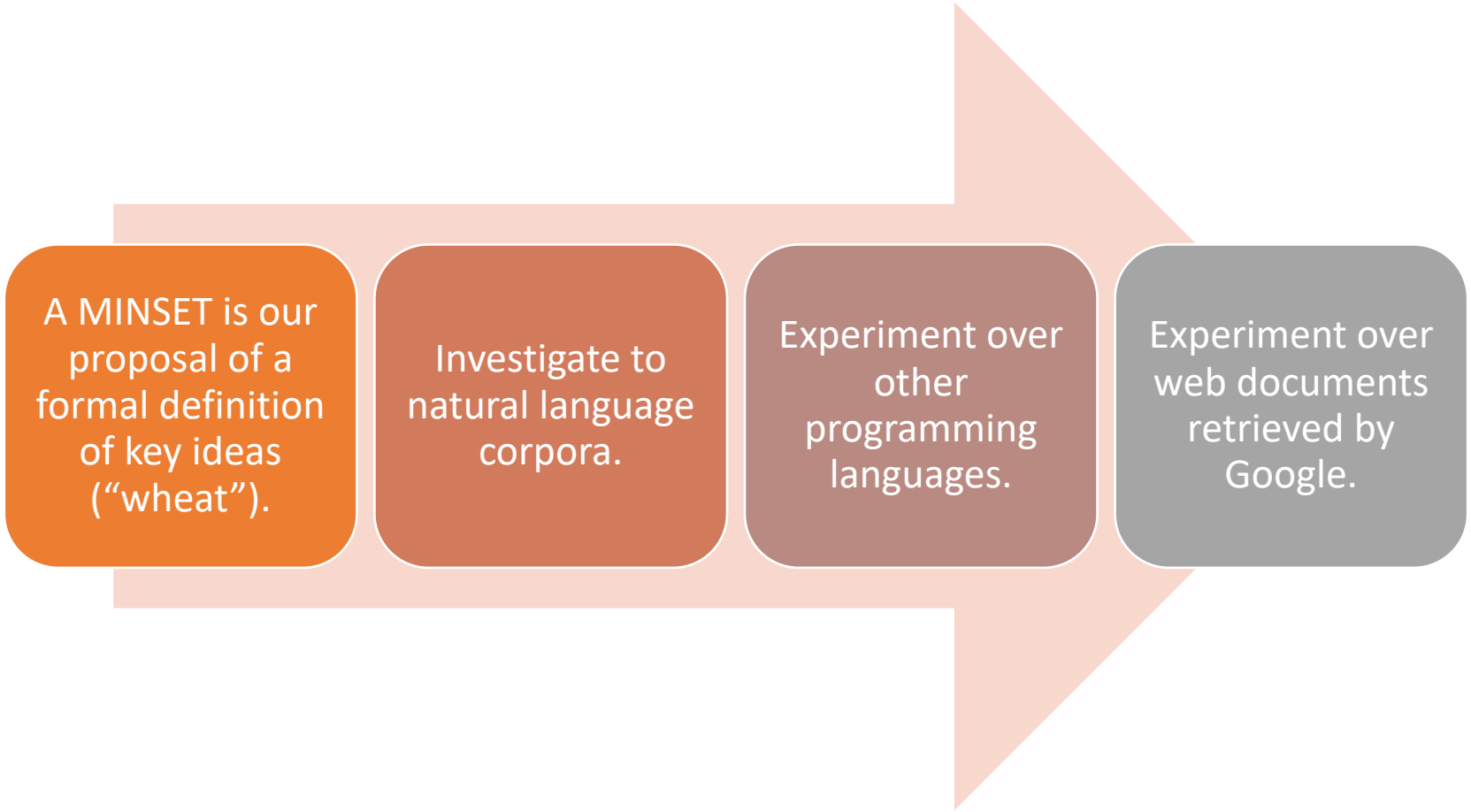
RQ2: How much of code is needed to distinguish?

Over LEX, 96% is chaff. Mean Minset size is 1.56 words.





- MIN4 is good in that
  - methods are still distinguishable (44.79%), and
  - minsets are still small (3.06%).
- With multiplicity and no abnormally large methods, 61.74% distinguishability.

RQ3: What is a natural, minimal lexicon?

Summary and Future Work