# Helping the Compiler Help You

Thomas Dy

# Programming

```
do {
  programmer.write_code();
  if(lazy) {
    sleep();
  }
  compile_code();
} while(compiler.has_errors());
```

# Compiler: Me no speaky English

- Programmer: “Compiler, what is 5 x 5?”
- Compiler: “Eh? What's 5 followed by an x?”
- Programmer: “You know, 5 *times* 5.”
- Compiler: “Eh? What's 5 followed by *times*?”
- Programmer: “5 * 5;”
- Compiler: “25”

# Novice Programmers

- Novice: “Compiler, what is 5 x 5?”
- Compiler: “Eh? What's 5 followed by an x?”
- Novice: *in a robotic voice* “What is 5 x 5?”
- Compiler: “Eh? What's 5 followed by an x?”
- Novice: “5 x 5”
- Compiler: “Eh? What's 5 followed by an x?”
- Novice: “Errr, uhhhh, ummm” *exit BlueJ*

# Literally non-literal

- “Bob and me study CS.”
- “Did you mean, Bob and I?”

- Non-literal errors
- “The house is red pretty.”
- “Did you mean, 'The house is red. Pretty.'?”

# Help the Compiler Help Who?

- Confused students
- Teachers of confused students
- Parents of confused students

# Others are Trying to Help Too

- QUT Framework
- Expresso
- HelpMeOut

# (Confused) Students

- Students of CS21a SY 2007-2010
- Introduction to Programming
- Java
- BlueJ

# The Computers Have Eyes

- Compilations are logged
  - Source code
  - Timestamp
  - Error message
  - ...and many more!
- BlueJ Extension sends code via network

# All Your Code Are Belong to Us

- BlueJ Browser, server web interface
- 

**View CompileData 3811**

| | |
|---|---|
| **Delta Session** | 2009-2010 cs21a a 3 f228_30 |
| **Timestamp** | 2009/07/23 13:47:14 |
| **File Name** | BankAccount.java |
| **Compile Successful** | No |
| **Message Text** | '{' expected |
| **Message Line Number** | 8 |
| **Compiles Per File** | 2 |
| **Total Compiles** | 2 |

```
1.
2.  /**
3.   * Write a description of class BankAccount here.
4.   *
5.   * @author
6.   * @version 1.1
7.   */
8.  public class BankAccount()
9.  {
10. balance = 1000;
```

# The Data

- Focus on top errors

| Error | Number |
|---|---|
| Unknown variable | 2772 |
| ';' expected | 1710 |
| '[', ']', '(', ')', '{', '}' expected | 1403 |
| Unknown method | 1382 |
| Incompatible types | 1131 |
| Missing return statement | 999 |
| Illegal start of expression | 762 |
| Unknown class | 617 |
| Identifier expected | 543 |
| Class or interface expected | 393 |

# Stuff We Did

- Start from compiler's output
- Check the source as well
- Look for patterns
  - <number> x <number>
  - <method definition> (
  - class <name> <name>

# Program Execution

```
compile_code();
if(compiler_error() == “; expected”) {
  // match known “; expected” errors
}
else if(compiler_error() == “missing var”) {
  // match known “missing var” errors
}
else if // check other error types
...
output_actual_error();
```

# Looks Promising

- Data sets
  - SY 07-08 for training
  - SY 09-10 for testing
- Generally ok

| Error | Percent Accuracy |
|---|---|
| Cannot find symbol | 86% (out of 100) |
| ';' expected | Not implemented |
| '(' expected | 44% (out of 100) |
| '(' or '[' expected and '[' expected | 100% (out of 80) |
| incompatible types | 87% (out of 100) |

# Problems?

- Lacking log data
- Accuracy could have been better
- Ambiguous errors
  - “The house is red pretty.”
    - “The house is red and pretty.”
    - “The house is red. Pretty.”

# Cascading Compiler Confoundment

- Errors tend to cascade
- “The house is red pretty.”
  - Missing '.' after “red”
  - “pretty” should be capitalized
- “5 x 5”
  - Missing ';' after first 5
  - “x” is not a statement
  - Missing ';' after “x”

# Being Lazy is Good

- Reuse data
- Reuse analysis techniques
- Let computer do the pattern matching

# No Such Thing as a  Free Lunch

- Data preparation is still important
  - Label the data
  - Write test cases
- Turn source code into “features”
  - Error class
  - Position of the error
  - Context

# Computers Can Learn

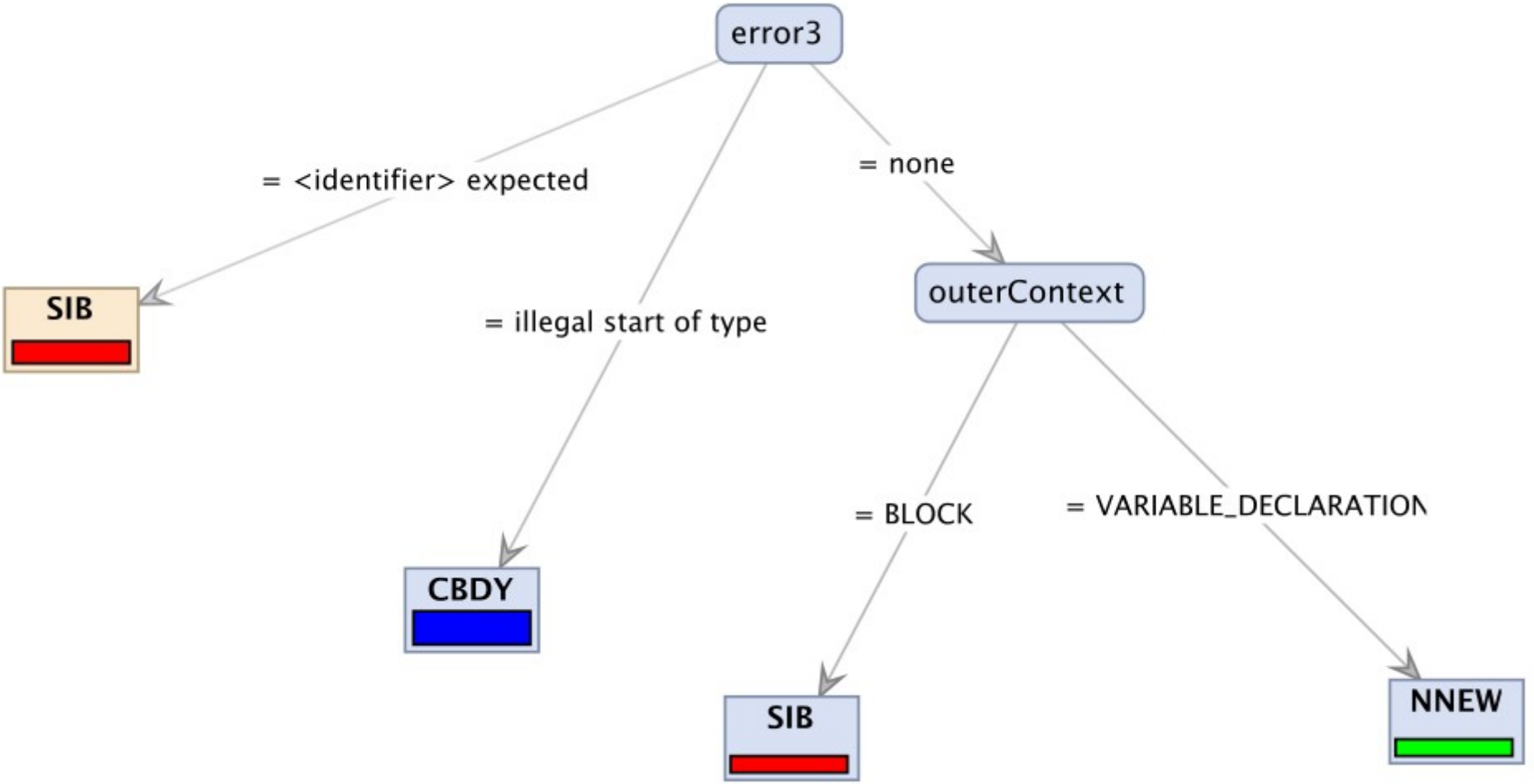
- Decision Trees
  - Flowchart-like model
  - If <something> go here, otherwise go there
- Rule Induction
  - List of rules
  - If <this> and <that> and <that>, it's <answer>
- Easy to implement in code

# Good News and Bad News

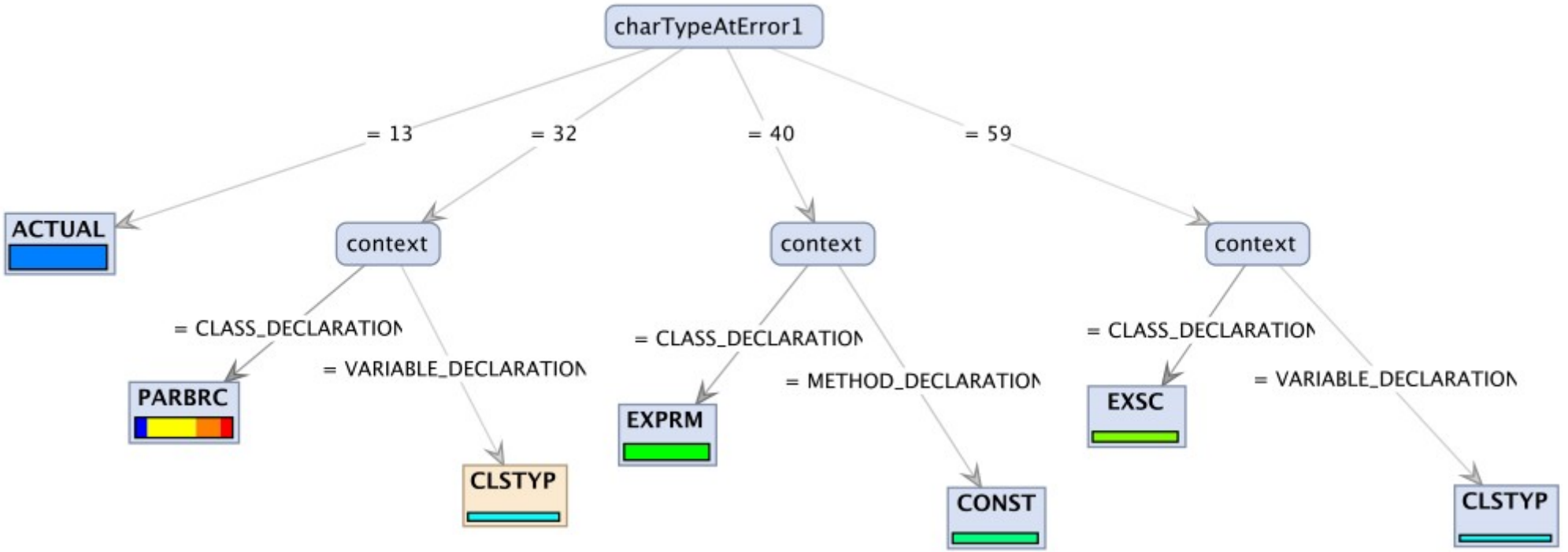
- Decision trees are marginally better
- Development set performance was amazing
- Actual test set performance was disappointing

| Error | % Acc. | Kappa |
|---|---|---|
| '{' expected | 57.00 | 0.443 |
| ')' expected | 57.58 | 0.380 |
| ']' expected | 25.00 | 0.057 |
| '(' or '[' expected | 58.97 | 0.259 |
| '[' expected | 60.00 | 0.444 |
| <identifier> expected | 50.00 | 0.339 |
| illegal start of expression | 61.00 | 0.368 |
| ';' expected | 70.26 | 0.318 |

# Why So Bad?


error3
= <identifier> expected
= none
= illegal start of type
SIB
outerContext
CBDY
= BLOCK
= VARIABLE_DECLARATION
SIB
NNEW

# But!


charTypeAtError1
= 13
= 32
= 40
= 59
ACTUAL
context
context
context
= CLASS_DECLARATION
= VARIABLE_DECLARATION
= CLASS_DECLARATION
= METHOD_DECLARATION
= CLASS_DECLARATION
= VARIABLE_DECLARATION
PARBRC
CLSTYP
EXPRM
CONST
EXSC
CLSTYP

# So Now, What Do?

- Fusion!
- Test out on a better dataset
- Filter out cascading errors

# Helping the Compiler Help You

Thomas Dy

## Programming

```
do {
  programmer.write_code();
  if(lazy) {
    sleep();
  }
  compile_code();
} while(compiler.has_errors());
```

In general, the process of programming is a cycle of writing source code and compiling it.

A programmer writes code, then compiles it. If there are any errors, he will go back and edit the code and compile it again. So on and so forth until you have a working program.

## Compiler: Me no speaky English

- Programmer: “Compiler, what is 5 x 5?”
- Compiler: “Eh? What's 5 followed by an x?”
- Programmer: “You know, 5 *times* 5.”
- Compiler: “Eh? What's 5 followed by *times*?”
- Programmer: “5 * 5;”
- Compiler: “25”

We might assume that a compiler will know how to correct all our mistakes. But as any seasoned programmer would know, this is not always the case.

Compilers are programs themselves and they tend towards more formal standards such as strict grammars as those make it easier for the compiler and eliminate ambiguity.

Neither can they read our minds and understand what we mean when we commit errors when we code.

## Novice Programmers

- Novice: “Compiler, what is 5 x 5?”
- Compiler: “Eh? What's 5 followed by an x?”
- Novice: *in a robotic voice* “What is 5 x 5?”
- Compiler: “Eh? What's 5 followed by an x?”
- Novice: “5 x 5”
- Compiler: “Eh? What's 5 followed by an x?”
- Novice: “Errr, uhhhh, ummm” *exit BlueJ*

In addition, it's even more problematic for novice programmers since they don't fully grasp the syntax of the language yet.

## Literally non-literal

- “Bob and me study CS.”
- “Did you mean, Bob and I?”

- Non-literal errors
- “The house is red pretty.”
- “Did you mean, 'The house is red. Pretty.'?”

We therefore categorize compiler errors as either literal or non-literal.

Literal errors are those where the compiler's message corresponds with the error. This is the case where the compiler just works, so all is fine and dandy.

Non-literal errors are those where the compiler's message does not point towards a solution to the problem that you have.

## Help the Compiler Help Who?

- Confused students
- Teachers of confused students
- Parents of confused students

The most to benefit from an informative compiler would be novice programmers who are just starting to learn the language. Non-literal errors can leave them in a state of helplessness which could turn them away from continuing to learn.

## Others are Trying to Help Too

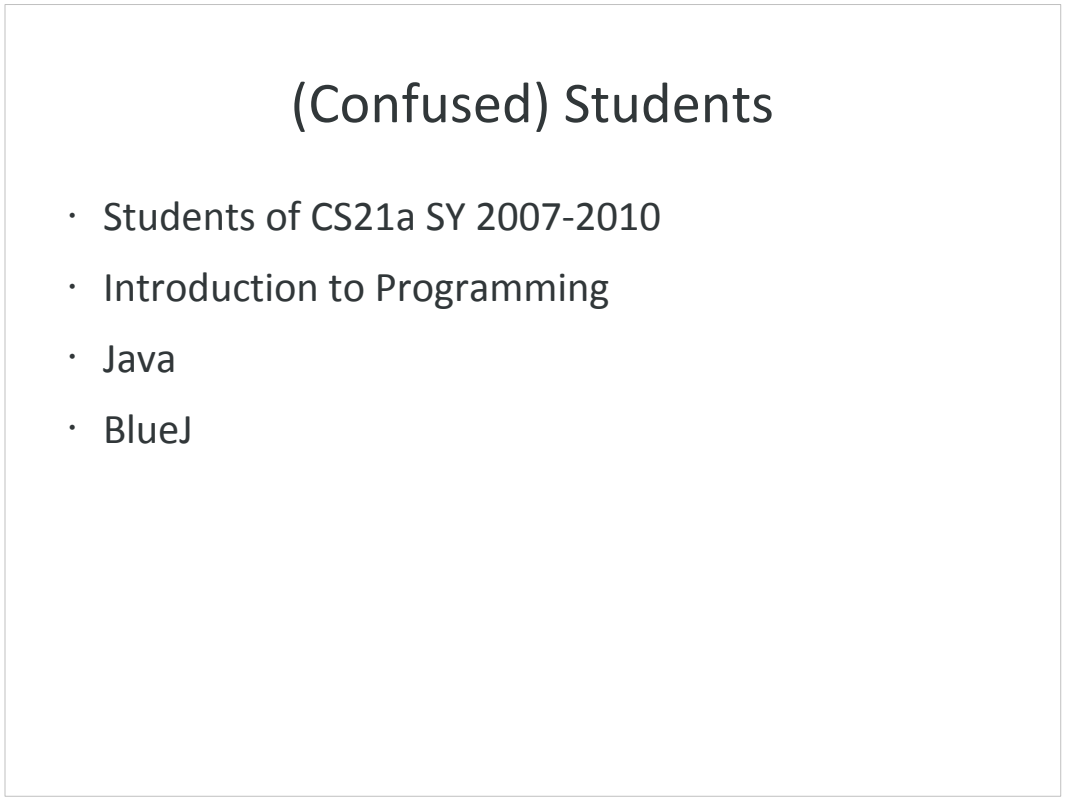
- QUT Framework
- Expresso
- HelpMeOut

There have been other attempts to solve the problem of non-literal errors.

QUT had developed a program that gives fill-in-the-blanks style programming so students don't have to bother with unneeded syntax at the moment thus reducing the chance of getting non-literal errors.

Expresso scans source code for common mistakes before passing it on to the compiler.

HelpMeOut crowdsources the detection and solutions of non-literal errors. Students' compilations are uploaded to a server. When a student encounters an error, the server checks if somebody else had encountered it and shows a fix for the said problem.




For the study, we used data from students taking CS21a, an introductory programming course. Our data spanned years 2007 up to 2010. In the course, the students are introduced to Java. In line with this, they use BlueJ and IDE made specifically for novice programmers.

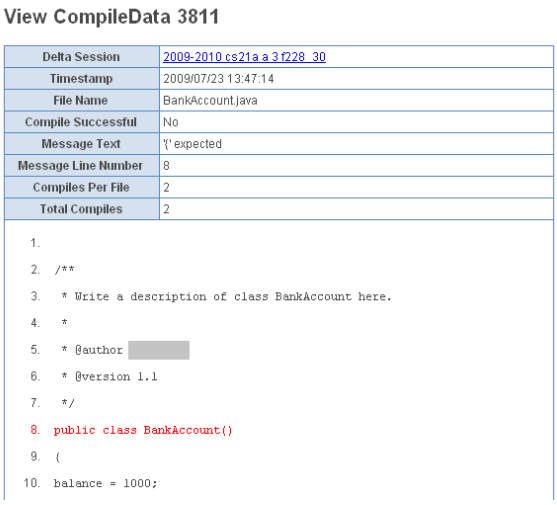
## The Computers Have Eyes

- Compilations are logged
  - Source code
  - Timestamp
  - Error message
  - ...and many more!
- BlueJ Extension sends code via network

We collected the students' compilation data. Every time a student compiles his or her code, it is sent to a server where this information is saved. Some information that is recorded are the actual source code, the time it was submitted, the error message, etc.

This was done via an extension to BlueJ. The extension was actually developed in the University of Kent by Matthew Jadud in line with his dissertation.

## All Your Code Are Belong to Us

- BlueJ Browser, server web interface
- 

View CompileData 3811

| | |
|---|---|
| Delta Session | 2009-2010 cs21a a 3 f228 30 |
| Timestamp | 2009/07/23 13:47:14 |
| File Name | BankAccount.java |
| Compile Successful | No |
| Message Text | '{' expected |
| Message Line Number | 8 |
| Compiles Per File | 2 |
| Total Compiles | 2 |

```
1.
2.  /**
3.   * Write a description of class BankAccount here.
4.   *
5.   * @author
6.   * @version 1.1
7.   */
8.  public class BankAccount()
9.  {
10. balance = 1000;
```

At the time, the logs were simply SQLite files. Think entire databases in a single file, or more loosely and Excel file. And each file corresponded to one student for one lab session. This is not exactly conducive for inspecting the logs.

To get over that problem, we also developed a web interface for the server, or the BlueJ Browser. It could take in the many log files and put them in a central place which would be easier to browse and even search for code.

Another nice thing we could do was to highlight the line which had the error. No longer did we have to copy out the code, paste it to Notepad++ and then check the line number.

Now, the BlueJ Browser also serves as the logging server instead of just importing it from the individual files.

## The Data

- Focus on top errors

| Error | Number |
|---|---|
| Unknown variable | 2772 |
| ';' expected | 1710 |
| '[', ']', '(', ')', '{', '}' expected | 1403 |
| Unknown method | 1382 |
| Incompatible types | 1131 |
| Missing return statement | 999 |
| Illegal start of expression | 762 |
| Unknown class | 617 |
| Identifier expected | 543 |
| Class or interface expected | 393 |

We obviously didn't want to look at all of the data. That would have been too time consuming and the effort spent on it would not have been worth it.

So first we took a look at the most frequently occuring errors and focused on those. Here is a list from the 07-08 data that we had. So for the first part, we did only the top 5 and later on the top 10.

## Stuff We Did

- Start from compiler's output
- Check the source as well
- Look for patterns
  - <number> x <number>
  - <method definition> (
  - class <name> <name>

So now that we have a target, we had to think of an approach to detecting non-literal errors.

First, we imagined that even if the compiler does give non-literal errors, it's still an amazing piece of software and it might be *somewhat* right if not exactly. So we worked off what the compiler outputs.

Another thing we would need to do is to read the source files, otherwise we really wouldn't have anything to base our detection on.

Then, we looked for patterns in the errors that were non-literal. Stuff like, oh lots of people do a 5 x 5 or a 6 x 2 so maybe we should detect that. So we'd program a detector for sequences of <number> x <number>.

## Program Execution

```
compile_code();
if(compiler_error() == “; expected”) {
  // match known “; expected” errors
}
else if(compiler_error() == “missing var”) {
  // match known “missing var” errors
}
else if // check other error types
...
output_actual_error();
```

The general flow of our detector is first we compile the code using the compiler. Then we check what the compiler outputs. If it's some error, say semicolon expected. We then match all non-literal errors that we know will occur given a semicolon expected error. We did this for the top 5 errors described earlier.

# Looks Promising

- Data sets
  - SY 07-08 for training
  - SY 09-10 for testing
- Generally ok

| Error | Percent Accuracy |
|---|---|
| Cannot find symbol | 86% (out of 100) |
| ';' expected | Not implemented |
| '(' expected | 44% (out of 100) |
| '(' or '[' expected and '[' expected | 100% (out of 80) |
| incompatible types | 87% (out of 100) |

The results seem okay...

## Problems?

- Lacking log data
- Accuracy could have been better
- Ambiguous errors
  - “The house is red pretty.”
    - “The house is red and pretty.”
    - “The house is red. Pretty.”

Some problems we had with this approach were that the log data we used wasn't exactly complete. For example, we only had data on what line the error occurred in but not the exact position.

Accuracy could have been better. We could also have included more errors into the detection.

We also have the problem of ambiguous errors. It might be possible to fix something in different ways and both are perfectly valid. You can tell however which one is more correct based on the context of the code, but this reasoning is difficult to put into a program.

## Cascading Compiler Confoundment

- Errors tend to cascade
- “The house is red pretty.”
  - Missing '.' after “red”
  - “pretty” should be capitalized
- “5 x 5”
  - Missing ';' after first 5
  - “x” is not a statement
  - Missing ';' after “x”

So we decided to take a different approach. One thing we noticed was that errors tended to cause other errors after them. For example, missing the “and” in “The house is red and pretty.” Might make the compiler mistake the sentence to end right after “red” this would then make the compiler mistake “pretty” to be the start of a sentence and therefore should be capitalized.

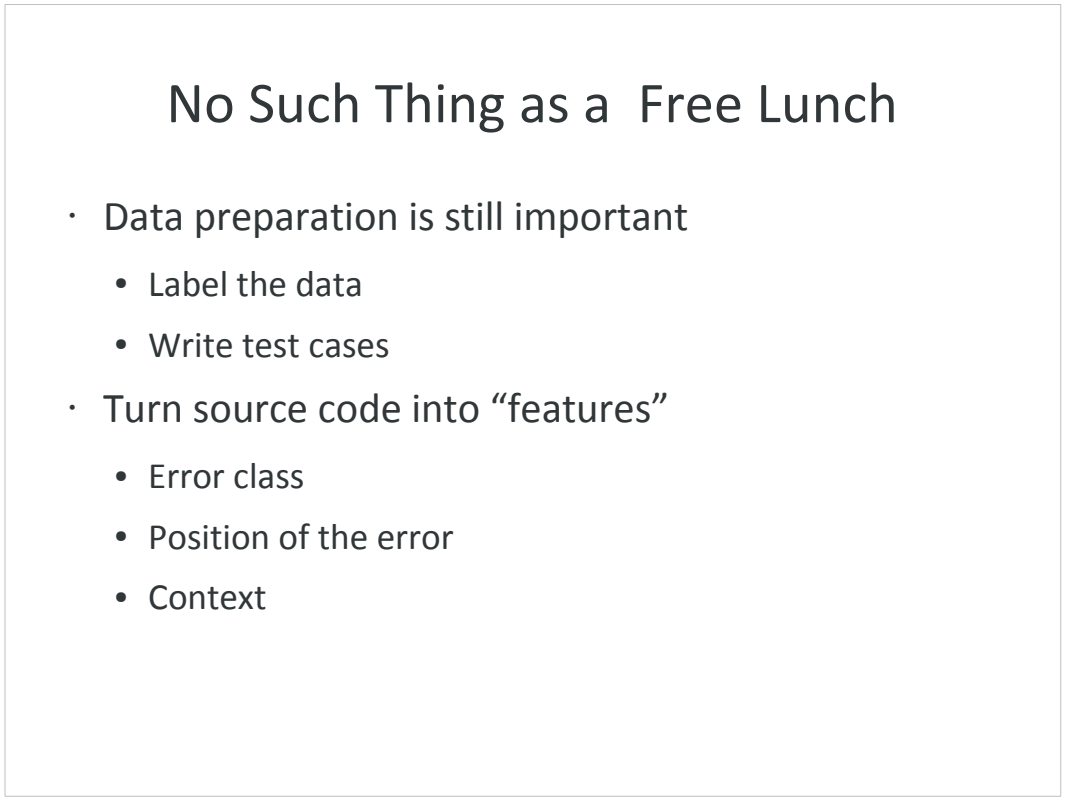
These errors might be called cascading errors.

## Being Lazy is Good

- Reuse data
- Reuse analysis techniques
- Let computer do the pattern matching

Maybe it would also be good to be lazy. We could reuse the data that we already have. We could reuse the analysis techniques that we did. Checking the top errors, and then focusing on that. And also still working off the compiler.

We could also let the computer do the pattern matching instead of us manually doing it. Data mining techniques exist just for that particular problem.




It's not all rainbows and butterflies with data mining though. We would still need to at least label the data to serve as examples for the computer to learn. We would also need to write test cases to better isolate the errors we want to study. This served as the development set we would always test on.

Another thing we needed to do was to convert the source code into features or simple exact descriptions of the code as source code is too free form and that doesn't work too well with data mining. So a description of a source file might be something like what the error class is, since the error message has some other information in it like variable names and etc.

## Computers Can Learn

- Decision Trees
  - Flowchart-like model
  - If <something> go here, otherwise go there
- Rule Induction
  - List of rules
  - If <this> and <that> and <that>, it's <answer>
- Easy to implement in code

We used 2 data mining algorithms with our data. One is decision trees which creates a flowchart like model. If you've ever seen something like “What to do in a fire.” You might have something like, “Is the fire big?” If yes, get out now. If no, “Is there a fire extinguisher in the room?” If yes, try to put it out. If no, etc.

We also have rule induction which is more like a set of laws. “If the fire is small and there is a fire extinguisher in the room, try to put it out.”

What's nice about these algorithms is that the models they make can be easily put into code. You can just use if-statements to do it.

## Good News and Bad News

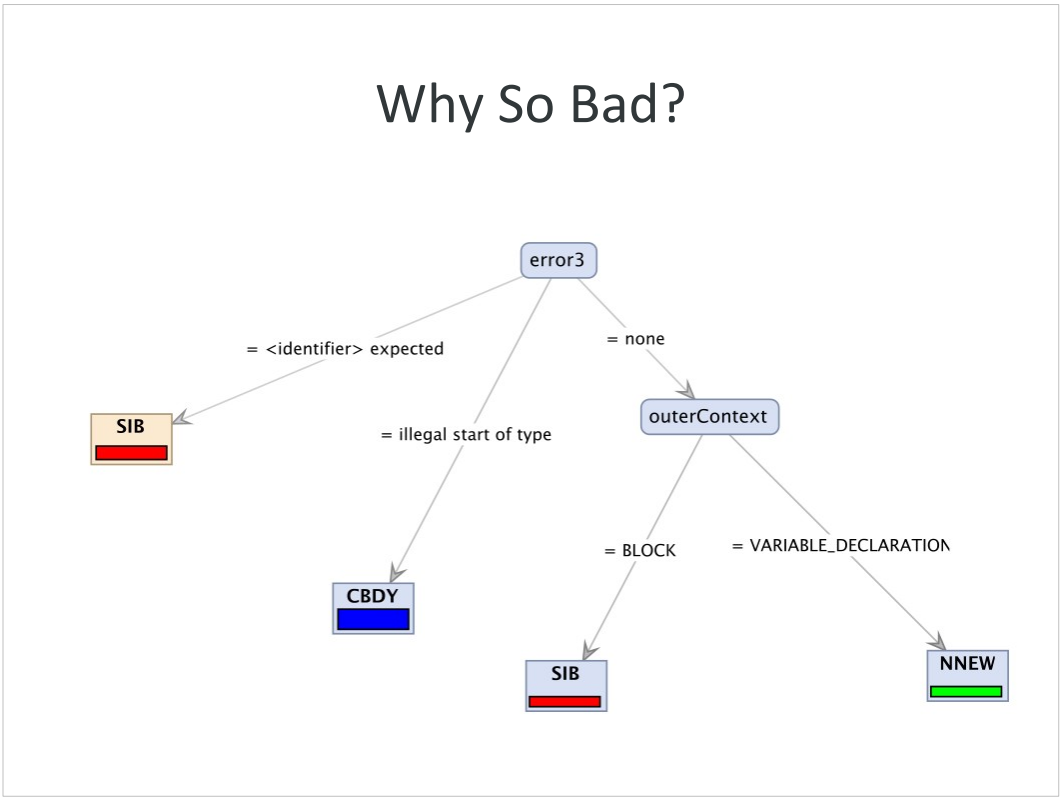
- Decision trees are marginally better
- Development set performance was amazing
- Actual test set performance was disappointing

| Error | % Acc. | Kappa |
|---|---|---|
| '{' expected | 57.00 | 0.443 |
| ')' expected | 57.58 | 0.380 |
| ']' expected | 25.00 | 0.057 |
| '(' or '[' expected | 58.97 | 0.259 |
| '[' expected | 60.00 | 0.444 |
| <identifier> expected | 50.00 | 0.339 |
| illegal start of expression | 61.00 | 0.368 |
| ';' expected | 70.26 | 0.318 |

After evaluating both algorithms, we found the decision tree performed marginally better so we only used that for the rest of the tests.

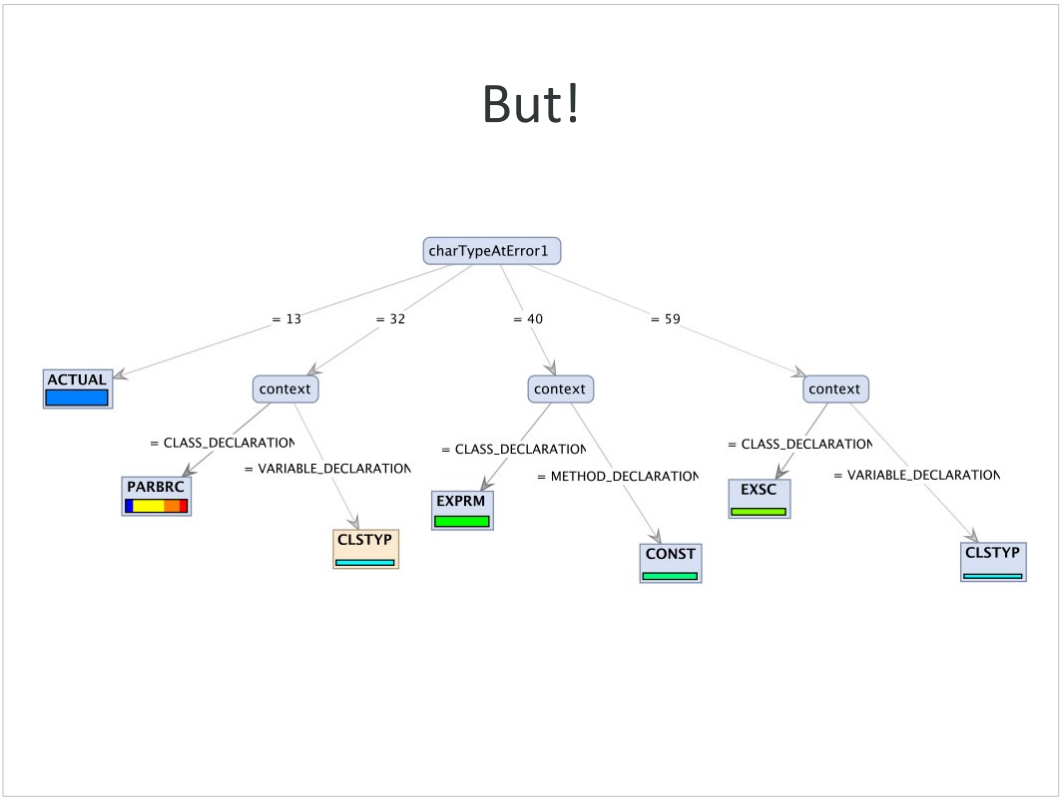
The results were a mixed bag, but more of a disappointment. Obviously, results on the development set were great since that's what we were using to improve the detector.

The actual test set though, which was composed of random labeled student code, was bad.




One reason why it's bad is that the development set we were working with didn't really reflect the actual conditions of code. The code in the development set is comprised of source files with only one error each. In order to better isolate them. However, the models that we get rely on this fact. Hence the low accuracy.

We can remedy this by diversifying the development set by including source files with more than 1 error.




Notice though, how some errors tend to clump together under just one branch. This does not exactly contribute to accuacy for the detector. But for those errors, we can actually use what we developed earlier. By mixing the two techniques, we can save time by having a detector for some errors be automatically generated. And then just manually doing whatever is left.

## So Now, What Do?

- Fusion!
- Test out on a better dataset
- Filter out cascading errors

So, what we can do is, try out combining the 2 techniques and seeing where that takes us. It might also be better to get better datasets to test and develop on.

And if possible, a really nice application for this is to filter out the cascading errors so we're only left with the actual errors we have to fix.