Common Abstract Syntax Tree (AST) for Automated Software Analysis of Homework Assignments in Submitty

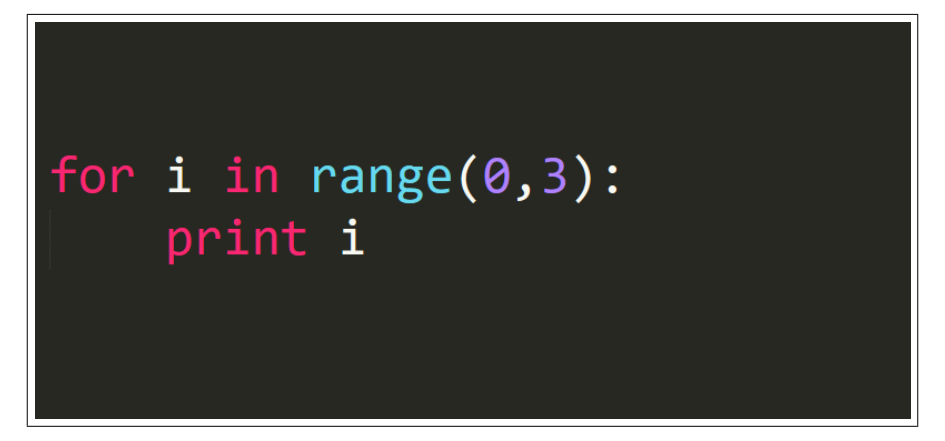
Elizabeth Dinella, Ana Milanova, and Barbara Cutler

Rensselaer Center for Open Source

Static Analysis

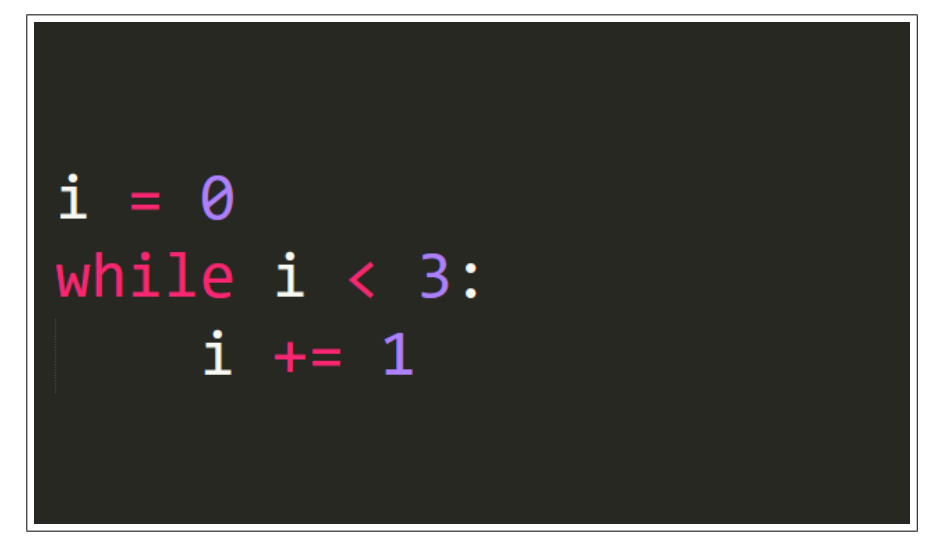
Static analysis is a critical tool to ensure software security and dependability. In industry and academia, static analysis can be used to find potential bugs or design problems. On "Submitty", an open source homework server created at RPI, static analysis is used to ensure structural correctness on homework assignments. This functionality is used in RPI's Computer Science 1 (CSCI 1100) to automatically grade small in-lecture exercises. Information beyond the textual output of the program would be valuable to assess knowledge. For example, students in CS1 are typically asked to "rewrite" a simple for loop using a while loop (figure a). In this exercise, the code containing a for loop, and the code containing a while loop would have the same textual output. However, it would be impractical for each assignment to be individually graded by the teaching staff. Static analysis tools are, in some cases, necessary to accurately asses student's progress.

Additionally, static analysis is superior to simplier alternative methods. For example, running grep on the student's code could cause false positives. This situation is shown in figure c. Aditionally, static analysis can be used to detect more complex structures such as class heirarchies and exception handling.



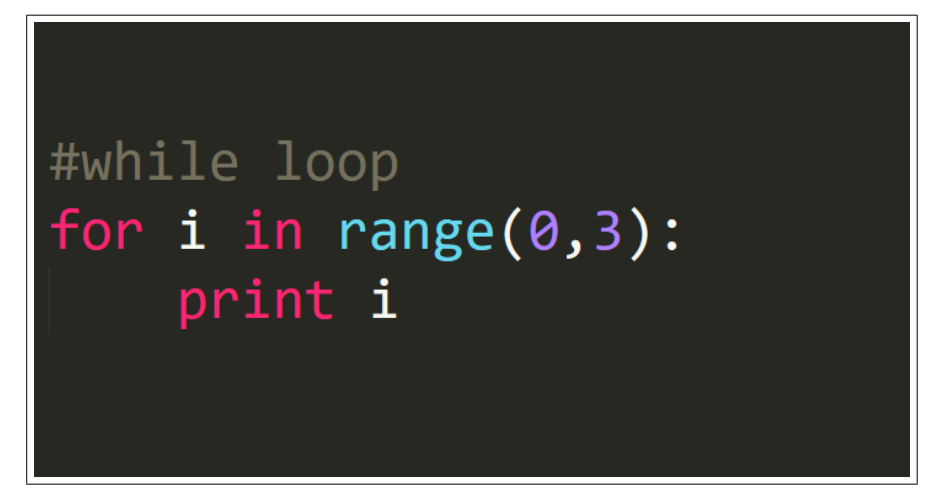
```
for i in range(0,3):
    print i
```

(a) Initial Code



```
i = 0
while i < 3:
    i += 1
```

(b) Correct Student Rewrite



```
#while loop
for i in range(0,3):
    print i
```

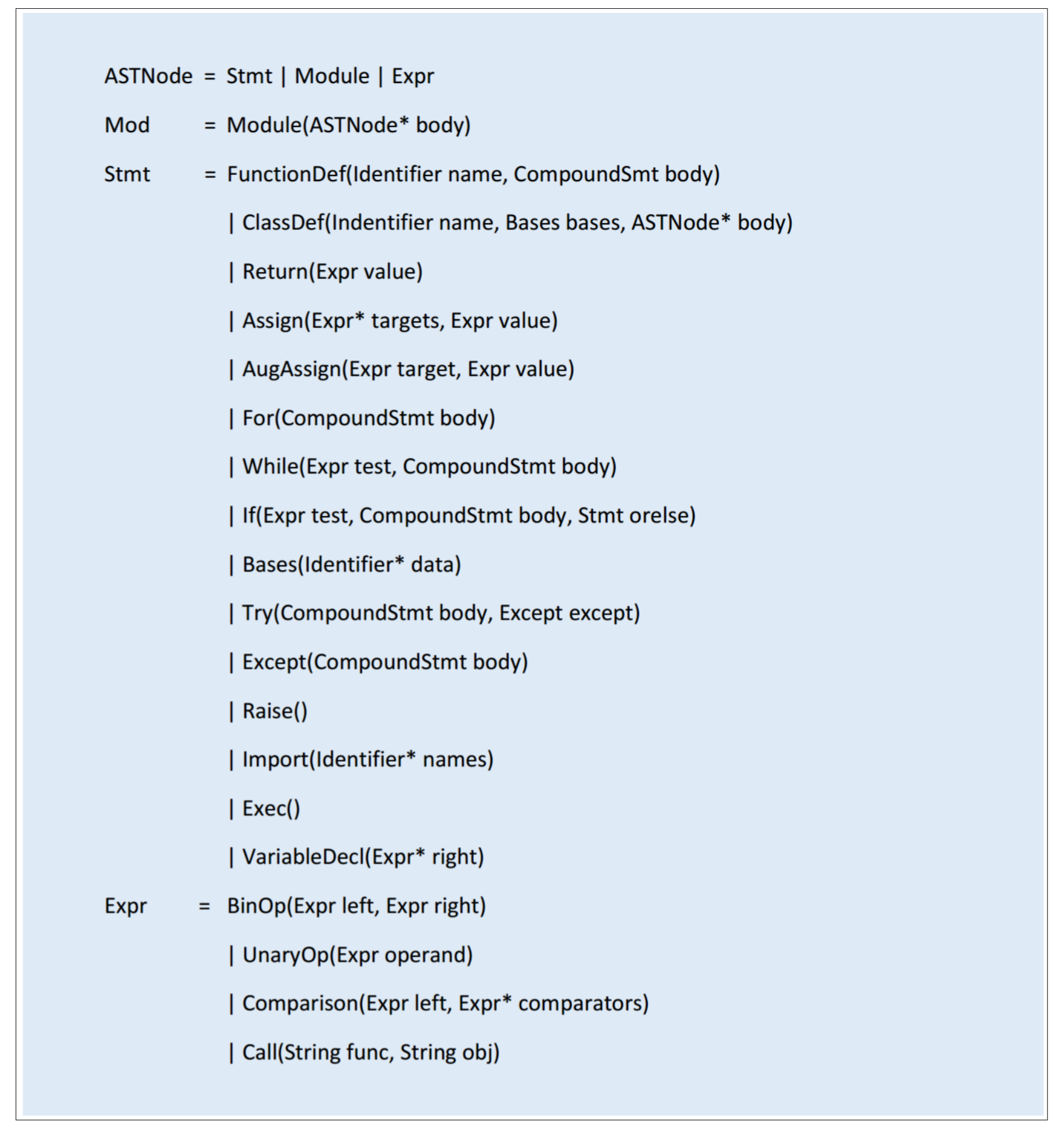
(c) Incorrect Student Rewrite

Motivation

Static analysis tools have been requested in many other courses. These curriculums include multiple programming languages such as Python, C++, and Java. To extend simple static analysis of while/for loop detection to C++ and Java, the entire framework must be re-written in each language. This requires a large code base that depends upon extensive development each time a new language is used. It may seem excessive to create an entire new framework for each language, but it is necessary due to the uniqueness of each languages intermediate code representation. During parsing and semantic analysis, an Abstract Syntax Tree (AST) is created to represent code structure. In each language, there are drastic AST differences.

Introduction

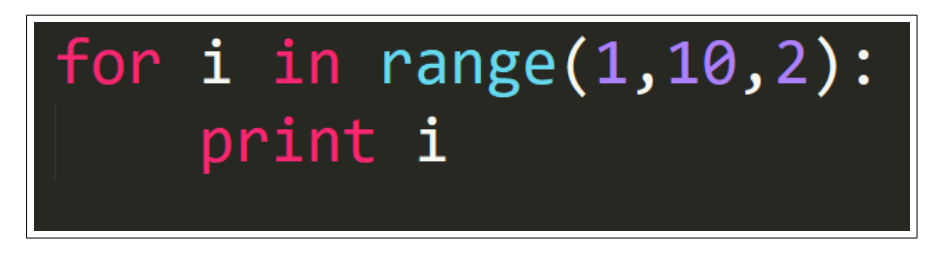
Our project builds a common AST which captures the structural similarity of the different ASTs and covers the use cases for static analysis on Submitty. The common AST is extracted from C++ and Python code and is stored in a standardized XML format. Extraction occurs using clang/llvm AST matchers and the Python AST module. This framework allows a simple process for adding static analysis cases in any programming language. The common AST captures relevant structures from Python and C++.



```
ASTNode = Stmt | Module | Expr
Mod     = Module(ASTNode* body)
Stmt    = FunctionDef(Identifier name, CompoundSmt body)
          | ClassDef(Indentifier name, Bases bases, ASTNode* body)
          | Return(Expr value)
          | Assign(Expr* targets, Expr value)
          | AugAssign(Expr target, Expr value)
          | For(CompoundStmt body)
          | While(Expr test, CompoundStmt body)
          | If(Expr test, CompoundStmt body, Stmt orelse)
          | Bases(Identifier* data)
          | Try(CompoundStmt body, Except except)
          | Except(CompoundStmt body)
          | Raise()
          | Import(Identifier* names)
          | Exec()
          | VariableDecl(Expr* right)
Expr    = BinOp(Expr left, Expr right)
          | UnaryOp(Expr operand)
          | Comparison(Expr left, Expr* comparators)
          | Call(String func, String obj)
```

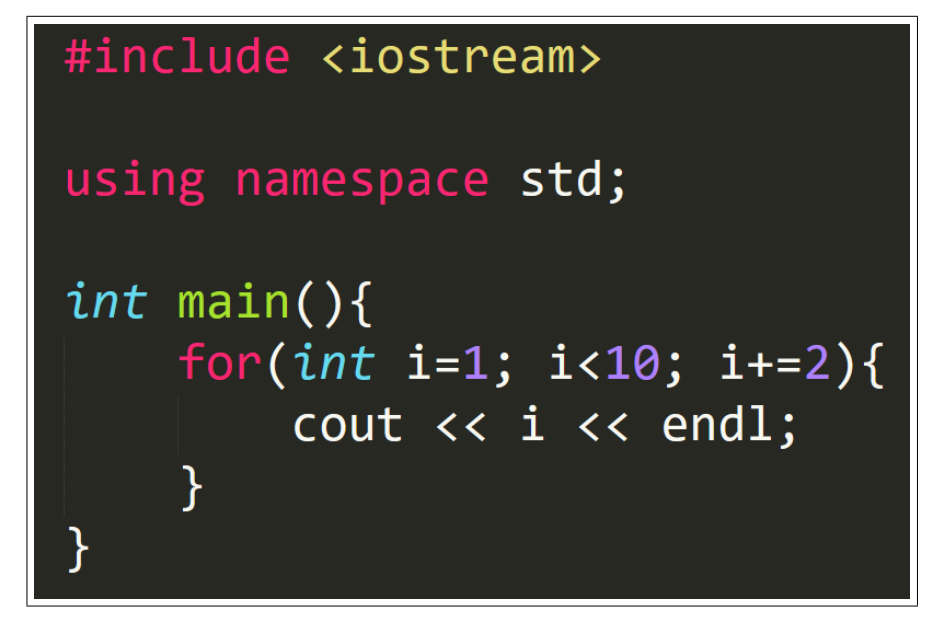
(d) Common AST

Consider the following motivating example: A student in an introductory computer science course is learning about for loops. A homework problem is assigned to test the student's knowledge. The assignment requires that the student creates a loop in Python or C++ that prints all odd numbers between 1 and 10. By only checking textual output, Submitty would not verify that the student's code actually includes any for loops. Without static analysis, a non-automated TA or instructor grade would be required. Consider the following correct solutions to this assignment:



```
for i in range(1,10,2):
    print i
```

(e) Python Solution



```
#include <iostream>

using namespace std;

int main(){
    for(int i=1; i<10; i+=2){
        cout << i << endl;
    }
}
```

(f) C++ Solution

Standardized XML Format

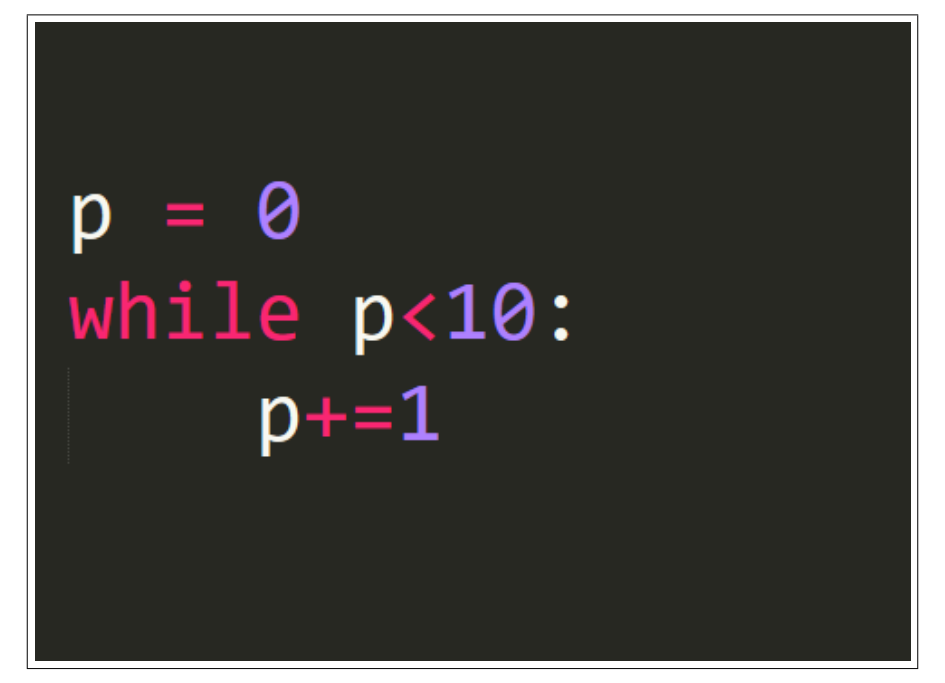
The first phase in a compiler is lexical analysis. In this phase, a compiler that strips all irrelevant information such as comments and whitespace from the source code. It then breaks the code into a series of tokens to be used in further phases in compiling.

We begin with a process similar to lexical analysis. In the phase, we:

- Remove all information that is irrelevant to the Submitty use cases
  - Like lexical analysis, comments and whitespace are also removed
- output a sequence of tokens in the following format: `<node, level>`
  - `node` is the name of the corresponding structure in the source code. It may represent any of the productions in the common grammar (figure d).
  - `level` represents the depth in the syntax tree. For example, in figure d, the while loop is a child of the outer python module. So, the while loop has a level of 2. The while loop holds its body as a child. The body contains one child, the augmented assignment of p. So, "p++" has a level of 4, and the token reads: `<augAssign,4>`

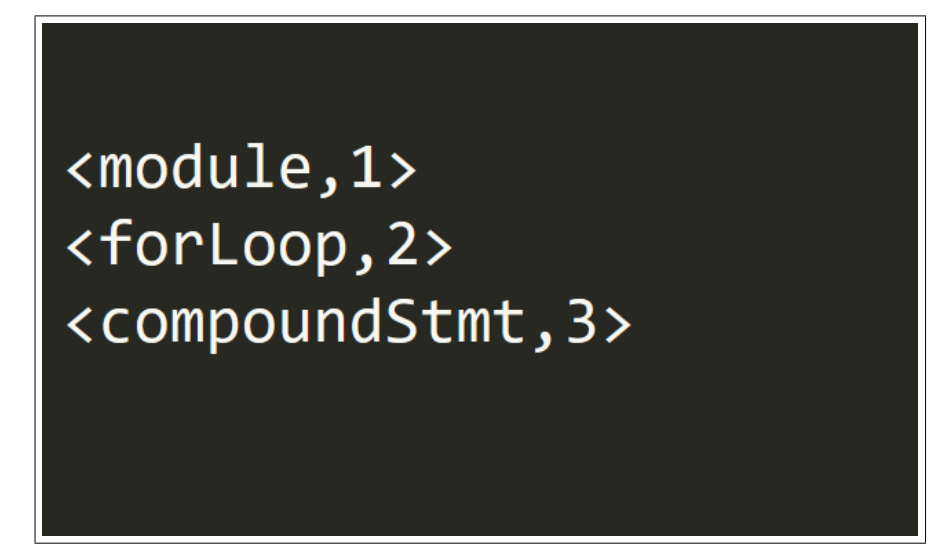
This format is created using third party libraries that allow users to process the abstract syntax trees of external programs. For C++, Clang/LLVM AST Matchers are used to traverse the AST of an input file. During traversal, when a node relevant to the use cases is found, a token is created and printed to an output file. A similar process occurs for Python using the Python AST library.

Figures e and f show the standardized XML output from the two assignment solutions.



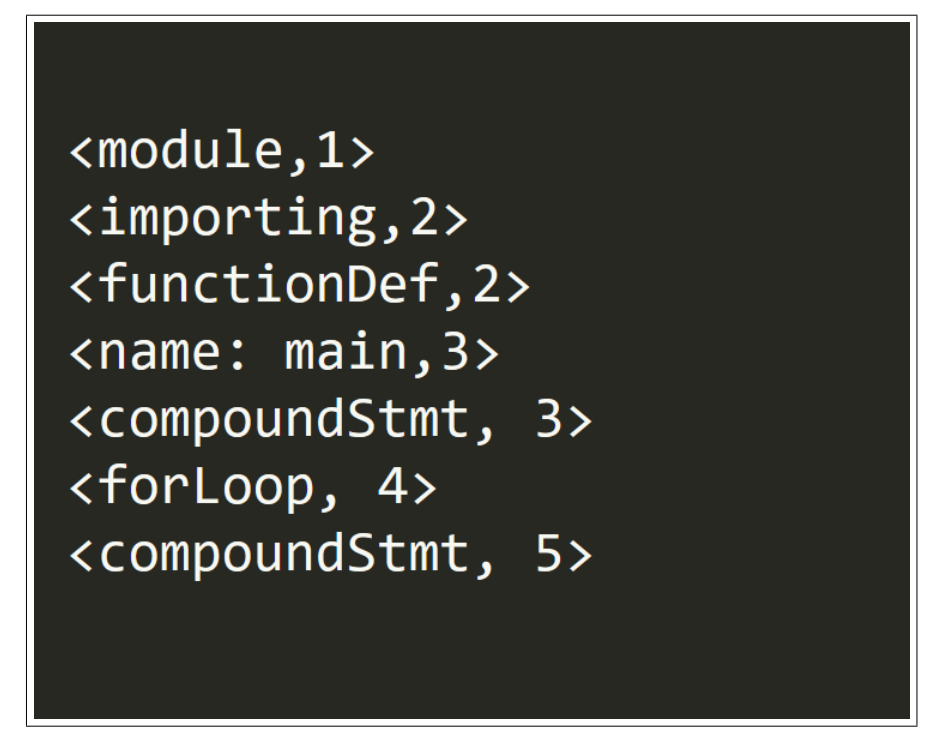
```
p = 0
while p<10:
    p+=1
```

(g) Level Example



```
<module,1>
<forLoop,2>
<compoundStmt,3>
```

(h) Python Standardized XML



```
<module,1>
<importing,2>
<functionDef,2>
<name: main,3>
<compoundStmt, 3>
<forLoop, 4>
<compoundStmt, 5>
```

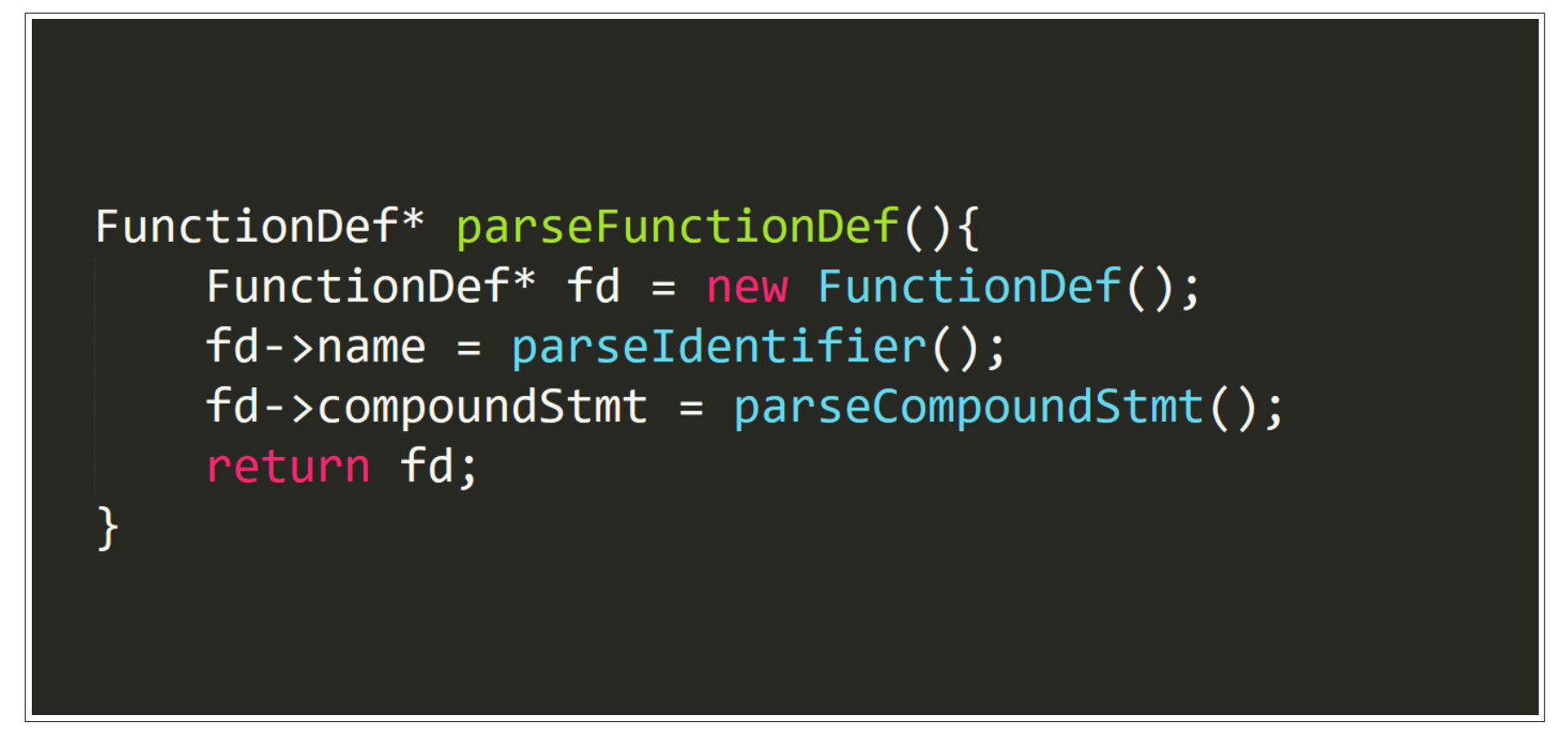
(i) C++ Standardized XML

Recursive Decent Parser

After lexical analysis, a compiler typically performs syntax analysis. In this phase, tokens from lexical analysis are taken as input. The tokens are compared to the grammar and a syntax tree is generated.

We again, perform a similar process to recursively create a syntax tree in a depth first top-down order.

- The standardized XML output from the first step is used as input
- At each level of the recursion the token in the AST is dispatched to an appropriate processing function based on the node type
  - There is a processing function for each production that strongly resembles the grammer. For example, the production for a FunctionDef shown in figure d, has an Identifier for its name and a CompoundStmt for its body. The processing function for a FunctionDef is shown in figure j. In each processing function, a class of type corresponding to the input token is created. The member variables are then initialized recursively. In this way, a node's children are stored in a has-a pointer relationship.



```
FunctionDef* parseFunctionDef(){
    FunctionDef* fd = new FunctionDef();
    fd->name = parseIdentifier();
    fd->compoundStmt = parseCompoundStmt();
    return fd;
}
```

(j) FunctionDef Processing Function

Use Cases

The following use cases were determined by considering input from instructors using Submitty:

- Detecting or counting if statements
  - Nested if statements
  - Dangling else cases
- Detecting or counting for/while loops
  - Loops containing nested if statements
- Detecting language specific keywords such as "goto", "malloc", "auto", etc
- Detecting member function calls from an outside class such as "vector.erase()"
- Counting a specific function call
- Detecting exceptions and ensuring a corresponding handler
- Detecting class hierarchies and relationships

Ongoing Work

- Implementation of use cases
- Large scale testing on RPI Data Structures (CS 1200) and RPI Computer Science 1 (CS 1100) assignments
- Possible extension of framework to include Java

Related Submitty Publications

- *User Experience and Feedback on the RPI Homework Submission Server*, Wong, Sihsobhon, Lindquist, Peveler, Cutler, Breese, Tran, Jung, and Shaw, SIGCSE 2016 Poster
- *A Flexible Late Day Policy Reduces Stress and Improves Learning* Tyler, Peveler, and Cutler, SIGCSE 2017 Poster
- *Submitty: An Open Source, Highly Configurable Platform for Grading of Programming Assignments*, Peveler, Tyler, Breese, Cutler, and Milanova, SIGCSE 2017 Demo Presentation

Acknowledgements

- Red Hat Software
- Rensselaer Center for Open Source (RCOS)
- `http://submitty.org/`
- `https://github.com/Submitty/Submitty`