# QA Language Compiler Opimizations: By Ben Ni and Dmitry Vasin

## Optimization 1: Dead Code Analysis

### Brief description

Code that can never be reached is removed from the compiled program

### Motivation

If code can never be executed then it does nothing for the program and can be removed with no side effects. Removing the dead code will make the program smaller and at the same time not impact the correctness. At the same time there's no confusion for the user because from their perspective both the testcase and the execution of the script looks the same.

### Implementation/Description

The most effective way we thought of that we can do this is to build a graph of all the nodes. A node is either a function or a step. The first node is the main function. The parent is the node from which the function or the step was called, and the child is the function or step being called. For example for something like

Step 1: go to http://google.com |

Step 2: go to http://youtube.com |

You would have a graph of main->step1->step2. This process is complicated by the existence of go to statements, functions, and if statements, so all three have to be handled separately. Before outputting the program we run through the entire code to build our graph. You cannot know if a node is live until all the code has been traversed because later steps can refer to earlier steps.

After the graph is built we run a breadth first search starting from the main function, as anything that cannot be reached from the main function is dead code. Then we simply step through the input like we did before, but check if the node we are currently looking at is live. If it is, we add it to the final program.

## Optimization 2: Collapsing Steps (Inlining)

### Brief description

Due to the lack of gotos in python, each step is a separate function. If two steps are always executed consecutively then we can collapse them into a single step within the program.

### Motivation

Since each step calls the following step, the amount of used stack space can get quite large. This can be minimized by inlining the steps (collapsing two steps into one) which do not need to be separate functions. This is a relatively frequent case so we expect it will benefit the program significantly, especially in very large test cases.

### Implementation/Description

During the scanning phase of optimization 1 we also build a list of nodes that are in either goto statements or in loop statements. You cannot collapse a step if it is the destination of a goto statement, and you cannot collapse the node OR its direct child if it is a loop statement. This creates unintended behaviour. When we are building the final program we check to see if the node is the target of a loop statement, goto statement, or a child of a loop statement. If none of these are true then we collapse the steps into one function.

**Examples**

DeadCodeTest01.txt: Despite many gotos, exits and if statements with two gotos, the only dead code is the refresh13 function

DeadCodeTest02.txt: Everything beyond step 2 is dead due to an exit

DeadCodeTest03.txt: The goto and exit combined mean that only steps 1,2 and 10 are live.

DeadCodeTest04.txt: Tests that the step after an if statement with two gotos is dead unless called

DeadCodeTest05.txt: Checks that a step following an if statement with only one goto is live.

ConcatCodeTest01.txt: tests all aspects of collapsing steps, if statements, gotos, and loops

ConcatCodeTest02.txt: basic step collapsing

**Script:**

run Sprint4\qaCompiler\buildv3.bat to regenerate the IR if needed

go to Sprint4\qaCompiler\src\PythonBackend

To compile the code run: python backend.py [IR.xml] [outputFileName] [preprocessorFileName]

For example to compile test case ConcatCodeTest01.txt run

python backend.py concatCodeTest01.html.xml concatCodeTest01.py

The command to compile all the testcases is in compileS4Test.bat so you can just use that instead.

The compiled python code is located at: Sprint4\qaCompiler\outputs\CompiledCode with the filename you provided.

**Quality of Documentation:**

All the necessary components like description, motivation and implementation are outlined above.

**Correctness and Robustness:**

Our test cases cover each of the possible edge cases for dead code and collapsible steps, namely gotos, functions, and if statements. In terms of if statements it checks for ifs with two gotos/functions (making the next step dead) and 1 or less goto/function (making the following step alive).

**Verifiability:**

You can verify it by running it on our testcases.