QA Language Compiler Sprint 3: By Ben Ni and Dmitry Vasin

We have translated our compiler from XQuery to python. The updated code is located at Sprint3\qaCompiler\src\PythonBackend

Type Extension: Preprocessed Variables

Brief description:

Allows a developer to specify preprogrammed elements which can later be used in the test case.

Motivation:

When we were making testcases for our program we found it very difficult to find websites which had simple elements described by our language, for example buttons that had the text “search” as the label. For example, youtube’s search button has a picture of a magnifying glass as a label for its search button. How do you programmatically specify that in an English readable manner? For our earlier versions we turned to using IDs but that violated our principles of non-programmers being able to understand our language easily. Thus we came up with our idea of a preprocessor.

Implementation/Description:

Our solution to the problem was allowing the developers to describe complex entities in a preprocessor file, which the users can later use. For example a developer can specify that a search button has the ID of “x879asd6” and a pixel width of 80. He/She can assign that button to the label “SearchButton” which abstracts away all the under the hood tech information from the user. The user would simply see the step say “press predefined SearchButton”. Thus they do not need to look for IDs in the source code, or check the pixel size, but only know which label the button refers to, which should be relatively clear from the title. The compiler then find instances of predefined buttons and pulls the actual button from the code written by the developer.

Type Extension: Error Checking

Brief description:

We detect some common sense errors which a developer would expect the compiler to catch

Motivation:

We wanted our compiler to be more useful so we decided to add basic error checking. If errors are caught then the python script is still generated but instead of following the steps it simply prints which errors are in the testcase.

Implementation/Description:

There are a number of error checks we have implemented. The first one is checking whether the variants of our IR are all specified by our language. If we receive an unexpected variant of a statement, or result function, we let the user know. Another error check we have is the user trying to run functions which don’t exist in which case we recommend them to create it. A third type of error checking is whether or not the website is loaded before we try to access elements on it. It tells the user this cannot be done. We also check if the steps are out of order and if so which steps do not conform. All of these

checks are done with simple if statements and a global variable which keeps track of errors, that way we output all the error at once rather than saying one error occurred and the next time the compiler is ran another one occurred.

Optimization: Dead Code

Brief description:

Code that is never executed shows a warning to the user.

Motivation:

It's possible that some of the code we have is never executed and thus if it is removed the program will run faster and be more understandable. For example a program like

step1: exit |

step2: go to http://google.com |

could have the second step eliminated with no effect on the program. There are far more complex cases though for example

step 1: go to step 6|

step 2: exit |

step 3: go to http://test.com |

step 4: go to http://youtube.com |

step 5: go to http://test.com |

step 6: if current webpage contains "Search" go to step 4 otherwise go to step 5

In this case only step 3 is dead code but it requires in depth analysis to see that.

Implementation/Description:

In order to do this we needed to create a graph between all the nodes of the code. The parent is the caller and the child node is the one being called. As we step through the code we add elements to the graph and connect them accordingly. In the end, if starting from the very first main function call some of the nodes are unreachable then it is declared dead code. As of right now we only print a warning if a function is never called because optimizations are designed for next sprint. However the base code is there.

Set of examples:

Test case 16: tests the type extension as it has a predefined button which is defined in prepro.txt

Test case 17: webpage is not loaded error

Script:

go to Sprint3\qaCompiler\src\PythonBackend

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

To compile test case 16 run

python backend.py outputs16.html.xml outputs16_Code.py prepro.txt

To compile test case 17 run

python backend.py outputs17.html.xml outputs17_Code.py

The compiled python code is located at: Sprint3\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 the predefined type and error checking which is all that we wanted to cover. They work exactly as expected.

Verifiability:

You can verify it by running it on our testcases.