CS 335 Semester 2019–2020-II: Project Description

Due Your submission is due by Mar 13th 2020 11:59 PM IST.

General Policies

- Do not plagiarize or turn in solutions from other sources. We WILL check your submission(s) with plagiarism checkers. You will be PENALIZED if caught.
- Each group will get a total of FOUR FREE DAYS to help with your project submission deadlines. You can use them as and when you want throughout the duration of the project.

Description

The goal of this project is to implement a compilation toolchain, where the input is in Java language and the output is x86 code.

Milestone 2

In this milestone, you will extend Milestone 1 to implement support for the symbol table data structure, perform semantic analysis to do limited error checking, and then convert the input source program into an Intermediate Representation (IR) to be used by the later stages for your mini compiler (for e.g., target code generation).

Declaration statements

Extract relevant information from declaration statements and store in the symbol table. Useful information include types of variables and functions and the sizes and offsets of variables.

Non-declarative statements

- Use information from the symbol table to ensure that variables are used (i) within the scope of their declarations, and (ii) are used in a type-correct manner. The program is rejected if there is an error.
- Generate 3-address code (3AC) if the program is syntactically and semantically correct (i.e., there are no errors related to type and scope).

Suggestions

Make use of support for "semantic actions" in your lexer/parser to do semantic checking and to generate IR.

Symbol table. You can use a hierarchical two-level symbol table structure. For e.g.,

- A global symbol table that maps function names to their local symbol tables,
- A local symbol table for every function that contains the relevant information for the parameters and the local variables of the function.

Feel free to adapt the above structure according to your taste. Make sure your symbol table implementation is *extensible* since you might discover the need to store new information as the project progresses.

Generating IR.

- You will need to define 3AC instructions corresponding to various constructs allowed in the input program.
- Maintain the 3AC in an in-memory data structure (e.g., List or Vector).

Operator/Function disambiguation

Decide on the exact operator function that will be used in an expression. For example, for `x + y`:

(a) If `x` and `y` are both `int`s, resolve the `+` to say `+int`,

(b) If `x` and `y` are both `float`s, resolve the `+` to `+float`.

Type casting

For code `x + y`, if `x` is an `int` and `y` is a `float`, cast `x` to a `float` and resolve the `+` to `+float`. The 3AC will be as follows.

```
1    t1 = cast_to_float x
2    t2 = t1 +float y
```

Output. For correct input programs, your implementation should ouput the following:

1. A dump of the symbol table of each function as a CSV file (the columns of the CSV can be of your choice), and
2. A dump of the 3AC of the functions in text format.

There is no fixed format for the CSV header. For e.g., the columns can be the syntactic category or the token, the lexeme, the type, the line number in the source at the least.

For erroneous input programs, the output should provide meaningful messages that help identify the error that caused the program to be rejected. Do not just print `"error at line YYY"`. The following are a few examples of acceptable error messages: `"Type mismatch in line 72"`, `"Incompatible operator + with operand of type string"`, and `"Undeclared variable on line 38"`.

Here is an *incomplete* list of errors that your implementation should handle:

- All forms of type errors. For e.g., array index is not an integer and variables are declared as void type.
- All form of scoping errors.
- Non-context-free restrictions on the language. For example, an array indexed with more indices than its dimension and functions called with incorrect number of parameters.

Submission.

- Submission will be through Canvas.
- Create a zip file named "`cs335_project_ <roll>.zip`". The zipped file should contain a folder `milestone2` with the following contents:

- All your source files must be in `milestone2/src` directory. You are free to choose your implementation language.
- Create a directory `milestone2/tests` for your test cases. Name the test files as "`test_ <serial number>.java`".
- Use `Makefile` (or equivalent build tools like `ant`) for your implementation. You are free to use wrapper scripts to automate building and executing your compiler.
- Your submission must include a `milestone2/doc` directory and a `PDF` file. The `PDF` file should describe any tools that you used, and should include compilation and execution instructions. Document all command line options.

  You SHOULD USE LaTeX typesetting system for generating the PDF file.

Evaluation

- Your implementation SHOULD FOLLOW THE PRESCRIBED STEPS so that the expected output format (if any) is respected.
- We will evaluate your implementations on a Unix-like system, for example, a recent Debian-based distribution.
- We will evaluate the implementations with our OWN inputs and test cases, so remember to test thoroughly.
- The TAs will meet with each group for evaluation. Make sure that your implementation builds and runs correctly. A typical evaluation session could be as follows.

```
$ cd <milestone2>/src
$ make
$ ./milestone2 ../tests/test5.java
```