CSC 410 Project 3 Report

How To Use:

Prerequisites: In order for the program to run, make sure python3 is installed as the the process to run it requires the call to python3.

To run: *python3 run.py*

Inside the project directory, the run.py file contains the file reader which passes the data to minic to transform it to a C abstract syntax tree. The FunctionalTranslator object transforms the minic object to a func ast object which prints out the functional language equivalent. The output of run.py contains the original C source file, the output caml equivalent with simplification. By default, it outputs all the files from the inputs/final_inputs directory. You can also provide an argument "-f" followed by the path to a C input file. This will output the functional equivalent of that specific file.

The checkin directory also contains checkin python files similar to run.py that translates files in the input/checkin directories.

The Imperative-to-Functional Translation Process:

The general translation process is to take the minic C AST and convert it to a functional equivalent AST. This requires a tree walkthrough of the C AST by building a custom NodeVistor class that has methods that operate whenever a specific node is visited. For example, a basic AST object will have a parent node of a block, followed by children nodes of additional expressions. These expressions may be assignments, operations, functional calls, or whatever expressions you see in C. Our NodeVistor class also contains sets to keep track of variables that are read and written. This is important because our functional program contains parameters of the read variables and a return tuple containing our written variables.

Handling Assignments:

One of the most common expressions in C are assignment statements (I.E x = 1). The AST for an assignment statement for both languages contain a right and left side. Usually the left side contains a node for a variable or an array reference at an index. The right side can be an expression such as a constant value, binary operation, ternary expression, or a function call. The right side may contain variables that are read, so the tree NodeVisitor will search the right expression to capture any variables. Nodevisitor contains specific node methods for the expressions I mentioned earlier which will visit the ID method at the very end if there exist variables. If visited, it will add the variable name to the read set. The next step is to create an equivalent functional AST node. Since there are no assignments in functional programming, we use a let id = <expr1> in <expr2> binding instead. The id will be equivalent to the variable name of the assignment while the expression will be the equivalent functional expression of the C assignment value. Since the C AST nodes also have children nodes, we must translate all of those to equivalent functional AST nodes as well. The NodeVisitor also contains an helper expr function that recursively translates the nodes. The expr1 would be assigned to the output of the

expr function. The expr2 will be assigned to None as we do not know if there is another line in the C program after this binding. Essentially the translation focuses on building the functional equivalent by treating let bindings as linked lists and appending additional let bindings as the C program gets visited. After all the nodes have been visited in the C program, the tail expression of the let binding can be set with a return tuple of written variables. Later on we can replace the variables in the return tuple with expressions if the simplification rules hold true.

Handling If Statements:
The core components for if-statements contains the condition expression, the expression when the condition holds true, and the expression when the condition is false. First we visit the node for the condition expression to capture all the variables that are read. The expressions when it is true or when it is false undergoes the same process. We create a separate NodeVisitor object for each expression in order to visit the nodes inside. This is important because this allows us to create a specific let binding for that expression without interfering with the current one we are keeping track of. We combine the write sets of both of the AST's and use that as the id for the let binding. We set the corresponding function AST nodes as the expressions. Finally, we attach the previous binding with this if-statement let binding.

Handling Loops:
The loops that we expect from the C program are while loops and for loops. For both cases we use a recursive style syntax binding let rec <id> <args> = if <condition> then <expression> else <expression>. The ids contain the variables that are written throughout the loop and the condition is the same as the one that needs to hold for the loop. The body of the loop will be set as the first expression node. Like for if statements, we create a separate NodeVisitor object to create the let binding. To exit the loop, the expression when the if statement is false will return the written variables. Since for loops contain initialization and incrementation we will add the initialization let binding (i.e let i = 0 in …) outside the let rec binding. We also add the incrementation binding to the end of the if true expression binding. For while loops, we assume that there are no infinite loops so we do not worry about initialization and incrementation.

Handling Simplification:
After the whole process of translating the C program has been completed, we simplify the head let binding top down until we reach the tail. Since we know what variables are written and read already, we can take out specific bindings if they meet three rules.

1. Eliminating a let binding if the variable that is bound is never used except in the last tuple, and it does not use any variables that are bound below it.

2. A variable that is only bound once to a constant can be replaced by this constant in its scope.

3. A variable that is written and read in the same binding can be eliminated if it is not written somewhere else. (i.e let k = k + 1 in …)

We can accomplish (1) since we keep track of a write and read set. We can accomplish (2) by checking if the expression contains no variables then keeping a dictionary to replace the return tuple at the end. Finally we can accomplish (3), similarly to (1).

Syntax of Functional Language:

The syntax for our functional language is based off of Caml. We primarily use let bindings (i.e let <id> = <expression> in <expression>) and chain further let bindings to the second expression treating it as a linked list. For if statements, we use an if <condition> then <expression> else <expression>, and for recursion, we use let rec <id> <args>. The let binding is then wrapped with an outer function (i.e fun code_block(args) return (<return tuple>) = <expr>).