410 final report

Name of Group members:
Zhe Kai Wu, Keyan Wang, Yuan Meng

In this project, we build a tool to convert a C-like function body into functional body.

Suppose you have a block of c-like code in “xxx” file, run “python project3.py xxx”. The program would first generate a C-like function prototype with arguments variables in a file called “<fileName>Dummy” under the same directory that “xxx” file is in. The program will then read the c function in the dummy file to generate the appropriate functional language translation to the c code provided.

Translations steps:

1.
The program reads in a block of c code and wraps the code block in a simple c function.
So code like the following:

A = 1;
B = C;

Will be wrapped into a function like this:

```
Int * block_function() {
        A = 1;
        B = C;

        Return 0;
}
```

The produced function is then written into a dummy file to be used later for C to functional language translation. The wrapped C code is printed to the terminal.

2.
The program calls on the pycparser’s parse_file to produce abstract syntax tree (AST) of the C program.The pycparser AST is then converted to minic_ast typed AST by calling on transform.

3.
The program then reads in the produced minic AST and translates it to myfunctional_ast typed AST to produce the functional language translation. The function used for this conversion is called minicToFunctional. The output functional language translation is then printed to the terminal.

4.
After the functional language equivalent to the C code is made (in step 3), we then calls on the function ‘simplify(functionalAST)’ to build a simplified version of the functional language translation.

Minic to functional translation function (minicToFunctional):
To build the function prototype for our functional language’s AST, all the written variables and all the undeclared variables were found within the minic AST. We made a class called LHSPrinter in ‘project3.py’ to help determine all the written variables and undeclared variables by traversing through the minic AST using the ‘visit’ function provided by the ‘NodeVisitor’ class of ‘mini_ast.py’

The function prototype (myfunctional_ast.FuncDef object) is then built by making a visitor using LHSPrinter to find all written and undeclared variables.

‘minicToFunctional’ function takes in a minic typed AST and convert it into our functional language’s AST (myfunctional_ast object).
The function reads in the minic typed AST and convert each assignment statement under the provided AST into the equivalent Let/Let rec translation.

The statements under each compound objects of the provided minic AST is read to help build the let/let rec objects recursively, and are used to determine the return tuple for the most inner let binding. Each body of the let and let rec is built using the statements following the current statement for the let object.

For the syntax of functional language, we are using
let identifier(s) = assignedExpression in bodyExpression
as our let binding constructs and
let rec arguments = assignedExpression in bodyExpression
as our recursive let binding construct to handle iteration statements.

In our tool, each let and let rec will have the following __str__ format:
self.level * "   " + "Let/let rec " + ... " = " ... + "\n" +
self.level * "   " + "in" + ...
so each 'let' and its 'in' should have lined up with the proper amount of indentation like this:
Let ident =
    assignedExpression
in
    bodyExpression

OR

Let ident = assignedExpression
in bodyExpression

Depending on whether assignedExpression and bodyExpression's __str__ output is a single line or multiple lines.

Each assignment, and initialized declaration statements are converted to the appropriate let statements.
Each left-hand side variable is then added to the list of modified variables and will be used to help build the return tuple for the last statement in the code block.
Each if statement is converted to the appropriate ternary expression, with a condition (cond), statements to execute if condition is true (iftrue), and statements to execute if condition is false (iffalse). The function called minicToFunctional then determine all the variables modified within the if and the else expression and constructs a let statement with identifiers to be the set of variables returned by the if expression. The assigned expression to this let is then set to the ternary equivalent of the if statement.

```
If (a > 0) {
        b++;
} else {
        c = 2;
}
```

C code like the one shown above would be converted into the following before simplification:

```
 func block_function(a, c, b) return (c, b) =
        Let (c, b) =
                if (a > 0)
                        then
                Let b = (b + 1)
```

```
                    in (c, b)
            else
                    (2, b)
    in (c, b)
```

If the if statement does not have an else statement, an else statement is made to return the name of the set of variables modified in the iftrue statement block.

The ternary expression works just like the if statement from above since they are both converted to myfunctional_ast.TernaryOp classed object.

The binary operators from are converted to myfunctional_ast.BinaryOp classed object, which is computed by computing for the functional programming equivalent objects for both the minic object expression on the left and the right of the operator.

Unary operators are converted to myfunctional_ast.UnaryOP classed object to store both the operator and the expression. Usually a unary statement is surrounded by brackets so out like !(a > 0) would end up becoming !a > 0.

When no more statement is to be found in the compound block provided by the minic AST, a return tuple is built using the list of modified variables given by the previous statements. An object called myfunctional_ast.ReturnTuple is built and used as the body expression for the last let/let rec statement built.

Each function call expression under the minic AST is converted to the functional programming equivalent so the entire AST can be printed by calling on the __str__ method directly.

Each array reference is converted to myfunctional_ast.ArrayRef object to help determine the list variables that are modified in the C code provided.
If we have:

```
if (a > 0) {

    b[1]++;
}
```

Then the equivalent functional programming is:

```
func block_function(a, b) return (b) =
    Let b =
        if (a > 0)
        then
            Let b[1] = (b[1] + 1)
            in b
        else
            b
    in b
```

Each While loop is converted to a let rec that is assigned to a ternary who performs the steps in the while if the condition is true and then recursively calls on the same let rec. If the condition is false, the variables that are modified in the loop are simply returned.
The body expression to a let rec is either a ReturnTuple, a let statement, or another let rec (a while, a for loop, or a do while that is not nested in the current while loop).

```
while(cond) {
    a++;
    b = a * a;
}
```

For input C code like the one above would produce output below

```
func block_function(a, b, cond) return (a, b) =
        let rec loop a b =
                if cond
                then
                        Let (a, b) =
                                Let a = (a + 1)
                                in
                                        Let b = (a * a)
                                        in (a, b)
                        in loop a b
                else
                        (a, b)
        in (a, b)
```

Each for loop object in the minic AST is converted let rec form by first build let for the initialized statement in the for loop. Then the next statement in the for loop is added into the compound statement provided by the minic AST as the last statement to be executed. Then the condition of the loop is used to help build the let rec in the same way while's functional programming equivalent is built. The let for the initialized statement then takes this let rec as body expression to produce the functional programming equivalent for C's for loop.