PROJECT 3

Anthony Samaha 1000652661 samahaa2

Introduction:

The main focus of this project was to convert c code to functional programming code. In particular, the functional programming code outputted was based on ocaml. To achieve this, the project was split into multiple check-ins to slowly build the functional language. Over the semester, I worked with Victor Nicolet to achieve the desired results through multiple checkins.

Overall, the project really helped me to understand more about abstract syntax tree and how they are built from the ground up. The general idea is to keep the nodes as simple as possible so they can easily be extendable and in a sense building blocks to place together and create your language.

How to use the tool:

If you would just like to convert a c file to function programming code, take a look at **tool.py** in the main folder of the project. Simply change the f variable on line 28 to the file you would like to convert.

Important Files:

- functions.py:
    - Contains helper functions used throughout the project.
        - functions to get all input and output variables to a function.
        - Functions to clean up strings
- convertctf.py
    - File used to convert the minic ast to our functional programing ast.
- func_ast.py
    - File used to represent our functional programming ast.
- Checkin files:
    - check-in1.py, check-in2-t1.py, check-in2-t2.py, checkin3.py, checkin4.py, checkin5.py, checkin6.py
        - All used to display the progress of the project.

Simple variables, constants, expressions:

Variables, constants and expressions will look the exact same in c as they do in the functional programming language.

Let Constructs:

For example:

```
b = a + 3;
```

Would be converted to

```
let b = a + 3 in
```

Essentially all variable declarations and assignments will look the same with a let prepended to the beginning and a in appended to the end.

Run: python checkin3.py - to see the construction of let statements

If Statements:

Simple if statements (no else block):

For example:

```
if(cond > 0){a = a + 1}
```

will be converted to

```
let a = if cond > 0 then a + 1 else a
```

In particular, a else block must be created in a functional programming language even if it does not exist in the original (ie c). Otherwise, the if block is turned into a ternary statement.

Regular if statements(if and else block):

For example:

```
if(a == 0) {
   b = c;
} else {
   c = b + 1;
}
```

Would be converted to

```
let (b, c) = if a == 0 then (c, c) else (c, b + 1) in (b, c)
```

In this case, the else block if filled just like the if block.

Run: python checkin4.py - to see the construction of if statements

Let Rec:

For example:

```
for(i = 0; i < n; i++){
   sum = sum + a[i];
}
```

Would be converted to

```
let i = 0 in
let (i, sum) =
   let rec loop0 i sum =
      if i < n
      then
        let sum = sum + a.(i) in
          let i = i + 1 in
            loop0 i sum
      else (i, sum)
   in loop0 i sum
in
```

In particular, we convert any type of loops to a recursive function by first initializing any required variables, and then executing the body of the loop recursively and only stoping when the loop condition is broken.

Run: python checkin6.py - to see the construction of let statements

Optimizations:

The following optimizations were done the our functional programming code:

1. A first criteria for eliminating a let binding can be that 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 (in the let-bindings).
2. A second criteria for eliminating a let binding can be that 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 (in the let-bindings).
3. If a variable x is declared and used, and later re-declared but never used again, we will remove the second declaration only and add it straight to the output tuple.

Run: python checkin5.py - to see the construction of if statements

Conclusion:

Through running this tool, all required check ins were completed and c code can be converted to a functional programming representation.

Moving forward, topics that I would want to extend would be to look for further optimizations that could be done. In particular, the main focus of the optimizations of this project was to look for unused variables or variables that act like constants. I personally think it would be really interesting to try and implement more complex optimizations that some compilers would do such as loop optimizations.