CS3071 – Exercise 4 Report

Edward Bergman
#15335633

Task:

Our task was to introduce the ternary operator to the language as specified by the Tastier.ATG.

```
<identifier>:=<condition>?<expression1>:<expression2>
```

Method:

The first step was to more clearly understand the Tastier.ATG file, rewriting this to reduce visual noise (see attached ATG-less-noise.txt). Once this was done we could introduce our RegularExpr. in terms of the Production Rules already stated.

```
Stat = Ident ( “:=” Expr [‘?’ Expr ‘:’ Expr ]‘;’ | “();” )
      | ....
```

Rather than introduce a narrow scope to where the ternary operator could be used it would make sense to modify the Expr production rule as the result of a ternary operator simplifies to an expression.

```
Expr = SimExpr [RelOp SimExpr [‘?’ Expr ‘:’ Expr]]
```

Finally, to reduce the complexity of Expr, we move the ternary operator to its own production rule

```
TernaryOp = ‘?’ Expr ‘:’ Expr
Expr = SimExpr [RelOp SimExpr [TernaryOp]]
```

As we are limited to the scope of the ATG file and the nature of the ternary operater is more advanced than a simple RelOp, AddOp or MulOp, it was decided that Ternary Op should include the Expr in its own production rule, rather than try implement it as an actual Op object.

The last, and most difficult part was getting the correct assembly to be produced. Using the ‘if/else’ to understand labels and branching code gen along with a long period of studying the relationships between the production rules and their generated code, we could implement the correct code generation to perform the ternary operation.

Remarks:

The RegularExpression part was straightforward enough, the issues arose once it came to code generation. Firstly, there was an issue that made debugging harder in that there was no exception thrown for assinging a variable to a type other than its own type.

```
int i;
i := true;
i := 3==4;
```

This would compile fine and made tracking down where the Assembly code was going wrong difficult as the behaviour was undefined. This issue was tracked down to an ‘if’ statement contained in the original ATG.

```
//Stat Production Rule
      ...
      ...
                  (.    if(type == obj.type)          .)
                  (.          ...do some things...    .)
                  (.    <missing else statment >      .)
```

Adding ‘else SemErr(‘type mismatch’);’ fixed this issue and meant the code would not compile when appropraite. (Yay no more debugging ARM that didn’t make sense)

Lastly, the first Expr in ‘? Expr : Expr’ was loading into the wrong register while the second Expr was loading into the correct register. This was tracked down to the fact that the SimpExpr in the Expr Production rule was incrementing the register use count while the first Expr would then clear the register use count. To solve this, we cleared the register count as we begin the TernaryOp. This is allowable as the result of the condition (SimpExpr RelOp SimpExpr) is calculated before we need to evalute either Expr for the TernaryOp.

Implementing things in this way allowed us to perform nested expressions in our TernaryOp i.e.

```
      int i;
      i := 3 < 4? 10 : (2+3) < 6? 20 : false? 30 : 40;
      //functional programmers rejoice
```

Learnt:

The main points learnt during this exercise was how one could implement language features using production rules, generalize production rules so they’re usable in the widest applicable context and compilers are hard...