Compiler Design
Lab 5 Report

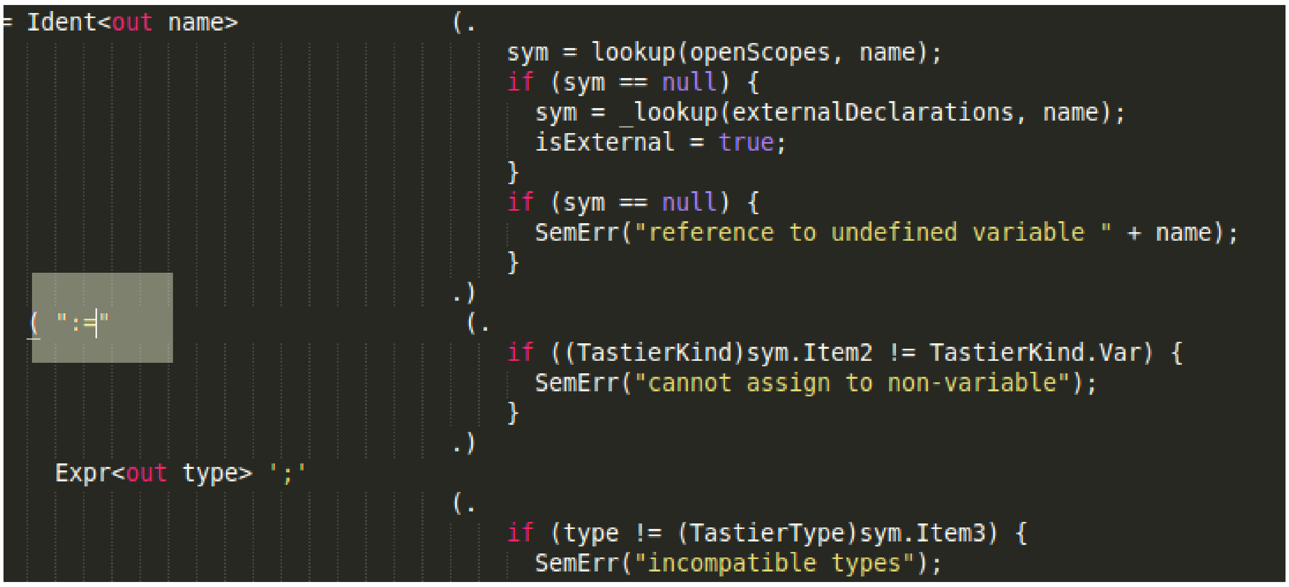
Donal O'Sullivan
12308492

For this assignment the task was modify the Tastier compiler in Coco/R to alter and add to the functionality of the compiler.

1) The first of these tasks was to replace the assignment operator. Changing it from '=' to ":=".

This was relatively simple. All that had to be done was to find the operator in the Tastier attributed translation grammar and replace it with our new operator.

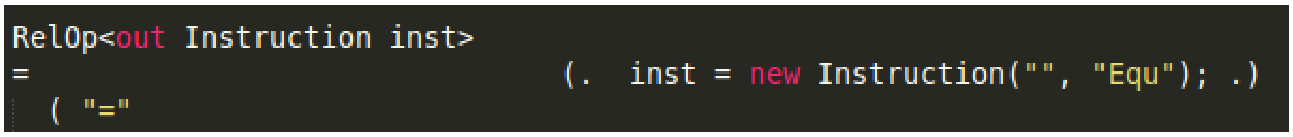
The old assignment operator was only used at one point in the original grammar.



```
= Ident<out name>                (.
                                    sym = lookup(openScopes, name);
                                    if (sym == null) {
                                      sym = _lookup(externalDeclarations, name);
                                      isExternal = true;
                                    }
                                    if (sym == null) {
                                      SemErr("reference to undefined variable " + name);
                                    }
                                 .)
  ( ":="                          (.
                                    if ((TastierKind)sym.Item2 != TastierKind.Var) {
                                      SemErr("cannot assign to non-variable");
                                    }
                                 .)
    Expr<out type> ';'
                                 (.
                                    if (type != (TastierType)sym.Item3) {
                                      SemErr("incompatible types");
```

Also for this part of the assignment the relational operator "==" had to be replaced with '='.

This was a similar process. The equality operator was found grouped with the other relational operators in the grammar and a simple substitution of the old operator for the new one achieved the desired results.

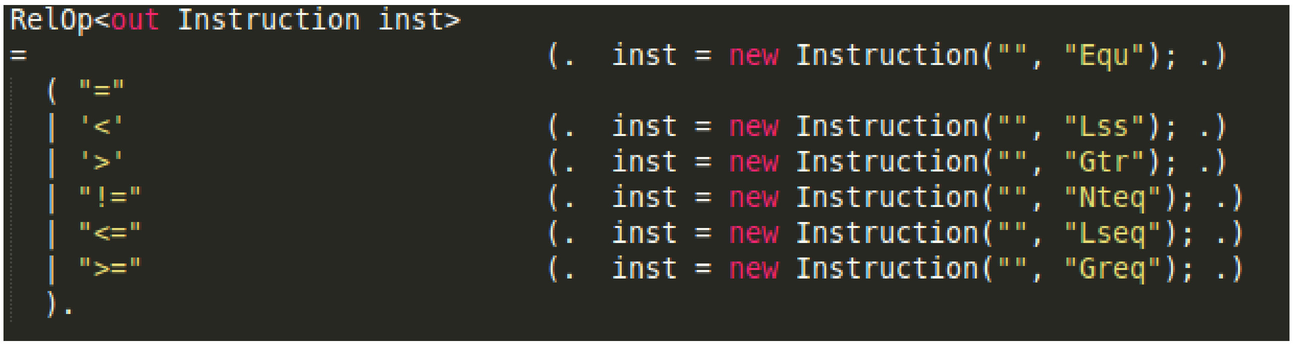


```
RelOp<out Instruction inst>
=                                  (.  inst = new Instruction("", "Equ"); .)
  ( "="
```

It was also asked that we extend the language with 3 new operators.

- "!=" → Not Equal
- "<=" → Less than or equal to
- ">=" → Greater than or equal to

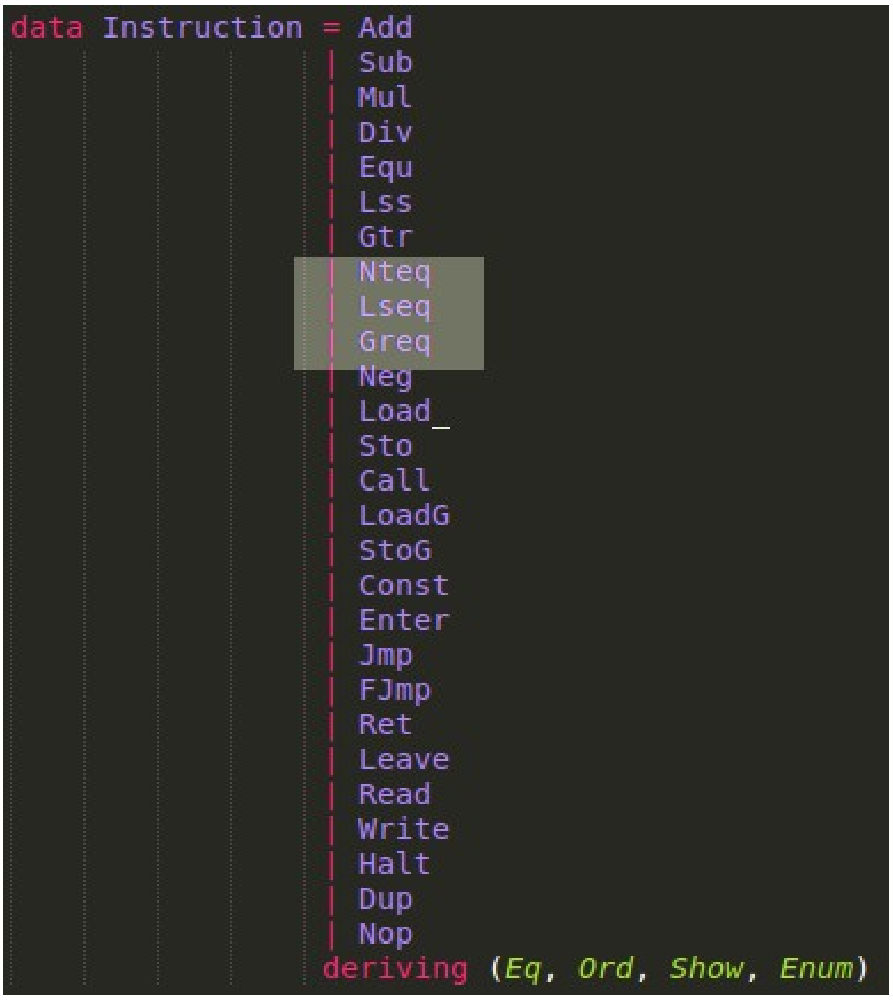
The implementations for these operators were carried out by, first of all, adding the new operators to the attributed grammar and generating instruction names for them.



```
RelOp<out Instruction inst>
=                                  (.  inst = new Instruction("", "Equ"); .)
  ( "="
  | '<'                            (.  inst = new Instruction("", "Lss"); .)
  | '>'                            (.  inst = new Instruction("", "Gtr"); .)
  | "!="                           (.  inst = new Instruction("", "Nteq"); .)
  | "<="                           (.  inst = new Instruction("", "Lseq"); .)
  | ">="                           (.  inst = new Instruction("", "Greq"); .)
  ).
```

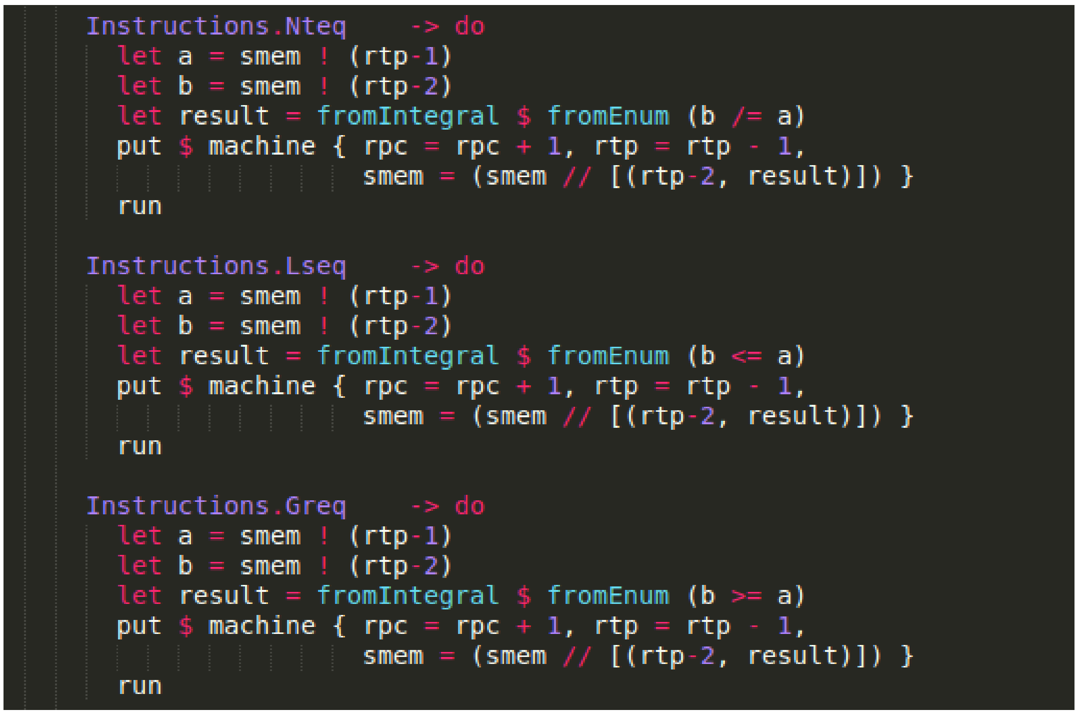
Once the instructions were added to the attributed grammar they had to be added to the instruction set for the machine. This was done by editing the instruction set for the language. The instructions are defined in the Instructions.hs file while there behaviour is defined in the Machine.hs file. The keywords for the instruction set also needed to be added to the parser so that the instructions could be recognised by the compiler.

In Instructions.hs the instruction type and all of the forms it can take is defined. This means we have to add our new instructions to the set of forms an instruction can have.



```
data Instruction = Add
                 | Sub
                 | Mul
                 | Div
                 | Equ
                 | Lss
                 | Gtr
                 | Nteq
                 | Lseq
                 | Greq
                 | Neg
                 | Load
                 | Sto
                 | Call
                 | LoadG
                 | StoG
                 | Const
                 | Enter
                 | Jmp
                 | FJmp
                 | Ret
                 | Leave
                 | Read
                 | Write
                 | Halt
                 | Dup
                 | Nop
                 deriving (Eq, Ord, Show, Enum)
```

After this was done inside Machine.hs I created functions that defined the behaviour of each new instruction.

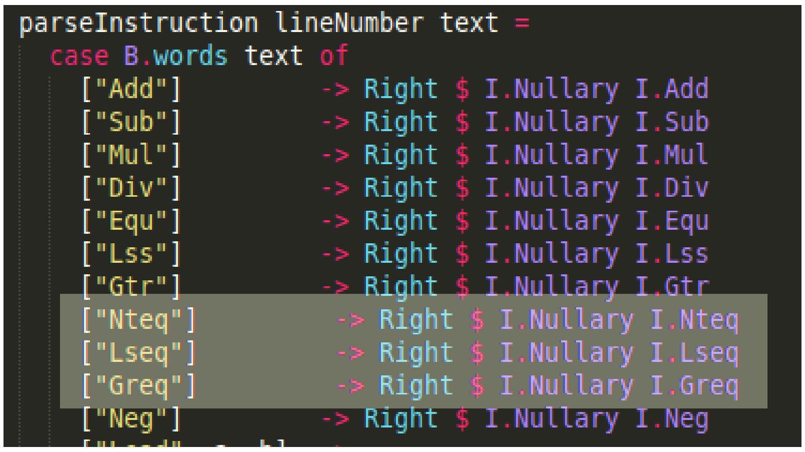


```
    Instructions.Nteq     -> do
      let a = smem ! (rtp-1)
      let b = smem ! (rtp-2)
      let result = fromIntegral $ fromEnum (b /= a)
      put $ machine { rpc = rpc + 1, rtp = rtp - 1,
                      smem = (smem // [(rtp-2, result)]) }
      run

    Instructions.Lseq     -> do
      let a = smem ! (rtp-1)
      let b = smem ! (rtp-2)
      let result = fromIntegral $ fromEnum (b <= a)
      put $ machine { rpc = rpc + 1, rtp = rtp - 1,
                      smem = (smem // [(rtp-2, result)]) }
      run

    Instructions.Greq     -> do
      let a = smem ! (rtp-1)
      let b = smem ! (rtp-2)
      let result = fromIntegral $ fromEnum (b >= a)
      put $ machine { rpc = rpc + 1, rtp = rtp - 1,
                      smem = (smem // [(rtp-2, result)]) }
      run
```

Here we have defined the functions that are called when the operators we specified are used. Once these were defined I added the instructions to the parser of the Tastier Machine. This is so that the instructions would be recognized by the compiler when encountered in code.

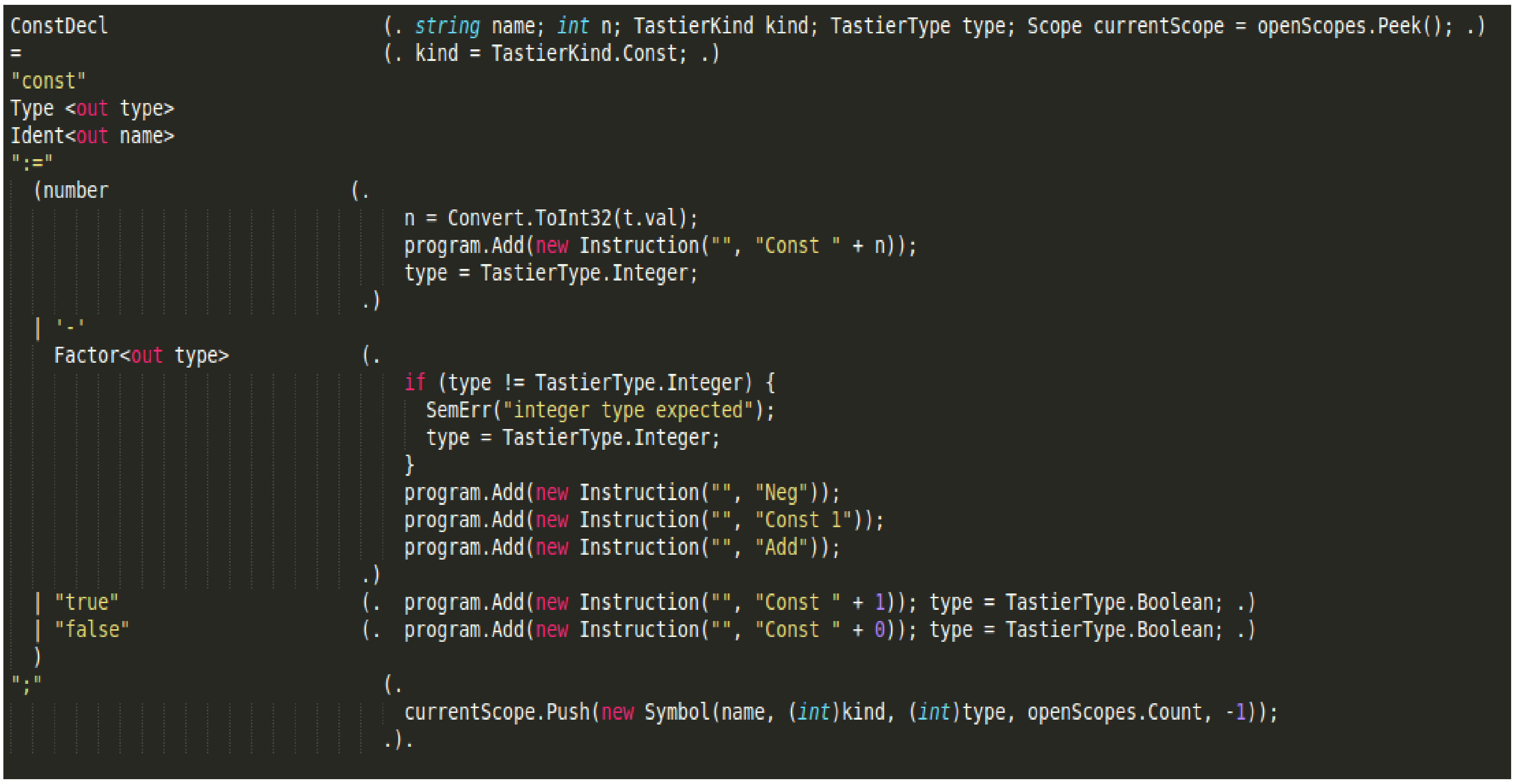


```
parseInstruction lineNumber text =
  case B.words text of
    ["Add"]         -> Right $ I.Nullary I.Add
    ["Sub"]         -> Right $ I.Nullary I.Sub
    ["Mul"]         -> Right $ I.Nullary I.Mul
    ["Div"]         -> Right $ I.Nullary I.Div
    ["Equ"]         -> Right $ I.Nullary I.Equ
    ["Lss"]         -> Right $ I.Nullary I.Lss
    ["Gtr"]         -> Right $ I.Nullary I.Gtr
    ["Nteq"]         -> Right $ I.Nullary I.Nteq
    ["Lseq"]         -> Right $ I.Nullary I.Lseq
    ["Greq"]         -> Right $ I.Nullary I.Greq
    ["Neg"]         -> Right $ I.Nullary I.Neg
```

The code discussed above is how I implemented the first part of the exercise.

2) The second part of the lab was to implement the constant definitions as part of the language. The brief specified that constants would have to be defined before variable declarations at the top of a program. Constants also should not be able to be reassigned values.

Firstly I defined a grammar for constant declarations that would enable the parser to adequately handle their occurrence.



```
ConstDecl                        (. string name; int n; TastierKind kind; TastierType type; Scope currentScope = openScopes.Peek(); .)
=                                (. kind = TastierKind.Const; .)
"const"
Type <out type>
Ident<out name>
":="
  (number                     (.
                                   n = Convert.ToInt32(t.val);
                                   program.Add(new Instruction("", "Const " + n));
                                   type = TastierType.Integer;
                                .)
  | '-'
    Factor<out type>            (.
                                   if (type != TastierType.Integer) {
                                     SemErr("integer type expected");
                                     type = TastierType.Integer;
                                   }
                                   program.Add(new Instruction("", "Neg"));
                                   program.Add(new Instruction("", "Const 1"));
                                   program.Add(new Instruction("", "Add"));
                                .)
  | "true"                      (.  program.Add(new Instruction("", "Const " + 1)); type = TastierType.Boolean; .)
  | "false"                     (.  program.Add(new Instruction("", "Const " + 0)); type = TastierType.Boolean; .)
  )
";"                              (.
                                   currentScope.Push(new Symbol(name, (int)kind, (int)type, openScopes.Count, -1));
                                 .).
```

The constant grammar I created generates code to handle the name of the constant, the kind of item it is (variable, procedure or constant), the type of the item(integer, boolean, undefined) and the scope of the item.

Firstly the compiler looks for the string “const” this indicates that is in fact a constant that is being declared.

After this the type of the constant must be declared boolean or integer.

Identifier for the name of the constant is then expected afterwhch the compiler will expect an assignment operator.

After this the compiler expects either a number or a boolean input depending on the type. In the case of a number being entered the value is attached to the int variable in the function handling the parsing. The type is also assigned to be a integer.

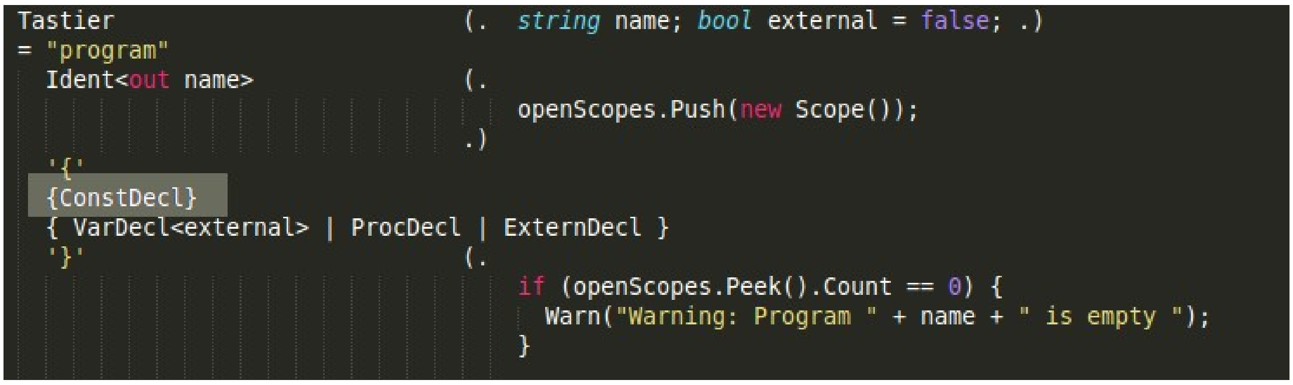
Next negative input is handled. Any constant that is not declared as being an integer and has a negative sign passed I is rejected. Otherwise the input is negated and assigned to our constant.

Finally booleans are also handled by the grammar. Boolean inputs are assigned to the constant as is the boolean type.

The constant is then added to the program.

I also added constant to the TastierKind enumeration so it would be recognised by the program and viewed as seperate from variables in order to prevent reassignment.

I also altered the grammar for Tastier programs in so that constant declarations would be expected before variable declarations as was specified in the brief.



```
Tastier                         (.  string name; bool external = false; .)
= "program"
  Ident<out name>               (.
                                    openScopes.Push(new Scope());
                                .)
  '{'
  {ConstDecl}
  { VarDecl<external> | ProcDecl | ExternDecl }
  '}'                           (.
                                    if (openScopes.Peek().Count == 0) {
                                      Warn("Warning: Program " + name + " is empty ");
                                    }
```

3) Add some form of string data type to the language so it is possible to output string values.

- Unfortunately I was unable to complete this part of the assignement in time for submission.