

# TRANSLATION FOR THE HIPSTER LANGUAGE - REPORT

Nigel Salamanes (43690599)

## Contents

Introduction .....	2
Design.....	3
Implementation .....	4
Testing.....	6

# Introduction

A translator, in programming languages, is a program in which translates a program written in a supplied programming language into another computer language, the target language. The translator must ensure that in the process the functionality or logical structure remains the same.

Hipster is a Domain Specific Language (DSL). As such, a user of the CellSim simulator can easily create and understand a Conway's game of Life environment using this language. Hipster is translated into the Java language and is then compiled. This removes any unnecessary and cumbersome learning that comes with programming in Java.

The provided framework contains all previous phases required in a translator. It also contains an incomplete translation module. By completing the translation module, it will provide us with insight into compilers and the procedures they use to check the code satisfies the rules of the programming language.

# Design

It was important to be familiar with the assignment documentation and the bundled framework. This provided guidance as to what was required to complete the translation module. Using the output of `run src/test/resources/life.hip` allowed me to compare the progress I was making with the `src/test/resources.life.out`. My solution must be able to produce the same results with the matching `.hip` file.

Both `ForStmt` and `IterateOverStmt` must be able to return a statement. `MiniJavaTree.scala` contains a subset of Java functions. However, there was no one to one mapping to directly translate these functions. For my solution I have used a `J.Block` with a vector of statements in both classes.

```
/**
 * A general <for> loop, iterating over integer ranges. Semantics:
 * - The <start>, <stop>, and <step> expressions are evaluated on
 *   entry and are not then re-evaluated.
 * - If <step> is 0 then we terminate without executing the body.
 * - If <start> = <stop> and <step> ≠ 0 then we execute the body once.
 */
case class ForStmt(
  idn : IdnDef,
  start : Expression,
  end : Expression,
  step : Option[Expression],
  body : Statement) extends Statement
```

```
/**
 * <iterate> statement for iterating over sets of neighbours.
 */
case class IterateOverStmt(
  idn : IdnDef,
  nbrs : NeighbourSet,
  body : Statement) extends Statement
```

The class signatures allowed me to clearly identify each input parameter and its use, as well as its extension. A connection can be made between the semi-formal notation in the documentation, class signature above, and the `life.hip` and `life.out`.

Once I was able to use the parameters as needed, completing the class statements was as simple as identifying statement calls and adding it to the vector of statements.

Thankfully `MiniJavaTree.scala` had `If` and `While` Java statements and these were used appropriately. These functions provide for an easy translation from Hipster to Java.

The implemented `ForStmt` used a Java While loop. The conditional statement proved to be quite tricky as there were multiple nested expressions within each other. The ordering of these expressions is important as the placement of brackets would change the order of operations.

For `IterateOverStmt` I have used a somewhat imperative approach. This allows us to neatly view and encode Scala's `map`, `flatMap`, `foreach`

# Implementation

My implementation of `IterateOverStmt`.

```
case s @ IterateOverStmt(i, ns, b) =>
  // Enclose in a block.
  J.Block(
    // declare a new identifier of type cell, also get unqualified name of name type. E.g: current, nbr, etc.
    J.Var(J.ClassType(J.IdnUse("Cell")), J.IdnDef(getTargetName(b, i.idn)), None)
    // now add the statements to a vector and append it to J.Block
    +=
    {
      for(
        // scan through the elements of the list of translated neighbours, binding n to each one
        n <- translateNeighbourSet(ns);
        // <code to generate Java tree for the statement <<id>> = n> ;
        a = J.Assign(J.IdnExp(J.IdnUse(J.IdnDef(getTargetName(b, i.idn)).idn)), n);
        // <code to generate Java tree for the translation of the <body> statement>
        s = translateStmt(b);
        r <- Vector(a,s)
      ) yield r
    }
  )
```

This uses a somewhat imperative approach with the for loop. Originally I had values outside the `J.Block` and used them within. After my solution worked, I tried to make my code as simple and clear as possible in the least amount of lines possible.

As seen above, everything is inside a `J.Block` and this is what is returned. Also, `J.Block` can't accept a `J.Var`, however this is possible because it is being appended with the `For` loop which yields a Vector. This way there are no declared values outside of the block.

My implementation of `ForStmt`.

```
J.Block(  
  // put statements in a vector  
  Vector(  
    // declare & set variables for "for" statement  
    J.Var(J.IntType(), IdnDef_idn, Some(translateExpr(s))),  
    J.Var(J.IntType(), IdnDef_end, Some(translateExpr(e))),  
    J.Var(J.IntType(), IdnDef_step, st.map(translateExpr).orElse(Some(J.IntExp(1)))),  
    J.Var(J.BooleanType(), IdnDef_dirn, Some(J.GreaterEqExp(J.IdnExp(J.IdnUse(IdnDef_step.idn)), J.IntExp(0)))),  
  
    // If "_step" is going in a direction (any number other than 0)  
    J.If(J.NotEqualsExp(J.IdnExp(J.IdnUse(IdnDef_step.idn)), J.IntExp(0)),  
      // If-then, go into while loop  
      J.While(  
        J.OrExp(  
          J.AndExp(  
            J.IdnExp(J.IdnUse(IdnDef_dirn.idn)),  
            J.GreaterEqExp(J.IdnExp(J.IdnUse(IdnDef_end.idn)), J.IdnExp(J.IdnUse(IdnDef_idn.idn))),  
          ),  
          J.AndExp(  
            J.NotExp(J.IdnExp(J.IdnUse(IdnDef_dirn.idn))),  
            J.LessEqExp(J.IdnExp(J.IdnUse(IdnDef_end.idn)), J.IdnExp(J.IdnUse(IdnDef_idn.idn))),  
          )  
        ),  
        // block for statements executed by the loop  
        J.Block(  
          // translate statements contained in loop  
          Vector(translateStmt(b),  
            // increment the loop counter  
            J.Assign(J.IdnExp(J.IdnUse(IdnDef_idn.idn)),  
              J.PlusExp(J.IdnExp(J.IdnUse(IdnDef_idn.idn)),  
                J.IdnExp(J.IdnUse(IdnDef_step.idn)))  
          )  
        ),  
      ),  
      // If-else  
      J.Empty()  
    )  
  )  
)
```

The majority of my code is contained in the `J.Block`. There are a few value declarations above to provide easy access to an identifier definition. I have also created these values for code visibility.

As seen above, the first four variables are the variables which are used in a `for` statement.

The next part is the Java `if` statement. This checks that the increment is going either in a positive or negative direction. This is important otherwise we would be in an infinite loop.

Nested in the If-Then statement is the `while` loop. The first is an expression is a condition to ensure that we have not crossed the bounds of the step. Its second parameter is a Vector to execute the statements. The Vector itself will hold the statement of the body as well as the loop incrementor.

# Testing

## Testing an empty for loop.

Contained in *forTest\_1.hip* are two for loops, one with empty space and one without. This is to test for loops without any statements to execute. The difference here is the amount of white space.

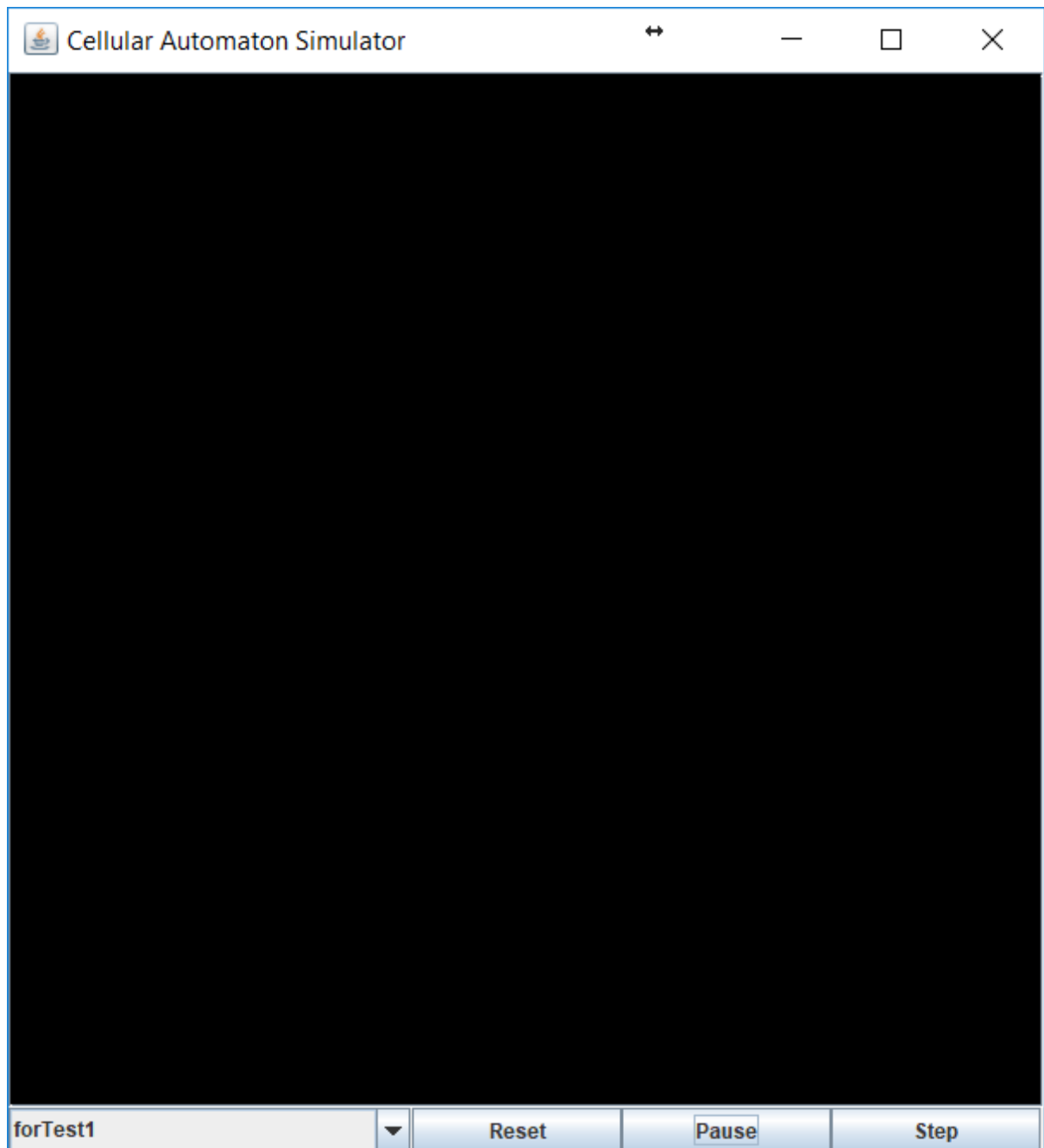
```
24 initialiser forTest1 {
25
26     for i = 0 to (width-1){
27     }
28
29     for i = 0 to (height-1){
30
31     }
32
33 }
```

The translated code is shown below. As we can see both loops contain an empty brace which is the body of the loop.

```
82 void forTest1$16() {
83 {
84     int i$19 = 0;
85     int _end$21 = width$5 - 1;
86     int _step$22 = 1;
87     boolean _dirn$20 = _step$22 >= 0;
88     if ( _step$22 != 0)
89         while ( _dirn$20 && _end$21 >= i$19 || (!_dirn$20 && _end$21 <= i$19)) {
90             { }
91             i$19 = i$19 + _step$22;
92         }
93     }
94     {
95         int i$23 = 0;
96         int _end$25 = height$6 - 1;
97         int _step$26 = 1;
98         boolean _dirn$24 = _step$26 >= 0;
99         if ( _step$26 != 0)
100             while ( _dirn$24 && _end$25 >= i$23 || (!_dirn$24 && _end$25 <= i$23)) {
101                 { }
102                 i$23 = i$23 + _step$26;
103             }
104     }
105 }
```

The code can also be generated, compiled and executed by CellSim. It's not very exciting as this test only contains very basic functions.

Information:12/11/2017 4:04 PM - Compilation completed successfully in 2s 187ms

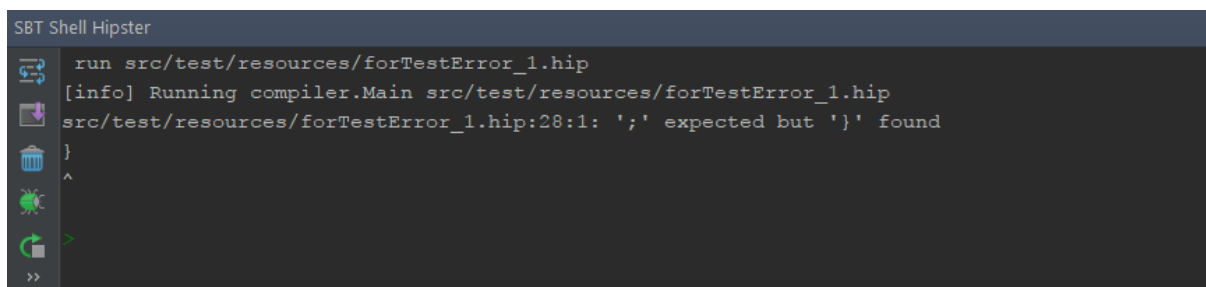




Contained in `forTestError_1.hip` is a for loop with no braces. Hipster language does allow for `for-loops` to have no braces. This combination has no braces or code to execute.

```
24 initialiser forTest1Error {  
25  
26     for i = 0 to (width-1)  
27  
28 }
```

This code does not compile successfully. As displayed, it is looking for a statement terminator but never finds one. For loops that do not use braces, they will use the next line of code as its body and start looping. This may cause undesired and unexpected results that will be revealed in later tests.



```
SBT Shell Hipster  
run src/test/resources/forTestError_1.hip  
[info] Running compiler.Main src/test/resources/forTestError_1.hip  
src/test/resources/forTestError_1.hip:28:1: ';' expected but '}' found  
}  
^  
^  
>>
```

## Testing for loops with one statement

Contained in `forTest_2.hip` are two for loops that execute one statement. The difference being the body of the last for loop is enclosed in braces. Here we are testing a combination of loops with braces and no braces.

```
24  initialiser forTest2 {
25      int x = 0;
26
27      for i = 0 to (width-1)
28          x = x + 1;
29
30      for i = 0 to (height-1){
31          x = x + 1;
32      }
33
34  }
```

As we can see the body of the loops will carry forward their braces in the translated code. For this test it is fine and does not affect the execution. They are semantically identical. However, this may cause unintended execution of code if not mindful of Hipsters for loop syntax.

```
82  void forTest2$16() {
83      int x$19 = 0;
84      {
85          int i$20 = 0;
86          int _end$22 = width$5 - 1;
87          int _step$23 = 1;
88          boolean _dirn$21 = _step$23 >= 0;
89          if (_step$23 != 0)
90              while (_dirn$21 && _end$22 >= i$20 || (!_dirn$21 && _end$22 <= i$20)) {
91                  x$19 = x$19 + 1;
92                  i$20 = i$20 + _step$23;
93              }
94      }
95      {
96          int i$24 = 0;
97          int _end$26 = height$6 - 1;
98          int _step$27 = 1;
99          boolean _dirn$25 = _step$27 >= 0;
100         if (_step$27 != 0)
101             while (_dirn$25 && _end$26 >= i$24 || (!_dirn$25 && _end$26 <= i$24)) {
102                 {
103                     x$19 = x$19 + 1;
104                 }
105                 i$24 = i$24 + _step$27;
106             }
107     }
108 }
```

## Testing for loops with different combinations

Contained in `forTest_3.hip` are two for loops without any braces. Syntactically this code will translate. However, it may not translate to what the programmer intended. This is to test loops with no braces with a statement to execute.

```
24  initialiser forTest3 {
25      int x = 0;
26      int y = 0;
27
28      for i = 0 to (width-1)
29
30      for j = 0 to (height-1)
31          x = x + 1;
32          y = y + 1;
33
34  }
```

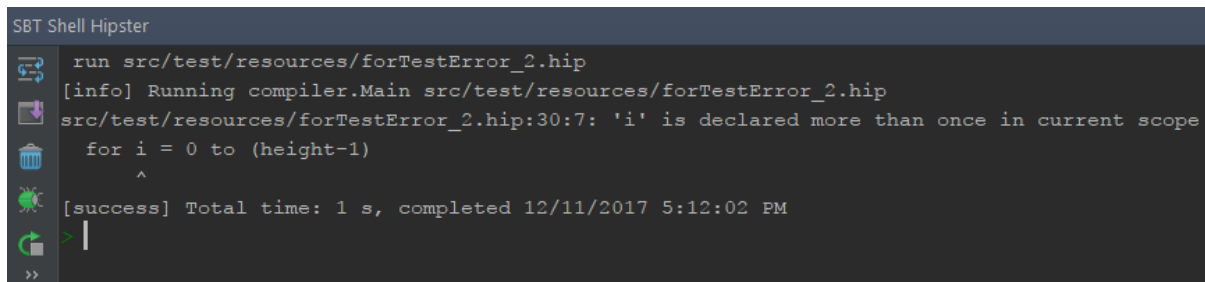
The result shows that the second for loop is used as the body of the first for loop. Furthermore, the body of the second for loop only contains the first statement after it and is translated to `x$19 = x$19 + 1;`. The second statement is not included in the loop and is executed after as shown by `y$20 = y$20 + 1;`.

```
82  void forTest3$16() {
83      int x$19 = 0;
84      int y$20 = 0;
85      {
86          int i$21 = 0;
87          int _end$23 = width$5 - 1;
88          int _step$24 = 1;
89          boolean _dirn$22 = _step$24 >= 0;
90          if (_step$24 != 0)
91              while (_dirn$22 && _end$23 >= i$21 || (!_dirn$22 && _end$23 <= i$21)) {
92                  {
93                      int j$25 = 0;
94                      int _end$27 = height$6 - 1;
95                      int _step$28 = 1;
96                      boolean _dirn$26 = _step$28 >= 0;
97                      if (_step$28 != 0)
98                          while (_dirn$26 && _end$27 >= j$25 || (!_dirn$26 && _end$27 <= j$25)) {
99                              x$19 = x$19 + 1;
100                             j$25 = j$25 + _step$28;
101                         }
102                     }
103                     i$21 = i$21 + _step$24;
104                 }
105             }
106             y$20 = y$20 + 1;
107     }
```

`forTestError_2.hip` contains the same structure of loops as above however they now use the same identifier in the loops as declared by `i`. This is to test nested loops and their scoping.

```
24  initialiser forTestError2 {
25      int x = 0;
26      int y = 0;
27
28      for i = 0 to (width-1)
29
30      for i = 0 to (height-1)
31          x = x + 1;
32          y = y + 1;
33
34  }
```

This code will not translate successfully. As previously tested, the non-existent use of braces can cause unexpected issues. The error message states that `i` has been declared more than once. Visually it seems like it is not. However, as seen in `forTest_3.hip`, the scope of the first loop falls into the second loop. It is for this reason that the translator is stating there are more than one declarations of `i`.



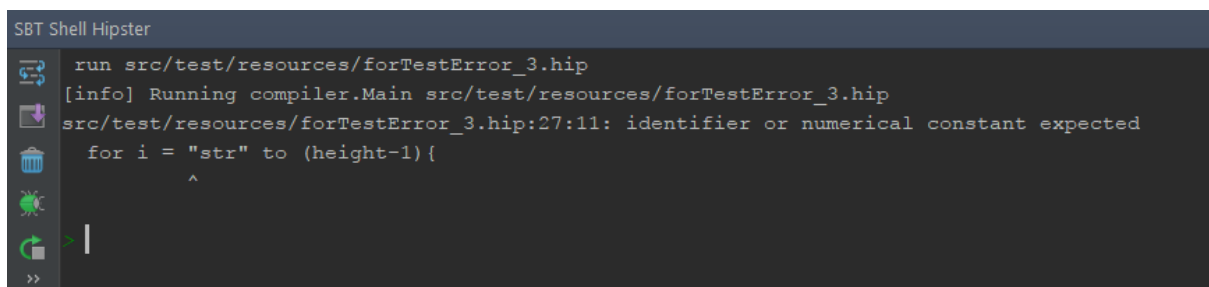
```
SBT Shell Hipster
run src/test/resources/forTestError_2.hip
[info] Running compiler.Main src/test/resources/forTestError_2.hip
src/test/resources/forTestError_2.hip:30:7: 'i' is declared more than once in current scope
  for i = 0 to (height-1)
    ^
[success] Total time: 1 s, completed 12/11/2017 5:12:02 PM
>
>>
```

## For loop parameters

`forTestError_3.hip` contains tests for the input parameters of the for loop. Here we are testing using different variable types in the parameters.

```
24  initialiser forTestError3 {  
25      int x = 0;  
26  
27      for i = "str" to (height-1){  
28          x = x + 1;  
29      }  
30  
31  }
```

The code unsuccessfully compiles. The `start` parameter expected an identifier or numerical constant. Instead we have passed a string.

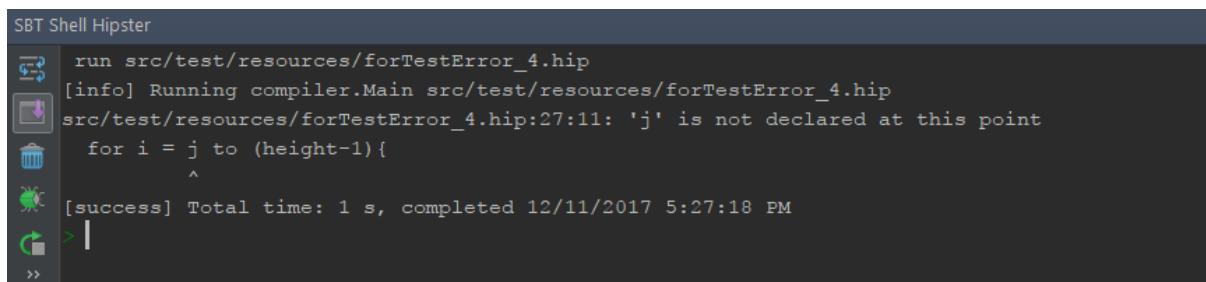


```
SBT Shell Hipster  
run src/test/resources/forTestError_3.hip  
[info] Running compiler.Main src/test/resources/forTestError_3.hip  
src/test/resources/forTestError_3.hip:27:11: identifier or numerical constant expected  
    for i = "str" to (height-1){  
           ^  
> |  
>>
```

`forTestError_4.hip` contains a for loop that uses `j` as its starting index.

```
24  initialiser forTestError4 {  
25      int x = 0;  
26  
27      for i = j to (height-1){  
28          x = x + 1;  
29      }  
30  
31 }
```

An identifier is a valid input for the input parameter. However, `j` has not been declared and cannot be used.

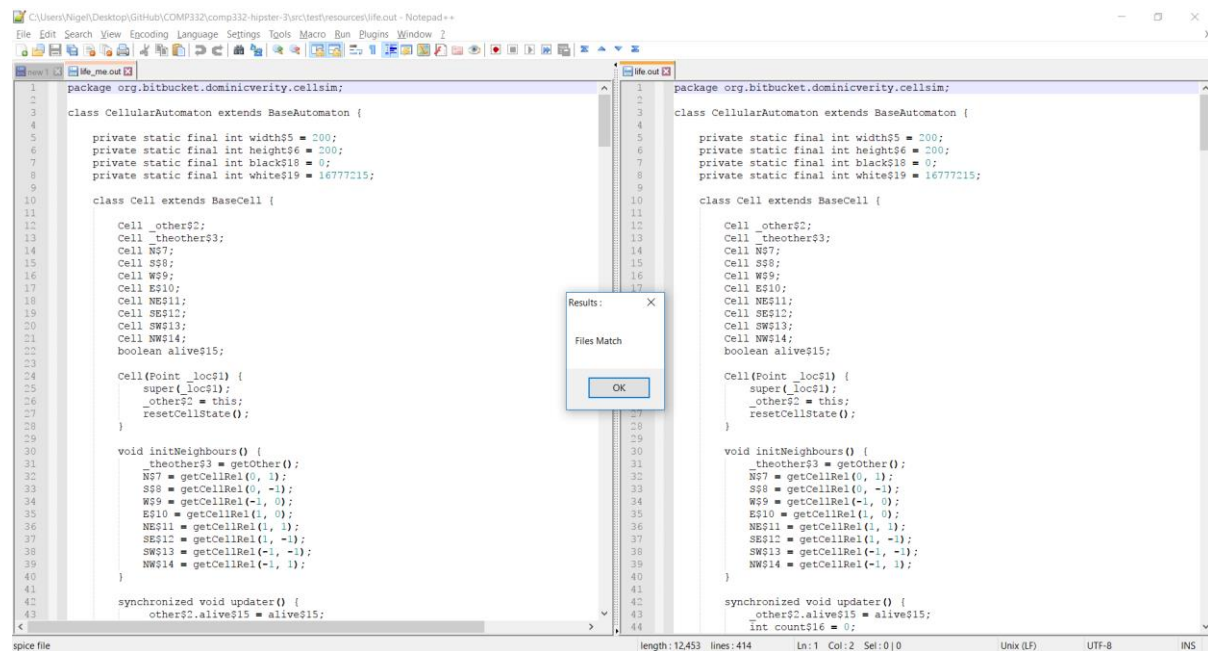


```
SBT Shell Hipster  
run src/test/resources/forTestError_4.hip  
[info] Running compiler.Main src/test/resources/forTestError_4.hip  
src/test/resources/forTestError_4.hip:27:11: 'j' is not declared at this point  
    for i = j to (height-1){  
           ^  
[success] Total time: 1 s, completed 12/11/2017 5:27:18 PM  
> |  
>>
```

## Testing all types of loops

Using `life.hip` as a test file is useful. It contains not only all the types of loops, but also other declaration and statements with it. So, it tests a variety of things.

Executing `run src/test/resources/life.hip` and saving its output to `life_me.out`, we can compare the output of my solution with the provided output. Using a comparing plug-in in Notepad++, it has informed me that they are identical.



Next, to verify that the translation is correct we can use the CellSim to compile it. The contents of `life_me.out` is put in `CellularAutomaton.java`, replacing everything in there.

We can now build and execute.



CellSim has successfully compiled and executed the code translated from Hipster.