Module code: CS4201
Matriculation number: 200017941

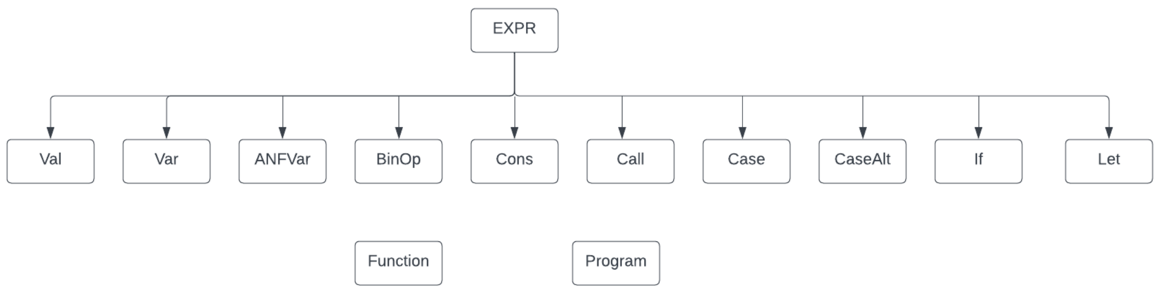
P2 - Intermediate Representation

Overview

In this practical I have implemented part of a compiler for a small functional language (Defun). This involved converting between different intermediate representations and then finally to Java which is the target language. The three compilation steps involve: defunctionalizing the program, converting the defunctionalized program to ANF and generating Java code from the ANF. I chose to use Java as the implementing language, and the compiler that I wrote achieves two out of three of these steps. It converts the AST of the functional language to ANF and then generates Java code. It does this for all the given expression types and therefore completely implements the language end-to-end as long as functions aren't being passed as arguments.

Design and Implementation

The AST of this functional language follows an object-oriented representation where each type of expression defined in the given Haskell data structure is defined as a class in Java. *EXPR* is an abstract parent class, and all the other expression types extend this parent class. The class structure is displayed in the diagram below:

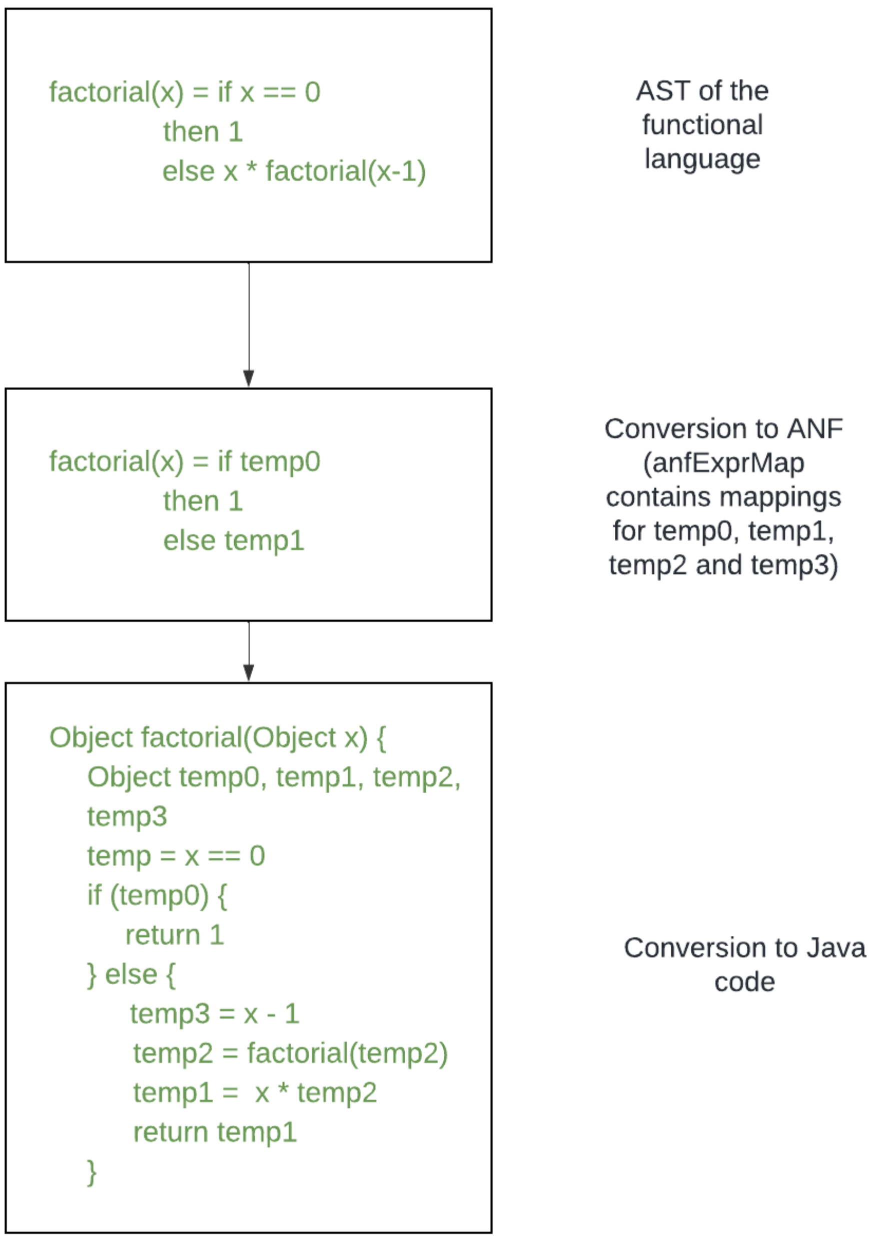


The main functionality of this compiler is encapsulated in two classes. The first is *ANFConversion* which converts the AST to ANF. A recursive function is defined to go through the given AST due to its recursive nature, and a switch conditional is used within this function to identify each type of expression to perform the appropriate action. This structure is used throughout the compiler for each step. The main thing to highlight here is the use of a map collection (*anfExprMap*) which maps from *EXPR* to *EXPR*. A new key to the collection is a new expression type which was created to represent expressions in ANF, called *ANFVar*. Therefore, a new class has been added that extends the EXPR abstract class. *ANFVar* is the variable name given to complex expressions and the value for this is the actual AST of the complex expression. Note that these variables have "temp" in their name. This conversion is done for *BinOp*, *Call*, parameters of *Call*, *Cons*, arguments of *Cons*, *Let* bound expression, and in the body of conditionals which can contain the above-mentioned expressions. The second key thing highlight is the use of another map collection (*anfStrMap*), which maps from *EXPR* to *List<String>.* The purpose of this collection is to take each *ANFVar* and map it to the list of sub-expressions that it contains as strings. This is useful in situations where an expression is recursively defined multiple times. For example, if a *BinOp* expression contains two other *BinOp* expressions: *(x + y) – (a * b).* This then needs to be converted to *temp0 = temp1 – temp2, temp1 = x + y and temp2 = a * b*. The key to *anfExprMap* would then be *temp0*, *temp1* and *temp2*, which contains their respective *BinOp* expression as the value. The new collection *anfStrMap* then maps they key temp0 to the values *temp0 = temp1 – temp2, temp1 = x + y and temp2 = a * b*, where each expression is an index into the list. Both these maps form the foundation for the operation of the compiler and are used in the next step which is to generate Java.

The second class, *GenerateJava*, converts the ANF to java code and then evaluates this code to produce an output. Again, both these steps make use of a recursive function. To generate the Java code, *anfStrMap* is used so that whenever an *ANFVar* is detected, the appropriate expressions are printed, which is a list of subexpressions. So, the detected *ANFVar* acts as the key to this collection and the list of sub-expressions is retrieved and printed. Note that to allow this, a case for *ANFVar* is added to the switch conditional. To evaluate the Java code, this time *anfExprMap* is used since the processing of the AST requires the actual expression to be retrieved instead of its string representation.

The *Compiler* class contains the ASTs of the functional language which are given as input to first *ANFConversion* and then *GenerateJava* for compilation, through the *toJava()* function*.* For each compilation, two functions are involved, the main method and the actual function which produce an output. A key thing to highlight here is that the parameters given in the main method and the arguments given in the function signature are mapped to each other using yet another map collection (*argMap*). This allows the compiler to evaluate the generated java code using the given parameters and works along the *anfExprMap* map to provide this functionality.

The diagram below shows the compilation steps for *factorial(x)* which is one of the given examples in the starter code:


AST of the functional language
Conversion to ANF (anfExprMap contains mappings for temp0, temp1, temp2 and temp3)
Conversion to Java code

```
factorial(x) = if x == 0
          then 1
          else x * factorial(x-1)
```

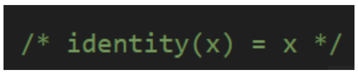
```
factorial(x) = if temp0
          then 1
          else temp1
```

```
Object factorial(Object x) {
   Object temp0, temp1, temp2,
   temp3
   temp = x == 0
   if (temp0) {
      return 1
   } else {
      temp3 = x - 1
      temp2 = factorial(temp2)
      temp1 =  x * temp2
      return temp1
   }
```

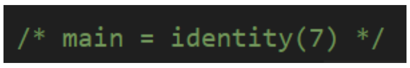
Testing

To thoroughly test the compiler, I defined several ASTs representing a diverse range of functions defined in the functional language. This involves functions which perform basic arithmetic operations to functions that have several nested *if* and *case* conditionals. The result of all the testing that was conducted during my development is given below. For each test case, I have specified the function which is being compiled and the parameters being passed to this function in the main method. This is written in the functional language and represents the AST that the compiler takes as input. Along with this, I have included a screen shot of the generated java code and the evaluated output which is the output of the compiler. Note that all these examples are included in the *Compiler* class and can be run individually by commenting out all the other calls to *toJava()* except for the program being compiled. This is precisely how I tested the compiler.

- Function which returns the given parameter.
    - Input:

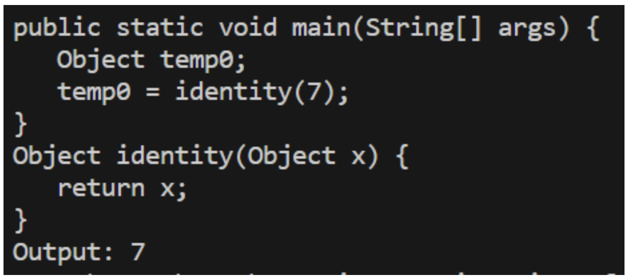


```
/* identity(x) = x */
```



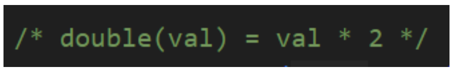
```
/* main = identity(7) */
```

    - Output:

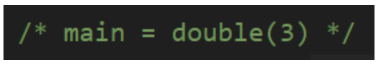


```
public static void main(String[] args) {
    Object temp0;
    temp0 = identity(7);
}
Object identity(Object x) {
    return x;
}
Output: 7
```

- Function which performs a simple arithmetic operation.
    - Input:

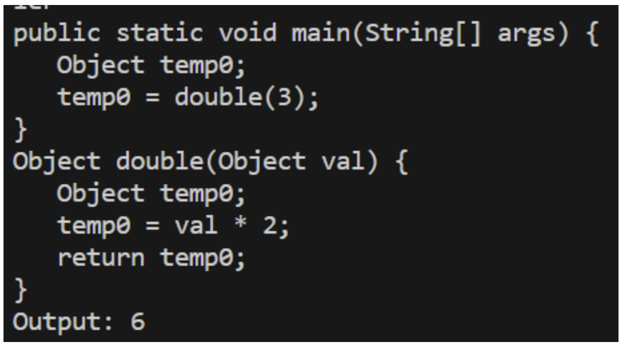


```
/* double(val) = val * 2 */
```



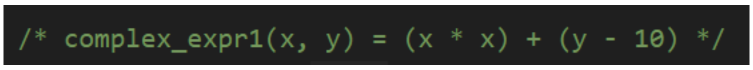
```
/* main = double(3) */
```

    - Output:

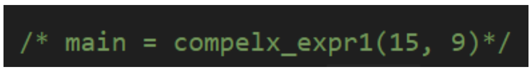


```
public static void main(String[] args) {
    Object temp0;
    temp0 = double(3);
}
Object double(Object val) {
    Object temp0;
    temp0 = val * 2;
    return temp0;
}
Output: 6
```

- Function which performs a more complex arithmetic operation.
    - Input:

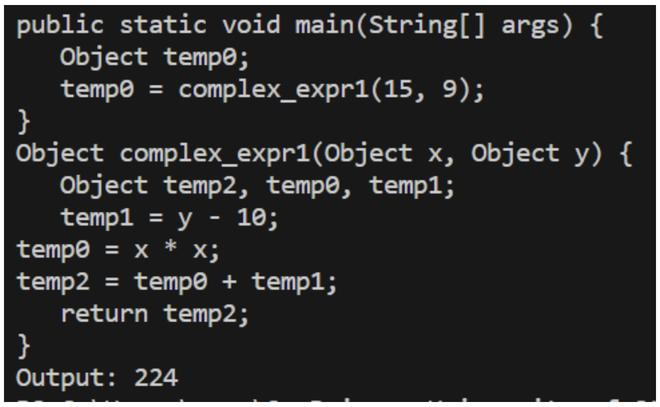


```
/* complex_expr1(x, y) = (x * x) + (y - 10) */
```



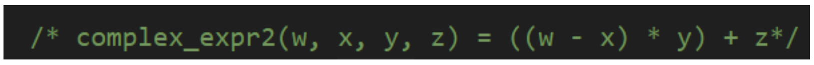
```
/* main = compelx_expr1(15, 9)*/
```

  - Output:

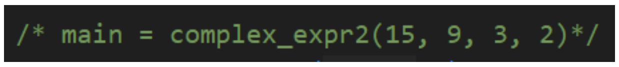


```
public static void main(String[] args) {
   Object temp0;
   temp0 = complex_expr1(15, 9);
}
Object complex_expr1(Object x, Object y) {
   Object temp2, temp0, temp1;
   temp1 = y - 10;
temp0 = x * x;
temp2 = temp0 + temp1;
   return temp2;
}
Output: 224
```

- Function which performs a more complex arithmetic operation.
  - Input:

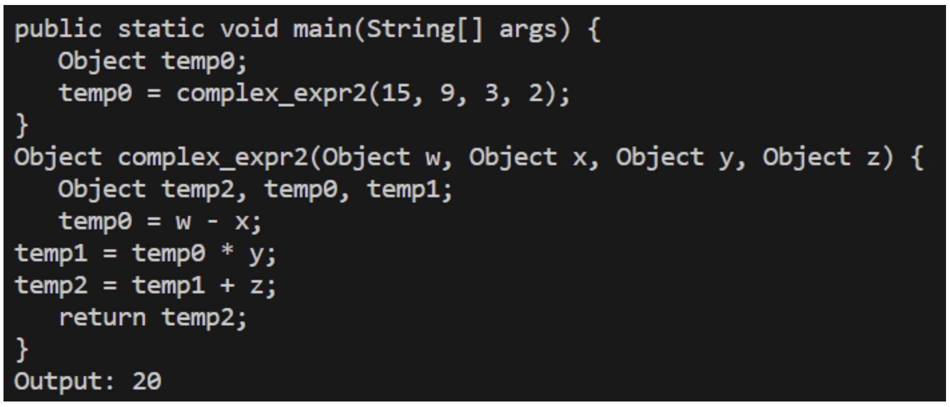


```
/* complex_expr2(w, x, y, z) = ((w - x) * y) + z*/
```



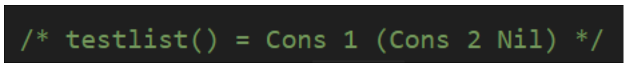
```
/* main = complex_expr2(15, 9, 3, 2)*/
```

  - Output:

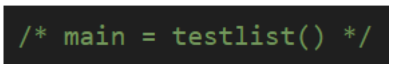


```
public static void main(String[] args) {
   Object temp0;
   temp0 = complex_expr2(15, 9, 3, 2);
}
Object complex_expr2(Object w, Object x, Object y, Object z) {
   Object temp2, temp0, temp1;
   temp0 = w - x;
temp1 = temp0 * y;
temp2 = temp1 + z;
   return temp2;
}
Output: 20
```

- Function which returns a list constructor.
  - Input:

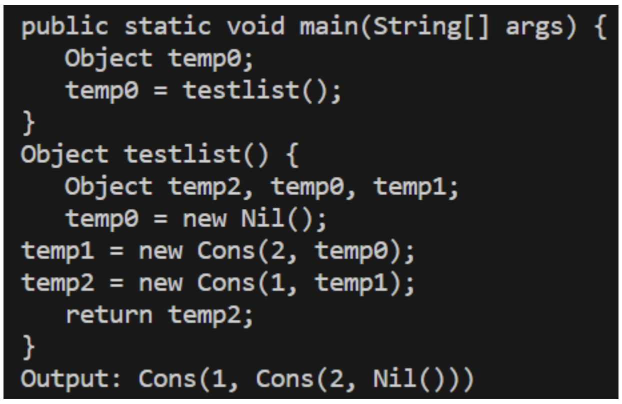


```
/* testlist() = Cons 1 (Cons 2 Nil) */
```



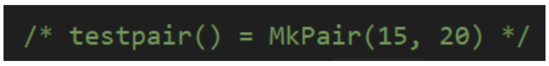
```
/* main = testlist() */
```

  - Output:

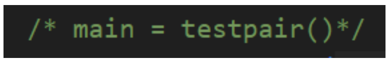


```
public static void main(String[] args) {
   Object temp0;
   temp0 = testlist();
}
Object testlist() {
   Object temp2, temp0, temp1;
   temp0 = new Nil();
temp1 = new Cons(2, temp0);
temp2 = new Cons(1, temp1);
   return temp2;
}
Output: Cons(1, Cons(2, Nil()))
```

- Function which returns a pair constructor.
  - Input:

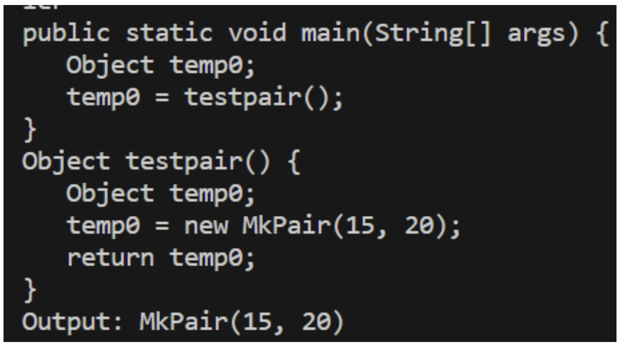


```
/* testpair() = MkPair(15, 20) */
```



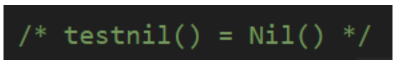
```
/* main = testpair()*/
```

  - Output:

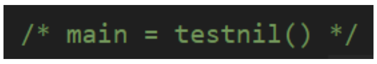


```
public static void main(String[] args) {
   Object temp0;
   temp0 = testpair();
}
Object testpair() {
   Object temp0;
   temp0 = new MkPair(15, 20);
   return temp0;
}
Output: MkPair(15, 20)
```

- Function which returns a Nil constructor.
  - Input:

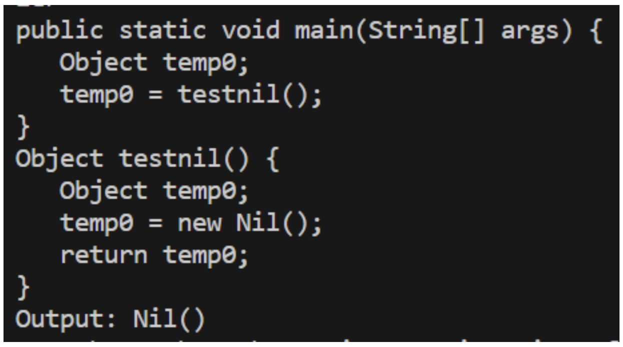


```
/* testnil() = Nil() */
```



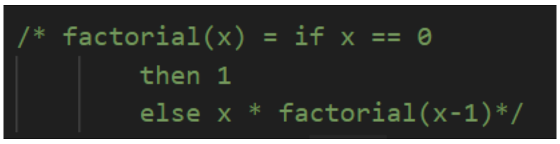
```
/* main = testnil() */
```

  - Output:

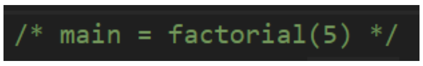


```
public static void main(String[] args) {
   Object temp0;
   temp0 = testnil();
}
Object testnil() {
   Object temp0;
   temp0 = new Nil();
   return temp0;
}
Output: Nil()
```

- Factorial function
    - Input:

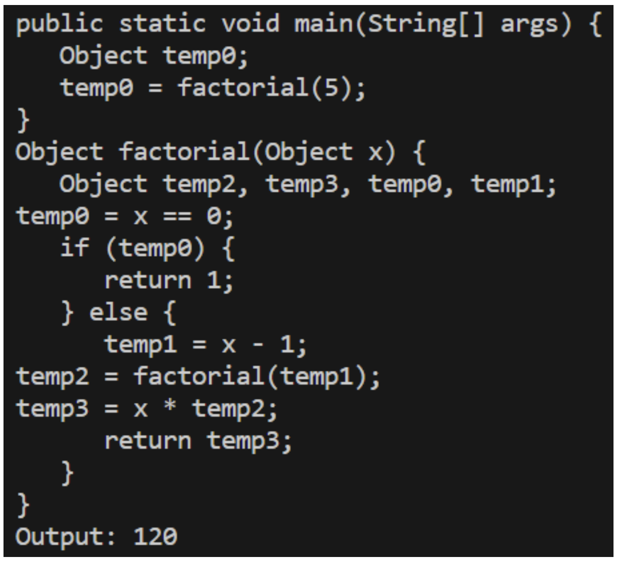


```
/* factorial(x) = if x == 0
        then 1
        else x * factorial(x-1)*/
```



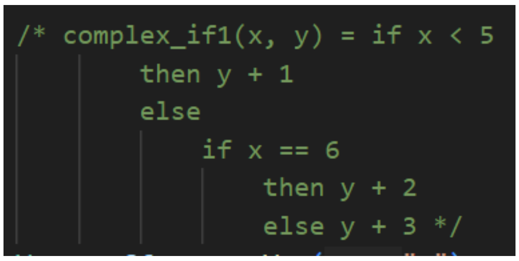
```
/* main = factorial(5) */
```

    - Output:

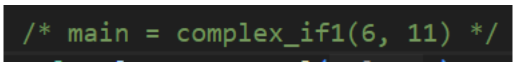


```
public static void main(String[] args) {
   Object temp0;
   temp0 = factorial(5);
}
Object factorial(Object x) {
   Object temp2, temp3, temp0, temp1;
temp0 = x == 0;
   if (temp0) {
      return 1;
   } else {
      temp1 = x - 1;
temp2 = factorial(temp1);
temp3 = x * temp2;
      return temp3;
   }
}
Output: 120
```

- Function which has a nested if conditional
    - Input:

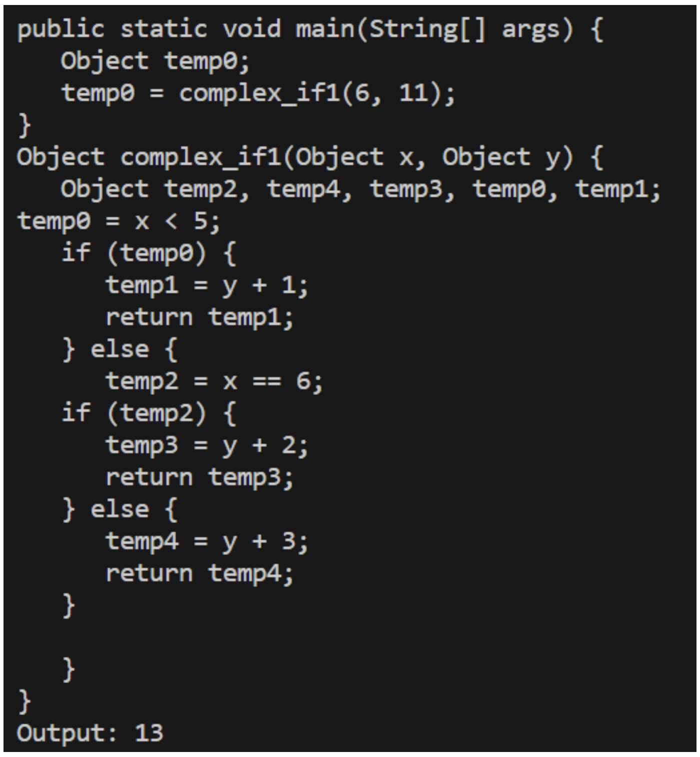


```
/* complex_if1(x, y) = if x < 5
        then y + 1
        else
            if x == 6
                then y + 2
                else y + 3 */
```



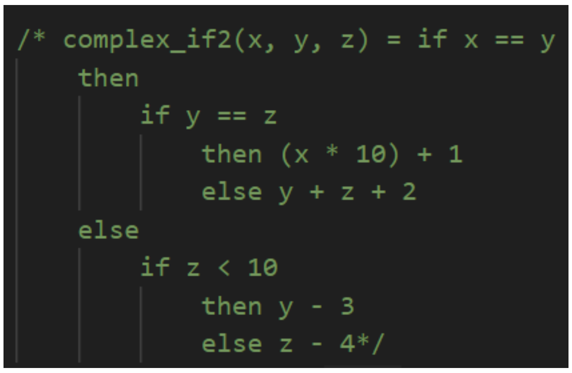
```
/* main = complex_if1(6, 11) */
```

- Output:

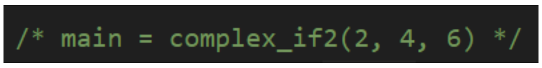


```
public static void main(String[] args) {
   Object temp0;
   temp0 = complex_if1(6, 11);
}
Object complex_if1(Object x, Object y) {
   Object temp2, temp4, temp3, temp0, temp1;
temp0 = x < 5;
   if (temp0) {
      temp1 = y + 1;
      return temp1;
   } else {
      temp2 = x == 6;
   if (temp2) {
      temp3 = y + 2;
      return temp3;
   } else {
      temp4 = y + 3;
      return temp4;
   }

   }
}
Output: 13
```

- Function which has multiple nested if conditionals.
  - Input:

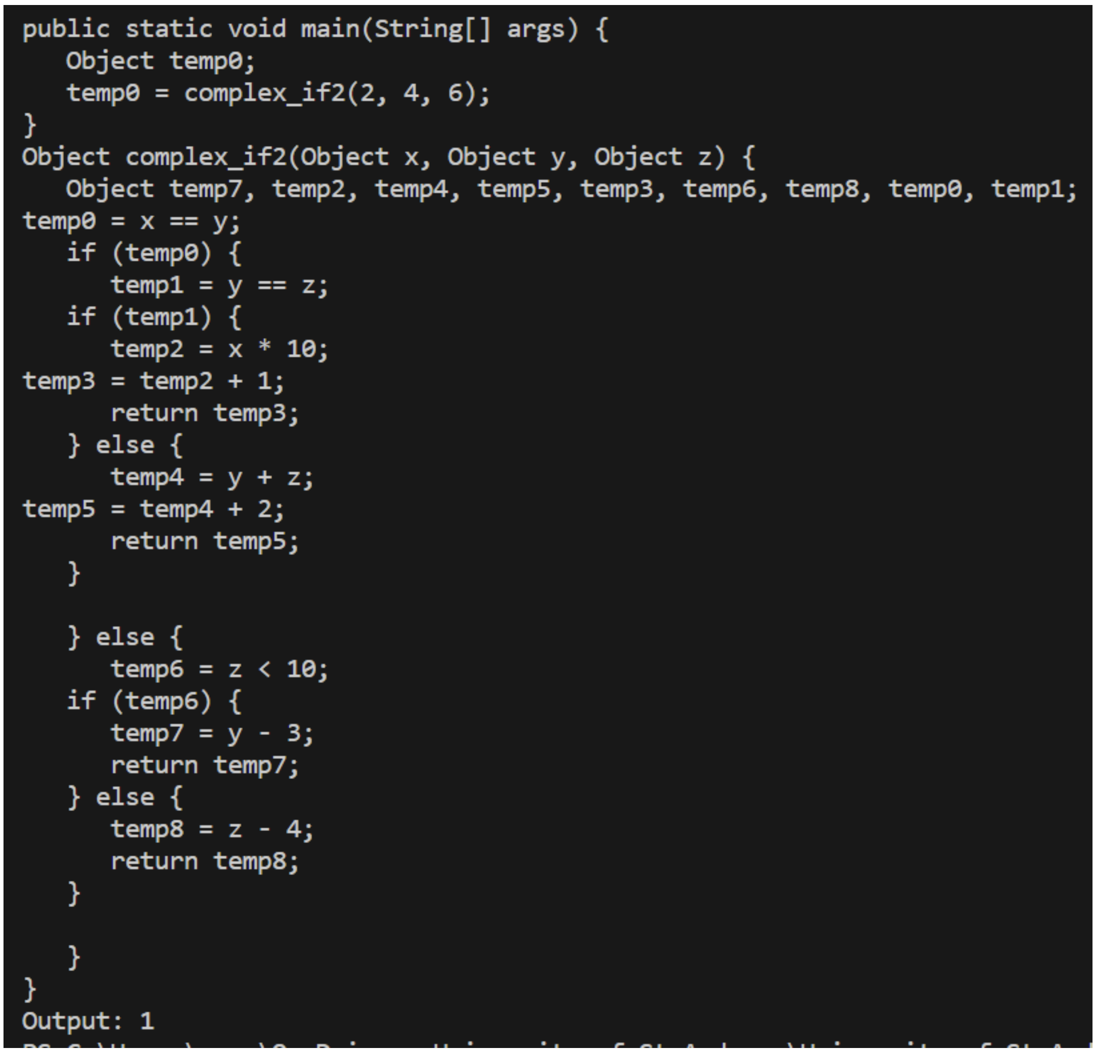


```
/* complex_if2(x, y, z) = if x == y
    then
        if y == z
            then (x * 10) + 1
            else y + z + 2
    else
        if z < 10
            then y - 3
            else z - 4*/
```



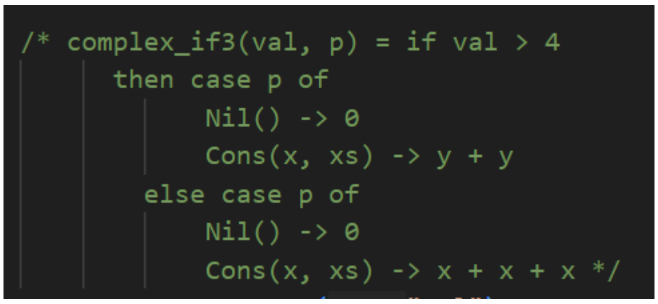
```
/* main = complex_if2(2, 4, 6) */
```

  - Output:

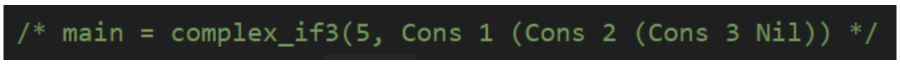


```
public static void main(String[] args) {
   Object temp0;
   temp0 = complex_if2(2, 4, 6);
}
Object complex_if2(Object x, Object y, Object z) {
   Object temp7, temp2, temp4, temp5, temp3, temp6, temp8, temp0, temp1;
temp0 = x == y;
   if (temp0) {
      temp1 = y == z;
   if (temp1) {
      temp2 = x * 10;
temp3 = temp2 + 1;
      return temp3;
   } else {
      temp4 = y + z;
temp5 = temp4 + 2;
      return temp5;
   }

   } else {
      temp6 = z < 10;
   if (temp6) {
      temp7 = y - 3;
      return temp7;
   } else {
      temp8 = z - 4;
      return temp8;
   }

   }
}
Output: 1
```

- Function which contains a case conditional nested within an if conditional.
    - Input:

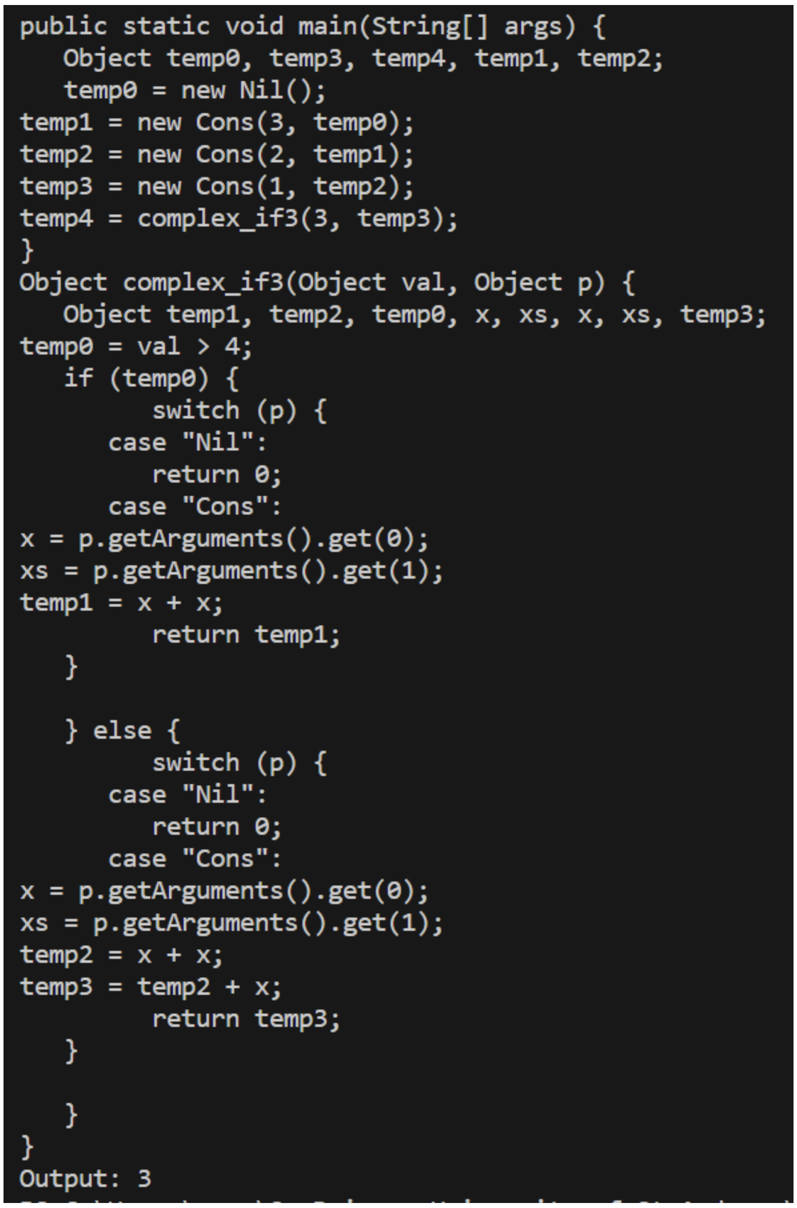


```
/* complex_if3(val, p) = if val > 4
      then case p of
            Nil() -> 0
            Cons(x, xs) -> y + y
        else case p of
            Nil() -> 0
            Cons(x, xs) -> x + x + x */
```



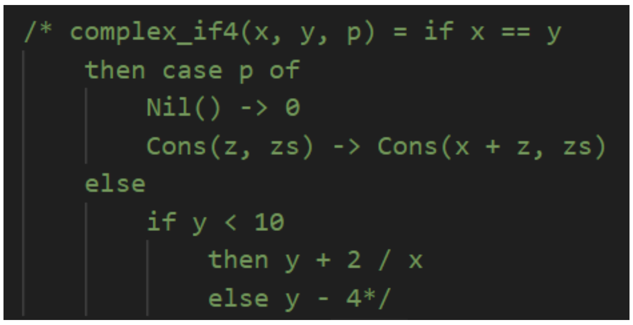
```
/* main = complex_if3(5, Cons 1 (Cons 2 (Cons 3 Nil)) */
```

    - Output:

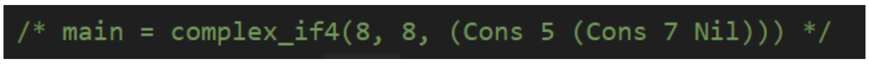


```
public static void main(String[] args) {
   Object temp0, temp3, temp4, temp1, temp2;
   temp0 = new Nil();
temp1 = new Cons(3, temp0);
temp2 = new Cons(2, temp1);
temp3 = new Cons(1, temp2);
temp4 = complex_if3(3, temp3);
}
Object complex_if3(Object val, Object p) {
   Object temp1, temp2, temp0, x, xs, x, xs, temp3;
temp0 = val > 4;
   if (temp0) {
         switch (p) {
      case "Nil":
         return 0;
      case "Cons":
x = p.getArguments().get(0);
xs = p.getArguments().get(1);
temp1 = x + x;
         return temp1;
   }

   } else {
         switch (p) {
      case "Nil":
         return 0;
      case "Cons":
x = p.getArguments().get(0);
xs = p.getArguments().get(1);
temp2 = x + x;
temp3 = temp2 + x;
         return temp3;
   }

   }
}
Output: 3
```

- Function which contains a nested if and case conditional within an if conditional.
    - Input:

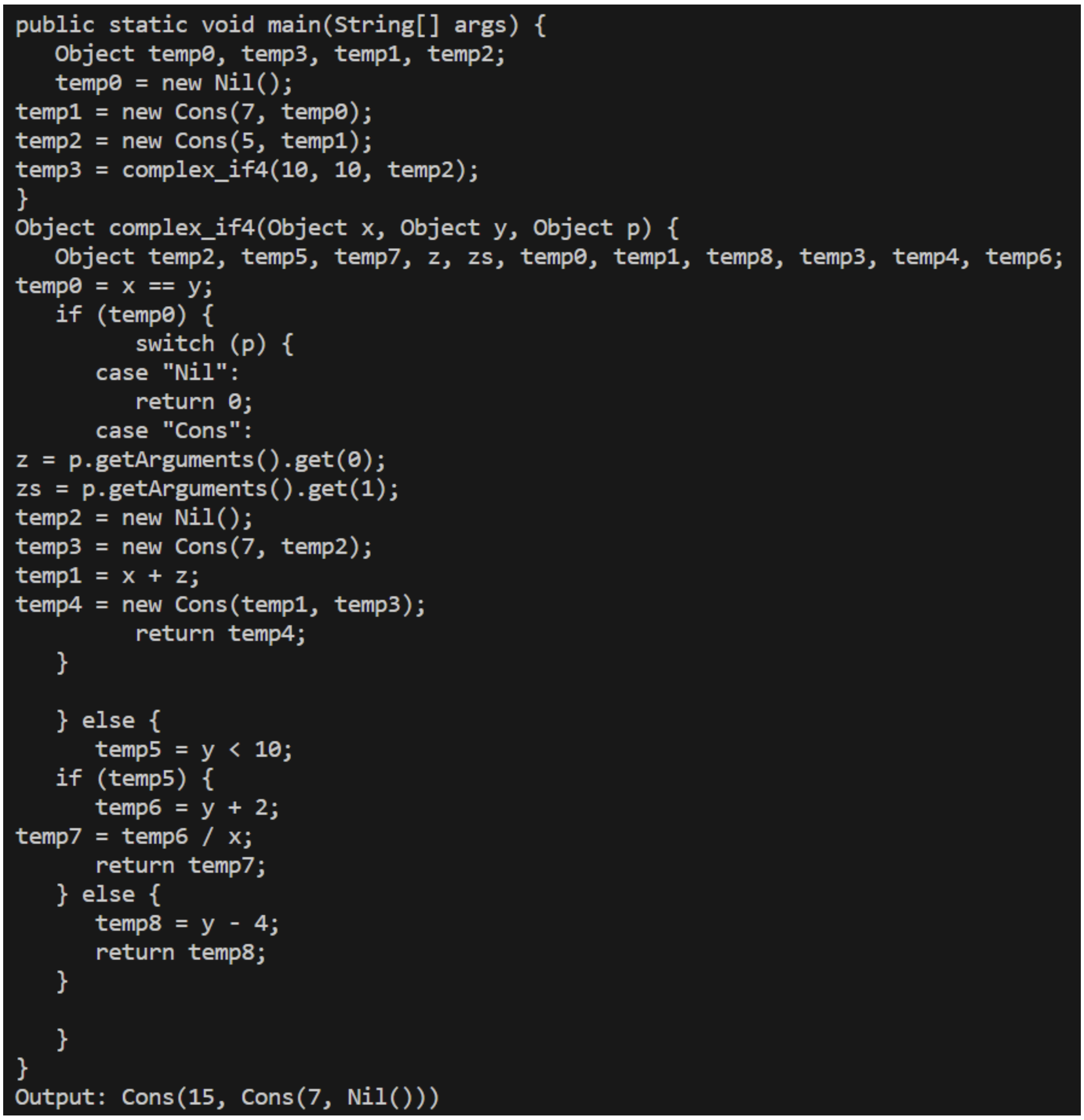


```
/* complex_if4(x, y, p) = if x == y
    then case p of
        Nil() -> 0
        Cons(z, zs) -> Cons(x + z, zs)
    else
        if y < 10
            then y + 2 / x
            else y - 4*/
```



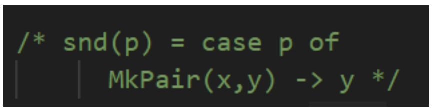
```
/* main = complex_if4(8, 8, (Cons 5 (Cons 7 Nil))) */
```

- Output:

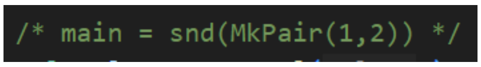


```
public static void main(String[] args) {
   Object temp0, temp3, temp1, temp2;
   temp0 = new Nil();
temp1 = new Cons(7, temp0);
temp2 = new Cons(5, temp1);
temp3 = complex_if4(10, 10, temp2);
}
Object complex_if4(Object x, Object y, Object p) {
   Object temp2, temp5, temp7, z, zs, temp0, temp1, temp8, temp3, temp4, temp6;
temp0 = x == y;
   if (temp0) {
         switch (p) {
      case "Nil":
         return 0;
      case "Cons":
z = p.getArguments().get(0);
zs = p.getArguments().get(1);
temp2 = new Nil();
temp3 = new Cons(7, temp2);
temp1 = x + z;
temp4 = new Cons(temp1, temp3);
         return temp4;
   }

   } else {
      temp5 = y < 10;
   if (temp5) {
      temp6 = y + 2;
temp7 = temp6 / x;
      return temp7;
   } else {
      temp8 = y - 4;
      return temp8;
   }

   }
}
Output: Cons(15, Cons(7, Nil()))
```

- Function which contains a case conditional.
  - Input:

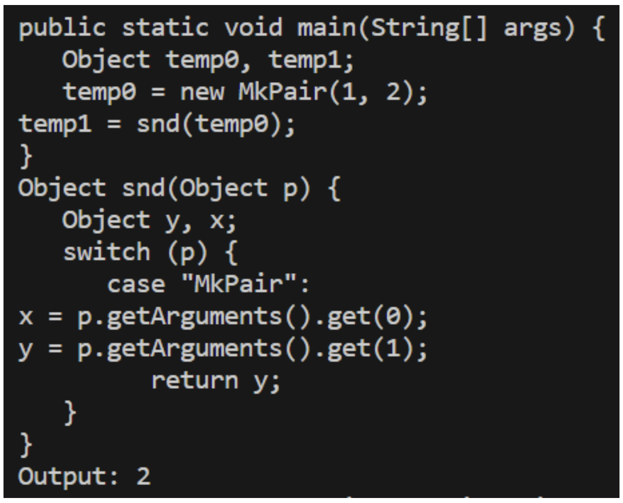


```
/* snd(p) = case p of
      MkPair(x,y) -> y */
```



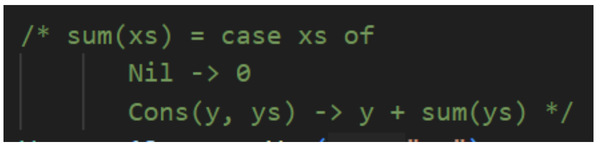
```
/* main = snd(MkPair(1,2)) */
```

  - Output:

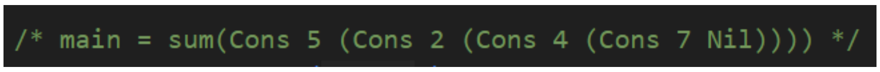


```
public static void main(String[] args) {
   Object temp0, temp1;
   temp0 = new MkPair(1, 2);
temp1 = snd(temp0);
}
Object snd(Object p) {
   Object y, x;
   switch (p) {
      case "MkPair":
x = p.getArguments().get(0);
y = p.getArguments().get(1);
         return y;
   }
}
Output: 2
```

- Function which contains a case conditional with a recursive call.
    - Input:

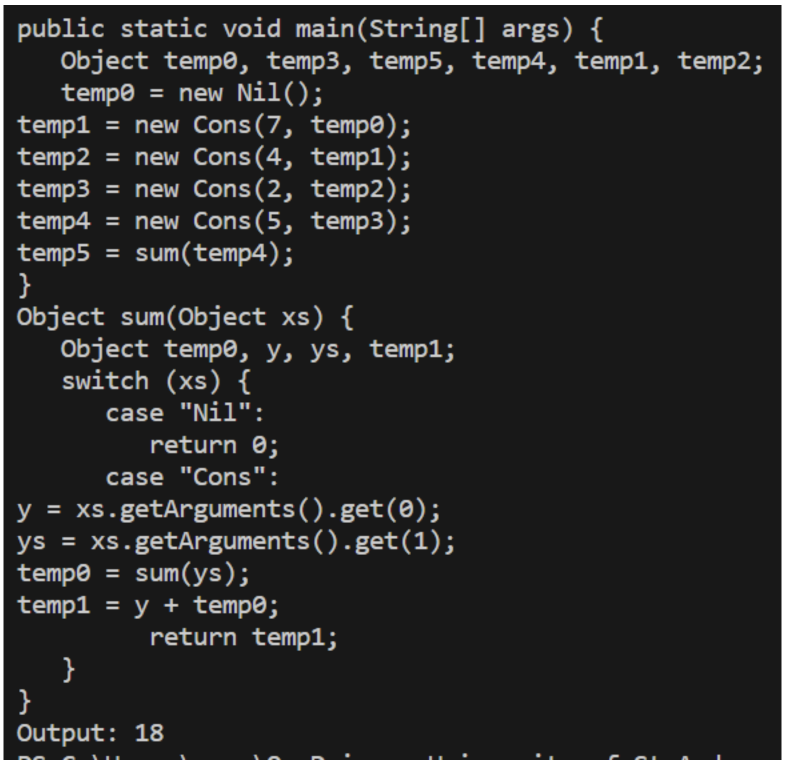


```
/* sum(xs) = case xs of
        Nil -> 0
        Cons(y, ys) -> y + sum(ys) */
```



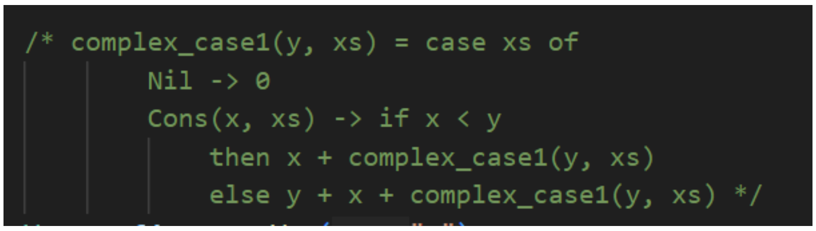
```
/* main = sum(Cons 5 (Cons 2 (Cons 4 (Cons 7 Nil)))) */
```

- Output:

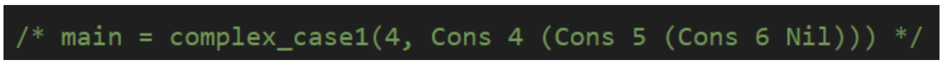


```
public static void main(String[] args) {
   Object temp0, temp3, temp5, temp4, temp1, temp2;
   temp0 = new Nil();
temp1 = new Cons(7, temp0);
temp2 = new Cons(4, temp1);
temp3 = new Cons(2, temp2);
temp4 = new Cons(5, temp3);
temp5 = sum(temp4);
}
Object sum(Object xs) {
   Object temp0, y, ys, temp1;
   switch (xs) {
      case "Nil":
         return 0;
      case "Cons":
y = xs.getArguments().get(0);
ys = xs.getArguments().get(1);
temp0 = sum(ys);
temp1 = y + temp0;
         return temp1;
   }
}
Output: 18
```

- Function which contains an if conditional nested within a case conditional with a recursive call.
    - Input:

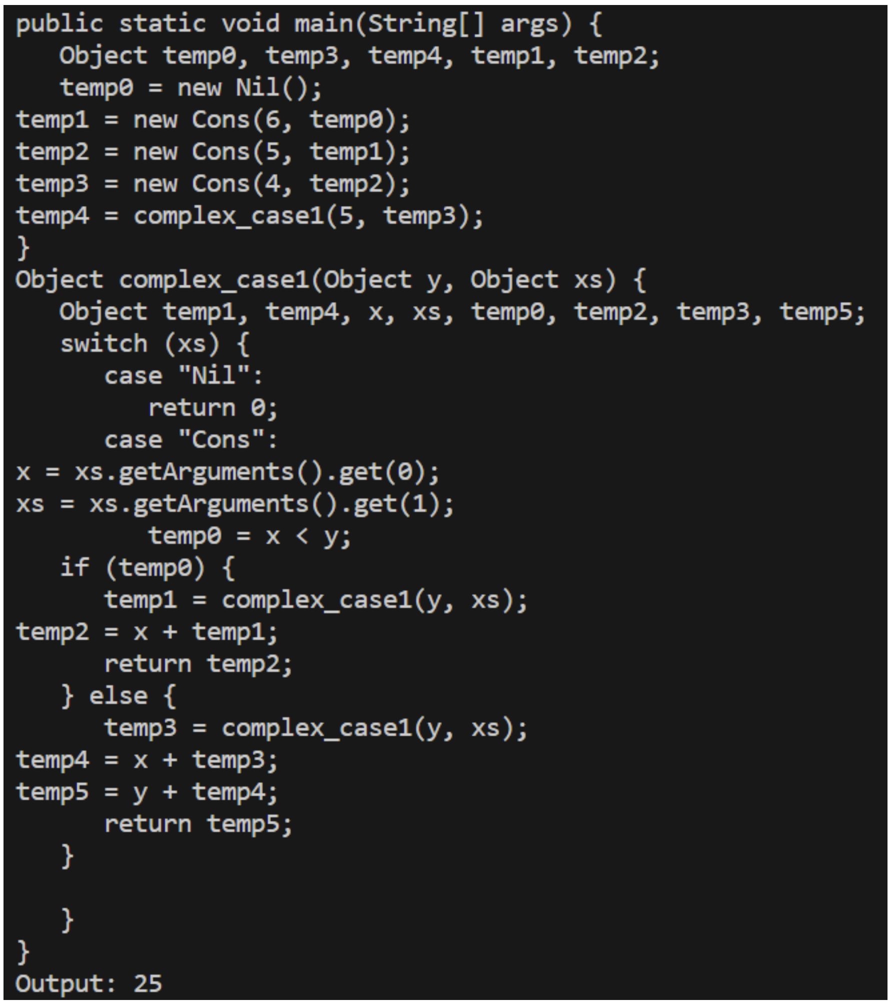


```
/* complex_case1(y, xs) = case xs of
        Nil -> 0
        Cons(x, xs) -> if x < y
            then x + complex_case1(y, xs)
            else y + x + complex_case1(y, xs) */
```



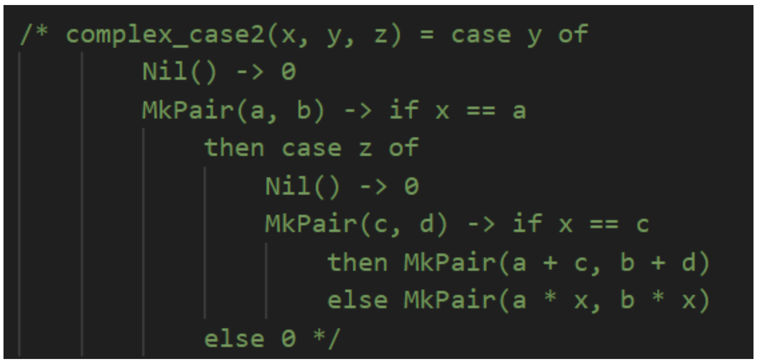
```
/* main = complex_case1(4, Cons 4 (Cons 5 (Cons 6 Nil))) */
```

    - Output:

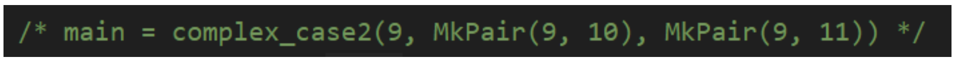


```
public static void main(String[] args) {
   Object temp0, temp3, temp4, temp1, temp2;
   temp0 = new Nil();
temp1 = new Cons(6, temp0);
temp2 = new Cons(5, temp1);
temp3 = new Cons(4, temp2);
temp4 = complex_case1(5, temp3);
}
Object complex_case1(Object y, Object xs) {
   Object temp1, temp4, x, xs, temp0, temp2, temp3, temp5;
   switch (xs) {
      case "Nil":
         return 0;
      case "Cons":
x = xs.getArguments().get(0);
xs = xs.getArguments().get(1);
         temp0 = x < y;
   if (temp0) {
      temp1 = complex_case1(y, xs);
temp2 = x + temp1;
      return temp2;
   } else {
      temp3 = complex_case1(y, xs);
temp4 = x + temp3;
temp5 = y + temp4;
      return temp5;
   }

   }
}
Output: 25
```

- Function which contains a case conditional nested within a case conditional.
    - Input:

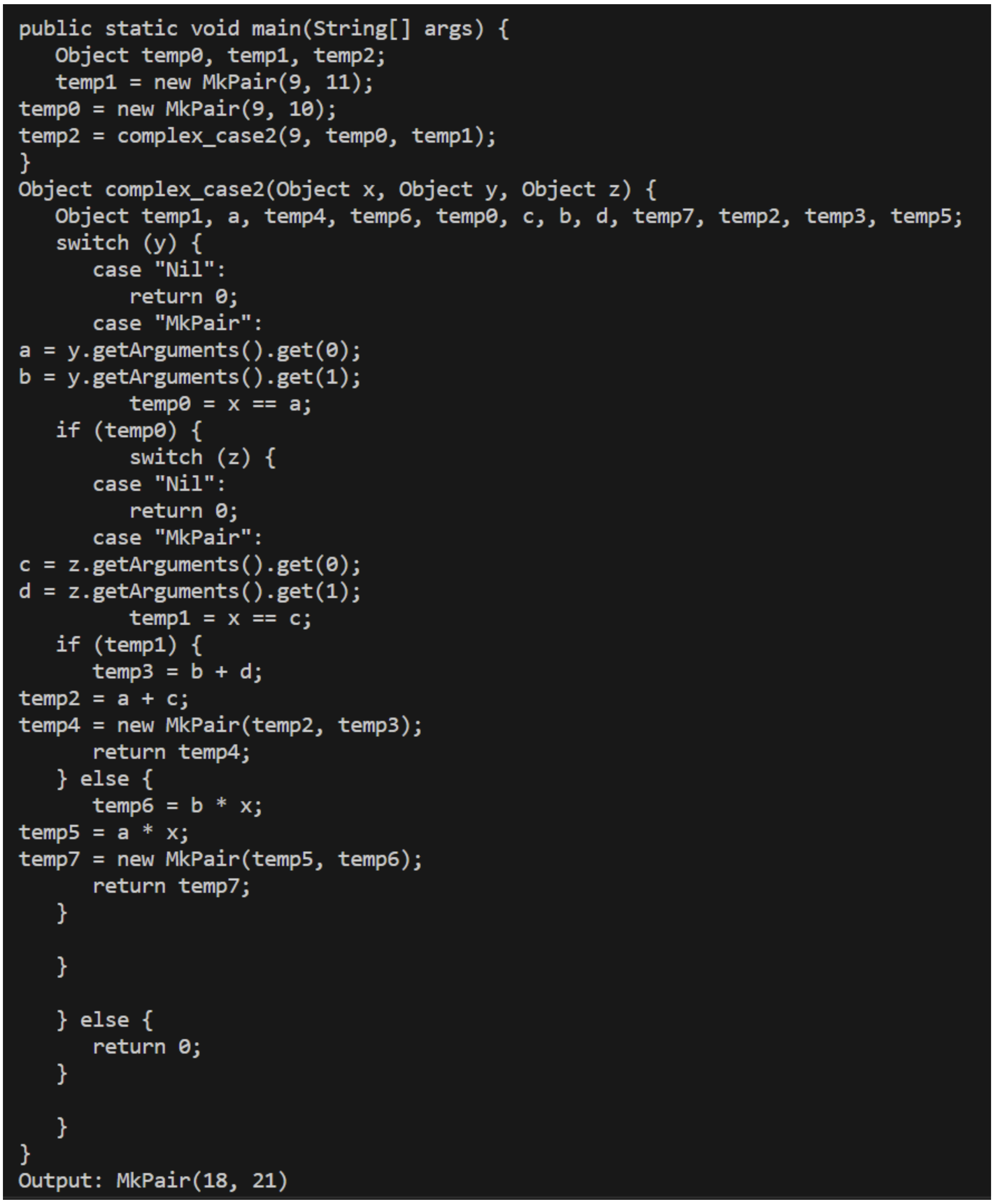


```
/* complex_case2(x, y, z) = case y of
            Nil() -> 0
            MkPair(a, b) -> if x == a
                then case z of
                    Nil() -> 0
                    MkPair(c, d) -> if x == c
                        then MkPair(a + c, b + d)
                        else MkPair(a * x, b * x)
                else 0 */
```



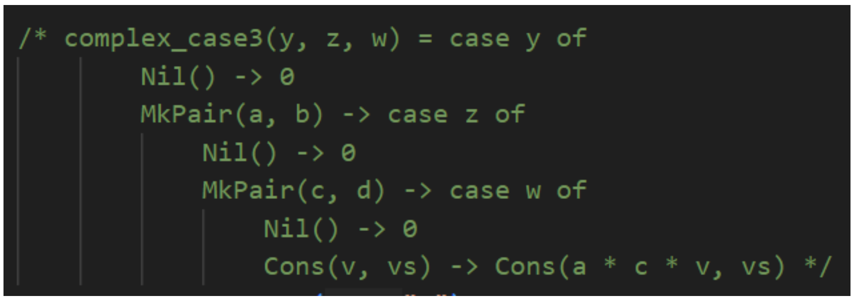
```
/* main = complex_case2(9, MkPair(9, 10), MkPair(9, 11)) */
```

- Output:

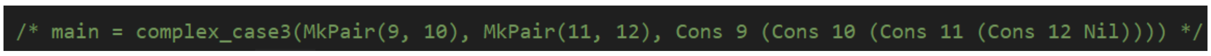


```
public static void main(String[] args) {
   Object temp0, temp1, temp2;
   temp1 = new MkPair(9, 11);
temp0 = new MkPair(9, 10);
temp2 = complex_case2(9, temp0, temp1);
}
Object complex_case2(Object x, Object y, Object z) {
   Object temp1, a, temp4, temp6, temp0, c, b, d, temp7, temp2, temp3, temp5;
   switch (y) {
      case "Nil":
         return 0;
      case "MkPair":
a = y.getArguments().get(0);
b = y.getArguments().get(1);
         temp0 = x == a;
   if (temp0) {
         switch (z) {
      case "Nil":
         return 0;
      case "MkPair":
c = z.getArguments().get(0);
d = z.getArguments().get(1);
         temp1 = x == c;
   if (temp1) {
      temp3 = b + d;
temp2 = a + c;
temp4 = new MkPair(temp2, temp3);
      return temp4;
   } else {
      temp6 = b * x;
temp5 = a * x;
temp7 = new MkPair(temp5, temp6);
      return temp7;
   }

   }

   } else {
      return 0;
   }

   }
}
Output: MkPair(18, 21)
```

- Function which contains three nested case conditionals.
    - Input:

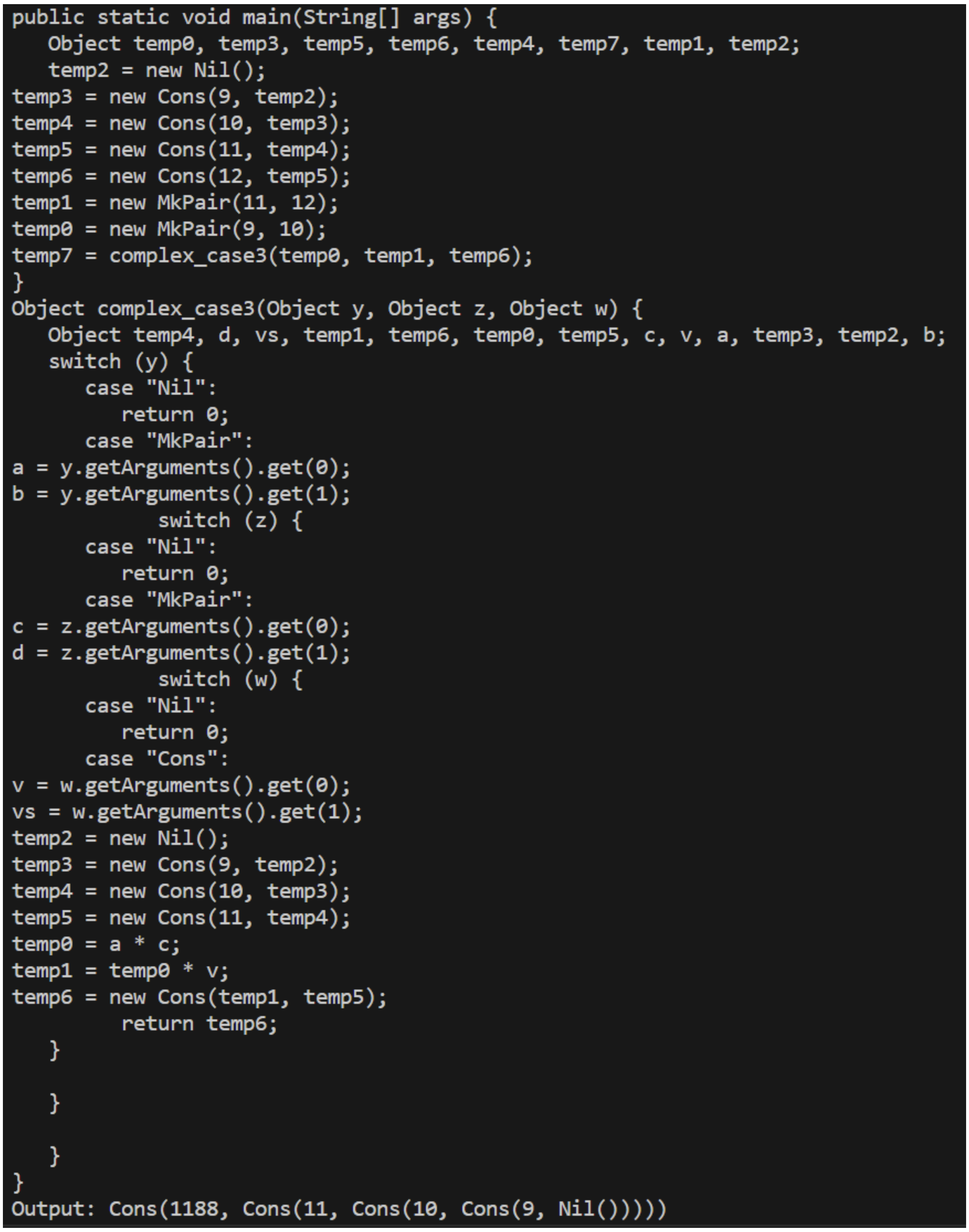


```
/* complex_case3(y, z, w) = case y of
        Nil() -> 0
        MkPair(a, b) -> case z of
            Nil() -> 0
            MkPair(c, d) -> case w of
                Nil() -> 0
                Cons(v, vs) -> Cons(a * c * v, vs) */
```



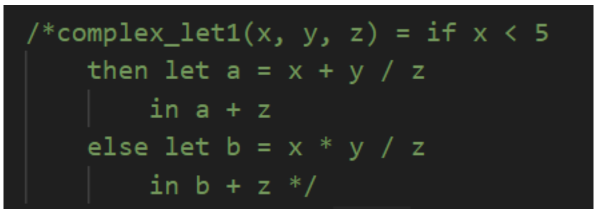
```
/* main = complex_case3(MkPair(9, 10), MkPair(11, 12), Cons 9 (Cons 10 (Cons 11 (Cons 12 Nil)))) */
```

- Output:

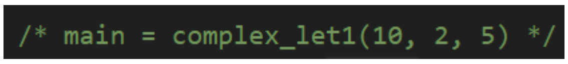


```
public static void main(String[] args) {
   Object temp0, temp3, temp5, temp6, temp4, temp7, temp1, temp2;
   temp2 = new Nil();
temp3 = new Cons(9, temp2);
temp4 = new Cons(10, temp3);
temp5 = new Cons(11, temp4);
temp6 = new Cons(12, temp5);
temp1 = new MkPair(11, 12);
temp0 = new MkPair(9, 10);
temp7 = complex_case3(temp0, temp1, temp6);
}
Object complex_case3(Object y, Object z, Object w) {
   Object temp4, d, vs, temp1, temp6, temp0, temp5, c, v, a, temp3, temp2, b;
   switch (y) {
      case "Nil":
         return 0;
      case "MkPair":
a = y.getArguments().get(0);
b = y.getArguments().get(1);
            switch (z) {
      case "Nil":
         return 0;
      case "MkPair":
c = z.getArguments().get(0);
d = z.getArguments().get(1);
            switch (w) {
      case "Nil":
         return 0;
      case "Cons":
v = w.getArguments().get(0);
vs = w.getArguments().get(1);
temp2 = new Nil();
temp3 = new Cons(9, temp2);
temp4 = new Cons(10, temp3);
temp5 = new Cons(11, temp4);
temp0 = a * c;
temp1 = temp0 * v;
temp6 = new Cons(temp1, temp5);
         return temp6;
   }

   }

   }
}
Output: Cons(1188, Cons(11, Cons(10, Cons(9, Nil()))))
```

- Function which has a let statement within an if conditional.
    - Input:

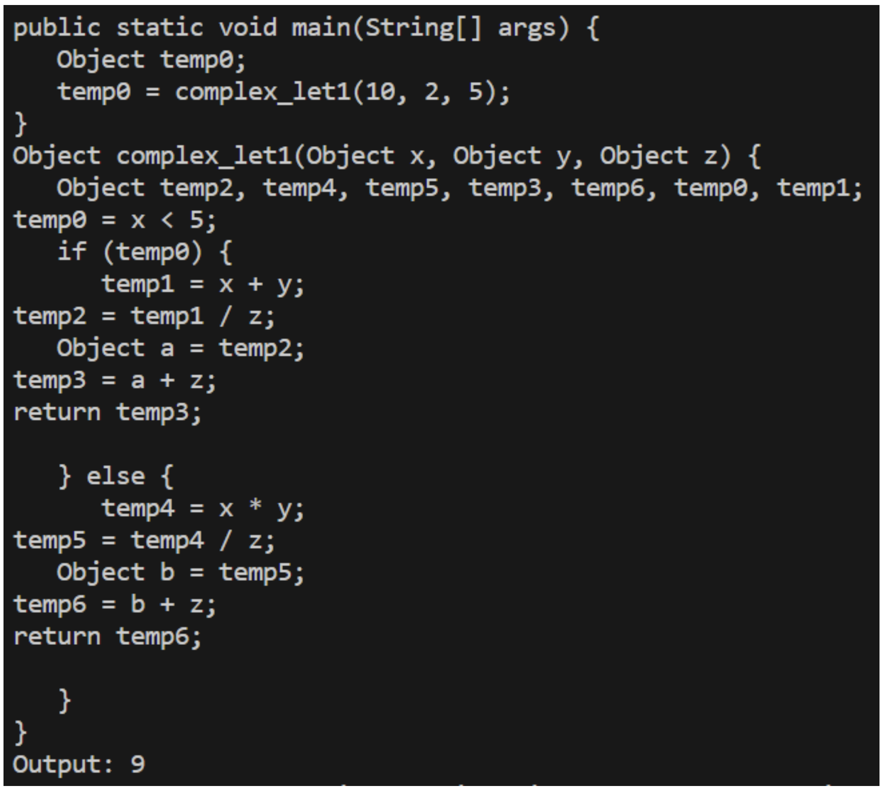


```
/*complex_let1(x, y, z) = if x < 5
    then let a = x + y / z
        in a + z
    else let b = x * y / z
        in b + z */
```



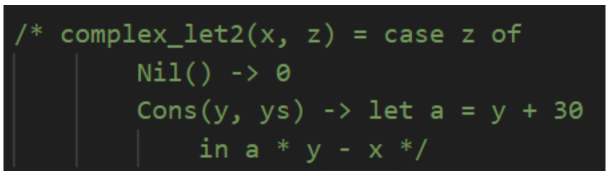
```
/* main = complex_let1(10, 2, 5) */
```

- Output:

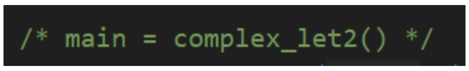


```
public static void main(String[] args) {
   Object temp0;
   temp0 = complex_let1(10, 2, 5);
}
Object complex_let1(Object x, Object y, Object z) {
   Object temp2, temp4, temp5, temp3, temp6, temp0, temp1;
temp0 = x < 5;
   if (temp0) {
      temp1 = x + y;
temp2 = temp1 / z;
   Object a = temp2;
temp3 = a + z;
return temp3;

   } else {
      temp4 = x * y;
temp5 = temp4 / z;
   Object b = temp5;
temp6 = b + z;
return temp6;

   }
}
Output: 9
```

- Function which contains a let statement within a case conditional.
  - Input:

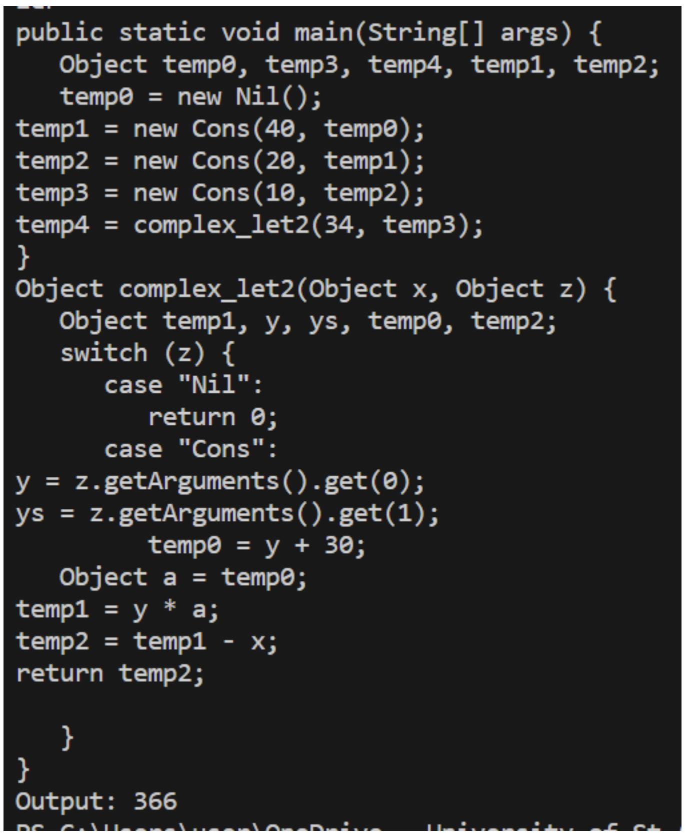


```
/* complex_let2(x, z) = case z of
        Nil() -> 0
        Cons(y, ys) -> let a = y + 30
            in a * y - x */
```



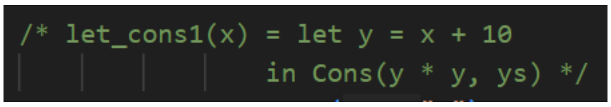
```
/* main = complex_let2() */
```

  - Output:

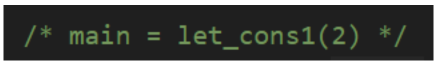


```
public static void main(String[] args) {
   Object temp0, temp3, temp4, temp1, temp2;
   temp0 = new Nil();
temp1 = new Cons(40, temp0);
temp2 = new Cons(20, temp1);
temp3 = new Cons(10, temp2);
temp4 = complex_let2(34, temp3);
}
Object complex_let2(Object x, Object z) {
   Object temp1, y, ys, temp0, temp2;
   switch (z) {
      case "Nil":
         return 0;
      case "Cons":
y = z.getArguments().get(0);
ys = z.getArguments().get(1);
         temp0 = y + 30;
   Object a = temp0;
temp1 = y * a;
temp2 = temp1 - x;
return temp2;

   }
}
Output: 366
```

- Function which has a let statement with a list constructor.
  - Input:

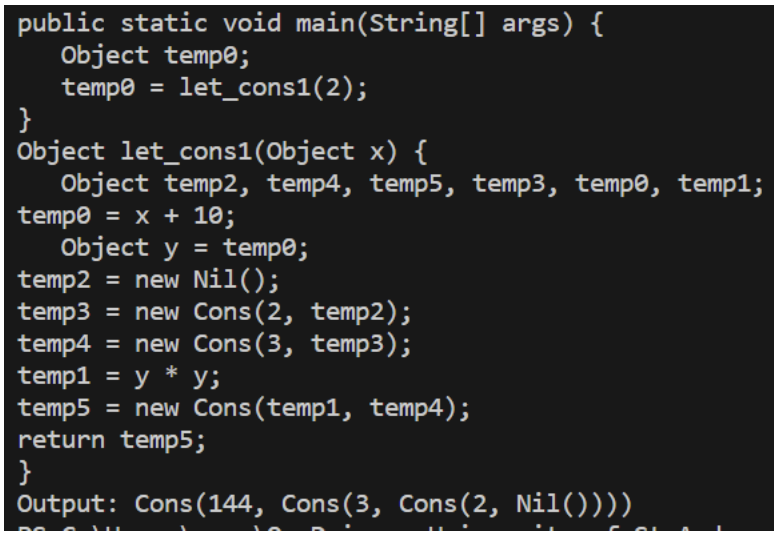


```
/* let_cons1(x) = let y = x + 10
                  in Cons(y * y, ys) */
```



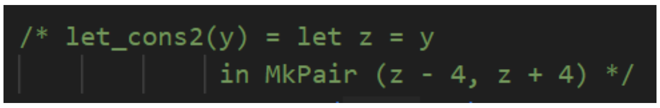
```
/* main = let_cons1(2) */
```

  - Output:

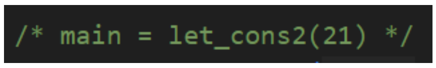


```
public static void main(String[] args) {
   Object temp0;
   temp0 = let_cons1(2);
}
Object let_cons1(Object x) {
   Object temp2, temp4, temp5, temp3, temp0, temp1;
temp0 = x + 10;
   Object y = temp0;
temp2 = new Nil();
temp3 = new Cons(2, temp2);
temp4 = new Cons(3, temp3);
temp1 = y * y;
temp5 = new Cons(temp1, temp4);
return temp5;
}
Output: Cons(144, Cons(3, Cons(2, Nil())))
```

- Function which has a let statement with a pair constructor.
  - Input:

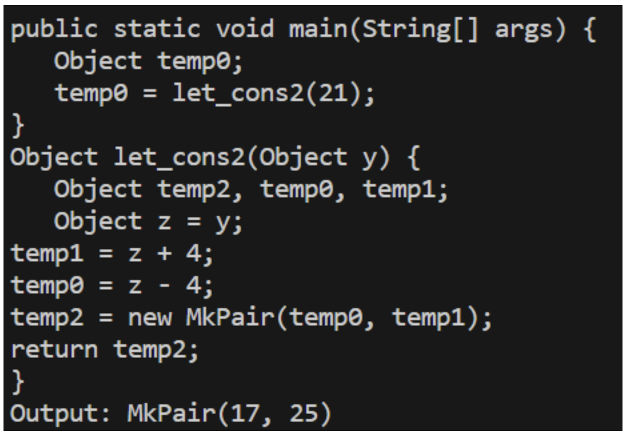


```
/* let_cons2(y) = let z = y
                  in MkPair (z - 4, z + 4) */
```



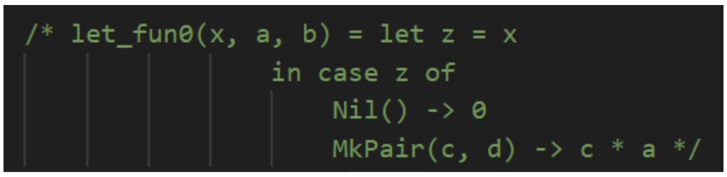
```
/* main = let_cons2(21) */
```

  - Output:

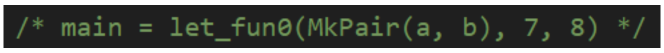


```
public static void main(String[] args) {
   Object temp0;
   temp0 = let_cons2(21);
}
Object let_cons2(Object y) {
   Object temp2, temp0, temp1;
   Object z = y;
temp1 = z + 4;
temp0 = z - 4;
temp2 = new MkPair(temp0, temp1);
return temp2;
}
Output: MkPair(17, 25)
```

- Function which contains a let statement which takes a pair constructor and a case conditional.
  - Input:

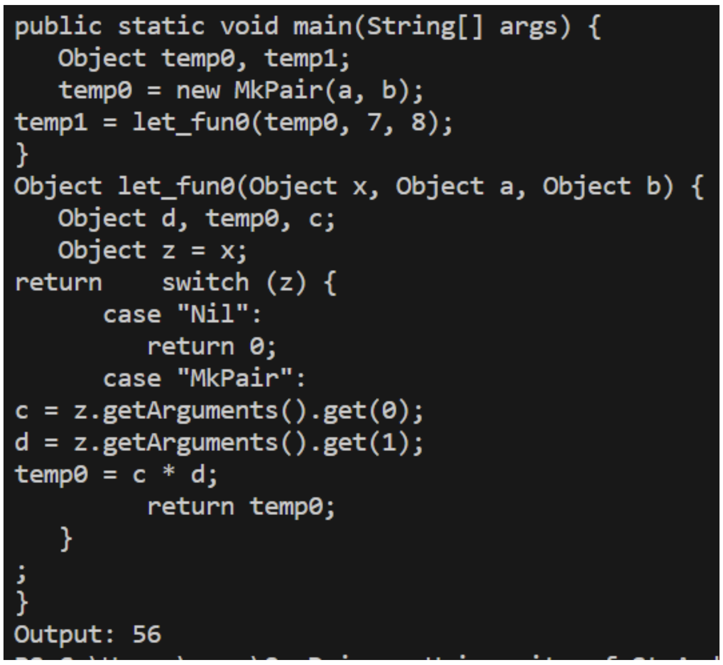


```
/* let_fun0(x, a, b) = let z = x
                       in case z of
                           Nil() -> 0
                           MkPair(c, d) -> c * a */
```



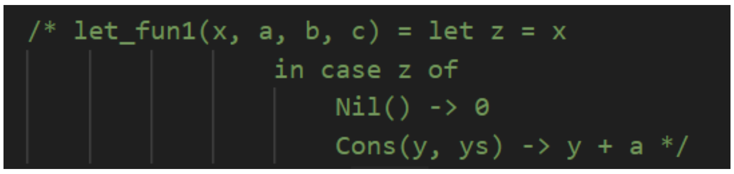
```
/* main = let_fun0(MkPair(a, b), 7, 8) */
```

  - Output:

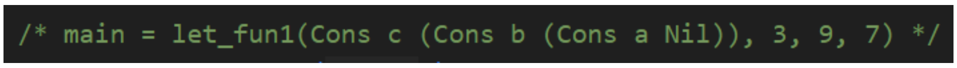


```
public static void main(String[] args) {
   Object temp0, temp1;
   temp0 = new MkPair(a, b);
temp1 = let_fun0(temp0, 7, 8);
}
Object let_fun0(Object x, Object a, Object b) {
   Object d, temp0, c;
   Object z = x;
return    switch (z) {
      case "Nil":
         return 0;
      case "MkPair":
c = z.getArguments().get(0);
d = z.getArguments().get(1);
temp0 = c * d;
         return temp0;
   }
;
}
Output: 56
```

- Function which contains a let statement which takes a list constructor and a case conditional.
  - Input:

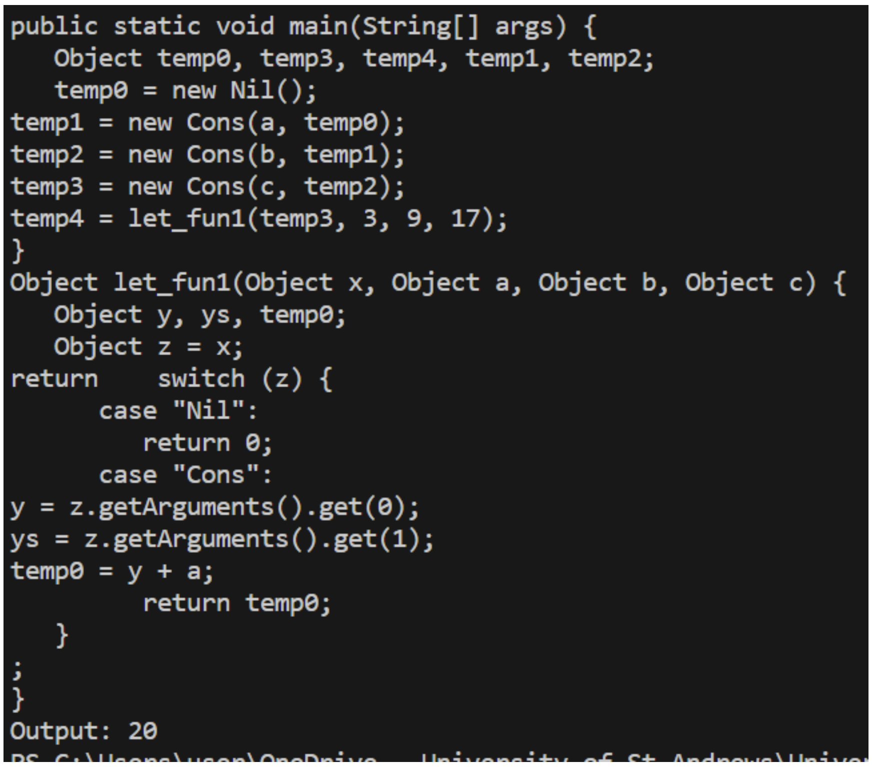


```
/* let_fun1(x, a, b, c) = let z = x
                          in case z of
                             Nil() -> 0
                             Cons(y, ys) -> y + a */
```



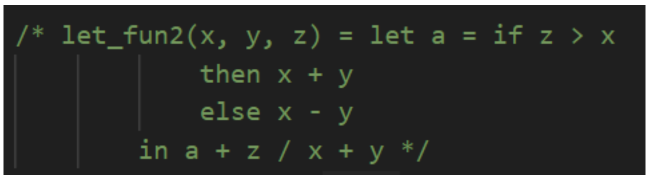
```
/* main = let_fun1(Cons c (Cons b (Cons a Nil)), 3, 9, 7) */
```

  - Output:

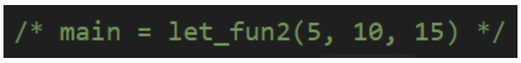


```
public static void main(String[] args) {
   Object temp0, temp3, temp4, temp1, temp2;
   temp0 = new Nil();
temp1 = new Cons(a, temp0);
temp2 = new Cons(b, temp1);
temp3 = new Cons(c, temp2);
temp4 = let_fun1(temp3, 3, 9, 17);
}
Object let_fun1(Object x, Object a, Object b, Object c) {
   Object y, ys, temp0;
   Object z = x;
return    switch (z) {
      case "Nil":
         return 0;
      case "Cons":
y = z.getArguments().get(0);
ys = z.getArguments().get(1);
temp0 = y + a;
         return temp0;
   }
;
}
Output: 20
PS C:\Users\user\OneDrive - University of St Andrews\Univer
```

- Function which has a let statement which takes an if conditional.
  - Input:

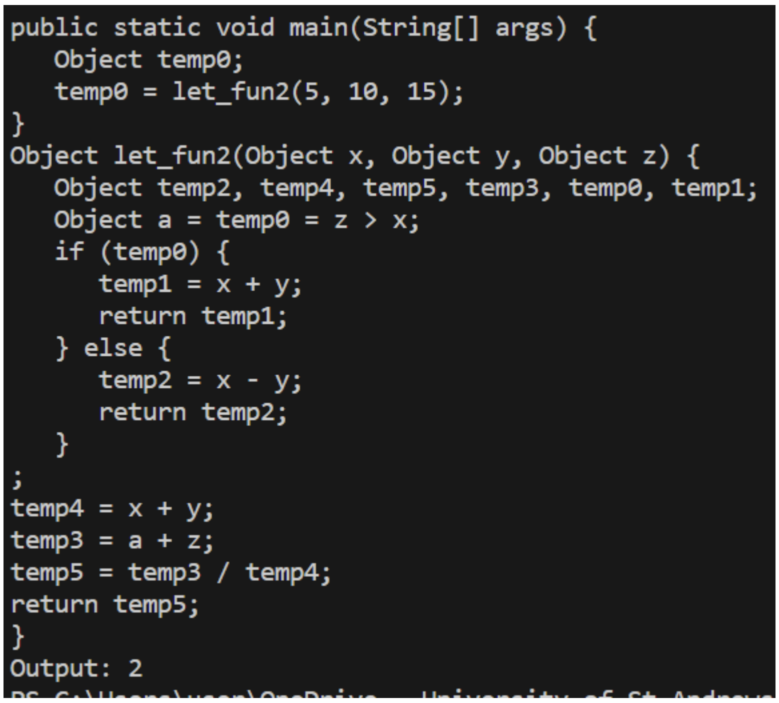


```
/* let_fun2(x, y, z) = let a = if z > x
                                 then x + y
                                 else x - y
                            in a + z / x + y */
```



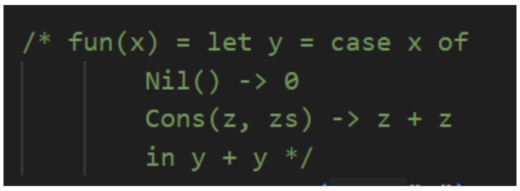
```
/* main = let_fun2(5, 10, 15) */
```

  - Output:

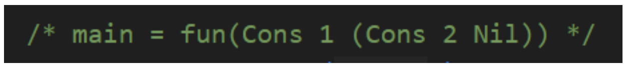


```
public static void main(String[] args) {
   Object temp0;
   temp0 = let_fun2(5, 10, 15);
}
Object let_fun2(Object x, Object y, Object z) {
   Object temp2, temp4, temp5, temp3, temp0, temp1;
   Object a = temp0 = z > x;
   if (temp0) {
      temp1 = x + y;
      return temp1;
   } else {
      temp2 = x - y;
      return temp2;
   }
;
temp4 = x + y;
temp3 = a + z;
temp5 = temp3 / temp4;
return temp5;
}
Output: 2
```

- Function which has a let statement which takes a case conditional.
  - Input:

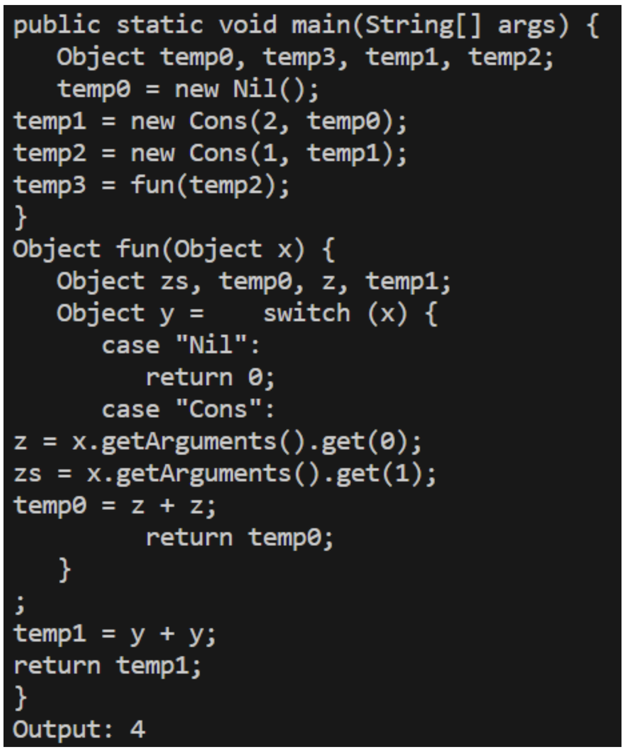


```
/* fun(x) = let y = case x of
            Nil() -> 0
            Cons(z, zs) -> z + z
            in y + y */
```



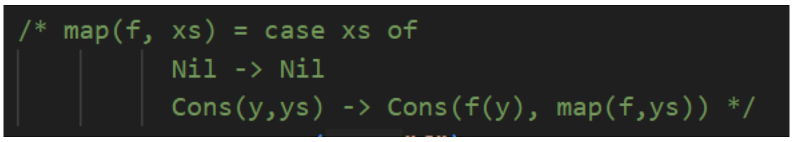
```
/* main = fun(Cons 1 (Cons 2 Nil)) */
```

  - Output:

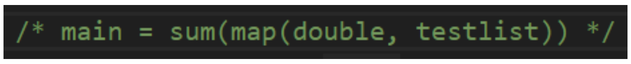


```
public static void main(String[] args) {
   Object temp0, temp3, temp1, temp2;
   temp0 = new Nil();
temp1 = new Cons(2, temp0);
temp2 = new Cons(1, temp1);
temp3 = fun(temp2);
}
Object fun(Object x) {
   Object zs, temp0, z, temp1;
   Object y =    switch (x) {
      case "Nil":
         return 0;
      case "Cons":
z = x.getArguments().get(0);
zs = x.getArguments().get(1);
temp0 = z + z;
         return temp0;
   }
;
temp1 = y + y;
return temp1;
}
Output: 4
```

- Function which contains a case conditional which uses first-order functions <u>(the output generated is incorrect and causes an error because this is part of defunctionalization which hasn't been implemented).</u>
  - Input:

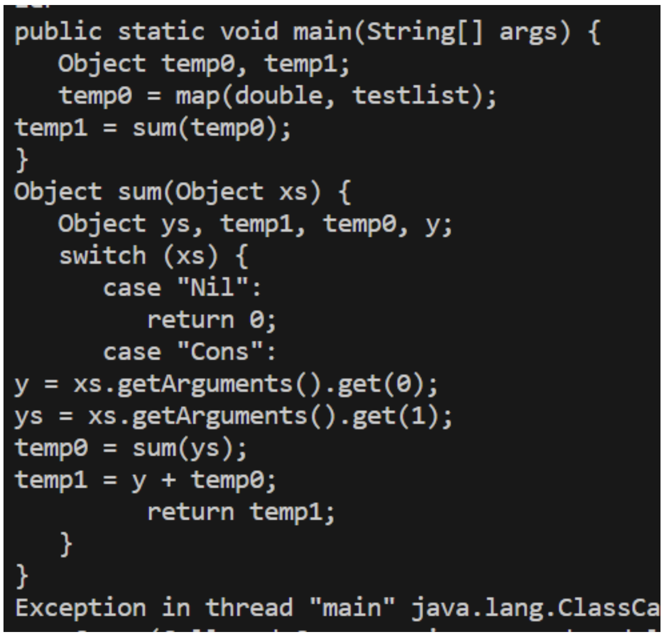


```
/* map(f, xs) = case xs of
          Nil -> Nil
          Cons(y,ys) -> Cons(f(y), map(f,ys)) */
```



```
/* main = sum(map(double, testlist)) */
```

  - Output:



```
public static void main(String[] args) {
   Object temp0, temp1;
   temp0 = map(double, testlist);
temp1 = sum(temp0);
}
Object sum(Object xs) {
   Object ys, temp1, temp0, y;
   switch (xs) {
      case "Nil":
         return 0;
      case "Cons":
y = xs.getArguments().get(0);
ys = xs.getArguments().get(1);
temp0 = sum(ys);
temp1 = y + temp0;
         return temp1;
   }
}
Exception in thread "main" java.lang.ClassCa
```

Evaluation and Conclusion

All the tests that I ran were passed successfully as the compiler generates the appropriate Java code and produces the correct output. It handles all types of expressions in the functional language and is therefore able to compile any given function. However, there is one issue with the generated Java code. If the body of a function contains a let statement which has a conditional as the bounded expression, the generated Java code for these cases is incorrect. The compiler simply assigns the conditional statement to the variable in the let statement which is syntactically incorrect. This is because *ANFConversion* converts all types of expression into ANF except for when expressions contain conditional statements. However, the evaluated output is still correct. Another issue which is minor is that fact that the indentations aren't being done correctly, however this is still syntactically correct since the generated code would still run. Lastly, as mentioned in the last test case above, whenever a function uses first-order functions, this produces an error since this is outside the scope of the compiler that I implemented.

In conclusion, the compiler that I implemented provides a simplified demonstration on how an actual compiler operates when handling intermediate representations. It involves converting an intermediate representation into a simpler intermediate representation so that all the unnecessary complexities that are produces in the generated AST are simplified, allowing an easier mapping of the source language to the target language. This is achieved in my implementation by converting the AST to ANF, which simplifies complex expressions into variables with represent simplified sub-expressions. Then the target code can be generated trivially since there is a one-to-one mapping of the language in ANF to Java code.