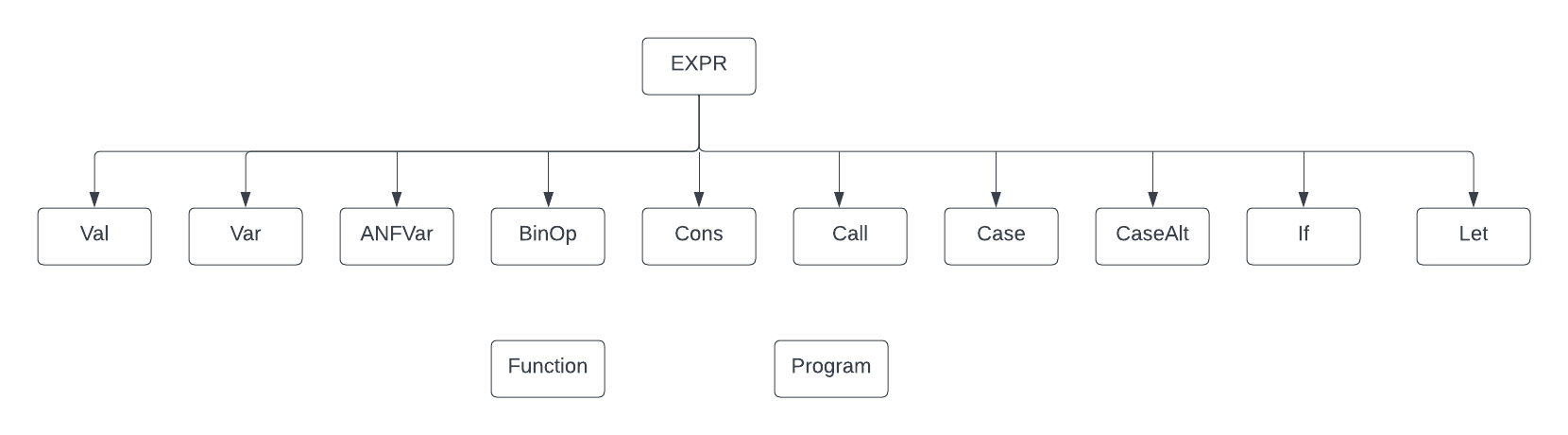
**P2 - Intermediate Representation**

**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:

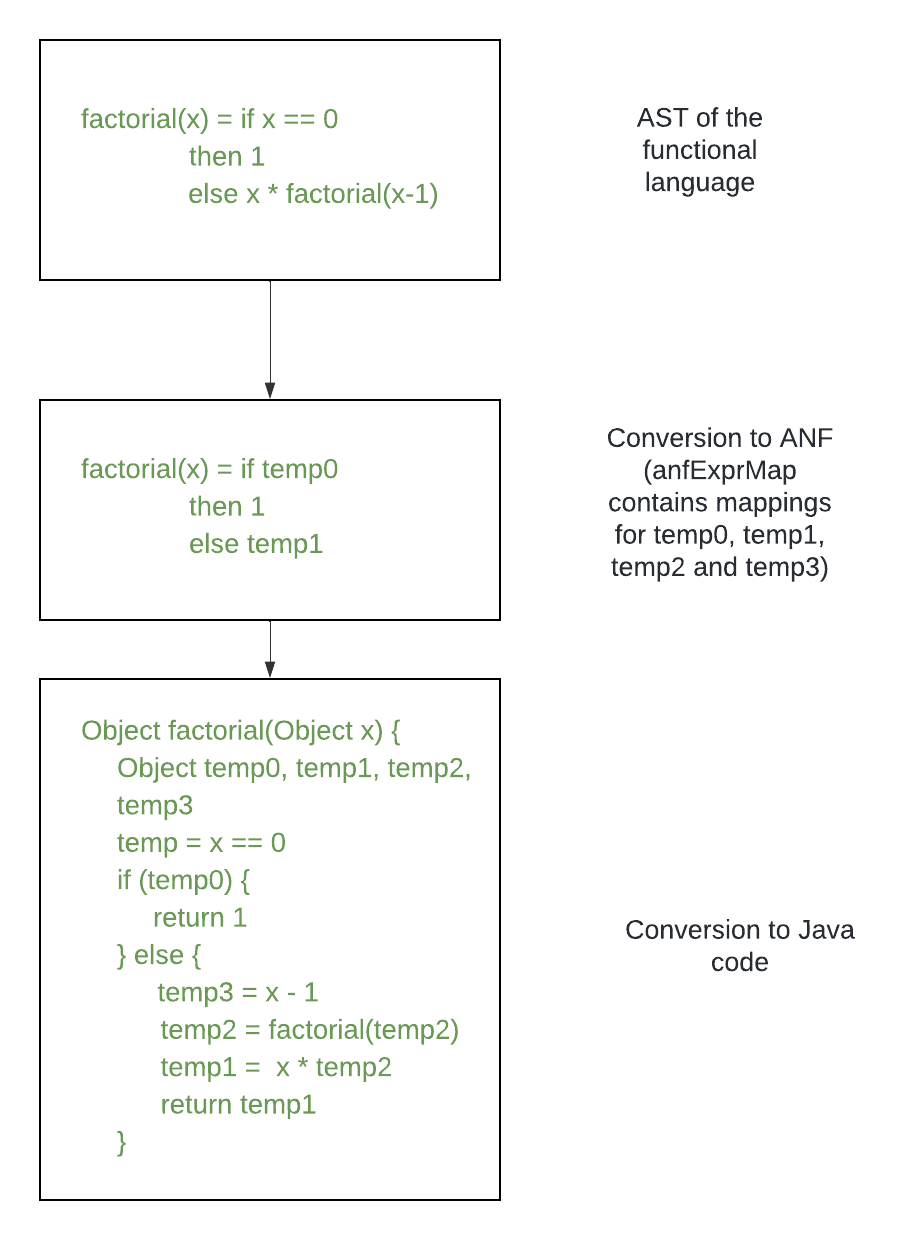


The main functionality of this compiler is encapsulated in two classes. The first is *ANFConversion* which converts the AST to ANF. A recursive function is defined to go through the given AST due to its recursive nature, and a switch conditional is used within this function to identify each type of expression 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:



**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:





* + Output:

A computer screen shot of white text

Description automatically generated

* Function which performs a simple arithmetic operation.
  + Input:





* + Output:

A computer screen with white text

Description automatically generated

* Function which performs a more complex arithmetic operation.
  + Input:





* + Output:

A computer screen shot of a program code

Description automatically generated

* Function which performs a more complex arithmetic operation.
  + Input:





* + Output:

A computer screen shot of a code

Description automatically generated

* Function which returns a list constructor.
  + Input:





* + Output:

A computer screen with white text

Description automatically generated

* Function which returns a pair constructor.
  + Input:





* + Output:

A computer screen with white text

Description automatically generated

* Function which returns a Nil constructor.
  + Input:





* + Output:

A computer screen with white text

Description automatically generated

* Factorial function
  + Input:

A black background with white text

Description automatically generated



* + Output:

A computer screen shot of a program code

Description automatically generated

* Function which has a nested if conditional
  + Input:

A screenshot of a computer screen

Description automatically generated



* + Output:

A screen shot of a computer program

Description automatically generated

* Function which has multiple nested if conditionals.
  + Input:

A black screen with green text

Description automatically generated



* + Output:

A computer screen shot of a program code

Description automatically generated

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

A screenshot of a computer program

Description automatically generated



* + Output:

A screen shot of a computer program

Description automatically generated

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

A black screen with white text

Description automatically generated



* + Output:

A screen shot of a computer program

Description automatically generated

* Function which contains a case conditional.
  + Input:

A black background with green and yellow text

Description automatically generated



* + Output:

A computer screen shot of white text

Description automatically generated

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

A black background with white text

Description automatically generated



* + Output:

A computer screen shot of a program code

Description automatically generated

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

A computer screen with text

Description automatically generated



* + Output:

A screen shot of a computer program

Description automatically generated

* Function which contains a case conditional nested within a case conditional.
  + Input:

A computer screen with green text

Description automatically generated



* + Output:

A computer screen shot of white text

Description automatically generated

* Function which contains three nested case conditionals.
  + Input:

A computer screen shot of a black background with green text

Description automatically generated



* + Output:

A screen shot of a computer program

Description automatically generated

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

A black background with green letters

Description automatically generated



* + Output:

A computer screen shot of a black screen

Description automatically generated

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

A black background with many symbols

Description automatically generated with medium confidence



* + Output:

A computer screen shot of a program code

Description automatically generated

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

A black background with white text

Description automatically generated



* + Output:

A computer screen shot of a black screen

Description automatically generated

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

A black background with green and white text

Description automatically generated



* + Output:

A computer screen shot of a black screen

Description automatically generated

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

A black background with white text

Description automatically generated



* + Output:

A computer screen shot of a black screen

Description automatically generated

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

A black background with white text

Description automatically generated



* + Output:

A computer screen shot of a program code

Description automatically generated

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

A black background with green letters

Description automatically generated



* + Output:

A computer screen shot of a program

Description automatically generated

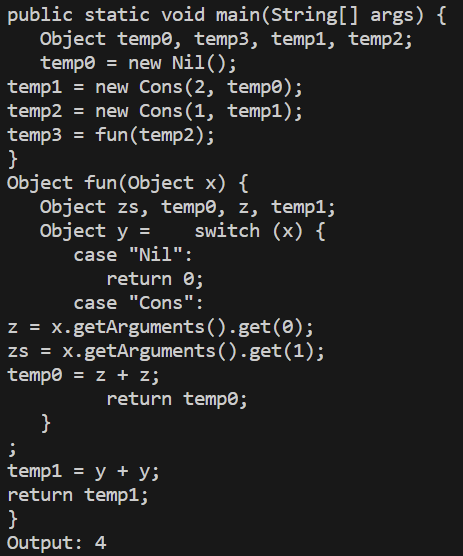
* Function which has a let statement which takes a case conditional.
  + Input:

A black background with white text

Description automatically generated



* + Output:



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

A black background with white text

Description automatically generated



* + Output:

A computer screen shot of a program code

Description automatically generated

**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.