Programming Language Design and Implementation Practical 1 : A Compiler to JVM Bytecode

150000845

October 2017

1 Introduction

The aim of this practical is to implement a compiler for abstract syntax trees of the Episcopal language defined in the specification into JVM bytecode.

2 Design

2.1 Abstract Syntax Tree Definitions

The structures defined with the abstract syntax trees will be modelled in the data structure format of the implementation language used. The values kept by these data structures will then define the details of what gets compiled. These abstract structures are defined below.

2.1.1 Program

This the highest level structure in an Episcopal program. It's important features are:

- Program ID : The name/identifier of the program.
- Expression : An expression for defining the top level abstract definition of the program.
- Set of Queries : The details that define aspects of the expression, such as constants.

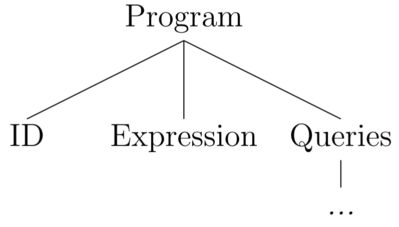

Figure 1: AST Representation of a Program

2.1.2 Query

A query structure is used to calls to functions or distributions with arguments to achieve some kind of output.

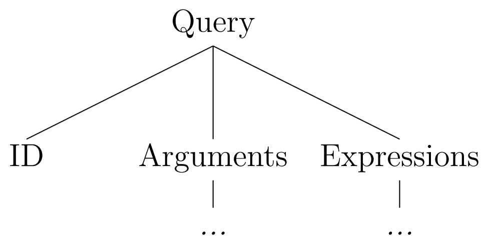

Figure 2: AST Representation of a Query

2.1.3 Expression

The expression takes a number of forms and is used for producing or assigning values. In the Episcopal language, they also are used for observing whether or not a set of expressions are valid and also for sampling a distribution.

A constant should return only the value that the constant expresses and nothing else.

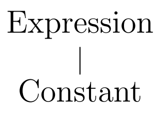

Figure 3: AST Representation of an Expression returning a Constant

Local definition expressions have 2 important parts : the list of definitions, followed by the sub expression which these definitions apply to.

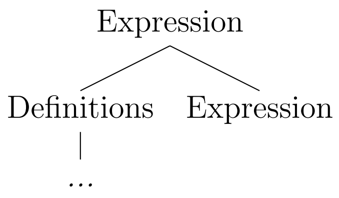

Figure 4: AST Representation of an Expression of local definitions

An observation statement is made up of two expressions. If the first expression is determined to be valid (true) then the result of this expressions is the result of the second expression. Otherwise, nothing is returned as the expression is deemed invalid.

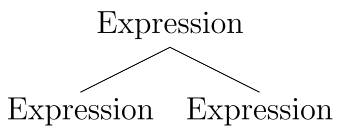

Figure 5: AST Representation of an Expression of an Observation

A sample call samples a distribution which is defined in the expression associated with the sample call.

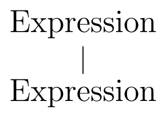

Figure 6: AST Representation of a Sample call

An expression that holds a distribution returns an instance of the defined distribution with the passed parameters so that it can be used, for example in sampling.

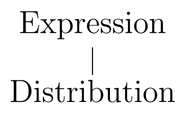

Figure 7: AST Representation of an Expression returning a Distribution

An expression with an identifier is likely to be used in the context of retrieving the variable value associated with the ID.

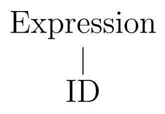

Figure 8: AST Representation of an Expression returning an ID

A call to a function requires the ID of the function to be called along with a set of arguments to pass to the function when called. These arguments are a list of expressions.

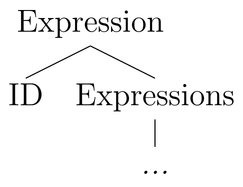

Figure 9: AST Representation of an Expression of a Function Call

Binary operations are also a type of expression to be evaluated - they apply the given operation to 2 expressions.

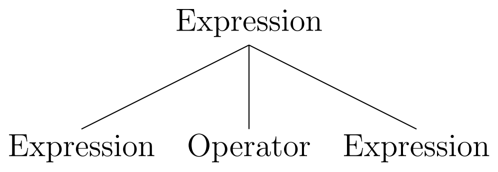

Figure 10: AST Representation of an Expression of a binary operation

The final type of an expression is an expression that has been bracketed. In the abstract form, this simply evaluates to the same expression without the brackets.

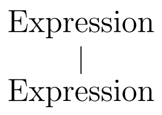

Figure 11: AST Representation of an Expression with brackets

2.1.4 Constants

A constant expression will only resolve to one thing, which is its associated value. As stated in the specification, a constant value can be any of an integer, a float, a boolean or a percentage.

2.1.5 Definitions

The definition construction is the mechanism for defining variables, functions and user defined distributions in the scope of another sub expression.

When no arguments are given in a definition, it can be treated as a variable, as what the variable can equate to is not going to dynamically change - the only possible way it can differ according to the Episcopal specification on different observations is if it is defined in terms of one or more distributions. Therefore, we shall say that a declaration with no arguments is called a variable declaration.

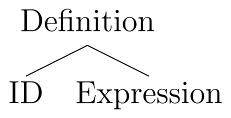

Figure 12: AST Representation of a Variable Declaration

When a definition has arguments, it can be treated as a function definition. It will be declared with its given ID and number of arguments, and will be defined with the expressions given to it.

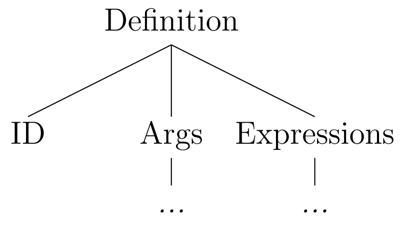

Figure 13: AST Representation of a Function Declaration

2.1.6 Args

There is a definition in the Episcopal specification for args for defining and calling functions. These are simply a list of IDs, which are just a type of string with no whitespace characters. Therefore, the AST will resolve to a list of strings.

2.1.7 Distribution

The set of distributions are defined in the grammar and can be used for sampling and observations in particular in an Episcopal program.

A Bernoulli distribution takes a value p and when sampled will return either 1 with probability p or 0 with probability $1 - p$. Therefore, the distribution only needs to be defined with one argument - its p value.

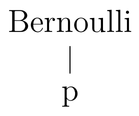

Figure 14: AST Representation of defining a Bernoulli Distribution

A Flip distribution is similar to Bernoulli - it too only takes one argument to define it, its probability p of returning True. It has a probability, therefore, of $1 - p$ for returning False.

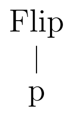

Figure 15: AST Representation of defining a Flip Distribution

A Beta distribution is a continuous distribution function which takes two values, α and β, that determine its probability density function and thus its sampling. Therefore, an instance needs only be constructed using 2 values.

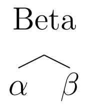

Figure 16: AST Representation of defining a Beta Distribution

A normal distribution is a continuous distribution defined in terms of its mean, μ, and its standard deviation, σ, to generate its probabilities.

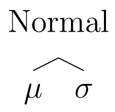

Figure 17: AST Representation of defining a Beta Distribution

All of these distributions can be defined in their given terms and used for sampling with only the values defined in the leaf nodes of their ASTs.

2.2 Design Decisions

2.2.1 Overall Structure

Since only AST representations are needed, there is no need for any kind of front end in the compiler. There will be a command handler to act as an entry point for the compiler and handle arguments for the compiler process. To obtain results from the compiler, a module will hold a collection of test ASTs to pass to the compiler which will each be output into files which Jasmin can convert into bytecode form. The result from compiling an AST will be a single output file to be compiled by Jasmin that defines a class, with the name of the class being the ID of the program at the highest level structure in the AST. This will always contain a main method so that it can be run on the JVM.

2.2.2 Variables

There are a few possible strategies for compiling variables. One option is to store every variable in the JVM heap, with the compiler keeping a track of variable ID to heap address. This would work, however most Episcopal programs are quite small, so it is unlikely that large data types that are expensive to replicate will be used, and since fetching from the heap is expensive, it does not make sense to store small number values on there to be mutated. A better solution will be for the compiler to keep track of the expression that the variable evaluates to and then replicating this expression whenever the variable is called. While this does mean that there will be code duplication where a variable is called excessively, it does also mean that a variable is only ever evaluated if it is called, as the compiler keeps track of the instructions that the variable is associated with, but if the variable is never called then it never needs to output these instructions. This is another advantage over the use of the heap - we may store the value and then never retrieve it if it is not used. Therefore, my strategy for variable storage and used will take these steps

- Read the variable and store its evaluated expression in a compile time map structure, with the ID of the variable acting as the key.
- When the variable is called, fetch the associated instructions from the compile time map structure and use those.

Since variable declarations are recursively defining expressions, it means that this structure also works well. When a variable is declared, the map is changed and sent to the child expression to be used. The child then may add its own definitions to pass on to further expressions and will add its own values, and so this defines how scoping can be done in this system.

2.2.3 Functions

We could also define functions in a similar fashion to how we have described variable definitions above - we could only use one main method for the entire Episcopal program and when a function is called, we fetch an associated set of instructions to use and feed in the arguments of the function to change the code. However, function definitions are usually longer and more expensive code snippets to store and reproduce compared to variables - if a function is called repeatedly, then it can be expensive to keep rewriting the structure, and it means that the compile time map could become large quickly too. This is a simple way of implementing function definitions, but as we can see it does not scale well. Therefore, a more difficult but effective way would be to create a new method for every function definition. When we come across a function definition in compiling, we can add the evaluated definition code to the back of the queue of instructions to be written. The difficult part of this system is that in compiling the Episcopal language, we may come across a complete function definition halfway through compiling another expression. If we simply added the method instructions to the end of the queue in this state, we would have half of the expression written out, followed by the method corresponding with the defined function, followed by the rest of the expression, and this problem only becomes more difficult as we nest more functions. However, I have decided that it will be a good challenge to solve and will make function definitions more efficient. It also has an advantage in that the main method stack will not grow too large, as tasks are distributed between different methods.

2.2.4 Types

With the above method of function definitions being used, types become important for function arguments and also their return type. Primitives are smaller compared to objects and are also quick to perform operations on. However, in the Episcopal system it is not easy to determine what type an argument should be for a function or even what type a function returns, which could be difficult when implementing the above system for functions. It may be better therefore for the data types themselves to have more responsibilities associated with them, as they will know more about their own type compared to what would be easy to deduce from the compiler. Objects have this notion, as they can call methods on themselves, which is easier than determining the type of some data before we can pass it to a function. This means also that we can group certain data types together better, such as program results, for which we need to make sure that both numbers and observation results can print their results. Therefore, in this case objects are preferable to primitives for simplicity's sake, even though with a good type inference system we could better use primitives for numbers for example.

At the start of implementation, floats will be used for constant values as every one of the Episcopal constant types can be modelled as a float and so this system can work well with no type inference. Therefore, each of the constant types will be implemented as float objects at the start, with the intention of having time to implement a good type inference system for determining the types for arguments and return values of methods so that more precise constant types can be used.

2.2.5 Distributions

Distributions can be modelled as objects. When a user defines a distribution with a set of arguments, a new instance of the distribution will be created. This also means that the distributions can be grouped together under an interface which defines what methods an interface should define, which will make implementing calls such as sampling a distribution easier as the compiler does not need to know which concrete class is being sampled. It also means that the distributions can override the print methods. It is more difficult to implement creating new objects than other methods, but they offer the above advantages. Therefore, my implementation strategy will be to create an IDistribution interface for the distributions to implement. These will be written in Java and then called as a library at first, as we need to make sure that the method calls are correctly implemented, and having a proof of concept Java compiled library will assist with this.

2.3 Observations

Observations should output whether or not a program is valid and also the result if it is valid. Since we will not always know at compile time what the observation validation expression will evaluate to, for example if it is defined in terms of a distribution, and so therefore we need to have a branching mechanism in the code to carry out different actions depending on the outcome of the validity check. A separate results class should also be used, which should contain a boolean value corresponding to whether or not the expression was valid, and an extra float value which is the value that the executed expression returns if the expression was valid.

2.3.1 Compilation Process

A compile function can be modelled in terms of an expression that needs to be compiled and the environment from which its being compiled in. This environment can include certain variables, such as variables defined further up the scope or the name of the class. Each expression should either evaluate to a set of instructions to be compiled or mutate the compile time map of variables. However, not only will compiling an expression return what it evaluates too but also what expressions that come after it evaluate to, such as further function definitions, and the order of these will need to be kept track of as described in the discussion about functions.

3 Implementation

I have implemented the project in Haskell, with all systems laid out in the design section being implemented. I chose Haskell as its pattern matching makes expressing and handling ASTs clear.

3.1 Build and Run

In the project, all of the source code has been kept in the src directory. It is a Cabal project but I have added an extra script to be run to carry out extra tasks automatically. The script in the project will:

- Build the compiler.
- Run the compiler with the test ASTs.
- Output the result of compiling the ASTs.
- Compile these output files with Jasmin into class files.
- Test each class file with its expected output, showing the test being carried out (i.e its file name and the Episcopal code that the AST being tested translates to.)

The results of the tests and build will be outputted as they are being executed. To run this script, run

```
$ ./run.sh
```

in the project directory. Code outputted from the compiler will be put in the directory *output_progs* which can be viewed to see how the AST compiled.

4 Testing

Results from the testing can be found in the file called tests located in the project root directory. The tests can also be replicated by following the build and run method described in the implementation section.

5 Conclusion

I enjoyed this practical a lot - I felt that I gained large amounts of knowledge on both designing and implementing compilers and what should be considered in doing so. This practical also gave me a better understanding of the architecture of the Java virtual machine. It was challenging, but a very good experience.
To improve on it further if I had time, I would have looked at implementing a system of type inference by extending the return values from expression evaluation to include the type of the evaluated expression and by also storing the type of a variable in the map. I would also have liked to extend operations on distributions. This can be done by adding additional properties to each distribution class, such as multiplier and adder, which are applied to the value of a sampling of a distribution before it is returned.