

HamSkill: Run Haskell Anywhere
with ANTLR and Scala

CS252 Project Final Report

Zayd Hammoudeh
(zayd.hammoudeh@sjsu.edu)

April 2, 2016

Contents

List of Figures	iv
1 Running in the Java Virtual Machine	1
2 Key Project Requirements	1
3 <i>HamSkill</i> 's Software Architecture	2
3.1 ANTLR	2
3.1.1 ANTLR Version 4 Grammar	2
3.1.1.1 <i>HamSkill</i> 's Haskell Grammar	3
3.1.1.2 <i>HamSkill</i> 's ScalaOutput Grammar	4
3.1.2 ANTLR-Generated Java Classes	5
3.1.2.1 HaskellTokensToScala Java Class	5
3.1.2.2 ScalaOutputTokensToHaskellFormat Java Class	5
3.2 Transpiler Output Language: Scala	6
3.2.0.1 <i>HamSkill</i> Standard	7
3.2.0.2 <i>HamSkill</i> +	8
4 Overview of <i>HamSkill</i> 's Test Strategy and Architecture	9
4.1 System Level and Regression Testing	9

4.2	Test Bench Implementation Overview	10
4.3	Test Cases	10
5	Haskell Features Supported by <i>HamSkill</i>	11
5.1	Single Haskell File	11
5.2	Support for Functions	11
5.2.1	Explicit Function Type Signature	11
5.2.2	Pattern Matching	12
5.2.3	Recursion	12
5.2.4	A Single Executable Expression per Pattern Matching Statement	13
5.2.5	Empty Line After the Last Function	13
5.3	Haskell's <code>main</code> Function	13
5.4	Currying	14
5.5	Calling a Function	15
5.5.1	Using the <code>\$</code> Operator	15
5.5.2	Surround Each Function Parameter in Parentheses	16
5.5.3	Triple Parentheses Notation	18
5.6	Higher-Order Function Support	19
5.7	Conversion from Haskell Functions to Scala Object Methods	19
5.8	Partially Applied Functions	20
5.9	Converting a Haskell Lambda Function to a Scala Anonymous Function	20
5.10	Supported Types	21
5.11	Lazy Evaluation and Immutability	22

5.12	Defining Scope via Haskell’s <code>module</code> Keyword	22
5.13	<code>Maybe</code> Monad Support	22
5.14	<code>if</code> Conditionals	23
5.15	<code>case</code> Statements	24
5.16	Preserving Comments	25
6	Future Work and Improvements	26
7	Conclusions	27

List of Figures

1	<i>HamSkill</i> Project Architecture	3
2	Printing a Three Element List in Haskell	4
3	Printing a Three Element List in Scala	4
4	Tranpiling from Haskell to Scala with <i>HamSkill+</i>	8
5	Compiling the Transpiled Scala Code in <i>HamSkill+</i>	8
6	Executing the Scala Code and Piping the Output to <i>HamSkill</i> Java Class “ ScalaOutput ”	9
7	Simple <i>HamSkill</i> Function	11
8	<i>HamSkill</i> Pattern Matching Example	12
9	A Recursive Call to a Haskell Function that Sums Together All Integers in a List	12
10	Haskell Function that Takes and Prints Console Inputs	14
11	<i>HamSkill</i> Generated Scala Code to take and Print Console Inputs	14
12	<i>HamSkill</i> Generated Scala Code for Function “ myFunc ” with No Currying	15
13	<i>HamSkill</i> Generated Scala Code for Function “ myFunc ” with Currying	15
14	Specifying Haskell Function Parameters using “ \$ ” in Haskell . . .	16

15	<i>HamSkill</i> Transpiled Scala Code of Functions using “\$” to Specify Function Parameters	16
16	Haskell Code where Parentheses around Arguments are Used to Specify Function Arguments	17
17	Transpiled <i>HamSkill</i> Code for Function Parameters Each with their Own Parentheses	17
18	Haskell Code where a Function’s Parameters are Specified Using Triple Parentheses Notation	18
19	<i>HamSkill</i> Transpiled Code for the factorial Function with Triple Parentheses Function Call Notation	18
20	<i>HamSkill</i> Supported Implementation of Haskell’s filter Function	19
21	<i>HamSkill</i> Supported Implementation of Haskell’s map Function .	19
22	Partially Applied Haskell Function addTwoPlusOne	20
23	Scala Version of Haskell Function “ addTwoPlusOne ” Generated by <i>HamSkill</i>	20
24	A Haskell Lambda Function Supported by <i>HamSkill</i>	21
25	A Scala Anonymous Function Generated by <i>HamSkill</i>	21
26	<i>HamSkill</i> Parsable “ module ” Statement	22
27	Haskell Maybe Code Supported by <i>HamSkill</i>	23
28	<i>HamSkill</i> Generated Scala Option Monad Code	23
29	Haskell “ if ” Statement Supported by <i>HamSkill</i>	24
30	Scala Transpiled “ if ” Statement Generated by <i>HamSkill</i>	24
31	Haskell case Statement	25
32	Scala Transpiled “ case ” Statement Generated by <i>HamSkill</i> . . .	25
33	Haskell Header Comment	26
34	Scala Transpiled Header Comment	26

1 Running in the Java Virtual Machine

C is one of the most commonly used languages when the primary object is maximum performance. However, C’s “write once, compile anywhere” paradigm limits its portability. In contrast, the near ubiquity of the Java Virtual Machine (JVM) allows Java to be “write once, run anywhere.”

Developers have often leveraged the JVM’s “run anywhere” capability to run languages other than Java. Examples include: JRuby for the Ruby programming language [1], Jython for the Python programming language [2], Renjin for the R programming language [3], and Scala [4].

Currently, there is no full implementation of Haskell in the JVM. One Haskell dialect that is runnable in Java is Frege [11].

This project implements, *HamSkill*, which is a transpiler from Haskell to Scala; *HamSkill* enables a dialect of Haskell to run in the JVM.

2 Key Project Requirements

When designing and implementing this project, there were four primary goals:

1. **Runnable in the Java Virtual Machine** - As explained in section 1, Java’s Virtual Machine enables significant machine independence, which Haskell does not currently have.
2. **Minimal JVM Requirements** - In addition to just running in the JVM, *HamSkill* was created to be as standalone as possible. As an example, it was not expected that in most application the user would have Scala installed on their machine. To achieve this maximum portability, some advanced features may not be supported.
3. **Identical Input and Output Between Haskell and *HamSkill*** - In many scenarios, it may not be sufficient for a Haskell program to run inside the JVM. Rather, it is more likely that the output generated by the two environments will need to be identical. As such, *HamSkill* includes an additional post-processing step to ensure its output is identical to that of Haskell.
4. **Human Readable Output Code** - A transpiler is any program that takes source code from one programming language and outputs code in another programming language with a similar level of abstraction [8]. To

enable increased reuse of the outputted code, *HamSkill* uses techniques such as indenting, newlines, etc. to maximize the readability of the generated output. While this is not a necessary requirement for the complete system to work properly, it enhances the tool’s potential.

3 *HamSkill*’s Software Architecture

HamSkill is a transpiler that takes Haskell code as an input, converts it to Scala, and then runs the transpiled code in the JVM. The *HamSkill* implementation consists of six major components. They are:

- ANTLR Lexer and Parser
- Haskell Antlr Grammar
- ScalaOutput Antlr Grammar
- Scala Runtime Environment
- HamSkillMain Java Class
- ScalaOutput Java Class

The relationships between these components are shown in figure 1, and the following subsections describe the role of each component in the overall system.

3.1 ANTLR

ANTLR (Another Tool for Language Recognition) is an adaptive left-to-right, left-most derivation (LL(*)) lexer and parser written in the Java programming language. ANTLR’s primary function is to read, parse, and process structured text (e.g. Haskell code) [10].

3.1.1 ANTLR Version 4 Grammar

A grammar is a formal description of language; it is based off the concept of a context-free grammar in automata theory. ANTLR version 4’s (v4) grammar files (denoted by the file extension “.g4”) explicitly define how ANTLR will parse the structured text. The grammar file may contain token definitions (which always start with a capital letter) and/or parser rules (which always

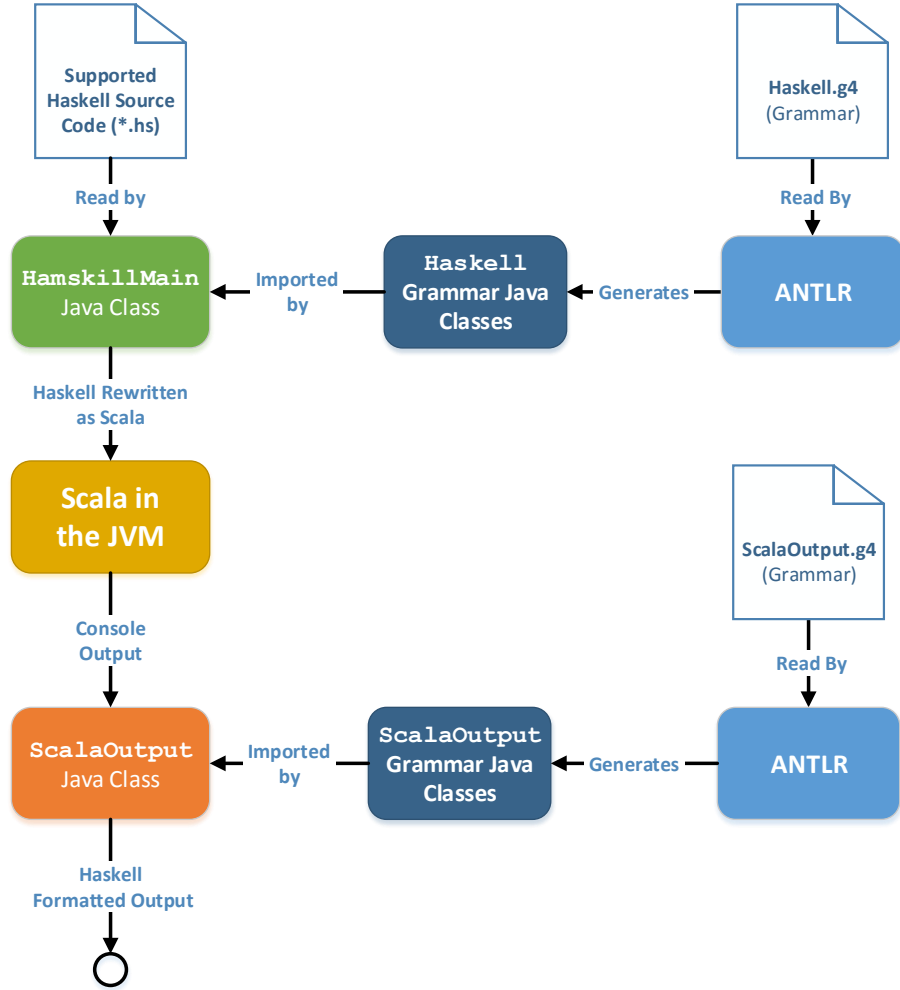


Figure 1: *HamSkill* Project Architecture

start with a lowercase letter). A token is a group of characters that form a single object; the parser uses these tokens to recognize sentence structures within a corpus.

This project utilizes two separate grammars namely: *Haskell* and *ScalaOutput*. These grammars are described in the following two subsections.

3.1.1.1 *HamSkill*'s Haskell Grammar

The ANTLR grammar, `Haskell`, is contained in the file “`Haskell.g4`.” This grammar defines both the tokens and abstract syntax tree for supported Haskell code. Some of the Haskell language features that are supported by this grammar include:

- `case` Statements
- `if`, `then`, `else` Conditionals
- `Maybe` Monads
- Currying
- Partially Applied Functions
- Higher Order Functions
- Lazy Evaluation

For the complete feature set as well as any Haskell syntax requirements, see section 5.

3.1.1.2 *HamSkill*’s `ScalaOutput` Grammar

As mentioned in section 2, one of the key requirements of this project was that the outputs from Haskell and *HamSkill* be identical. Figure 2 and 3 show the Haskell and Scala code respectively to print to the console a list containing integers “2”, “3”, and “4”. Note that the second line in each figure (which show each language’s respective console output) are very different.

```
Prelude> putStrLn $ show [2, 3, 4]
[2,3,4]
```

Figure 2: Printing a Three Element List in Haskell

```
scala> println(List(2,3,4))
List(2, 3, 4)
```

Figure 3: Printing a Three Element List in Scala

The ANTLR grammar, `ScalaOutput`, (contained in the file “`ScalaOutput.g4`”) is used to account for these types of differences by parsing the console output of a Scala program and converting that output to a syntax that is more similar to that of Haskell.

3.1.2 ANTLR-Generated Java Classes

Programs that use ANTLR do not generally operate directly on the grammar. Rather, the grammar is converted by ANTLR into a set of Java classes; given an grammar (e.g. “`GrammarName.g4`”), ANTLR creates the following files:

- `GrammarNameLexer.java` - This class is the the lexer defined by the input grammar file. It extends ANTLR’s base `Lexer` class.
- `GrammarNameParser.java` - Each rule in the original grammar constitutes a method in this class; it forms the parser class definition associated with the input grammar file.
- `GrammarName.tokens` - Assigns a token type (e.g. integer, identifier, floating point number, etc.) to each token in the input grammar file.
- `GrammarNameListener.java` & `GrammarNameBaseListener.java` - ANTLR applies the grammar to a text input to build an Abstract Syntax Tree (AST). While walking the tree, ANTLR fires events that can be captured by a listener in these two classes[10].

Most programs that want to use ANTLR override the functions in the file `GrammarNameBaseListener.java`.

3.1.2.1 HaskellTokensToScala Java Class

The class “`HaskellTokensToScala`” overrides some of the methods ANTLR auto-generated in the file “`HaskellBaseListener.java`”; note that all of these overridden methods are derived from the grammar “`Haskell.g4`” described in section 3.1.1.1. In the end, it is the Java code in this file that performs the actual, low-level transpilation from Haskell to Scala.

3.1.2.2 ScalaOutputTokensToHaskellFormat Java Class

Similar to the `HaskellTokensToScala` class, the class, “`ScalaOutput`” extends the ANTLR auto-generated class, “`ScalaOutputBaseListener`”.

The listeners in this class are primarily responsible for transforming two different categories of Scala outputs, specifically:

- **Lists** - As shown in figures 2 and 3, the console output of lists in Scala and Haskell are significantly different. The methods in this class coupled with the `ScalaOutput` grammar convert printed lists from Scala to Haskell format.
- **Maybe Monad** - Scala’s approximate equivalent of Haskell’s `Maybe` monad is named `Option`. While the syntax for the two are similar, they are not identical. For example, “`Just`” and “`Nothing`” in Haskell are referred to as “`Some`” and “`None`” in Scala. The monad naming convention conversion for console outputs from Scala back to Haskell is handled by this class.

Any other Scala console outputs that do not fall into these two categories are passed through unchanged.

3.2 Transpiler Output Language: Scala

As defined previously, a transpiler takes source code from one programming language and converts it to code in another language. It was also previously explained that the primary criteria when deciding on *HamSkill*’s output language was that it needed to be runnable in the Java Virtual Machine. Other important criteria that guided language selection were:

- **Higher-Order Function Support** - Any language that is devoid of higher-order function support would be a poor match for a purely functional language like Haskell.
- **Similar Syntactic Structure** - Closer alignment between the input and output languages simplifies the transpilation. Inevitably though, some degree of restructuring and reformatting is required.
- **Personal Interest** - Increased personal interest in a project topic generally leads to a more fulfilling outcome for the student and usually a better project overall.

The language that best fit these criteria is Scala, which is a functional language that is run inside the JVM [7]. What is more, Scala natively supports

many of Haskell’s core features including: functions written in a pattern matching style, immutability of objects, currying, partially applied functions, lazy evaluation, static typing, etc. One of the disadvantages of Scala is its weaker type inference in comparison to Haskell; due to this, specific requirements were placed on the Haskell dialect supported by *HamSkill* as described in section 5. It must be noted that the author has a predisposed interest towards learning Scala given its extensive support by Apache Spark. Given all of these factors, the selection of Scala for this project became an obvious choice.¹

HamSkill implements two different schemes for running the transpiled Scala code, namely *HamSkill* Standard and *HamSkill*+. They are described in the following subsections.

3.2.0.1 *HamSkill* Standard

Since Scala is a compiled language, it does not lend itself tremendously well to Java runtime compilation and execution. To overcome this challenge, *HamSkill* leverages Twitter’s `util-eval` library, which allows runtime compilation and execution of Scala code entirely within a Java program [5]. This approach is known as “*HamSkill* Standard”; the advantages and disadvantages of using the `util-eval` library include:

- **Advantages:**

1. **Reduced JVM Requirements** - Twitter’s `util-eval` library is a JAR file that only requires the presence of Scala’s compile and main library JAR files to run. Hence, this eliminates the need for the user to have Scala installed in their environment. This addresses one of the key requirements enumerated in section 2.
2. **Simplified Usage** - When running *HamSkill* Standard, only a single function call is required to transpile, compile, then run the Scala code. In contrast, *HamSkill*+ at least four steps to do the same task, which inevitably introduces additional possible points of error.

- **Disadvantages:**

1. **Reduced Feature Set** - The `util-eval` library does not support all Scala features. An example of a missing feature is the function “`readLine`”, which takes console inputs.

¹Additional recognition to Dr. Thomas Austin is required for pointing the author in the direction of Scala.

2. **Reliance on Third Party Developers** - As mentioned previously, `util-eval` was developed by Twitter. Hence, it can be deprecated at any time (as Twitter had previously planned [6]). While the library is open-source, picking up a dropped project introduces an additional set of complications.

3.2.0.2 *HamSkill+*

Unlike *HamSkill* Standard, *HamSkill+* requires Scala to be installed on the host PC. What this enables is that any feature that is supported by the installed version of Scala could in theory be utilized by *HamSkill+*. Currently, running a Haskell program with *HamSkill+* requires four separate steps, all of which are controlled by a `bash` script as shown in figures 4, 5, and 6; the steps are:

1. **Transpile the Haskell code to Scala** - This is done by running the `main` method in the `HamSkillMain` class as shown in figure 4. Note that “`“$@”`” is `bash` notation that entails that any parameters passed to the `bash` script are directly passed as arguments into the JAR’s `main` method. In *HamSkill+*, the last argument passed to the JAR is the name of the file where the Scala code will be written.

```
java -jar hamskill.jar “$@”
```

Figure 4: Tranpiling from Haskell to Scala with *HamSkill+*

2. **Compile the Transpiled Scala Code** - The `bash` command to compile Scala is “`scalac`” as shown in figure 5. Note that “`$scalaSrcFile`” is the filename where the transpiled Scala code was written in the previous step.

```
scalac $scalaSrcFile
```

Figure 5: Compiling the Transpiled Scala Code in *HamSkill+*

3. **Execute the Transpiled Scala Code** - The `bash` command to run compiled Scala code is simply “`scala`” as shown in figure 6. Note that “`$scalaObject`” is the name of the Scala object (similar to a class in Java) in the compiled Scala file.

```
scala -cp . $scalaObject  
      | java -cp hamskill.jar hamskill.ScalaOutput
```

Figure 6: Executing the Scala Code and Piping the Output to *HamSkill* Java Class “ScalaOutput”

4. **Pipe the Scala Output to the ScalaOutput Class** - The console output of the Scala code is piped (via “|”) into the Java class `ScalaOutput` as shown in figure 6. As explained in section 3.1.2.2, this class performs any necessary transformations on the console outputs and then prints the transformed output to the console as if the original code had been run in Haskell.

4 Overview of *HamSkill*’s Test Strategy and Architecture

Software testing detects defects in software programs; there are different types of software testing including: unit testing, module level testing, and system testing. Given that ANTLR relies heavily on the Listener pattern, it reduces the ease at which unit testing can be performed. What is more, given that *HamSkill* relies on the end to end operation of programs written in Haskell, ANTLR, Java, and Scala, it lends itself to a system-level testing methodology.

4.1 System Level and Regression Testing

HamSkill’s uses black-box system level testing. A set of use cases (i.e. Haskell files) were generated by me; these files are then run through both Haskell (to verify they are valid Haskell code) and through *HamSkill*; the resulting outputs are then compared. If the outputs are the same, the test is considered successful; in the case of any differences, the test is classified as a failure. The full set of *HamSkill* test cases are included with this submission in the folder `Test_Bench/test_cases`.

Regression testing verifies that new features do not affect/compromise existing ones. When developing *HamSkill*, new features were iteratively added. After each feature was added, the entire test bench was run to ensure that no new (detectable) issues were introduced. If an issue did arise, it would be immediately addressed before adding new features to ensure that fundamental

problems did not compound upon themselves. This formed a type of regression testing throughout *HamSkill*'s development.

4.2 Test Bench Implementation Overview

The *HamSkill* test architecture is written as a `bash` script (see the file “`Test_Bench/test_bench.sh`” included with the submission). The function “`perform_hamskillStd_and_hamskillPlus_test`” is the most important in the script as it performs most of the testing; the basic operational flow of this function is:

- **Step #1:** Provide the `bash` function with the name of a Haskell program written in the dialect supported by *HamSkill*, but without the file extension.
- **Step #2:** Run the specified Haskell program through Haskell and store the program's output to a specific file on disk using the `bash` command “`>`”.
- **Step #3:** Run *HamSkill* Standard and output the results to a file using the `bash` command “`>`”.
- **Step #4:** Perform a file difference operation (using the `bash` “`diff`” command) on the output files from Haskell and *HamSkill* Standard. If the two files are identical, then mark the test as passing; if they are different, the test failed.
- **Step #5:** Repeat steps #3 and #4 using *HamSkill+*.

There is an additional function in the `bash` test bench named “`print_final_results`” that prints the final results number of passing tests versus the total number of tests.

4.3 Test Cases

From the perspective of the test bench, a single Haskell file being run through either *HamSkill* Standard or *HamSkill+* represents a single test case. In section 5, the test bench file used to verify each *HamSkill* feature is documented for reference.

5 Haskell Features Supported by *HamSkill*

This section enumerates the Haskell features that are supported by *HamSkill*. It also details any specific formatting requirements for the *HamSkill* Haskell dialect.

5.1 Single Haskell File

Currently, *HamSkill* only supports a single Haskell file at a time. If the user wants to run code across multiple files, s/he could transpile the code to Scala, manually configure the imports, and then run the Scala code manually. While this requirement may be a bit onerous, it should not limit the tool’s overall capabilities in a meaningful way.

5.2 Support for Functions

Functions written in Haskell can be interpreted by *HamSkill*. Figure 7 is a function from the *HamSkill* test case “`simple_function_call.hs`”.

```
myFunc :: [Int] -> Int -> Int
myFunc x y = 3 + 5
```

Figure 7: Simple *HamSkill* Function

The following subsections enumerate the requirements a Haskell function must satisfy to be supported by *HamSkill*.

5.2.1 Explicit Function Type Signature

All functions (excluding partially applied functions as described in section 5.8) must have an explicit type signature. Note that only a subset of Haskell’s base classes are supported (see section 5.10 for details on the supported types).

Currently, type variables (even if they map to a supported type) are not supported.

5.2.2 Pattern Matching

As shown in figure 7, pattern matching of function variables is supported. Special pattern matching cases that *HamSkill* handles include:

- Ignored variables via the Underscore (`_`) Notation
- Prepend using Colon (`:`)
- Empty Lists (`[]`)

Figure 8 is a function from the *HamSkill* test case “`partially_applied_example.hs`” that uses all three of the previously mentioned special pattern matching cases.

```
zayd_foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int
zayd_foldr _ acc [] = acc
zayd_foldr f acc (x:xs) = f (x) (zayd_foldr (f) (acc) (xs))
```

Figure 8: *HamSkill* Pattern Matching Example

Guards are not currently supported.

5.2.3 Recursion

Recursion is supported by *HamSkill*; however, there are requirements on how function parameters must be specified (see in section 5.5).

Figures 8 and 9 are recursive functions supported by *HamSkill*; note that the function in figure 9 is in the *HamSkill* test case “`addList.hs`”.

```
addList :: [Int] -> Int
addList [] = 0
addList (x:xs) = x + ((addList xs))
```

Figure 9: A Recursive Call to a Haskell Function that Sums Together All Integers in a List

5.2.4 A Single Executable Expression per Pattern Matching Statement

For each pattern matching case, only a single expression is allowed; Haskell's **where** and **let** syntaxes are not supported. As an example, the first pattern matching case in figure 8 simply returns “acc” while the second pattern matching case returns the result of calling the function “f” on two parameters (with the second parameter being a recursive function call).

If a user wants to define subvariables or execute multiple commands for a single pattern matching case, these additional steps must be defined as separate functions. While this requirement may make a developer's code more verbose, it should not limit the capabilities of the user.

5.2.5 Empty Line After the Last Function

After the last function in the file, there must be at least one empty line. This requirement was included because it simplifies the parsing process.

5.3 Haskell's main Function

Every Haskell file that is run in either *HamSkill* Standard or *HamSkill+* must have a **main** function. This function is sole entry point into a file. As with all standard functions, the **main** method must have its explicit type signature.

Unlike most other functions, multiple distinct statements may appear in the **main** function; these statements may even use the syntax “<-” as long as they are being used to unbox a Haskell IO **String** object.

Figure 10 is a recursive, multi-instruction **main** function that supports taking text inputs from the console, and then prints those inputs to the screen. This figure also shows the special syntax required when recursing on **main** where the function name must be surrounded by triple parentheses (i.e. “(((main)))”). This was done to simplify the parsing to let *HamSkill* know that this is a recursive call and that when converting this code to Scala, it must pass it an argument as shown in figure 11; **main** can only be called (recursively or otherwise) within the **main** method itself.

```

main :: IO ()
main = do
  x <- getLine
  putStrLn $ x
  if (length x == 0)
  then
    (return () )
  else
    ( (((main))) )

```

Figure 10: Haskell Function that Takes and Prints Console Inputs

```

def main(args : Array[String]){
  lazy val x = scala.io.StdIn.readLine();
  println (x)
  if ((x).length() == 0)
  {
    return;
  }
  else{
    main(args)
  }
}

```

Figure 11: *HamSkill* Generated Scala Code to take and Print Console Inputs

Note that the code shown in figure 11 is only supported in *HamSkill+*.

5.4 Currying

Haskell is a fully curried language [9]. In contrast, a Scala function’s type signature must be specially formatted to support native currying. Figure 12 shows Scala code generated by *HamSkill* with currying disabled (note the Haskell version of “myFunc” is shown figure 7); in contrast, figure 13 shows the *HamSkill* generated Scala code with currying enabled. Notice that when currying is enabled, each of the function parameter **must** be inside its own set of parentheses.

```

def myFunc( ___0___ : => List[Int], ___1___ : => Int) :
    Int = (___0___, ___1___) match {

    case (x, y) => 3 + 5

    }

```

Figure 12: *HamSkill* Generated Scala Code for Function “myFunc” with **No** Currying

```

def myFunc ( ___0___ : List[Int]) ( ___1___ : Int) :
    Int = (___0___, ___1___) match {

    case (x, y) => 3 + 5

    }

```

Figure 13: *HamSkill* Generated Scala Code for Function “myFunc” **with** Currying

HamSkill enables currying for all functions that appear in the input Haskell file.

5.5 Calling a Function

Many languages (e.g. Java, C/C++, Scala) require that a function’s parameter be explicitly defined (usually by comma separating them within parentheses) the parameters of functions. In contrast, Haskell’s compiler and linker identify function calls and automatically manage passing that function the appropriate number of arguments. This complicates the transpilation of code from Haskell to languages like Scala.

In *HamSkill*, there are three separate approaches that can be used to tell the parser the parameters of a function.

5.5.1 Using the \$ Operator

If a function takes only a single parameter, the right-associative dollar sign (“\$”) operator can be used to tell *HamSkill* that the subsequent value is a func-

tion parameter. An example of this is shown in the Haskell code in figure 14². Note that functions “`putStrLn`” and “`show`” both take a single argument and that both use the “`$`” syntax.

```
main :: IO ()
main = do
  putStrLn $ show 3
  putStrLn $ show 4
  if (5 > 3)
    then ( putStrLn $ show 1 )
    else ( putStrLn $ show 0 )
```

Figure 14: Specifying Haskell Function Parameters using “`$`” in Haskell

As a point of reference, figure 15 shows the `main` function from figure 14 transpiled by *HamSkill* into Scala.

```
def main(args : Array[String]){
  println ((3).toString())
  println ((4).toString())
  if(5 > 3)
  {
    println ((1).toString())
  }
  else{
    println ((0).toString())
  }
}
```

Figure 15: *HamSkill* Transpiled Scala Code of Functions using “`$`” to Specify Function Parameters

5.5.2 Surround Each Function Parameter in Parentheses

As explained in section 5.4, Scala requires that each parameter of a curried function be surrounded by parentheses. If this same formatting is used in the Haskell source code, then *HamSkill* will automatically treat the statement as a function call. An example usage of this Haskell syntax is shown in figure 16.³

²The Haskell code in this example is in the *HamSkill* test case: “`main_print_string.hs`”.

³The Haskell code in this figure is in the *HamSkill* test case: “`compare_ops.hs`”.

Note that for the function calls to `notEqual` in `main`, the parameters (e.g. “5” and “4” as well as “3” and “3”) are each surrounded by their own sets of parentheses.

```
notEqual :: Int -> Int -> String
notEqual x y = if (x /= y)
                then ("not equal")
                else ("equal")

main :: IO ()
main = do
    putStrLn $ notEqual (5)(4)
    putStrLn $ notEqual (3)(3)
```

Figure 16: Haskell Code where Parentheses around Arguments are Used to Specify Function Arguments

For completeness, figure 17 shows the *HamSkill* transpiled version of the Haskell code in figure 16.

```
private def notEqual (___0___ : Int) (___1___ : Int)
    : String = (___0___, ___1___) match {
    case (x, y) => if (x != y)
    {
        "not equal"
    }
    else {
        "equal"
    }
}

def main(args : Array[String]){
    println (notEqual (5) (4))
    println (notEqual (3) (3))
    ...
}
```

Figure 17: Transpiled *HamSkill* Code for Function Parameters Each with their Own Parentheses

5.5.3 Triple Parentheses Notation

Rather than surrounding each individual function parameter by its own set of parentheses, *HamSkill* supports surrounding the entire function call (i.e. the function name and all its parameters) in triple parentheses. This syntax is shown in figure 18; note that both the initial and recursive calls to the “**factorial**” function use triple parentheses.

```
factorial :: Int -> Int
factorial n = if (n == 0)
               then (1)
               else ( n * (((factorial (n-1) ))) )

main :: IO ()
main = putStrLn $ show $ (((factorial 10)))
```

Figure 18: Haskell Code where a Function’s Parameters are Specified Using Triple Parentheses Notation

Figure 19 shows the Haskell **factorial** triple parentheses function call transpiled by *HamSkill* to Scala.

```
private def factorial ( ___0___ : Int) :
                                     Int = ( ___0___ ) match {
  case (n) => if(n == 0)
    {
      1
    }
    else{
      n * factorial((n -1))
    }
}

def main(args : Array[String]){
  println ((factorial(10)).toString())
}
```

Figure 19: *HamSkill* Transpiled Code for the **factorial** Function with Triple Parentheses Function Call Notation

5.6 Higher-Order Function Support

One of the most important and widely used features of Haskell is its ability to treat functions as first class objects. Because of that, it was very important to build higher-order function support into *HamSkill*.

Figure 8 shows Haskell’s `foldr` function implemented in a *HamSkill* test bench file. Other higher-order function implementations that are part of *HamSkill*’s test bench include: “`filter`” (see figure 20⁴) and “`map`” (see figure 21⁵).

```
zayd_filter :: (Int -> Bool) -> [Int] -> [Int]
zayd_filter - [] = []
zayd_filter f (x:xs) = if ( ((f x)))
                        then ( (x):(zayd_filter(f) (xs) ))
                        else (zayd_filter(f)(xs))
```

Figure 20: *HamSkill* Supported Implementation of Haskell’s `filter` Function

```
zayd_map :: (Int -> Int) -> [Int] -> [Int]
zayd_map - [] = []
zayd_map f (x:xs) = ( f(x) ):( zayd_map(f)(xs) )
```

Figure 21: *HamSkill* Supported Implementation of Haskell’s `map` Function

5.7 Conversion from Haskell Functions to Scala Object Methods

Since Haskell does not support objects, it only has functions; Haskell does not have methods. In contrast, Scala supports objects. Hence, when transpiling Haskell code to Scala, the parser must identify those Haskell functions that must be converted to Scala methods.

HamSkill supports converting two types of functions to methods namely: “`show`” which converts values to strings and “`length`”, which returns the number of elements in a list. Figures 11 and 19 show Haskell code from the *HamSkill* test bench where functions are converted to methods.

⁴The equivalent “`filter`” function is in the test bench file “`filter_example.hs`”.

⁵The equivalent “`map`” function is in the test bench file “`map_example.hs`”.

5.8 Partially Applied Functions

One of the benefits of function currying is that it enables partially applied functions. *HamSkill* supports partially applied functions with only with integers as the function input and output parameters. What is more, partially applied functions need to be defined as a dedicated function and are the only exception to the requirement that functions have an explicit type signature (for more details, see section 5.2.1).

Figure 22 shows a partially applied Haskell function (“addTwoPlusOne”) that is supported by *HamSkill*.⁶ Figure 23 shows the transpiled version of addTwoPlus. Note that Scala’s partially applied function call is followed by an underscore (“_”). This notation is used to tell the Scala compiler that the object is a partially applied function and is the reason for the *HamSkill* requirement that partially applied functions be their own functions with no type signature.

```
add3 :: Int -> Int -> Int -> Int
add3 x y z = x + y + z

addTwoPlusOne = add3(1)
```

Figure 22: Partially Applied Haskell Function addTwoPlusOne

```
private def add3 (___0___ : Int) (___1___ : Int) (___2___ : Int)
               : Int = (___0___, ___1___, ___2___) match {
    case (x, y, z) => x + y + z
}

def addTwoPlusOne = {add3 (1) _ } /
```

Figure 23: Scala Version of Haskell Function “addTwoPlusOne” Generated by *HamSkill*

5.9 Converting a Haskell Lambda Function to a Scala Anonymous Function

Haskell’s lambda functions allow a program to create a function without explicitly defining it. The concept of lambda functions exists in Scala where they

⁶The function “addTwoPlusOne” is in the *HamSkill* test bench file “partially_applied.example.hs”.

are known as “anonymous functions”.

Figure 24 shows an example lambda function that is parsable by *HamSkill*; this function returns a **Boolean** value depending on whether `xINT` is greater than 30.⁷ Also, the suffix “INT” at the end of the variable name tells *HamSkill* the parameter’s type (integer in this example).

To simplify parsing, *HamSkill* requires that lambda functions be surrounded by parentheses.

```
main :: IO ()
main = putStrLn $ show $ ( zayd_filter ( (\xINT -> xINT > 30) )
                               ([1, 459, 43, (-30), 34]) )
```

Figure 24: A Haskell Lambda Function Supported by *HamSkill*

Figure 25 is the *HamSkill* generated Scala code for the Haskell lambda function in figure 24.

```
def main(args : Array[String]){
  println (((zayd_filter (((xINT: Int) => xINT > 30))
                        (List(1, 459, 43, (-30),
34))))).toString())
}
```

Figure 25: A Scala Anonymous Function Generated by *HamSkill*

5.10 Supported Types

HamSkill only supports a subset of Haskell’s available types, namely: **Bool**, **Integer** (i.e. bounded), and finite **Lists**. The Haskell integer operators that are supported are: addition (“+”), subtraction (“-”), multiplication (“*”), division (“div” - infix only), and modulus (“mod” - infix only)⁸. In addition, the following integer comparison operators are supported: equal (“==”), not equal (“/=”), greater than (“>”), greater than or equal (“>=”), less than (“<”), and less than or equal (“<=”) ⁹.

⁷The example lambda function is in the *HamSkill* test bench file “filter.example.hs”.

⁸The integer mathematical operators are verified in the *HamSkill* test bench file “simple_math.hs”.

⁹The comparison operators are verified in the *HamSkill* test bench file “compare_ops.hs”.

While implementing floating point numbers would not add substantial complexity at a basic level, ensuring that the floating point behavior of *HamSkill* (i.e. Scala) and Haskell are identical is beyond the scope of this project.

There is also limited support for **String** values as well; however, operations (e.g. concatenation) on **String** values is not supported.

5.11 Lazy Evaluation and Immutability

Since Haskell is a purely functional language, all values are immutable; what is more, all evaluation in Haskell is lazy. In contrast, while Scala is a functional language, it is not purely function. As such, variables must be explicitly declared as immutable; they can also be declared as lazy; this is done by defining the variable with the keywords “**val**” and “**lazy**” respectively. *HamSkill*’s conversion of Haskell values to lazy, immutable Scala values is shown figure 11 in section 5.3.

5.12 Defining Scope via Haskell’s module Keyword

A program in Haskell is composed of a set of “module” files. The “module” keyword is used to specify the scope of functions (i.e. publicly visible or private). Any Haskell function that is not included in a “module” statement will be defined as **private** in the transpiled Scala code (with the exception of “**main**”). Figure 26 shows a *HamSkill* parsable “module” block.¹⁰

```
module Simple_Function_Call (
    myFunc2,
    myFunc
) where
```

Figure 26: *HamSkill* Parsable “module” Statement

5.13 Maybe Monad Support

One of the stretch goals for this project was monad support; this goal was met by as *HamSkill* supports Haskell’s **Maybe** monad (referred to as “**Option**”

¹⁰This “module” block is in the *HamSkill* test bench file “**simple_function_call.hs**”.

in Scala). The requirements for use of `Maybe` are:

1. Use of the “do” syntax. Note bind (“>=”) is not supported.
2. The last line of the “do” block must be `return` with some value.
3. The `Maybe` do block cannot be in the “main” function.

Figures 27 and 28 show the original Haskell `Maybe` Monad code and the *HamSkill* transpiled Scala code respectively. Note that “do” in Haskell is changed to “for” in Scala while “return” becomes “yield”.¹¹

```
doubleIncrementMonad :: Int -> (Maybe Int)
doubleIncrementMonad x = do
    y <- maybeAddOne (x)
    z <- maybeAddOne (y)
    return z
```

Figure 27: Haskell `Maybe` Code Supported by *HamSkill*

```
private def doubleIncrementMonad (___0___ : Int)
    : Option[ Int] = (___0___) match {
    case (x) => (for {
        y <- (maybeAddOne (x))
        z <- (maybeAddOne (y))
    }
    yield (z))
}
```

Figure 28: *HamSkill* Generated Scala `Option` Monad Code

5.14 if Conditionals

Like most languages, Haskell supports a conditional statement via the “if”, “then”, “else” model. Figure 29 shows Haskell code that is parsable by *HamSkill*. One specific formatting requirement of *HamSkill* is that contents of the

¹¹The “`Maybe`” monad example is in the test bench file “`maybe.monad.hs`”.

“if”/“then”/“else” must each be enclosed within parentheses as shown in the figure.

```
ifCheck :: Bool -> String
ifCheck x = if (x) then ("True") else ("False")
```

Figure 29: Haskell “if” Statement Supported by *HamSkill*

Figure 30 shows the transpiled output generated by *HamSkill*.¹²

```
private def ifCheck (---0--- : Boolean)
                  : String = (---0---) match {
  case (x) => if(x)
  {
    "True"
  }
  else {
    "False"
  }
}
```

Figure 30: Scala Transpiled “if” Statement Generated by *HamSkill*

5.15 case Statements

case statements are often preferable over **if** conditional statements due to **case**’s conciseness. What is more, **case** in Haskell is often more useful when performing pattern matching on boxed results such as a monad.

HamSkill supports Haskell’s **case** statement when the syntax meets the following two criteria:

1. The compare parameter must be enclosed in parentheses.
2. The **case** statement ends with an “otherwise” clause.

¹²The “if” example code shown in these figures is included in the *HamSkill* test case “if_statement”.

Figure 31 shows a *HamSkill* supported **case** statement.¹³ Figure 32 is *HamSkill* transpiled Scala code for the **case** statement in figure 31.

```
case_example :: Int -> [Char]
case_example x = case (x) of
    1 -> "Positive One"
    (-1) -> "Negative One"
    0 -> "Zero"
    otherwise -> error("Something went wrong here")
```

Figure 31: Haskell **case** Statement

```
private def case_example (---0--- : Int)
    : String = (---0---) match {
    case (x) => (x) match {
        case (1) => ("Positive_One")
        case ((-1)) => ("Negative_One")
        case (0) => ("Zero")
        case whatever => (sys.error ("Something_went_wrong_here"))
    }
}
```

Figure 32: Scala Transpiled “**case**” Statement Generated by *HamSkill*

Note that the Haskell “**error**” function in figure 31 was transpiled to “**sys.error**” in figure 32.

5.16 Preserving Comments

As mentioned in section 2, one of the goals of this project was to generate Scala code that was maximally human readable. This goal extends to preserving Haskell comments in the transpiled Scala code.

Figure 33 shows a header block of comments at the top of the test case file “**simple_function_call.hs**”, and figure 34 shows the transpiled Scala version.

¹³This “**case**” statement is in the *HamSkill* test case “**case_example.hs**”.

```
{-
  Name: Zayd Hammoudeh
  Class: CS 252
  Assignment: Final Project
  Date: April 1, 2016
  Description: simple_function_call test case.
-}
```

Figure 33: Haskell Header Comment

```
/*
  Name : Zayd Hammoudeh
  Class : CS 252
  Assignment : Final Project
  Date : April 1 , 2016
  Description : simple_function_call test case.
*/
```

Figure 34: Scala Transpiled Header Comment

Inline comments from Haskell are also preserved in the transpiled code.

6 Future Work and Improvements

To some degree, the amount of features supported by a tool like *HamSkill* is proportional to the amount of time spent developing the tool. Given more time, support for the following features may add significant value for users:

1. Multiple files
2. Entry to the file via functions other than `main`
3. Algebraic data types
4. `where` and `let` functionality
5. Key Haskell library integration

What is more, rather than transpiling to Scala, a transpilation to Java byte-code (rather than Scala) would be the ideal solution to ensure the minimum restrictions on the input Haskell implementation.

7 Conclusions

HamSkill provides a foundation for running a dialect of Haskell inside the JVM. All of the original key features outlined in the project proposal were implemented as well as additional features including Monads and the Scala output reformatter. While *HamSkill* does have limitations as all languages do (in particular those with a scope and duration as limited as this), it is viewed by the developer as a very successful project.

Bibliography

- [1] Home `jruby.org`. <http://jruby.org/>. (Accessed on 02/24/2016).
- [2] Platform specific notes. <http://www.jython.org/archive/21/platform.html>. (Accessed on 02/25/2016).
- [3] Renjin.org — about. <http://www.renjin.org/about.html>. (Accessed on 02/25/2016).
- [4] The scala programming language. <http://www.scala-lang.org/>. (Accessed on 02/25/2016).
- [5] `twitter/util`: Wonderful reusable code from twitter. <https://github.com/twitter/util>. (Accessed on 03/30/2016).
- [6] `util-eval` refers to classes that are removed or deprecated in scala 2.11.x issue #105 `twitter/util`. <https://github.com/twitter/util/issues/105>. (Accessed on 03/30/2016).
- [7] What is scala? — the scala programming language. <http://www.scala-lang.org/what-is-scala.html>. (Accessed on 03/30/2016).
- [8] Remo H. Jansen. *Learning TypeScript*. Packt Publishing, 2015.
- [9] Lipovaca Miran. *Learn you a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011.
- [10] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2012.
- [11] Ingo Wechsung. `Frege/frege`: Frege is a haskell for the jvm. it brings purely functional programing to the java platform. <https://github.com/Frege/frege>. (Accessed on 02/25/2016).