MAlice Compiler Project

Magdalena Gocek Varun Verma Harshwardhan Ostwal

December 17, 2012

1 Introduction

This specification provides an overview for the project which involved building a compiler for language specification MAlice.

2 Overview

2.1 Analysis

The compiler which has been built allows the user to provide a MAlice(*.alice) file and generate equivalent code for the Malice file in llvm bytecode or x86 Assembly Instructions. It supports the functionality which was specified in the specification for the language malice. The tools which have been used in development for this compiler are LLVM and ANTLR. The scope of these tools have been described further in there respective sections.

2.1.1 ANTLR (ANother Tool for Language Recognition)

We used ANTLR to perform syntactic and semantic analysis. It allows the user to define the grammar in a specific syntax and generates Lexer and Parser classes for this grammar in a specified output language. ANTLR uses Top-Down LR parsing. Due to this, backtracking option has to be enabled. This compromises with the performance. This is the most straightforward way for parsing but at the same time it also slows down the performance. When the backtracking option is enabled and the parsing fails during the execution of a defined rule in the grammar, the parser has to track back to the previous node and check if there are any other viable routes which could be taken to parse the given input. Because of the way ANTLR parses the grammar, it was not the most suitable tool for this project. On the other hand ANTLR does provide some user-friendly graphical tools to verify the grammar, but they are not always accurate. ANTLR tutorials[1] was used to learn how to use ANTLR.

2.1.2 LLVM (Low Level Virtual Machine)

We used LLVM as the target language for the MAlice compiler. This is a widely used compiler's back-end tool which allows the user to output to various different architectures, such as ARM and x86. LLVM provides the user with three different optimisation levels. We opted for this tool because it is widely-used and has a very active development group. Which is why LLVM would be able to support new architectures as long as they have backwards compatibility. This means that our compiler will be able to operate on x86 and various other machines. LLVM demo[2] was used to figure out the syntax which was required for code generation.

2.2 Functionality

This compiler currently supports all the statements from the MAlice language specification. It provides the support for loops, conditional statements, functions, procedures and mathematical operations. Currently, there are some bugs occurring in array processing, nested functions and code blocks. The problems which we encountered while working on nested functions and code blocks are because free variable analysis is required to perform Lambda Lifting. This is not currently supported by our compiler. Another performance issue which came to our notice is that the nodes of the Abstract Syntax Tree (AST) are represented by Strings, which means that String comparison has to be performed in order to analyse the nodes. This has a significant effect on the performance, as String comparison is highly resource-expensive in terms of processing and memory. Instead, building our own tree nodes would've allowed us to use enumerated types to represent tokens. This would have also significantly improved the performance of parser.

2.3 Optimisations

Malice compiler supports following optimisations:

- Evaluation of a constant expression
- Elimination of unreachable code

For constant expressions, we took the approach of evaluating values of each of the arguments for a given operator in a function. In the cases where all the operands evaluate to some integers, the operator is applied on those values and it's result is returned. Elimination of dead code is done similarly. When an argument can't be evaluated, it's corresponding code is declared dead. Currently the internal representation of our compiler treats an integer value 1 as true and 0 to be false. The type of expression is first checked in during Semantic Analysis. This ensures that if a given expression returns 1 then it means true and not the evaluated integer value 1.

3 Design

The design uses modular structure to allow each component to only carry out one task at a time which simplifies componenet-level testing. Most of them have their own test suites, which make use of JUnit4 environment to run the tests. The design is structured so that different packages are used for various parts of the compiler. Because of this, replacing any class with a new or different version of it would require minimal modifications. Extensive testing of all major classes enabled us to find the bugs quickly and efficiently and allowed us to produce more robust and reliable product.

4 Extensions

MAlice compiler has been implemented with three extensions. Some of these are still not working for all cases and would require more testing and development in order to reach a stable working version. These extensions vary vastly from modifying grammar to changing the generated abstract syntax tree using the current supported functionality but with easier syntax for the user.

- Import Statements
- Higher Order Function Map
- Mathematical Library

4.1 Import Statement

Malice supports Import statements in the following format:

```
    Alice wants <function names> from <filename>.
    Alice wants everything from "filename".
    Alice wants function1, function2 from <filename>.
```

The first statement describes the basic structure of an Import statement. The keyword "everything" can be used to import all functions from the given file. Currently, there is no way to import any global declarations from a MAlice file. Functions can be imported individually and the user would have to make sure that any global statements which might be required are either added to the file otherwise the program would not pass the semantics check. Import statements can only be added as an header for a Malice file.

Malice compiler checks for cyclic dependencies by using a depth first search approach. The file which has been imported is stored in a Hash Set data structure and if any previous seen file is again encountered then an error message specifying the cyclic dependencies is printed. This extension allows a user to write modularised code as well as provides a better collaboration with the team, such as different people in the team can work on different modules in Malice. It also allows user to write their own standard library which they might use quite often.

4.2 Higher Order Function Map

Malice supports Map function in the following format:

```
    map(functionName, input, output).
```

Map(map) function in malice requires three arguments. The first argument is name of a single arity function. This function has to have parameter of the same type as the type of element of the 'input' array. This function when applied to a given value should return a value of type which has to be same as the type for the element of the output array. The function map in Malice is translated into a while loop which executes the calls for function call iteratively with help of a variable to keep track of the current index.

4.3 Mathematical Library

Malice compiler now supports a built in library for the mathematical functions which are very commonly used in a lot of widely used programming languages. The functions which are supported are all single arity functions and include ”sin”, ”cos”, ”tan”, ”arctan” , ”arcsin”, ”arccos”, ”cosh”, ”sinh”, ”tanh”, ”exp” and ”sqrt”. After building the library it was observed that we would require another type to represent a floating point number in malice in order to use some of these functions. For example ‘arcsin‘ requires an argument between -1 and 1 and since in Malice we only have number type to represent integer, so we can only use one value which is 0. Currently a float type has not yet been added but in later stages of the development adding another type to represent floating point number would allow the use of the functions.

5 Summary

Git was used for version control of the project. It allowed smoother team work by providing each individual in the team to work on a different part of the project. One of the main issues which has been encountered has been while using ANTLR. Recently a new version of ANTLR has been released and it seemed to have quite a lot of bugs which caused us a lot of delays in generating the AST. The lack of analysis on each of individual tools which could have been used caused quite a lot of problems. ANTLR is very straight forward tool to use but it does have its limitations and its not the most suitable tool for this language.

References

[1] ANTLR Tutorials, (`http://javadude.com/articles/antlr3xtut/`)

[2] LLVM Demo, (`http://llvm.org/demo/index.cgi`)