MAlice Compiler Project

Magdalena Gocek Varun Verma Harshwardhan Ostwal

December 14, 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)

ANTLR has been used to perform syntactic and semantic analysis. It allows the user to define the grammar in a specific syntax and then generates Lexer and Parser classes for the given grammar in the specified output language. ANTLR uses Top-Down LR parsing. Due to this backtracking option has to be enabled which compromises with the performance issues. This is the most straight forward way for parsing but at the same time it also slows down the performance. When the backtracking option is enabled, if the parsing fails in the middle of any defined rule in the grammar, the parser has to track back to the previous node and see if there are any other viable routes which could be taken to parse the given input. ANTLR was not the most suitable tool for this project because of the method of parsing which it uses which is not most suited to the grammar. On the other hand ANTLR does provides very user-friendly graphical tools in order to verify the grammar but there are a lot of disadvantages to it.

2.1.2 LLVM (Low Level Virtual Machine)

LLVM has been used as the target language for the MAlice compiler. This is a widely used compiler's back-end tool which allows user to be able to output to various different architectures, such as ARM and x86. LLVM provides the user with three different optimisation levels which can be used if the user requires to use them. This tool has been used because it is widely-used and has a very active development group which would allow the user to be able to use the optimisations and output architectures if any new architectures are introduced in future as long as LLVM supports backwards compatibility.

2.2 Functionality

Malice compiler currently supports all the statements which are specified in the language specification for Malice. It provides the support for loops, conditional statements, functions, procedures and mathematical operations. Currently there are some bugs beings experienced in the array processing, nested functions and code blocks. The problem which has been encountered for nested functions and code blocks are because it requires Lambda Lifting which requires a free variable analysis. This is currently not been supported by the compiler. Another performance issue which has been noticed is due to string comparison for the tokens in the nodes of the Abstract Syntax Tree (AST). The tree which ANTLR produces stores the nodes with a text field for the token. This affects significantly on the performance due to string comparison being highly resource expensive in terms of processing and memory. If the tree node were self-built then tokens, which would be an enumerated type would be used.

2.3 Optimisations

Malice compiler supports following optimisations:

- Evaluation of constant expression
- Elimination of unreachable code

Optimisation for constant expressions evaluation is done by evaluating the values for each of the argument for a given operator in a function and then if they evaluate to an integer then the operator is applied and the evaluated result is returned. The optimisation for elimination of dead code is done by a similar approach to the evaluation of the constant expressions. Currently the internal representation of the compiler treats an integer value 1 as true and 0 to be false. The type of expression has previously been checked in the semantics analysis part which ensure that if a given expression returns 1 then it is not evaluated integer value 1.

3 Design

The design uses a module structure to allow each individual component only carrying out one task which makes it easier to carry out component level testing. Most of the component which have been written have their own test suite which uses JUnit4 for the environment to run the test suites. The design is very well structured, different packages for different parts of the compilers are used. It can be easily seen what replacing any class with a new or different version of that component would require minimal modifications.

Testing has been a major part of the development of the compiler since it has been able to direct in the correct direction for existence of a bug. It allowed the development part to produce more robust and reliable product.

4 Extensions

Malice compiler has been implemented with three extensions which have been tested. Some of these are still not working for all cases and would require some more of 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 imports statements in the following format:

```
Alice wants <function names> from <filename>.
Alice wants everything from "filename".
Alice wants initialise, sumArray from <filename>.
```

The first statement describes the basic structure of the import statement. The import statements can only be added as an header for a Malice file. Keyword "everything" can be used which imports all the global declarations and functions from the given file. Currently to import any global declarations from a Malice file can only be done using the keyword "everything". Only 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.

Malice compiler checks for cyclic dependencies by using a depth first search implementation. 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 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 has its limitations and its not the most suitable tool for this language.