The MAlice Compiler: Milestone 1

Second Year Computing Laboratory
Department of Computing
Imperial College London

Summary

You are to provide a language specification for MAlice detailing your understanding of the language based on a set of example MALice programs and their behaviour. This specification will provide the basis for the front-end of your compiler. A formal specification is not required. You should, however, aim to be accurate enough that were your specification to be given to a competent coder unfamiliar with the MAlice language they would be able to write a valid program.

Details

In order to write a compiler for a language it is important to be clear what the definition of that language is. This definition should convey not only the syntax of the language (the symbols, keywords and statements that make up a valid program) but also the semantics of the language.

Consider Java; the Java syntax for declaration of a primitive type is:

```
TYPE IDENTIFIER SEMICOLON
```

Thus '`int x;`' is valid and '`char;`' is invalid. The semantics of Java[1] dictate further rules regarding the scope and meaning of the declaration '`int x;`'. For example, a redeclaration of `x` in the same scope is defined as a compile time error. Taking these two combined factors we can be sure that

```
int x;
int y;
int z;
```

is a valid Java program and

```
int x;
int x;
int;
```

is an invalid Java program.

While you are not expected to provide a full formal language specification you may find it helpful for future milestones if you provide a BNF (Backus-Naur Form) grammar to describe the syntax of MAlice.

You will be provided with a number of sample MAlice programs and their output. To begin this exercise you should look through these and try to work out what each does. In doing this you should also consider what constitutes invalid programs.

Some languages also include "Implementation Defined" behaviour in their specification. This can cause portability issues but allows a language designer to not have to design around the limits of any particular system. Consider, as an example, the number of bits in a byte. While you are no doubt used to a byte being precisely 8 bits, a byte historically merely described a collection of enough bits to contain the full character set of the system. The ANSI C standard defines a byte as:

[1] Available for light bedtime reading at `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`

Byte — the unit of data storage in the execution environment large enough to hold any member of the basic character set of the execution environment. It shall be possible to express the address of each individual byte of an object uniquely. A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit.

With the size of a byte providing the abstraction from bit level specifics, a char can be defined as a unit of storage one byte wide on all systems. This gives programmers a consistent expectation across systems, while leaving the actual details of implementation to the compiler writer.

You should make decisions regarding what features of the language will be "Implementation Defined" and include these in your specification. Consider carefully the trade offs you are making between code portability and system abstraction when you make these decisions.

You should also describe what has not been defined by the example programs. Details of the language which cannot be discerned should be noted in your specification along with a description of the assumption you would make as to their semantic meaning.

Submit by: 5th November 2012

What To Do:

1. Each group has been provided with a bare Git repository on the department's `labranch` server. This is where you should work on your compiler in the later milestones. To obtain this skeleton repository you will need to clone it into your local workspace. You can do this with the following command:

   ```
   git clone ssh://<login>@labranch.doc.ic.ac.uk:10022/lab/groups/<groupNo>/1213/malice
   ```

   replacing `<login>` with your normal college login and `<groupNo>` with your group number. You can find your group number (and a list of your group's members) on the `malice` exercise page of the `labranch` repository webpages available at:

   `https://www.doc.ic.ac.uk/~tora/firstyear/lab/`

   The provided files in this repository are:

`malice-spec.tex`	a LaTeX source file that can be edited to create your specification document
`README.txt`	a text file explaining how the auto-tester experts to build and run your compiler

2. We have set up a world-readable Git repository that contains a number of example MAlice programs. You can clone a copy of this repository into your local workspace with the following command:

   ```
   git clone ssh://<login>@labranch.doc.ic.ac.uk:10022/lab/provided/1213/malice_examples
   ```

   replacing `<login>` with your normal college login. Note that you can create your own examples, but you cannot push any changes back to the `malice_examples` repository.

   The `malice_examples` repository contains the following files:

`ex##.alice`	20 example MAlice programs, some correct, others with syntactic/semantic errors
`output`	a file displaying the output of the compilation and execution of the example programs

   At the start of Milestone 2 we will be pushing additional examples to this repository to help you write your compiler.

3. Read through the provided programs and work out, from their expected output, the language constructs of MAlice.

4. Create a Language Specification Document. This should be a **maximum of four pages**. It should specify the BNF for the language, and should explain clearly the semantics of the language as far as you understand it from the examples.

The purpose of this exercise is to prepare for the second milestone, which involves building a compiler for this language. You may, of course, start work on this as soon as you like.

Additional Help Getting Started

Try defining the rules for arithmetic expressions first. Try asking yourself some of the following questions.

- What are the operators that MAlice allows?
- What meaning do these operators have?
- What is the correct result for (5 + 5)?
- What about (3 + 5 * 6)?
- Should (6 / 0) be a compile time error, a run time error, or should it be allowed to cause a processor exception?
- What data types will the operations be legal for?
- What should the result of ("hello" + 5) be?
- Is there a limit to the magnitude an arithmetic expression can evaluate to? What happens if this limit is exceeded?
- Can you justify this limit on all systems, or is the limit a consequence of the execution environment?
- What implications do your decisions have for your implementation?

Creating the Document

You should produce your document as a text file or a PDF. To create a PDF you should use LaTeX, or a word processor that can export or save to PDF format. **Do not print out your document and scan it** as this can produce very large files and your submission for this milestone is limited to 1MB.

Using LaTeX

The provided file `malice-spec.tex` can be edited to produce your document if you wish. The file contains some examples of how to produce different types of output using LaTeX. To generate `malice-spec.pdf` from the source file, use the command:

```
pdflatex malice-spec
```

For a good introduction to using LaTeXsee `http://en.wikibooks.org/wiki/LaTeX`. There are also many good books and other websites available.

Submission

You should submit your specification document, either `malice-spec.pdf` or `malice-spec.txt`, as a group to CATe by midnight on 5th November.

Assessment

In total there are 25 marks available in this milestone. These are allocated as follows:

Language Syntax Definition	5
Language Semantics Definition	15
Precision, Clarity and Readability	5

This milestone will constitute 15% of the marks for the MAlice exercise. Feedback on your work will be returned by 19th November 2012.