CS4201 Prog. Lang. Design & Implementation

Practical 1: A Compiler to JVM Bytecode **Due Date Tuesday 4th November, 2014**

This practical represents 50% of the assessed practical work for this module. Its aims are:

- to provide experience with designing abstract syntax structures representing concrete syntax forms;
- to provide exposure to the consequences of some simple language design issues;
- to provide practical exposure to the problems involved in writing the back-end of a simple compiler.

The Language

The concrete syntax for the CF language is as follows:

```
<class> ::=                                 CF class definition
     "class" <id> <stats>


<stats> ::=
      <stat>
   |  "{" ( <stat> ";" )* "}"               statements


<stat> ::=
     <int>                                  integer value
   | <float>                                floating point value
   | <bool>                                 boolean value
   | "new" <id> "=" <stats>                 new identifier
   | <id> "=" <stats>                       assignment
   | "while" <stats> "dont" <stats>         unwhile-loop
   | "comefrom" <label>                     comefrom statement
   | "label" <id>                           label
   | "print" <stats>                        output operator
   | <id>                                   identifier
   | <stats> <op> <stats>                   binary operator
   | "not" <stats>                          negation operator
   | "(" <stats> ")"                        grouping


<op> ::=                                    operators
      "+" | "*" | "-" | "/"| "<"| "<=" | "=="
```

A CF program is a named set of semicolon-separated statements. Statements include variable definitions, assignments, unwhile-loops, print statements, variables and basic values, plus comefrom statements and their associated labels. Operators have their usual meanings.

For example,

```
     class foo { new v = 21; print v * 2; }
```

prints 42;

```
     class bar
       {
          new x = 0;
          while not (x <= 1) dont x = x + 1;
       }
```

sets x to be 2;

```
     class foobar {
       new v = 100;
       label l;
       print false;
       comefrom l;
       print true;
       print v;
     }
```

prints `true 100`

Practical Requirements

i) Define data types (or class equivalents) to represent ASTs for CF programs.

ii) Write a *compiler* to translate your CF ASTs to JVM bytecodes. Test your compiler on a suitable range of programs. The *jasmin* bytecode assembler (http://jasmin.sourceforge.net/) will assemble JVM bytecodes into class files which you can run as normal. You *do not need to write a lexer or parser of any kind* for CF - *you can compile directly from the AST.*

You may use any programming language and software development methodology you like. The only criterion is that I must be able to understand what your program does! If you think that I may not understand it, add comments as appropriate!!

Note that in order to write a correct compiler, you may need to take a few *small* language design decisions. I will not impose particular solutions on you provided you are consistent with the specification and examples above, but some language designs may well be better than others, so be prepared to justify your decisions. Even if you do not fully complete the practical, you will still obtain a good grade if you have produced a good, well-written, working and tested solution that covers a good subset of the requirements. Even if you do not fully complete all the details of the practical, you will still obtain a good grade (grade 13.5 or better) if you have produced a good, well-written, working and carefully tested solution that covers the most important requirements. First class solutions (grade 16.5 or better) are expected to show some flair, and to cover all the requirements.

You should hand in, through MMS:

- Sources for your AST, commented to explain the relationship with the concrete syntax;
- Sources for your compiler, adequately commented;
- Explanations of any language design decisions that you have taken, and any important implementation decisions;
- Copies of output from sample test cases. I will take this as evidence of *everything* your compiler can do, so be thorough;
- Explanations for any features that you have not implemented, or any extensions that you have made to CF.

There is no need to write a more detailed report.

Please submit either PDF or plain text: I will not accept word-processor source files. If I cannot read it/print it, I cannot mark it!

Time Management

Note that late work will be automatically penalised by MMS — it is therefore generally better to hand in an OK practical on time rather than a good practical late; and you may want to hand in something that is working early, and then improve it.