Assignment 1
Ad-Hoc Syntax-Directed Compilation

Due: 11:59pm Wednesday, September 12

Goals for this Assignment:

1. Hand-implementing a parser
2. Syntax-directed translation to JVM bytecode.

E-mailed questions about this assignment should be sent to `cs132help@cs.hmc.edu`.

Instructions

Get the files from the repository and extend them as specified below. The `syndir.sml` file is designed to be compiled with `sml` via

```
use "syndir.sml";
```

As stated in the syllabus, you may work in pairs if you wish; only one person from each team needs to submit a solution as long as both names appear prominently in the comments.

The Basic Input Language

Your primary goal will be to write a compiler that generates JVM bytecode for the language described in Figure 1. It's a very simple imperative language with variables, integers, addition, subtraction, assignments, comparisons, and printing of integers, and conditional statements (`if-then-else`) .

Output Format

Java class files are actually in an unpleasant binary format. What we'd like is an assembler for JVM, but Sun doesn't provide one. Fortunately, we can use an open-source tool like `Jasmin` (`http://jasmin.sourceforge.net`).

```
<program> ::= <block>

<block> ::= <statement>
          | { <list> }

<list> ::=
         | <statement> <list>

<statement> ::= print <expr> ;
              | <identifier> = <expr> ;
              | if <expr> then <block> else <block>

<expr> ::= <integer-constant>
         | <identifier>
         | add <expr> <expr>
         | sub <expr> <expr>
```

Figure 1: Grammar for the Input Language

- The `Jasmin` distribution can be found in `/cs/cs132/jasmin-1.1/` on Knuth. To run `Jasmin` on a file `myfile.j` then, the command is

  ```
  java  -jar /cs/cs132/jasmin-1.1/jasmin.jar  myfile.j
  ```

  This translates your file into a class file that can be executed (via a command like `java Main`).

- Your program should be translated into a class named `Main`, containing a single `main` function. That is, your `Jasmin` output should be a file with the following form:

  ```
  .class public Main
  .super java/lang/Object
  .method public static main([Ljava/lang/String;)V
  .limit locals  Integer number of local variables required
  .limit stack   Maximum stack depth required

   Your code, terminating with a return instruction

  .end method
  ```

  (An easy way to get a loose upper bound on the amount of stack you will require is just to count the number of instructions generated.)

- The JVM centers on a single stack where most of the operations occur (as in PostScript or RPN calculators), along with a collection of numbered local variables for each procedure.

  The following instructions may be useful in your code:

  1. `iload` *local-variable-number*
     Pushes an integer from the specified variable onto the stack.
  2. `istore` *local-variable-number*
     Pops an integer from the stack and stores it into the specified variable.
  3. `ldc` *integer-constant*
     Pushes the given constant onto the stack.
  4. `dup`
     Push a copy of the top stack element onto the stack.
  5. `swap`
     Exchange the top two stack items
  6. `iadd`
     Replaces top two values of the stack (which must be integers) by their sum.
  7. `isub`
     Replaces top two values of the stack (which must be integers) by their difference.
  8. *label*:
     Target for a jump. The label is an arbitrary identifier, and must be followed by the colon.
  9. `goto` *label*
     The obvious
  10. `ifeq` *label*
      `ifne` *label*
      `ifle` *label*
      `iflt` *label*
      `ifge` *label*
      `ifgt` *label*
      Compares the top of the stack to zero and jumps to the given label if the comparison holds. (Consumes that value on the top of the stack in the process.)
  11. `return`
      Returns from the function or method.

  An integer on top of the stack can be printed as follows:

```
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(I)V
```

Your Assignment

1. Extend the skeleton code in to obtain a function

   ```
   compile : string -> unit
   ```

   That takes in the name of a file containing the source program, and generates the corresponding `Jasmin` output.

   You should do this by implementing a recursive-descent parser that generates the output as a list of lines of jasmin code.

   Recall that a recursive-descent parser defines a function for each nonterminal in the grammar, reads of tokens in the input stream corresponding to one of the productions for that nonterminal. (You need not follow the given grammar exactly if a small change makes life easier, as long as you accept the same programs.)

   For this assignment, your parsing functions should return the corresponding Java bytecodes for whatever source-language expression was recognized (rather than returning a syntax tree for that expression), and also any remaining unconsumed tokens.

2. Do at least two of the following extensions, and make sure these are well-documented.

   (a) Add a new statement

   ```
   while <exp> do <block>
   ```

   to the implementation, that repeats the block zero or more times, until the given expression evaluates to zero.

   (b) Change the syntax of arithmetic from prefix to infix (i.e., `<expr> + <expr>` instead of `add <expr> <expr>`, and allow parenthesized expressions).

   (c) Have the compiler compute an improved bound on the amount of stack space required.

Hints

1. The better the error messages you generate (e.g., showing the token you found and the token you were expecting, rather than just quitting with an uncaught exception), the easier it is to debug your parser.

2. Efficiency is far less important than correctness and clarity of your code.

3. If you're curious what sort of code `javac` creates, you can see the bytecodes in any class file by running

   ```
   /cs/cs132/bin/jad -dis Myclass.class
   ```

   (or whatever the class file is called) on Knuth or Wilkes.