The SMALL Programming Language

SMALL is a very simple programming languages. It uses only integer variables expect in one special case (it can print strings). Below is the usual example:

```
print “Hello World!” // add a semicolon (‘;’) if you wish.
```

The output from this is a file called `hello.j` which is a translation of the program into assembler language. The language of the .j file can be converted by jasmin (a JVM – Java Virtual Machine - assembler) into a java class file which can be executed like any java program.

1. SMALL Syntax

```
Program : StatementList

StatementList : ((‘;’)* Statement )*

Statement : IfStatement
          | WhileStatement
          | PrintStatement
          | ReadStatement
          | Assignment
          |  (an empty statememement is legal)

IfStatement  : ‘if’ Expression ‘then’ StatementList ‘end’

WhileStatement : ‘while’ Expression ‘do’ StatementList ‘end’

PrintStatement : ‘print’ Item (‘,’ Item)*

Item           : StringLiteral | Expression

ReadStatement  : ‘read’ Name (‘,’ Name)*

Assignment     : Name ‘=’ Expression

Expression     : usual expressions
```

2. The Structure of the SMALL Compiler

The code for the compiler has three stages: the lexer breaks the program into individual tokens; the syntax analyser reads the tokens and generates a parse tree; and the code generator produces JVM assembly language. A separate program (jasmin) converts the output into a java .class file. Jasmin is actually called from within the compiler so could be seen as a fourth stage. After successful compilation of a program (called, say, MyTest.sm – though any or no extension is acceptable) you should find in your current directory two files: MyTest.class is a Java class file which can be run as a program, MyTest.j is the readable assembly language program which Jasmin converts to MyTest.class.

Here is an example program:

```
sum = 0
x = 1
while x != 0 do
     print "Enter number (end with 0): "
     read x
     sum = sum + x
end

print "Sum is ", sum, "\n";
```

SMALL is notable for what it doesn't do. There are no `break` or `continue` commands for loops, `if`-statements don't have an `else` part and though in the grammar it says *`usual expressions`* a number of important ones are missing, in particular logical '`and`', '`or`' and '`not`'. You have `*, +, /, %,` plus `<, <=, >, >=, ==, !==.`

You don't have to declare variables but if a variable is first used inside a while- or an if-statement it is forgotten when the statement ends. For example:

```
x = 1
while x > 0 do
     y = 1
     x = 0
end
print y
```

will produce an error message (cased by the `print` statement) saying that `y` is used before it is initialised – the y inside the while loop has been forgotten.

4. Building and Running SMALL

Commands to build SMALL and compile SMALL programs are in the folder *uweflp\w (Windows/DOS users)* or *uweflp/b* for Linux users. Set *uweflp* as your current directory and at the command line type

Linux	Windows	
./b/build	w\build	Build the compiler
./b/small *filename*	w\small *filename*	Compile *filename* leaving a *.j* and a *.class* version in *the* current directory.
./b/run *classname*	w\run *classname*	Run the program
./b/smallgo *filename*	w\smallgo *filename*	Compiles the source, guesses the class name and if no errors, runs the program.

5. Compiler Output

The SMALL compiler is written in Java. The source code is in the download as two packages: sal.util.*.java contains general purpose code which you don't need to look at and sal.small.*.java which contains the compiler.

As mentioned above, SMALL produces two files in the current directory. A file called X.sm when compiled without errors produces X.j and X.class. The second file is a Java class file which can be run by typing w\run X .

X.j can be opened with a text editor. It contains Jasmin JVM assembly code. The output for the example "Hello World" program looks like this. The grey sections are standard and can be ignored. They will be in every .j file and will always be the same (except the number in the last but one line which is produced automatically by the compiler.

```
.class public HelloWorld
.super java/lang/Object

.method public <init>()V
.limit stack 10
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
.limit stack 10

    ldc "Hello World!\n"

    invokestatic sal/Library/print(Ljava.lang.String;)V

    return
.limit locals 1
.end method
```

The active part of the program is the line: `ldc "Hello World!\n"` which loads a constant (in this case a String literal) onto the stack, and the line: `invokestatic sal/Library/print(Ljava.lang.String;)V` which calls a static method `sal.Library.print` which has return type void (the `'V'` at the end), with a single String parameter (String's complete java name is `java.lang.String` but in a .class file it is stored as `Ljava.lang.String;` - why `'L'` nobody seems to know!

If you look at the lines before and after the `ldc/invokestatic` lines you may get a clue as to what the SMALL compiler does. The line `.method public static main([Ljava/lang/String;)V` will be familiar to you in its Java version:

```
public static void main(String[] args)
```

SMALL's output becomes the body of the Java `main` method.

6. JVM Instructions

The SMALL compiler uses only a few of the JVM assembler instructions. With the exception of printing strings, all the operations used work on integers. Calls to methods are only used for input and output operations (defined in sal\Library.java). All you need for this course is loading and storing integers, arithmetic and jumps (‘goto’s) within code.

6.1 Arithmetic

The JVM is a stack based machine. An example should clarify what happens.

```
ldc   4        load 4 onto the stack
ldc   2        load 2 onto the stack
iadd           add top two values, leave the result on the stack
```

This is the basic model for all arithmetic. Here is a more complicated example: 3+4*(1+2)

```
ldc   3        load 3 onto the stack
ldc   4        load 4 onto the stack
ldc   1        load 1 onto the stack
ldc   2        load 1 onto the stack
iadd           adds 1 and 2 leaves 3 on stack
imul           multiplies top two values 4 and 3 (result of add)
iadd           adds top two values 3 and 12 (result of multiply)
```

A diagram may be clearer:

after	stack			
ldc 3	3			
ldc 4	3	4		
ldc 1	3	4	1	
ldc 2	3	4	1	2
iadd	3	4	3	
imul	3	12		
iadd	15			

In the previous section there is a line: `.limit stack 10` which tells the JVM that a 10 item stack is enough. I chose 10 as a guess. Try writing a an expression that needs a bigger stack than 10!

In addition to `iadd` and `imul` there are `isub`, `idiv` and `irem` (remainder). There is also `ineg` which negates the top stack element (e.g. as in `x = -y`), plus shifts operations.

6.2 Load and Store

So far the examples have used `ldc` to load constants. The instructions iload and istore are used to load (integer) variables onto the stack and to store the value at the top of a stack into a variable.

The names used for variables aren't used by the assembler, instead they are numbered (from 0). The compiler automatically allocates suitable numbers. The SMALL statement x = y / 4, assuming x is variable 1 and y is variable 2 will produce the code:

```
iload  2       ; i.e. the value of variable 2 (y)
ldc    4       ; constant 4
idiv           ; divide
istore 1       ; result to variable 1 (x)
```

Just as `iload` adds an element to the stack, `istore` stores a value and removes it from the top of the stack.

6.3 Jumps

There are no `while` or `if` statements at machine code level: they are converted into sequences of instructions involving tests and jumps. Jumps transfer control to labelled statements in the assembly code. Every while loop generates code based on this skeleton:

```
NEXT_LOOP:          ; a label ':' marks the end of it
     code for test  ; is while condition true
     if test false goto EXIT_LOOP
                    ; if not jump out of the loop
     statements inside while loop
     goto NEXT_LOOP ; at end, go back to test again
EXIT_LOOP:          ; outside the loop.
```

Two labels are created, one to jump out of the loop if the test is false (EXIT_LOOP), and one to go back and repeat the test (NEXT_LOOP). If you had two while loops in your program there would be a problem: two EXIT_LOOP and two NEXT_LOOP labels. To resolve this, SMALL adds a unique suffix to every label. What you will see if you look inside AddNumbers.j, for example, is NEXT_LOOP#0 and EXIT_LOOP#1. If there was another while loop it would use labels NEXT_LOOP#2 and EXIT_LOOP#3. I could have used just #0, #1 … for labels but EXIT_LOOP and NEXT_LOOP have been added to make the code more readable.

Testing whether something is true or false isn't quite a simple as I would like. Here is the code for while x < 0

```
     iload 1             ; assuming x is variable number 1
     ldc    0            ; load zero
     if_icmplt  TRUE_VAL ; if x < 0 jump to 'true case'
     iconst_0            ; more efficient equivalent of ldc 0
     goto FALSE_VAL      ; test done jump to end
TRUE_VAL:                ; here if test was true
     iconst_1            ; equivalent to ldc 1
FALSE_VAL:               ; code continues here
```

(As before the labels will have unique numbers added.) This leaves 1 (true) or 0 (false) on the stack. It looks messy and it is. Especially when you see the next line:

```
ifeq EXIT_LOOP      ; if TOS is 0 jump to loop exit
```

If I was writing this in assembler I'd replace it with:

```
iload 1             ; load x
ifge EXIT_LOOP      ; end of loop
```

Seven instructions replaced by two! I will discuss later in the course how we could make this code more efficient.

You have now seen some of the tests available in Jasmin. Here is a complete list

if_icmp instructions	**a & b on stack; jump if a op b condition is true.**
if_icmpne label	if a != b goto label
if_icmpeq label	if a == b goto label
if_icmplt label	if a < b goto label
if_icmple label	if a <= b goto label
if_icmpgt label	if a > b goto label
if_icmpge label	if a >= b goto label
if instructions	**a on stack; jump if a condition is true of a.**
ifne label	if a != 0 goto label
ifeq label	if a == 0 goto label
iflt label	if a < 0 goto label
ifle label	if a <= 0 goto label
ifgt label	if a > 0 goto label
ifge label	if a >= 0 goto label
goto	**unconditional jump**
goto label	exactly!