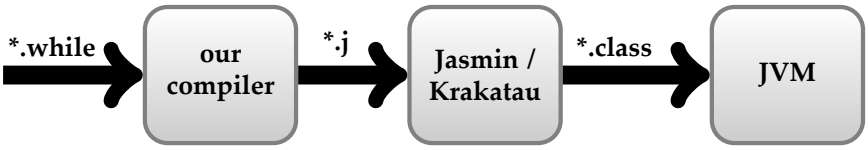
Handout 7 (Compilation)

The purpose of a compiler is to transform a program a human can read and write into code the machine can run as fast as possible. The fastest code would be machine code the CPU can run directly, but it is often good enough for improving the speed of a program to target a virtual machine instead. This produces not the fastest possible code, but code that is often pretty fast. This way of producing code has also the advantage that the virtual machine takes care of things a compiler would normally need to take care of (hairy things like explicit memory management).

As a first example in this module we will implement a compiler for the very simple WHILE-language that we parsed in the last lecture. The compiler will target the Java Virtual Machine (JVM), but not directly. Pictorially the compiler will work as follows:



The input will be WHILE-programs; the output will be assembly files (with the file extension .j). Assembly files essentially contain human-readable machine code, meaning they are not just bits and bytes, but rather something you can read and understand—with a bit of practice of course. An *assembler* will then translate the assembly files into unreadable class- or binary-files the JVM can run. Unfortunately, the Java ecosystem does not come with an assembler which would be handy for our compiler-endeavour (unlike Microsoft's Common Language Infrastructure for the .Net platform which has an assembler out-of-the-box). As a substitute we shall use the 3rd-party programs Jasmin and Krakatau

- `http://jasmin.sourceforge.net`
- `https://github.com/Storyyeller/Krakatau`

The first is a Java program and the second a program written in Python. Each of them allow us to generate *assembly* files that are still readable by humans, as opposed to class-files which are pretty much just (horrible) zeros and ones. Jasmin (respectively Krakatau) will then take our assembly files as input and generate the corresponding class-files for us.

What is good about the JVM is that it is a stack-based virtual machine, a fact which will make it easy to generate code for arithmetic expressions. For example when compiling the expression $1 + 2$ we need to generate the following three instructions

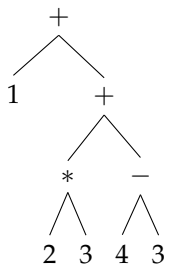
```
ldc 1
ldc 2
iadd
```

The first instruction loads the constant 1 onto the stack, the next one loads 2, the third instruction adds both numbers together replacing the top two elements of the stack with the result 3. For simplicity, we will consider throughout only arithmetic involving integer numbers. This means our main JVM instructions for arithmetic will be `iadd`, `isub`, `imul`, `idiv` and so on. The `i` stands for integer instructions in the JVM (alternatives are `d` for doubles, `l` for longs and `f` for floats etc).

Recall our grammar for arithmetic expressions ($\boldsymbol{E}$ is the starting symbol):

$$\begin{array}{lcl} \boldsymbol{E} & ::= & \boldsymbol{T} + \boldsymbol{E} \;\mid\; \boldsymbol{T} - \boldsymbol{E} \;\mid\; \boldsymbol{T} \\ \boldsymbol{T} & ::= & \boldsymbol{F} * \boldsymbol{T} \;\mid\; \boldsymbol{F} \setminus \boldsymbol{T} \;\mid\; \boldsymbol{F} \\ \boldsymbol{F} & ::= & (\,\boldsymbol{E}\,) \;\mid\; \boldsymbol{Id} \;\mid\; \boldsymbol{Num} \end{array}$$

where ***Id*** stands for variables and ***Num*** for numbers. For the moment let us omit variables from arithmetic expressions. Our parser will take this grammar and given an input program produce an abstract syntax tree. For example we will obtain for the expression $1 + ((2 * 3) + (4 - 3))$ the following tree.



To generate JVM code for this expression, we need to traverse this tree in *post-order* fashion and emit code for each node—this traversal in *post-order* fashion will produce code for a stack-machine (which is what the JVM is). Doing so for the tree above generates the instructions

```
ldc 1
ldc 2
ldc 3
imul
ldc 4
ldc 3
isub
iadd
iadd
```

If we "run" these instructions, the result 8 will be on top of the stack (I leave this to you to verify; the meaning of each instruction should be clear). The result being on the top of the stack will be an important convention we always observe in our compiler. Note, that a different bracketing of the expression, for example $(1 + (2 * 3)) + (4 - 3)$, produces a different abstract syntax tree and thus also a different list of instructions.

Generating code in this post-order-traversal fashion is rather easy to implement: it can be done with the following recursive *compile*-function, which takes the abstract syntax tree as an argument:

$$
\begin{array}{lcl}
\mathit{compile}(n) & \stackrel{\text{def}}{=} & \texttt{ldc}\ n \\
\mathit{compile}(a_1 + a_2) & \stackrel{\text{def}}{=} & \mathit{compile}(a_1) \mathbin{@} \mathit{compile}(a_2) \mathbin{@} \texttt{iadd} \\
\mathit{compile}(a_1 - a_2) & \stackrel{\text{def}}{=} & \mathit{compile}(a_1) \mathbin{@} \mathit{compile}(a_2) \mathbin{@} \texttt{isub} \\
\mathit{compile}(a_1 * a_2) & \stackrel{\text{def}}{=} & \mathit{compile}(a_1) \mathbin{@} \mathit{compile}(a_2) \mathbin{@} \texttt{imul} \\
\mathit{compile}(a_1 \backslash a_2) & \stackrel{\text{def}}{=} & \mathit{compile}(a_1) \mathbin{@} \mathit{compile}(a_2) \mathbin{@} \texttt{idiv}
\end{array}
$$

This is all fine, but our arithmetic expressions can contain variables and we have not considered them yet. To fix this we will represent our variables as *local variables* of the JVM. Essentially, local variables are an array or pointers to memory cells, containing in our case only integers. Looking up a variable can be done with the instruction

```
iload index
```

which places the content of the local variable *index* onto the stack. Storing the top of the stack into a local variable can be done by the instruction

```
istore index
```

Note that this also pops off the top of the stack. One problem we have to overcome, however, is that local variables are addressed, not by identifiers (like `x`, `foo` and so on), but by numbers (starting from 0). Therefore our compiler needs to maintain a kind of environment where variables are associated to numbers. This association needs to be unique: if we muddle up the numbers, then we essentially confuse variables and the consequence will usually be an erroneous result. Our extended *compile*-function for arithmetic expressions will therefore take two arguments: the abstract syntax tree and an environment, E, that maps identifiers to index-numbers.

$$
\begin{array}{lcl}
\mathit{compile}(n, E) & \stackrel{\text{def}}{=} & \texttt{ldc}\ n \\
\mathit{compile}(a_1 + a_2, E) & \stackrel{\text{def}}{=} & \mathit{compile}(a_1, E) \mathbin{@} \mathit{compile}(a_2, E) \mathbin{@} \texttt{iadd} \\
\mathit{compile}(a_1 - a_2, E) & \stackrel{\text{def}}{=} & \mathit{compile}(a_1, E) \mathbin{@} \mathit{compile}(a_2, E) \mathbin{@} \texttt{isub} \\
\mathit{compile}(a_1 * a_2, E) & \stackrel{\text{def}}{=} & \mathit{compile}(a_1, E) \mathbin{@} \mathit{compile}(a_2, E) \mathbin{@} \texttt{imul} \\
\mathit{compile}(a_1 \backslash a_2, E) & \stackrel{\text{def}}{=} & \mathit{compile}(a_1, E) \mathbin{@} \mathit{compile}(a_2, E) \mathbin{@} \texttt{idiv} \\
\mathit{compile}(x, E) & \stackrel{\text{def}}{=} & \texttt{iload}\ E(x)
\end{array}
$$

In the last line we generate the code for variables where $E(x)$ stands for looking up the environment to which index the variable x maps to. This is similar to the interpreter we saw earlier in the module, which also needs an environment: the difference is that the interpreter maintains a mapping from variables to current values (what is the currently the value of a variable?), while compilers need a mapping from variables to memory locations (where can I find the current value for the variable in memory?).

There is a similar *compile*-function for boolean expressions, but it includes a "trick" to do with `if`- and `while`-statements. To explain the issue let us first describe the compilation of statements of the WHILE-language. The clause for `skip` is trivial, since we do not have to generate any instruction

$$\mathit{compile}(\texttt{skip}, E) \stackrel{\text{def}}{=} ([], E)$$

whereby [] is the empty list of instructions. Note that the *compile*-function for statements returns a pair, a list of instructions (in this case the empty list) and an environment for variables. The reason for the environment is that assignments in the WHILE-language might change the environment—clearly if a variable is used for the first time, we need to allocate a new index and if it has been used before, then we need to be able to retrieve the associated index. This is reflected in the clause for compiling assignments, say $x := a$:

$$\mathit{compile}(x := a, E) \stackrel{\text{def}}{=} (\mathit{compile}(a, E) \mathbin{@} \texttt{istore}\ \mathit{index}, E')$$

We first generate code for the right-hand side of the assignment (that is the arithmetic expression a) and then add an `istore`-instruction at the end. By convention running the code for the arithmetic expression a will leave the result on top of the stack. After that the `istore` instruction, the result will be stored in the index corresponding to the variable x. If the variable x has been used before in the program, we just need to look up what the index is and return the environment unchanged (that is in this case $E' = E$). However, if this is the first encounter of the variable x in the program, then we have to augment the environment and assign x with the largest index in E plus one (that is $E' = E(x \mapsto \mathit{largest\_index} + 1)$). To sum up, for the assignment $x := x + 1$ we generate the following code snippet

```
iload n_x
ldc 1
iadd
istore n_x
```

where n_x is the index (or pointer to the memory) for the variable x. The Scala code for looking-up the index for the variable is as follow:

$$\mathit{index} = E.\mathit{getOrElse}(x, |E|)$$

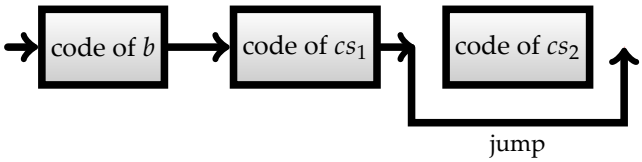
This implements the idea that in case the environment E contains an index for x, we return it. Otherwise we "create" a new index by returning the size $|E|$ of

the environment (that will be an index that is guaranteed not to be used yet). In all this we take advantage of the JVM which provides us with a potentially limitless supply of places where we can store values of variables.

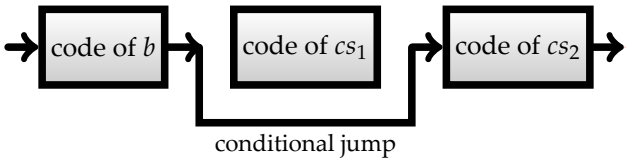
A bit more complicated is the generation of code for `if`-statements, say

```
if b then cs1 else cs2
```

where b is a boolean expression and where both $cs_{1/2}$ are the statements for each of the `if`-branches. Let us assume we already generated code for b and and the two if-branches $cs_{1/2}$. Then in the true-case the control-flow of the program needs to behave as



where we start with running the code for b; since we are in the true case we continue with running the code for cs_1. After this however, we must not run the code for cs_2, but always jump to after the last instruction of cs_2 (the code for the `else`-branch). Note that this jump is unconditional, meaning we always have to jump to the end of cs_2. The corresponding instruction of the JVM is `goto`. In case b turns out to be false we need the control-flow



where we now need a conditional jump (if the if-condition is false) from the end of the code for the boolean to the beginning of the instructions cs_2. Once we are finished with running cs_2 we can continue with whatever code comes after the if-statement.

The `goto` and the conditional jumps need addresses to where the jump should go. Since we are generating assembly code for the JVM, we do not actually have to give (numeric) addresses, but can just attach (symbolic) labels to our code. These labels specify a target for a jump. Therefore the labels need to be unique, as otherwise it would be ambiguous where a jump should go to. A label, say `L`, is attached to code like

```
L:
  instr_1
  instr_2
    ⋮
```

where the label needs to be followed by a colon. The task of the assembler (in our case Jasmin or Krakatau) is to resolve the labels to actual (numeric) addresses, for example jump 10 instructions forward, or 20 instructions backwards.

Recall the "trick" with compiling boolean expressions: the *compile*-function for boolean expressions takes three arguments: an abstract syntax tree, an environment for variable indices and also the label, *lab*, to where an conditional jump needs to go. The clause for the expression $a_1 = a_2$, for example, is as follows:

$$\begin{array}{l} compile(a_1 = a_2, E, lab) \stackrel{\text{def}}{=} \\ \quad compile(a_1, E) \mathbin{@} compile(a_2, E) \mathbin{@} \texttt{if\_icmpne}\ lab \end{array}$$

where we are first generating code for the subexpressions a_1 and a_2. This will mean after running the corresponding code there will be two integers on top of the stack. If they are equal, we do not have to do anything (except for popping them off from the stack) and just continue with the next instructions (see control-flow of ifs above). However if they are *not* equal, then we need to (conditionally) jump to the label *lab*. This can be done with the instruction

```
if_icmpne lab
```

To sum up, the third argument in the compile function for booleans specifies where to jump, in case the condition is *not* true. I leave it to you to extend the *compile*-function for the other boolean expressions. Note that we need to jump whenever the boolean is *not* true, which means we have to "negate" the jump condition—equals becomes not-equal, less becomes greater-or-equal. Other jump instructions for boolean operators are

$$\begin{array}{lcl} \neq & \Rightarrow & \texttt{if\_icmpeq} \\ < & \Rightarrow & \texttt{if\_icmpge} \\ \leq & \Rightarrow & \texttt{if\_icmpgt} \end{array}$$

and so on. If you do not like this design (it can be the source of some nasty, hard-to-detect errors), you can also change the layout of the code and first give the code for the else-branch and then for the if-branch. However in the case of while-loops this "upside-down-inside-out" way of generating code still seems the most convenient.

We are now ready to give the compile function for if-statements—remember this function returns for statements a pair consisting of the code and an environment:

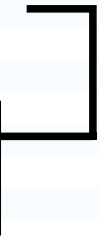
$$
\begin{array}{l}
\mathit{compile}(\texttt{if}\ b\ \texttt{then}\ cs_1\ \texttt{else}\ cs_2, E) \stackrel{\text{def}}{=} \\
\quad L_{\mathit{ifelse}}\ \text{(fresh label)} \\
\quad L_{\mathit{ifend}}\ \text{(fresh label)} \\
\quad (is_1, E') = \mathit{compile}(cs_1, E) \\
\quad (is_2, E'') = \mathit{compile}(cs_2, E') \\
\quad (\mathit{compile}(b, E, L_{\mathit{ifelse}}) \\
\quad\ @\ is_1 \\
\quad\ @\ \texttt{goto}\ L_{\mathit{ifend}} \\
\quad\ @\ L_{\mathit{ifelse}}: \\
\quad\ @\ is_2 \\
\quad\ @\ L_{\mathit{ifend}}:, E'')
\end{array}
$$

In the first two lines we generate two fresh labels for the jump addresses (just before the else-branch and just after). In the next two lines we generate the instructions for the two branches, is_1 and is_2. The final code will be first the code for b (including the label just-before-the-else-branch), then the `goto` for after the else-branch, the label L_{ifesle}, followed by the instructions for the else-branch, followed by the after-the-else-branch label. Consider for example the if-statement:

```
if 1 = 1 then x := 2 else y := 3
```

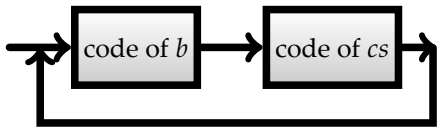
The generated code is as follows:

```
1    ldc 1
2    ldc 1
3    if_icmpne L_ifelse
4    ldc 2
5    istore 0
6    goto L_ifend
7  L_ifelse:
8    ldc 3
9    istore 1
10 L_ifend:
```

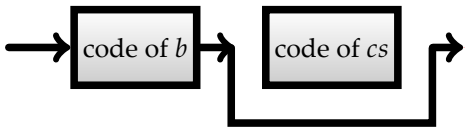


The first three lines correspond to the the boolean expression $1 = 1$. The jump for when this boolean expression is false is in Line 3. Lines 4-6 corresponds to the if-branch; the else-branch is in Lines 8 and 9. Note carefully how the environment E is threaded through the recursive calls of *compile*. The function receives an environment E, but it might extend it when compiling the if-branch, yielding E'. This happens for example in the if-statement above whenever the variable `x` has not been used before. Similarly with the environment E'' for the second call to *compile*. E'' is also the environment that needs to be returned as part of the answer.

The compilation of the while-loops, say `while` b `do` cs, is very similar. In case the condition is true and we need to do another iteration, and the control-flow needs to be as follows



Whereas if the condition is *not* true, we need to jump out of the loop, which gives the following control flow.



Again we can use the *compile*-function for boolean expressions to insert the appropriate jump to the end of the loop (label L_{wend} below).

$$
\begin{array}{l}
compile(\texttt{while}\ b\ \texttt{do}\ cs, E) \stackrel{\text{def}}{=} \\
\quad L_{wbegin}\ \text{(fresh label)} \\
\quad L_{wend}\ \text{(fresh label)} \\
\quad (is, E') = compile(cs_1, E) \\
\quad (L_{wbegin} : \\
\quad\ @\ compile(b, E, L_{wend}) \\
\quad\ @\ is \\
\quad\ @\ \text{goto}\ L_{wbegin} \\
\quad\ @\ L_{wend} :, E')
\end{array}
$$

I let you go through how this clause works. As an example you can consider the while-loop

```
while x <= 10 do x := x + 1
```

yielding the following code

```
1  L_wbegin:
2     iload 0
3     ldc 10
4     if_icmpgt L_wend
5     iload 0
6     ldc 1
7     iadd
8     istore 0
9     goto L_wbegin
10 L_wend:
```

As said, I leave it to you to decide whether the code implements the usual controlflow of while-loops.

Next we need to consider the WHILE-statement `write x`, which can be used to print out the content of a variable. For this we shall use a Java library function. In order to avoid having to generate a lot of code for each `write`-command,

we use a separate helper-method and just call this method with an appropriate argument (which of course needs to be placed onto the stack). The code of the helper-method is as follows.

```
1 .method public static write(I)V
2     .limit locals 1
3     .limit stack 2
4     getstatic java/lang/System/out Ljava/io/PrintStream;
5     iload 0
6     invokevirtual java/io/PrintStream/println(I)V
7     return
8 .end method
```

The first line marks the beginning of the method, called `write`. It takes a single integer argument indicated by the `(I)` and returns no result, indicated by the `V` (for void). Since the method has only one argument, we only need a single local variable (Line 2) and a stack with two cells will be sufficient (Line 3). Line 4 instructs the JVM to get the value of the member `out` of the class `java/lang/System`. It expects the value to be of type `java/io/PrintStream`. A reference to this value will be placed on the stack.[1] Line 5 copies the integer we want to print out onto the stack. In the line after that we call the method `println` (from the class `java/io/PrintStream`). We want to print out an integer and do not expect anything back (that is why the type annotation is `(I)V`). The `return`-instruction in the next line changes the control-flow back to the place from where `write` was called. This method needs to be part of a header that is included in any code we generate. The helper-method `write` can be invoked with the two instructions

```
iload E(x)
invokestatic XXX/XXX/write(I)V
```

where we first place the variable to be printed on top of the stack and then call `write`. The `XXX` need to be replaced by an appropriate class name (this will be explained shortly).

By generating code for a WHILE-program, we end up with a list of (JVM assembly) instructions. Unfortunately, there is a bit more boilerplate code needed before these instructions can be run. Essentially we have to enclose them inside a Java `main`-method. The corresponding code is shown in Figure 1. This boilerplate code is very specific to the JVM. If we target any other virtual machine or a machine language, then we would need to change this code. Interesting are the Lines 5 and 6 where we hardwire that the stack of our programs will never be larger than 200 and that the maximum number of variables is also 200. This seem to be conservative default values that allow is to run some simple WHILE-programs. In a real compiler, we would of course need to work harder and find out appropriate values for the stack and local variables.

[1] Note the syntax `L …;` for the `PrintStream` type is not an typo. Somehow the designers of Jasmin decided that this syntax is pleasing to the eye. So if you wanted to have strings in your Jasmin code, you would need to write `Ljava/lang/String;` . If you want arrays of one dimension,

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

.method public static main([Ljava/lang/String;)V
    .limit locals 200
    .limit stack 200

      ...here comes the compiled code...

    return
.end method
```

Figure 1: The boilerplate code needed for running generated code. It hardwires limits for stack space and for the number of local variables.

To sum up, in Figure 2 is the complete code generated for the slightly nonsensical program

```
x := 1 + 2;
write x
```

I let you read the code and make sure the code behaves as expected. Having this code at our disposal, we need the assembler to translate the generated code into JVM bytecode (a class file). This bytecode is then understood by the JVM and can be run by just invoking the `java`-program. Again I let you do the work.

Arrays

Maybe a useful addition to the WHILE-language would be arrays. This would allow us to generate more interesting WHILE-programs by translating BF*** programs into equivalent WHILE-code. Therefore in this section let us have a look at how we can support the following three constructions

```
new(arr[15000])
x := 3 + arr[3 + y]
arr[42 * n] := ...
```

The first construct is for creating new arrays. In this instance the name of the array is `arr` and it can hold 15000 integers. We do not support "dynamic" arrays, that is the size of our arrays will always be fixed. The second construct is for referencing an array cell inside an arithmetic expression—we need to be able to look up the contents of an array at an index determined by an arithmetic

then use `[...`; two dimensions, use `[[...` and so on. Looks all very ugly to my eyes.

Figure 2: The generated code for the test program `x := 1 + 2; write x`. This code can be processed by a Java assembler producing a class-file, which can then be run by the `java`-program.

expression. Similarly in the line below, we need to be able to update the content of an array at an calculated index.

For creating a new array we can generate the following three JVM instructions:

```
ldc number
newarray int
astore loc_var
```

First we need to put the size of the array onto the stack. The next instruction creates the array. In this case the array contains `int`s. With the last instruction we can store the array as a local variable (like the "simple" variables from the previous section). The use of a local variable for each array allows us to have multiple arrays in a WHILE-program. For looking up an element in an array we can use the following JVM code

```
aload loc_var
index_aexp
iaload
```

The first instruction loads the "pointer", or local variable, to the array onto the stack. Then we have some instructions calculating the index where we want to look up the array. The idea is that these instructions will leave a concrete number on the top of the stack, which will be the index into the array we need. Finally we need to tell the JVM to load the corresponding element onto the stack. Updating an array at an index with a value is as follows.

```
aload loc_var
index_aexp
value_aexp
iastore
```

Again the first instruction loads the local variable of the array onto the stack. Then we have some instructions calculating the index where we want to update the array. After that come the instructions for with which value we want to update the array. The last line contains the instruction for updating the array.

Next we need to modify our grammar rules for our WHILE-language: it seems best to extend the rule for factors in arithmetic expressions with a rule for looking up an array.

$$
\begin{array}{lcl}
E & ::= & T + E \;\mid\; T - E \;\mid\; T \\
T & ::= & F * T \;\mid\; F \setminus T \;\mid\; F \\
F & ::= & (E) \;\mid\; \underbrace{Id[E]}_{new} \;\mid\; Id \;\mid\; Num
\end{array}
$$

There is no problem with left-recursion as the E is "protected" by an identifier and the brackets. There are two new rules for statements, one for creating an array and one for array assignment:

$$
\begin{array}{rcl}
\textit{Stmt} & ::= & \ldots \\
& | & \texttt{new}(\textit{Id}\,[\,\textit{Num}\,]) \\
& | & \textit{Id}\,[\,\textit{E}\,] := \textit{E}
\end{array}
$$

With this in place we can turn back to the idea of creating WHILE-programs by translating BF programs. This is a relatively easy task because BF has only eight instructions (we will actually implement seven because we can omit the read-in instruction from BF). What makes this translation easy is that BF-loops can be straightforwardly represented as while-loops. The Scala code for the translation is as follows:

```
1  def instr(c: Char) : String = c match {
2    case '>' => "ptr := ptr + 1;"
3    case '<' => "ptr := ptr - 1;"
4    case '+' => "mem[ptr] := mem [ptr] + 1;"
5    case '-' => "mem [ptr] := mem [ptr] - 1;"
6    case '.' => "x := mem [ptr]; write x;"
7    case '['  => "while (mem [ptr] != 0) do {"
8    case ']'  => "skip};"
9    case _ => ""
10 }
```

The idea behind the translation is that BF-programs operate on an array, called here `mem`. The BF-memory pointer into this array is represented as the variable `ptr`. As usual the BF-instructions > and < increase, respectively decrease, `ptr`. The instructions + and - update a cell in `mem`. In Line 6 we need to first assign a `mem`-cell to an auxiliary variable since we have not changed our write functions in order to cope with writing out any array-content directly. Lines 7 and 8 are for translating BF-loops. Line 8 is interesting in the sense that we need to generate a `skip` instruction just before finishing with the closing `"}"`. The reason is that we are rather pedantic about semicolons in our WHILE-grammar: the last command cannot have a semicolon—adding a `skip` works around this snag.

Putting this all together and we can generate WHILE-programs with more than 15K JVM-instructions; run the compiled JVM code for such programs and marvel at the output…

…Hoooooray, after a few more tweaks we can finally run the BF-mandelbrot program on the JVM (after nearly 10 minutes of parsing the corresponding WHILE-program; the size of the resulting class file is around 32K—not too bad). The generation of the picture completes within 20 or so seconds. Try replicating this with an interpreter! The good point is that we now have a sufficiently complicated program in our WHILE-language in order to do some benchmarking. Which means we now face the question about what to do next…

Optimisations & Co

Every compiler that deserves its name performs optimisations on the code. If we make the extra effort of writing a compiler for a language, then obviously we want to have our code to run as fast as possible. So we should look into this.

There is actually one aspect in our generated code where we can make easily efficiency gains: this has to do with some of the quirks of the JVM. Whenever we push a constant onto the stack, we used the JVM instruction `ldc some_const`. This is a rather generic instruction in the sense that it works not just for integers but also for strings, objects and so on. What this instruction does is putting the constant into a *constant pool* and then to use an index into this constant pool. This means `ldc` will be represented by at least two bytes in the class file. While this is sensible for "large" constants like strings, it is a bit of overkill for small integers (which many integers will be when compiling a BF-program). To counter this "waste", the JVM has specific instructions for small integers, for example

- `iconst_0`,..., `iconst_5`
- `bipush n`

where the `n` is `bipush` is between -128 and 128. By having dedicated instructions such as `iconst_0` to `iconst_5` (and `iconst_m1`), we can make the generated code size smaller as these instructions only require 1 byte (as opposed the generic `ldc` which needs 1 byte plus another for the index into the constant pool). While in theory the use of such special instructions should make the code only smaller, it actually makes the code also run faster. Probably because the JVM has to process less code and uses a specific instruction in the underlying CPU. The story with `bipush` is slightly different, because it also uses two bytes—so it does not result in a reduction in code size. But againm, it probably uses a specific instruction in the underlying CPU that make the JVM code run faster. This means when generating code we can use the following helper function

```
def compile_num(i: Int) =
  if (0 <= i && i <= 5) i"iconst_$i" else
  if (-128 <= i && i <= 127) i"bipush $i" else i"ldc $i"
```

that generates the more efficient instructions for pushing a constant onto the stack. Note the JVM also has special instructions that load and store the first three local variables. The assumption is that most operations and arguments in a method will only use very few local variables. So the JVM has the following instructions:

- `iload_0`,..., `iload_3`
- `istore_0`,..., `istore_3`
- `aload_0`,..., `aload_3`
- `astore_0`,..., `astore_3`

Having implemented these optimisations, the code size of the BF-Mandelbrot program reduces and also it runs faster. According to my very rough experiments:

	class-size	runtime
Mandelbrot:		
unoptimised:	33296	21 secs
optimised:	21787	16 secs

Quite good! Such optimisations are called *peephole optimisations*, because it is type of optimisations that involve changing a small set of instructions into an equivalent set that has better performance.

If you look careful at our code you will quickly find another source of inefficiency in programs like

```
x := ...;
write x
```

where our code first calculates the new result the for `x` on the stack, then pops off the result into a local variable, and after that loads the local variable back onto the stack for writing out a number. If we can detect such situations, then we can leave the value of `x` on the stack with for example the much cheaper instruction `dup`. Now the problem with this optimisation is that it is quite easy for the snippet above, but what about instances where there is further WHILE-code in *between* these two statements? Sometimes we will be able to optimise, sometimes we will not. The compiler needs to find out which situation applies. This can become quickly much more complicated. So we leave this kind of optimisations here and look at something more interesting and possibly surprising.

As you have probably seen, the compiler writer has a lot of freedom about how to generate code from what the programmer wrote as program. The only condition is that generated code should behave as expected by the programmer. Then all is fine…mission accomplished! But sometimes the compiler writer is expected to go an extra mile, or even miles and change the meaning of a program in unexpected ways. Suppose we are given the following WHILE-program:

```
new(arr[10]);
arr[14] := 3 + arr[13]
```

Admittedly this is a contrived program, and probably not meant to be like this by any sane programmer, but it is supposed to make the following point: We generate an array of size 10, and then try to access the non-existing element at index 13 and even updating element with index 14. Obviously this is baloney. Still, our compiler generates code for this program without any questions asked. We can even run this code on the JVM…of course the result is an exception trace where the JVM yells at us for doing naughty things. (This is much better than C, for example, where such errors are not prevented and as a result insidious attacks can be mounted against such kind C-programs. I

assume everyone has heard about *Buffer Overflow Attacks*.) Now what should we do in such situations? Index over- or underflows are notoriously difficult to detect statically (at compiletime), so it seem raising an exception at run-time like the JVM is the best compromise.

Well, imagine we do not want to rely in our compiler on the JVM for producing an annoying, but safe exception trace, rather we want to handle such situations ourselves according to what we thing should happen in such cases. Let's assume we want to handle them in the following way: if the programmer access a field out-of-bounds, we just return a default 0, and if a programmer wants to update an out-of-bounds field, we want to "quietly" ignore this update.

arraylength