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 enough to improve the speed of a program by just targeting a virtual machine. This produces not the fastest possible code, but code that is fast enough and has the advantage that the virtual machine takes care of things a compiler would normally need to take care of (like explicit memory management). Why study compilers? John Regher gives this answer in his compiler blog:[1]

> *"We can start off with a couple of observations about the role of compilers. First, hardware is getting weirder rather than getting clocked faster: almost all processors are multicores and it looks like there is increasing asymmetry in resources across cores. Processors come with vector units, crypto accelerators, bit twiddling instructions, and lots of features to make virtualization and concurrency work. We have DSPs, GPUs, big.little, and Xeon Phi. This is only scratching the surface. Second, we're getting tired of low-level languages and their associated security disasters, we want to write new code, to whatever extent possible, in safer, higher-level languages. Compilers are caught right in the middle of these opposing trends: one of their main jobs is to help bridge the large and growing gap between increasingly high-level languages and increasingly wacky platforms. It's effectively a perpetual employment act for solid compiler hackers."*

As a first example in this module we will implement a compiler for the very simple While-language. It will generate code for the Java Virtual Machine (JVM). This is a stack-based virtual machine, a fact which will make it easy to generate code for arithmetic expressions. For example for generating code for 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 2, the third instruction adds both numbers together replacing the top two elements of the stack with the result 3. For simplicity, we will throughout consider only integer numbers and results. Therefore we can use the JVM instructions `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).

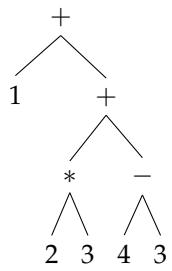
Recall our grammar for arithmetic expressions (E is the starting symbol):

$$\begin{array}{lcl} \langle E\rangle & ::= & \langle T\rangle + \langle E\rangle \;\mid\; \langle T\rangle - \langle E\rangle \;\mid\; \langle T\rangle \\ \langle T\rangle & ::= & \langle F\rangle * \langle T\rangle \;\mid\; \langle F\rangle \setminus \langle T\rangle \;\mid\; \langle F\rangle \end{array}$$

[1] http://blog.regehr.org/archives/1419

$$\langle F \rangle ::= (\ \langle E \rangle\ ) \ \mid\ \langle Id \rangle \ \mid\ \langle Num \rangle$$

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



To generate 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 (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 a convention we always observe in our compiler, that is the results of arithmetic expressions will always be on top of the stack. Note, that a different bracketing of the expression, for example $(1 + (2 * 3)) + (4 - 3)$, produces a different abstract syntax tree and thus potentially also a different list of instructions. Generating code in this fashion is rather easy to implement: it can be done with the following recursive *compile*-function, which takes the abstract syntax tree as argument:

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

However, our arithmetic expressions can also contain variables. We will represent them as *local variables* in 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, 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 the environment, E, that maps identifiers to index-numbers.

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

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.

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

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

$[]$ 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, we need to be able to retrieve the associated index. This is reflected in the clause for compiling assignments:

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

We first generate code for the right-hand side of the assignment and then add an `istore`-instruction at the end. By convention the result of the arithmetic expression a will be on top of the stack. After 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 $E' = E(x \mapsto \mathit{largest\_index} + 1)$). That means for the assignment $x := x + 1$ we generate the following code

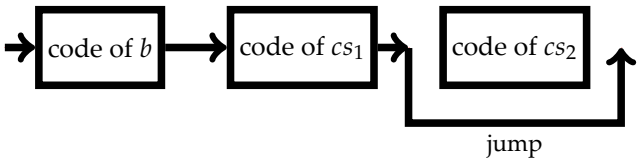
```
iload n_x
ldc 1
iadd
istore n_x
```

where n_x is the index for the variable x.

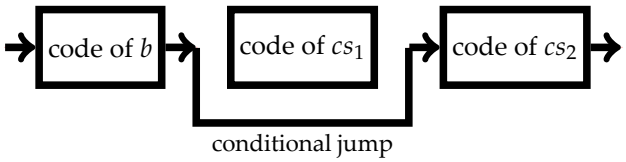
More complicated is the code for `if`-statments, say

```
if b then cs_1 else cs_2
```

where b is a boolean expression and the $cs_{1/2}$ are the statements for each `if`-branch. Lets assume we already generated code for b and $cs_{1/2}$. Then in the true-case the control-flow of the program needs to be



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 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 a label is indicated by a colon.

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
```

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. 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. 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 way of generating code still seems the most convenient.

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

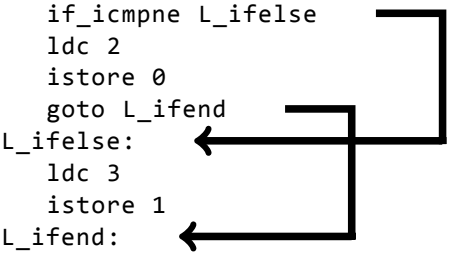
$$
\begin{array}{l}
\mathit{compile}(\texttt{if}\ b\ \texttt{then}\ cs_1\ \texttt{else}\ cs_2, E) \stackrel{\text{def}}{=} \\
\quad L_{\mathit{ifelse}}\ \text{(fresh label)} \\
\quad L_{\mathit{ifend}}\ \text{(fresh label)} \\
\quad (is_1, E') = \mathit{compile}(cs_1, E) \\
\quad (is_2, E'') = \mathit{compile}(cs_2,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

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
```

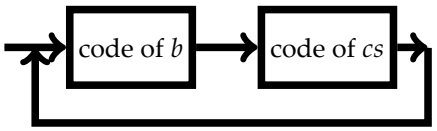
The generated code is as follows:

```
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:
```

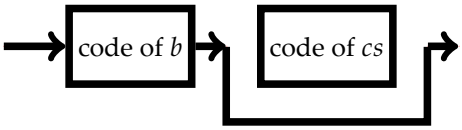


The first three lines correspond to the the boolean expression $1 = 1$. The jump for when this boolean expression is false is in Line 3. Lines 4-6 corresponds to the if-branch; the else-branch is in Lines 8 and 9. Note carefully how the environment E is threaded through the recursive calls of *compile*. 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.



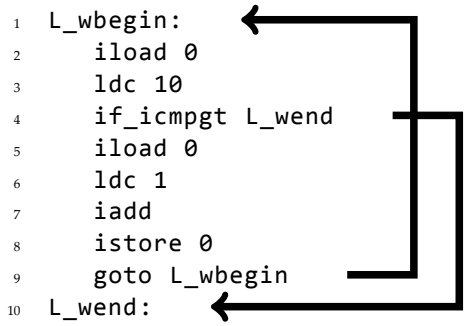
Again we can use the *compile*-function for boolean expressions to insert the appropriate jump to the end of the loop (label L_{wend} below).

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

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:
```

Next we need to consider the statement `write x`, which can be used to print out the content of a variable. For this we need to 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 argument (which 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`. 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 field `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. Line 5 copies the integer we want to print out onto the stack. In the next line 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

```
1 iload E(x)
2 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. The complete 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. Lines 4 to 8 in Figure 1 contain a method for object creation in the JVM; this method is called *before* the `main`-method in Lines 10 to 17. Interesting are the Lines 11 and 12 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.

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

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

.method public <init>()V
    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return
.end method

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

      …here comes the compiled code…

    return
.end method
```

Figure 1: Boilerplate code needed for running generated code.

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

Having this code at our disposal, we need the assembler to translate the generated code into JVM bytecode (a class file). This bytecode is understood by the JVM and can be run by just invoking the `java`-program.

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

.method public <init>()V
   aload_0
   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

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

.method public static main([Ljava/lang/String;)V
   .limit locals 200
   .limit stack 200
   ldc 1
   ldc 2
   iadd
   istore 0
   iload 0
   invokestatic test/test/write(I)V
   return
.end method
```

Figure 2: Generated code for a test program. This code can be processed by an Java assembler producing a class-file, which can be run by the `java`-program.