Handout 7 (Compilation)

The purpose of a compiler is to transform a program, a human can 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 care of things a compiler would normally need to take care of (like explicit memory management).

As an example we will implement a compiler for the very simple While-language. We will be generating code for the Java Virtual Machine. 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

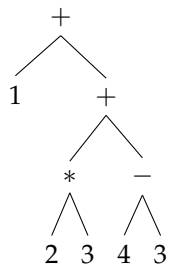
```
ldc 1
ldc 2
iadd
```

The first instruction loads the constant 1 onto the stack, the next one 2, the third instruction adds both numbers together replacing the top elements of the stack with the result 3. For simplicity, we will throughout consider only integer numbers and results. Therefore we can use the 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):

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

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 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 *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.

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

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.

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 explain first the compilation of statements of the While-language. The clause for `skip` is trivial, since we do not have to generate any instruction

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

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:

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

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

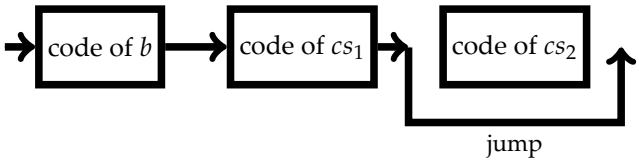
```
iload n_x
ldc 1
iadd
istore n_x
```

where n_x is the index for the variable x.

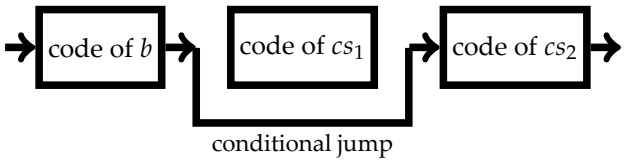
More complicated is the code for `if`-statments, say

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

where b is a boolean expression and the cs_i are the instructions 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 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 addresses, but need to attach 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. A labels, say `L`, is attached to code like

```
L:
  instr_1
  instr_2
    ⋮
```

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:

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

We are 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 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

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

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—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.

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:

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