Handout 7 (Compilation of the WHILE-Language)

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).

We will be generating code for the Java Virtual Machine. This is a stack-based virtual machine which will make it easy to generate code for arithmetic expressions. For example for generating code for the expression $1 + 2$ we need to issue the following three instructions

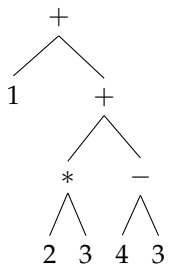
```
ldc 1
ldc 2
iadd
```

The first instruction loads the constant 1 on the stack, the next one 2, the third instruction add both numbers together replacing the top elements of the stack with the result 3. 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 (alternatives are `d` for doubles, `l` for longs and `f` for floats).

Recall our grammar for arithmetic expressions:

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

where $\langle Id\rangle$ stands for variables and $\langle Num\rangle$ for number. 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—this will produce code for a stack-machine, like the JVM. Doing so gives 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. This will be a convention we always observe, namely that the results of arithmetic expressions will always be on top of the stack. Note, that a different bracketing, 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 simple: it can be done by the following *compile*-function:

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

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 the memory containing in our case only integers. Looking up a variable can be done by the instruction

```
iload index
```

which places the content of the local variable *index* onto thestack. 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 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 (similar to the interpreter) 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 result will usually be an erroneous result. Therefore our *compile*-function will 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 to which index the variable x maps to.