# Handout 8 (A Functional Language)

The language we looked at in the previous lecture was rather primitive and the compiler rather crude—everything was essentially compiled into a big monolithic chunk of code inside the main function. In this handout we like to have a look at a slightly more comfortable language, which I call Fun-language, and a tiny-teeny bit more realistic compiler. The Fun-language is a small functional programming language. A small collection of programs we want to be able to write and compile is as follows:

```
def fib(n) = if n == 0 then 0
             else if n == 1 then 1
             else fib(n - 1) + fib(n - 2);

def fact(n) = if n == 0 then 1 else n * fact(n - 1);

def ack(m, n) = if m == 0 then n + 1
                else if n == 0 then ack(m - 1, 1)
                else ack(m - 1, ack(m, n - 1));

def gcd(a, b) = if b == 0 then a else gcd(b, a % b);
```

Compare the code of the fib-program with the same program written in the While-language…Fun is definitely more comfortable. We will still focus on programs involving integers only, that means for example that every function in Fun is expected to return an integer. The point of the Fun language is to compile each function to a separate method in JVM bytecode (not just a big monolithic code chunk). The means we need to adapt to some of the conventions of the JVM about methods.

The grammar of the Fun-language is slightly simpler than the While-language, because the main syntactic category are expressions (we do not have statements). The grammar rules are as follows:

$$
\begin{array}{rcl}
\boldsymbol{Exp} & ::= & \boldsymbol{Id} \;|\; \boldsymbol{Num}\\
 & | & \boldsymbol{Exp} + \boldsymbol{Exp} \;|\; \ldots \;|\; (\boldsymbol{Exp})\\
 & | & \texttt{if}\; \boldsymbol{BExp}\; \texttt{then}\; \boldsymbol{Exp}\; \texttt{else}\; \boldsymbol{Exp}\\
 & | & \texttt{write}\; \boldsymbol{Exp}\\
 & | & \boldsymbol{Exp}\; ;\; \boldsymbol{Exp}\\
 & | & \mathit{FunName}\; (\boldsymbol{Exp}, \ldots, \boldsymbol{Exp})\\
\boldsymbol{BExp} & ::= & \ldots\\
\boldsymbol{Decl} & ::= & \boldsymbol{Def}\; ;\; \boldsymbol{Decl} \;|\; \boldsymbol{Exp}\\
\boldsymbol{Def} & ::= & \texttt{def}\; \mathit{FunName}\; (x_1, \ldots, x_n) = \boldsymbol{Exp}
\end{array}
$$

where, as usual, ***Id*** stands for variables and ***Num*** for numbers. We can call a function by applying the arguments to a function name (as shown in the last clause of ***Exp***). The arguments in such a function call can be again expressions, including other function calls. In contrast, when defining a function (see ***Def***-clause) the arguments need to be variables, say $x_1$ to $x_n$. We call the expression on the right of = in a function definition as the *body of the function*. We have the restriction that the variables inside a function body can only be those that are mentioned as arguments of the function. A Fun-program is then a sequence of function definitions separated by semicolons, and a final "main" call of a function that starts the computation in the program. For example

```
def fact(n) = if n == 0 then 1 else n * fact(n - 1);
write(fact(5))
```

would be a valid Fun-program. The parser of the Fun-language produces abstract syntax trees which in Scala can be represented as follows:

```
abstract class Exp
abstract class BExp
abstract class Decl

case class Var(s: String) extends Exp
case class Num(i: Int) extends Exp
case class Aop(o: String, a1: Exp, a2: Exp) extends Exp
case class If(a: BExp, e1: Exp, e2: Exp) extends Exp
case class Write(e: Exp) extends Exp
case class Sequ(e1: Exp, e2: Exp) extends Exp
case class Call(name: String, args: List[Exp]) extends Exp

case class Bop(o: String, a1: Exp, a2: Exp) extends BExp

case class Def(name: String,
               args: List[String],
               body: Exp) extends Decl
case class Main(e: Exp) extends Decl
```

Let us first look at some clauses for compiling expressions. The compilation of arithmetic and boolean expressions is just like for the While-language and does not need any modification (recall that the *compile*-function for boolean expressions takes a third argument for the label where the control-flow should jump when the boolean expression is *not* true—this is needed for compiling `ifs`). One additional feature in the Fun-language are sequences. Their purpose is to do one calculation after another or printing out an intermediate result. The reason why we need to be careful however is the convention that every expression can only produce s single result (including sequences). Since this result will be on the top of the stack, we need to generate a pop-instruction for sequences in order to clean-up the stack. For example, for an expression of the form `exp1 ; exp2` we need to generate code where after the first code chunk a pop-instruction is needed.

```
.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
```

Figure 1: The helper function for printing out integers.

```
compile(exp1)
pop
compile(exp2)
```

In effect we "forget" about the result the first expression calculates. I leave you to think about why this sequence operator is still useful in the Fun-language, even if the first result is just "discarded".

There is also one small modification we have to perform when calling the write method. Remember in the Fun-language we have the convention that every expression needs to return an integer as a result (located on the top of the stack). Our helper function implementing write, however, "consumes" the top element of the stack and violates this convention. Therefore before we call, say, `write(1+2)`, we need to duplicate the top element of the stack like so

```
compile(1+2)
dup
invokestatic XXX/XXX/write(I)V
```

We also need to first generate code for the argument-expression of write, which in the While-language was only allowed to be a single variable.

Most of the new code in the compiler for the Fun-language comes from function definitions and function calls. For this have a look again at the helper function in Figure 1. Assuming we have a function definition

```
def fname (x1, ... , xn) = ...
```

then we have to generate

```
.method public static fname (I...I)I
   .limit locals ??
   .limit stack ??
   ...
  ireturn
.method end
```

where the number of `I`s corresponds to the number of arguments the function has, say `x1` to `xn`. The final `I` is needed in order to indicate that the function returns an integer. Therefore we also have to use `ireturn` instead of `return`. However, more interesting are the two `.limit` lines. `Locals` refers to the local variables of the method, which can be queried and overwritten using the JVM instructions `iload` and `istore`, respectively. Before we call a function with, say, three arguments, we need to ensure that these three arguments are pushed onto the stack (we will come to the corresponding code shortly). Once we are inside the method, the arguments on the stack turn into local variables. So in case we have three arguments on the stack, we will have inside the function three local variables that can be referenced by the indices 0..2. Determining the limit for local variables is the easy bit. Harder is the stack limit.

Calculating how much stack a program needs is equivalent to the Halting problem, and thus undecidable in general. Fortunately, we are only asked how much stack a *single* call of the function requires. This can be relatively easily compiled by recursively analysing which instructions we generate and how much stack they might require.

$$
\begin{array}{lcl}
\mathit{estimate}(n) & \overset{\mathrm{def}}{=} & 1 \\
\mathit{estimate}(x) & \overset{\mathrm{def}}{=} & 1 \\
\mathit{estimate}(a_1\ \mathit{aop}\ a_2) & \overset{\mathrm{def}}{=} & \mathit{estimate}(a_1) + \mathit{estimate}(a_2) \\
\mathit{estimate}(\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & \overset{\mathrm{def}}{=} & \mathit{estimate}(b) + \\
 & & \quad \mathit{max}(\mathit{estimate}(e_1), \mathit{estimate}(e_2)) \\
\mathit{estimate}(\texttt{write}(e)) & \overset{\mathrm{def}}{=} & \mathit{estimate}(e) + 1 \\
\mathit{estimate}(e_1; e_2) & \overset{\mathrm{def}}{=} & \mathit{max}(\mathit{estimate}(e_1), \mathit{estimate}(e_2)) \\
\mathit{estimate}(f(e_1, ..., e_n)) & \overset{\mathrm{def}}{=} & \sum_{i=1..n}\ \mathit{estimate}(e_i) \\
\mathit{estimate}(a_1\ \mathit{bop}\ a_2) & \overset{\mathrm{def}}{=} & \mathit{estimate}(a_1) + \mathit{estimate}(a_2)
\end{array}
$$

This function overestimates the stack size, for example, in the case of `if`s. Since we cannot predict which branch will be run, we have to allocate the maximum of stack each branch might take. I leave you also to think about whether the estimate in case of function calls is the best possible estimate. Note also that in case of `write` we need to add one, because we duplicate the top-most element in the stack.

With this all in place, we can start generating code, for example, for the two functions:

```
def suc(x) = x + 1;

def add(x, y) = if x == 0 then y
                else suc(add(x - 1, y));
```

The successor function is a simple loading of the argument $x$ (index 0) onto the stack, as well as the number 1. Then we add both elements leaving the result of the addition on top of the stack. This value will be returned by the `suc`-function. See below:

```
.method public static suc(I)I
.limit locals 1
.limit stack 2
  iload 0
  ldc 1
  iadd
  ireturn
.end method
```

The addition function is a bit more interesting since in the last line we have to call the function recursively and “wrap around” a call to the successor function. The code is as follows:

```
.method public static add(II)I
.limit locals 2
.limit stack 5
  iload 0
  ldc 0
  if_icmpne If_else
  iload 1
  goto If_end
If_else:
  iload 0
  ldc 1
  isub
  iload 1
  invokestatic XXX/XXX/add(II)I
  invokestatic XXX/XXX/suc(I)I
If_end:
  ireturn
.end method
```

The locals limit is 2 because add takes two arguments. The stack limit is a simple calculation using the estimate function. We first generate code for the boolean expression `x == 0`, that is loading the local variable 0 and the number 0 onto the stack (Lines 4 and 5). If the not-equality test fails, we continue with returning $y$, which is the local variable 1 (followed by a jump to the return instruction). If the not-equality test succeeds, then we jump to the label `If_else` (Line 9). After that label is the code for `suc(add(x - 1, y))`. We first have to evaluate the argument of the suc-function. But this means we first have to evaluate the two arguments of the add-function. This means loading $x$ and 1 onto the stack and subtracting them. Then loading $y$ onto the stack. We can then make a recursive call to add (its two arguments are on the stack). When this call returns we have the result of the addition on the top of the stack and just need to call suc. Finally, we can return the result on top of the stack as the result of the add-function.

## Tail-Call Optimisations

Let us now briefly touch upon the vast topic of compiler optimisations. As an example, let's perform tail-call optimisations for our Fun-language. Consider the following version of the factorial function:

```
def facT(n, acc) =
  if n == 0 then acc
  else facT(n - 1, n * acc);
```

The corrsponding JVM code for this function is below:

```
1  .method public static facT(II)I
2  .limit locals 2
3  .limit stack 6
4    iload 0
5    ldc 0
6    if_icmpne If_else_2
7    iload 1
8    goto If_end_3
9  If_else_2:
10   iload 0
11   ldc 1
12   isub
13   iload 0
14   iload 1
15   imul
16   invokestatic fact/fact/facT(II)I
17 If_end_3:
18   ireturn
19 .end method
```

The interesting part is in Lines 10 to 16. Since the `facT` function is recursive, we have also a recursive call in Line 16 in the JVM code. The problem is that before we can make the recursive call, we need to put the two arguments, namely `n - 1` and `n * acc`, onto the stack. That is how we communicate arguments to a function. To see the the difficulty, imagine you call this function 1000 times recursively. Each call results in some hefty overhead on the stack—ultimately leading to a stack overflow. Well, it is possible to avoid this overhead completely in many circumstances. This is what tail-call optimisations are about.

Note that the call to `facT` in the program is the last instruction before the `ireturn` (the label in Line 17 does not count since it is not an instruction). Also remember, before we make the recursive call the arguments of `facT` need to put on the stack. Once we are "inside" the arguments on the stack turn into local variables. Therefore `n` and `acc` are referenced inside the function with `iload 0` and `iload 1` respectively.

The idea of tail-call optimisation is to eliminate the expensive recursive functions call and replace it by a simple jump back to the beginning of the function.

To make this work we have to change how we communicate the arguments to the next level of "recursion": we cannot use the stack, but have to load the argument into the corresponding local variables. This gives the following code

```
1  .method public static facT(II)I
2  .limit locals 2
3  .limit stack 6
4  facT_Start:
5    iload 0
6    ldc 0
7    if_icmpne If_else_2
8    iload 1
9    goto If_end_3
10 If_else_2:
11   iload 0
12   ldc 1
13   isub
14   iload 0
15   iload 1
16   imul
17   istore 1
18   istore 0
19   goto facT_Start
20 If_end_3:
21   ireturn
22 .end method
```

In Line 4 we introduce a label for indicating where the start of the function is. Important are Lines 17 and 18 where we store the values from the stack into local variables. When we then jump back to the beginning of the function (in Line 19) it will look like to the function as if the function had been called the normal way via values on the stack. But because of the jump, clearly, no memory on the stack is needed. In effect we replaced a recursive call with a simple loop.

Can this optimisation always be applied? Unfortunately not. The recursive call needs to be in tail-call position, that is the last operation needs to be the recursive call. This is for example not the case with the usual formulation of the factorial function. Consider again the Fun-program

```
def fact(n) = if n == 0 then 1
              else n * fact(n - 1)
```

In this version of the factorial function the recursive call is *not* the last operation (which can also been seen very clearly in the generated JVM code). Because of this, the plumbing of local variables would not work and in effect the optimisation is not applicable. Very roughly speaking the tail-position of a function is in the two highlighted places

- `if Bexp then` $\boxed{\texttt{Exp}}$ `else` $\boxed{\texttt{Exp}}$
- `Exp ;` $\boxed{\texttt{Exp}}$

To sum up, the compiler needs to recognise when a recursive call is in tail-position. It then can apply the tail-call optimisations technique, which is well known and widely implemented in compilers for functional programming languages.