# Handout 9 (LLVM, SSA and CPS)

Reflecting on our tiny compiler targetting the JVM, the code generation part was actually not so hard, no? Pretty much just some post-traversal of the abstract syntax tree, yes? One of the reasons for this ease is that the JVM is a stack-based virtual machine and it is therefore not hard to translate deeply-nested arithmetic expressions into a sequence of instructions manipulating the stack. The problem is that "real" CPUs, although supporting stack operations, are not really designed to be *stack machines*. The design of CPUs is more like, here is a chunk of memory—compiler, or better compiler writers, do something with it. Consequently, modern compilers need to go the extra mile in order to generate code that is much easier and faster to process by CPUs. To make this all tractable for this module, we target the LLVM Intermediate Language. In this way we can take advantage of the tools coming with LLVM. For example we do not have to worry about things like register allocations.

LLVM[1] is a beautiful example that projects from Academia can make a difference in the world. LLVM started in 2000 as a project by two researchers at the University of Illinois at Urbana-Champaign. At the time the behemoth of compilers was gcc with its myriad of front-ends for other languages (C++, Fortran, Ada, Go, Objective-C, Pascal etc). The problem was that gcc morphed over time into a monolithic gigantic piece of m…ehm software, which you could not mess about in an afternoon. In contrast, LLVM is designed to be a modular suite of tools with which you could play around easily and try out something new. LLVM became a big player once Apple hired one of the original developers (I cannot remember the reason why Apple did not want to use gcc, but maybe they were also just disgusted by its big monolithic codebase). Anyway, LLVM is now the big player and gcc is more or less legacy. This does not mean that programming languages like C and C++ are dying out any time soon—they are nicely supported by LLVM.

We will target the LLVM Intermediate Language, or Intermediate Representation (short LLVM-IR). As a result we can benefit from the modular structure of the LLVM compiler and let for example the compiler generate code for different CPUs, like X86 or ARM. That means we can be agnostic about where our code actually runs. We can also be ignorant about optimising code and allocating memory efficiently. However, what we have to do is to generate code in *Static Single-Assignment* format (short SSA), because that is what the LLVM-IR expects from us.

The idea behind the SSA format is to use very simple variable assignments where every variable is assigned only once. The assignments also need to be primitive in the sense that they can be just simple operations like addition, multiplication, jumps, comparisons and so on. Say, we have an expression $((1 + a) + (3 + (b * 5)))$, then the SSA is

[1] http://llvm.org

```
1 let tmp0 = add 1 a in
2 let tmp1 = mul b 5 in
3 let tmp2 = add 3 tmp1 in
4 let tmp3 = add tmp0 tmp2 in
5   tmp3
```

where every variable is used only once (we could not write `tmp1 = add 3 tmp1` in Line 3 for example). There are sophisticated algorithms for imperative languages, like C, that efficiently transform a high-level program into SSA format. But we can ignore them here. We want to compile a functional language and there things get much more interesting than just sophisticated. We will need to have a look at CPS translations, where the CPS stands for Continuation-Passing-Style—basically black programming art or abracadabra programming. So sit tight.

## LLVM-IR

Before we start, lets first have a look at the *LLVM Intermediate Representation* in more detail. What is good about our toy Fun language is that it basically only contains expressions (be they arithmetic expressions or boolean expressions or if-expressions). The exception are function definitions. Luckily, for them we can use the mechanism of defining functions in LLVM-IR. For example the simple Fun program

```
def sqr(x) = x * x
```

can be compiled into the following LLVM-IR function:

```
define i32 @sqr(i32 %x) {
   %tmp = mul i32 %x, %x
   ret i32 %tmp
}
```

First notice that all variable names in the LLVM-IR are prefixed by %; function names need to be prefixed with @-symbol. Also, the LLVM-IR is a fully typed language. The `i32` type stands for 32-bit integers. There are also types for 64-bit integers (`i64`), chars (`i8`), floats, arrays and even pointer types. In the code above, `sqr` takes an argument of type `i32` and produces a result of type `i32` (the result type is before the function name, like in C). Each arithmetic operation, like addition or multiplication, are also prefixed with the type they operate on. Obviously these types need to match up... but since we have in our programs only integers, `i32` everywhere will do.

Conveniently, you can use the program `lli`, which comes with LLVM, to interpret programs written in the LLVM-IR. So you can easily check whether the code you produced actually works. To get a running program that does something interesting you need to add some boilerplate about printing out numbers and a main-function that is the entrypoint for the program (see Figure 1

```
1  @.str = private constant [4 x i8] c"%d\0A\00"
2
3  declare i32 @printf(i8*, ...)
4
5  ; prints out an integer
6  define i32 @printInt(i32 %x) {
7     %t0 = getelementptr [4 x i8], [4 x i8]* @.str, i32 0, i32 0
8     call i32 (i8*, ...) @printf(i8* %t0, i32 %x)
9     ret i32 %x
10 }
11
12 ; square function
13 define i32 @sqr(i32 %x) {
14    %tmp = mul i32 %x, %x
15    ret i32 %tmp
16 }
17
18 ; main
19 define i32 @main() {
20   %1 = call i32 @sqr(i32 5)
21   %2 = call i32 @printInt (i32 %1)
22   ret i32 %1
23 }
```

Figure 1: An LLVM-IR program for calculating the square function. The interesting function is `sqr` in Lines 13 – 16. The main function calls `sqr` and prints out the result. The other code is boilerplate for printing out integers.

for a complete listing). You can generate a binary for this program using `llc`-compiler and `gcc`—`llc` can generate an object file and then you can use `gcc` (that is clang) for generating an executable binary:

```
llc -filetype=obj sqr.ll
gcc sqr.o -o a.out
./a.out
> 25
```

## Our Own Intermediate Language

Remember compilers have to solve the problem of bridging the gap between "high-level" programs and "low-level" hardware. If the gap is too wide for one step, then a good strategy is to lay a stepping stone somewhere in between. The LLVM-IR itself is such a stepping stone to make the task of generating and optimising code easier. Like a real compiler we will use our own stepping stone which I call the *K-language*. For this remember expressions (and boolean expressions) in the Fun language. For convenience the Scala code is shown on top of Figure 2. Below is the code for the K-language. There are two kinds of syntactic entities, namely *K-values* and *K-expressions*. The central point of the K-language is the `KLet`-constructor. For this recall that arithmetic expressions such as $((1 + a) + (3 + (b * 5)))$ need to be broken up into smaller “atomic” steps, like so

```
let tmp0 = add 1 a in
let tmp1 = mul b 5 in
let tmp2 = add 3 tmp1 in
let tmp3 = add tmp0 tmp2 in
  tmp3
```

Here `tmp3` will contain the result of what the expression stands for. In each step we can only perform an “atomic” operation, like addition or multiplication. We could not for example have an if-condition on the right-hand side of an equals. These constraints enforced upon us because how the SSA format works in the LLVM-IR. By having in `KLet`, first a string (standing for an intermediate result) and second a value, we can fulfil this constraint—there is no way we could write anything else than a value. To sum up, K-values are the atomic operations that can be on the right-hand side of equal-signs. The K-language is restricted such that it is easy to generate the SSA format for the LLVM-IR.

## CPS-Translations

Another reason why it makes sense to go the extra mile is that stack instructions are very difficult to optimise—you cannot just re-arrange instructions without messing about with what is calculated on the stack. Also it is hard to find out if all the calculations on the stack are actually necessary and not by chance dead code. The JVM has for all this sophisticated machinery to make such “high-level” code still run fast, but let’s say that for the sake of argument we do not want to rely on it. We want to generate fast code ourselves. This means we have to work around the intricacies of what instructions CPUs can actually process.

```scala
// Fun-language (expressions)
abstract class Exp
abstract class BExp

case class Call(name: String, args: List[Exp]) extends Exp
case class If(a: BExp, e1: Exp, e2: Exp) extends Exp
case class Write(e: Exp) extends Exp
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 Sequence(e1: Exp, e2: Exp) extends Exp
case class Bop(o: String, a1: Exp, a2: Exp) extends BExp



// K-language (K-expressions, K-values)
abstract class KExp
abstract class KVal

case class KVar(s: String) extends KVal
case class KNum(i: Int) extends KVal
case class Kop(o: String, v1: KVal, v2: KVal) extends KVal
case class KCall(o: String, vrs: List[KVal]) extends KVal
case class KWrite(v: KVal) extends KVal

case class KIf(x1: String, e1: KExp, e2: KExp) extends KExp
case class KLet(x: String, v: KVal, e: KExp) extends KExp
case class KReturn(v: KVal) extends KExp
```

Figure 2: Abstract syntax trees for the Fun language.