# lab session 4: compiler construction
# (IR) code generation for Mini Pascal

January 8, 2018

The 4th (and last) lab session of the course *Compiler Construction* consists of extending the front end of the MiniPas compiler with a simple backend that generates compilable and executable Intermediate Representation (IR) code (quadruples in C (see lecture slides week 7). Note that it is not needed to implement the full miniPAS language, although interested students are welcome to do so. It suffices to implement a compiler for miniPAS programs that consists of only a main program (i.e. no function/procedure calls). It is actually not hard to implement function/procedures (and the corresponding calls), but it is simply more work.

I noticed that some students do not construct an abstract syntax tree (AST) in the front end part of the compiler. This is not really a problem (and it was not required), since the miniPas language is small enough to extend the actions from the semantic analyzer with extra actions that directly generate IR-code. This is a perfectly fine solution, however its drawback is that the code generator has already produced output (i.e. code) on detection of a syntax/semantic error later on. A second method would be to make a two pass-compiler: the first pass does the syntax/semantic checking. If this phase completes successfully, then a second pass can be performed to do code generation (most of the parsing/symbol table code of the first pass can then be reused). Either solution will be accepted as a valid solution.

The backend of the miniPAS compiler should produce compilable C code, so we basically build a miniPAS to minimalistic-C compiler. The code generator should not produce `while-`, `for-` or `do-while`-loops. Instead, `if-else`, jumps (i.e. `goto`s), and labels must be used. Moreover, the compiler may only produce *quadruple* assignments. You are not required to implement any optimizations (although you are allowed to implememt (some of) them). As an example of a correct translation, consider the following code in miniPAS:

```
PROGRAM euclid (input, output);
{ MiniPAS implementation of Euclid's algorithm for computing GCDs }
VAR a, b : integer;

BEGIN
  readln(a, b);   { read a and b from the keyboard }
  WHILE a <> b DO
  BEGIN
    WHILE a > b DO a := a - b;
    WHILE b > a DO b := b - a
  END;
  writeln(a)
END.
```

A possible translation in C would be (note that indentation has been added for readability, but your code generator is not required to do this):

```
#include <stdio.h>

int a, b;

int main(void) {
  scanf("%d %d", &a, &b);
```

```
 lab1: ;
  int t1 = a;
  int t2 = b;
  int t3 = t1 - t2;
  if (t3 == 0) goto lab2;
 lab3: ;
  int t4 = a;
  int t5 = b;
  int t6 = t4 - t5;
  if (t6 <= 0) goto lab4;
  int t7 = a;
  int t8 = b;
  int t9 = t7 - t8;
  a = t9;
  goto lab3;
 lab4: ;
 lab5: ;
  int t10 = b;
  int t11 = a;
  int t12 = t10 - t11;
  if (t12 <= 0) goto lab6;
  int t13 = b;
  int t14 = a;
  int t15 = t13 - t14;
  b = t15;
  goto lab4;
 lab6: ;
  goto lab1;
 lab2: ;
  printf("%d\n", a);

  return 0;
}
```

Note that we actually generate C99 (or higher) code, and not ansi C. This allows declaration of auxiliary temporary variables (like `int t1 = ..`) on the fly, which is not allowed in ansi C. Moreover, note that label declarations are directly followed by a semicolon. In fact, this declares a label, directly followed by an empty statement. The reason for this (trick) is that a statement like `lab 1:  int t1 = a;` will result in a compilation error like "`a label can only be part of a statement and a declaration is not a statement`". This is simply solved, by generating code like `lab 1:/* empty statement */; int t1 = a;` instead.