Exam Compiler Construction—4 april 2014

- Write your name, student number, number the sheets, and write the total number of sheets on the first sheet.
- You may answer the problems in Dutch or in English.
- Please read each problem fully before making it. Write neatly and carefully. If the handwriting is unreadable, or needs guessing to make something out of it, then the answer is rejected.
- The exam consists of five problems. The problems 2, 3, 4 and 5 are worth 20 points each, problem 1 is worth 10 points and you get 10 points for free (total 100 points).

1. Consider the following code fragment:

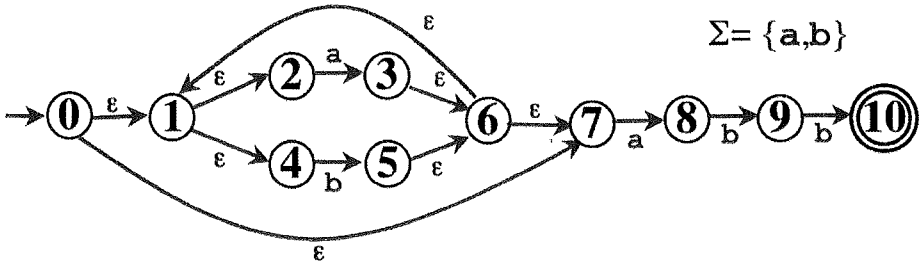
```
int x;

int g(int a, int b) {
  int c = a + b;
  /* location 2 */
  return c;
}

void f(int a, int b, int c) {
  int d = 2*a;
  int e = 3*b + c;
  /* location 1 */
  x = g(d, e);
}

int main() {
  f(15, 2, 6);
  /* location 3 */
  printf("%d\n", x);
  return 0;
}
```

The program is executed. Make a sketch of the memory layout (heap + stack of activation records) when execution of the program reaches the three marked locations. Assume that there are no optimizations at all, so parameters are not passed via registers. Moreover, on function entry/exit there is no need to save registers.

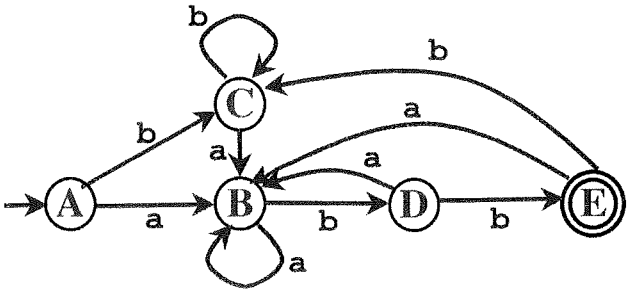
2. **(a)** Consider the language L consisting of all strings w over the alphabet $\Sigma = \{a, b, c\}$ such that w contains at least two times a symbol a, and each occurrence of a symbol b is directly followed by a symbol c.

   Give a regular expression that describes the language L, and draw a deterministic finite state automaton (DFA) that accepts L.

   (b) Consider the following nondeterministic finite state automaton (NFA).



   The above NFA is equivalent to the following deterministic finite state automaton (DFA). Show all steps of the conversion process that converts the NFA into the DFA.



3. Consider the following grammar, in which upper case letters denote the non-terminals and lower case letters the terminals:

$$
\begin{array}{lcl}
S & \rightarrow & A \\
S & \rightarrow & B \\
A & \rightarrow & a\ A \\
A & \rightarrow & a\ B \\
B & \rightarrow & B\ b \\
B & \rightarrow & c
\end{array}
$$

   (a) For all non-terminals, determine the *First* and *Follow* sets. Explain why this grammar is not LL(1).

   (b) Convert the above grammar into an equivalent LL(1) grammar.

   (c) Compute the LR(0) item sets of the given grammar.

   (d) Is the grammar LR(0), SLR(1), LR(1)? In case of conflicts be sure to identify them clearly (you do not have to solve them).

4. Given is the following LL(1) grammar for expressions (non-terminals start with an upper case letter, terminals are completely in lower case, and ε denotes the empty string):

$$
\begin{array}{lcl}
E & \rightarrow & T\ Etail \\
Etail & \rightarrow & plus\ T\ Etail \\
Etail & \rightarrow & minus\ T\ Etail \\
Etail & \rightarrow & \varepsilon \\
T & \rightarrow & F\ Ttail \\
Ttail & \rightarrow & times\ F\ Ttail \\
Ttail & \rightarrow & divide\ F\ Ttail \\
Ttail & \rightarrow & \varepsilon \\
F & \rightarrow & leftpar\ E\ rightpar \\
F & \rightarrow & number \\
F & \rightarrow & identifier
\end{array}
$$

The terminal symbols of this grammar are represented by the enumeration type `tokens`:

```
typedef enum {
  divide, identifier, minus, number, plus, times, leftpar, rightpar
} tokens;
```

A global variable `currentToken` and the function `accept` are used for interfacing with the lexer (you don't have to write the lexer, you may assume that `yylex()` exists, but you are only allowed to call it via `accept`). You can initialize `currentToken` using the function `initParser`.

```
tokens currentToken;

void initializeParser() {
  currentToken = yylex();
}

int accept(tokens tok) {
  if (currentToken == tok) {
    currentToken = yylex();
    return 1;
  }
  return 0;
}
```

There is also a void function `syntaxError` available, that prints an error message and aborts:

```
void syntaxError(){
  printf("Syntax error: abort\n");
  exit(EXIT_FAILURE);
}
```

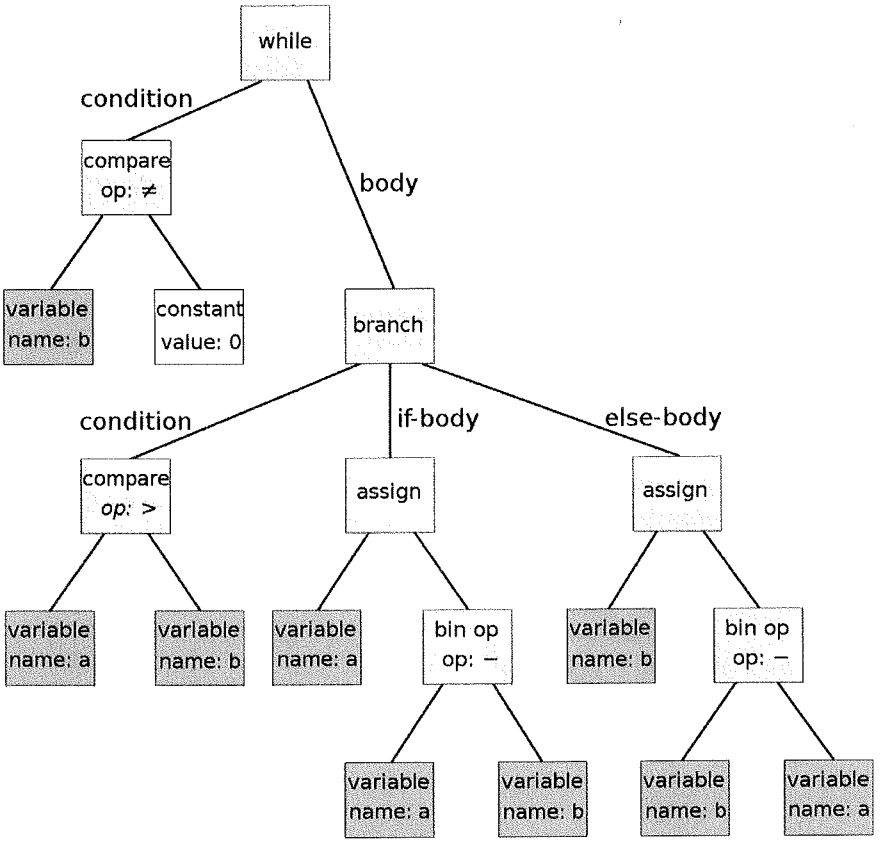
Write a *Recursive Descent Parser* for the above grammar for expressions. The parser should print an error message if it detects a syntax error (using `syntaxError`).

5. **(a)** Consider the following two code fragments (in which all variables are of type `int`):

```
b = a;                    b = a;
c = b + 1;                c = a + 1;
d = b;                    d = a;
b = d + c;                b = a + c;
b = d;                    b = a;
```

A compiler that uses *copy propagation* translates the code fragment on the left hand side into the equivalent code fragment on the right hand side. Explain step by step which actions the compiler performs during this translation.

(b) The following figure shows an Abstract Syntax Tree (AST) of the Euclidean algorithm for finding the greatest common divisor of the variables `a` and `b`.



Give (pseudo-)code for a recursive algorithm `void genIRcode(ASTtree tree)` that translates ASTs like the above AST into an intermediate code representation that uses quadruples (i.e. assignments with a left hand side, and a right hand side with at most 2 operands and an operator), and conditional jumps (`if (variable) goto label`). You do not need to specify the AST data structure: you may use pseudo-instructions like "`if node is a while-node {...}`". You may assume that all variables are of type `int`, and that there is an infinite amount of temporary variables `t0, t1, ...` that you can use without declaration.

(c) What is the code that would be generated by `genIRcode()` if we apply it to the AST of the Euclidean algorithm?