# Exercises on Compilers

**Jordi Cortadella**

February 7, 2022

## Code generation

1. Generate code for the following statements:

   - `a = b[i] + c[i]`
   - `a[i] = b*c - b*d`
   - `x = f(y+1) + 2`
   - `*(p+2) = *(q+1) + y`

   The code should be generated as any compiler would do by traversing the AST (i.e., no optimizations should be applied). You can assume that all basic data types are integers (4 bytes) and all the rhs expressions are evaluated before the lhs expressions.

   **Solution:**

```
a = b[i] + c[i]

t1 = i * 4
t2 = b[t1]
t3 = i * 4
t4 = c[t3]
t5 = t2 + t4
a = t5
```

```
a[i] = b*c - b*d

t1 = b * c
t2 = b * d
t3 = t1 - t2
t4 = i * 4
a[t4] = t3
```

```
x = f(y+1) + 2

t1 = y + 1
param t1
t2 = call f,1
t3 = t2 + 2
x = t3
```

```
*(p+2) = *(q+1) + y

t1 = 1 * 4
t2 = q + t1
t3 = *t2
t4 = t3 + y
t5 = 2 * 4
t6 = p + t5
*t6 = t4

---------------

t1 = q + 4
t2 = *t1
t3 = t2 + y
t4 = p + 8
*t4 = t3
```

2. Generate code for the following C++ function:

```
int search(int *a, int n, int x) {
    int i = 0;
    while (i < n and a[i] != x) i = i + 1;
    if (i < n) return i;
    return -1;
}
```

The code should be generated as any compiler would do by traversing the AST (i.e., no optimizations should be applied).

**Note:** Use backpatching for the evaluation of Boolean expressions.

**Solution:**

```
    t1 = 0
    i = t1
LBeginWhile:
    if i < n goto LContinueAnd
    goto LEndWhile
LContinueAnd:
    t2 = i * 4             (*1)
    t3 = a                 (*2)
    t4 = t3[t2]            (*3)
    if t4 != x goto LBodyWhile
    goto LEndWhile
LBodyWhile:
    t5 = i + 1
    i = t5
    goto LBeginWhile
LEndWhile:
    if i < n goto LBodyIf
    goto LEndIf
LBodyIf:
    return i
LEndIf:
    return -1
```

and with some code optimization:

```
    t1 = 0
    i = t1
LBeginWhile:
    if i >= n goto LEndWhile
    t2 = i * 4
    t3 = a
    t4 = t3[t2]
    if t4 = x goto LEndWhile
    t5 = i + 1
    i = t5
    goto LBeginWhile
LEndWhile:
    if i >= n goto LEndIf
    return i
LEndIf:
    return -1
```

```
(*1) a is pointer to (an array of) int's (4 bytes)
(*2) a contains the address of an array (int *a), now in t3
(*3) base address: t3,  offset: t2
```

3. Consider the following C++ function:

```
int f(vector<int>& A, double d) {
    struct {double a; int b;} s;
    vector<int> X(100);
    int i;
    ...
    L1:
    X.resize(200);
    ...
    while (i > 0) f
        int j = 0;
        vector<double> M(i);
        L2:
        ...
    }
    ...
    L3:
    return X[0];
}
```

- Describe the contents of the symbol table when compiling the code at lines L1, L2 and L3. For each entry, indicate the symbol, the type of symbol (local var, parameter), the data type, the size, and the offset in the stack.
- Describe the allocated memory (stack and heap) at the same lines during the execution of the program.
- Describe a situation in which a program could generate memory leaks, i.e., not all the allocated objects are deallocated before the completion of the program.
- Consider a C++ program that never uses the new/delete statements. Can it generate memory leaks? Reason your answer.

For the description of the symbol table, you can assume that `int` and `double` use 4 and 8 bytes, respectively, and that a `vector<T>` descriptor is a `struct` with three fields: *size* (`int`), *capacity* (`int`) and *data* (`T*`). You can assume that pointers use 8 bytes.

For the activation record of the function, you can assume that all parameters and variables are allocated in the stack and that the return address and the previous FP are stored in the region FP[0..15]. The result is stored in FP+16 and the parameters are stored after the result from last to first.

**Solution:**

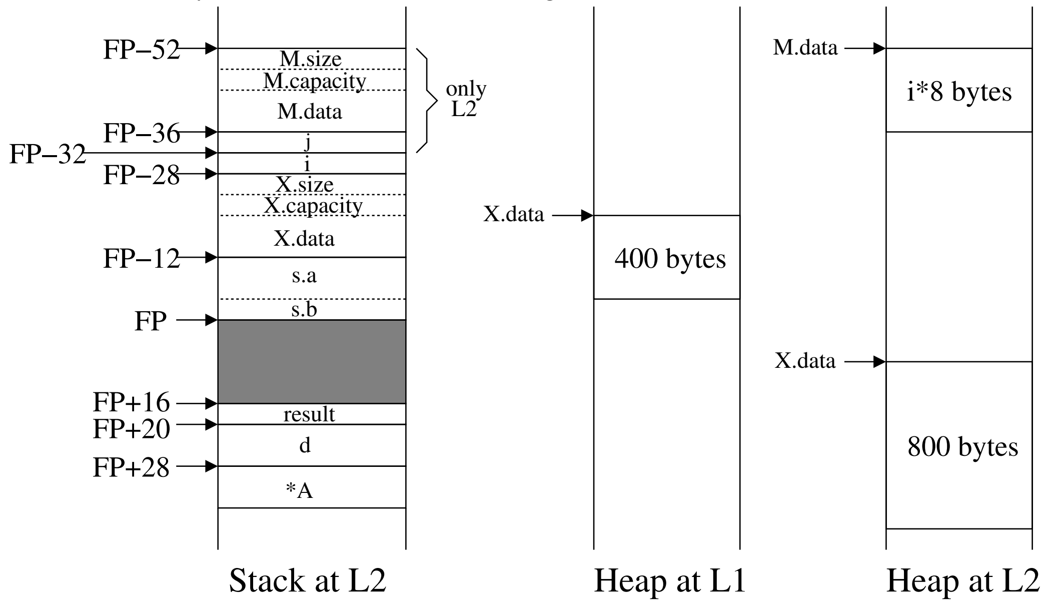
At `L1` and `L3`, the symbol table has the following contents:

| **sym** | **type** | **data type** | **size** | **offset** |
|---|---|---|---|---|
| `A` | param | pointer to struct {int size; int capacity; int* data; } | 8 | 28 |
| `d` | param | double | 8 | 20 |
| `s` | local var | struct { double a; int b;} | 12 | -12 |
| `X` | local var | struct {int size; int capacity; int* data; } | 16 | -28 |
| `i` | local var | int | 4 | -32 |

At L2, a new scope would be added to the previous table with the following contents:

| **sym** | **type** | **data type** | **size** | **offset** |
|---|---|---|---|---|
| `j` | local var | int | 4 | -36 |
| `M` | local var | struct {int size; int capacity; double* data; } | 16 | -52 |

The allocated memory would be as shown in this figure:



At `L1` and `L3` the stack would have the same contents except the top space referred as "only L2". The heap at `L3` would be the same as in `L2` except for the space of `M.data`. This figure does not show the space allocated at the heap for vector `A` before the call to `f`.

**Memory leaks:** A memory leak would be produced when some memory is allocated by means a `new` statement but it is not deallocated with a `delete`. An example would be a function that allocates memory, returns a pointer to that memory regions and the caller never deallocates the memory.

A C++ program that does not use `new`/`delete` statements will never produce memory leaks. This is true as long as the classes used by the program (e.g., vectors, lists, maps, etc,...) are properly designed in such a way that the constructors and destructors allocate and deallocate memory correctly. C++ implicitly calls the constructors and destructors upon the calls from and returns to functions, respectively.

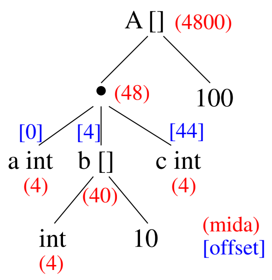
4. Consider the following data structure and assignment:

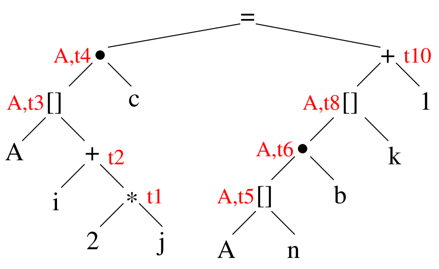
```
struct {
    int a;
    int b[10];
    int c;
} A[100];
```

```
A[i+2*j].c = A[n].b[k] + 1;
```

- Describe the AST of the data structure and assignment.
- Calculate the size of the data structure and the offset associated to each field of the structure. Assume that `int` takes 4 bytes. You must annotate the information next to the nodes in the AST.
- Generate the code of the assignment. Each node of the AST must be annotated with the variable (and offset, when applicable) that contains the value calculated for that node. You must generated unoptimized code, in a similar way as a code generator would do when executing a post-order traversal of the AST.

**Solution:**





```
t1 = 2*j
t2 = i + t1
t3 = t2 * 48
t4 = t3 + 44
t5 = n * 48
t6 = t5 + 4
t7 = k * 4
t8 = t6 + t7
t9 = A[t8]
t10 = t9 + 1
A[t4] = t10
```

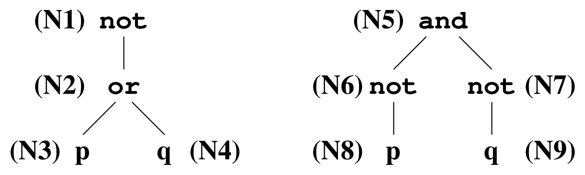
5. De Morgan's law says that $\neg(p \vee q) \equiv \neg p \wedge \neg q$.

Show that the code generated by the following two statements (using backpatching) is identical:

- `if not (p or q) then S1 else S2`
- `if not p and not q then S1 else S2`

**Solution:**

Let us consider the AST of the Boolean expressions where $N_i$ refer to the nodes of the tree:



where `p` and `q` represent any generic expression. For simplicity, we will assume they are Boolean literals and that the function `EvalBoolExpr(p, L1, L2)` generates the code:

```
if p goto L1
goto L2
```

Let us assume that `Ltrue` and `Lfalse` are the labels created by `Execute(if B then S1 else S2)` to invoke `EvalBoolExpr(B, Ltrue, Lfalse)`. We will see that the code to evaluate the Boolean expression `B` is the same for both trees.
For the first tree:

```
EvalBoolExpr(N1, Ltrue, Lfalse) -> EvalBoolExpr(N2, Lfalse, Ltrue)

EvalBoolExpr(N2, Lfalse, Ltrue) -> EvalBoolExpr(p, Lfalse, Lx)
                                -> EvalBoolExpr(q, Lfalse, Ltrue)
```

and the (unoptimized) code for `N2` is:

```
    if p goto Lfalse
    goto Lx:
Lx:
    if q goto Lfalse
    goto Ltrue
```

For the second tree:

```
EvalBoolExpr(N5, Ltrue, Lfalse) -> EvalBoolExpr(N6, Lx, Lfalse)
                                -> EvalBoolExpr(N7, Ltrue, Lfalse)

EvalBoolExpr(N6, Lx, Lfalse)    -> EvalBoolExpr(p, Lfalse, Lx)
EvalBoolExpr(N7, Ltrue, Lfalse) -> EvalBoolExpr(q, Ltrue, Lfalse)
```

thus deriving the same code as above.

6. Consider the following code with nested static scopes:

```
func P (int a) {
    int x;
    func Q (int b) {
        int y;
        x = a + y;       // S1
    }
    func R () {
        int a;
        func S() {
            int c;
            c = a - x;   // S2
            P(a);        // S3
        }
    }
}
```

Generate code for the three statements in the lines with comments `S1`, `S2` and `S3`.

**Solution:**

```
S1: x = a + y              S2: c = a - x              S3: P(a)

t1 = static_link           t1 = static_link           t1 = static_link
t1 = t1 + offset(P::a)     t1 = t1 + offset(R::a)     t1 = *t1
t1 = *t1                   t1 = *t1                   t1 = *t1
t2 = t1 + y                t2 = static_link           param t1
t3 = static_link           t2 = *t2                   t2 = static_link
t3 = t3 + offset(P::x)     t2 = t2 + offset(P::x)     t2 = t2 + offset(R::a)
*t3 = t2                   t2 = *t2                   param t2
                           t3 = t1 - t2               call P,2
                           c = t3
```