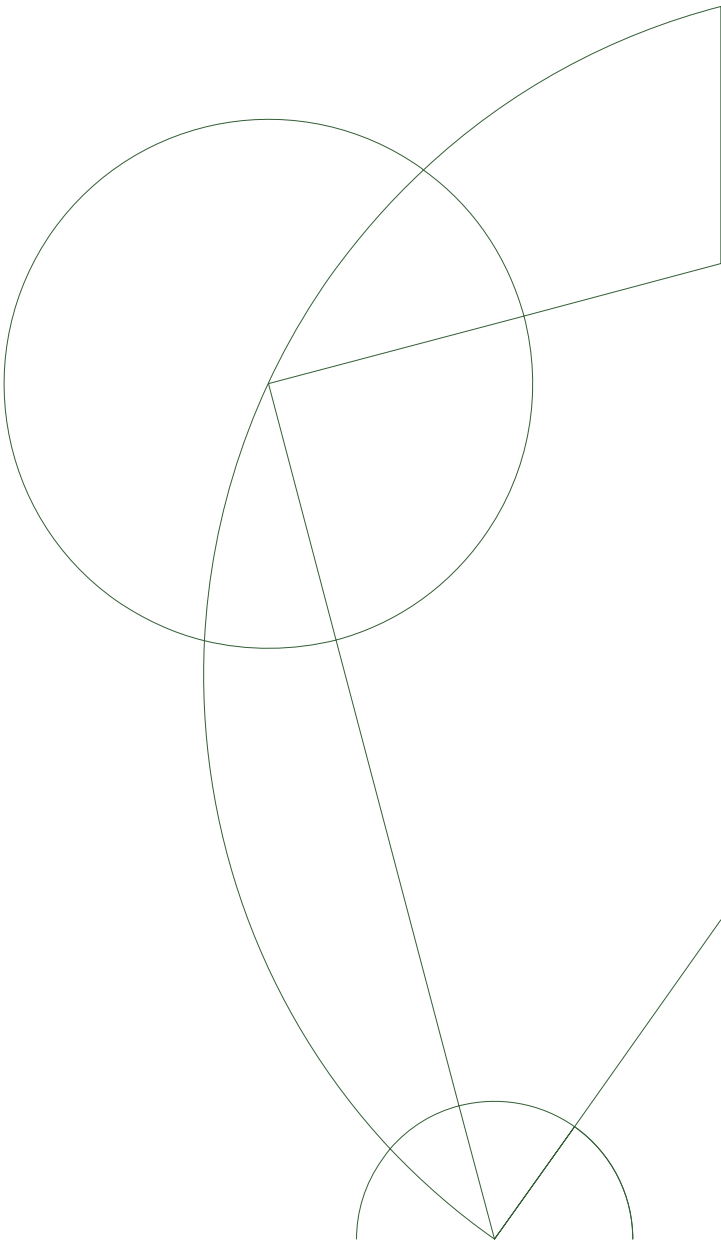
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

CompSys: A1

Unicode

Frederik Ingwersen
Benjamin Amram
Rasmus Winther

September 22, 2018



Program

The program *file.c* has the purpose of checking file types of a given number of files, this is accomplished by checking the encoding. In this iteration of our program, we will be implementing type checks for the file types of *ISO-8859, UTF-8, UTF-16-LE/BE*.

How to compile and run tests: For compiling the code run:

```
$ gcc file.c -o file -std=c11 -Wall -Werror -Wextra -pedantic -g3
```

Then you want to run the code giving it an arbitrary number of files as arguments, like so:

```
$ ./file fileOne ../fileTwo directory/fileThree
```

As for the tests, simply run:

```
$ ./test.sh
```

All of our "is" functions are fairly simple, all following the logic of checking byte by byte and comparing to given hexadecimal values. These all return 0 for success and 1 for failure to determine a file's type.

There is a part starting at line 164, in which we initiate the check for file type *UTF-8*; we want to keep track of how many bytes we should check in sequence, this is done by creating a "numOfBytes" variable which is set by the output of our helper function "followingBytes", which checks for the conditions explained in our assignment text.

We then use the "numOfBytes" variable to create an array which is to store our sequential bytes, feed that to the "isUTF_8" function which then loops over each byte and checks the validity in regards to *UTF-8*.

The "determine_type" function uses all our helper functions to analyse the files byte for byte, which results in our best guess of the file type.

Assembly

Program I

```
int p1(int x) {
    int y = x;
    x = -x;
    if(x<0) {
        return x;
    } else {
        return y;
    }
}
```

The function takes a value x and stores it in another variable y and negates x. If x then is less than zero, then x is returned otherwise y is returned.
This results in a program that returns the negative input value, so if 3 is given p1 will return -3. If -3 is given the function will then return -3.

The assembly code's first line moves an int value to the register %rax which end up being the return value.
testq tests whether the value on register rdi is less than zero, in the C program variable x. Then cmovs moves the value on register rdi to rax if the SF flag was set to one by the testq operation. These two lines results in the "if else" statement where the return statement in the C code is equal to returning the value in the rax register.

Program II

```
P2:
    movl $0, %ecx
    movl $0, %eax
    jmp L
```

The assembly code seems to start by assigning value 0 to two register addresses. It then jumps to L3, skipping L4.

```
L4:
    addq $1, %rax
```

Here we increment $\%rax$ by 1.

```
L3:
    movq (%rsi,%rax,8), %rdx
```

This is the same as taking the value of the address $\%rsi + \%rax * 8$ and assigning it to $\%rdx$. Since $\%rax$ is initially set to 0, the address will just be $\%rsi$.

```
testq %rdx, %rdx
je L5
```

testq uses the "and" operator without actually assigning the resulting value, but this means that the zero flag gets set ONLY IF $\%rdx$ is 0. Since *je* will only jump when the zero flag is set, the jump will only happen when $\%rdx$ is 0.

```
cmpq %rdi, %rdx
je L4
```

cmpq compares two quad words, meaning it subtracts the first argument from the second, and the resulting value will then set the conditional flags. Again we only jump if the zero flag is set, and that only happens when $\%rdi$ and $\%rdx$ are the same.

```
movq %rdx, (%rsi,%rcx,8)
addq $1, %rcx
jmp L4
```

Here we assign value $\%rdx$ to the address $\%rsi+\%rcx*8$. We then increment $\%rcx$ and jump back to L4.

```
L5:
    movq $0, (%rsi,%rcx,8)
    subq %rcx, %rax
    ret
```

Here we assign value 0 to the address $\%rsi + \%rcx * 8$. After that we subtract $\%rax$ by $\%rcx$ and assign it to $\%rax$. Lastly $\%rax$ gets returned.

Here is the decompiled C code:

```
// assembly registers = c variables:
// %rxc = i
// %rax = j
// %rsi = p
// %rdx = a
// %rdi = n

int program2(int* p, int n) {
    int i = 0;
    int j = 0;
    int a = 1;
    while (a != 0) {
        a = *(p + sizeof(int) * j);
        if (n == a) {
```

```
            j++;
        }
        else {
            *(p + sizeof(int) * i) = a;
            i++;
            j++;
        }
    }
    *(p + sizeof(int) * i) = 0;
    return j - i;
}
```

From what we can gather, the program seems to take a pointer to a quad word variable p and a quad word variable n as arguments. The program only returns when $a! = 0$, and j gets incremented every iteration of the while loop. i only gets incremented when the value at address $(p + sizeof(int) * j)$ is equal to n. The while loop therefore continues until it hits an address containing value 0.

The return value $j - i$ will be the amount of values matching n until hitting the 0 value.

One could think of $(p + sizeof(int) * i)$ as the i'th index of an array starting at address p. The problem is that we don't know the size of the array, so we cycle through the metaphorical array using pointer arithmetic.