Technical University of Munich
School of Engineering and Design
Prof. Dr. Martin Werner

Computational Foundations II (Summer Term 2022)
Tutorial 1

Tasks marked with a star like **Optional Task*** are optional. Tasks marked like **Hard Task+** are given, but it is not expected that you solve them now. It is great if you learn to solve them during the lecture. Go back to them after a few weeks and see your own progress.

Learning Outcome: Number Representation, Boolean Algebra, Combinatorial Circuits and LL Parser

Task 1: Number Representation Systems

Number Representation in terms of positional number systems is a key insight enabling the representation of more or less everything in binary representation. From a digital electronics perspective, this is great as common problems of real-world electronic systems, especially noise, can be removed (mostly!) by having large regions of voltages representing the bits 0 and 1. Furthermore, it is interesting to know that modern (future) hardware might not be as binary anymore. Maybe read on Resistive Random Access Memory (RRAM) if interested.

a. **Number Conversions:**Convert decimal numbers 10, 4, 2.4, 1.25, and $1\frac{1}{3}$ into binary, octal, and hexadecimal values.

b. **Base32** Write a C/C++ program that converts a memory area (given by a pointer and a length variable) into a stream of Base32 characters. Choose one of the available Base32 character sets.

c. **Conversion Tool**: Write a C/C++ program to convert integer numbers between decimal and binary system.

d. **FP Conversion***: Write a program that convers a decimal floating point number given as a string into IEEE 754 float representation.

e. **GeoHash:** Base32 has been used to define a representation of spatial locations on Earth surface in short strings of characters. Try to find information on it on the Internet and implement a conversion from WGS84 GPS coordinates into strings.

Task 2: Boolean Expressions

The Boolean Algebra is an interesting algebra as it allows to be implemented quite easily in digital electronic circuits. In this task, we will try to implement and simplify some specific digital circuits.

a. **NAND is a universal gate:** From the lecture, we know that AND and NOT suffice to implement a min-term and that OR suffices to implement every binary function $\mathbb{B}^k \to \mathbb{B}$. Implement these three gates in terms of a NAND gate only.

b. **Truth Table and Expressions:** Build a Truth Table of 3 input variables and assign an output with 5 ones (randomly placed as you like). Derive the SOP representation, simplify it using algebraic identities, and implement the shortest circuit you can find.

c. **Combinatorial Complexity:** How many Boolean functions $\mathbb{B}^k \to \mathbb{B}^l$ do exist? How does this compare to the number of atoms in the universe? This task shows that the combinatorial complexity is so high that verification of circuits is going to be a challenge.

Task 3: Expression Parsing

As a first part of a digital design flow, one might be interested in automating the implementation of truth tables and expressions. Turning a given truth table into an expression is extremely simple due to the SOP representation. However, the step from such an expression to a circuit is a bit more complicated and provides an excellent (despite not very easy) progrmaming task.
We want to parse (and evaluate) an expression using a concept called LL parser (which is sufficient for expressions). In such a parser, we will always look at the next char only (Note that we only have one-digit numbers 0 and 1 and we will only allow character a,b,c,... to appear in an expression). This is called the next token.
The grammar for a simple expression parser (which can even compute the expression) is given, for example, as follows:

```
Expression --> Expression + Term | Term
Term --> Identifier | (Expression)
Identifier --> a|b|c
```

When you try to implement an LL parser, then each such rule is implemented as a function. The return value of this function could be a pointer or value, in our start implementation it is nothing. If you want to implement a calculator, you can try to return a value. If you want to implement some decent algorithm, e.g., for simplifying a circuit, you would maybe return a pointer to an object and would wire everything together into a parsing tree (which is the opposite of the circuit, remember?).
One thing to note is that LL parsers cannot implement left-recursive text replacements (e.g., first rule) as this would loop infinitely: Calling expression (because we expect an expression) would immediately call expression which would call expression and so on. Therefore, we replace this with an iterative version of the rule:

```
Expression --> Term { + Term }*
```

This is, we expect something different (term) which will be terminating and possibly (in a while loop) more terms appended with +.
To get you started into this abstract way of writing programs, here is a first version of such an LL parser implementing only identifier a and b and an operation +.

```
#include<iostream>


using namespace std;

char *ptoken;
int where = 0;
char token()
{
   return *ptoken;
}
```

```
bool nexttoken()
{
    ptoken ++;
    where ++;
    if (*ptoken == '\0')
       return false;
    return true;
}

void expression();

void identifier(){
    cout << "Identifier at " << where << std::endl;

   if (token() == 'a' || token() == 'b')
   {
   nexttoken();
   std::cout << "PARSED IDENTIFIER a or b" << std::endl;
   }else
    throw(std::runtime_error("identifier not a or b"));
}

void term()
{
    cout << "Term at " << where << std::endl;
    if (token() == '(')
    {
        nexttoken();
   expression();
   if (token() != ')') throw std::runtime_error("Expected )");
   nexttoken();
    }else
       identifier();
}

void expression()
{
// this is an iterative version of the left-recursive rule to avoid infinite
    loop
// analyze this change
// the modified grammar rule is expression--> Term { + Term}* where {*}
    denotes that this can happen as often as possible
// more down to Earth, this means we cut off the left-most Term and continue
    as long as there are some terms.
cout << "Expression at " << where << std::endl;
term(); // this is exactly, where we cut off
while (token() == '+')
{
    nexttoken();
    term();
}

}



int main(void){
   string expr;
```

```
    cout << "Expression: ";
    cin >> expr;
    ptoken = &expr[0];
    expression();
    return 0;
}
```

- **Understand:** Run this parser with various expressions in a debugger and see how the parsing actually works
- **Boolean Algebra:** Complete this parser to accept all expressions built from "a,b,c,+,*,(,)".
- **Negation:** Try to implement negation "!a". This is a new grammar rule as it is valid to negate expressions and identifiers, for example "!a" and "!(a+b)" are valid expressions.
- **Truth Table (optional):** Use this parser to compute a Truth table. Therefore, run the parser for each row of the Truth table and replace a,b,c with their real values (return them) and implement the operations.

If you want to continue along these lines, a nice student tool would be one that checks your simplification results. That is, write a program that given two expressions proofs by exhaustion that two expressions are same. If they are not, the program shall return the first or all contradictions to convince you.
Another nice student tool could be one that computes a truth table expression by expression showing intermediate results just in the way we did it in the lecture.
If you want to know how this is done for real, read on YACC (yet another compiler compiler), GNU bison, and flex. And if you want to see it in industrial grade software, have a look at postgresql.