The following summary is based on the latest state of implementation. This means that although some parts changed over time, only the latest state is described. E.g. the attribute for constant folding was first implemented as global variable, later as pointer to integer and at the end it was realized as struct, even its name changed over time. But here it is mentioned only with its final name and as struct (although *struct* was implemented later than the attribute itself). This summary is a rough overview about our changes on selfie but it is not complete as it should not be longer than one page.

For assignment 0 we added code to print out our team name on the console in function *main*.

Then we implemented the four logical bitwise shift instructions of MIPS *sll, srl, sllv* and *srlv*. First we added global constants to decoder, assigned the value as defined for MIPS to it and then modified function *initDecoder*. At the emulator we added new instructions for the four functions. These instructions do the bitwise shift by calling the library functions *leftShift* or *rightShift* and increase the program counter after that. As functions *sll* and *srl* uses a constant for shifting we added the global variable *shamt* to decoder, added the parameter *shamt* to functions *emitRFormat, encodeRFormat, decodeRFormat* and implemented the function *getShamt*. If the global variable *shamt* is greater than zero, we do the shift in function *fct_sll*, in other case we call *fct_nop*. In function *execute* we added code to call one of the four functions according to given MIPS function value.

Next we implemented the scanning, parsing and code generation for the shift operators (<<, >>). First we added global constants to scanner and modified function *initScanner*. Then we changed function *getSymbol* to differentiate shift from comparison symbols. Furthermore we added function *isLogicalShift* to parser, which returns whether a shift symbol is found or not. Between functions *gr_simpleExpression* and *gr_expression* we added the function *gr_logicalShift* and changed the calls in and of these functions accordingly to ensure the satisfaction of precedence as defined for C operators. Function *gr_logicalShift* emits the MIPS instruction depending on which shift symbol is found and whether the shift amount is a constant or an expression. After that we changed functions *leftShift* and *rightShift* so that shift operators are used instead of multiplication, division and function *twoToThePowerOf*.

The next part of our implementation was constant folding in arithmetic expressions. At parser we added a struct *attribute* which we used to store a flag that shows whether a constant was found and a variable that contains the constants value. In function *gr_cstar* we created a pointer to the one and only instance of *attribute* with the name *infos*. This pointer we put as parameter to all functions which needs access to its data. We changed function *gr_factor* to set flag and store value at *infos* if an integer (constant) is found. Then we changed functions *gr_term, gr_simpleExpression* and *gr_logicalShift* to do arithmetic operations at compile time if both operands are constants and store the result at *infos*. In other case we ensured that code is generated to load both operands into registers. Function *gr_expression* generates code for loading the value into register if an operand is a constant before doing comparison. This means that for comparison constant folding does not work. For code generation we used function *load_integer_after_check* which we created to load also negative constants correctly into register.

After that we implemented scanning, parsing and code generation for one and two dimensional arrays. We implemented global and local array declaration, array access and array parameter. We added further data regarding array size, elements type and size of second dimension to symbol table. For scanning we added code to recognize left and right bracket. Then we implemented array access at parser by changing function *gr_factor* and creating functions *gr_array* and *gr_selector*. Function *gr_array* is responsible to generate code for computing the address by adding arrays address and given index and for loading this address into register. With *gr_selector* the index expression is parsed and code is generated. Constant folding is applied to index expressions. If index is constant, it is checked whether it lies within the arrays bounds and then code is generated to load index into register. At function *gr_procedure* we created a new pointer *additionalMemorySpace* which is modified in function *gr_variable* and then used to allocate enough space for array. Also it was necessary to consider array size in function *emitGlobalsStrings*.

Next we implemented scanning, parsing and code generation for struct. We implemented the global struct definition, global and local pointer-to-struct declaration, struct access and pointer-to-struct parameter. Again we expanded the symbol table with further data. We added functions *gr_record, gr_field* and *gr_fieldAccess*. The first two functions are necessary to recognize the struct definition with its fields. If such a struct definition is found only a symbol table entry with a list of all its fields is created but no memory allocation is done. This is done as recently as a struct declaration is found. The third mentioned function is used for struct access.

After that we implemented scanning, parsing and code generation for boolean operators (*!*, &&, ||) with lazy evaluation. For this we implemented a list which we realized as struct and added to *attribute* two lists, one to store the true jumps and the other to store the false jumps. These lists we used as fixup chains. The negation we realized by generating code to flip the result of the relevant boolean expression. For precedence purpose we added the function which recognizes AND expressions after comparison (*gr_expression*) and the function for OR expressions after that.

Finally we implemented the MIPSter syscalls *emitFree* and *implementFree* to free memory. The first one generates code to call the second one which stores a pointer to given memory into free list. On *malloc* this free list is checked whether there is some free memory which can be reused. The reuse only works if given size of *malloc* is equal to the size of a symbol table entry. The function *free* works only without errors on allocated memory which is greater or equal than the size of a symbol table entry.

Thus we implemented all assignments as asked for.