# Comp

CMPE230 Project 1

Alper Çakan
2016400000

# 1. Summary

In this document, we elaborate the design of a compiler which compiles a simple language, named **comp**, into A86 Assembly code.

# 2. Design Principles

Programmatically generating code is a challenging task. The foremost reason is that, the code generated for a single expression, assignment and so on should fit anywhere in the resulting, complete code. Otherwise, we would need to produce different codes for the same expression depending on where it will be inserted; and that is a very complex problem.

So, here is a simple example for this. Consider that we need to generate code for calculating `x + y`. If we can guarantee that we always use a particular register for holding the results of expressions, say AX, then we can simply insert the following Assembly code whenever we see the expression `x + y`:

```
push BX
mov BX, [y]
mov AX, [x]
add AX, BX
pop BX
```

The idea here is that, every programmatically generated code must have a very strict set of assumptions, such as; the registers it modifies, the registers it uses as *input*, the registers it places the result in, will the top of the stack be the same as it was before this code snippet and so on. When we carefully design these assumptions for each programmatically generated code **and** for each helper subroutine that we have written by hand (such as 32-bit number print subroutine etc.), then the code generation will be as easy as looking up code snippets from a dictionary. Hence, we have created such a design. Preconditions, postconditions and assumptions of our subprograms can be found in the file `handcrafted.asm` in the project.

Also, we used the subprogram's name as a prefix for each label which is used internally by that subprogram. This way, we eliminate the possibility of name collision.

Furthermore, we do not use the variable names written by the user. Whenever, a variable name (ID) is encountered while parsing, it searched among the current set of *seen* IDs. If it is new, it is inserted to the set. Then, while generating the code, we create a data segment placeholder (variable) in the A86 Assembly code as using a prefix and the index. For example, if a variable named `thisIsVariable` has index 3 in the variable set generated while parsing the code, then we generate two variables in the Assembly named `compxh3` and `compxl3`, which will respectively be the high word (16 bit) and the low word of the variable. The prefix `compx` is for preventing possible name collisions with any helper code that was handwritten, which deliberately includes nothing starting with the character `compx`. With this method, we do not have to handle many problems about the user-created variable names, such as case sensitivity and so on.

## 3. Addition

We are expected to be able to add 32 bit numbers. However, the problem is that the CPU is 16 bit. But, addition of any two numbers can at most one carry bit, hence we can just low words of inputs to obtain the low word of the result; and add the high words on top of the carry from the low word addition to obtain the high word of the result. Further details can be found in the subprogram `ADD_DWORD` in the file `handcrafted.asm` of the project.

## 4. Multiplication

The most challenging part about the multiplication is that we need to be able to multiply 32 bit integers, whereas A86 supports multiplication of 16 bit integers. However, using the following formula, 32 bit multiplication can be reduced to additions and multiplications of 16 bit integers:

Let `ba` and `dc` be 32 bit integers, where `ba = b*2`$^{16}$ `+ a` and similarly for `dc`. Then, `ba * dc` can be calculated as `b*d*2`$^{32}$ `+ b*c*2`$^{16}$ `+ a*d*2`$^{16}$ `+ a*c`. However, we have another assumption: the result must fit in 32 bits. Then this means `b*d` must be zero, because even it's coefficient is 32 bits. Furthermore, we can assume d = 0, because if not, we can just swap the `ba` and `dc`. Now, our result is `b*c*2`$^{16}$ `+ a*c`. Then, the least significant word of the result will be the least significant word of `a*c`. And, the most significant word of the result will be `b*c` (this is one word because otherwise the coefficient `2`$^{16}$ would make overall result more than 32 bits) plus the most significant word of `a*c`. Using this mathematical concepts, we designed a subprogram named `MUL_DWORD` which calculates `(BX AX)*(DX CX)` and places the result into `(BX AX)`.

## 4. Exponentiation

For exponentiation, the repeated squaring algorithm was employed. However, the significant problem was that we did not have a place for keeping the "running" result of the multiplication, since all four of the working registers was being used for keeping the inputs. However, the solution we came up with was using two words from the data segment to keep it. Further details can be found in the file `handcrafted.asm` of the project and in the homework lecture note in Piazza.

## 5. Parsing

For parsing, we used an optimized version of the popular top-down parsing algorithm, LL(k) (where k ≈ 4 for our case). By "optimized", we mean that there are some significant changes from the original algorithm; however, these changes work only for the syntax of **comp** but they do make parsing easier and faster.

The first change is that we do line by line parsing. Instead of designing a complicated set of production rules to express line by line parsing, we simply ignore formalism about lines and just iterate line by line while parsing.

The second change is that, we use *arbitrary lookahead* while parsing. Instead of a constant lookahead length, we just search the character '=' on the line (which means lookahead length can be as long as the line length) to decide if we should parse this line as an assignment or as a an expression. If there is '=' on the line, we just divide the string from that character, and then parse left substring as

variable name/ID, parse right substring as expression. This method saves a lot of time while parsing because we just eliminate a long set of rules just to formalize assignments.

The last difference is that, instead of using many rules for *skipping* the whitespaces (skipping as in C-like syntaxes), we handle the skipping directly in the parsing code.

Other than these, we just use the classic algorithm. However, it is important to note that *looking-ahead* to decide which production rule use is just used at one place at our parsing algorithm, because only one nonterminal has more than one production rule. For nonterminals with only one production rule, we just call the methods of the left hand side of the rule consecutively to consume the input; and if after all, the line is not wholly consumed, this means this line does not conform with the production rule, and hence is syntactically wrong.

Below is the grammar we used for expression parsing. Note that we do not have a formal grammar for assignments because we just parse them as `<id> = <expr>`

```
<expr> → <term> <more-terms>
<more-terms> → + <term> { ‘+’ } <more-terms> |
               ε

<term> → <factor> <more-factors> |
<more-factors> → * <factor> { ‘*’ } <more-factors> |
                 ε

<factor> → (<expr>) |
           <id> { id } |
           <num> { num } |
           <pow>

<pow> → pow(<expr> { expr }, <expr> { expr }) { ‘^’ }
```

Note that whatever written inside { } in the above grammar are not part of the syntax, they just denote what to put on the top of the postfix stack.

## 6. Code Generation

As mentioned in the design principles section, we have written a set of subprograms in Assembly that have very strict pre- and postconditions. Thanks to that, code generation is just a simple dictionary lookup algorithm. Further details can be found in the source code. However, one important part to note is that, for generating the code for expression calculation, we first generate a postfix stack because evaluation of postfix expressions is easier than that of infix expressions. Another important note is about our code generation code. Although the project contains code generation only for A86 Assembly, we designed the code structure so that parsing of the language **comp** and the Assembly code generation is well separated. As a result, it is very easy to implement a feature so that **comp** is compiled to another language, say x86 Assembly, or implementing a feature so that another language is compiled to A86 assembly.

## 7. Documentation

For documentation, we used Doxygen's Javadoc mode, which allows us to write Javadoc like comments in C++ code.