

# Practical Work: Compilation

Master CSMI  
Compilation & Performance  
Bérenger Bramas

October 15, 2024

## 1 Summary

In this practical work, you implement some parts of a compiler.

## 2 Ressources

- Emacs regexp: <https://www.emacswiki.org/emacs/RegularExpression>
- C++ regexp: [https://fr.cppreference.com/w/cpp/regex/basic\\_regex](https://fr.cppreference.com/w/cpp/regex/basic_regex)
- C++ regexp help: <https://www.regular-expressions.info/stdregex.html>
- Register allocation: [https://en.wikipedia.org/wiki/Register\\_allocation](https://en.wikipedia.org/wiki/Register_allocation)
- Online regexp tester: <https://regex101.com/>

### 2.1 Get the practical work

Consider you are in your project directory do the following:

```
# Clone my repo
# If you use SSH, use:
# git clone git@git.unistra.fr:bbramas/csmi-tp-2024.git --branch=TP3 csmi-tp3
# With https
git clone https://git.unistra.fr/bbramas/csmi-tp-2024.git --branch=TP3 csmi-tp3
# Go in the newly created directory
cd csmi-tp3
```

### 2.2 Add your repository as remote

You will push on your own repository:

```
# Rename my remote
git remote rename origin old-origin
# Add your own remote
# If you use an SSH key:
# git remote add origin git@git.unistra.fr:[YOU LOGIN HERE]/cnp-tp-2024.git
# If you use https:
git remote add origin https://git.unistra.fr/[YOU LOGIN HERE]/cnp-tp-2024.git
# Push the current branch and active the tracking
git push -u origin TP3
```

## 2.3 Compilation

To compile, we use CMake:

```
cd TP3
mkdir build
cd build
cmake ..
make # Will make all
make something # Will build only something
VERBOSE=1 make # Will show the commands used to compile (including the flags)
```

By default CMake will not use any optimization flag, and looking at the output of *VERBOSE=1 make* will show you that there is *-Ox*. To enable optimization you have to specify to cmake to generate a make file with the correct flags. To so, you can use *ccmake*. (in the build directory, note that *ccmake* should be installed) and edit *DCMAKE\_BUILD\_TYPE*, or directly set the variable with *cmake.. -DCMAKE\_BUILD\_TYPE = Release* in the build directory.

## 3 Reminder

A compiler is usually subdivide into three parts: the front end, the middle optimization layer, and the back end. In the front end, the lexical analysis step uses regular expressions to find token in the text. These tokens are later uses to analyzes the syntax and validate the grammar, before generating IR. The optimization layers work on the IR to perform optimization in multi-passes. Finally, in the back end, the compiler has to decide where the variable should be stored (in memory, in register) and how data should be moved between both.

## 4 Regex

Update the *regexTest.cpp* file to add three regular expressions to catch: C++ multilines comments, decimal numbers and variable declarations (considering only int, float, double and string types).

Example of output for the given test case:

```
$ ./regexTest ../test-input.cpp
Will read ../registerAllocations.cpp
Match var_decl: string line;
Match var_decl: double num = 1.2;
Match var_decl: float anotherNum = 1123123.3123123E-1;
Match decimal_number: 1.2
Match decimal_number: 1123123.3123123E-1
Match decimal_number: 0
Match comments: /* I am a comment */
Match comments: /* I am a multiline
comment */
Match comments: /** I am a difficult
multiline
comment *
with star
*/
```

You can extend the decimal numbers with scientific notations.

Note that, ideally, a regex should be defined such that it does not catch more strings than expected.

Here is a formal definition of the elements:

Specification: an identifier is an arbitrary long sequence of digits, underscores, lowercase and uppercase Latin letters. A valid identifier must begin with a non-digit character (Latin letter or underscore). Identifiers are case-sensitive (lowercase and uppercase letters are distinct), and every character is significant.

Integer in *c* (but we use only decimal in our case, and without suffix):

Specification: an integer literal is a primary expression of the form:

1. *decimal-literal integer-suffix*(optional)

where

- *decimal-literal* is a non-zero decimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9), followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Floating point numbers in C (we do not use suffix in our case):

Specification: floating-point literals have two syntaxes. The first one consists of the following parts:

- nonempty sequence of decimal digits containing a decimal point character (defines significand)
- (optional) e or E followed with optional minus or plus sign and nonempty sequence of decimal digits (defines exponent)

The second one consists of the following parts:

- nonempty sequence of decimal digits (defines significant)
- e or E followed with optional minus or plus sign and nonempty sequence of decimal digits (defines exponent)

## 5 Optimization

In the file `registerAllocation.cpp`, a list of instructions is stored in a vector.

Original instructions:

```
>>t0 = $0
>>t1 = $10
>>t3 = t0+x
>>t4 = t0*t1
>>t5 = t4+t0
>>t6 = y*t1
>>t7 = t6*t1
>>t8 = t6*t1
>>t9 = t0*t8
>>t10 = t4*t8
>>t11 = t4*t8
>>t12 = t3*t7
>>t13 = t12*t11
>>t14 = t12*t11
>>return t13
```

You have to optimize those instructions.

**Constant propagation** In the instructions, some values are already known and some operations could be evaluated at compile time. You have to implement a constant propagation mechanism where each known variable is replaced by the formula to evaluate.

For example:

```
t0=$10 // t0 is actually equivalent to 10
t1=t0*t0 // we know the value of t0, so it should be t1=($10*$10)
// and if t1 is later used, we also know its value
```

A possible implementation of this mechanism is to keep in a map all the constant variables (*map[name] = formula*) and to propagate them.

In the test case this should give:

After constant propagation:

```
>>t0 = $0
>>t1 = $10
>>t3 = $0+x
>>t4 = $($0*$10)
>>t5 = $($($0*$10)+$0)
>>t6 = y*$10
```

```

>>t7 = t6*$10
>>t8 = t6*$10
>>t9 = $0*t8
>>t10 = $($0*$10)*t8
>>t11 = $($0*$10)*t8
>>t12 = t3*t7
>>t13 = t12*t11
>>t14 = t12*t11
>>return t13

```

**Remove unused variables** A single value is returned at the end of the function (instruction list). Therefore, several variables are actually unused: Create the code to remove them. One possible approach, is to start from the returned value and to iterate on the instructions backward to see the dependencies, and to store in a set all the used variables. Then, a second iteration can simply remove all the variables that are not in the set.

Proceed to remove unused variables:

```

Erase : t0
Erase : t1
Erase : t4
Erase : t5
Erase : t9
Erase : t10
Erase : t14

```

After removing unused variables:

```

>>t3 = $0+x
>>t6 = y*$10
>>t7 = t6*$10
>>t8 = t6*$10
>>t11 = $($0*$10)*t8
>>t12 = t3*t7
>>t13 = t12*t11
>>return t13

```

**Find duplicate** We leave it aside, but you will notice that some variables are equivalent.

## 6 Back end

Now that our list of instructions is optimized (at a high-level at least), we will focus on the allocation of the registers. In fact, the instruction  $x = yOPz$  cannot be converted into a single instruction if  $x/y/z$  are not in registers. More precisely, we consider in our model that each instruction should put its result in register and at least one operand of an instruction should be in register (but a constant would be considered as a register).

So if the user write the code  $x = yOPz$ , we should have one instruction to move  $y$  in a register, call  $OP$ , and move the result at the address of  $x$ .

In this context there is the following terminology:

- Allocation: deciding which values to keep in registers
- Assignment: choosing specific registers for values
- Spilling: storing a variable into memory instead of registers
- Move: moving a variable from memory/register to register/memory

Also, there are important issues when placing a value in a register:

- Aliasing: several addresses may point to the same memory, and thus coherency has to be maintained
- Dirty: if a variable is moved to register and updated, the original memory should be considered a dirty and updated before the end.

In the code, a naive register allocation mechanism is provided. It simply move variables before each operation in the registers (always the first variable). This is far from optimal, and your objective is to do better.