Practical Work: Compilation

Master CSMI
Compilation & Performance
Bérenger Bramas

October 19, 2020

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`
- C++ regexp help: `https://www.regular-expressions.info/stdregex.html`
- Register allocation: `https://en.wikipedia.org/wiki/Register_allocation`
- Online regexp tester: `https://regex101.com/`

3 Practical work organization (always the same)

In the practical work, you will obtain the code from my repository and push it to your repository. Therefore, you will have to clone one branch per session and push it to your own repository. It is mandatory that you **commit and push** frequently (after each question and at the end of the session, at least) such that I can easily look at what you have coded at the end of the session, how you did progress (and potentially compare it with the latest version you will have).

It is required that you filled the report.md file to let me know what you did.

In the rest of the document, we consider you have a repository named *cnp-tp-2020* on *git.unistra.fr* that is private but that I can access in read.

3.1 Get the practical work

Consider you are in your project directory do the following:

```
# Clone my repo
git clone https://git.unistra.fr/bbramas/csmi-tp-2020.git --branch=TP3 csmi-tp3
# If you use SSH, use:
# git clone git@git.unistra.fr:bbramas/csmi-tp-2020.git --branch=TP3 csmi-tp3
# Go in the newly created directory
cd csmi-tp3
```

3.2 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
git remote add origin https://git.unistra.fr/[YOU LOGIN HERE]/cnp-tp-2020.git
```

```
# If you use an SSH key:
# git remote add origin git@git.unistra.fr:[YOU LOGIN HERE]/cnp-tp-2020.git
# Push the current branch and active the tracking
git push -u origin TP3
```

3.3 During the session and while you work on the project

After each question or important modification push the current changes:

```
# No matter where you are in the project directory
git commit -a -m "I did something"
git push
```

3.4 When you are done

You have fully finished your work (at most D+14 H-2):

```
git commit -a -m "I did something"
git push
```

3.5 Important!

Do not forget that you have to fill the report.md file and commit it. Remember to commit regularly to keep track of your work and let me see a history of it if I need it. Do not share any code with someone else, as I am here to answer all questions and support all of you. Remember that you have questions to answer in the moodle before the end of the session and that you must push your branch at the end of the session too. Additional credits can be obtained if you make some modifications after the session to improve your solution. Changes can be made until two weeks after the session minus two hours, ie you must push before the beginning of the n+2 practical work. Do not remove code from the test functions as I use them to evaluate your code. Therefore, if you need you can add extra functions for your own testing/debugging. If some of them are showing interesting things about your code, simply leave a comment in the code and the report.md.

3.6 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)
```

4 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.

5 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 std::string types).

Example of output for the given test case:

```
$ ./regexTest ../registerAllocations.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 multine
comment */
Match comments: /** I am a difficult
multiline
comment *
with star
*/
```

You can extend the decimal numbers with scientific notations.

6 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.

7 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.