Compilation

TP 5: A syntax-driven compiler
A. Isoard & C. Alias

It is now time to translate programs to the assembly language we saw in TP0. More specifically, we will try to compile in one pass: the assembly code will be produced by the parser (without going through an intermediate representation).

Reminder of the targeted assembly language

- **add** $r_{\text{dest}}, r_1, r_2$: adds the content of registers $r_1$ and $r_2$, and puts the result inside register $r_{\text{dest}}$.
- **sub** $r_{\text{dest}}, r_1, r_2$: computes $r_1 - r_2$ and puts the result inside register $r_{\text{dest}}$.
- **ld** $r_{\text{dest}}, [r_{\text{base}} + \text{imm7}]$: loads inside register $r_{\text{dest}}$ the data at the memory address $r_{\text{base}} + \text{imm7}$, where \text{imm7} is a 7 bit integer. Also referred to as an immediate value, since the actual value is immediately in the statement.
- **st** $r_1, [r_{\text{base}} + \text{imm7}]$: stores the value of register $r_1$ into memory at address $r_{\text{base}} + \text{imm7}$.
- **ble** $r_1, r_2, \text{imm7}$: if $r_1 \leq r_2$, jumps to the instruction located at address pc + \text{imm7} + 1. (else, continue to the next instruction). \text{imm7} can be negative (using 2-complement), which allows to jump backward. This instruction allow to implement for loops, while loops and if.
- **ldi** $r_{\text{dest}}, \text{imm8}$: writes the immediate 8 bit integer value \text{imm8} in register $r_{\text{dest}}$.
- **ja** $r_1, r_2$: jumps to the memory address (13 bits) defined by $r_2$ (for the first less significants 8 bits) and $r_1$ (for the most significants 5 bits).
- **j** \text{imm13}: jumps to the memory address (13 bits) \text{imm13}, where \text{imm13} is an immediate 13 bits integer value. This instruction, with ja, allows to implement function calls.

There are only 8 registers, each one having a width of 8 bits. Data memory addresses are also on 8 bits (total maximum of 256 bytes!) while instruction memory addresses are on 13 bits (there can be up to 8192 instruction in the program).

Description of compiler classes

As usual, we reuse the compiler of the previous TP. You should know most of those files now. Here is the list:

- **lexer.l**: Syntactic Analyzer (not changed)
- **parser.ypp**: Grammar of our C like language.

- **(New) Attributes.h/.cc**: Data structures to store information related to left hand side (lhs) and right hand side (rha) of expressions. Try to guess the meaning of those class attributes.
- **(TP3) Type.h/.cc** and **SymbolTable.h/.cc**: Classes used for type-checking. The symbol table implements the context $\rho$, that matches each variable (argument or local variable) to the temporary in which it is stored.

- **(New) Label.h/.cc**: Manages a lot of counters in order to generate new labels with unique names in the assembly program.
- **(New) Register.h/.cc**: Produce temporary “fresh” variables (improperly called registers).
- **(New) CodeDigmips.h/.cc**: Contains functions that produce the assembly code on the standard output.
Exercise 1. Ready... Steady... Generate!

Manip.

- **Look at** `Label.h/.cc` and `Temporary.h/.cc`. Try to create new labels and fresh temporary variables.

- **(Idioms)** Open `CodeDigmips.h/.cc` and complete the holes in the `cjump` macro.

- **(Expressions/Conditions)** Open `parser.ypp`. Complete the holes inside parts 1/ Expressions 2/ Conditions, by drawing your inspiration from the other rules already completed.

- **(Control)** Go to part 3/ Statements and implement the missing parts for translating `while` and `for` control structures. In case you have difficulties, look at the `if/then/else` one.

- **(Memory Allocation)** Open `Type.cc` and study the `allocate()` function.

- **(Functions)** Find where are functions translated. How $\rho$ is build? How is it used inside the expressions? How is the result returned?

Exercise 2. The use of the compiled code

Manip.

- Run your compiler on the provided examples from the `test` directory.

- Can the produced code be directly sent to the simulator under `diglog`? What is missing?

Exercise 3. Translation by hand

Apply, by hand, the translation rules onto the recursive form of the factorial function.