Lab 5
Syntax-based translation

In this lab, we will translate programs to the DIGMIPS assembly language. 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*).

Objective

- Write translation rules.
- Produce code from the grammar itself ("syntax-based translation").

First, get the archive.

5.1 Description of the given code

The archive contains:

- **digcc.lex**. Syntactic Analyzer (not changed)
- **digcc.ypp**: Grammar of our C-like language.
- **(New) Attributes.h/.cpp**: Data structures to store information related to left hand side (`lhs`) and right hand side (`rhs`) of expressions. Try to guess the meaning of those class attributes.
- **(Lab3) Type.h/.cpp** and **SymbolTable.h/.cpp**: Classes used for type-checking. The symbol table implements the context ρ, that matches each variable (argument or local variable) to the temporary in which it is stored.
- **(New) Label.h/.cpp**: Manages a lot of counters in order to generate new labels with unique names in the assembly program.
- **(New) Register.h/.cpp**: Produces temporary "fresh" variables (improperly called registers).
- **(New) CodeDigmips.h/.cpp**: Contains functions that produce the assembly code on the standard output.

5.2 Code generation by direct translation

Note that the code we provide compiles but segfaults even for little examples such that `tests/exassign.c`.

For the first exercice, only consider that the fonction `new_register()` will give you a new fresh temporary (as an int).

EXERCISE #1 ▶ **Code generation for assignments**

Complete the parser so that to generate code for expressions and assignments, with the help of what is already done for "plus expressions":

- What is the generated code for an int constant? What is the generated code for a more complex numerical expression ?
- Look at **CodeDigmips.h/.cpp** to see how the code for add/ldi is generated (printed on the standard output).
- How is the multiplication handled ?
- What does the function `rhs` does ?
- Complete the code generation for all kinds of expressions (excluding records for the moment).

Carefully test with the help of `tests/exassign.c`. Test all kinds of numerical expressions. Test the generated code with DIGEVAL (we modified the DIGEVAL evaluator to handle more registers).

EXERCISE #2 ▶ **Code generation for tests statements (IF)**

For tests, we will have to use labels, temporaries, and evaluate conditions, then jump at new labels.

- Have a look at `Label.h/.cpp` and `Temporary.h/.cpp` to see how labels and fresh variables are generated.
- Redo by hand the code generation for the following code:
  `if (x<=4) then y=2; else y=3;`
- Verify with what is already implemented in `digcc.ypp` for the IF rule (`TK_IF test stmt_block`).
- (Info) The `cjump` function (in `CodeDigmips.h/.cpp`) is used to generate adequate DIGMIPS assembler code for all kinds of tests :
  `cjump(int r1, test_t test, int r2, string target_label)`
  generates the code for the test (`exp1 LE exp2` for $\leq$, for instance), provided `r_1` contains the actual value of `exp1` and `r_2` the actual value for `exp2`. The target value is the label where to jump if the expression is evaluated to `true`. There remains some cases to implement but skip it for the moment.
- Complete the parser so that to generate code for boolean expressions $expr1 \leq exp2$. At this step, you should be able to test your code generation for this kind of tests. Carefully test with the help of the given file `tests/extests.c`.
- For the other kinds of tests, complete the parser in the same way. You also have to complete the remaining cases in the `cjump` implementation to generate code for all kinds of tests (some tests are not natively handled by the DIGMIPS assembler, you have to simulate them "by hand").

EXERCISE #3 ► **While loops**
First, recall the translation rules for `while` loops. Implement them inside `digcc.ypp`. Write your own test files.

5.3 A bit more

EXERCISE #4 ► **Remaining work**
There remains to do:
- Code generation for records (structs) inside expressions.
- Code generation for `for` loops.

EXERCISE #5 ► **Functions**
First, comment out the lines inside the parser (rules `prog` and `function`). Find where the body of functions are translated. Make the link with Lab#4 (allocation of temporaries).
- How is the context ρ build? How is it used inside the expressions?
- How is the result returned?
- Have a look at the generated code for the prelude. What does `push_all` means? Propose a way to implement it.

5.4 Standard Library and runtime

The generated code cannot be executed at this on the DIGMIPS architecture. Read the last section of Lab#4 and have a look at `runtime.c`. The `asm` version of this library should thus be appended to the generated code to be executed on the "real" machine.