Compilation (CS444)

Cahier de TD/TP, 2022-23

Contents

Credits

This sequence of compilation labs has been inspired by those designed by C. Alias and G. Iooss for ENSL in 2013/2014 and for the analysis part, by S. Castellan and L. Gonnord in 2015/2016.

In 2016/17 we changed the support language for Python, and the target machine was the LC3, in 2017/18 LEIA and in 2018/19 SARUMAN. In 2018/19 we added a nice test infrastructure, thanks to Matthieu Moy. This year we change the target machine again (RISCV). All the material will be on the course webpages (bookmark now), for CAP :

`https://etudes.ens-lyon.fr/course/view.php?id=4814`

and for MIF08 :

`https://compil-lyon.gitlabpages.inria.fr/compil-lyon/index-MIF08.html`

Teaching staff Here is the list of all people that were involved in the two courses “CAP” and “MIF08”:

- 2016/2017: Guillaume Bouchard, Sylvain Brandel, Aurélien Cavelan, Thierry Excoffier, Serge Guelton, Laure Gonnord, Erwan Guillou, Nicolas Louvet, Lionel Morel, Xavier Urbain.
- 2017/2018: Aurore Alcolei, Guillaume Bouchard, Sylvain Brandel, Thierry Excoffier, Serge Guelton, Laure Gonnord, Valentin Lorentz, Juan Martinez, Matthieu Moy, Guillaume Salagnac, Xavier Urbain.
- 2018/2019: Laure Gonnord, Rémy Grünblatt, Matthieu Moy and Christan Docskal ; Guillaume Bouchard, Sylvain Brandel, Thierry Excoffier, Nicolas Louvet, Loris Marchal, Juan Martinez, Guillaume Salagnac.
- 2019/2020: Laure Gonnord, Matthieu Moy, Ludovic Henrio and Marc de Vismes ; Thierry Excoffier, Nicolas Louvet, Guillaume Bouchard, Laureline Pinault.
- 2020/21: Laure Gonnord, Matthieu Moy, Ludovic Henrio, Gabriel Radanne and Paul Ianneta.
- 2021/22: Laure Gonnord, Matthieu Moy, Ludovic Henrio, Gabriel Radanne, Nicolas Chappe and Rémi Di Guardia.

Lab 1
Lexing and Parsing with ANTLR4

Objective

- Understand the software architecture of ANTLR4.
- Be able to write simple grammars and correct grammar issues in ANTLR4.

Todo in this lab:
- Install and play with ANTLR.
- Implement your own grammars.
- Understand and extend an arithmetic evaluator (with semantic actions).
- Understand our future test infrastructure for our compiler.
- Deliver a code for two of these exercises **according to strict specifications.**

1.1 Getting started!

EXERCISE #1 ► **Git clone**
On your personal machines or on the school machines, you have to clone the git repository where all codes will be given:

```
git clone https://gricad-gitlab.univ-grenoble-alpes.fr/gonnord/cs444-labs22
```

EXERCISE #2 ► **Tool requirements (personal machines)**
Skip if you use the school machines, everything should be already installed! On your machines, you need to install **in this order**

- Python3, its package manager pip and pytest and other stuff

  ```
  sudo apt install python3 python3-pip
  python3 -m pip install pytest pytest-cov pytest-xdist pyright --user
  ```

  (pyright is not mandatory)

- An ANTLR4 python runtime ($\geq$ 4.8), let's use 4.9.2:

  ```
  python3 -m pip install antlr4-python3-runtime==4.9.2 --user
  ```

- The adequate ANTLR4 jar file:

  ```
  mkdir ~/lib     # or anywhere else.
  cd ~/lib
  wget https://www.antlr.org/download/antlr-4.9.2-complete.jar
  ```

- You may need to install java on your machine:

  ```
  sudo apt install default-jre
  sudo apt install default-jdk
  ```

EXERCISE #3 ► **System paths (all machines), only todo ONCE**

Adapt your `~/.bashrc` (or `Documents_distants/.bashrc`) to your configuration [1], for instance on my machine I have:

[1] on school machines use `/opt/antlr-4.9.2-complete.jar`

```
export CLASSPATH=".:$HOME/lib/antlr-4.9.2-complete.jar:$CLASSPATH"
export ANTLR4="java -jar $HOME/lib/antlr-4.9.2-complete.jar"
alias antlr4="java -jar $HOME/lib/antlr-4.9.2-complete.jar"
```

Then source your `.bashrc`:

```
source ~/.bashrc
```

If ANTLR4 is set correctly, then typing:

```
$ antlr4
```

in your terminal should output a documentation.

1.2 Simple examples with ANTLR4

1.2.1 Structure of a `.g4` file and compilation

Links to a bit of ANTLR4 syntax:

- Lexical rules (extended regular expressions): `https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md`
- Parser rules (grammars) `https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md`

The compilation of a given `.g4` (for the PYTHON back-end) is done by the following command line if you modified your `.bashrc` properly (note: `antlr4`, not `antlr` which may also exist but is not the one we want):

```
antlr4 -Dlanguage=Python3 filename.g4
```

1.2.2 Up to you!

EXERCISE #4 ► **Demo files**
Work your way through the two examples (open them in your favorite editor!) in the directory `demo_files`:

ex1: lexer grammar and PYTHON **driver** A very simple lexical analysis[2] for simple arithmetic expressions of the form `x+3`. To compile, run:

```
antlr4 -Dlanguage=Python3 Example1.g4
```

This generates a lexer in `Example1.py` (you may look at its content, and be happy you didn't have to write it yourself) plus some auxiliary files. We provide you a simple `main.py` file that calls this lexer (this one is hand-written and readable):

```
python3 main.py
```

(or type `make run`, which re-generates the lexer as needed and runs `main.py`).

To signal the program you have finished entering the input, use **Control-D** (you may need to press it twice).

Examples of runs: [`^D` means that I pressed Control-D]. What I typed is in boldface.

```
1+1
^D^D
[@0,0:0='1',<2>,1:0]
[@1,1:1='+',<1>,1:1]
[@2,2:2='1',<2>,1:2]
[@3,4:3='<EOF>',<EOF>,2:0]
)+
^D^D
```

[2]Lexer Grammar in ANTLR4 jargon

```
line 1:0 token recognition error at: ')'
[@0,1:1='+',<1>,1:1]
[@1,3:2='<EOF>',<-1>,2:0]
%
```

Questions:

- Reproduce the above behavior.
- Read and understand the code.
- Allow for parentheses to appear in the expressions.
- What is an example of a recognized expression that looks odd (i.e. that is not a real arithmetic expression)? To fix this problem we need a syntactic analyzer (see later).
- Observe the correspondance between token names and token numbers in `<..>` (see the generated `Example1.token` file)
- Observe the PYTHON `main.py` file.

From now on you can alternatively use the commands `make` and `make run` instead of calling `antlr4` and `python3`.

ex2: full grammar (lexer + parser) and PYTHON driver Now we write a grammar for valid expressions. Observe how we recover information from the lexing phase (for ID, the associated text is `$ID.text`). The grammar includes Python code and therefore works only with the PYTHON driver.

If these files read like a novel, go on with the other exercises. Otherwise, make sure that you understand what is going on. You can ask the Teaching Assistant, or another student, for advice.

From now you will write your own grammars. Be careful the ANTLR4 syntax use unusual conventions: *"Parser rules start with a lowercase letter and lexer rules with an upper case."*[a]

[a]`https://stackoverflow.com/questions/11118539/antlr-combination-of-tokens`

EXERCISE #5 ▸ **Well-founded parenthesis**
Write a grammar and files to make an analyser that:

- skips all characters but '(', ')', '[', ']' (use the lexer rule `CHARS: ~[()[\]] -> skip ;` for it)
- accepts well-formed parenthesis.

Thus your analyser will accept "(hop)" or "[()](tagada)" but rejects "plop]" or "[)". Test it on well-chosen examples. *Begin with a proper copy of ex2, change the name of the files, name of the grammar, do not forget the main and the Makefile, and THEN, change the grammar to answer the exercise.*

EXERCISE #6 ▸ **Another grammar**
Write a grammar that accepts the language $\{a^n b^{2n}\}$. Letters other than a and b, and spaces are ignored, other symbols are rejected by the lexer.

Important remark From now on, we will use Python at the right-hand side of the rules. As Python is sensitive to indentation, there might be some issues when writing on several lines. You can often avoid the problem by defining a function in the Python header and then call it in the right-hand side of the rules.

1.3 Grammar Attributes (actions), `ariteval/` directory

Until now, our analyzers are passive oracles, i.e. language recognizers. Moving towards a "real compiler", a next step is to execute code during the analysis, using this code to produce an intermediate representation of the recognized program, typically ASTs. This representation can then be used to generate code or perform program analysis (see next labs). This is what *attribute grammars* are made for. We associate to each production a piece of code that will be executed each time the production is processed/recognized. This piece of code is called *semantic action* and computes attributes of non-terminals.

We consider a simple (commented) grammar of non empty lists of arithmetic expressions:

$P \to S+$ // + denotes a list and EOF is implicit
$S \to E;$ // expressions are followed by a semicolon
$E \to E + E$
$E \to E * E$
$E \to F$
$F \to int$ // int denotes any constant int value provided by the lexer
$F \to (E)$

The object of the demo is to understand how semantic actions work, and also to play with the test infrastructure we will use in the next labs.

EXERCISE #7 ▸ **Test the provided code (`ariteval/` directory)**

First, have a look on the `README.md` file which gives you some information. You will have to edit it later. Now, have a look onto the grammar in file `Arit.g4`. Remark how each grammar rule has been implemented. Each grammar rule is followed by a **semantic action** at its right-hand side.

To test the provided code, just type:

1. Type

   ```
   make && python3 arit.py tests/test-plus.txt
   ```

   This should print:

   ```
   1+2 = 3
   ```

   on the standard output. **It seems that you have been given a calculator code !**

2. Type:

   ```
   make test
   ```

   This should print:

   ```
   test_ariteval.py::TestEVAL::test_expect[./tests/test-mult.txt] FAILED                                [ 50%]
   test_ariteval.py::TestEVAL::test_expect[./tests/test-plus.txt] PASSED                                [100%]

   ================================================= FAILURES =================================================
   _______________________________ TestEVAL.test_expect[./tests/test-mult.txt] ________________________________

   self = <test_ariteval.TestEVAL object at 0x7f5be113a100>, filename = './tests/test-mult.txt'

       @pytest.mark.parametrize('filename', ALL_FILES)
       def test_expect(self, filename):
           expect = self.get_expect(filename)
           eval = self.evaluate(filename)
   >       assert expect == eval
   E       AssertionError: assert testresult(ex...'1+3*2 = 7\n') == testresult(exi...1+3*2 = 43\n')
   E         At index 1 diff: '1+3*2 = 7\n' != '1+3*2 = 43\n'
   E         Full diff:
   E         - testresult(exitcode=0, output='1+3*2 = 7\n')
   E         ?                                        ^
   E         + testresult(exitcode=0, output='1+3*2 = 43\n')
   E         ?                                        ^^

   test_ariteval.py:24: AssertionError
   ==================================== 1 failed, 1 passed in 0.18 seconds ====================================
   ```

   This test means that you have been given only a partial solution ! Indeed, the semantic action for * given in the skeleton (Arit.g4) is obviously buggy. As the diagnosis suggests, the skeleton returns 43 when it should return 7.

EXERCISE #8 ► **Understand the test infrastructure**
We saw in the previous exercice an example for test run. In the repository, we provide you a script that enables you to test your code [3]. It tests files of the form `tests/test*.txt`. Just type:

```
make test
```

and your code will be tested on these files.

We will use the same exact script to test your code in the next labs (but with our own test cases!).

A given test has the following behavior: if the pragma `// EXPECTED` is present in the file, it compares the actual output with the list of expected values (see `tests/test01.txt` for instance). There is also a special case for errors, with the pragma `// EXITCODE n`, that also checks the (non zero) return code *n* if there has been an error followed by an `exit`.

The following (two) exercises are due on Chamilo. Instructions and deadline on Chamilo too.

EXERCISE #9 ► **Play with the testsuite**
As we saw above, the skeleton is buggy, and we only have two test cases.

1. Fix the code of the semantic action for `*` in `Arit.g4`, re-run `make test` and check that you have no failure.
2. Write one test, in `./tests/test-mult-parents.txt` that checks the proper evaluation of the expression `(1+3)*2`, to make sure you understand the test infrastructure.

In an ideal world, we should also check the behavior in case of syntax error, but ANTLR's error-recovery mechanism makes this a bit too tricky, so we won't do it in this lab.

EXERCISE #10 ► **Add features**
Implement binary and unary minus. Test. Be careful with operators precedence and associativity **to simplify, you can make as if - is prioritary to +**. Unary minus can apply to any expression (`--1` is accepted, with the same meaning as Python). **Of courses, you will have to write adequate tests for these new features.**

1.4 Bonus !

If you have finished in advance, we provide you a bonus exercice in the `bonus/` directory.

[3]This infrastructure is based on pytest, you can have a quick look at the `test_*.py` to understand it. These files do the following: for each test file, execute the code on it, and compare its output with the specification written at the end in comments.

Lab 2
Warm-up : the target machine : RISCV

Objective

- Be familiar with the RISCV instruction set.
- Understand how it executes on the RISCV processor with the help of a simulator.
- Write simple programs, assemble, execute.

2.1 The RISCV processor, instruction set, simulator

EXERCISE #1 ► **Lab preparation**
Save your modifications from the last lab, then pull the lab repository.

EXERCISE #2 ► **Installations - Linux machines at Esisar**
On the Esisar machines, all installations have been done for you. However, you might have to add some stuff in your `PATH`.

EXERCISE #3 ► **Installations - on your machines**
See the `README.md` in the repository.

EXERCISE #4 ► RISCV **C-compiler and simulator, first test**
In the directory `TP_RISCV/startup/`:

- Compile the provided file `ex1.c` with:
  `riscv64-unknown-elf-gcc ex1.c -o ex1.riscv`
  It produces a RISCV binary.
- Execute the binary with the RISCV simulator:
  `spike pk ex1.riscv`
  This should print 42. If you get a runtime exception, try running `spike -m100 pk ex1.riscv` instead: this limits the RAM usage of spike to 100 MB (the default is 2 GB).
- The corresponding RISCV can be obtained in a more readable format by:
  `riscv64-unknown-elf-gcc ex1.c -S -o ex1.s -fverbose-asm`
  (have a look at the generated .s file!)

The objective of the CS444 sequence of labs will be to design **our own (subset of) C compiler for** RISCV.

EXERCISE #5 ► **Documents**
Some documentation about the RISCV ISA can be found on the Chamilo webpage.

EXERCISE #6 ► **A first** RISCV **program**
On paper, write (in RISCV assembly language) a program which initializes the t_0 register to 1 and increments it until it becomes equal to 8.

2.1.1 Assembling, disassembling

EXERCISE #7 ► **Hand assembling, simulation of the hex code**
Assemble by hand (on paper) the instructions:

```
       .globl main
main:
        addi a0, a0, 1
        bne a0, a0, main
end:
        ret
```

You will need the set of instructions of the RISCV machine and their associated opcode. All the info is in the ISA documentation.

To check your solution (**after** you did the job manually), you can redo the assembly using the toolchain:

```
riscv64-unknown-elf-as -march=rv64g asshand.s -o asshand.o
```

`asshand.o` is an ELF file which contains both the compiled code and some metadata (you can try `hexdump asshand.o` to view its content, but it's rather large and unreadable). The tool `objdump` allows extracting the code section from the executable, and show the binary code next to its disassembled version:

```
riscv64-unknown-elf-objdump -d asshand.o
```

Check that the output is consistent with what you found manually.

From now on, we are going to write programs using an easier approach. We are going to write instructions using the RISCV assembly.

2.2 RISCV **Simulator**

EXERCISE #8 ► **Execution and debugging**
See `https://www.lowrisc.org/docs/tagged-memory-v0.1/spike/` for details on the Spike simulator.

`test_print.s` is a small but complete example using Risc-V assembly. It uses the `println_string`, `print_int`, `print_char` and `newline` functions provided to you in `libprint.s`. Each function can be called with `call print_...` and prints the content of register `a0` (`call newline` takes no input and prints a newline character).

1. First test assembling and simulation on the file `test_print.s`:
```
riscv64-unknown-elf-as -march=rv64g test_print.s -o test_print.o
```
2. The `libprint.s` library must be assembled too:
```
riscv64-unknown-elf-as -march=rv64g libprint.s -o libprint.o
```
3. We now link these files together to get an executable:
```
riscv64-unknown-elf-gcc test_print.o libprint.o -o test_print
```
The generated `test_print` file should be executable, but since it uses the Risc-V ISA, we can't execute it natively (try `./test_print`, you'll get an error like `Exec format error`).
4. Run the simulator:
```
spike pk ./test_print
```
The output should look like:
```
bbl loader
HI CS444!
42
a
```
The first line comes from the simulator itself, the next two come from the `println_string`, `print_int` and `print_char` calls in the assembly code.
5. We can also view the instructions while they are executed:
```
spike -l pk ./test_print
```
Unfortunately, this shows all the instructions in `pk` (Proxy Kernel, a kind of mini operating system), and is mostly unusable. Alternatively, we can run a step-by-step simulation starting from a given symbol. To run the instructions in `main`, we first get the address of `main` in the executable:
```
$ riscv64-unknown-elf-nm test_print | grep main
000000000001014c T main
```
This means: `main` is a symbol defined in the `.text` section (T in the middle column), it is global (capital T), and its address is 1014c. Now, run spike in debug mode (`-d`) and execute code up to this address (`until pc 0 1014c`, i.e. "Until the program counter of core 0 reaches 1014c"). Press Return to move to the next instruction and `q` to quit:
```
$ spike -d pk ./test_print
: until pc 0 1014c
bbl loader
```

```
:
core   0: 0x000000000001014c (0xff010113) addi    sp, sp, -16
:
core   0: 0x0000000000010150 (0x00113423) sd      ra, 8(sp)
:
core   0: 0x0000000000010154 (0x0000e517) auipc   a0, 0xe
:
core   0: 0x0000000000010158 (0x41450513) addi    a0, a0, 1044
: q
$
```

Remark: For your labs, you may want to assemble and link with a single command (which can also do the compilation if you provide `.c` files on the command-line):

```
riscv64-unknown-elf-gcc -march=rv64g libprint.s test_print.s -o main
```

In real-life, people run compilation+assembly and link as two different commands, but use a build system like a `Makefile` to re-run only the right commands.

EXERCISE #9 ▸ Algo in RISCV assembly

Write (in fact, complete `minmax.s` between TODO and END TODO) a program in RISCV assembly that computes the min of two integers, and stores the result in a precise location of the memory that has the label `min`. Try with different values. We use 64 bits of memory to store ints, i.e., use `.dword` directive and `ld` and `sd` instructions.

EXERCISE #10 ▸ Routines in assembly (read-only exercise)

In `len.s` we give you an exemple of a routine and a call to this routine (using a stack). Read and explain why we need to save the ra register on the stack. Illustrate for instance by removing the routine prelude and postlude and making a call to an external printing function.

EXERCISE #11 ▸ Caesar code

The following exercise only will be graded. Instructions and deadline on Chamilo too.

Create a file named `str_codecesar.s`, starting with:

```
 # Code de César en RISCV
 # CS444, binôme : NOM1, NOM2
.section .text
.globl main
main:
        addi    sp,sp,-16
        sd      ra,8(sp)

        ...
```

A chain *s* being stored in memory, as well as a *dec* number, compute the Caesar code of the chain: shift every letter value by dec.

Your code should print the input string and the encoded one (thanks to a call to `print_string`. For instance, with the Hello World chain and a dec equal to 4:

```
Hello world!
Lipps${svph%
```

Lab 3
Interpreters and Types

Objective

- Understand visitors.
- Implement typers, interpreters as visitors.

The evaluator will be graded. Instructions and deadline on Chamilo too.

EXERCISE #1 ► **Lab preparation**
`git pull` will provide you all the necessary files for this lab in `TP03` and `MiniC`. ANTLR4 and `pytest` should be installed and working like in Lab 1, if not [1]:

```
python3 -m pip install --user pytest pytest-cov pytest-xdist
python3 -m pip install --user --upgrade coverage
```

3.1 Demo: Implicit tree walking using Visitors

This is a demo and a startup. Nothing has to be put on Chamilo for this section

3.1.1 Interpret (evaluate) arithmetic expressions with visitors

In the first lab, we used an "attribute grammar" to evaluate arithmetic expressions during parsing. Today, we are going to let ANTLR build the syntax tree entirely, and then traverse this tree using the *Visitor* design pattern[2]. A visitor is a way to seperate algorithms from the data structure they apply to.

For every possible type of node in your AST, a visitor will implement a function that will apply to nodes of this type.

EXERCISE #2 ► **Demo: arithmetic expression interpreter (`TP03/arith-visitor/`)**
Observe and play with the `Arit.g4` grammar and its PYTHON Visitor on `myexample` :

```
$ make ; make ex
```

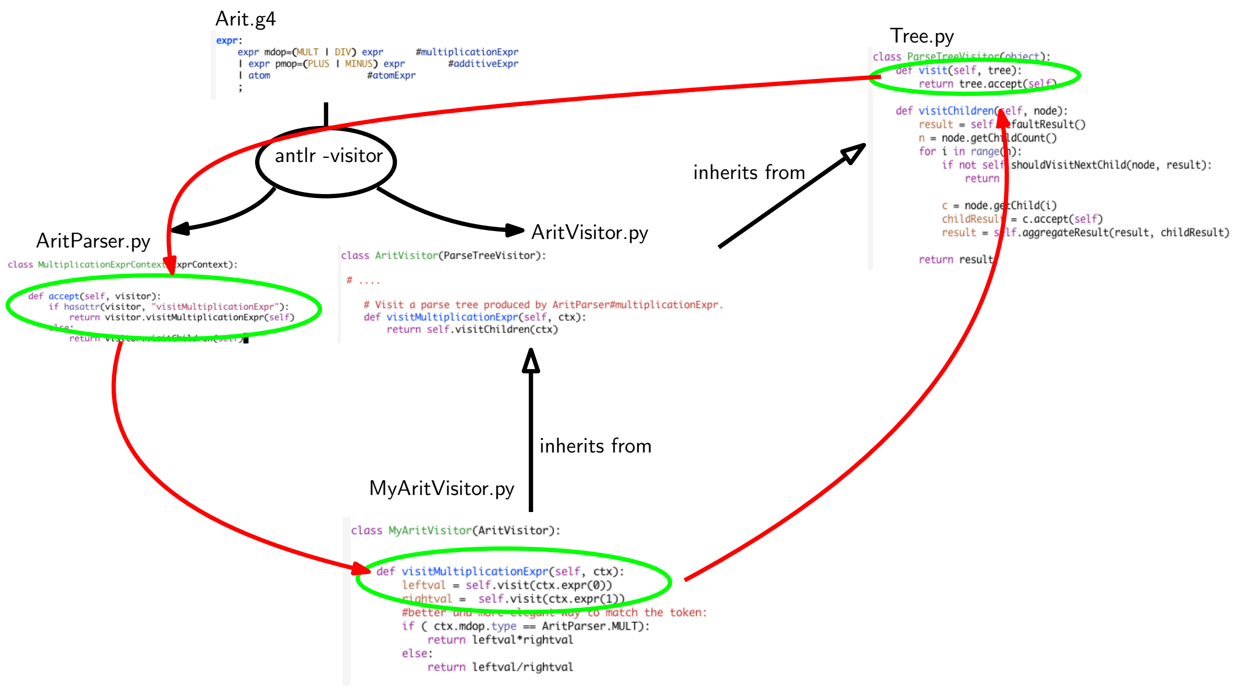
Note that unlike the "attribute grammar" version that we used previously, the `.g4` file does not contain Python code at all.

Have a look at the `AritVisitor.py`, which is automatically generated by ANTLR4: it provides an abstract visitor whose methods do nothing except a recursive call on children. Have a look at the `MyAritVisitor.py` file, observe how we override the methods to implement the interpreter, and use `print` instructions to observe how the visitor actually works (print some node contents).

Also note the `#blabla` pragmas after each rules in the g4 file. They are here to provide ANTLR4 a name for each alternative in grammar rules. These names are used in the visitor classes, as method names that get called when the associated rule is found (eg. `#foo` will get `visitFoo(ctx)` to be called).

We depict the relationship between visitors' classes in Figure 3.1.

[1]The second line is not always needed but may solve compatibility issues between versions of pytest-cov and coverage, yielding `pytest-cov: Failed to setup subprocess coverage` messages in some situations.
[2]`https://en.wikipedia.org/wiki/Visitor_pattern`



```
expr:
    expr mdop=(MULT | DIV) expr      #multiplicationExpr
    | expr pmop=(PLUS | MINUS) expr        #additiveExpr
    | atom                     #atomExpr
    ;
```

```
class MultiplicationExprContext(ExprContext):

    def accept(self, visitor):
        if hasattr(visitor, "visitMultiplicationExpr"):
            return visitor.visitMultiplicationExpr(self)
        else:
            return visitor.visitChildren(self)
```

```
class AritVisitor(ParseTreeVisitor):

 # ....

    # Visit a parse tree produced by AritParser#multiplicationExpr.
    def visitMultiplicationExpr(self, ctx):
        return self.visitChildren(ctx)
```

```
class ParseTreeVisitor(object):
    def visit(self, tree):
        return tree.accept(self)

    def visitChildren(self, node):
        result = self.defaultResult()
        n = node.getChildCount()
        for i in range(n):
            if not self.shouldVisitNextChild(node, result):
                return

            c = node.getChild(i)
            childResult = c.accept(self)
            result = self.aggregateResult(result, childResult)

        return result
```

```
class MyAritVisitor(AritVisitor):


    def visitMultiplicationExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval =  self.visit(ctx.expr(1))
        #better and more elegant way to match the token:
        if ( ctx.mdop.type == AritParser.MULT):
            return leftval*rightval
        else:
            return leftval/rightval
```

Figure 3.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the `ParseTreeVisitor` class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls the highest level `visit`, which itself calls the accept method of the Parser object of the good type (in `AritParser`) which finally calls your implementation of `MyAritVisitor` that match this particular type (here Multiplication). This process is depicted by the red cycle.

Example: when a ANTLR4 rule contains an operator alternative such as:

```
| expr addop=(PLUS | MINUS) expr #additiveExpr
```

you can use the following code in your implementation of `visitAdditiveExpr` to match the operator:

```
if ctx.addop.type == AritParser.PLUS:
    ...
```

To get the list of the two `expr` operands of the rule, you can use `ctx.expr()`. To get e.g. the second `expr` operand, you can use `ctx.expr(1)`. Be careful: if there is only one `expr` (for instance in the `visitParens` case) then `ctx.expr()` (and not `ctx.expr(0)` !) gives you the operand.

In other words, **what you can write in Python code is dictated by the `.g4` file**.

The objective is now to use visitors, to type and interpret MiniC programs, whose syntax is depicted in Figure 3.2.

```
grammar MiniC;

prog: function* EOF #progRule;

// For now, we don't have "real" functions, just the main() function
// that is the main program, with a hardcoded profile and final
// 'return 0'.
function: INTTYPE ID OPAR CPAR OBRACE vardecl_l block
       RETURN INT SCOL CBRACE #funcDef;

vardecl_l: vardecl* #varDeclList;

vardecl: typee id_l SCOL #varDecl;


id_l
    : ID #idListBase
    | ID COM id_l #idList
    ;

block: stat* #statList;

stat
    : assignment SCOL
    | if_stat
    | while_stat
    | print_stat
    ;

assignment: ID ASSIGN expr #assignStat;

if_stat: IF OPAR expr CPAR then_block=stat_block
       (ELSE else_block=stat_block)? #ifStat;

stat_block
    : OBRACE block CBRACE
    | stat
    ;

while_stat: WHILE OPAR expr CPAR body=stat_block #whileStat;


print_stat
    : PRINTLN_INT OPAR expr CPAR SCOL #printlnintStat
    | PRINTLN_FLOAT OPAR expr CPAR SCOL #printlnfloatStat
    | PRINTLN_BOOL OPAR expr CPAR SCOL #printlnboolStat
    | PRINTLN_STRING OPAR expr CPAR SCOL #printlnstringStat
```

Figure 3.2: MiniC syntax. We omitted here the subgrammar for expressions

EXERCISE #3 ► **Be prepared!**

In the directory `MiniC/` (outside `TP03/`), you will find:

- The MiniC grammar (`MiniC.g4`).
- Our "main" program (`MiniCC.py`) which will be the driver of our compiler:

  ```
  python3 MiniCC.py
  usage: MiniCC.py [-h] --mode {parse,typecheck,eval} [--debug]
                   [--disable-typecheck]
                   filename
  ```

  In this lab, you will use `-mode typecheck` or `-mode eval` The driver will launch the parsing of the input file, then the Typing visitor, and if the file is well typed, the Interpreter visitor.
- One complete visitor: `TP03/MiniCTypingVisitor.py`, and one to be completed: `TP03/MiniCInterpretVisitor.py`.
- Some test cases, and a test infrastructure.

3.2 Typing the MiniC-language (`MiniC/`)

The code of the typer is completely given to you. The objective is here to read and understand code. The explicit questions are only to help you in this purpose. They will not be graded.

The informal typing rules for the MiniC language are:

- Variables must be declared before being used, and can be declared only once ;
- Binary operations (+, -, *, ==, !=, <=, &&, ||, ...) require both arguments to be of the same type (e.g. `1 + 2.0` is rejected) ;
- Boolean and integers are incompatible types (e.g. `while(1)` is rejected) ;
- Binary arithmetic operators return the same type as their operands (e.g. `2. + 3.` is a float, `1 / 2` is the integer division) ;
- + is accepted on string (it is the concatenation operator), no other arithmetic operator is allowed for string ;
- Comparison operators (==, <=, ...) and logic operators (&&, ||) return a Boolean ;
- == and != accept any type as operands ;
- Other comparison operators (<, >=, ...) accept int and float operands only.

For now, we do not consider real functions, so all your test cases will contain only a `main` function, without argument and returning an integer. We will extend your code and write test-cases with several function definitions and calls in a further lab.

EXERCISE #4 ▶ **Demo: play with the Typing visitor**

We provide you the code of the typer for the MiniC-language, whose objective is to implement the typing rules of the course. Open and observe `TP03/MiniCTypingVisitor.py`.

Answer the following questions:

- Find the visitor base fonctions. How are their names constructed from the grammar rules (look at the g4 file) ?
- What are the basic types used in our Typer ?
- How is a typing exception handled ?
- Have a specific look at the assignment visitor ("ID=expr"): what is its name in the code ? What it is supposed to do ? Observe how it handles the type of the expression and the type of the variable.

Now predict the behavior of the typer on the following MiniC input:

```
int x;
x="blablabla";
```

type `make` to generate lexer, parser and visitor files. Then, launch the "compiler" in typer mode with:

```
python3 MiniCC.py --mode=typecheck TP03/tests/provided/examples-types/bad_type00.c
```

Observe the behavior of the visitor on all test files in `example-types/` **one by one**, and answer the following questions:

- What is the name of the function that checks for type equality ?
- How do we handle a multiplication between `int` and `string` operands?
- How do we handle an assignement between a variable and an expression with the "wrong" type?
- How to we remember the declared type for each variable? (*symbol table*)?

EXERCISE #5 ▶ **Demo: test infrastructure for bad-typed programs**

On incorrectly typed programs, what we expect from a good test infrastructure is that is is capable of checking if we handled properly the case. This is solved by augmenting the pragma syntax of the previous lab. For instance:

```
int x;
x="blablabla";
// EXPECTED
// In function main: Line 5 col 2: type mismatch for x: integer and string
// EXITCODE 2
```

will be a successful test case. Any error (typing or runtime) must raise the exit code different from 0 (details below). Now, type[3]:

```
make TEST_FILES='TP03/**/*type*.c' test
```

[3]If it takes time, you can *temporarily* remove the pyright rule in the Makefile.

As our makefile rule tests the whole typing and evaluator, it would fail on somes cases. We only test here "bad typing files", for which we already know their typing failure. You should obtain:

```
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type01.c] PASSED [ 16%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type_bool_bool.c] PASSED [ 33%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type04.c] PASSED [ 50%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type00.c] PASSED [ 66%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type03.c] PASSED [ 83%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type02.c] PASSED [100%]
```

If you get an error about the `--cov` argument, you didn't properly install `pytest-cov`.

Note on the -unit- tests The test infrastructure is implemented for you: we test your output and compare to the expected output. When another compiler is available (gcc, or a teacher version), we compare the different outputs.

The test infrastructure gives you a coverage score in order to evaluate how tests cover your code. Have a look on this score during the lab sequence.

The test infrastructure also uses pyright [4] to fast check typing annotations enabled in Python3 languages (cf `https://docs.python.org/3/library/typing.html`). These typing annotations are used in provided code to help you calling functions (enforcing the right types in function calls and returns).

Your own tests You will later add your own tests: add them all in the `tests/students/` directory (mandatorily).

3.3 An interpreter for the MiniC-language

3.3.1 Informal Specifications of the MiniC Language Semantics

MiniC is a small imperative language inspired from C, with more restrictive typing and semantic rules. Some constructs have an undefined behavior in C and well defined semantics in MiniC:

- Variables that are not explicitly initialized in the program are automatically initialized
  - to `0` for `int`,
  - to `0.0` for `float`,
  - to `false` for `bool`,
  - to the empty string `""` for `string`.
- Divisions and modulo by 0 must print the message "Division by 0" and stop program execution with status 1 (use `raise MiniCRuntimeError("Division by 0")` to achieve this in the interpreter).

Note on printing library To allow compiling your MiniC programs with a regular C compiler (for tests, for instance), a `printlib.h` file is provided, and should be `#include`d in all your MiniC test cases[5].

3.3.2 Implementation of the Interpreter

The semantics of the MiniC language (how to evaluate a given MiniC program) is defined by induction on the syntax. You already saw how to evaluate a given expression, this is depicted in Figure 3.3.

EXERCISE #6 ► **Interpreter rules (on paper)**
First fill the empty cells in Figure 3.4, then ask your teaching assistant to correct them.

EXERCISE #7 ► **Interpreter**
Now you have to implement the interpreter of the MiniC-language. We give you the structure of the code and the implementation for numerical expressions and boolean expressions (`except modulo!`). You can reason in terms of "well-typed programs", since badly typed programs should have been rejected earlier.

Type:

[4] `https://github.com/microsoft/pyright`)
[5] Note that unlike real C, # is a comment in MiniC to avoid actually having to deal with `#include`

literal constant c	`return int(c) or float(c)`
variable name x	`find value of x in dictionary and return it`

e_1+e_2	`v1 <- e1.visit()`
	`v2 <- e2.visit()`
	`return v1+v2`

true	`return true`
$e_1 < e_2$	`return e1.visit()<e2.visit()`

Figure 3.3: Interpretation (Evaluation) of expressions

x := e	`v <- e.visit()`
	`store(x,v) #update the value in dict`

println_int(e)	`v <- e.visit()`
	`print(v) # python's print`

S1; S2	`s1.visit()`
	`s2.visit()`

`if` b `then` $S1$ `else` $S2$	
`while` b `do` S `done`	

Figure 3.4: Interpretation for Statements

```
make
python3 MiniCC.py --mode=eval TP03/tests/provided/examples/test_print_int.c
```

and the interpreter will be run on `test_print_int.c` (it should print 42 on this particular exemple). **On the particular example `test_print_int.c` observe how integer values are printed.**

Open the interpretor source code(`MiniCInterpretVisitor.py`), and answer the following questions to understand the code:

- How is the memory initialised ? (compare to the memorytypes in the typer)
- Which values types can have a mini C expression (line 7)?
- How are string variables initialized in our interpreter?

TODO You still have to implement (in `MiniCInterpretVisitor.py`) [6]:

1. The modulo version of Multiplicative expressions (be careful to raise an exception if the operation is not valid). **The expected behavior for modulo is the same than in Python, in which `%` can have 2 floating arguments:**
   ```
   laure@vernet:~/$ python3
   Python 3.10.6 (main, Aug 10 2022, 11:40:04) [GCC 11.3.0] on linux
   Type "help", "copyright", "credits" or "license" for more information.
   >>> 3%2
   1
   >>> 3%2.9
   0.10000000000000009
   >>> 8.9%2.9
   0.20000000000000062
   >>> 9.2%0
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   ZeroDivisionError: float modulo
   ```
2. Variable declarations (`varDecl`) and variable use (`idAtom`): your interpreter should use a table (*dict* in PYTHON) to store variable definitions and check if variables are correctly defined and initialized. **Do not forget to initialize dict with the initial values (`0`, `0.0`, `False` or `""` depending on the type) for all variable declarations.** Refer to the test files `bad_xxx.c` for the expected error messages.
3. Statements: assignments, conditional blocks, tests, loops.

Test your implementation after each new feature:

```
python3 MiniCC.py --mode=eval /path/to/example/.c
```

Error codes The exit code of the interpreter should be:

- 1 in case of runtime error (e.g. division by 0, absence of main function)
- 2 in case of typing error
- 3 in case of syntax error
- 4 in case of internal error (i.e. error that should never happen except during debugging)
- 5 in case of unsupported construct (should not be used in lab3, but you will need it for strings and floats during code generation)
- And obviously, 0 if the program is typechecked and executed without error.

The file `MiniCInterpreter.py` in the skeleton already does this for you if you raise the right exception. These are also the values you will have to use in the `// EXITCODE` directives in your tests.

EXERCISE #8 ► Automated tests for final delivery

The rule `make test-interpret` launches everything. We recall that you can use the `TEST_FILES` makefile variable to launch only on a subset of your files.

You must provide your own tests. The only outputs are the one from the `println_*` function or the following error messages: "`m has no value yet!`" (or possibly "`Undefined variable m`", but this error should

[6] Search for `NotImplementedError()`, they should all be removed for the source code.

never happen if the typechecker did its job properly) where *m* is the name of the variable. In case the program has no `main` function, the typechecker accepts the program, but it cannot be executed, hence the interpreter raises a “`No main function in file`” error.

Test Infrastructure - a bit more explanation Tests work mostly as in the previous lab, with `// EXPECTED` and `// EXITCODE` *n* pragmas in the tests. They are special comments (the `//` is needed to keep compatibility with C, only the testsuite considers them as special). The `EXITCODE` corresponds to the exit codes described in Section 3.3.2.

For instance, if you fail `test_print_int.c` because you printed `43` instead of `42`, using the command
`make test TEST_FILES='TP03/tests/provided/examples/test_print_int.c'`
you will get this error:

```
_____________ TestCodeGen.test_expect[/path/test_print_int.c] _____________

self = <test_interpreter.TestCodeGen object at 0x7f0e0aa369b0>
filename = '/path/to/test_print_int.c'

    @pytest.mark.parametrize('filename', ALL_FILES)
    def test_expect(self, filename):
        expect = self.extract_expect(filename)
        eval = self.evaluate(filename)
        if expect:
>           assert(expect == eval)
E           assert '43\n1\n' == '42\n1\n'
E             - 43
E             + 42
E               1

test_interpreter.py:59: AssertionError
```

And if you did not print anything at all when `42` was expected, the last lines would be this instead:

```
        if expect:
>           assert(expect == eval)
E           assert '42\n1\n' == '1\n'
E             - 42
E               1

test_interpreter.py:59: AssertionError
```

Lab 4
Syntax-Directed Code Generation

Objective

During the previous lab, you have written your own interpreter of the MiniC language. In this lab the objective is to generate *valid* RISCV codes from MiniC programs:

- Generate 3-address code for the MiniC language.
- Generate executable "dummy" RISCV from programs in MiniC via two simple allocation algorithms.
- **Please follow instructions and COMMENT YOUR CODE!**

Student files are in the Git repository.

Startup On the Esisar machines

1. copy your former saved bashrc.
2. edit your bashrc to use Python 3.10:
   `PATH=/opt/bin:$PATH`

All : Make sure your Git repository is up-to-date, using `git pull`. (We already pushed the directory for next lab `MiniC/TP05`, ignore it)

4.1 Preliminaries

This section must be read **carefully**.

Important remark From now on, we add the following restriction to the MiniC language: Values (variables, argument of `println_int`) are of type (signed) `int` or `bool` only (no float, no string). Thus all values can be stored in regular registers or in one cell (64 bits) in memory. You can let your program crash (`raise MiniCUnsupportedError(...)`) if another type of variable is provided.

Note that real compilers would perform the code generation from a decorated AST (with type annotations attached to nodes). For simplicity, we will work on the non-decorated AST: our language is simple enough to generate code without decorations.

Structure of the compiler's code In the `MiniC/Lib` folder, we provide you with many utility functions. A detailed documentation of the library is given in the repository, you can access it offline by opening `docs/index.html` in a web browser.

As for other files in the MiniC directory:

- `TP04/MiniCCodeGen3AVisitor.py` is the code generation algorithm, implemented as a visitor.
- The main Python file, `MiniCC.py` as in lab3, now accepts new options related to code generation (check `python3 MiniCC.py --help` for a full list). Running `python3 MiniCC.py --mode codegen-linear <file>` launches the chain: production of 3-address code with temporaries, allocation, replacement, print.
- The script `test_codegen.py` will help you test your code. We will use it in Section 4.3 through Makefile targets.
- The `README-codegen.md` file is to be completed progressively during the lab.

EXERCISE #1 ► RISCV **Simulator - test**
Re-test the command-line version of the RISCV simulator, for example with code given in the `TP_RISCV` directory.

```
cd ../TP_RISCV/riscv/
riscv64-unknown-elf-gcc libprint.s test_print.s -o test_print.riscv
spike -m100 pk test_print.riscv
cd ../../MiniC/
```

4.1.1 Conventions used in the assembly code

- All data items are stored on 64 bits (double-words, 8 bytes).
- Registers `s1`, `s2`, and `s3` are reserved for temporary computations (e.g. to compute an address before or after an `sd` or a `ld`, or to store a value between a memory access and an arithmetic operation). Note that `s0` is an alias for `fp`, hence `s0` must not be used as a general purpose register either.
- Registers `s4`, ..., `s11`, `t0`, ..., `t6` are general purpose registers, that can be used freely by the code generator. In your Python code, you can access the list of general-purpose registers with `Operands.GP_REGS`. `s`*i* and `t`*i* registers will behave differently in presence of function calls, but are considered equivalent for us, since we don't deal with functions.
- To store properly in memory, it is mandatory to compute offsets from the "reserved" register `fp`. To be compatible with the RISCV ecosystem, we will use a stack **growing with decreasing addresses**. Thus data in the stack is accessed by adding a **negative offset** (multiple of 8) to `fp`. In other words, we use the memory locations `-8(fp)`, `-16(fp)`, ... The `sp` register points to the first data contained in the stack. It is always 16-byte (2 double-words) aligned.
- Registers `a1` to `a7` are not used at all

EXERCISE #2 ▶ **Understand the library**
Look at the library files and find registers, and understand the data structure behind a riscv code. Carefully look at the typing idioms in Python (perhaps you should refer yourself to `https://coderslegacy.com/python/typing-library/`), for instance to understand what an union is.

4.1.2 Conventions used in the testsuite

A few reminders and new features of the testsuite:

- Test files should contain directives giving the expected behavior:
  - `// EXPECTED` and the following lines to give the expected output;
  - `// EXITCODE` *n* gives the expected return code of the compiler, i.e. `// EXITCODE 1` when the code should be rejected by your typechecker (see previous lab for the specification of different exit codes);
  - `// SKIP TEST EXPECTED` to specify that this test should not be ran through `test_expect` (see below);
- Several tests can be run on each `.c` files:
  - `test_expect`, that compiles the file using `riscv64-unknown-elf-gcc`. It checks that `EXPECTED` directives are correct, but doesn't test your compiler.
  - `test_naive_alloc`, `test_alloc_mem`, `test_smart_alloc` that compiles the file using your compiler, using the corresponding register allocation algorithm. The testsuite leaves generated `.s` files next to the `.c` source file.

4.2 First step: three-address code generation

In this section you have to implement the course rules in order to produce RISCV code with temporaries. These rules are given in Figure 4.2 on page 29 and Figure 4.3 on page 30.

Here is an example of the expected output of this part. From the following MiniC program:

```
#include "printlib.h"

int main() {
    int a,n;
    n = 1;
    a = 7;
    while (n < a) {
      n = n+1;
    }
    println_int(n);
    return 0;
}
```

the following code is supposed to be generated.

```
##Automatically generated RISCV code, MIF08 & CAP
##non executable 3-Address instructions version


##prelude
# [...] Some automatically generated code that will be explained in a future lab

##Generated Code
# [...] Some automatically generated code that will be explained in a future lab
        li temp_0, 0
        li temp_1, 0
        # (stat (assignment n = (expr (atom 1))) ;)
        li temp_2, 1
        mv temp_0, temp_2
        # (stat (assignment a = (expr (atom 7))) ;)
        li temp_3, 7
        mv temp_1, temp_3
        # (stat (while_stat while ( (expr (expr (atom n)) < (expr (atom a))) ) (
    stat_block { (block (stat (assignment n = (expr (expr (atom n)) + (expr (atom 1)))
    ) ;)) })))
lbl_begin_while_1_main:
        li temp_4, 0
        bge temp_0, temp_1, lbl_end_relational_3_main
        li temp_4, 1
lbl_end_relational_3_main:
        beq temp_4, zero, lbl_end_while_2_main
        # (stat (assignment n = (expr (expr (atom n)) + (expr (atom 1)))) ;)
        li temp_5, 1
        add temp_6, temp_0, temp_5
        mv temp_0, temp_6
        j lbl_begin_while_1_main
lbl_end_while_2_main:
        # (stat (print_stat println_int ( (expr (atom n)) ) ;))
        mv a0, temp_0
        call println_int
# [...] Some automatically generated code that will be explained in a future lab

##postlude
# [...] Some automatically generated code that will be explained in a future lab
```

EXERCISE #3 ▸ **3-address code generation**

In the skeleton, we provide you an incomplete `MiniCCodeGen3AVisitor.py` (find the TODOs). To test it, type

`python3 MiniCC.py --mode codegen-linear TP04/tests/provided/step1/test00.c --reg-alloc=none`
Don't forget to run `make` before if you need to regenerate the lexer and parser with ANTLR (i.e. if `python3` complains with `No module named 'MiniCLexer'`). Observe the generated code in `<samepath>/test00.s`[1]. You now have to implement the 3-address code generation rules seen in the course. Code and test incrementally[2]:

- We give you the code generation for the `println_int` instruction. It basically produces a call to the proper function in the library.
- Implement numerical expressions without variables (constants are expected to hold on 64 bits, no boolean expression for now). We advise you to postpone the implementation of MultiplicativeExpr, and first finish this Lab without them (details are given section 4.6).
- Then check that (numerical) expressions with variables work (assignment and usage of variables in expressions are given);

At this step, the code generation is not finished. In the next steps, we will do some allocation to be able to test properly. All examples in `tests/provided/step1` directory should generate code without any error at this point:

```
for i in TP04/tests/provided/step1/*.c; do
  echo "file="$i; python3 MiniCC.py --mode codegen-linear --reg-alloc=none $i >/dev/null;
done
```

4.3 Testing with the trivial allocator (and real RISCV instructions)

The code generated at this point is not executable since it uses temporaries. We provide you with an allocation method which allocates temporaries in registers as long as possible, and fails if there is no more available registers. The process takes as input the former 3-address code and transforms each instruction according to the allocation function.

EXERCISE #4 ► **Testing the trivial allocator**
Open, read, understand the `NaiveAllocator` implementation in `Lib/Allocator.py` (`https://drup.github.io/cap-lab22/api/Lib.Allocator.html#Lib.Allocator.NaiveAllocator`) and how it is used to perform the actual RISCV code generation [3]. Then, intensively test your former code generation with this allocator [4]:

```
make TEST_FILES="TP04/tests/provided/step1/*.c" test-naive MODE=codegen-linear
```

This script tests all files specified in `TEST_FILES` (or, if not specified, all files in the `*/tests/*` directories except those whose name start with a special character):

- if the pragma `// EXPECTED` is present in the file, it compares the actual output after assembling and simulating with the list of expected values. For instance:

```
int main(){
  int x, y;
  x = 42;
  println_int(x);
  y = x + 8;
  println_int(y);
  return 0;
}
// EXPECTED
// 42
// 50
```

  is an example test case to test assignments.

[1]We generated RISCV comments with MiniC statements for debug.
[2]Using files in the `TP04/tests/*` directories. All the test files you use will have to be in your archive.
[3]All available registers are in a list named `GP_REGS`.
[4]Be careful, this allocator fails if there is more than a certain number of temporaries!

- If the `AllocationError` exception is raised by the naive allocator, the test is considered "skipped" (i.e. it's not a failure, but not really a success either, we can't conclude; the same test case will be used for other allocation strategies).
- If the compilation succeeded, it compares the actual output after assembling and simulating to the `// EXPECTED` statements given in the file (which are themselves compared to the output given by `riscv64-unknown-elf-gcc`).
- For debugging, you can obviously launch your compiler manually with e.g.
  ```
  pyright &&
  python3 MiniCC.py --mode codegen-linear --reg-alloc naive --stdout <file>
  ```
  Run `python3 MiniCC.py --help` or see `MiniCC.py` for more options. The `--debug` option allows getting some debug output. Alternatively, you can run the testsuite on a single test file with:
  ```
  make TEST_FILES=TP04/tests/provided/test_while2b.c test-naive MODE=codegen-linear
  ```
- When making tests with `make test-naive`, a coverage of your code is created in a folder `htmlcov`. You can look at the file `TP04_MiniCCodeGen3AVisitor_py.html` to check which part of your code has been executed during the tests. If some lines of code you wrote have been missed during the tests, then you must write your own tests for these parts!

At this step, the tests should be OK or SKIPPED for all files given in directory `tests/step1/`:
```
make test-naive MODE=codegen-linear
[...]
============================ xx passed, xx skipped in xx seconds ========
```
Now that we have a way to test our code generation for tiny MiniC codes, we can come back to it.

4.4 Finish 3 address code generation

Now that you know how to test your code using the naive allocator, go back to code generation and finish it.

EXERCISE #5 ► **A few corner-cases**
Some points may require extra care, in the implementation or in the tests:

- Don't forget the automatic initialization (in MiniC, unlike real C). Unlike the interpreter, initialization cannot be done by initializing a Python dictionary. Make sure the initialization code is properly generated.
- Don't forget the explicit errors for division by zero. We provide you a piece of assembly code raising the error (see `LinearCode.print_code()` and follow the trail), you need to generate the instruction to jump to this label (we get the label with `self._current_function.fdata.get_label_div_by_zero()`) when the right operand of a division or modulo is 0.
- `float` and `string` are unsupported. The compiler raises `MiniCUnsupportedError` when encountering any of them. Tests are provided for this.

Note that testing the division by 0 requires a bit of attention. We need to check that the executable exits with code 1 at runtime, that the output is correct, but we can't check that GCC gives the same behavior because GCC doesn't give a clean error message. A test case may therefore be:
```
#include "printlib.h"

int main(){
        println_int(1 / 0);
        return 0;
}
// SKIP TEST EXPECTED
// EXECCODE 1
// EXPECTED
// Division by 0
```

EXERCISE #6 ▸ **End of 3-address code generation for MiniC**
Implement the 3-address code generation rules:

- for boolean expressions and numerical comparison: compute 1 (true) or 0 (false) in the destination register; be careful the `not` boolean instruction is not as easy as you wish;
- while loops;
- if then else.

At this point all the tests should be ok for all files in directory `TP04/tests/provided/step2/`. However these tests are not sufficient, you should add some other ones (in the directory `TP04/tests/students/`). Run the testsuite with `make test-naive MODE=codegen-linear` to use all the test files.

About `if` and `while` For tests (and boolean expressions), make sure you generate "conditional jumps" with:

```
self._current_function.add_instruction(
          RiscV.conditional_jump(label, op1, cond, op2))
```

where `op1` (resp `op2`) is the left operand (resp right operand or the numerical constant 0, nothing else), i.e. a register or a value of the boolean condition `cond` (`Condition('beq')` for equality, for instance) [5], and `label` is a label to jump to if the condition evaluates to true.

4.5 RISCV code with "all-in-mem" allocation of temporaries

Tests Up to now, you used `make test-naive MODE=codegen-linear` to test your code, and at this point all tests should pass, or be skipped (do not forget to make a test where the naive allocation uses too many registers!) From now on, you should use the more complete `make test-lab4 MODE=codegen-linear` command, that tests everything with the provided naive allocator, and the all-in-memory allocator you have to write. If you use `MiniCC.py` directly, the corresponding option is `--reg-alloc=all-in-mem`.

Check that `make test-lab4 MODE=codegen-linear` does fail.

Implementation As the number of registers for allocation is bounded by the number of available general purpose registers, i.e. `len(Operands.GP_REGS)`, the naive allocator cannot deal with more temporaries than general-purpose registers: we have to find a way to store the results elsewhere. In this particular lab, we will use the following solution:

- The generated code will use memory locations in the stack.
- All values that are propagated from one rule to another (sub-expressions, ...) must be stored in the stack, whose address will be stored in FP.
- $s1, s2, s3$ will be used to compute the value to store or as a destination register for the value(s) to read. **Technically, only 2 of these registers are mandatory, but you should be cautious if you try a 2-registers-only solution.**
- In order to know if a given (temporary) operand should be read and/or written, use the `is_read_only` method of the `Instruction` class.

Figure 4.1 depicts the stack implementation for the RISCV machine, that follows the RISC-V calling convention (stack growing downwards, stack-pointer always 16-bytes aligned).

Following the convention that `fp` always stores the "begining of stack address", pushing the content of register $s3$ in the stack will be done following the steps:

- compute a new offset (call to the `fresh_offset` method).
- generate the following instruction:

```
sd s3, -offset*8(fp)
# sd = store double = 64-bits store
# -offset*8(fp) = memory location at address fp-offset*8
```

Getting back the value is similar.

[5] We suggest to use `grep` and find this class definition and this method somewhere in the library we provide.

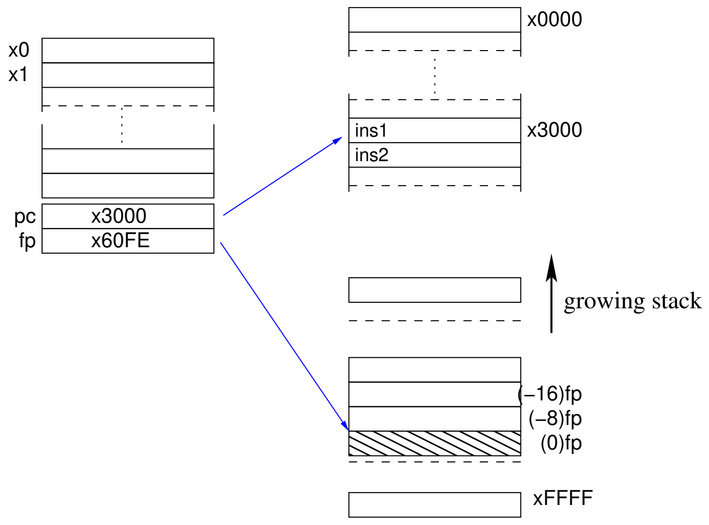


Figure 4.1: Memory model for RISCV

EXERCISE #7 ► **Manual translation**
Complete the expected output for the following two statements (13/15 lines of RISCV code). The temporary `temp_3` is located at −32(fp) and `temp_4` is located at −40(fp):

```
int x, y;
x=4;
y=12+x
```

Listing 4.1: 'all in mem alloc for test_while2b.c'

```
##Generated code without prelude and postlude
        # (stat (assignment x = (expr (atom 4))) ;)
        # li temp_2, 4
        li s2, 4
        sd s2, -24(fp)
        # end li temp_2, 4
        # mv temp_1, temp_2
        ld s1, -24(fp)
        mv s2, s1
        sd s2, -16(fp)
        # end mv temp_1, temp_2
        # (stat (assignment y = (expr (expr (atom 12)) + (expr (atom x)))) ;)
        # li temp_3, 12
        # TODO 2 lines


        # end li temp_3, 12
        # add temp_4, temp_3, temp_1
        # TODO 4 lines




        # end add temp_4, temp_3, temp_1
        # mv temp_0, temp_4
        # NOT TODO
```

EXERCISE #8 ▸ **Implement**
Now you are on your own to implement this code generation. The relevant file is `TP04/AllInMemAllocator.py`. Here are the main steps (less than 50 locs of PYTHON):

1. We have implemented for you an `AllInMemAllocator.prepare()` method. This method only maps each temporary to a new offset in memory (in a PYTHON `dict`), allowing you to use the method `get_alloced_loc()` on an Temporary used in the code.
2. Complete the method `AllInMemAllocator.replace(old_instr)` that takes as input a "3-address with temporaries" RISCV code and outputs a list of instructions as a replacement. For instance, each time we access a source operand, we have to load it from memory before, thus the `replace` should contain something like

```
# regxxx is the register used to hold the value between the load and
# the operation itself (one of s1, s2, s3).
# loc is the place in memory where the temporary is allocated (of
# the form Offset(..., fp), obtained with get_alloced_loc().
before.append(RiscV.ld(regxxx, loc))
```

The files you generate have to be tested with the RISCV simulator with the same script as before. **Of course, with "all-in-mem" allocation, tests that were "skipped" due to the lack of registers with the naive allocation should not be skipped any more.**

More tests Now that your compiler can deal with a large number of temporaries, make sure all features are heavily tested (the testsuite we provide is in no way sufficient).

4.6 Multiplicative Expressions (multiplication, division, modulo)

EXERCISE #9 ▸ **3-address code generation for multiplicative expressions**
If not already done, extend your work to multiplicative expressions. Conventions for division and multiplication should be the same as in C: division is truncated toward zero, and modulo is such that $(a/b)*b+a\%b = a$.

$$\begin{array}{rcrcrcr} 4/3 & = & 1 & \quad & 4\%3 & = & 1 \\ (-4)/3 & = & -1 & & (-4)\%3 & = & -1 \\ 4/(-3) & = & -1 & & 4\%(-3) & = & 1 \\ (-4)/(-3) & = & 1 & & (-4)\%(-3) & = & -1 \end{array}$$

4.7 Delivery

This lab will be graded, but you have more than the habitual amount of time to finish it. Instructions will follow on Chamilo.

EXERCISE #10 ▸ **Delivery - grading criteria**
Python code and C testcases will be graded. We recall that your work is **personal (pairs are allowed)** and code copy from any source or sharing is **strictly forbidden**. As usual, upload an archive containing the whole `MiniC` directory (`make tar` does that for you).

Do not forget to edit `README-gencode.md`.

Evaluation criteria:

- Correctness of your code generator
- Correctness of your allocator
- Correctness of the annotations (`// EXPECTED`, ...) in your test files.
- Coverage of your testsuite (on the `MiniCCodeGen3AVisitor.py` and `SimpleAllocations.py` source files)

c	`dest <- fresh_tmp()`
	`code.add("li dest, c")`
	`return dest`

x	`# get the temporary associated to x.`
	`reg <- symbol_table[x]`
	`return reg`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dest <- fresh_tmp()`
	`code.add("add dest, t1, t2")`
	`return dest`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dest <- fresh_tmp()`
	`code.add("sub dest, t1, t2")`
	`return dest`

true	`dest <-fresh_tmp()`
	`code.add("li dest, 1")`
	`return dest`

$e_1 < e_2$	`dest <- fresh_tmp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`endrel <- new_label()`
	`code.add("li dest, 0")`
	`# if t1>=t2 jump to endrel`
	`code.add("bge endrel, t1, t2")`
	`code.add("li dest, 1")`
	`code.addLabel(endrel)`
	`return dest`

Figure 4.2: 3@ Code generation for numerical or Boolean expressions

x = e	`dest <- GenCodeExpr(e)`
	`loc <- symbol_table[x]`
	`code.add("mv loc, dest")`

S1; S2	`# Just concatenate codes`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` b then $S1$ `else` $S2$	`lelse <- new_label()`
	`lendif <- new_label()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add("beq lelse, t1, 0")`
	`GenCodeSmt(S1) # then`
	`code.add("j lendif")`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) # else`
	`code.addLabel(lendif)`

`while` b do S done	`ltest <- new_label()`
	`lendwhile <- new_label()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add("beq lendwhile, t1, 0")`
	`GenCodeSmt(S) # execute S`
	`code.add("j ltest") # and jump to the test`
	`code.addLabel(lendwhile) # else it is done.`

Figure 4.3: 3@ Code generation for Statements

Lab 5
Code generation with smart IRs

Objective

- Understand the CFG construction.
- Compute live ranges, construct the interference graph.
- Allocate registers and produce final "smart" code.
- Two sessions for this lab.
- **Lab delivery: with Lab4, all together, on Chamilo. Instructions and deadline on Chamilo**. Do not forget massive tests (on each step of the smart allocation.)

5.1 Preliminaries

(save your lab4 somewhere before doing anything else). First, get the latest version of the skeleton with:

```
git pull
```

If you get painful conflicts, you may re-clone the whole repository (see lab 1).

Be careful to not crash your last Lab (especially the file : `TP04/MiniCCodeGen3AVisitor.py`). Some files should be updated in this directory.

During the previous lab, you wrote a dummy code generator for the MiniC language. In this lab the objective is to generate a more efficient RISCV code.

You will extend your previous code (in the second part of the lab, see Section 5.4, in the same `MiniC` project, but in the `TP05/` subdirectory

Installations We are going to use graphviz for visualization. If it is not already installed (e.g. on your personal machine), install it, for instance with:

```
sudo apt-get install graphviz graphviz-dev
```

You may have to install the following PYTHON packages:

```
python3 -m pip install --user networkx
python3 -m pip install --user graphviz
python3 -m pip install --user pygraphviz
```

If the last command errors out complaining about a missing `Python.h`, run:

```
sudo apt-get install python3-dev
```

and then relaunch the command `python3 -m pip install ...` On the Esisar machines, you might have to update existing already installed packages:

```
python3 -m pip install --user --upgrade networkx graphviz pygraphviz
```

5.2 CFG construction

The CFG construction is given, you only have to understand how it is implemented. (The code to observe is in the `MiniC/TP04/` directory).

During class we presented Control Flow Graphs with maximal basic blocks. In this section we will read and understand how this CFG generation is implemented.

During this phase, the linear code produced in the 3-address code generation produces a graph.

EXERCISE #1 ► **CFG By hand**
What is the expected result of the CFG construction for each of these programs?

Listing 5.1: `df01.c`

```
int n,u,v;
n=6;
u=12;
v=n+u;
print_int(v);
```

Listing 5.2: `df04.c`

```
int x;
x=2;
if (x < 4)
   x=4;
else
   x=5;
print_int(x)
```

Listing 5.3: `df05.c`

```
int x;
x=0;
while (x < 4){
   x=x+1;
}
```

5.2.1 Understand provided code

In `TP04/BuildCFG.py` we give you the entire code for the CFG construction (whose datastructure is defined in the Lib).

EXERCISE #2 ► **Finding the leaders**
In the course on intermediate representations, we have defined the notion of *basic blocks* and *leaders*, which designate the indices of the instructions starting a block. We define the `_find_leaders` procedure as taking the list of instructions and returning a list of leaders. The list of indices `leaders` should have the following properties:

- `leaders[i]` is the starting instruction of block i.
- Each interval `leaders[i]` to `leaders[i+1]-1` delimits the instructions of a block.
- We have `leaders[0]=0` (first block) and `leaders[-1]=len(instructions)`.
- There are no duplicated indices in the list.

Compute the leaders by hand on the following example.

```
0 subi temp2, temp2, 4
1 beq temp2, zero, lelse1
2 li temp4, 7
3 mv temp1, temp4
4 jump lendif1
5 lelse1:
6 addi temp3, temp2, 1
7 mv temp1, temp3
8 lendif1:
```

EXERCISE #3 ► **Understand CFG Construction (file `TP04/BuildCFG.py`)**
The `CFG` file contains all the utilities related to Control Flow Graphs:

- the `Block` class, representing a basic block,
- the `CFG` class, representing a complete function in CFG form.

`Blocks` have a list of predecessors (`self._in`) and successors (accessible via the Terminator of the block and the function `CFG.out_blocks(block)`) and a `CFG` contains the initial control point (`self._start`) from which we can traverse the graph. This feature allows us to construct the CFG of a program.

The procedure to build the CFG is split into several pieces. The `__init__` function builds the class and sets all utility counters. The `_find_leaders` function returns a list of all the leaders. The `_add_blocks` function populates the control flow graph with the blocks extracted using the list of leaders.

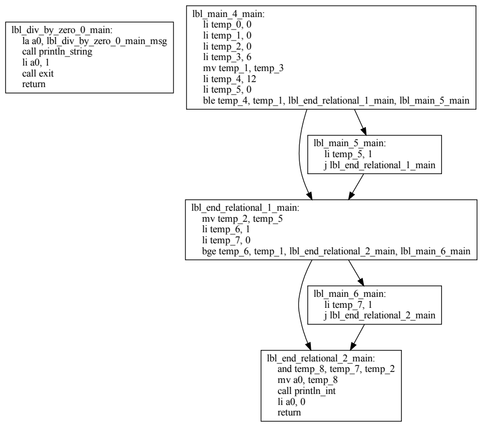
- Is the `_find_leaders` implementation conform to the one of the course?
- How is the chaining realized in the case of forward reference ? backward reference ?

EXERCISE #4 ► **Check and test your CFGs**
When run with `--graphs`, `MiniCC.py` prints the CFG as a PDF file (using the tool "dot"). The file is printed as `<name>.dot.pdf` in the same directory as the source file and opened automatically.

Check the output for all files in `TP04/tests/provided/dataflow/` with the following command line:

```
python3 MiniCC.py --mode codegen-cfg --reg-alloc naive \
--graphs TP04/tests/provided/dataflow/df02.c
```

For example, the CFG for `df02.c` should look like:



```
lbl_div_by_zero_0_main:
    la a0, lbl_div_by_zero_0_main_msg
    call println_string
    li a0, 1
    call exit
    return
```

```
lbl_main_4_main:
    li temp_0, 0
    li temp_1, 0
    li temp_2, 0
    li temp_3, 6
    mv temp_1, temp_3
    li temp_4, 12
    li temp_5, 0
    ble temp_4, temp_1, lbl_end_relational_1_main, lbl_main_5_main
```

```
lbl_main_5_main:
    li temp_5, 1
    j lbl_end_relational_1_main
```

```
lbl_end_relational_1_main:
    mv temp_2, temp_5
    li temp_6, 1
    li temp_7, 0
    bge temp_6, temp_1, lbl_end_relational_2_main, lbl_main_6_main
```

```
lbl_main_6_main:
    li temp_7, 1
    j lbl_end_relational_2_main
```

```
lbl_end_relational_2_main:
    and temp_8, temp_7, temp_2
    mv a0, temp_8
    call println_int
    li a0, 0
    return
```

Observe how straight-line code, if statements, while loops are converted.

Important remark Note that register allocation does not affect the CFG printed by `--graphs`, as it is output before register allocation. In the rest of the lab, your compiler should do the allocation on the CFG, and all the tests from the previous lab should still pass (with `-reg-alloc all-in-mem` option).

5.3 Liveness analysis and Interference graph

Be prepared! From now on, we work in `MiniC/TP05/` directory. You should ignore the file LivenessSSA.py, which will not be used this year.

In order to implement the "smart allocation", you have to modify `MiniC/TP05/SmartAllocator.py`.You already used `NaiveAllocator` and `AllInMemAllocator` in the previous lab (the mapping from temporary to physical register or memory location was provided to you, and you had to modify the 3 address code to take this mapping into account). We will now write the `SmartAllocator`, that maps temporaries to physical registers in an optimized way, and use memory (spilling) only when necessary. Read the body of `SmartAllocator.prepare()`, that gives the main steps of the allocation:

- liveness dataflow analysis,
- conflict graph,
- graph coloring
- and finally 3 address code modification to get the final executable.

Similarly to previous lab, `SmartAllocator.rewriteCode()` then applies the allocation to the generated code. To help you with debugging, a `raise NotImplementedError` statement in `prepare` stops the execution. Move it down as you progress in the lab, and remove it completely when you are done.

For the liveness analysis, we recall the notations. A variable at the left-hand side of an assignment is *killed* by the block. A variable whose value is used in this block (before any assignment) is *generated.*

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.

EXERCISE #5 ▸ **Liveness Analysis, Implement the Initialisation**

In this exercice, you have to complete the method `set_gen_kill()` of the `CFG.Block` class. This method is called for each block by `LivenessDataFlow.set_gen_kill()` and initialises the Gen(B) and Kill(B) sets for each basic block (add, let, ...) in the program. To help you, each instruction provides a `.defined()` (respectively `.used()`) method that returns the set of temporaries assigned (respectively used) in the instruction. Be careful to properly handle the following cases:

```
addi temp1, temp1, 12
```

and

```
bge temp_1, temp_3, lbl_foo  # temp_1 is read from
```

To test/debug this initialisation, the following statements in `SmartAllocator.py` (in `SmartAllocator.prepare()`) should help you (use with `MiniCC.py --debug`, which sets `debug=True` for you):

```
    if self._debug:
        self._liveness.print_gen_kill()
```

As an example, once initialization is implemented, running the command (without –graphs!)
`python3 MiniCC.py --mode codegen-cfg --debug  --reg-alloc smart TP05/tests/provided/dataflow/df02.c`
should give the expected initialisation for `TP05/tests/provided/dataflow/df02.c`:

```
Dataflow Analysis, Initialisation
block lbl_div_by_zero_0_main : 0
gen: {}
kill: {}

block lbl_end_relational_2_main : 1
gen: {temp_7,temp_2}
kill: {temp_8}

block lbl_main_6_main : 2
gen: {}
kill: {temp_7}

block lbl_end_relational_1_main : 3
gen: {temp_1,temp_5}
kill: {temp_6,temp_7,temp_2}

block lbl_main_5_main : 4
gen: {}
kill: {temp_5}

block lbl_main_4_main : 5
gen: {}
kill: {temp_0,temp_2,temp_3,temp_4,temp_5,temp_1}
```

Note:

- If you ran the compiler with `--graphs`, the CFG should be displayed in a graphical window. The names of basic blocks in the text output correspond to the leading label in the blocks shown in the graphical window.
- Only temporaries are shown. Block `lbl_div_by_zero_0_main` uses physical registers but no temporary, hence `gen` and `kill` sets are empty.

EXERCISE #6 ► Liveness Analysis, fixpoint. (Only test!)

We implemented for you the fixpoint iteration as a method (`run_dataflow_analysis`) in `SmartAllocator.py` "while it is not finished, store the old values, do an iteration, decide if its finished". The `run_dataflow_analysis` method makes calls to `dataflow_one_step` instruction methods. The result (liveness information is stored in member sets of the `._liveness` attribute of the `SmartAllocator` class).

What you have to do in this exercice is to check that the results that are obtained with with analysis are correct at least for the examples of the `TP05/tests/provided/dataflow/` directory.

To do so, the following lines should help you (again, using `--debug`) in the same file:

```
self._liveness.run() #compute liveness sets
if self._debug:
    self._liveness.print_map_in_out()
```

As an example, here is the expected output for `TP05/tests/provided/dataflow/df02.c`:

```
Dataflow Analysis
finished in 2 iterations
In: {lbl_div_by_zero_0_main: {},
lbl_end_relational_2_main: {temp_7,temp_2},
lbl_main_6_main: {temp_2},
lbl_end_relational_1_main: {temp_1,temp_5},
lbl_main_5_main: {temp_1}, lbl_main_4_main: {}}

Out: {lbl_div_by_zero_0_main: {}, lbl_end_relational_2_main: {},
 lbl_main_6_main: {temp_7,temp_2},
lbl_end_relational_1_main: {temp_7,temp_2},
lbl_main_5_main: {temp_1,temp_5},
lbl_main_4_main: {temp_1,temp_5}}
```

EXERCISE #7 ► Interference graph: Implement interfere

The interference graph contains an edge (x, y) if temporaries x and y are in conflict. It is built by `SmartAllocator.build_interference_graph`, that calls the `interfere` method.

We recall that two temporaries x, y are in conflict if they are simultaneously alive after a given **instruction**[1], which means:

- There exists an instruction i and $x, y \in LV_{out}(i)$
- OR There exist an instruction i such that $x \in LV_{out}(i)$ and y is defined in the instruction
- OR the converse.

To understand why the last two cases are needed, consider the following list of instructions:

```
// Can x and y be mapped to the same place? Obviously not.
y=2
x=1 // dead assignment
println_int(y)
x=666 // x not live before this
```

The live ranges of x and y are disjoint, however x is in conflict with y since we generate the code for `x=1` while y is alive[2].

From the result of the previous exercise, the code in `LivenessDataFlow` builds a `self._liveness._liveout` field of type `Dict[Instruction, Set[Operand]]` that gives the set of temporaries that are live after a given **instruction**[3]. Use this data to construct the interference graph (the job is done by function `build_interference_graph`) of your program. We give you a undirected graph API (`Lib/Graphes.py`) for that. Use the `print_dot` (it should be called with the `-graph` option) method and relevant tests to validate your code.

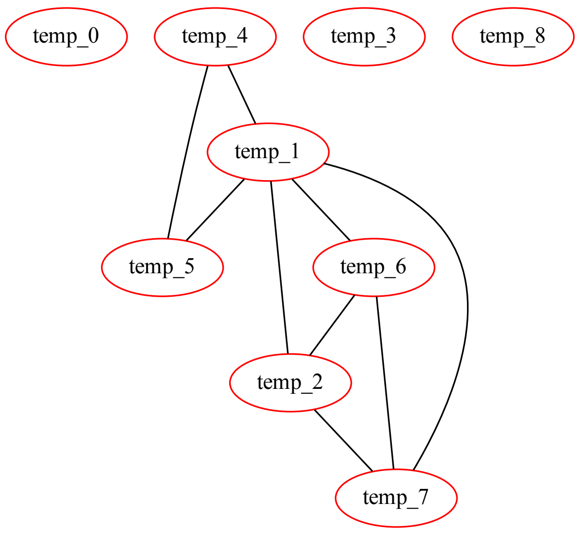
[1] yes, an instruction, not a block

[2] Another solution consists in eliminating dead code before generating the interference graph.

[3] The dataflow analysis is indeed performed at the block level, and a postanalysis is run afterwards to get the information at the instruction level.

In this exercise, we care about correctness more than complexity. It is OK to write an $O(n^3)$ algorithm (for each t_1, for each t_2, for each control point c, check whether t_1 and t_2 have a conflict).

As an example, here is the conflict graph that should be obtained for `df02.c` (command line as usual):



5.4 “Smart’ register allocation and code production

We will implement the following algorithm for register allocation:

- Color the interference graph with an infinite number of colors, using the first ones as much as possible.
- The first `len(GP_REGS)` colors will be mapped to registers.
- All the other variables will be allocated on the stack. For each color, we use a memory location according to their coloring number.

Then the 3 address code modification:

- For non-spilled variable: replace the temporary with its associated color/register, as we did for the naive allocator.
- For spilled variables: add `ld` / `sd` statements as needed and replace the temporary with one of `s1`, `s2`, `s3` as we did for the “all in mem” allocator.

Some help:

- GP_REGS is an array of registers available for the register allocator.
- An element of type Register can be obtained from a given register color with the helper function `GP_REGS[coloringreg[xxx]]`, where `coloringreg` is graph coloring returned by the `.color()` function, and for offsets you have the method `self._function_code.new_offset()` that returns a fresh one (all in `Operands.py`).
- The easiest way to build `alloc_dict` is probably to iterate over all the temporaries of the program (available in `self._function_code._pool._all_temps`), and for each temporary check the corresponding color to associate it to the right register or memory location in `alloc_dict`.

EXERCISE #8 ► **Smart Register Allocation: implement!**
In this exercice, you have first to complete method `SmartAllocation.smart_alloc()` to perform an allocation based on a graph coloring. The purpose of this method is to allocate a physical register or a memory location for each temporary in the program. Next, you will have to complete the function `replace_smart()` that replaces the temporary operands of a given instruction according to the allocation computed by `smart_alloc()`.

Use the algorithm and the coloration method of the `LibGraphes` class to allocate registers (or a memory location) in `smart_alloc()`. Comments will help you design this (non trivial) function. The allocation is followed by statement rewriting, like in previous lab. You need to implement it in `SmartAllocation.py` (`replace_smart`): it is very similar to the previous lab's version, but you have to deal with both memory locations and registers in the same function.

Validate your allocation on tiny well chosen test files (especially tests that augment the register pressure) and all the benchmarks of the previous lab. We adapted the previous script for that.

Each color+shape pair indicates a different location. Temp numbering and coloring may be different in your output.

EXERCISE #9 ▸ **Massive tests**
Debug, ... and test on all test files you have (test cases in `TP05/tests/students`, please). The testsuite must not open any PDF file (this is normally OK since it launches the compiler without the `--debug` option).

5.5 Final delivery

Instructions will be given on Chamilo. Lab 4 and Lab 5 will be delivered at the same time, and the two commands `make test-lab4` and `make test-codegen` should perform correctly.

Appendix A
RISCV Assembly Documentation (ISA), rv64g

About

- RISCV is an open instruction set initially developed by Berkeley University, used among others by Western Digital, Alibaba and Nvidia.
- We are using the rv64g instruction set: **R**isc-**V**, 64 bits, **G**eneral purpose (base instruction set, and extensions for floating point, atomic and multiplications), without compressed instructions. In practice, we will use only 32 bits instructions (and very few of floating point instructions).
- Document: Laure Gonnord and Matthieu Moy, for CAP and MIF08.

This is a simplified version of the machine, which is (hopefully) conform to the chosen simulator.

A.1 Installing the simulator and getting started

To get the RISCV assembler and simulator, follow instructions of the first lab (git pull on the course lab repository).

A.2 The RISCV architecture

Here is an example of RISCV assembly code snippet (a proper `main` function would be needed to execute it, cf. course and lab):

```
  addi a0, zero, 17  # initialisation of  a register to 17
loop:
  addi a0, a0, -1    # subtraction of an immediate
  j  loop            # equivalent to jump xx
```

The rest of the documentation is adapted from `https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md` and `https://github.com/jameslzhu/riscv-card/blob/master/riscv-card.pdf`

A.3 RISC-V Assembly Programmer's Manual - "simplified"

A.3.1 Copyright and License Information - Documents

- Official Specifications webpage: https://riscv.org/specifications/
- Latest Specifications draft repository: https://github.com/riscv/riscv-isa-manual

This document has been modified by Laure Gonnord & Matthieu Moy, in 2019.

A.3.2 Registers

Registers are the most important part of any processor. RISC-V defines various types, depending on which extensions are included: The general registers (with the program counter), control registers, floating point registers (F extension), and vector registers (V extension). We won't use control nor F or V registers.

General registers

The RV32I base integer ISA includes 32 registers, named `x0` to `x31`. The program counter `PC` is separate from these registers, in contrast to other processors such as the ARM-32. The first register, `x0`, has a special function: Reading it always returns 0 and writes to it are ignored.

In practice, the programmer doesn't use this notation for the registers. Though `x1` to `x31` are all equally general-use registers as far as the processor is concerned, by convention certain registers are used for special tasks. In assembler, they are given standardized names as part of the RISC-V **application binary interface** (ABI). This is what you will usually see in code listings. If you really want to see the numeric register names, the `-M` argument to objdump will provide them.

Register	ABI	Use by convention	Preserved?
x0	zero	hardwired to 0, ignores writes	*n/a*
x1	ra	return address for jumps	no
x2	sp	stack pointer	yes
x3	gp	global pointer	*n/a*
x4	tp	thread pointer	*n/a*
x5	t0	temporary register 0	no
x6	t1	temporary register 1	no
x7	t2	temporary register 2	no
x8	s0 *or* fp	saved register 0 *or* frame pointer	yes
x9	s1	saved register 1	yes
x10	a0	return value *or* function argument 0	no
x11	a1	return value *or* function argument 1	no
x12	a2	function argument 2	no
x13	a3	function argument 3	no
x14	a4	function argument 4	no
x15	a5	function argument 5	no
x16	a6	function argument 6	no
x17	a7	function argument 7	no
x18	s2	saved register 2	yes
x19	s3	saved register 3	yes
x20	s4	saved register 4	yes
x21	s5	saved register 5	yes
x22	s6	saved register 6	yes
x23	s7	saved register 6	yes
x24	s8	saved register 8	yes
x25	s9	saved register 9	yes
x26	s10	saved register 10	yes
x27	s11	saved register 11	yes
x28	t3	temporary register 3	no
x29	t4	temporary register 4	no
x30	t5	temporary register 5	no
x31	t6	temporary register 6	no
pc	*(none)*	program counter	*n/a*

Registers of the RV32I. Based on RISC-V documentation and Patterson and Waterman "The RISC-V Reader" (2017)

As a general rule, the **saved registers** `s0` to `s11` are preserved across function calls, while the **argument registers** `a0` to `a7` and the **temporary registers** `t0` to `t6` are not. The use of the various specialized registers such as `sp` by convention will be discussed later in more detail.

A.3.3 Instructions

Arithmetic

add, addi, sub, classically.

```
addi a0, zero, 42
```

initialises a0 to 42.

Labels

Text labels are used as branch, unconditional jump targets and symbol offsets. Text labels are added to the symbol table of the compiled module.

```
loop:
        j loop
```

Jumps and branches target is encoded with a relative offset. It is relative to the beginning of the current instruction. For example, the self-loop above corresponds to an offset of 0.

Branching

Test and jump, within the same instruction:

```
beq a0, a1, end
```

tests whether a0=a1, and jumps to ‘end’ if its the case.

Absolute addressing

The following example shows how to load an absolute address:

```
.section .text
.globl _start
_start:
        lui a0,       %hi(msg)       # load msg(hi)
        addi a0, a0,  %lo(msg)       # load msg(lo)
        jal ra, puts
2:      j 2b

.section .rodata
msg:
        .string "Hello World\n"
```

which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
   0:   000005b7            lui a1,0x0
            0: R_RISCV_HI20 msg
   4:   00858593            addi    a1,a1,8 # 8 <.L21>
            4: R_RISCV_LO12_I   msg
```

Relative addressing

The following example shows how to load a PC-relative address:

```
.section .text
.globl _start
_start:
1:      auipc a0,     %pcrel_hi(msg) # load msg(hi)
        addi  a0, a0, %pcrel_lo(1b)  # load msg(lo)
        jal ra, puts
```

```
2:      j 2b

.section .rodata
msg:
        .string "Hello World\n"
```

which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
   0:   00000597           auipc   a1,0x0
            0: R_RISCV_PCREL_HI20   msg
   4:   00858593           addi    a1,a1,8 # 8 <.L21>
            4: R_RISCV_PCREL_LO12_I .L11
```

Load Immediate

The following example shows the `li` pseudo instruction which is used to load immediate values:

```
        li a0, 0x76543210
```

which generates the following assembler output as seen by objdump (generated code will be different depending on the constant):

```
   0:   76543537               lui     a0,0x76543
   4:   2105051b               addiw   a0,a0,528
```

Load Address

The following example shows the `la` pseudo instruction which is used to load symbol addresses:

```
.section .text
.globl _start
_start:

        la a0, msg

.section .rodata
msg:
        .string "Hello World\n"
```

A.3.4 Assembler directives (subset)

Both the RISC-V-specific and GNU .-prefixed options.

The following table lists assembler directives:

Directive	Arguments	Description
.align	integer	align to power of 2 (alias for .p2align)
.file	“filename”	emit filename FILE LOCAL symbol table
.globl	symbol_name	emit symbol_name to symbol table (scope GLOBAL)
.local	symbol_name	emit symbol_name to symbol table (scope LOCAL)
.section	[{.text,.data,.rodata,.bss}]	emit section (if not present, default .text) and make current
.size	symbol, symbol	accepted for source compatibility
.text		emit .text section (if not present) and make current
.data		emit .data section (if not present) and make current
.rodata		emit .rodata section (if not present) and make current

Directive	Arguments	Description
.string	“string”	emit string
.equ	name, value	constant definition
.word	expression [, expression]*	32-bit comma separated words
.balign	b,[pad_val=0]	byte align
.zero	integer	zero bytes

A.3.5 Assembler Relocation Functions

The following table lists assembler relocation expansions:

Assembler Notation	Description	Instruction / Macro
%hi(symbol)	Absolute (HI20)	lui
%lo(symbol)	Absolute (LO12)	load, store, add
%pcrel_hi(symbol)	PC-relative (HI20)	auipc
%pcrel_lo(label)	PC-relative (LO12)	load, store, add

A.3.6 Instruction encoding

Credit This is a subset of the RISC-V greencard, by James Izhu, licence CC by SA, `https://github.com/jameslzhu/riscv-card`

Core Instruction Formats

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

“imm[x:y]” means “bits x to y from binary representation of imm”. “imm[y|x]” means “bits y, then x of imm”.

RV32I Base Integer Instructions (subset)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
`add`	ADD	R	`0110011`	`0x0`	`0x00`	`rd = rs1 + rs2`	
`sub`	SUB	R	`0110011`	`0x0`	`0x20`	`rd = rs1 - rs2`	
`xor`	XOR	R	`0110011`	`0x4`	`0x00`	`rd = rs1 ^ rs2`	
`or`	OR	R	`0110011`	`0x6`	`0x00`	`rd = rs1 | rs2`	
`and`	AND	R	`0110011`	`0x7`	`0x00`	`rd = rs1 & rs2`	
`slt`	Set Less Than	R	`0110011`	`0x2`	`0x00`	`rd = (rs1 < rs2)?1:0`	
`sltu`	Set Less Than (U)	R	`0110011`	`0x3`	`0x00`	`rd = (rs1 < rs2)?1:0`	zero-extends
`addi`	ADD Immediate	I	`0010011`	`0x0`		`rd = rs1 + imm`	
`xori`	XOR Immediate	I	`0010011`	`0x4`		`rd = rs1 ^ imm`	
`ori`	OR Immediate	I	`0010011`	`0x6`		`rd = rs1 | imm`	
`andi`	AND Immediate	I	`0010011`	`0x7`		`rd = rs1 & imm`	
`lb`	Load Byte	I	`0000011`	`0x0`		`rd = M[rs1+imm][0:7]`	
`lw`	Load Word	I	`0000011`	`0x2`		`rd = M[rs1+imm][0:31]`	
`lbu`	Load Byte (U)	I	`0000011`	`0x4`		`rd = M[rs1+imm][0:7]`	zero-extends
`sb`	Store Byte	S	`0100011`	`0x0`		`M[rs1+imm][0:7] = rs2[0:7]`	
`sw`	Store Word	S	`0100011`	`0x2`		`M[rs1+imm][0:31] = rs2[0:31]`	
`beq`	Branch ==	B	`1100011`	`0x0`		`if(rs1 == rs2) PC += imm`	
`bne`	Branch !=	B	`1100011`	`0x1`		`if(rs1 != rs2) PC += imm`	
`blt`	Branch <	B	`1100011`	`0x4`		`if(rs1 <  rs2) PC += imm`	
`bge`	Branch ≥	B	`1100011`	`0x5`		`if(rs1 >= rs2) PC += imm`	
`bltu`	Branch < (U)	B	`1100011`	`0x6`		`if(rs1 <  rs2) PC += imm`	zero-extends
`bgeu`	Branch ≥ (U)	B	`1100011`	`0x7`		`if(rs1 >= rs2) PC += imm`	zero-extends
`jal`	Jump And Link	J	`1101111`			`rd = PC+4; PC += imm`	
`jalr`	Jump And Link Reg	I	`1100111`	`0x0`		`rd = PC+4; PC = rs1 + imm`	
`lui`	Load Upper Imm	U	`0110111`			`rd = imm << 12`	
`auipc`	Add Upper Imm to PC	U	`0010111`			`rd = PC + (imm << 12)`	

Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning

`la rd, symbol`	`auipc rd, symbol[31:12]`
	`addi rd, rd, symbol[11:0]`
		Load address

`{lb|lh|lw|ld} rd, symbol`	`auipc rd, symbol[31:12]`
	`{lb|lh|lw|ld} rd, symbol[11:0](rd)`
		Load global

`{sb|sh|sw|sd} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`s{b|h|w|d} rd, symbol[11:0](rt)`
		Store global

`{flw|fld} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fl{w|d} rd, symbol[11:0](rt)`
		Floating-point load global

`{fsw|fsd} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fs{w|d} rd, symbol[11:0](rt)`
		Floating-point store global

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine

`call offset`	`auipc x1, offset[31:12]`
	`jalr x1, x1, offset[11:0]`
		Call far-away subroutine

`tail offset`	`auipc x6, offset[31:12]`
	`jalr x0, x6, offset[11:0]`
		Tail call far-away subroutine

`fence`	`fence iorw, iorw`	Fence on all memory and I/O

RV32M Multiply Extension (basic instructions)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
`mul`	MUL	R	`0110011`	`0x0`	`0x01`	`rd = (rs1 * rs2)[31:0]`
`div`	DIV	R	`0110011`	`0x4`	`0x01`	`rd = rs1 / rs2`
`rem`	Remainder	R	`0110011`	`0x6`	`0x01`	`rd = rs1 % rs2`

Appendix B
A bit of PYTHON 3 & ANTLR4

B.1 PYTHON

`https://docs.python.org/fr/3.5/tutorial/`
`htpp://perso.limsi.fr/pointal/_media/python:cours:mementopython3.pdf`

Coding Style :

`https://www.python.org/dev/peps/pep-0008/`

We strongly recommand to use:

`flake8 filename.py`

on each file.

Exceptions in PYTHON Recall that in PYTHON errors can be declared, thrown and caught as depicts Figure B.1

```
# declare !
class MyError(Exception):
    pass

# catch!
        try:
            ...
        except MyError:
            ...

# launch !
        raise MyError("Error␣Message")
```

Figure B.1: Exceptions in PYTHON

B.2 ANTLR4

A nice book:

`https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference`

A nice tutorial:

`https://tomassetti.me/antlr-mega-tutorial/`