Compilation (MIF08)

Cahier de TD/TP, janvier 2020

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 infrastrcture, 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://compil-lyon.gitlabpages.inria.fr/compil-lyon/`

and for MIF08 :

`http://laure.gonnord.org/pro/teaching/compilM1.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 and TBA

Lab 1
Warm-up : Python and the target machine : RISCV

Objective

- Start with Python.
- Be familiar with the RISCV instruction set.[1]
- Understand how it executes on the RISCV processor with the help of a simulator.
- Write simple programs, assemble, execute.

Todo in this lab:
- Play and learn Python!
- Play and learn the RISCV ISA.
- Finish at home, nothing will be evaluated in this lab.

1.1 Quick intro to Python - 1h max

This part is strongly inspired by the Project 1 of ENSL (L3).

Please use a correct text editor ! We don't really care if it is SublimeText, Emacs, Atom or Vim, but please use a text editor made for programming.

`https://www.python.org/` Official tutorial: `https://docs.python.org/3/tutorial/` An amazing interactive one `http://www.learnpython.org/en/Welcome`

1.1.1 Inside the interpreter

And now, let's get to the heart of the matter.

EXERCISE #1 ► **Launch!**
Launch the Python interpreter (`python3`, in the terminal). Which version is it ? Use a version of Python not older than 3.5. Quit the interpreter with CTRL-D or quit().

EXERCISE #2 ► **Strings**
Try the following code:

```
x = 'na'
'Ba' + 2 * x
```

Then write "j'aime les bons bonbons" with the same technique.

Lists

EXERCISE #3 ► **Lists**
Create a list `li` of integers containing various éléments. Replace one of the elements with a new value. At last, use + or += to add elements at the end of the list.

EXERCISE #4 ► **Sorts**
Sort a list using function `sorted`. What is the complexity in the worst case? In the best case? Use function `len()`; same questions.

[1] todo lablbalbla

Print

EXERCISE #5 ► **Formatting**
Give 3 different ways of building the following character string:
`"2.21 Gigawatts !! 2.21 Gigawatts !! My godness !"` using one variable `x = 2.21`, and another variable that uses `str()`, then the operator `%`, then the method `.format()`.

1.1.2 Tiny programs

Now, write your programs in `.py` files (with an editor). If you get encoding issue, add this at the beginning, but it shouldn't be needed with Python 3:

```
# -*- coding: utf-8 -*-
```

EXERCISE #6 ► **Hello**
Edit a file named `hello.py` with the following content:

```
print("Hello World")
```

Save, execute with: `python3 hello.py`.

EXERCISE #7 ► **If then else**
Write a program that initializes an int value to a number given by the user (use `input()`) and prints a different message according to its parity (odd/even).

EXERCISE #8 ► **While**
Write a program that declares two integer values `a` and `b`, then computes and prints their pgcd.

EXERCISE #9 ► **Imperative For**
Using the construction `for i in ...`, write a program that sums all even *i* from 2 to 42 (inclusive).

EXERCISE #10 ► **For expression / Lists**

- Write a program that declares and initialises a list, and computes the sum of all its elements.
- Write a 1-line code that, from a list `l`, returns a list whose elements are the squares of the elements in `l`.
- Write a 1-line code that, from a list `l`, returns a list containing the even elements of `l.l`.

EXERCISE #11 ► **Dicts**

1. What are the types of `{}`, `{'a'}`, `{'a', 'b'}` and `{'a': 'b'}`?
2. What is the following code doing (where `t` is a dictionary):

   ```
   while key in t:
          key = t[key]
   print(key)
   ```

   What is the problem?
3. Write a code doing the same operation but without the same drawback (*i.e.*: if needed, it doesn't print anything)

EXERCISE #12 ► **Functions**

1. Declare a function `fact` that computes the factorial of a number.
2. What does `help(fact)` display? If it is not done, document your function (add a docstring).

1.2 The RISCV processor, instruction set, simulator

EXERCISE #13 ► **Lab preparation**
Clone the github repository for this year's labs:

```
git clone https://github.com/lauregonnord/mif08-labs19.git
```

Then, follow the instructions to compile `riscv-xxx-gcc` and `spike` on your machine (see `INSTALL.md` file). On the Nautibus machines, all installations have already been done for you. However, you still have to add the following lines to your `.bashrc` :

```
RISCV=/home/tpetu/Enseignants/matthieu.moy/mif08/riscv
export PATH="$RISCV"/bin:"$PATH"
export LD_LIBRARY_PATH="$RISCV"/libexec/gcc/riscv64-unknown-elf/9.2.0:"$LD_LIBRARY_PATH"
```

EXERCISE #14 ► RISCV **C-compiler and simulator, first test**
In the directory `TP01/code/` :

- 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.
- 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 this sequence of labs is to design **our own (subset of) C compiler for** RISCV.

EXERCISE #15 ► **Documents**
Some documentation can be found in the RISCV ISA on the course webpage and in Appendix A.

`https://compil-lyon.gitlabpages.inria.fr/compil-lyon/`

The assembly language for this year is RISCV. We already played a bit with it in the exercise session.

1.2.1 Assembling, disassembling

EXERCISE #16 ► **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.

1.2.2 RISCV **Simulator**

EXERCISE #17 ► **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 `print_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 MIF08!
   42
   a
   ```
   The first line comes from the simulator itself, the next two come from the `print_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
   000000000001015c 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 1015c. Now, run spike in debug mode (`-d`) and execute code up to this address (`until pc 0 1015c`, i.e. "Until the program counter of core 0 reaches 1015c"). Press Return to move to the next instruction and `q` to quit:
   ```
   $ spike -d pk ./test_print
   : until pc 0 1015c
   bbl loader
   :
   core   0: 0x000000000001015c (0xff010113) addi    sp, sp, -16
   :
   core   0: 0x0000000000010160 (0x00113423) sd      ra, 8(sp)
   :
   core   0: 0x0000000000010164 (0x0001d7b7) lui     a5, 0x1d
   :
   core   0: 0x0000000000010168 (0x02078513) addi    a0, a5, 32
   : 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 #18 ► **Algo in** RISCV **assembly**
Write (in `minmax.s`) 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 32 bits of memory to store ints, i.e., use `.word` directive and `lw` and `sw` instructions.

EXERCISE #19 ► **(Advanced) Algo in** RISCV **assembly**

Write and execute the following programs in assembly:

- Count the number of non-nul bits of a given integer, print the result.
- Draw squares and triangles of stars (character '*') of size n, n being stored somewhere in memory.
  Examples:
  n=3 square:
  ```
  ***
  ***
  ***
  ```
  n=3 triangle:
  ```
    *
   * *
  * * *
  ```

1.2.3 Finished?

If you're done with the lab, do the python tutorial at the following address:

`https://docs.python.org/fr/3.5/tutorial/`

Lab 2
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. **This will be evaluated next lab!**
- Understand and extend an arithmetic evaluator (with semantic actions).
- Understand our future test infrastructure.

EXERCISE #1 ► **Lab preparation**
In the lab's git repository(`mif08-labs19`)[1]:

```
git commit -a -m "my changes to LAB1" #push is not allowed
git pull
```

will provide you all the necessary files for this lab in `TP02`. You also have to install ANTLR4. For tests, we will use `pytest`, you may have to install it:

```
pip3 install pytest --user
```

2.1 User install for ANTLR4 and ANTLR4 Python runtime

2.1.1 User installation

EXERCISE #2 ► **Install**
To be able to use ANTLR4 for the next labs, download it and install the python runtime:

```
mkdir ~/lib
cd ~/lib
wget http://www.antlr.org/download/antlr-4.7.1-complete.jar
pip3 install antlr4-python3-runtime --user
```

Then add to your `~/.bashrc`:

```
export CLASSPATH=".:$HOME/lib/antlr-4.7.1-complete.jar:$CLASSPATH"
export ANTLR4="java -jar $HOME/lib/antlr-4.7.1-complete.jar"
alias antlr4="java -jar $HOME/lib/antlr-4.7.1-complete.jar"
alias grun="java org.antlr.v4.gui.TestRig"
```

Then source your `.bashrc`:

```
source ~/.bashrc
```

Tests will be done in Section 2.2.2.

[1] if you don't have it already, get it from `https://github.com/lauregonnord/mif08-labs19.git`

2.2 Simple examples with ANTLR4

2.2.1 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:

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

If you did not define the alias or if you installed the `.jar` file to another location, you may also use:

```
java -jar /path/to/antlr-4.7-complete.jar -Dlanguage=Python3  filename.g4
```

(note: `antlr4`, not `antlr` which may also exists but is not the one we want)

2.2.2 Up to you!

EXERCISE #3 ► **Demo files**
Work your way through the three examples (open them in your favorite editor!) in the directory `demo_files`:

ex1 with ANTLR4 + **Java:** A very simple lexical analysis[2] for simple arithmetic expressions of the form `x+3`. To compile, run:

```
antlr4 Example1.g4
javac *.java
```

This generates Java code and then compiles them. You can finally execute using the Java runtime with:

```
grun Example1 tokens -tokens
```

To signal the program you have finished entering the input, use **Control-D** (you may need to press it twice).
Examples of run: [`^D` means that I pressed Control-D]. What I typed is in boldface.

```
1+1
^D^D
[@0,0:0='1',<DIGIT>,1:0]
[@1,1:1='+',<OP>,1:1]
[@2,2:2='1',<DIGIT>,1:2]
[@3,4:3='<EOF>',<EOF>,2:0]
)+
^D^D
line 1:0 token recognition error at: ')'
[@0,1:1='+',<OP>,1:1]
[@1,3:2='<EOF>',<EOF>,2:0]
%
```

Questions:
- Read and understand the code.
- Allow for parentheses to appear in the expressions.
- What is an example of a recognized expression that looks odd? To fix this problem we need a syntactic analyzer (see later).

[2]Lexer Grammar in ANTLR4 jargon

ex1b: same with a PYTHON file driver:

```
antlr4 -Dlanguage=Python3 Example1b.g4
python3 main.py
```

test the same expressions. Observe the PYTHON file.

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

ex2: 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] http://stackoverflow.com/questions/11118539/antlr-combination-of-tokens

EXERCISE #4 ► **Well-founded parenthesis**
Write a grammar and files to make an analyser that:

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

Thus your analyser will accept "(hop)" or "[()](tagada)" but reject "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.*

This is the kind of exercise that will be graded at the beginning of Lab 3.

EXERCISE #5 ► **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.

2.3 Grammar Attributes (actions)

Until now, our analyzers are passive oracles, ie 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 will be reduced. This piece of code is called *semantic action* and computes attributes of non-terminals.

This exercice is a demo - no grade will be given We consider a simple grammar of non empty lists of arithmetic expressions:

$$S \to Z+$$
$$Z \to E;$$
$$E \to E + E$$
$$E \to E * E$$
$$E \to F$$
$$F \to int$$
$$F \to (E)$$

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

EXERCISE #6 ► **Test the provided code (`ariteval/` directory)**
To test the provided code, just type:

1. Type

   ```
   make ; python3 arit1.py testfiles/test01.txt
   ```

   This should print:

   ```
   1+2 = 3
   ```

   on the standard output.

2. Type:

   ```
   make tests
   ```

   This should print:

   ```
   test_ariteval.py::TestEVAL::test_expect[./testfiles/test01.txt] PASSED   [ 50%]
   test_ariteval.py::TestEVAL::test_expect[./testfiles/test02.txt] PASSED   [100%]
   ```

EXERCISE #7 ► **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. For the moment it only tests files of the form `testfiles/test*.txt`. Just type:

```
make tests
```

and your code will be tested on these files.

To test on other tests files, you may have to open the `test_ariteval.py` and change some paths.

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 `testfiles/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`.

EXERCISE #8 ► **Write tests**
Write tests.

EXERCISE #9 ► **Optional**
Implement binary and unary minus. Test.

Lab 3
Interpreters and Types

Objective

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

EXERCISE #1 ► **Lab preparation**
In the `mif08-labs19` directory:
```
git pull
```
will provide you all the necessary files for this lab in `TP03`. ANTLR4 and `pytest` should be installed and working like in Lab 2, if not :
```
pip3 install --user pytest
```
The testsuite also uses `pytest-cov`, to be installed with[1]:
```
pip3 install --user pytest-cov
pip3 install --user --upgrade coverage
```

3.1 Demo: Implicit tree walking using Visitors

3.1.1 Interpret (evaluate) arithmetic expressions with visitors

In the previous 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.

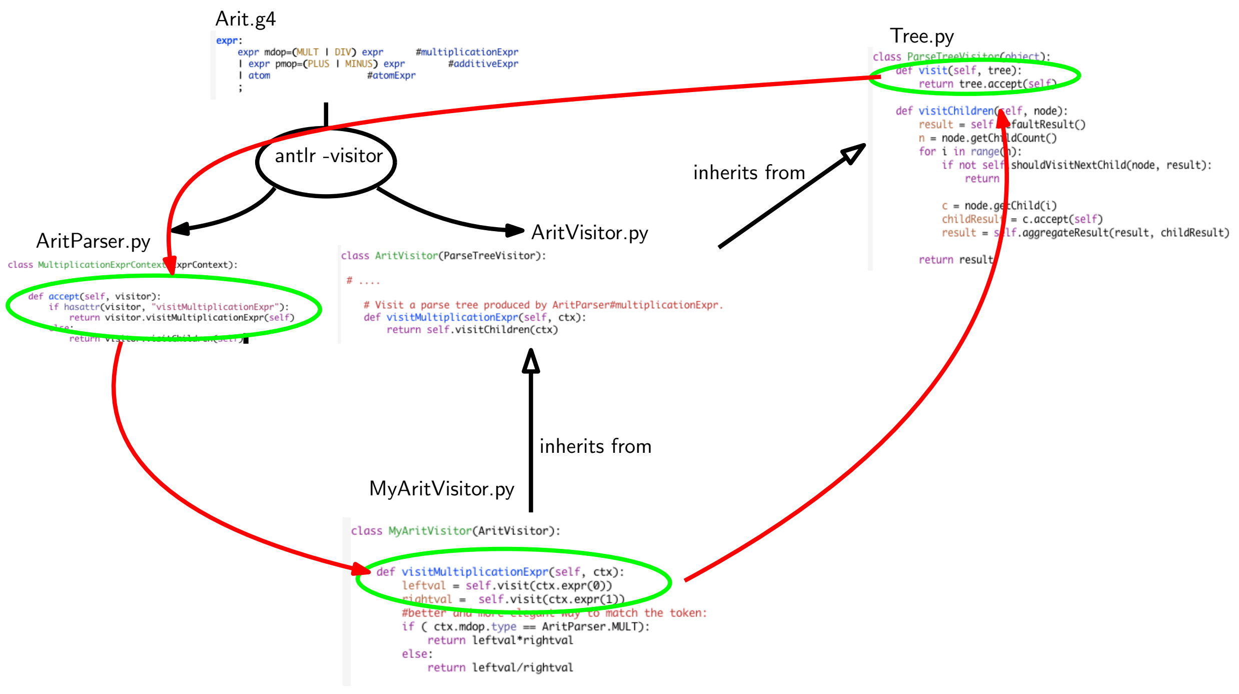
EXERCISE #2 ► **Demo: arithmetic expression interpreter (`arith-visitor/`)**
Observe and play with the `Arit.g4` grammar and its PYTHON Visitor :
```
$ make ; make run < myexample
```
Note that unlike the "attribute grammar" version that we used previously, the `.g4` file does not contain Python code 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 interpret, and use `print` instructions to observe how the visitor actually work (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`



Arit.g4
```
expr:
    expr mdop=(MULT | DIV) expr      #multiplicationExpr
    | expr pmop=(PLUS | MINUS) expr        #additiveExpr
    | atom                       #atomExpr
    ;
```

Tree.py
```
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
```

AritParser.py
```
class MultiplicationExprContext(ExprContext):

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

AritVisitor.py
```
class AritVisitor(ParseTreeVisitor):

 # ....

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

MyAritVisitor.py
```
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 ParseTree visitor 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.

A last remark: when a ANTLR4 rule contains an operator alternative such as:

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

you can use the following code to match the operator:

```
if ( ctx.pmop.type == AritParser.PLUS):
    ...
```

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

EXERCISE #3 ► **Be prepared!**

In the directory `MiniC-type-interpret/`, you will find:

- The MiniC grammar (`MiniC.g4`).
- A `Main.py` that parses the command line, does the lexical analysis and syntax analysis of the input file, then launches the Typing visitor, and if the file is well typed, launches the Interpreter visitor.
- One complete visitor: `MiniCTypingVisitor.py`, and one to be completed: `MiniCInterpretVisitor.py`.
- Some test cases, and a test infrastructure.

```
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 #funcDecl;

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 condition_block (ELSE IF condition_block)* (ELSE stat_block)? #ifStat;

condition_block: OPAR expr CPAR stat_block #condBlock;

stat_block
    : OBRACE block CBRACE
    | stat
    ;

while_stat: WHILE OPAR expr CPAR stat_block #whileStat;


print_stat
```

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

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

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 `MiniCTypingVisitor.py`, and predict its behavior on the following MiniC file:

```
int x;
x="blablabla";
```
Then, test with:
```
make run TESTFILE=ex-types/bad_type00.c
```
Observe the behavior of the visitor on all test files in `ex-types/`. How do we handle:

- Multiplicative expressions with int and string operands ?
- Assignements to a variable which is not of the same type as the expression ?
- The variable type declarations ?

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

On bad 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 unit test. Any error (typing or runtime) must raise the exit code 1. Now, type:
```
make tests
```
and observe (Typing tests are those concerning files in `ex-types/`). If you get an error about the `--cov` argument, you didn't properly install pytest-cov. To allow compiling your MiniC programs with a regular C compiler, a `printlib.h` file is provided, and should be `#included` in all your MiniC test cases.

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)
- And obviously, 0 if the program is typechecked and executed without error.

3.3 An interpreter for the MiniC-language

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.

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

e_1+e_2	`let v1 = e1.visit() and v2 = e2.visit() in`
	`    return v1+V2`

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

Figure 3.3: Interpretation (Evaluation) of expressions

EXERCISE #6 ► **Interpreter rules (on paper)**

First fill the empty cells in Figure 3.4, then ask your teaching assistant to correct them.

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

print_int(e)	`let v = e.visit() in`
	`   print(v) #python print`

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

`if` *b* `then` *S*1 `else` *S*2	
`while` *b* `do` *S* `done`	

Figure 3.4: Interpretation for Statements

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!`) For now, you can reason in terms of "well-typed programs".

Type:
```
make run TESTFILE='ex/testxx.c'
```
and the interpreter will be run on `ex/testxx.c` (or on `ex/test00.c` if you do not specify variable `TESTFILE`). **On the particular example `ex/test00.c` observe how integer values, strings, boolean, floats values are printed.**

You still have to implement (in `MiniCInterpretVisitor.py`):

1. The modulo version of Multiplicative expressions.
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 special value None for all variable declarations.** Refer to the three test files `ex/bad_defxx.c` for the expected error messages.
3. Statements: assignments, conditional blocks, tests, loops.

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)
- And obviously, 0 if the program is typechecked and executed without error.

EXERCISE #8 ▶ **Unit tests**

Test with `make tests` **and appropriate test-suite**. If you get an error about the `--cov` argument, you didn't properly install pytest-cov. 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 never happen if your 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. To properly test the `ex/bad_def*` files, you will have to edit the python test script `test_interpreter.py`.

Test Infrastructure Tests work mostly as in the previous lab, with `// EXPECTED` and `// EXITCODE n` pragmas in the tests (be careful, it's now `//` for the comments, not `#`).

For instance, if you fail `test00.c` because you printed 42 instead of `99.00`, you will get this error:

```
______________ TestCodeGen.test_expect[ex/test00.c] ______________

self = <test_interpreter.TestCodeGen object at 0x7f0e0aa369b0>
filename = 'ex/test00.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 '99.00\n1\n' == '42\n1\n'
E             - 99.00
E             + 42
E               1

test_interpreter.py:59: AssertionError
```

And if you did not print anything at all when `99.00` was expected, the last lines would be this instead:

```
        if expect:
>           assert(expect == eval)
E           assert '99.00\n1\n' == '1\n'
E             - 99.00
E               1

test_interpreter.py:59: AssertionError
```

EXERCISE #9 ▶ **Archive**

The interpreter (all exercises in Section 3.3) is due on TOMUSS on Friday, 17/01/2020 right after the demo (5pm). Type `make tar` to obtain the archive to send (change your name in the Makefile before!). Your archive must also contain tests (Tests should be in `ex/` and `ex-types/`.) and a `README.md` with your name, the functionality of the code, how to use it, your design choices, and known bugs, tests (There is an example in `TP02/ariteval`.

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.

You may have to install some additional Python libs:

```
pip3 install networkx graphviz --user
```

And on your personal machines:

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

4.1 Preliminaries

This section must be **carefully** read.

Important remark From now on, we add some restrictions to the MiniC language:

- Values (variables, argument of `println_int`) are of type (signed) `int` or `bool` only (no float, no string, no char). Thus all values can be stored in regular registers or in one cell (64 bits) in memory. You can let your program crash 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 `APIRiscV.py` we provide you with utility functions to encode 3-address RISCV instructions. Instruction classes are in `Instruction3A.py` and `Operands.py`. An Instruction is either a Comment, a Label, or a `Instru3A`; it has arguments which can be immediate numbers (of type `Immediate`), temporaries (of type `Temporary`), regular registers (`Register` [1]), offsets in memory (`Offset`).

- A RISCV program contains a list of instructions, and also a temporary pool (temporary variables).

- In Section 4.2, you will use an instance of the `RiscVFunction` class in order to construct a list of such instructions via calls to `addInstructionXXX` methods. A call to the `printCode` method will dump this code into a text file.

- File `Allocations.py` is responsible for the allocation part. From a `RiscVFunction` with temporaries (instructions formed with temporaries), producing an actual RISCV program (instructions with regular registers or memory accesses) is done by the two following steps:

  - First, compute an allocation for each temporary (in the current `RiscVFunction` instance). In Section 4.3, we provide you with `NaiveAllocator.run()` in `Allocations.py` which computes such a (naive) allocation, you will have to design your own allocation function in Section 4.4.

[1] in the library, registers are in capital letters, but in lowercase when they are printed.

  - For each instruction of the program, if the instruction contains a read or write access to a temporary, replace operands with the corresponding actual registers/memory location (and possibly add some instructions before and after). This is done by the use of the `RiscVFunction.iter_instructions` iterator on instructions and `Allocations.replace_reg` methods. In Section 4.4 you will have to write such a "replacement" function.

- The file `MiniCTypingVisitor.py` is the same as the skeleton provided for lab3. You can copy your lab3's `MiniCTypingVisitor.py`, and if your typechecker is buggy, you can use the compiler's `--disable-typecheck` to run the code generation without typechecking (it is activated in the `Makefile` when you run `make run`, this may be changed by setting `DISABLE_TYPECHECK` in `test_codegen.py`).
- The file `Main.py` launches the chain: production of 3-address code with temporaries, allocation, replacement, print.
- The script `test_codegen.py` will help you to test your code. We will use it in Section 4.3.
- A `README.md` file to be completed progressively during the lab.

EXERCISE #1 ► RISCV **Simulator - test**
Re-test the command-line version of the RISCV simulator:

```
riscv64-unknown-elf-gcc toto.s xxx.s -o toto.riscv
spike pk toto.riscv
```

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 a `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 now.
- 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`. The `sp` register points to the first data contained in the stack. It is always 16-byte (2 double-words) aligned.

4.2 First step: three-address code generation

In this section you have to implement the course rules (Figures 4.2 and 4.3) in order to produce RISCV code with temporaries.

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 2019
##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
        # (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_l_while_begin_0:
        li temp_4, 0
        bge temp_0, temp_1, lbl_end_relational_1
        li temp_4, 1
lbl_end_relational_1:
        beq temp_4, zero, lbl_l_while_end_0
        # (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_l_while_begin_0
lbl_l_while_end_0:
        # (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 #2 ► **3-address code generation**
In the archive, we provide you a main and an incomplete `MiniCCodeGen3AVisitor.py`. To test it, type
`make TESTFILE=tests/step1/test00.c`
and observe the generated code in `<samepath>/test00.s`[2]. You now have to implement the 3-address code generation rules seen in the course. Code and test incrementally [3]:

- We give you the code generation for the `println_int` instruction. It basically produces a call to the proper function in the library.
- numerical expressions without variables (constants are expected to hold on 64 bits, no boolean expression for the moment!).
- then (numerical) expressions with variables (assignment is given); we advise you to postpone the implementation of MultiplicativeExpr, and first finish this Lab without them.

[2]We generated RISCV comments with MiniC statements for debug.
[3]Using files in the `TP04/tests/*` directories. All the test files you use will have to be in your archive.

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

```
for i in tests/step1/*.c; do echo "file="$i; python3 Main.py --reg-alloc=none $i > /dev/null; done
```

4.3 Testing with the trivial allocator (and real RISCV instructions), then end of 3@ code generation

The former code 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 available registers. The process takes as input the former 3-address code and transforms each instruction according to the allocation function.

EXERCISE #3 ▸ **Testing the trivial allocator**

Open, read, understand the `NaiveAllocator` implementation in `Allocations.py` and how it is used to perform the actual RISCV code generation [4]. Then, intensively test your former code generation with this allocator [5]:

1. Have a look at the `test_codegen.py` script: comment or uncomment files to test, and what to test.
2. Test with:
   `make TEST_FILES='tests/step1/*.c' tests-naive`
   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 a great test case to test assignments.
   - If the `AllocationError` exception is raised by the naive allocator, the test is skipped.
   - 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.
     `python3 Main.py --reg-alloc naive --stdout tests/step1/test00.c`
     Run `python3 Main.py --help` or see `Main.py` for more options. The `--debug` option allows getting some debug output. Alternatively, you can run the testsuite on a single testfile with:
     `make TEST_FILES=tests/step1/test00.c tests-naive`

At this step, the tests should be OK or SKIPPED for all files given in directory `tests/step1/`:

```
make tests
[...]
========================== xx passed, xx skipped in xx seconds ========
```

"skipped" here means that we cannot compare the output to the ideal output since some of our 3 adress-codes cannot be allocated with registers only. That's life !

Now that we have a way to test our code generation for tiny MiniC codes, we can come back to it.

[4]All available registers are in a list named GP_REGS

[5]Be careful, this allocator crashes if there is more than a certain number of temporaries!

EXERCISE #4 ► **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 what you want.
- while loops;
- if then else. **Be careful with nested ifs and their labels!**.

At this point all the tests should be ok for all files in directory `tests/step2/` (You should modify the test script pathes). However these tests are not sufficient, you should add some other ones (in the directory `tests/mine/`). Run the testsuite with `make tests-naive` to use all the test files.

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

```
self._prog.addInstructionCondJUMP(label, op1, cond, op2)
```

where `op1` (resp `op2`) is the left operand (resp right operand or the numerical constant 0, nothing else), ie a register or a value of the boolean condition (`Condition('eq')` for equality, for instance) [6], and `label` is a label to jump to if the condition evaluates to true.

About nested if-then-else (a bit more difficult) There is an issue with nested ifs. Indeed, how can we remember where to jump after one CondBlock (in `visitCondBlock(self, ctx)`)? We propose to use a label stack called `self.ctx_stack`: each time we enter `visitIfStat`, we push the end label. This label is used in all `visitCondBlock` (at some point you have to insert a jump instruction to the `cond_if` label). At the end of the `visitIfStat` function this label is popped out.

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

Tests Up to now, you used `make tests-naive` to test your code, and at this point all tests should pass. From now, you should use the more complete `make tests-notsmart` command, that tests everything except the smart allocator (that we'll write during the next lab).

Check that `make tests-notsmart` does fail.

Implementation As the number of registers for allocation is bounded by N [7], the naive allocator cannot deal with more than N temporaries: 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, and will not use registers a_1 to a_7 at all for the moment.
- but 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 (as defined in RiscVFunction.printCode).
- $s1, s2, s3$ 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 precautionous 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_ony` method of the `Instruction3A` 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 at will be done following the steps:

- compute a new offset (call to the `new_offset` method of the class `RiscVFunction`).
- generate the following instruction [8]

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

[7] The size of the `GP_REGS` list in the `Operands.py` file, i.e. `len(Operands.GP_REGS)`

[8] **The first version of the codegen course had some errors: you should store 64 bits words, thus use sd and not sw. Refresh the webpage to get the new version of the slides.**

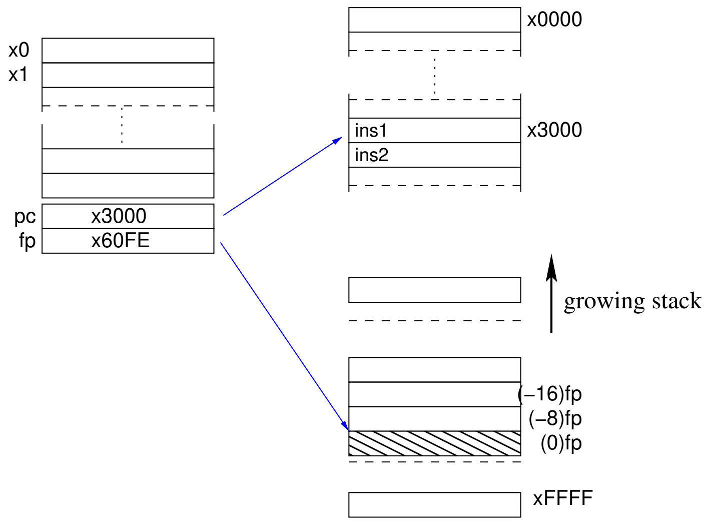


Figure 4.1: Memory model for RISCV

```
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.

EXERCISE #5 ▶ **Manual translation**
Complete the expected output for the following two statements (13/15 lines of RISCV code). `temp_3` is located at -16(fp) and `temp_4` is located at -32(fp):

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

Listing 4.1: 'all in mem alloc for test00b.c'

```
##Generated code without prelude and postlude
        # (stat (assignment x = (expr (atom 4)) ;))
        # li temp_2, 4
        li s3, 4
        sd s3, -48(fp)
        # end li temp_2, 4
        # mv temp_1, temp_2
        ld s2, -48(fp)
        mv s3, s2
        sd s3, -24(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 #6 ► **Implement**

Now you are on your own to implement this code generation. Here are the main steps (less than 50 locs of PYTHON):

1. We have implemented for you an `AllInMemAllocator.run()` method in `Allocations.py`. This method only maps each temporary ("temporary") to a new offset in memory (in a PYTHON `dict`), then iterates the `replace_mem` function on all instructions of the three adress program to perform the actual allocation.

2. In `Allocations.py`, implement a `replace_mem(old_i)` 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_mem` should contain something like

```
# regxxx is the register used to hold the value between the load and
# the operation itself (one of t0, t1, t2).
# operand is the place in memory where the temporary is allocated (of
# the form Offset(..., fp), obtained with get_alloced_loc().
before.append(Instru3A('ld', regxxx , operand))
```

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, there should not be any "skipped" test 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).

4.5 Multiplicative Expressions (multiplication, division, modulo)

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 $(a/b)*b+a\%b=a$.

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

EXERCISE #7 ► **Tests, TOMUSS**

We provide you the same test infrastructure as in Lab 3. Same instructions as the former lab for the archive deposit on Tomuss. **Please to not modify the Makefile, nor the Grammar, nor the code filenames, its structure.** Deadline is January, 21/01 6pm.

c	`dr <-newTemp()`
	`code.add(InstructionLI(dr, c))`
	`return dr`
x	`#get the place associated to x.`
	`regval<-getTemp(x)`
	`return regval`
e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`
e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionSUB(dr, t1, t2))`
	`return dr`
true	`dr <-newTemp()`
	`code.add(InstructionLI(dr, 1))`
	`return dr`
$e_1 < e_2$	`dr <- newTemp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`endrel <- newLabel()`
	`code.add(InstructionLI(dr, 0))`
	`#if t1>=t2 jump to endrel`
	`code.add(InstructionCondJUMP(endrel, t1, '>=' , t2)`
	`code.add(InstructionLI(dr, 1))`
	`code.addLabel(endrel)`
	`return dr`

Figure 4.2: 3@ Code generation for numerical or Boolean expressions (t1 and t2 are already defined)

x = e	`  dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`  find loc the location for var x`
	`  code.add(instructionMV(loc,dr))`

S1; S2	`#concat codes`
	`   GenCodeSmt(S1)`
	`   GenCodeSmt(S2)`

`if` *b* `then` *S*1 `else` *S*2	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add(InstructionCondJUMP(lelse, t1, "=", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

`while` *b* `do` *S* `done`	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add(InstructionCondJUMP(lendwhile, t1, "=", 0))`
	`GenCodeSmt(S)                      #execute S`
	`code.add(InstructionJUMP(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

- Construct the CFG.
- Compute live ranges, construct the interference graph.
- Allocate registers and produce final "smart" code.

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. We recall you that you have slides in the course to help you.

In the code we give you, there is a mention of for loops, and other string treatments, this is an error this code should not have been given to you (and you do not have to implement these todo). We appologize for this inconvience.

You will extend your previous code, in the same directory. People in advance are encouraged to keep their current code, students with more difficulties will be provided a working 3 address code generation Visitor named `MiniCCodegen3AVisitor-correct.py` on Tuesday, 21, end of day. Your work is due on Tomuss at the end of the week.

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:

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

You may have to install the following PYTHON packages:

```
pip3 install --user networkx
pip3 install --user graphviz
pip3 install --user pygraphviz \
  --install-option="--include-path=/usr/include/graphviz" \
  --install-option="--library-path=/usr/lib/graphviz/"
```

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

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

and then relaunch the command `pip3 install ...` On the university machines, you might have to update existing already installed packages:

```
pip3 install --user --upgrade networkx graphviz pygraphviz
```

5.1 CFG construction

In class we have presented CFGs with maximal basic blocks. In this lab we will implement CFGs with minimal basic blocks that is CFG with one node per line of code/instruction (even comments).

EXERCISE #1 ► **CFG By hand**
What are the expected result of the CFG construction from the 3-address code of Lab5 for each of these programs ?

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

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

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

EXERCISE #2 ► **CFG Construction**
The `APIRiscV` is able to deal with CFGs. `Instructions` have a list of predecessors (`self._in`) and successors (`self._out`) and a `RiscVFunction` contains the initial control point (`self._start`) from which we can traverse the graph. This feature allows us to easily construct the CFG of a program.

We give you the construction for all idioms. Each time your Visitor creates a new RISCV instruction, the CFG updates itself automatically: when adding an instruction, it creates an edge between the last instruction (`self._end`) and the instruction to be added.

In this exercise, you only have to understand (look at the API!) and test the provided code.

When ran with `--graphs`, `Main.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.

Now you can launch:
`python3 Main.py --reg-alloc smart --graphs /path/to/example.c`

1. Test for lists of assignments (for instance `testdataflow/df01.c`) You should see a chain of blocks.
2. Same for boolean expressions, and tests.
3. Same for while loops ....
4. Propose appropriate examples and draw nice pictures!

5.2 Liveness analysis and Interference graph

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

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

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

The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.
From now on, you have to modify `Allocations.py`

EXERCISE #3 ► **Liveness Analysis, Initialisation**
Initialise the $\mathsf{Gen}(B)$ and $\mathsf{Kill}(B)$ for each kind of instruction (add, let, ...). This is done by the `set_gen_kill` method of `SmartAllocator`. 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 `Allocations.py` (function `SmartAllocator.run()`) should help you (use with `Main.py --debug`, which sets debug=True for you):

```
if debug:
    self.print_gen_kill()
```

As an example, here is the expected initialisation for `testsdataflow/df04.c`, obtained by:
`python3 Main.py --debug --reg-alloc smart testsdataflow/df04.c`

```
instr 0: comment
gen: {}
kill: {}

instr 1: li temp_2, 2
gen: {}
kill: {temp_2}

instr 2: mv temp_1, temp_2
gen: {temp_2}
kill: {temp_1}

instr 3: comment
gen: {}
kill: {}

instr 6: li temp_3, 4
gen: {}
kill: {temp_3}

instr 8: li temp_4, 0
gen: {}
kill: {temp_4}

instr 9: bge temp_1, temp_3,
             lbl_end_relational_2
gen: {temp_1,temp_3}
kill: {}

instr 10: li temp_4, 1
gen: {}
kill: {temp_4}

instr 7: lbl_end_relational_2
gen: {}
kill: {}

instr 11: beq temp_4, zero, lbl_end_cond_1
gen: {temp_4}
kill: {}

instr 12: li temp_5, 4
gen: {}
kill: {temp_5}

instr 13: mv temp_1, temp_5
gen: {temp_5}
kill: {temp_1}

instr 14: j lbl_end_if_0
gen: {}
kill: {}

instr 5: lbl_end_cond_1
gen: {}
kill: {}

instr 15: li temp_6, 5
gen: {}
kill: {temp_6}

instr 16: mv temp_1, temp_6
gen: {temp_6}
kill: {temp_1}

instr 4: lbl_end_if_0
gen: {}
kill: {}
```

EXERCISE #4 ► **Liveness Analysis, fixpoint. (Only test!)**

We implemented for you the fixpoint iteration as a method (run_dataflow_analysis) in Allocations.py "while it is not finished, store the old values, do an iteration, decide if its finished". The run_dataflow_analysis program method makes calls to dataflow_one_step instruction methods. The result (live in, live out sets of variables, are stored in _mapin and _mapout member sets of the SmartAllocator class).

All 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 testsdataflow/ directory.

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

```
mapin, mapout = self.run_dataflow_analysis()
if debug:
    self.print_map_in_out()
```

As an example, here is the expected output for testsdataflow/df04.c:

```
In: {0: {}, 1: {}, 2: {temp_2}, 3: {temp_1}, 4: {}, 5: {},
     6: {temp_1}, 7: {temp_4}, 8: {temp_1,temp_3},
     9: {temp_1,temp_4,temp_3}, 10: {}, 11: {temp_4},
     12: {}, 13: {temp_5}, 14: {}, 15: {}, 16: {temp_6}, 17: {}, 18: {}}
```

```
Out: {0: {}, 1: {temp_2}, 2: {temp_1}, 3: {temp_1}, 4: {}, 5: {},
      6: {temp_1,temp_3}, 7: {temp_4}, 8: {temp_1,temp_4,temp_3},
      9: {temp_4}, 10: {temp_4}, 11: {},
     12: {temp_5}, 13: {}, 14: {}, 15: {temp_6}, 16: {}, 17: {}, 18: {}}
```

EXERCISE #5 ► **Interference graph**

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

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

For the two last cases, consider the following list of instructions:

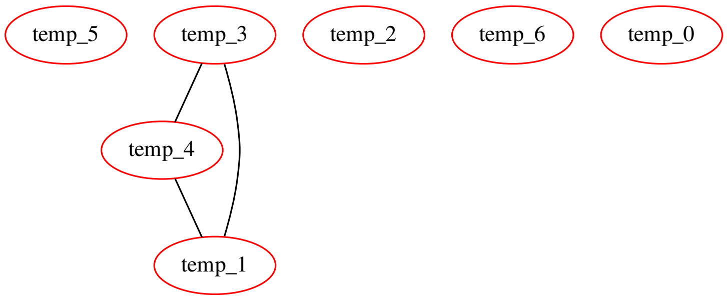
```
y=2
x=1
z=y+1
```

where x is not alive after the `x=1` statement, however x is in conflict with y since we generate the code for `x=1` while y is alive[1].

From the result of the previous exercise, construct the interference graph (complete the `build_interference_graph` function) of your program (each time a pair of temporaries are in conflict, add an edge between them). We give you a non-oriented graph API (`LibGraph.py`) for that. Use the `print_dot` method and relevant tests to validate your code.

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 `df04.c` (command line as usual):



5.3 Register allocation and code production

Instead of the iterative algorithm of the course, we will implement the following algorithm for k register allocation:

- Color the interference graph with an infinite number of colors, using the first ones as much as possible.
- The first $k-3$ 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 memory allocation:

- For non-spilled variable: replace the temporary with its associated color/register.
- For spilled variables: allocate in memory.

Some help:

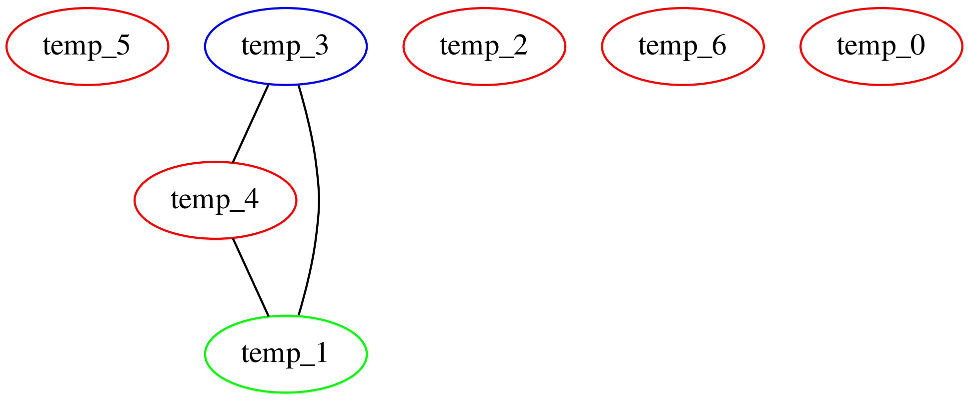
[1] Another solution consists in eliminating dead code before generating the interference graph.

- 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 a constructor Offset(SP, xxx) (all in Operands.py).
- Be careful with types when dealing with the graph. As the comment in Allocations.py states, self._igraph contains only elements of type string, while the alloc_dict map given to self._pool.set_temp_allocation() must have Temporary objects as keys. There is no easy way to retrieve a Temporary object from its name, but it is easy to get the name as a string from a Temporary: just use str(). The easiest way to build alloc_dict is probably to iterate over all the temporaries of the program (available in self._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 #6 ► **Smart Register Allocation: implement!**
Use the algorithm and the coloration method of the LibGraphes class to allocate registers (or a place in memory). For that, you have to complete the program method SmartAllocator.run() (in file Allocations.py). Comments will help you design this (non trivial) function. [2]. The allocation is followed by statement rewriting, like in previous lab. You need to implement it in Allocations.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.

On the df04.c example, the graph coloring succeeds with:



EXERCISE #7 ► **Massive tests**
Comment out all the print_dot instructions, debug, ... and test on all test files you have. **In particular, we do not want any pdf file to be opened when we will use make tests on your delivered code.**

EXERCISE #8 ► **Lab delivery**
Make a clean archive with README.md, your test files, ... (same instructions as before) and put it on TOMUSS before Sunday January, 26th, 23:59.

http://tomuss.univ-lyon1.fr

[2]it seems that we also gave you the interfere function this year, this was not supposed to happen, ...

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 - adapted for CAP and MIF08

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:

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

.equ CONSTANT, 0xcafebabe

        li a0, CONSTANT
```

which generates the following assembler output as seen by objdump:

```
0000000000000000 <_start>:
   0:   00032537            lui     a0,0x32
   4:   bfb50513            addi    a0,a0,-1029
   8:   00e51513            slli    a0,a0,0xe
   c:   abe50513            addi    a0,a0,-1346
```

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 for CAP and MIF08

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

Directive	Arguments	Description
.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
.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

RV32I Base Integer Instructions - CAP 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`		`rd = (rs1 < rs2)?1:0`	
`sltu`	Set Less Than (U)	R	`0110011`	`0x3`		`rd = (rs1 < rs2)?1:0`	zero-extends
`addi`	ADD Immediate	I	`0010011`	`0x0`	`0x00`	`rd = rs1 + imm`	
`xori`	XOR Immediate	I	`0010011`	`0x4`	`0x00`	`rd = rs1 ^ imm`	
`ori`	OR Immediate	I	`0010011`	`0x6`	`0x00`	`rd = rs1 | imm`	
`andi`	AND Immediate	I	`0010011`	`0x7`	`0x00`	`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

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/