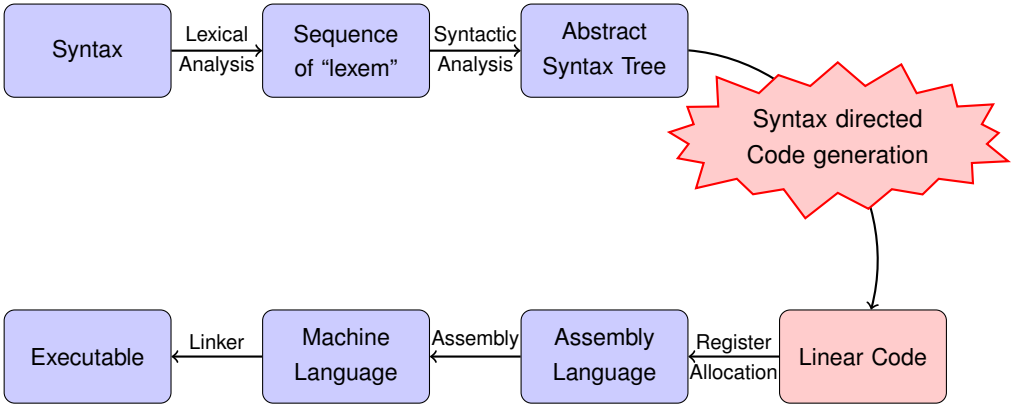
# CAP— Compilation (#4) : Syntax-Directed Code Generation

Laure Gonnord & Matthieu Moy & Gabriel Radanne & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2025-2026


# Big Picture


Syntax
Lexical Analysis
Sequence of “lexem”
Syntactic Analysis
Abstract Syntax Tree
Syntax directed Code generation
Linear Code
Register Allocation
Assembly Language
Assembly
Machine Language
Linker
Executable

# Rules of the Game here

For this code generation:

- Still no functions and no non-basic types. (MiniC w/o functions and strings)
- Syntax-directed: one grammar rule $\rightarrow$ a set of instructions.
  - Code redundancy.
- No register reuse: each value is stored in its own location (potential memory waste)

The Target Machine: RISCV (cf. previous course)

# Code Generation vs Memory/Register Allocation

- Code generation in two steps:
  1. Generate instructions without deciding where data is stored (put everything it temporaries)
  2. Decide where each temporary is allocated (register? stack?)
- Temporary (sometimes called “virtual register”): temporary where data can be stored. Difference with (physical) registers:
  - They don’t exist in the real processor / instruction set
  - There are an infinity of them

# A first example (1/2)

How do we translate:

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

- Variable decl's visitor gives a temporary to each variable: $x \mapsto temp0$, $y \mapsto temp1$.
- Compute $4$, store somewhere, then copy in $x$'s temporary.
- Compute $12 + x$ : 12 in `temp2`, copy the value of x in `temp3`, then add, store in `temp4`, then copy into $y$ (i.e. `temp1`).

▶ Create temporaries whenever needed.

# A first example: 3@code (2/2)

“Compute 4 and store in x (temp0)”:

```
li temp2, 4
mv temp0, temp2
```

# Objective

**3-address** RISCV **Code Generation** for the Mini-While language:

- All variables are int/bool.
- All variables are global.
- No functions (just `main()`).

with syntax-directed translation. Implementation in Lab (`MiniC/`)

- This is called **three-address code generation**

# Code generation utility functions

We will use:

- A new (fresh) temporary can be created with a `fresh_tmp()` function.
- A new fresh label can be created with a `fresh_label()` function.
- The generated instructions are close to the RISCV ones.

# Abstract Syntax

Expressions:

| | | | |
|---|---|---|---|
| $e$ | $::=$ | $c$ | constant |
| | $\mid$ | $x$ | variable |
| | $\mid$ | $e + e$ | addition |
| | $\mid$ | $e$ or $e$ | boolean or |
| | $\mid$ | $e < e$ | less than |
| | $\mid$ | ... | |

Statements:

| | | | |
|---|---|---|---|
| $S$ | $::=$ | $x := expr$ | assign |
| | $\mid$ | $skip$ | do nothing |
| | $\mid$ | $S_1; S_2$ | sequence |
| | $\mid$ | `if` $b$ `then` $S_1$ `else` $S_2$ | test |
| | $\mid$ | `while` $b$ `do` $S$ `done` | loop |

# Code generation for expressions, example

| e ::= c (cte expr) | dest <- fresh_tmp()<br>code.add("li *dest*, *c*")<br>**return** dest |
|---|---|

- this rule gives a way to generate code for any constant.

# Code generation for a boolean expression, example

| e ::= $e_1 < e_2$ | |
|---|---|

```
dest <- fresh_tmp()
t1 <- GenCodeExpr(e1)
t2 <- GenCodeExpr(e2)
endrel <- fresh_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
```

- integer value 0 or 1 to represent true/false.

# Second example: a boolean test

Let us generate the code for $x < 4$ (assuming $x$ is stored in `temp0`):

# Second example: a boolean test

Let us generate the code for $x < 4$ (assuming $x$ is stored in `temp0`):

```
li temp3, 4 // get 4
li temp2, 0
bge temp0, temp3, lbl0 // >= comp + jump
li temp2, 1
lbl0:
// Here, temp2 contains 1 if x<4, 0 otherwise
```

# Code generation for commands, example

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

# Example for `if/else`

Let us generate the code for `if (x<4) then y=7 else ...` ($y$ in `temp1`)

# Example for `if/else`

Let us generate the code for `if (x<4) then y=7 else ...` ($y$ in `temp1`)

```
## code from previous slide here to compute x<4
beq temp2, zero, lelse1 // if false, jump
li temp4, 7
mv temp1, temp4 // y gets 7
j lendif1 // don't forget this one!
lelse1:
  # code for else branch
lendif1:
```

# Example for `if/else`

Let us generate the code for `if (x<4) then y=7 else ...` ($y$ in `temp1`)

```
## code from previous slide here to compute x<4
beq temp2, zero, lelse1 // if false, jump
li temp4, 7
mv temp1, temp4 // y gets 7
j lendif1 // don't forget this one!
lelse1:
  # code for else branch
lendif1:
```

Note: more efficient version possible by branching directly to `lendif1`/`lelse1` in the generation of $x < 4$ (get rid of `temp2`, save one `beq`), but this is sufficient for us.

# From 3@ code to valid RISCV

3@code is not valid RISCV code!

We explore several allocation algorithms:

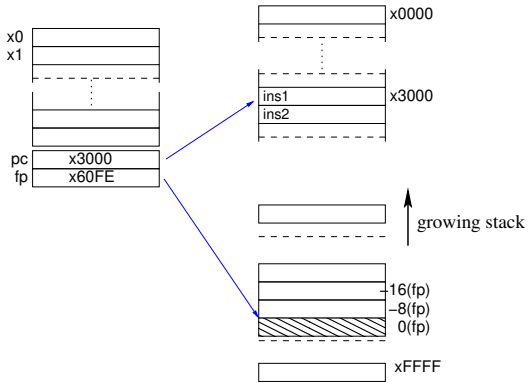
- All in registers $temp_i \rightarrow register$ $\leftarrow$ very, very naive
- All in memory $temp_i \rightarrow memory$ $\leftarrow$ very naive
- Store several temporaries in the same place, use registers and memory
  $\leftarrow$ yes, we'll do smart stuff too :-)

# A stack, why?

- Store local variables of each functions
- Provide an easy way to communicate arguments values
- Give place to store intermediate values (e.g. $2*3$ in `x = 1 + 2 * 3`)

# Stack with RISCV

- Special register `fp` = Frame Pointer, points to the current stack frame.
- Store and loads from `fp`




Nice picture by N. Louvet - adapted in 2019

# How to store into the stack

**Store (the content of) $s_3$ on the stack at offset `offset`:**

```
sd s3, -offset*8(fp)
# To generate from Python:
# sd(s3, Offset(FP, -offset*8))
# "write the value of s3 at address fp - offset*8"
```

# Exercice: 3 address code generation for

```
i = 0;
if (i == 10) {
   i = i + 1;
} else {
   i = i - 1;
}
```

# Exercice: naive allocation (all in registers)

```
1 li temp_0, 42
2 li temp_1, 1
3 add temp_2, temp_1, temp_0
```

# Exercice: “all in mem” allocation

```
1 li temp_0, 42
2 li temp_1, 1
3 add temp_2, temp_1, temp_0
```

# Code Generation

Input: a MiniC file:

```
int main(){
int n;
n=6;
return 0;}
```

Output: a RISCV file:

```
1  [...]
2        ;; (stat (assignment n = (expr (atom 6)) ;))
3        LI t1, 6    ; t1 is a riscv register.
4        MV t2, t1
5  [...]
```

# New elements in this Lab

Compared to previous labs:

- A new Compiler library that you will have to use
- Several steps that you will have to assemble
- Even more tests !

# Two-Step process

As we have seen, we proceed in two steps:

1. Emit 3 Address Code
2. Do register allocation

# 3 Address Code

We generate Linear code (class `Lib/LinearCode.py`).

*class* **Lib.LinearCode.LinearCode**(*name: str*) [source]

Bases: `object`

Representation of a RiscV program as a list of instructions.

**add_comment**(*s: str*) → None [source]

Add comment *s* at the end of the program.

**add_instruction**(*i: Lib.Statement.Comment | Lib.Statement.Label | Lib.Statement.Instru3A | Lib.Statement.AbsoluteJump | Lib.Statement.ConditionalJump*) → None [source]

Add instruction *i* at the end of the program.

# Code Generation, first step

- 3-address codegen according to the code generation rules of the course:

```
// e1+e2 code generation rule
  temp_1 <- GenCodeExpr(e_1)
  temp_2 <- GenCodeExpr(e_2)
  dest_tmp <- fresh_tmp()
  code.add(add(dest_tmp, temp_1, temp_2))
  return dest_tmp
```

- **TODO: implement them**:

```
tmpl = self.visit(ctx.expr(0))
tmpr = self.visit(ctx.expr(1))
dest_temp = self.fresh_tmp()
if ctx.myop.type == MiniCParser.PLUS:
    self.add_instruction(RiscV.add(dest_temp, tmpl, tmpr))
```

# Result after first step

The previous step uses instructions of an API like:

```
self._current_function.add_instruction(RiscV.add(dest_temp, tmpl, tmpr))
```

whose side effect is to construct a RISCV prog as a list of 3 adresses instructions with temporaries (virtual registers, from the class `Temporary`).

This list can be dumped (with `printCode` in the API) into a .s file:

```
1        ;; (stat (assignment n = (expr (atom 6)) ;))
2        li temp_1, 6
3        mv temp_2, temp_1
```

We cannot test: it is not executable!

# Code Generation, second step

The allocation process (`Lib/Allocator.py`):

- takes as input the preceding result
- modifies the list of instructions with temporaries into list of instructions with physical registers or accesses to memory.
- a trivial (“Naive”) allocator is given.

**TODO : all in memory allocation (see course)**

# Code Infrastructure (only files for THIS LAB))

```
MiniC$ ls
Makefile MiniC.g4 test_codegen.py MiniCC.py [...]

MiniC$ ls Lib/
Allocator.py  Errors.py  LinearCode.py  Operands.py  RiscV.py  Statement.py

MiniC$ ls TP04/
MiniCCodeGen3AVisitor.py AllInMemAllocator.py
```

- The MiniC grammar in `MiniC.g4`, a Makefile, as usual.
- Unit tests in `test_codegen.py`.
- An API for compiler writing `Lib/` ...
- **TODO : edit and fill the files in `TP04` ONLY**

# Compiler Library

In this Library:

- Classes for statements and instructions `Lib/Statement.py`: `Statement`, `Instru3A`, `Label`, ...
- A 3 address instruction contains operands `Lib/Operands.py`: `Immediate`, `Temporary`, `Register`, or a `Condition`
- A program (`Lib/LinearCode.py`) contains a list of instructions
- RiscV-specific instructions (“add”, “li”, ...) and tools to create new temporaries and labels in `Lib/RiscV.py`
- The base class for allocators `Lib/Allocator.py`.

# Coding tools

Documentation for `Lib/` on the website, and with `make doc`.

Read the documentation!!

We use `pyright` for typing.

Use the typechecker!!!

# tests

While developing, write appropriate (mini) tests and use :

```
pyright && python3 MiniCC.py --reg-alloc=xxx /path/to/example.c
```

and have a look at the generated file.

Next step is to verify everything:

```
make test
```

to launch all tests in `tests*/*` files.

# Drawbacks of this “all in mem” allocation

Drawbacks:

- Memory intensive loads and stores (each operation loads and store from memory)
- Uses a lot of memory (no reuse of memory for different computations)

Next:

- Store as many temporaries as possible in the same location
  $\rightsquigarrow$ Are two temporaries in conflict (i.e. used at the same time)?

▶ We need a more efficient data structure to reason on: **the control flow graph (CFG)**. (see next course)

# Summary : 3 address code generation