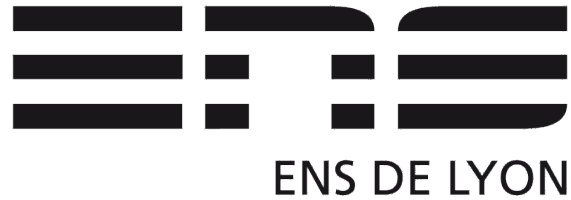
Compilation (#5) : Syntax-Directed Code Generation

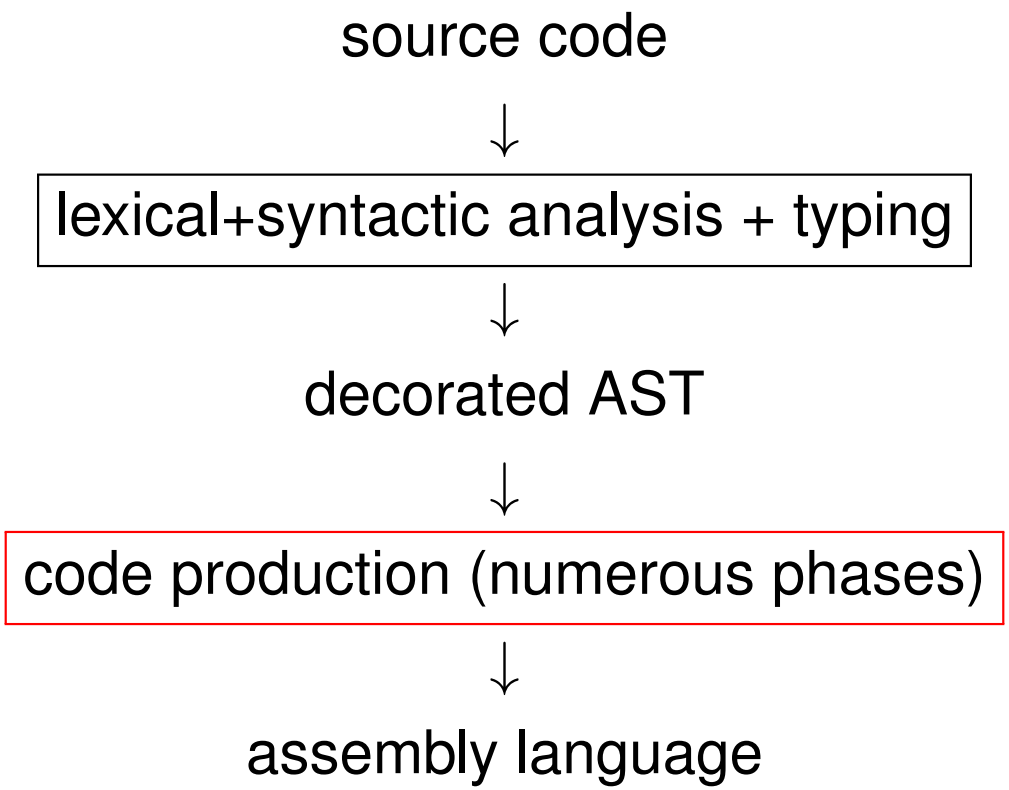
Laure Gonnord & Matthieu Moy & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2021-2022



Big picture


source code
lexical+syntactic analysis + typing
decorated AST
code production (numerous phases)
assembly language

Rules of the Game here

For this code generation:

- Still no functions and no non-basic types. (mini-while)
- Syntax-directed: one grammar rule $\rightarrow$ a set of instructions.
  - Code redundancy.
- No register reuse: everything will be stored on the stack.

The Target Machine : RISCV (course #1)

A first example (1/2)

How do we translate:

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

- Variable decl's visitor gives a place to each variable: $x \mapsto place0$, $y \mapsto place1$.
- Compute 4, store somewhere, then copy in x's place.
- Compute $12 + x$: 12 in `place2`, copy the value of x in `place3`, then add, store in `place4`, then copy into $y's$ place.

▶ the code generator will use a place generator called `new_tmp()`

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

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

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

Code generation utility functions

We will use:

- A new (fresh) temporary can be created with a `new_tmp()` function.
- A new fresh label can be created with a `new_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
	$\mid$	...	

and statements:

$S(Smt)$	$::=$	$x := expr$	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)	dr <-new_tmp() code.add(InstructionLI(dr, c)) return dr

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

Code generation for a boolean expression, example

e ::= $e_1 < e_2$	

```
dr <-new_tmp()
t1 <- GenCodeExpr(e1)
t2 <- GenCodeExpr(e2)
endrel <- new_label()
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
```

- integer value 0 or 1.

Second example: a boolean test

Let us generate the code for $x < 4$:

```
li temp3, 4 // get 4
li temp2, 0
geq temp0, temp3, lbl0 // >= comp + jump
li temp2, 1
lbl0:
```

(temporary values on board)

Code generation for commands, example

`if` b `then` $S1$ `else` $S2$	`lelse,lendif <-new_labels()`
	`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)`

Example for tests.

Let us generate the code for `if x<4 then y=7 else ...`

```
## preceding code
beq tmp2, zero, lelse1 // if false, jump
li temp4, 7
mv temp1, temp4 // y gets 7
jump lendif1
lelse1:
  # code for else branch
lendif1:
```

From 3@ code to valid RISCV

3@code is not valid RISCV code !

3 “kinds of allocation”:

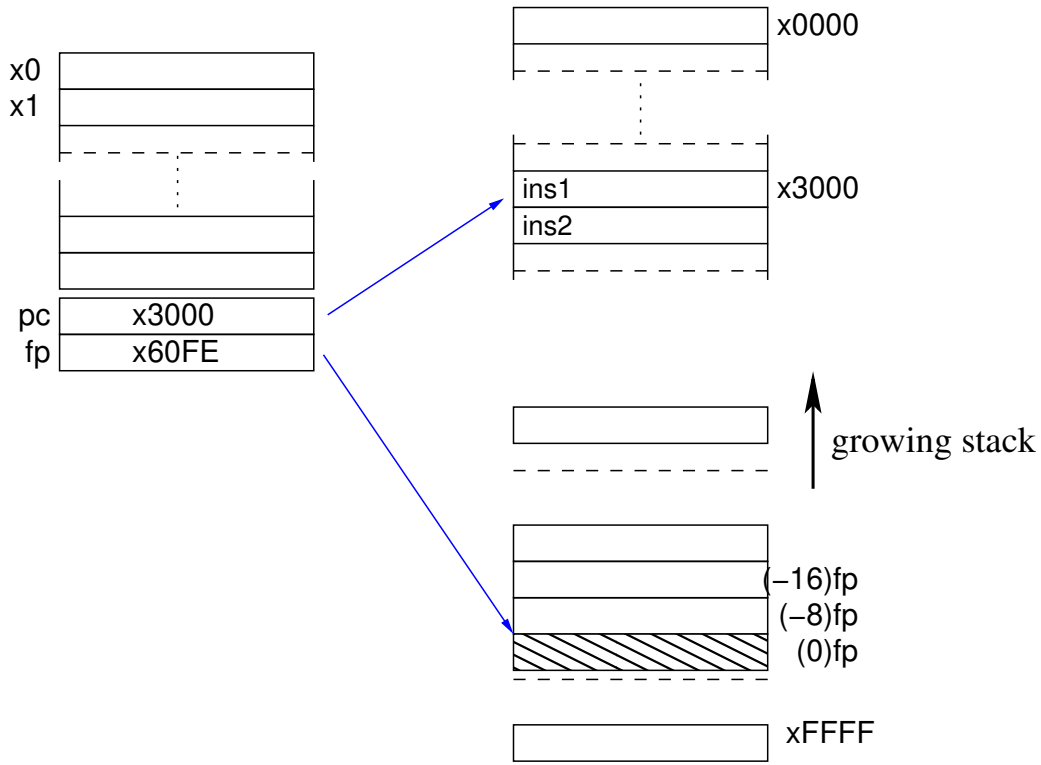
- All in registers (but ?) $place_i \rightarrow register$
- All in memory (here!) $place_i \rightarrow memory$
- Something in the middle (later!)

A stack, why ?

- Store constants, strings, . . .
- Provide an easy way to communicate arguments values (see later)
- Give place to store intermediate values (here)

Stack with RISCV

- There is a special register `fp`.
- 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)
# Instru3A('sd', s3, Offset(FP, -offset*8))
# "write the value of s3 at address fp - offset*8"
```

Code Generation

Input: a MiniC file:

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

Output: a RISCV file:

```
  [...]
2         ;; (stat (assignment n = (expr (atom 6)) ;)
   )
          li t1, 6     ; t1 is a riscv register.
          mv t2, r1
  [...]
```

Steps

- 3-address code generation according to the code generation rules.
- Simple register/memory allocation + pretty print.

Details in the dedicated video/slides.

Drawbacks of the former translation

Drawbacks:

- redundancies (constants recomputations, . . .)
- memory intensive loads and stores.

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

Summary : 3adress code generation

1. 3-address syntax-directed Code Generation
   - Rules
2. Memory allocation
3. LAB: Direct Code Generation
4. Conclusion

Exercice: 3 address code generation for

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

Exercice: naive allocation (all in registers)

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

Exercice: “all in mem” allocation

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

Empty slide for drawing (1)

Empty slide for drawing (2)

Empty slide for drawing (3)

Empty slide for drawing (4)