Compilation and Program Analysis (#5) : Syntax-Directed Code Generation

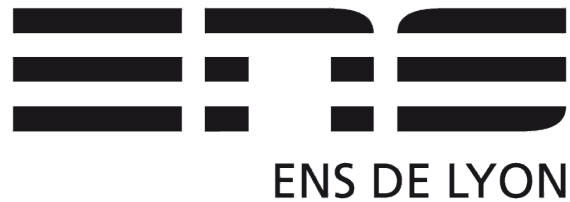
Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

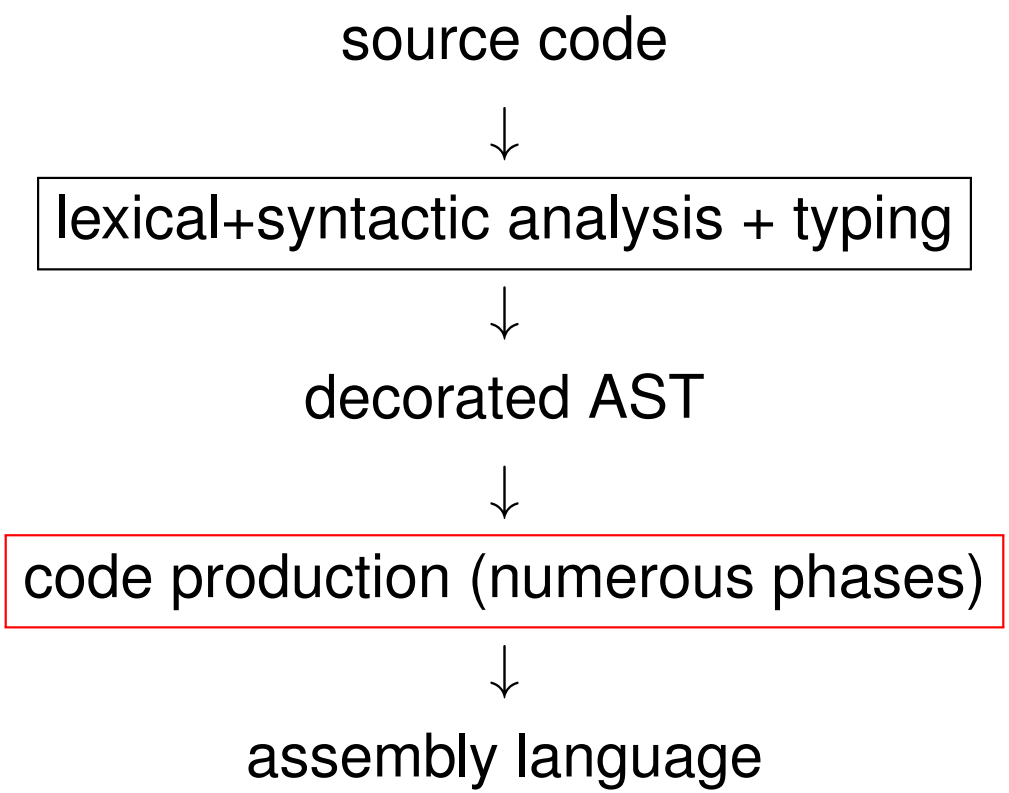
Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

oct 2017



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 : LEIA (course #1)

A first example (1/4)

How do we translate :

```
var x;y:int;
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 `place1`, copy the value of x in `place2`, then add, store in `place3`, then copy into $y's$ place.

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

A first example : 3@code (2/4)

"Compute 4 and store in x (temp0)" :

```
.let temp2 4
copy temp0 temp2
```

Objective

3-address LEIA **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.

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

Code generation utility functions

We will use :

- A new (fresh) temporary can be created with a `newtemp()` function.
- A new fresh label can be created with a `newlabel()` function.
- The generated instructions are closed to the LEIA ones (except for `snif`)

Abstract Syntax

Expressions :

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

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 <-newTemp() code.add(InstructionLET(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 <-newTemp()
t1 <- GenCodeExpr(e1)
t2 <- GenCodeExpr(e2)
dr <- newTemp()
endrel <- newLabel()
code.add(InstructionLET(dr, 0))
#if t1>=t2 jump to endrel
code.add(InstructionCondJUMP(endrel, t1, ">=" , t2)
code.add(InstructionLET(dr, 1))
code.addLabel(endrel)
return dr
```

- integer value 0 or 1.

Code generation for commands, example

if b then $S1$ else $S2$	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)

A first example : from 3@ code to valid LC-3 (3/5)

3@code is not valid LEIA code !

3 “kinds of allocation” :

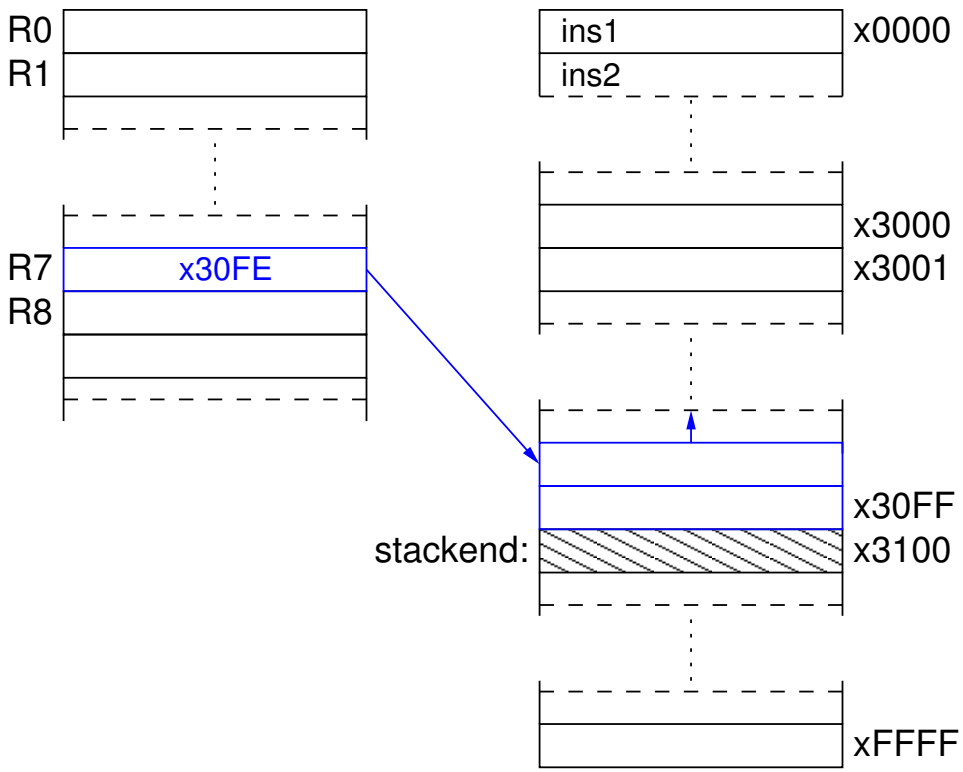
- All in registers (but ?) $place_i \rightarrow register$
- All in memory (here !) $place_i \rightarrow memory$
- Something in the middle (later !)

A stack, why ?

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

LEIA stack emulation - from the archi course

- r_6 is initialised to the `stack` address.
- addresses will be computed from this base.
- The stack grows in the dir. of **decreasing addresses !**.



Nice picture by N. Louvet

A first example : prelude/postlude 4/5

Here **store r_1 on the stack !**

```
[init r6]
.let r1 4
sub r0 r6 1   ; first dec from r6 (and store some info!)
wmem r1 [r0]  ; now r1 can be recycled
```

A first example : prelude/postlude 5/5

The rest of the code generation :

```
.set r6 stack
[...]
jump 0
.align16
stackend:
.reserve 42
stack:
```

- This is valid LEIA code that can be assembled and executed

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)