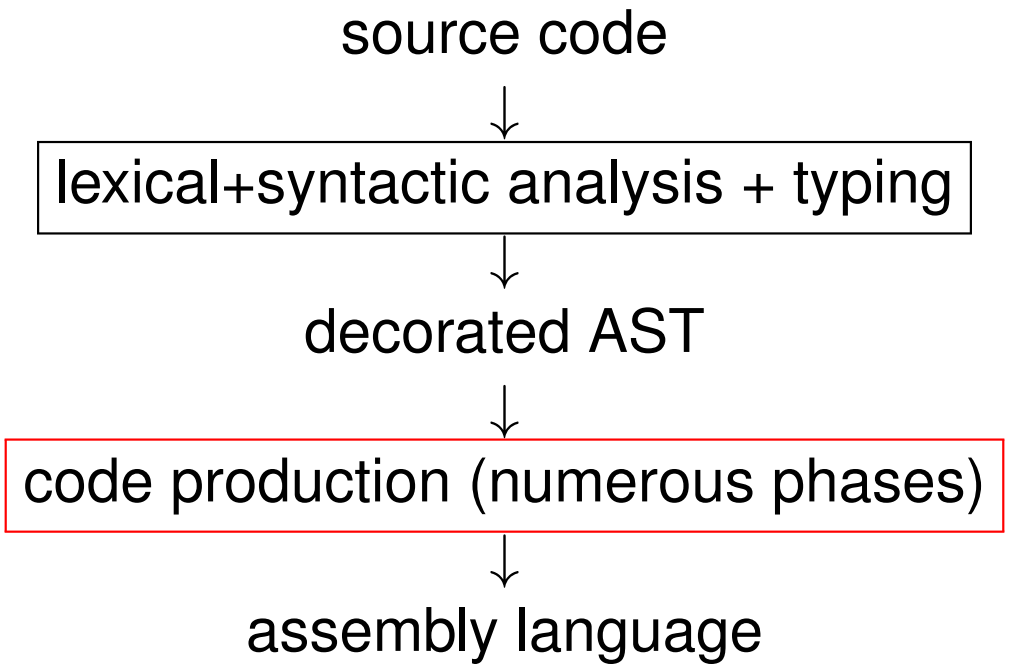
Code Generation

MIF08

Laure Gonnord
Laure.Gonnord@univ-lyon1.fr

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)

Outline

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**

Outline

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 close 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
	$\mid$	...	

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)`

Outline

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

3@code is not valid LEIA code !
3 “kinds of allocation”:

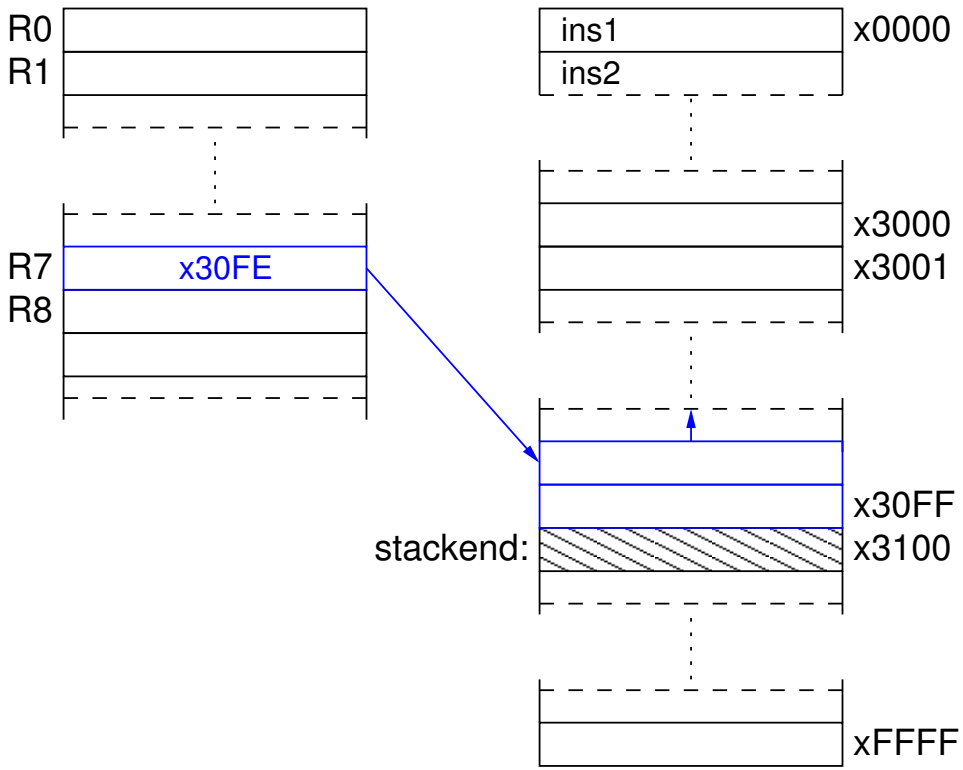
- All in registers (but ?) $place_i \rightarrow register$
- All in memory (here!) $place_i \rightarrow memory$
- Something in the middle (later!)

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

Outline

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)

Compilation and Program Analysis (#6) :

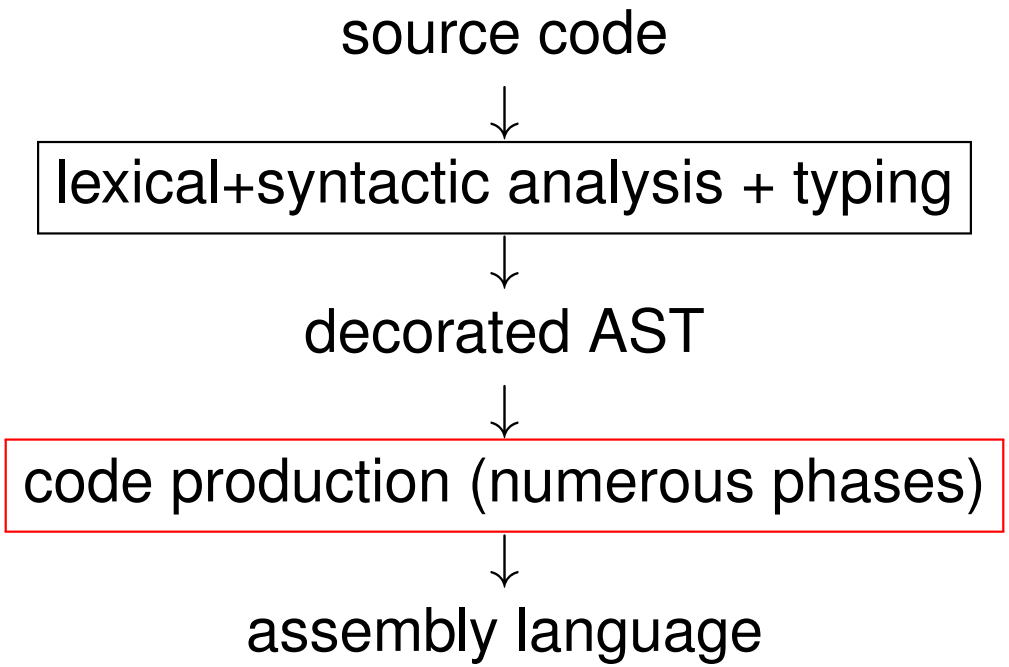
Control Flow graphs and dataflow analysis

MIF08

Laure Gonnord

Laure.Gonnord@univ-lyon1.fr

Big picture


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

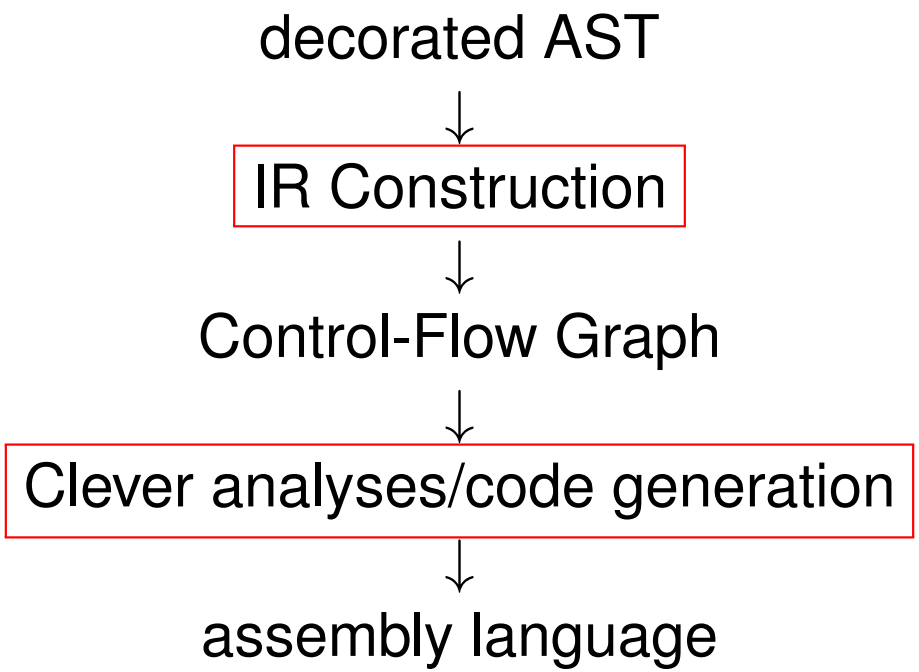
In context 1/2

In the last course we saw the need for a better data structure to propagate and infer information. We need:

- A data structure that helps us to reason about the flow of the program.
- Which embeds our three address code.

- Control-Flow Graph.

In context 2/2



Outline

Definitions

Basic Block

Basic block: largest (3-address LEIA) instruction sequence without label. (except at the first instruction) and without jumps and calls.

CFG

It is a directed graph whose vertices are basic blocks, and edge $B_1 \to B_2$ exists if B_2 can follow immediately B_1 in an execution.

- two optimisation levels: local (BB) and global (CFG)

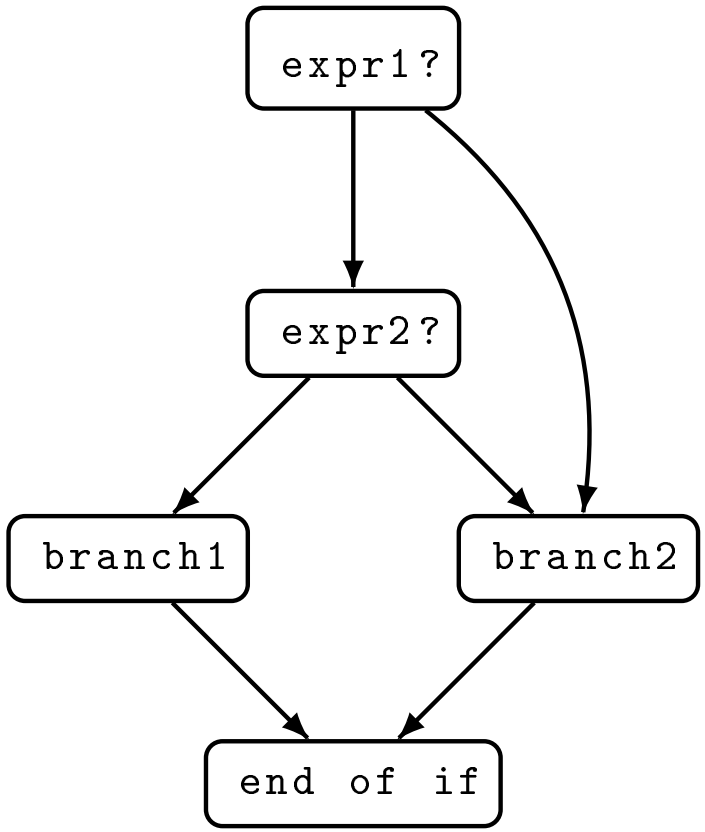
Exercise

Generate the “high level” CFG for the given program:

```
p:=0;i:=1;
while (i <= 20) do
  if p>60 then
     p:=0;i:=5;
  endif
  i:=2*i+1;
done
k:=p*3;
```

CFG for tests

```
if (expr1 and expr2)
   ...branch1...
else
   ...branch2...
```



(blocks are subgraphs)

Lab remark

1 “three adress statement” per block in the lab.

Outline

Registers, memory conflicts

How to decide if two temporaries can be stored in the same register/place in memory ? We need **information**.

Liveness analysis

In the sequel we call **variable** a pseudo-register or a physical register.

Alive Variable

In a given program point, a variable is said to be *alive* if the value it contains may be used in the rest of the execution.

May: non decidable property ▶ overapproximation.

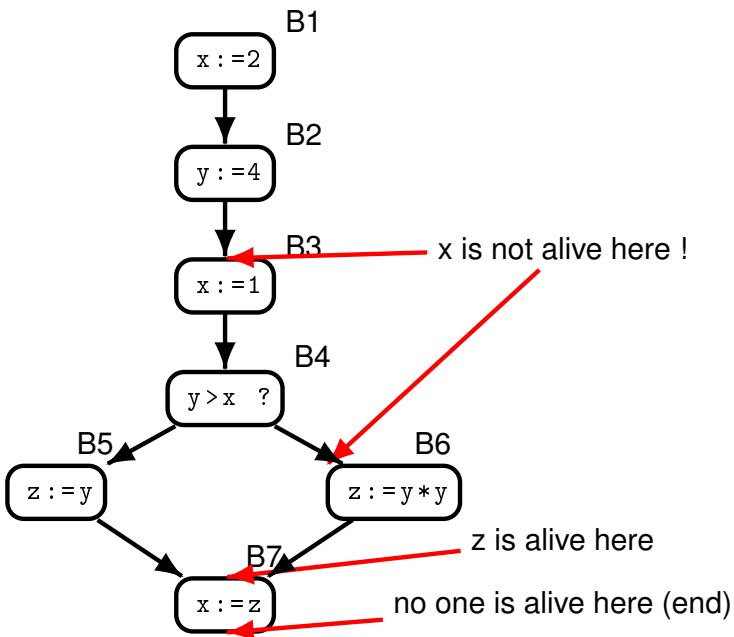
Important remark: here a block = a statement/program point.
We have the same kind of analyses with block=basic block.

An example for live ranges

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

```
x:=2;
y:=4;
x:=1;
if (y>x)
   then z:=y
   else z=y*y ;
x:=z;
```



- The information flow is **backward**: from uses to definitions.

Data flow expressions

Definition

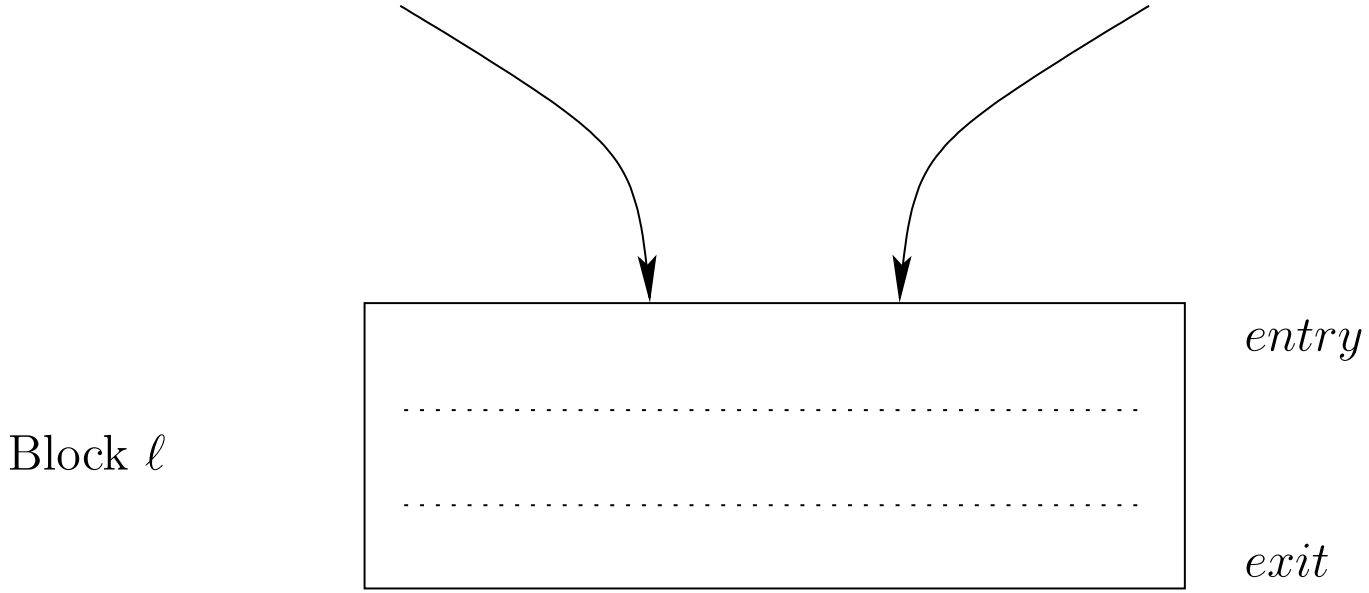
A variable that appears on the left hand side of an assignment is **killed** by the block. Tests do no kill variables.

Definition

A **generated** variable is a variable whose value is used in the block (before any assignment).

- Sets : $kill_{LV}(block)$ and $gen_{LV}(block)$

Data flow expressions



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

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

Data flow equation: solving

Here:

- Initialise LV sets to $\emptyset$.
- Compute LV_{entry} sets, then LV_{exit}, and continue.
- Stop when a fix point is reached.

▶ (vector of) Sets are strictly growing, and the live range set is at most the set of all variables, thus **this algorithm terminates**.

Steps

$LV_{entry}(\ell)$ denoted by $In(\ell)$, $LV_{entry}(\ell)$ by $Out(\ell)$ initilisation to emptysets is not depicted.

			Step 1		Step 2		Step 3 (stable)
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$
1	$\{x\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{y\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{y\}$	$\emptyset$
3	$\{x\}$	$\emptyset$	$\emptyset$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$
4	$\emptyset$	$\{x,y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$
5	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$	$\{z\}$	$\emptyset$	$\{z\}$	$\emptyset$	$\{z\}$

Final result and use

Backward analysis and we want the smallest sets, here is the final result : (we assume all vars are dead at the end).

ℓ	$LV_{entry}(\ell)$	$LV_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\emptyset$

- Use : Dead code elimination ! Note : can be improved by computing the use-defs paths. (see Nielson/Nielson/Hankin)