Code Generation

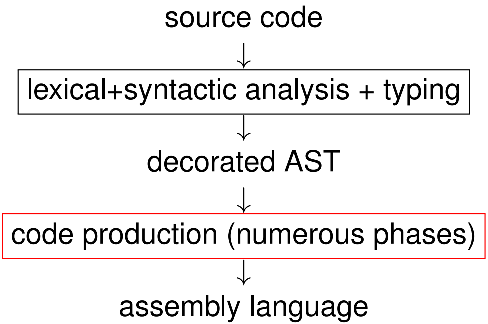
MIF08

Laure Gonnord
`Laure.Gonnord@univ-lyon1.fr`

oct 2016

Lyon 1

Big picture


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

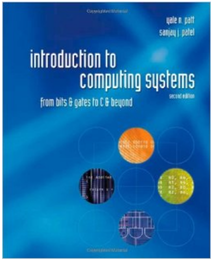
Rules of the Game here

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 : LC3 (course #1)

[*Introduction to Computing Systems: From Bits and Gates to C and Beyond*, McGraw-Hill, 2004].



See also:

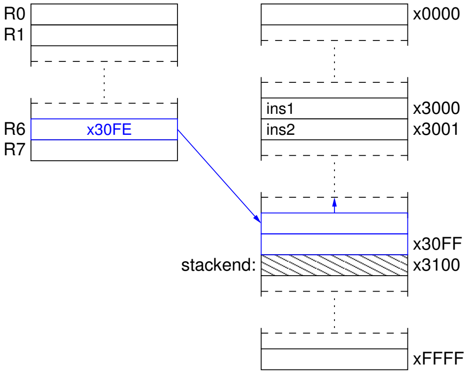
`http://highered.mcgraw-hill.com/sites/0072467509/`

A stack, why ?

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

LC3 stack emulation - from the archi course

- R6 is initialised to a "end of stack" address (stackend)
- R6 always stores the address of the last value stored in the stack.
- The stack grows in the dir. of **decreasing addresses!**



LC3 stack emulation: concretely 1/2

```
          .ORIG x3000
; Main program
main:     LD R6,spinit  ; stack pointer init
          ...
          HALT

; Stack management
spinit:   .FILL stackend
          .BLKW #15     ; this stack is rather small
stackend: .BLKW #1      ; end of stack address
          .END
```

LC3 stack emulation: concretely 2/2

Push the content of Ri:

```
ADD R6,R6,-1   ; move head of stack
STR Ri,R6,0    ; store the value
```

Pop the content of the stack in Ri:

```
LDR Ri,R6,0    ; pop the value
ADD R6,R6,1    ; head of stack restauration
```

Outline

A first example (1/4)

How do we translate:

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

- Compute 4
- Store somewhere `place0`, then link $x \mapsto place0$
- Compute $12 + x$: 12 in `place1`, x in `place2`, then addition, store in `place3`, then link $x \mapsto place3$

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

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

"Compute 4 and store in x":

```
AND temp1 temp1 0
ADD temp1 temp1 4
```

And $x \mapsto temp1$.

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

A first example: from 3@ code to valid LC-3 (3/4)

But this is not valid LC3 code !
We should use registers, but as they are only 8, we use the stack to store temporaries. Here **store R1 on the stack!**

```
AND R1 R1 0
ADD R1 R1 4
ADD R6 R6 -1 #here also store x -> R6 somewhere
STR R1 R6 0  #now R1 can be recycled
```

A first example: prelude/postlude 4/4

The rest of the code generation:

```
.ORIG X3000
LEA R6 data
[...]
stop: BR stop
data: .BLKW 42
.END
```

- This is valid LC-3 code that can be assembled and executed in Pennsim.

Objective of the rest of the course

3-address LC-3 **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.

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.

Outline

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
	$\mid$	$\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$	test
	$\mid$	$\texttt{while}\ b\ \texttt{do}\ S\ \texttt{done}$	loop

Code generation for expressions, example

e ::= c (cte expr)	`#not valid if c is too big` `dr <-newTemp()` `code.add(InstructionAND(dr, dr, 0))` `code.add(InstructionADD(dr, dr, c))` `return dr`

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

Code generation for a boolean expression, example

e $::= e_1 < e_2$	<pre>dr <-newTemp() t1 <- GenCodeExpr (e1-e2) #last write in register (lfalse,lend) <- newLabels() code.add(InstructionBRzp(lfalse)) #if =0 or >0 jump! code.add(InstructionAND(dr, dr, 0)) code.add(InstructionADD(dr, dr, 1)) #dr <- true code.add(InstructionBR(lend)) code.addLabel(lfalse) code.add(InstructionAND(dr, dr, 0)) #dr <- false code.addLabel(lend) return dr</pre>

- integer value 0 or 1.

Code generation for commands, example

if b then $S1$ else $S2$	
	``dr <-GenCodeExpr(b) #dr is the last written register``` ```lfalse,lendif=newLabels()``` ```code.add(InstructionBRz(lfalse) #if 0 jump to execute``` ```GenCodeSmt(S1) #else (execute S1``` ```code.add(InstructionBR(lendif)) #and jump to end)``` ```code.addLabel(lfalse)``` ```GenCodeSmt(S2)``` ```code.addLabel(lendif)```

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)