

Compilation (#6): Syntax-Directed Code Generation

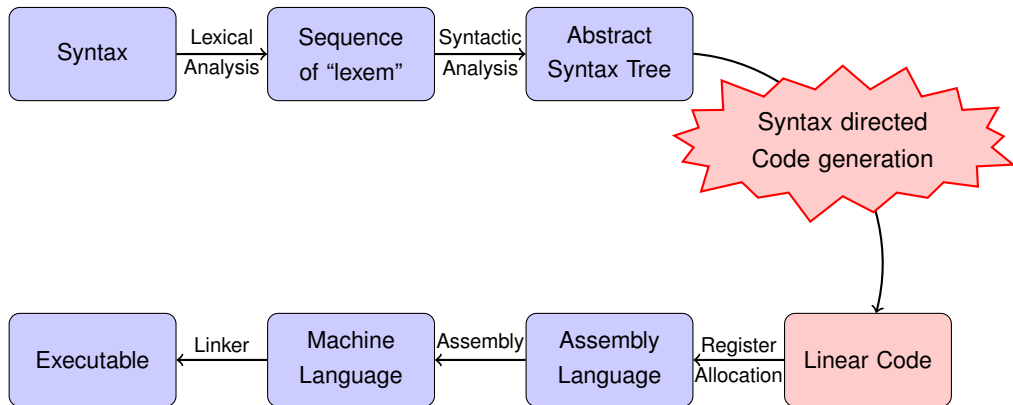
C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023



Big Picture



Nice picture from G. Radanne

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: everything will be stored on the stack.

The Target Machine: RISCV.

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

Code Generation vs Memory/Register Allocation

- Code generation in two steps:
 - 1 Generate instructions without deciding where data is stored (put everything in 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

Give the expected code for $y=12+x$?

Objective

3-address RISC-V 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-address code generation**

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

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 RISC-V ones.

Abstract Syntax

Expressions:

$e ::= c$	constant
x	variable
$e + e$	addition
$e \text{ or } e$	boolean or
$e < e$	less than
...	

Statements:

$S ::= x := expr$	assign
$skip$	do nothing
$S_1; S_2$	sequence
$\text{if } b \text{ then } S_1 \text{ else } S_2$	test
$\text{while } b \text{ do } S \text{ done}$	loop

Code generation for expressions, example

$e ::= c \text{ (cte expr)}$	<pre>dr <- new_tmp() code.add(li(dr, c)) return dr</pre>
------------------------------	---

- this rule gives a way to generate code for any constant. **Do it for constant 4!**

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(li(dr, 0))
#if t1>=t2 jump to endrel
code.add(condjump(endrel, t1, ">=" , t2))
code.add(li(dr, 1))
code.addLabel(endrel)
return dr
```

- the destination registers contains after execution an integer value 0 or 1. Try **(e==x<4)**

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

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(condjump(lelse, t1, "=", 0))
GenCodeSmt(S1) #then
code.add(jump(lendif))
code.addLabel(lelse)
GenCodeSmt(S2) #else
code.addLabel(lendif)
```

Do it for `if x<4 then y=7 else ...` - next slide

Example for tests.

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

jump lendif1 // don't forget this one!

lelse1:

code for **else** branch

lendif1:

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

From 3@ code to valid RISC-V

3@code is not valid RISC-V code!

We explore 3 allocation algorithms:

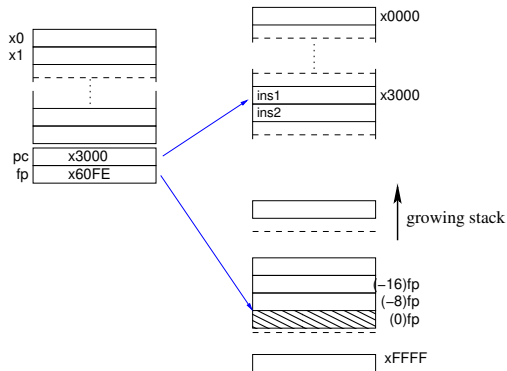
- All in registers $temp_i \rightarrow register$ \leftarrow very, very naive
- All in memory $temp_i \rightarrow memory$ \leftarrow very naive
- Something in between \leftarrow yes, we'll do smart stuff too :-)

A stack, why?

- Store constants, strings, ...
- Provide an easy way to communicate arguments values (see later)
- Give place to store intermediate values (e.g. $2*3$ in $x = 1 + 2 * 3$)

Stack with RISC-V

- 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)
```

```
# To generate from Python:
```

```
# sd(s3, Offset(FP, -offset*8))
```

```
# "write the value of s3 at address fp - offset*8"
```

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

Code Generation

Input: a MiniC file:

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

Output: a RISC-V file:

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

Steps of lab 4

- 3-address code generation according to the code generation rules.
- Simple register/memory allocation (naive, all-in-mem) + pretty print.

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

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 : 3address code generation

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