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

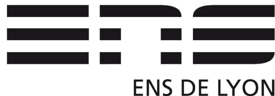
Laure Gonnord

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

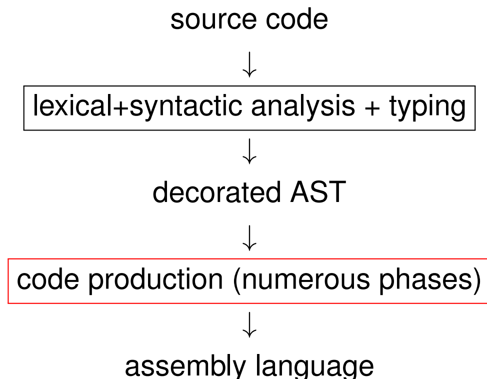
Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

2018-2019



Big picture



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 : TARGET18 (course #1)

1. 3-address syntax-directed Code Generation
   - Rules
2. Memory allocation
3. Code Generation Lab
4. Toward a more efficient Code Generation

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 `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 `newtmp()`

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

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

```
leti temp2 4
let temp0 temp2 ; this is copy for 2018
```

Objective

3-address TARGET18 **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**

1. 3-address syntax-directed Code Generation
   - Rules
2. Memory allocation
3. Code Generation Lab
4. Toward a more efficient 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 TARGET18 ones.

Abstract Syntax

Expressions :

$$
\begin{array}{rcll}
e & ::= & c & \text{constant} \\
 & | & x & \text{variable} \\
 & | & e + e & \text{addition} \\
 & | & e \text{ or } e & \text{boolean or} \\
 & | & e < e & \text{less than} \\
 & | & \ldots &
\end{array}
$$

and statements :

$$
\begin{array}{rcll}
S(Smt) & ::= & x := expr & \text{assign} \\
 & | & skip & \text{do nothing} \\
 & | & S_1; S_2 & \text{sequence} \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\
 & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{loop}
\end{array}
$$

Code generation for expressions, example

e ::= c (cte expr)	`dr <-newTemp()`
	`code.add(InstructionLETI(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 $S2$	<pre>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)</pre>

Exercise time !

1. 3-address syntax-directed Code Generation
2. Memory allocation
3. Code Generation Lab
4. Toward a more efficient Code Generation

A first example : from 3@ code to valid TARGET18 (3/5)

3@code is not valid TARGET18 code !

3 “kinds of allocation” :

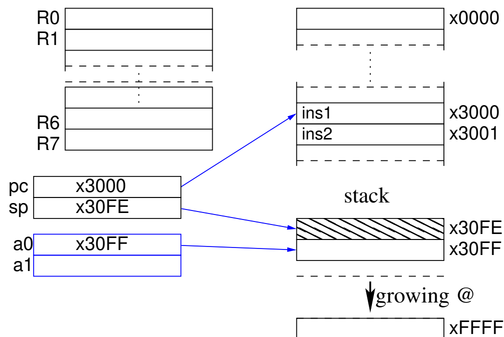
- All in registers (but ?) $place_i \rightarrow register$
- All in memory (here !) $place_i \rightarrow memory$
- Something in the middle (later !)

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 TARGET18

- There is a special register sp.
- Store and loads with instructions `setctr` and `getctr`.
- The stack grows in the dir. of **increasing addresses !**.



Nice picture by N. Louvet - adapted in 2018

An example

Here **Store (the content of)** r_1 **on the stack !**

```
GETCTR sp r0
ADD r0 r0 <valueofoffset*16>
SETCTR a0 r0
WRITE a0 16 r1
```

1. 3-address syntax-directed Code Generation
2. Memory allocation
3. Code Generation Lab
4. Toward a more efficient Code Generation

Code Generation

Input : a mu file :

```
var n:int;
n=6;
```

Output : a TARGET18 file :

```
1       ;; (stat (assignment n = (expr (atom 6)) ;))
        LETI r1 6
        LET r2 r1
```

Code Generation, first step

- 3-address code generation according to the code generation rules of the course :

```
t1 <- GenCodeExpr(e_1)
t2 <- GenCodeExpr(e_2)
dr <- newTemp()
code.add(InstructionADD(dr, t1, t2))
return dr
```

- **TODO 1 : implement them** :

```
tmpl = self.visit(ctx.expr(0))
tmpr = self.visit(ctx.expr(1))
dr = self._prog.new_tmp()
if ctx.myop.type == MuParser.PLUS:
    self._prog.addInstructionADD(dr, tmpl, tmpr)
```

- this is not executable code

Result after first step

The previous step uses instructions of an API like :

```
self._prog.addInstructionADD(dr, tmpl, tmpr)
```

whose side effect is to construct a TARGET18 prog as a list of 3 adresses instructions with temporaries (virtual registers, from the class `VirtualRegister`.

This list can be dumped (with `printCode` in the API) into a .s file :

```
;; (stat (assignment n = (expr (atom 6)) ;))
LETI temp_1 6
LET temp_2 temp_1
```

We cannot test !

Code Generation, second step

The allocation process :

- takes as input the preceding result
- modifies the list of instructions with temporaries into list of instructions with physical registers or accesses to memory.
- a trivial allocator is given.

TODO 2 : all in memory allocation

Code Infrastructure

```
...code/Mu-codegen$ ls
Allocations.py   Main.py    MyMuCodeGen3AVisitor.py
APICodeTARGET18.py  Makefile  test_codegen.py
ExpandJump.py    Mu.g4      Instruction3A.py Operands.py
```

- The Mu grammar in `Mu.g4`,
- A Makefile, Main as usual.
- Library and unit tests in `test*.py`.
- API for generating TARGET18 code : `APICodeTARGET18`, `Instruction3A.py`, `Operands.py`
- `ExpandJump.py` is used to dump the conditional jump instruction into legal TARGET18 code.
- `Allocations.py` : allocators for TARGET18 code.
- **TODO : edit and fill `MyMuCodeGen3AVisitor.py` mainly**. You may have other changes to make in other files (Allocation).

TARGET18 API

In this API (APICodeTARGET18, Instruction3A, Operands) :

- A class for a program `TARGET18Prog`. The program contains a list of instructions, methods to add instructions, to increment temporary numbers, . . .
- Classes for instructions : `Instruction`, `Instru3A`, `Label` . . .
- A 3 address instruction contains arguments that can be `Immediate`, `VirtualRegister`, `Register`, or a `Condition` in the special case of the condjump. . . .
- the `CondJump` instruction (label,dr1,cond,dr2) has the meaning :

  `if (dr1 cond dr2) jump to label.`

1. 3-address syntax-directed Code Generation
2. Memory allocation
3. Code Generation Lab
4. Toward a more efficient Code Generation

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)