Lab 4
Syntax-Directed Code Generation

Objective

- Generate 3-address code for the Mu language.

First download the archive from the course website.

During the previous lab, you have written your own evaluator of the Mu language. In this lab the objective is to generate *3-address LC3* code [1]. Due to lack of time, we will not implement the "all-in-the-stack" temporary allocation seen in the course.

WARNING This code generation (code, Readme, testfiles, Makefile and scripts.) is evaluated (personal work only). Deadline Tuesday, November 15th, 2016, 23h59, on TOMUSS.

An important remark on operator types Until now, when we had to match the type of an operator in a given visitor rule, we wrote a visitor rule that returns the chain that was returned by the lexing phase:

Listing 4.1: First version of the ANTLR4 grammar

```
expr:[..]
 | expr oplus expr          #additiveExpr

oplus: PLUS #opPlus
    | MINUS #opPlus
    ;
```

Listing 4.2: First version of the Visitor

```
def visitOpPlus(self,ctx):
    return ctx.getText();  #returns the associated chain

def visitAdditiveExpr(self,ctx):
     op = self.visit(ctx.oplus())
     if op == '+': [...]
```

In the grammar we give you in this lab, we modified a bit the definition inside the ANTLR4 grammar description file, and thus we obtain a more elegant version:

Listing 4.3: Second version of the ANTLR4 grammar

```
expr:[..]
 | expr myop=(PLUS|MINUS) expr          #additiveExpr
```

Listing 4.4: Second version of the Visitor

```
def visitAdditiveExpr(self,ctx):
     if (ctx.myop.type==MuParser.PLUS):
```

4.1 Three-address code

In this section you have to implement the course rules (Figures 4.2 and 4.3) in order to produce LC-3 code with temporaries, whose syntax is recalled in Figure 4.1.

[1] Namely, a code with LC3 instructions (each of them have at most 3 operands), with pseudo-registers, which still have to be allocated in registers or global memory.

EXERCISE #1 ► **Manual translation**

Give a (LC-3) three-address code for the following Mu program:

```
n=1;
a=3;
while (n<a) do
 n= n+1;
done
```

syntaxe	**action**	**NZP**	**codage**															
			opcode				arguments											
			F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
NOT DR,SR	DR ← not SR	*	1	0	0	1	DR			SR			1	1	1	1	1	1
ADD DR,SR1,SR2	DR ← SR1 + SR2	*	0	0	0	1	DR			SR1			0	0	0	SR2		
ADD DR,SR1,Imm5	DR ← SR1 + SEXT(Imm5)	*	0	0	0	1	DR			SR1			1	Imm5				
AND DR,SR1,SR2	DR ← SR1 and SR2	*	0	1	0	1	DR			SR1			0	0	0	SR2		
AND DR,SR1,Imm5	DR ← SR1 and SEXT(Imm5)	*	0	1	0	1	DR			SR1			1	Imm5				
LEA DR,label	DR ← PC + SEXT(PCoffset9)	*	1	1	1	0	DR			PCoffset9								
LD DR,label	DR ← mem[PC + SEXT(PCoffset9)]	*	0	0	1	0	DR			PCoffset9								
ST SR,label	mem[PC + SEXT(PCoffset9)] ← SR		0	0	1	1	SR			PCoffset9								
LDR DR,BaseR,Offset6	DR ← mem[BaseR + SEXT(Offset6)]	*	0	1	1	0	DR			BaseR			Offset6					
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] ← SR		0	1	1	1	SR			BaseR			Offset6					
BR[n][z][p] label	Si (cond) PC ← PC + SEXT(PCoffset9)		0	0	0	0	n		z	p	PCoffset9							
NOP	No Operation		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RET (JMP R7)	PC ← R7		1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
JSR label	R7 ← PC; PC ← PC + SEXT(PCoffset11)		0	1	0	0	1	PCoffset11										

Figure 4.1: LC3 simplified instruction set

EXERCISE #2 ► **3-address code generation**

In the archive, we provide you a main and an incomplete `MyMuCodeGenVisitor.py`. To compile and run it, type:

```
make FOO=ex/expr.mu
```

Observe the generated code in `ex/expr.asm`[2]. You now have to implement the 3-address code generation rules seen in the course. Code and test incrementally:

- numerical expressions without variables (constants are supposed to be small). We provide you a test file called `expr.mu`. **The arithmetic operator `^`, as well as `*` and `/`, are not supposed to be implemented, but an exception should be launched.**
- then assignments and expressions with variables.
- then boolean expressions: compute 1 or 0 in the destination register;
- finally if, while.

You are allowed to skip proper treatments of strings and the `log` operator. To test your 3-address code generation, we provide you a script that transform your temporaries into actual registers (if there is less than 8 temporaries in your asm file: test with tiny examples).

```
./test3a.sh myfile.asm
```

Then you will have to use Pennsim:

```
java -jar Pennsim
(as myfile.asm; open .obj file; set PC to x3000;...)
```

[2] We generated LC3 comments with the statement currently seen for debug.

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

e ::= x	`#get the place associated to x.`
	`regval=getTemp(x)`
	`return regval`

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

e ::= e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionNOT(dr, t2))`
	`code.add(InstructionADD(dr, dr, 1))`
	`code.add(InstructionADD(dr, dr, t1))`
	`return dr`

e ::= true	`dr <-newTemp()`
	`code.add(InstructionAND(dr, dr, 0))`
	`code.add(InstructionADD(dr, dr, 1))`
	`return dr`

e ::= $e_1 < e_2$	`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`

Figure 4.2: Code generation for expressions `GenCodeExpr`

x := e	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`if x has a location loc:`
	`  code.add(instructionADD(loc,dr,0))`
	`else:`
	`  storeLocation(x,dr)`

S1; S2	`#concat codes`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

if *b* then *S*1 else *S*2	`dr <-GenCodeExpr(b)  #dr is the last written register`
	`lfalse,lendif=newLabels()`
	`code.add(InstructionBRz(lfalse) #if 0 jump to execute S2`
	`GenCodeSmt(S1)  #else (execute S1`
	`code.add(InstructionBR(lendif)) #and jump to end)`
	`code.addLabel(lfalse)`
	`GenCodeSmt(S2)`
	`code.addLabel(lendif)`

while *b* do *S* done	`ltest,lendwhile=newLabels()`
	`code.addLabel(ltest)`
	`dr <-GenCodeExpr(b)  #dr is the last written register`
	`code.add(InstructionBRz(lendwhile) #if 0 jump to end`
	`GenCodeSmt(S)  #else (execute S`
	`code.add(InstructionBR(ltest))  #and jump to the test)`
	`code.addLabel(lendwhile)`

Figure 4.3: Code generation for Statements GenCodeSmt