Advanced Compiler Construction – Winter 2020 – Prof. Dr. Michael Franz

Course Project

The project assignment for this course will illustrate various aspects of optimizing compilers by way of a scaled-down example. You will be asked to construct an optimizing compiler for the simple programming language. The syntax of PL241 is given below; the semantics of the various syntactical constructs are hopefully more or less obvious. PL241 has integers, arrays, functions (which return a scalar result) and procedures (which don't return anything). There are three predefined procedures *InputNum*, *OutputNum*, and *OutputNewLine*. All arguments to functions and procedures are scalar (arrays cannot be passed as parameters).

First Step

You will build a simple recursive-descent parser that generates an intermediate representation appropriate for subsequent optimizations. The intermediate representation will be a dynamic data structure in memory and needs to provide control flow and dominator information for basic blocks. Instructions should be represented in *Static Single Assignment* form.

The operations encoded in instruction nodes consist of an operator and up to two operands. The following operators are available (the meaning of *Phi* and the use of operator *adda* will be explained in the lecture):

neg x	unary minus
add x y	addition
sub x y	subtraction
mul x y	multiplication
div x y	division
cmp x y	comparison
adda x y	add two addresses *x* und *y (used only with arrays)*
load y	load from memory address y
store y x	store *y* to memory address *x*
move y x	assign *x := y*
phi x1 x2	***compute Phi(x1, x2)***
end	end of program
bra y	branch to *y*
bne x y	branch to *y* on *x* not equal
beq x y	branch to *y* on *x* equal
ble x y	branch to *y* on *x* less or equal
blt x y	branch to *y* on *x* less
bge x y	branch to *y* on *x* greater or equal
bgt x y	branch to *y* on *x* greater

In order to model the built-in input and output routines, we add three more operations:

read	read
write x	write
writeNL	writeNewLine

When compiling a program that contains functions/procedures in addition to a main program body, each of these units is modeled as a separate control flow graph. You will need to introduce an additional type of node in your IR that represents such calls, linking the call location to the function/procedure being called and any actual parameters passed in the call to the formal parameters specified in the called function/procedure.

The intermediate representation generated by your compiler should be visualized using either a compiler-oriented tool such as *VCG/aiSee* (Visualization of Compiler Graphs) or a generic graph visualization tool using GraphML or similar. It is almost impossible to debug the kind of complex dynamic data structures that are used in optimizing compilers without such tools. Your output should consist of the CFG visually depicted by basic blocks (boxes) connected by control flow (lines), and within each basic block, the instruction list should be shown in a format similar to what we are using in class. Additionally, you should visualize the dominator tree.

Second Step

After you are confident that your conversion to SSA works correctly, extend your compiler by implementing *common subexpression elimination* and *copy propagation* on the control flow graph. In order to make this process visible to the user, introduce a trace mode that produces an elimination protocol. Display the resulting program after elimination in SSA form (without any *MOVE* instructions remaining) using graph visualization. Perform experiments to test your implementation for correctness. Pay attention especially to redundant array loads that can be eliminated safely, since loads going to memory are among the most expensive operations on almost any platform.

Third Step

Implement a global register allocator for your compiler. For this purpose, track the live ranges of all the individual values generated by the program being compiled, and build an interference graph. Color the resulting graph, assuming that the target machine has 6 general-purpose data registers. If more registers are required, map the values that cannot be accommodated onto virtual registers in memory. Eliminate all Phi-Instructions, inserting move-instructions wherever necessary. Display the final result using graph visualization, and perform experiments to test your implementation.

Fourth Step (required only for groups of two)

Write a code generator for the source language that emits optimized (CSE, copy propagation, register allocation) *native* programs in the *native load format of a real platform*. You may choose your target platform from x86/Windows, x86/Linux, or you may use the DLX processor simulator.

Optional Final Step

Perform instruction scheduling between the register allocation and code generation stages of your compiler. Try to find a scheduling heuristic that improves performance over non-scheduled code.

EBNF for PL241

letter = “a” | “b” | … | “z”.
digit = “0” | “1” | … | “9”.
relOp = “==“ | “!=“ | “<“ | “<=“ | “>“ | “>=“.

ident = letter {letter | digit}.
number = digit {digit}.

designator = ident{ "[" expression "]" }.
factor = designator | number | “**(**“ expression “**)**” | funcCall .
term = factor { (“*” | “/”) factor}.
expression = term {(“+” | “-”) term}.
relation = expression relOp expression .

assignment = “**let**” designator “**<-**” expression.
funcCall = “**call**” ident [“**(**“ [expression { “**,**” expression }] “**)**”].
ifStatement = “**if**” relation “**then**” statSequence [“**else**” statSequence] “**fi**”.
whileStatement = “**while**” relation “**do**” StatSequence “**od**”.
returnStatement = “**return**” [expression] .

statement = assignment | funcCall | ifStatement | whileStatement | returnStatement.
statSequence = statement { “**;**” statement }.

typeDecl = “**var**” | “**array**” “[“ number “]” { “[“ number “]” }.
varDecl = typeDecl indent { “**,**” ident } “**;**” .
funcDecl = (“**function**” | “**procedure**”) ident [formalParam] “**;**” funcBody “**;**” .
formalParam = “**(**“ [ident { “**,**” ident }] “**)**” .
funcBody = { varDecl } “**{**” [statSequence] “**}**”.

computation = “**main**” { varDecl } { funcDecl } “**{**” statSequence “**}**” “**.**” .

Predefined Function (functions return a scalar result)

InputNum()	read a number from the standard input

Predefined Procedure (procedures don’t return anything)

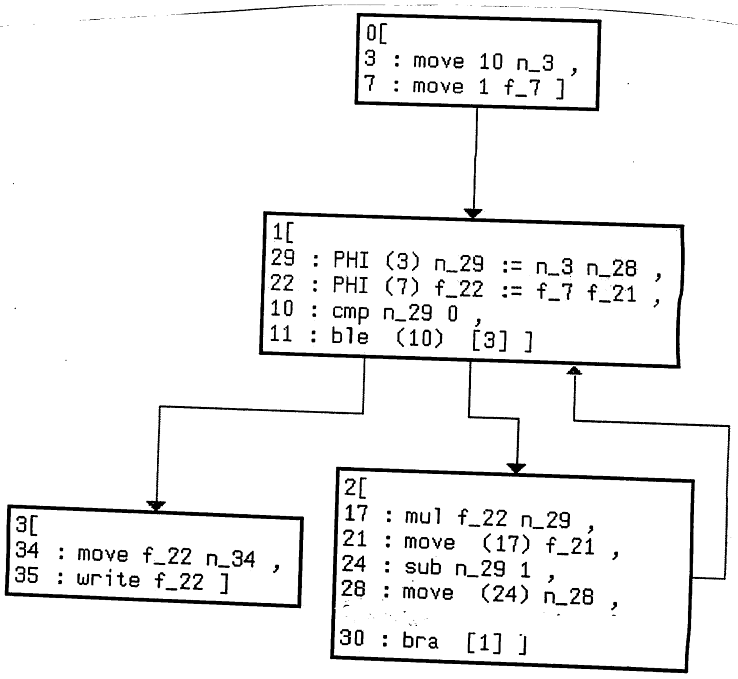
OutputNum(x)	write a number to the standard output
OutputNewLine()	write a carriage return to the standard output

```
main
  var f, g;
{
  let n <- 10;
  let f <- 1;

  while n > 0 do
    let f <- f * n;
    let n <- n - 1
  od;

  let n <- f;
  OutputNum(f)
}.
```

Source

```
graph: { title: "Control Flow Graph"
layoutalgorithm: dfs
manhattan_edges: yes
smanhattan_edges: yes
node: {
title: "0"
label: "0[
3 : move 10 n_3 ,
7 : move 1 f_7 ]"
}
edge: { sourcename: "0"
targetname: "1"
color: blue
}
node: {
title: "1"
label: "1[
29 : PHI (3) n_29 := n_3 n_28 ,
22 : PHI (7) f_22 := f_7 f_21 ,
10 : cmp n_29 0 ,
11 : ble  (10)  [3] ]"
}
edge: { sourcename: "1"
targetname: "2"
color: blue
}
node: {
title: "2"
label: "2[
17 : mul f_22 n_29 ,
21 : move  (17) f_21 ,
24 : sub n_29 1 ,
28 : move  (24) n_28 ,

30 : bra  [1] ]"
}
edge: { sourcename: "2"
targetname: "1"
color: red
}
edge: { sourcename: "1"
targetname: "3"
color: red
}
node: {
title: "3"
label: "3[
34 : move f_22 n_34 ,
35 : write f_22 ]"
}
}
```

vcg



```
0[
3 : move 10 n_3 ,
7 : move 1 f_7 ]
```

```
1[
29 : PHI (3) n_29 := n_3 n_28 ,
22 : PHI (7) f_22 := f_7 f_21 ,
10 : cmp n_29 0 ,
11 : ble  (10)  [3] ]
```

```
3[
34 : move f_22 n_34 ,
35 : write f_22 ]
```

```
2[
17 : mul f_22 n_29 ,
21 : move  (17) f_21 ,
24 : sub n_29 1 ,
28 : move  (24) n_28 ,

30 : bra  [1] ]
```