Compilation (CS444)

Exercise book

Contents

TD 1

Introduction and lexical analysis

1.1 Introduction

EXERCISE #1 ▶ **Java!**

Are these Java programs correct? If it is not the case, provide the name of the compilation phase that detects the error.

```
//Program 1
class A {
  int x?;
  public static void main(){
  }
}

//Program 2
class A {
  int x;
  public static void main(){
    String s = "hello;
    System.out.println(s);
  }
}

//Program 3
class A {
  public static void main(){
    int y=3;
    System.out.println(x+y);
  }
}

//Program 4
class A {
  static void m(int x){
    System.out.println(x+1);
  }
  public static void main(){
    int y=3;
    m(y=y+4);
    m(y==y+4);
  }
}

//Program 5
class A {
  static int m(int x,
               boolean b,
               double f){
    return b?x+(int)f:3;
  }
  public static void main(){
    m(4,3.14);
  }
}

//Program 6
class A {
  static int m(int x,
               boolean b,
               double f){
    return b?x+(int)f:3;
  }
  public static void main(){
    m(4.0,true,3.14);
  }
}

//Program 7
class A {
  static int m(int x){
    final int y=x+1;
    y++;
    return y;
  }
  public static void main(){
    m(4);
  }
}

//Program 8
class A {
  public static void main(
    String args[]){
    int t[]=new int[2];
    t[2]=42;
    System.out.println(t[2]);
  }
}
```

EXERCISE #2 ▶ **C: let us propagate information!**
Consider the program `foo.c` :

```
int foo (unsigned int x)
{
  if (x < 0)
    return 0;
  else
    return 1;
}
```

Give an informative information that the compiler can detect. In which phase? How can the compiler make usage of it?

EXERCISE #3 ▶ **Compilation with gcc**

Here are two simple files `expr.c` and `facti.c`. Running the command `gcc -fdump-tree-gimple -c` *file*`.c` produces (in addtion to `file.o`) the files *file*`.c.004t.gimple` shown below. What do you think these files are?

```
int expr(int a, int b, int c)
{
  return(a+b*c+b);
}
```

```
int fact(int n)
{
  int i,f=1;

  for (i=2; i<=n; i++)
    f *= n;
  return f;
}
```

```
expr (int a, int b, int c)
{
  int D.1761;
  int D.1762;
  int D.1763;

  D.1762 = b * c;
  D.1763 = D.1762 + a;
  D.1761 = D.1763 + b;
  return D.1761;
}
```

```
fact (int n)
{
  int D.1764;
  int i;
  int f;

  f = 1;
  i = 2;
  goto <D.1761>;
  <D.1760>:
  f = f * n;
  i = i + 1;
  <D.1761>:
  if (i <= n) goto <D.1760>; else goto <D
      .1762>;
  <D.1762>:
  D.1764 = f;
  return D.1764;
}
```

EXERCISE #4 ▶ **Yet another C Compiler**

```
#include <stdio.h>
```

```c
int main(){

  int mytab[12];

  for (int i=0;i<=12;i++)
    mytab[i]=42+i;

  printf("%d\n",mytab[12]);

  return 0;
}
```

Compile with `gcc -Wall`, observe the absence of warning. A compilation with `clang` would give:

```
gccclang.c:12:17: warning: array index 12 is past the end of the array (which
      contains 12 elements) [-Warray-bounds]
  printf("%d\n",mytab[12]);
                ^     ~~
gccclang.c:7:3: note: array 'mytab' declared here
  int mytab[12];
  ^
1 warning generated.
```

1.2 Lexical Analysis

EXERCISE #5 ▶ **Regular expressions for lexing**
Use the ANTLR syntax to define regular expressions as models for:

1. Identifiers: any sequence of letters, digits and _ that does not begin by a digit nor _.
2. Floats like -3.96 (the sign is optional, but the dot is not).
3. Numbers in scientific notation like $-1.6E-12$.
4. The keywords `begin`, `end` and `while`

For convenience, tools such as ANTLR provide much more regexp constructs than what is strictly necessary. See the list of ANTLR's main regexp constructs below. ANTLR also allows to define *fragments* (names for regexps that can then be used in other regexps).

EXERCISE #6 ▶ **Shadok numbers (Bonus)**
Write a ANTLR lexical file that reads and interprets shadok numerals. This is a base 4 system with digits ga bu zo meu. Hence zozo is $2\times4^1+2\times4^0=10$, bubuga is $1\times4^2+1\times4^1+0=20$, zozozo is ... 42, and bugagagagaga is 1024.

Use the skeleton provided on chamilo.

```
(...)      grouping
|          alternative, e.g. (a|b)
's'        char or string literal.  '\n' for newline.
.          any character
a .. b     range, e.g. '0'..'9'
+          1 or more, e.g. ('0' .. '9')+
*          0 or more
?          optional
[...]      choice between characters, e.g. [abc]
[a-z]      choice on interval, e.g. [0-9a-z]
~[...]     match not, e.g. ~[abc]
// ...     single-line comment
/* ... */  multi-line comment
```

Figure 1.1: ANTLR's main regexp constructs

TD 2

Grammars and Parsing (1)

2.1 Grammars

EXERCISE #1 ▶ **Arithmetic expressions**

Let us consider the following grammar for expressions:

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \\ T & \rightarrow & T * F \\ T & \rightarrow & F \\ F & \rightarrow & (E) \\ F & \rightarrow & nb \end{array}$$

Give a derivation and a parse tree for the chains:

- `nb + nb * nb`
- `nb * nb + nb`

What is the main property of this grammar?

EXERCISE #2 ▶ **Associativity**

Propose a grammar generating expressions of the form: $n \circ n \circ n \cdots \circ n$, with $\circ$ a binary operator that's:

1. left-associative
2. right-associative

EXERCISE #3 ▶ **Ambiguities: the case of if/then/else**

Let us consider the following grammar:

$$\begin{array}{lcl} instr & \rightarrow & \textbf{if } expr \textbf{ then } instr \\ instr & \rightarrow & \textbf{if } expr \textbf{ then } instr \textbf{ else } instr \\ instr & \rightarrow & \textbf{other} \\ expr & \rightarrow & \ldots \end{array}$$

- Show that the grammar is ambiguous.
- What will be the output of this program?

```c
#include <stdio.h>
int main()
{
  if (1) if (0) printf("Hi there\n");
  else printf("hey!\n");
}
```

2.2 Top-down parsing

EXERCISE #4 ► **From the course**

Finish the course exercise, chapter 2, page 38.

EXERCISE #5 ► **Prefix expressions**

Here's a grammar for prefix expressions. Uppercase letters are non-terminals, n is a digit terminal.

$$\begin{array}{lcl} E & \rightarrow & OEE|A \\ O & \rightarrow & +|* \\ A & \rightarrow & n|(E) \end{array}$$

1. Compute the *First* and *Follow* sets for the non terminals.
2. Is the grammar $LL(1)$? Otherwise, is there an equivalent grammar that's $LL(1)$?
3. Write a predictive analyzer using mutually recursive functions.
4. Show the execution steps of the program when analyzing the chain $+(*21)(+(+11)2)$, as shown below:

```
parse              ; call parse
  E                ;    call E
    O              ;      call O
      +            ;         call eat_tk('+'), then return
    E              ;      call E
    ...
```

5. Similarly, show the execution steps for the (rejected) chains $(+1(*2))$ and $+12($.

EXERCISE #6 ► **Navigation language**

Here's a grammar G for a simplified language of navigation instructions, as could be output by a route computation system.

G= { V_T, V_n , P, start } with V_T = { GO, TL, TR, SIGN, END } and V_N = { start, route, inst, sign_opt, turn }

Terminal symbols are GO (go forward), TL (turn left), TR (turn right), SIGN (sign here) and END (finish), P contains:

$$\begin{array}{rcl} \text{start} & \rightarrow & \text{route END} \\ \text{route} & \rightarrow & \text{inst} \mid \text{route inst} \\ \text{inst} & \rightarrow & \text{GO} \mid \text{sign\_opt turn} \\ \text{turn} & \rightarrow & \text{TL} \mid \text{TR} \\ \text{sign\_opt} & \rightarrow & \varepsilon \mid \text{SIGN} \end{array}$$

1. Show that G is not $LL(1)$.
2. Try to modify G into G' so that it is $LL(1)$.
3. Compute the *first* and *follow* sets for G'.
4. Produce the analysis table of the iterative (stack based) predictive analyser for G'.
5. Simulate the execution of this analyzer on the chain GO GO TL GO SIGN TR GO END (be meticulous!).
6. Draw the corresponding derivation tree.

TD 3
Grammars and Parsing (2)

3.1 Bottom-up parsing

EXERCISE #1 ▶ **Shift-reduce analysis**

Let's use the (very simple) grammar:

$$S \rightarrow 0\,S\,1 \mid 0\,1$$

1. Perform the shift-reduce analysis of the chain `000111`.
2. Show the corresponding rightmost derivation, highlighting the *handle* at each step.

EXERCISE #2 ▶ **Parsing lists**

Here are two "list" grammars Gl and Gr defined by:

```
Gl : S -> S x | x
```

```
Gr : S -> x S | x
```

1. Draw the derivation tree for chain xxxx for both grammars.
2. Show the shift-reduce analysis for this chain for both grammars.

EXERCISE #3 ▶ **Shift-reduce and ambiguous grammars**

Let's consider the following ambiguous grammar for arithmetic expressions:
`E → E-E | E+E | (E) | E*E | E/E | n`

- What are the two possible derivation trees for the chain $1-2*3$?
- Perform the shift-reduce analysis corresponding to the two trees.
- We know we can't accept ambiguities. Define a (out of the grammar) rule to resolve ambiguities. How can this rule be implemented in a shift-reduce analyzer?
- Same questions for the chain $1-2-5$.
- Is there an operator (eg in C) that is right-associative?
- Let's get back to exercise 3 in TD2. The C language defines a rule to resolve the ambiguity. How to implement this rule in a shift-reduce analyser?

EXERCISE #4 ▶ **Simple SLR(1) analysis**

Let us consider the simple grammar defining the *clock noise* language:

$$\begin{array}{lcl} S & \rightarrow & N \\ N & \rightarrow & N \text{ tick tock} \\ N & \rightarrow & \text{tick tock} \end{array}$$

1. Build the LR(0) automaton associated to this grammar.

2. Build the SLR(1) table.
3. Use the table to perform the analysis of the chain: tick tock tick tock.
4. Use the table to perform the analysis of the chain: tick tick tock tock.

EXERCISE #5 ► **More complex SLR(1) analysis**

Let us consider grammar G:

$$\begin{array}{lcl} S & \rightarrow & T \\ T & \rightarrow & XB \\ X & \rightarrow & aXb \\ X & \rightarrow & \varepsilon \\ B & \rightarrow & bB \\ B & \rightarrow & b \end{array}$$

1. Build the LR(0) automaton associated to G.
2. Build the SLR(1) table. Is this grammar SLR(1) ?
3. What language is generated by this grammar?
4. Use the table to perform the analysis of abb.

TD 4

Grammars: syntax-directed computations

4.1 Derivation trees and attributions

EXERCISE #1 ▶ **Check coherence of XML Files**

We give the following grammar:

$$\begin{array}{lcl} L & \rightarrow & EL|\varepsilon \\ E & \rightarrow & ALB|\text{ ident} \\ A & \rightarrow & <\text{ ident }> \\ B & \rightarrow & </\text{ ident }> \end{array}$$

1. Give the derivation tree for the chain `<html><head>toto</head>titi</foo>`.
2. Attribute this grammar to verify that opening and closing tags refer to the same identifiers.

EXERCISE #2 ▶ **Attributions to further propagate info: declarations**

Let us consider a "Pascal-like" grammar for variable declarations:

$$\begin{array}{lcl} D & \rightarrow & L : T \\ T & \rightarrow & \textbf{integer}|\textbf{char} \\ L & \rightarrow & L, \textbf{id}|\textbf{id} \end{array}$$

1. Give a grammar attribution to compute and store the type of each variable in the symbol table, by calling the void returning function `add_type(num, type)`.
2. Construct the parse tree and the dependency graph for the chain `i,j,k:integer`.
3. Modify the grammar in order to be able to only use synthetised attributes.

4.2 Typing

EXERCISE #3 ▶ **Different languages, different typing behaviors**

Listing 4.1: 'C code'

```c
#include <stdio.h>

int plusone(int i){
  return(i+1);
}

void test1(){
  printf("%i", plusone(3));
}

void test2(){
  printf("%i", plusone("three"));
}
```

Listing 4.2: 'Python code'

```python
def plusone(i):
    return i+1

def test1():
    print(plusone(3))

def test2():
    print(plusone('three'))

test1()
test2()
```

- Is the C program correct? Does the compiler accept it? Why, how ?
- Is the Python program correct? What is its runtime behavior, with or without the last line ?

4.2.1 Static Typing (in compilers)

We recall the MiniWhile syntax.

Expressions:

e	::=	c	*constant*
	|	x	*variable*
	|	$e+e$	*add*
	|	$e-e$	*sub*
	|	$e\times e$	*mult*
	|	e/e	*div*
	|	$-e$	*unary minus*
	|	$e<e$	*lt, similarly other comparators*
	|	$e\&\&e$	*and, similarly or*

Statements:

$S(Smt)$	::=	$x := expr$	assign
	|	$skip$	do nothing
	|	$S_1; S_2$	sequence
	|	`if` b `then` S_1 `else` S_2	test
	|	`while(` b `){` S `}`	loop

We recall (some of) the rules of the course for typing:

$$\frac{c \in \mathbb{Z}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash S_1; S_2 : \texttt{void}}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x = e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while(}\, b \,\texttt{)\{}\, S \,\texttt{\}} : \texttt{void}}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{void}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

If mandatory, other "similar" rules should be invented.

EXERCISE #4 ► **Typing under given environment**
Considering $\Gamma = \{x_1 \mapsto \texttt{int}\}$, prove that the program is well typed:

```
x1 = 3 ;
while (x1 < 15) { x1 = x1 + 1 ; }
```

EXERCISE #5 ► **Construction of Gamma with variable declarations**

Using the following rules:

$$\frac{}{t\ \ vlist; \rightarrow_d [v \mapsto t \text{ for v in } vlist]}$$

$$\frac{D_1 \rightarrow_d \Gamma_1 \quad D_2 \rightarrow_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D1; D_2 \rightarrow_d \Gamma_1 \cup \Gamma_2}$$

Compute Γ the typing environnement for the given list of declarations:

```
int x;
bool b1, b2;
```

EXERCISE #6 ► **Extend the grammar of expressions + typing rules**

Complete the abstract syntax and the static semantics (typing) of expressions with the new construction `e1? e2: e3` : if e_1 is true then the expression has value e_2 else e_3.

TD 5

3 Address Code Generation & Simple Allocation

5.1 Code generation with temporaries

The code we generate will have an unbounded number of temporaries (`tmp0`, `tmp1`, ...) but actual target instructions (add, and, ...).

The instruction set and documentation for the RISCV machine can be found in on the website (we already used it in previous sessions).

The code generation functions (see Section 5.3) have the following signatures:

`GenCodeExpr` : Expression → Code* × Temporary
`GenCodeSmt` : Statement → Code*

where Code* is a sequence of 3-address instructions (target machine, where physical registers have been replaced by temporaries). As a side effect, while processing variable declarations, the code generation for statements might update a map `symbol_table` *Var* → ℕ (program variable to a temporary where to find its current value).

Auxiliary functions to return fresh temporaries and labels:

`fresh_tmp()` : → Temporary
`new_label()` : → Label

EXERCISE #1 ▶ **By hand!**
Using the code generation rules for the RISCV machine, generate the three-address RISCV code for the body of the following (MiniC) function:

```
int main(){
    int a,n;

    n = 1;
    a = 7;
    while (n < a) {
      n = n+1;
    }

    return 0;
}
```

5.2 Language extensions

EXERCISE #2 ▶ **A new operator for expressions**
Write a code generation rule for the `xor` Boolean operator **without using the native** RISCV **operator.**

EXERCISE #3 ▶ **A new langage construct**
Write a code generation rule for the `repeat S until e` statement.

EXERCISE #4 ▶ **Another langage construct**
Invent a grammar rule and a code generation rule for `for loops` (Fortran style, C style)

5.3 Code Generation Rules

c	`dest <- fresh_tmp()`
	`code.add("li dest, c")`
	`return dest`

x	`# get the temporary associated to x.`
	`reg <- symbol_table[x]`
	`return reg`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dest <- fresh_tmp()`
	`code.add("add dest, t1, t2")`
	`return dest`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dest <- fresh_tmp()`
	`code.add("sub dest, t1, t2")`
	`return dest`

true	`dest <-fresh_tmp()`
	`code.add("li dest, 1")`
	`return dest`

$e_1 < e_2$	`dest <- fresh_tmp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`endrel <- new_label()`
	`code.add("li dest, 0")`
	`# if t1>=t2 jump to endrel`
	`code.add("bge endrel, t1, t2")`
	`code.add("li dest, 1")`
	`code.addLabel(endrel)`
	`return dest`

Figure 5.1: 3@ Code generation for numerical or Boolean expressions

x = e	`dest <- GenCodeExpr(e)`
	`loc <- symbol_table[x]`
	`code.add("mv loc, dest")`

S1; S2	`# Just concatenate codes`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` *b* `then` *S*1 `else` *S*2	`lelse <- new_label()`
	`lendif <- new_label()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add("beq lelse, t1, 0")`
	`GenCodeSmt(S1) # then`
	`code.add("j lendif")`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) # else`
	`code.addLabel(lendif)`

`while(` *b* `){` *S* `}`	`ltest <- new_label()`
	`lendwhile <- new_label()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add("beq lendwhile, t1, 0")`
	`GenCodeSmt(S) # execute S`
	`code.add("j ltest") # and jump to the test`
	`code.addLabel(lendwhile) # else it is done.`

Figure 5.2: 3@ Code generation for Statements

5.4 Allocations

EXERCISE #5 ▶ **Prepare the lab: allocations**
After code generation, we obtain the following code:

```
li temp_0, 42
li temp_1, 1
add temp_2, temp_1, temp_0
```

- Compute the naive allocation and rewrite the code accordingly.
- Compute the all-in-mem allocation and rewrite the code accordingly.

TD 6
Control Flow Graph and Local Optimisations

6.1 Control flow graph, basic blocks

EXERCISE #1 ▶ **CFG from 3-addresses code**

From the three-address code generation of the course, we get the following RISCV code with temporaries:

```
##Automatically generated RISCXX code
##non executable 3-Address instructions version
        li temp_2, 1
        mv temp_0, temp_2
        li temp_3, 7
        mv temp_1, temp_3
lbl_begin_while_1_main:
        li temp_4, 0
        bge temp_0, temp_1, lbl_end_relational_3_main
        li temp_4, 1
lbl_end_relational_3_main:
        beq temp_4, zero, lbl_end_while_2_main
        li temp_5, 1
        add temp_6, temp_0, temp_5
        mv temp_0, temp_6
        j lbl_begin_while_1_main
lbl_end_while_2_main:
        mv a0, temp_0
        call println_int
```

1. Find the leaders of blocks, and then draw the CFG of this program.
2. Give a MINIC program that could have been the source file.
3. (optional demo) Let your teaching assistant make a demo of the CFG construction with the help of the `clang` front-end analyzer.

EXERCISE #2 ▶ **CFG By hand**

What are the expected result of the 3@-code generation + CFG construction on this MINIC program?

```
int x;
x=0;
while (x < 4){
   x=x+1;
}
```

6.2 (Basic Block) Instruction Scheduling

In this section we will schedule instructions *inside a given basic block*. Scheduling a (given 3 address) instruction consists in computing a logical date t for it such that:

- all operands that are read in the instruction have been computed before (at a logical date strictly less than t)

- all use of the write operand (if any) are done later (at $t' > t$).

We recall below the Sethi-Ullman algorithm on trees, which takes as input an expression AST. The algorithm:

- Computes the minimal number of registers necessary to compute the given expression *without moves and without live range splitting.*
- Generates a code with a minimal number of registers.

Sethi-Ullman algorithm Let $\rho(node)$ denote the number of (pseudo)-registers necessary to compute an expression (AST), then:

- $\rho(leaf) = 1$
- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho(e_2)\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non "balanced" subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

Then the code is produced with postfix tree traversal, the biggest register consumers first.

EXERCISE #3 ▶ **Sethi-Ullman**

Consider the expression $E = ((a+b)*(a-b))+1$ where a and b are stored in **stack slots**. a and b will be refered as $[a]$ and $[b]$ in the load instruction.

1. What is the minimum amount of registers required to evaluate E?
2. Give the corresponding RISCV code (with temporaries, not physical registers).
3. Draw the liveness intervals for your code. Show an execution point with a maximum number of alive temporaries.

EXERCISE #4 ▶ **Sethi Ullman - 2**

Consider the expression $E = ((n*(n+1))+(2*n)$. We assume that the variable n is stored in the stack slot referred as $[n]$ in the load (ld) instruction (like in the previous exercise).

1. Use the Sethi Ullman algorithm to generate the code.
2. Show a difference with the code we would have obtained with the 3 address code generation. Show that no dependency has been introduced.
3. (Without applying liveness analysis) Draw the liveness intervals. How many registers are sufficient to compute this expression?

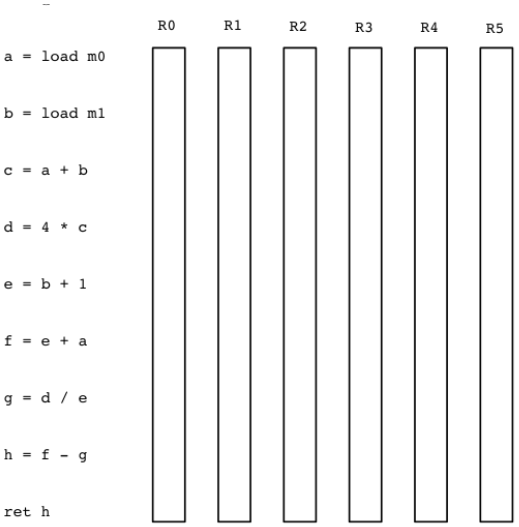
6.3 Toward Basic Block Register Allocation

Credits *From F. Pereira, UFMG, Brasil (DCC 888 Course)*

EXERCISE #5 ▶ **Manual register allocation on straight-line code**

Local register allocation is the problem of finding a physical location to all the variables that a program manipulates. These physical locations can either be registers, or memory.

1. Assuming an architecture with six registers, e.g., R0 to R5, perform register allocation for the program below. Use the bars to show where each variable will be in registers.

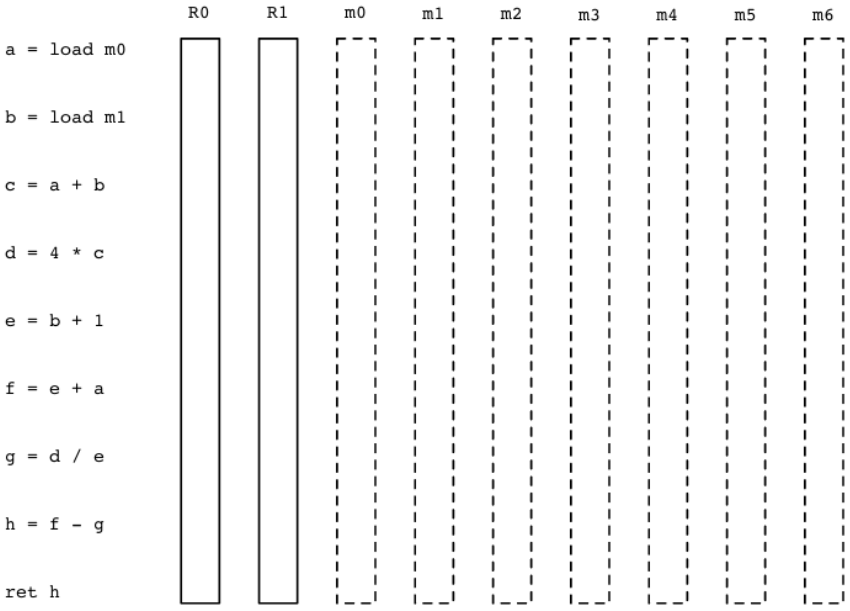


2. Explain informally how a bin packing like algorithm can be applied here.

EXERCISE #6 ▶ **Manual register allocation with spill**

Usually we do not have an unbounded surplus of registers, and spills might happen. Spilling is the act of mapping a variable to memory. We do this via special instructions, e.g., loads and stores. Perform register allocation in the program below, assuming an architecture with only two registers, R0 and R1. You must insert loads and stores in the program, to move variables to and from memory. The syntax of these instructions is as follows:

- `x = load m`: loads the contents in memory cell m into variable x.
- `store x m`: stores the contents of variable x into memory cell m.



Take away message : **liveness (range) information** is the key of basic block (local) and global optimisations (including register allocation).

TD 7

Dataflow analyses

7.1 Liveness analysis

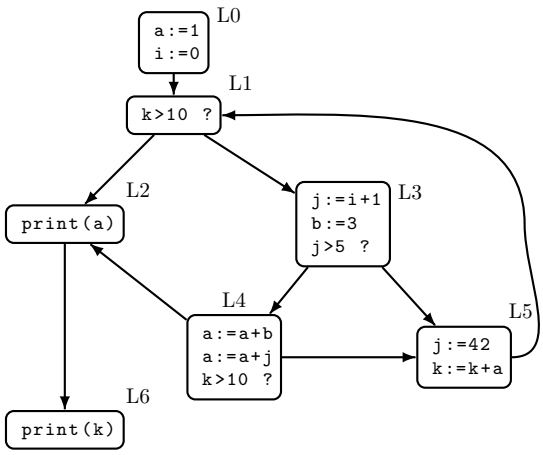
We recall that a *variable is alive after a block if there exists a path from this block to one use of this variable that does not contain a definition of it.*

The objective of liveness analysis is to compute this information for all control points or basic blocks of the program under analysis.

7.2 Liveness by hand

EXERCISE #1 ▶ **Liveness by hand - CC 2016**

In Figure 7.1, we give a CFG.



block	live variables at block exit
L0	
L1	
L2	
L3	
L4	
L5	
L6	∅

Figure 7.1: CFG and alive variables (to be completed)

1. (by hand) Fill the array with "out"-alive variables for each block.
2. Remove dead code.

7.3 Liveness with fixpoint

Let us recall the notations here: A variable at the left-hand side of an assignement is *killed* by the block. A variable whose value is used in this block (before any assignment) is *generated*.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup_{\ell' \in succ_G(\ell)} LV_{entry}(\ell') & \text{else} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.

EXERCISE #2 ► **Live variables**
Generate the CFG for the following program:

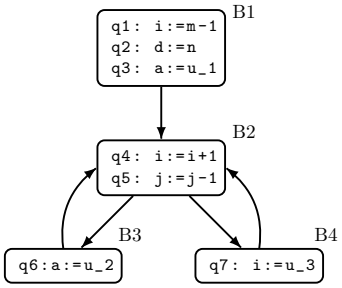
```
while (d>0) {
    a:=b+c;
    d:=d-b;
    e:=a+f;
    if (e>0) {
        f:=a+d;
        b:=d+f;
    } else {
        e:=a-c;
    }
    b:=a+c;
}
```

On this CFG:
- Compute Gen, $Kill$ for each block ℓ
- Compute $In(\ell) = LV_{entry}(\ell)$ and $Out(\ell) = LV_{exit}(\ell)$ iteratively.
- Suppress the dead code.

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

EXERCISE #3 ▶ **Live Variables**
After code generation, we obtain the following graph:



On this graph, perform liveness analysis and suppress the dead code.

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

7.4 Other dataflow analyses

EXERCISE #4 ▶ **Common subexpressions/Available expressions**
Consider the following program:

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```

- What optimisation would be desirable ?
- Give a definition for “an available expression”
- Design an analysis to compute this information.

EXERCISE #5 ► **Very busy expressions**

```
x = input
a = x - 1
b = x - 2
while (x > 0) {
   print( a * b - x )
   x = x - 1
}
output a * b
```

EXERCISE #6 ► **Constant propagation**
In this exercice we will study the following program:

```
z:=3
x:=1
while (x>0) {
  if (x=1) then
     y:=7;
  else
     y:=z+4;
  x:=3
  print y
}
```

1. Draw the CFG where blocks are statements.
2. Which of variables x,y,z are constants ? What modification can be done on the code if we know this information?
3. How is a "I'm a constant" information generated ? killed ? Is it a forward or backward dataflow propagation?
4. How to merge two incoming informations $x \mapsto 1$ and $x \mapsto 2$ in a test ?
5. Give an algorithm to compute all the constants available at the begining of each block.

TD 8

Register allocation and final code generation

In the following exercises we are looking for backend-independant optimisations (on the 3-address code). The goal here is to allocate registers with as few "spilled variables" as possible.

8.1 Register Allocation

EXERCISE #1 ▶ **Code production and register allocation**
Consider the expression $E = ((n * (n+1)) + (2 * n)$. We assume that we have:

- The variable n is stored in the stack slot referred as $[n]$ in the load (ld) instruction (we consider that variable n is stored in memory anyway here).
- For memory allocation: use $r_6, r_7, r_{...}$ for physical multi-purpose registers and s_1, s_2, s_3 for spilling dedicated registers.

1. Generate a 3 address-code with temporaries and `ld` instruction to load n. Do it as blindly as possible (no temporary recycling).
2. (Without applying the fixpoint iteration for liveness) Draw the liveness intervals. How many registers are sufficient to compute this expression?
3. Draw the interference graph (nodes are variables, edges are liveness conflicts).
4. Color this graph using the algorithm seen in the course (unbounded number of colors, first is red, then blue, then green, then ...).
5. Give a register allocation with $K = 2$ registers, and rewrite code.

EXERCISE #2 ▶ **Adapted from exam 2018**

We consider the following MiniC program:

```
int x,y,z,t;
x=12; y=3+x; z=4+y; t=x-y+z;
```

The 3 address code generation process of Lab 4/5 produces the following code, where $(t,z,y,x) \mapsto (temp\_0, temp\_1, temp\_2, temp\_3)$:

```
1  li temp_4, 12
2  mv temp_3, temp_4
3  li temp_5, 3
4  add temp_6, temp_5, temp_3
5  mv temp_2, temp_6
6  li temp_7, 4
7  add temp_8, temp_7, temp_2
8  mv temp_1, temp_8
9  sub temp_9, temp_3, temp_2
10 add temp_10, temp_9, temp_1
11 mv temp_0, temp_10
```

1. Fill the array with the result of the liveness analysis. Each star in a line will mean "the temporary is alive at the entry of this line":

code	temp_1	temp_2	temp_3	temp_4	temp_5	temp_6	temp_7	temp_8	temp_9	temp_10
`li temp_4, 12`										
`mv temp_3, temp_4`										
`li temp_5, 3`										
`add temp_6, temp_5, temp_3`										
`mv temp_2, temp_6`										
`li temp_7, 4`										
`add temp_8, temp_7, temp_2`										
`mv temp_1, temp_8`										
`sub temp_9, temp_3, temp_2`										
`add temp_10, temp_9, temp_1`										
`mv temp_0, temp_10`										

2. Draw the interference graph.

   In the rest of the exercice we will use the following notations: Color 1 is red, and is associated to register t1; Color 2 is blue, and is associated to register t2. If a third color is needed, it will be associated to memory location -8(fp).

3. Color the graph with the heuristic of the course and 2 colors. Do not forget to draw the color stack.

4. What are the variable(s) to spill? Allocate this/these variable in memory without live range splitting (second algorithm of the course) and generate code for the associated instructions.

5. (Course question) What could have been done to remove the useless moves? when ?

EXERCISE #3 ► (If time allows) Register allocation, adapted from Exam, 2016

We consider (in two columns) the following RISCV code. The tmp_i are temporaries to be allocated (in registers, in memory). For this exercise, we consider that we have two instructions that are capable to directly read/write at memory labels (`ld , sd`).

```
[...]
ld  tmp_1, label1
ld  tmp_2, label2
sub tmp_3, tmp_1, tmp_2
ld  tmp_4, label3
ld  tmp_5, label4
sub tmp_6, tmp_4, tmp_5
add tmp_7, tmp_6, 0
add tmp_8, tmp_3, tmp_7
sd  tmp_8, label5
ret
```

```
;;données/résultats (.dword = 8 bytes)
label1 : .dword 2
label2 : .dword 3
label3 : .dword -1
label4 : .dword 7
label5 : .dword 0
```

1. What is the computed expression? Where will it be stored?

2. Fill the following table with stars: put a star for a given temporary at a given line if and only if it is alive *at the entry of the instruction.* After the last store, all temporaries are supposed to be dead.

code	tmp_1	tmp_2	tmp_3	tmp_4	tmp_5	tmp_6	tmp_7	tmp_8
`ld tmp_1, label1`								
`ld tmp_2, label2`								
`sub tmp_3, tmp_1, tmp_2`								
`ld tmp_4, label3`								
`ld tmp_5, label4`								
`sub tmp_6, tmp_4, tmp_5`								
`add tmp_7, tmp_6, 0`								
`add tmp_8, tmp_3, tmp_7`								
`sd tmp_8, label5`								

3. Draw the interference graph.
4. Color the graph with the algorithm from the course with an infinite number of color (green, blue, red, black , . . . in this order).
5. (We make as it we had only 2 available registers). We decide to spill the tmp_3 register and place it in memory.

   Generate the final code with two registers (t_3, t_4), sp and fp for the stack, $s1, s2, s3$ for the spill management.