Practical #6:
Code Improvement

COS**341** Compiler Construction

Universiteit van Pretoria

Academic Year 2015

Submission due: Wednesday the **29th of April**, mid-day (online)
Presentation due: Wednesday the **29th of April**, evening (laboratory)

The usual **terms and conditions** [see Study-Guide] are **applicable without exception**

Motivation and Preparation

- After the compiler has produced intermediate code, a variety of automated improvements on the intermediate code are possible:
  - *In coder-jargon, those improvements are often called “optimisations”, though* ***it cannot be guaranteed*** *(from a theoretical point of view) that the “optimised” code is truly optimal.*
- This is the topic of this practical:
  - *Again we will use “BASIC” as our intermediate code, such that we will be able to test if our “optimisation”-improvement is working well.*

Here you can see an implementation of the QUICKSORT Algorithm as **Source-Code***

```
void quicksort( int m, int n )
   {
•  int i, j ;
•  int v, x ;
•  if (n <= m) then { return ; }
•  i := m-1; j := n; v := a[n];
•  while(true)do
   {
       do i++ while (a[i] < v);
       do j-- while (a[j] > v);
       if (i >= j) then { break ; }
       x := a[i]; a[i] := a[j]; a[j] := x;
   }
•  x := a[i]; a[i] := a[n]; a[n] := x;
•  quicksort( m , j);
•  quicksort(i+1, n);
   }
```

*** R. Sedgewick**:
"*Implementing Quicksort Programs*". **Communications of the ACM 21**, pp. 847-857, **1978**.

Here you can see an implementation of the QUICKSORT Algorithm as **Source-Code**

```
void quicksort( int m, int n )
   {
•  int i, j ;
•  int v, x ;
•  if (n <= m) then { return ; }
•  i := m-1; j := n; v := a[n];
•  while(true)
   {
       do i++ while (a[i] < v);
       do j-- while (a[j] > v);
       if (i >= j) then { break ; }
       x := a[i]; a[i] := a[j]; a[j] := x;
   }
•  x := a[i]; a[i] := a[n]; a[n] := x;
•  quicksort( m , j);
•  quicksort(i+1, n);
   }
```

This snippet,
in blue colour,
will be shown
as intermediate
code on the
following slide ➔

Snippet out of “Quicksort”, shown as Intermediate Code (in BASIC)*

- 01 I = M – 1
- 02 J = N
- 03 T1 = 4 * N
- 04 V = A(T1)
- **05** I = I + 1
- 06 T2 = 4 * I
- 07 T3 = A(T2)
- 08 IF T3 < V THEN **05**
- **09** J = J – 1
- 10 T4 = 4 * J
- 11 T5 = A(T4)
- 12 IF T5 > V THEN **09**
- 13 IF I >= J THEN **23**
- 14 T6 = 4 * I
- 15 X = A(T6)
- 16 T7 = 4 * I
- 17 T8 = 4 * J
- 18 T9 = A(T8)
- 19 A(T7) = T9
- 20 T10 = 4 * J
- 21 A(T10) = X
- 22 GOTO **05**
- **23** T11 = 4 * I
- 24 X = A(T11)
- 25 T12 = 4 * I
- 26 T13 = 4 * N
- 27 T14 = A(T13)
- 28 A(T12) = T14
- 29 T15 = 4 * N
- 30 A(T15) = X

*** Note**: Here we already include *Memory-Calculations* on the basis of ***4-Byte**-Fields*

Next we need to understand the concept of “**blocks**” in Intermediate Code, as follows:

- A “block” begins at some address **ADR** in case the intermediate code contains some command: GOTO **ADR**
- A “**block**” begins also immediately behind an **IF** statement
- A “**block**” ends before the beginning of its successor-”block”

Our Example Intermediate Code, now with **Blocks** being highlighted

```
01  I = M - 1
02  J = N
03  T1 = 4 * N
04  V = A(T1)
05  I = I + 1
06  T2 = 4 * I
07  T3 = A(T2)
08  IF T3 < V THEN 05
09  J = J - 1
10  T4 = 4 * J
11  T5 = A(T4)
12  IF T5 > V THEN 09
13  IF I >= J THEN 23
14  T6 = 4 * I
15  X = A(T6)
16  T7 = 4 * I
17  T8 = 4 * J
18  T9 = A(T8)
19  A(T7) = T9
20  T10 = 4 * J
21  A(T10) = X
22  GOTO 05
23  T11 = 4 * I
24  X = A(T11)
25  T12 = 4 * I
26  T13 = 4 * N
27  T14 = A(T13)
28  A(T12) = T14
29  T15 = 4 * N
30  A(T15) = X
```

Let us now analyse this Block:

- **23** T11 = 4 * I
- 24 X = A(T11)
- 25 T12 = 4 * I
- 26 T13 = 4 * N
- 27 T14 = A(T13)
- 28 A(T12) = T14
- 29 T15 = 4 * N
- 30 A(T15) = X

Let us now analyse this Block:

Here we see repeated identical calculations!
➔
T11 and T12 must have the same values!

```
23  T11 = 4 * I
24  X = A(T11)
25  T12 = 4 * I
26  T13 = 4 * N
27  T14 = A(T13)
28  A(T12) = T14
29  T15 = 4 * N
30  A(T15) = X
```

Let us now analyse this Block:

Here we see again repeated identical calculations!
➔
T13 and T15 must have the same values, too!

```
23  T11 = 4 * I
24  X = A(T11)
25  T12 = 4 * I
26  T13 = 4 * N
27  T14 = A(T13)
28  A(T12) = T14
29  T15 = 4 * N
30  A(T15) = X
```

Improvement :

```
23  T11 = 4 * I
24  X = A(T11)

26  T13 = 4 * N
27  T14 = A(T13)
28  A(T11) = T14

30  A(T13) = X
```

```
23  T11 = 4 * I
24  X = A(T11)
25  T12 = 4 * I
26  T13 = 4 * N
27  T14 = A(T13)
28  A(T12) = T14
29  T15 = 4 * N
30  A(T15) = X
```

TWO LINES OF CODE LESS ☺

Your **Tasks**

Presume that the **Tutor will provide** you with a BASIC Program (which represents some "Intermediate Code")

- **A) [2 Marks]**
  - Write a **"chopper" software** which correctly "chops" the given BASIC program into its "*Blocks*" (as defined)
- **B) [2 Marks]**
  - Write a **"duplication elimination software"** which can find and eliminate *from each Block* the kind of redundancies illustrated by the examples on the previous slides.

Note: If you implemented out your solution correctly, then the "optimised" BASIC program must be **I/O-equivalent** to the given BASIC program!

And now :

HAPPY CODING

☺