Code Optimisation

- An optimiser looks at a representation of the source program and tries to produce **shorter** or **faster** code (or both).
- There are essentially two ways in which optimisation can take place:
  - Reorganise the structure of the source algorithms to make them more efficient. This generally operates on the parse tree. This technique is machine independent.
  - Modification of the code produced by a simple translator to make it efficient. This phase operates on the object code.

Common Optimisation Tasks (1)

- Common Sub Expressions
  - An occurrence of an expression E is called a **common sub-expression** if E was previously computed and the values of the variables in E have not changed. In such cases we can avoid recomputing an expression of we can use the previously computed value.
- Copy Propagation
  - Reorganises assignment statements so that:

```
x = y
z = x
```

  - becomes

```
x = y
z = y
```

  - (more on this later)

Common Optimisation Tasks (2)

- Dead Code Elimination
  - A variable is 'live' at a point in a program if its value can be used subsequently, otherwise it is 'dead' at that point. Statements may compute values that may never be used in a program. While a programmer is unlikely to introduce dead code intentionally, it may appear as a result of previous transformations. Consider the statement:

    ```
    if (debug) then Print ...
    ```

  - By data flow analysis it may be deduced that no matter what path the program takes, when the statement is reached, the value of `debug` would always be false, so the test and printing may be removed from the object code.

Common Optimisation Tasks (3)

- Dead Code Elimination Continued
  - One advantage of copy propagation is that it often turns an assignment statement into dead code. For example copy propagation followed by dead code elimination would convert:

    ```
    x = t3
    a[t2] = t5
    a[t4] = x
    goto b2
    ```

  - By elimination of copy propagation:

    ```
    x = t3
    a[t2] = t5
    a[t4] = t3
    goto b5
    ```

  - By Dead code elimination:

    ```
    a[t2] = t5
    a[t4] = t3
    goto b5
    ```

Common Optimisation Tasks (4)

- Loop Optimisation
  - Loops are an important place where optimisations may occur. The running time of a loop may be improved if we decrease the number of instructions occurring inside. A common loop optimisation is called code motion.
  - Code motion attempts to move code out of the loop (though the expression must yield the same result). This transformation takes an expression that has the same evaluation independent of the number of times the loop executes (called a loop-invariant computation) and places it before the loop. For example the computation of `limit - 1`, is loop invariant in:

```
while (i < (limit - 1)) ...
```

  - So we can have:

```
t = limit - 1
while (i < t) ...
```