CD Moodle Assignment 6 

Jovial Joe Jayarson

25 June 2020

IES17CS016

1. Explain functional preserving transforms, its significance and types.

Ans 1:

These are the optimizing methods that optimizes code but preserves what the function computes.

1. **Common Subexpression Elimination** (CSE)
    - Subexpression which are redundant can be eliminated.
    - If E is an expression and E_n if and only if the latter has not been altered in between.
    - Consider the expression:

```
a := b + c ... (1)
b := a - d ... (2)
c := b + c ... (3)
d := a - d ... (4)
```

    `(2)` & `(4)` have same RHS expression, the value of neither `a` not `d` values change in between so the condensed / optimised expression will be:

```
a := b + c
b := a - d
c := b + d
d := b
```

2. **Copy Propagation**
    - Assignments of the form `y := x` are called statements.
    - These:

```
C := A
B := C
D := B
```

    - Can be written as:

```
C := A
B := A
D := A
```

3. **Dead Code Elimination**
    - The part of code which never gets executed is called dead code.
    - The `print` statement in the following code is unreachable.

```
def func(a, b):
    p = a + b
    return p
    print('Sum') # Dead code
```

4. **Constant folding**
    - Constants separated by expression can be folded / merged to reduce compile time overhead.
    - The statement `some_num := 4 + 7` can be simply written as: `some_num := 13`.

Some other optimization techniques are: Algebraic Transformations, Interchanging statements, Renaming temporary variables etc.

2. Discuss about peephole optimization

Ans 2:

- Peephole optimization is applied to improve performance of the program.
- Done using a sequence of instructions (like through a sliding window)
- There are many techniques:
    - **Redundant instruction elimination**: Consider the following:

```
...
MOV R0, a
; code for some other purpose does not use either R0 or a
MOV a, R0 ; redundant
...
```

The value of `a` can be directly accessed from `R0` so it need not be moved back to, which is a redundant.

- **Removal of unreachable code**: Certain statements never get executed which are useless and are called unreachable. e.g.

```
def func(a, b):
    p = a + b
    return p
    print('Sum') # Dead code
```

- **Flow control optimization**: Uncessay jumps can be eliminated.
    - Initial code

```
goto L1
L1: goto L2
```

```
L2: goto L3
L3: MOV R0, a
```

- Optimized Code

```
goto L3
; L1: goto L2 commented out
; L2: goto L3 commented out
L3: MOV R0, a
```

- **Algebraic Simplifications & Reduction in Strength**: Expression that can be simplified are done so to reduce compilation time.

```
x = 10

x = x * x * x * x # can be optimized to
x **= 4
```

- **Use of Machine Idioms**: Process of using powerful cpu features to get instruction performed.

```
MOV bx, offset var ; is equivalent to
LEA bx, var ; but slower
```

3. Explain the design issues of a code generator

Ans 3:

Design Issues of a code generator are:

1. **Input to the code generator**
   - Accepts optimized intermediate code as input from the code optimizer. This can be of the form:
     - Postfix Notation
     - Three Address Code
     - DAG or Syntax Directed tree
   - The assumption is that the input is free of errors.
2. **Target program**
   - The target program is usually machine language.
   - The requirement may be either absolute or relocatable
     - Absolute:
       - The memory location is fixed.
       - Suitable for small programs

        - Faster compilation and execution
    - Relocatable:
        - Linkers & loaders are used.
        - Several dynamic links are made, slower execution.
        - Suitable for large programs.
- Other target languages include assembly, C/C++ etc.
    - In assembly like languages target code is still human readable.
- Examples for target machines are: RISC, CISC Stack based VMs etc.

3. **Memory Management**
    - Symbol table is used to manage memory.
    - Mapping of variable names to address is done cooperatively by the front-end and code generator.
    - Labelling must be done properly esp. for jump statements.
4. **Instruction Selection**
    - Selection of instruction can determine the speed of execution.
    - For example the following code:

```
MOV R0, a
ADD R0, #1
MOV a, R0
```

    can be replaced with:

```
INC a
```

    which is faster and memory efficient.

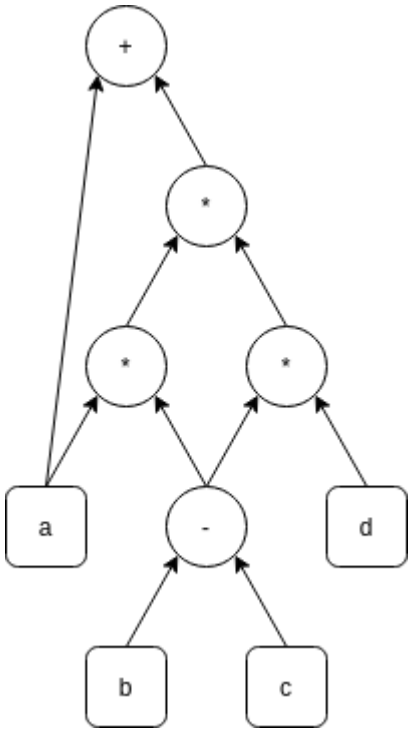
    - Uniformity & completeness of the instruction set are important factors.
5. **Register Allocation**
    - The key problem in code generation is what to hold in what registers.
    - Registers are fast and scarce and hence expensive. Since the ratio of the number of variables to number of registers is very high.
    - Register Allocation specifies which registers *contains* which values.
    - Register Assignment specifies which variables is *contained in* which registers.
    - Finding and optimal solution is an *NP complete* problem.
6. **Evaluation Order**

- The order in which the instructions are executed as well as the operations are performed will decide the efficiency.
- This is also an NP complete problem.

4. Explain about optimization of basic blocks

Ans 4:

- **Basic Blocks** are sets of statements executed in a sequence.
  - It has one entry and exit point.
  - There are neither controls nor conditional statements in this block.
- **Directed Acyclic Graph (DAG)**:
  - Many of the structure preserving transformations are implemented by constructing DAGs.
  - Properties:
    - Internal nodes represent operators / result of expression
    - Leaf Nodes represent identifiers / constants.
  - The DAG for the expression `a + a * (b - c) + (b - c) * d` will created like this:



  - DAG's help to determine common sub-expressions.
  - Determines which names are within the block and which can be computed outside.
  - A new node is created only when an unaltered subnode exists.
  - Thus DAGs help in optimizing the basic blocks.