Comp 204: Computer Systems and Their Implementation

Lecture 22: Code Generation and Optimisation

Today

- Code generation
  - Three address code
- Code optimisation
  - Techniques
  - Classification of optimisations
    - Time of application
    - Area of application

Intermediate Code

- Code can be generated from syntax tree…
  - However, this doesn't represent target code very well
  - Tree represents constructs such as conditionals (if… then…else) or loops (while…do)
  - Target code includes jumps to memory addresses
- Intermediate code represents a linearisation of the syntax tree
  - Postfix is an example of a stack-based linerisation
  - Typically related in some way to target architecture
  - Good for efficient code
    - Can be exploited by code optimisation routines

Three Address Code

- Reflects the notion of simple operations of the form:

  `x = y` *op* `z`

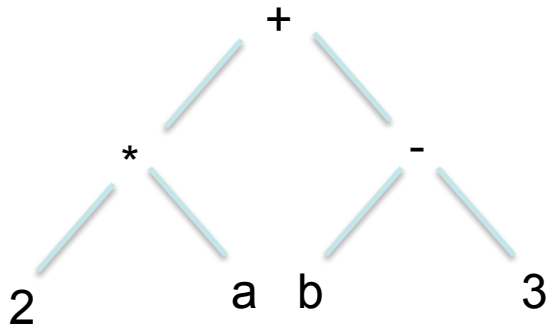
- Many instructions are of this form
  - Introduces the notion of temporary variables
    - These represent interior nodes in the tree
    - Usually assigned to registers
  - Represents a left-to-right linearization of the code
  - Other variants exist, e.g. for unary operations

    ```
    x = -y
    ```

Three Address Code

- Consider the arithmetic expression

```
2*a+(b-3)
```



- The corresponding three-address code is

```
t1 = 2 * a
t2 = b – 3
t3 = t1 + t2
```

Example: *factorial* function

```
read x;
if (0 < x) then
  fact := 1;
  repeat
    fact := fact * x;
    x := x – 1;
  until x = 0;
  write fact;
end
```

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x – 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

P-Code

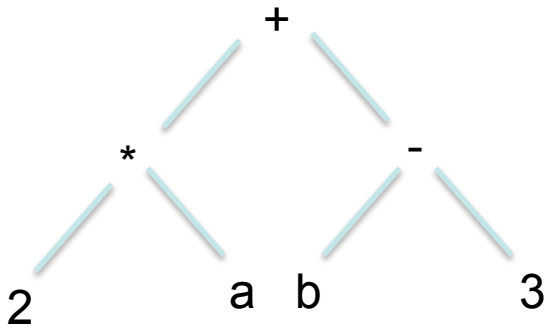
- Was initially a target assembly generated by Pascal compilers in early 70ies
- Format is very similar to assembly
  - designed to work on a hypothetical stack machine called a P-machine
  - aim was to aid portability
    - P-code instructions could then be mapped to assembly for target platform
- Simple, abstract version given on the next slide

P-Code

- Consider the arithmetic expression

  `2*a+(b-3)`

- The corresponding P-code is




```
lcd 2   ; load constant 2
lod a   ; load value of var a
mpi     ; integer multiplication
lod b   ; load value of var b
ldc 3   ; load constant 3
sbi     ; integer subtraction
adi     ; integer addition
```

Question

- Which of the following is NOT a form of intermediate representation used by compilers?

a) Postfix
b) Tuples
c) Context-free grammar
d) Abstract syntax tree
e) Virtual machine code

Answer: c
A context-free grammar defines the language used by the compiler; the rest are intermediate representations

Code Optimisation

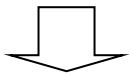
- Aim is to improve quality of target code

- Disadvantages
  - compiler more difficult to write
  - compilation time may double or triple
  - target code often bears little resemblance to unoptimised code
    - greater chance of translation errors
    - more difficult to debug programs

Optimisation Techniques

- Constant folding
  - can evaluate expressions involving constants at compile-time
  - aim is for the compiler to pre-compute (or remove) as many operations as possible

```
a := 3*16 - 2;
```



```
LOAD 1, #46
STORE 1, a
```

Techniques

- Global register allocation
  - analyse program to determine which variables are likely to be used most and allocate these to registers
  - good use of registers is a very important feature of efficient code
    - aided by architectures that provide an increased number of registers

Techniques

- Code deletion
  - identify and delete unreachable or dead code

```
boolean debug = false;
...
if (debug) {
    ...
}
```

No need to generate code for this

Techniques

- Common sub-expression elimination
  - avoid generating code for unnecessary operations by identifying expressions that are repeated

```
a := (b*c/5 + x) - (b*c/5 + y)
```

  - generate code for `b*c/5` only once

Exercise

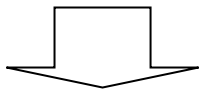
- Optimise the following:

```
a = 100-3*22;
b = (a-30)*5;
if (a<b){
   screen.println(a);
}
```

Techniques

- Code motion out of loops

```
for (int i=0; i <= n; i++) {
   x = a + 5;   //loop-invariant code
   Screen.println(x*i);
}
```



```
x = a + 5;
for (int i=0; i <= n; i++) {
   Screen.println(x*i);
}
```

Techniques

- Strength reduction
  - replace operations by others which are equivalent but more efficient
    e.g. a* 2

```
LOAD 1, a          LOAD 1, a
MULT 1, #2   ⇒     ADD 1, 1
```

Question

- What optimisation technique could be applied in the following examples?

  a = b^2

  a = a / 2

a) Constant Folding
b) Code Deletion
c) Common Sub-Expression Elimination
d) Strength Reduction
e) Global Register Allocation

Answer: d
Both expressions can be reduced by changing the operator:
a = b ^ 2 can be reduced to a = b * b
a = a / 2 is a right shift operation: a = a >> 1

Classification of Optimisations

- Optimisations can be classified according to their different characteristics

- Two useful classifications:
  - the period of the compilation process during which an optimisation can be applied
  - the area of the program to which the optimisation applies

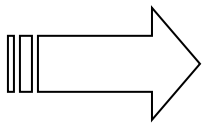
Time of Application

- Optimisations can be performed at virtually every stage of the compilation process
  - e.g. constant folding can be performed during parsing
  - other optimisations might be applied to target code
- The majority of optimisations are performed either during or just after intermediate code generation, or during target code generation
  - source-level optimisations do not depend upon characteristics of the target machine and can be performed earlier
  - target-level optimisations depend upon the target architecture
    - sometimes an optimisation can consist of both

Target Code Optimisations

- Optimisations performed on target code are known as peephole optimisations
  - scan target code, searching for sequences of target code that can be replaced by more efficient ones, e.g.

```
LOAD 1, a
ADD 1, #1
STORE 1, a
```



```
INC a
```

  - replacements may introduce further possibilities
  - effective and simple
  - sometimes tacked onto end of one-pass compiler

Area of Application

- Optimisations can be applied to different areas of a program
  - Local optimisations: those that are applied to ‘straight-line’ segments of code, i.e. with no jumps into or out of the sequence
    - easiest optimisations to perform
  - Global optimisations: those that extend beyond basic blocks but are confined to an individual procedure
    - more difficult to perform
  - Inter-procedural optimisations: those that extend beyond the boundaries of procedures to the entire program
    - most difficult optimisations to perform