Compiler Theory

(Intermediate Code Generation - Abstract Syntax + 3 Address Code)

006
Why *intermediate code*?

- Details of the source language are confined to the front-end (analysis phase) of a compiler, while details of the target machine are confined to the back-end (synthesis) part.
- This saves a considerable amount of effort since with $m$ front-ends and $n$ back-ends we have $m \times n$ compilers.

![Diagram of compiler structure](image)
Intermediate representations

- Syntax Trees
  - Code is represented in the form of a tree where nodes represent constructs in the source program; the children of a node represent the meaningful components of a construct.

- Three-Address Code
  - Made up of instructions of the general form $x=y \text{ op } z$
  - $X, y$ and $z$ are the three addresses.

- C
  - Often used as an intermediate representation. e.g. LOTOS language
Directed Acyclic Graphs (i)

- Nodes in a syntax tree represent constructs in the source program.

- A DAG is used to identify common sub-expressions. e.g. $a + a \times (b - c) + (b - c) \times d$

- By doing so it gives the compiler important hints on how to generate efficient code to evaluate the expressions.
DAG for $a+a*(b-c)+(b-c)*d$

Figure 6.3: Dag for the expression $a+a*(b-c)+(b-c)*d$
Three Address Code (TAC)

- An alternative form of intermediate (lower level) representation.
- \( x + y \times x \) becomes
  - \( t_1 = y \times z \)
  - \( t_2 = x + t_1 \)
- For expressions TAC is very similar to syntax trees.
- For statements it would produce labels and jumps in a similar fashion to machine code.
TAC for $a + a \times (b-c) + (b-c) \times d$

(a) DAG

(b) Three-address code

$t_1 = b - c$
$t_2 = a \times t_1$
$t_3 = a + t_2$
$t_4 = t_1 \times d$
$t_5 = t_3 + t_4$

Figure 6.8: A DAG and its corresponding three-address code
Addresses and Instructions

- An address can be
  - A name: source program names
    - In an implementation we would have these as pointers pointing towards the symbol table.
  - A constant
  - A compiler-generated temporary
    - To store temporary results
Addresses and Instructions

- Assignment instructions
  - \( x = y \ op \ z \) where \( op \) is binary
  - \( x = op \ y \) where \( op \) is unary (\(-\), \(!\), type casting)
- Copy Instructions
  - \( x = y \)
- Jumps
  - Unconditional: goto L
  - Conditional:
    - if \( x \) goto L
    - ifFalse \( x \) goto L
    - if \( x \ \text{relop} \ y \) goto L
Procedure Calls and returns
- Parameters : param x
- Procedure with n params: call p,n
- Function with n params : y = call f,n

Indexed Array[] Access
- x = y[i]
- x[i] = y

Address and Pointer Assignments (no need to be covered)
- x = &y
- x = *y
- *x = y
do $i = i + 1$; while ($a[i] < v$);

(a) Symbolic labels.  

(b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements.
Quadruples and Triples

- Data structures to hold three address code instructions.
- A Quadruple has four fields ( x = y+z)
  - Op (+)
  - Arg1 (y)
  - Arg2 (z)
  - Result (x)
An example ...

Note that in an actual implementation a, b and c should be pointers to the symbol table.

Figure 6.10: Three-address code and its quadruple representation
Triples

- Omit result field.
- Instead of a result field we can use pointers to the triple structure itself.
- This makes DAG and triple representation practically identical, since we are pointing to a node.
- In next example (n) indicates position n in the triple structure
An example ...

- Note similarity ...

Figure 6.11: Representations of $a + a \times (b - c) + (b - c) \times d$
Code Generation

- This is the final phase of a compiler
- Takes an intermediate representation and generates the equivalent target program
- Code optimisation (if any) occurs between the intermediate and target code generation
- We require that the code
  - is correct and
  - effectively uses the resources on the target machine
  - Is itself efficient in generating code
Whoever is designing the code generator must have a very good knowledge of the architecture of the target hardware and operating system.

Should keep in mind
- Memory management
- Instruction selection
- Register allocation
- Evaluation order

We shall look at generic issues
Input to the Code Generator

- Consists of
  - Intermediate Code produced by front end
  - Symbol Table to determine the runtime addresses of the data objects denoted by the names in the intermediate representation

- The underlying machine memory is byte-addressable and would have a number (say n) of general-purpose registers.
Assembly Language Instructions

- Most instructions consist of an operator, followed by a target, followed by a list of source operands.

- A label may precede an instruction

- We the next few slides we shall look at a number of different instruction classes.
Load Operations

- **LD \(dst, addr\)**
  - Loads the value in location \(addr\) into location \(dst\).
  - \(dst = addr\)
  - \(LD\ r, x\) loads the value at addr \(x\) into \(r\)
  - \(LD\ r1, r2\) loads the contents of register \(r2\) into register \(r1\)
Store operations

- ST \( x, r \)
  - Stored the value in register \( r \) into the location \( x \).
  - This instruction denotes the assignment \( x = r \).
  - Note difference from LD
Computation Operators

- **OP** $dst, src1, src2$
  - **OP** would be ADD, SUB etc
  - $dst, src1, src2$ are locations
  - Applies operation **OP** to the values in locations $src1$ and $src2$, and place the result of this operation in location $dst$
  - ADD $r1, r2, r3$ computes $r1=r2+r3$
  - Unary **OP** do not have $src2$
Unconditional Jumps

- **BR L**
  - Causes control to branch to the machine instruction with label L
  - BR stands for branch
Conditional Jumps

- **Bcond** $r, L$
  - Where $r$ is a register and $L$ is a label
  - Cond stands for any of the common tests on values e.g. $LT$, $GT$, etc
  - BLTZ $r, L$ causes a jump to label $L$ if the value in register $r$ is less than zero, and allows control to pass to the next machine instruction if not.
3AC to machine code (x=y-z)

- LD R1, y  // R1 = y
- LD R2, z  // R2 = z
- SUB R1, R1, R2  // R1 = R1 - R2
- ST x, R1  // x = R1
3AC to machine code (if x<y goto L)

- LD R1, x               // R1 = x
- LD R2, y               // R2 = y
- SUB R1, R1, R2         // R1 = R1 - R2
- BLTZ R1, M             // if R1<0 jump M
Instruction Selection

- Instruction selection effects
  - Execution speed and
  - Size

- A rich instruction set may provide several ways to perform any given operation.

- Typical e.g. … INC x instruction replaces ADD x + 1;
Register Allocation

Instructions involving registry operands are usually shorter and much faster than those involving memory operands. For this reason utilization of registers is important in generating fast code.

The use of registers is often subdivided into two problems
- During register allocation, we select the set of variables that will reside in registers at a point in the program
- During subsequent register assignments, we pick the specific register that the variable will be stored in

Optimal assignment of registers is NP-complete ... hence some heuristics have to be used.
Standard code optimizations (i)

- **Common Sub Expression**
  - 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 use the previously computed value of E.

- **Copy Propagation**
  - Reorganises assignment statements so that:
    - \( x = y \)
    - \( z = x \)
  - Becomes
    - \( x = y \)
    - \( z = y \)
Standard code optimizations (ii)

- Dead Code Elimination
  - A variable is 'live' at a point in a program if its value can be used subsequently, otherwise it is 'dead'. Statements may compute values that are never used in a program.
  - e.g. Computing expressions values which are never assigned
  - e.g. If (debug) then print ... and someone (using data flow analysis) the compiler can deduce that debug is always false. Check and print code can be removed.
Standard code optimizations (iii)

- copy propagations + dead code elimination
  - x = t3
  - a[t2] = t5
  - a[t4] = x
  - goto b2

- Elimination of copy propagation
  - x = t3
  - a[t2] = t5
  - a[t4] = t3
  - goto b5

- Elimination of dead code
  - a[t2] = t5
  - a[t4] = t3
  - goto b5
Loop optimisation

Loops are an important place where optimisations may occur. Clearly we try to reduce the number of instructions inside the loop! One option is to use code motion (and maintain semantics).

This transformation takes out of the loop any expressions that have the same evaluation independent of the number of times the loop executes (loop-invariant computation) and places it before the loop.

e.g. while (i <= (limit-1)) .... limit – 1 can be computed before the loop !! i <= t where t = limit-1