THEORY OF COMPUTATION AND COMPILERS

Unit - V

Object Code Generation| Error Recovery

Object Code Generation

- Object code forms
- Machine dependent code optimization (Peephole Optimization)
- Register allocation and assignment
- Generic code generation algorithms

Error Recovery

- Various errors in phases and recovery of errors in compilation
- Introduction to tools of compiler

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Generic Code Generation Algorithms

A Simple Code Generator

- Consider each statement
- Remember if operands are in registers

Register descriptor:

- Keep track of what is currently in each register.
- Initially all the registers are empty.

Address descriptor:

- Keep track of location where current value of the name can be found at runtime.
- The location might be a register, stack, memory-address or a set of those.

A Simple Code Generator

A Code-Generation Algorithm:

- The code-generation algorithm takes as input a sequence of three-address statements constituting a basic block.
- For each three-address statement of the form `x := y op z` we perform the following actions:

1. Invoke a function *getreg* to determine the location `L` where the result of the computation `y op z` should be stored. `L` will usually be a register, but it could also be a memory location.
2. Consult the address descriptor for `y` to determine `y'` (one of) the current location (s) of `y`. Prefer the register for `y'` if the value of `y` is currently both in memory and a register.

   If the value of `y` is not already in `L`, generate the instruction. `MOV y', L` to place a copy of `y` in `L`.

A Simple Code Generator

A Code-Generation Algorithm:

3. Generate the instruction `op z', L` where `z'` is current location of `z`. Again prefer a register for `z`. Update address descriptor of `x` to indicate `x` is in `L`.

   If `L` is a register, update its descriptor to indicate that it contains the value of `x`, and remove `x` from all other register descriptors.

4. If the current value of `y` and/or `z` have no next use and are not live (dead) on exit from the block, and are in registers, alter(change) the register descriptor to indicate that, after execution of `x := y op z`, those registers no longer will contain `y` and/or `z` , respectively.

A Simple Code Generator

The function *getreg*:

The function ***getreg*** returns the location L to hold the value of x for the assignment x := y op z.

Implementation of the function *getreg*:

1. If y is in register (that holds no other values) and y is not live and has no next use after x:= y op z, then return register of y for L.
2. Failing (1), return an empty register for L if there is one.
3. Failing (2), if x has a next use in the block or op is a operator, such as indexing that requires a register, find an occupied register R, store the value of R into a memory location M (by MOV R, M) and use it.

A Simple Code Generator

Implementation of the function *getreg*: cont'd

4. If x is not used in the block or no suitable occupied register can be found, select the memory location of x as L.

Code-Generation Algorithm: Example

Example: The Assignment d := (a - b) + (a - c) + (a - c) might be translated into the following three-address code sequence

t := a - b

u := a - c

v : = t + u

d := v + u

with d live at the end.

Code-Generation Algorithm: Example

The Code-sequence is given below:

Statements	Code Generated	Register Descriptor	Address Descriptor	Cost
-	-	Registers Empty	-	-
t:= a - b	MOV a, R_0 SUB b, R_0	R_0 contains t	t in R_0	2 2
u := a - c	MOV a, R_1 SUB c, R_1	R_0 contains t R_1 contains u	t in R_0 u in R1	2 2
v := t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R1 v in R_0	1
d := v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory	1 2
			Total Cost :	**12**

Generic code generation algorithms

Generating code for other types of statements:

Ex1: The code sequences generated for the indexed assignment statements

`a := b[i]`

`a[i] := b`, assuming `b` is statically allocated

Statement	`i` in Register R_i		`i` in memory M_i		`i` in stack	
	Code	**Cost**	**Code**	**Cost**	**Code**	**Cost**
`a := b[i]`	`MOV b(`R_i`), R`	2	`MOV `M_i`, R` `MOV b(R), R`	4	`MOV `S_i`(A), R` `MOV b(R), R`	4
`a[i]:= b`	`MOV b, a(`R_i`)`	3	`MOV `M_i`, R` `MOV b, a(R)`	5	`MOV `S_i`(A), R` `MOV b, a(R)`	5

Generic code generation algorithms

Generating code for other types of statements:

Ex2: The code sequences for pointer assignments

```
a := *p

*p := a
```

here, the current location of `p` determines the code sequence

Statement	**p** in Register R_p		**p** in memory M_p		**p** in stack	
	Code	**Cost**	**Code**	**Cost**	**Code**	**Cost**
`a := *p`	`MOV *`R_p`, a`	2	`MOV` M_p`, R` `MOV *R, R`	3	`MOV` S_p`(A), R` `MOV *R, R`	3
`*`p_i `:= a`	`MOV a, *` R_p	2	`MOV` M_p`, R` `MOV a, *R`	4	`MOV a, R` `MOV R, *`S_p`(A)`	4

Generic code generation algorithms

Conditional statements:

Machines implement conditional jumps in one of two ways.

1. One way is to branch if the value of a designated register meets one of the following six conditions:

    i. Negative

    ii. Zero

    iii. Positive

    iv. Non Negative

    v. Non Zero

    vi. Non Positive

    On such a machine a three-address statement such as `if x<y goto z` can be implemented by subtracting `y` from `x` in register `R`, and then jumping to `z` if the value in register `R` is negative.

Generic code generation algorithms

Conditional statements:

2. A second approach, common to many machines, uses a set of condition codes to indicate whether the last quantity computed or loaded into a register is negative, zero or positive. Often a compare instruction (`CMP` in our machine) has the desirable property that it sets the condition code without actually computing a value. That is, `cmp x, y` sets the condition code to positive `if x>y` and so on. A conditional-jump machine instruction makes the jump if a designated condition `<`, `=`, `>`, `<=`, `#`, or `>=` is met.

   We use the instruction `CJ <= z` to mean "jump to `z` if the condition code is negative or zero".

Generic code generation algorithms

Conditional statements:

For example: `if x < y goto z` cab be implemented by

```
CMP x, y
CJ < z
```

Ex: Code-Sequence for the following three-address statements:

```
x := y + z
if x < 0 goto z
```

by

```
MOV  y, R0
ADD  z, R0
MOV  R0, x
CJ < z
```

if we were aware that the condition code was determined by `x` after `ADD z, R0`

Machine dependent code optimization : Peephole Optimization

Peephole Optimization

- A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs.
- The quality of such target code can be improved by applying "optimizing' transformations to the target program.
- The term "optimizing" is some what misleading because there is no guarantee that the resulting code is optimal under any mathematical measure.
- Many simple transformations can significantly improve the running time or space requirement of the target program, so it is important to know what kinds of transformations are useful in practice.

Peephole Optimization

- A simple but effective technique for locally improving the target code is *Peephole Optimization*, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the *peephole*) and replacing these instructions by a shorter or faster sequence, whenever possible.
- We consider *peephole optimization* as a technique for improving the quality of the target code, the technique can also be applied directly after intermediate code generation to improve the intermediate representation.
- The *peephole* is a small, moving window on the target program. The code in the *peephole* need not be contiguous, although some implementations do require this. It is characteristic of *peephole optimization* that each improvement may spawn opportunities for additional improvements.

Peephole Optimization

Program Transformations that are characteristics of peephole optimizations:

[Techniques of people optimization]

1. Redundant-instruction elimination
2. Flow-of-control optimizations
3. Algebraic Simplifications
4. Use of machine idioms

Peephole Optimization

Techniques of people optimization

1. Redundant-instruction elimination

i. Redundant Loads and stores:

If we see the instruction sequence

(1) `MOV R0,a`

(2) `MOV a, R0`

we can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of '`a`' is already in register `R0`.

Note that if (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Put another way, (1) and (2) have to be in the same basic block for this transformation to be safe.

Peephole Optimization

Techniques of people optimization

1. Redundant-instruction elimination

ii. Unreachable code:

- The removal of unreachable instructions.
- An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.

Ex: For debugging purpose, a large program may have within it certain segments that are executed only if a variable `debug` is `1`.

In `C` language, the source code might look like:

```
# define debug 0
---
if (debug) {
            print debugging information
           }
```

Peephole Optimization

Techniques of people optimization

1. Redundant-instruction elimination

ii. Unreachable code:

Ex: cont'd.

In the intermediate representation the if-statement may be translated as:

```
if debug =1 goto L1
goto L2
L1: Print debugging information
L2:
```

Peephole Optimization

Techniques of people optimization

1. Redundant-instruction elimination

ii. Unreachable code:

Ex: cont'd.

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of `debug` (in the above), can be replaced by:

```
if debug ≠ 1 goto L2
     Print debugging information
L2:
```

Peephole Optimization

Techniques of people optimization

1. Redundant-instruction elimination

ii. Unreachable code:

Ex: cont'd.

Now, since `debug` is set `0` at the beginning of the program, constant propagation should replace the above by:

```
if 0 ≠ 1 goto L2
        Print debugging information
L2:
```

Peephole Optimization

Techniques of people optimization

2. Flow of Control Optimizations:

The intermediate code generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.

we can replace the jump sequence

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

by the sequence

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

Peephole Optimization

Techniques of people optimization

2. Flow of Control Optimizations:

If there are now no jumps to L_1, then it may be possible to eliminate the statement L_1: goto L_2 provided it is preceded by an unconditional jump.

Similarly, the sequence

```
if a < b goto L1
...
L1: goto L2
```

Can be replaced by

```
if a < b goto L2
L1: goto L2
```

Peephole Optimization

Techniques of people optimization

2. Flow of Control Optimizations:

Finally, suppose there is only one jump to L_1 and L_1 is replaced by an unconditional goto. Then the sequence

```
    goto L1
    ...
L1: if a < b goto L2
L3:
```

May be replaced by

```
    if a < b goto L2
    goto L3
    ...
L3:
```

Peephole Optimization

Techniques of people optimization

3. Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization.
- However, only a few algebraic identities occur frequently enough that it is worth considering implementing them.

Example: Statements such as

```
x := x + 0
```

or

```
x := x * 1
```

are often produced by straight forward intermediate-code generation algorithms, and they can be eliminated easily through peephole optimization.

Peephole Optimization

Techniques of people optimization

3. Algebraic Simplification:

i. Reduction in strength

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.
- Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For ex: x^2 is invariably cheaper to implement as `x * x` than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift floating-point division by a constant can be implemented (approximately) as multiplication by a constant, which may be cheaper.

Peephole Optimization

Techniques of people optimization

4. Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specification operations efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.

For ex:

- Some machines have `auto-increment` and `auto-decrement` addressing modes.
- These add or subtract one from an operand before or after using its value.
- The use of those modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing.
- These modes can also be used in code for statements like `i := i + 1`