RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Compiler Design**

Subject Code: **CS-603**

Semester: **6th**

UNIT- IV:

Intermediate code generation: Declarations, Assignment statements, Boolean expressions, Case statements, Back patching, Procedure calls Code Generation: Issues in the design of code generator, Basic block and flow graphs, Register allocation and assignment, DAG representation of basic blocks, peephole optimization, generating code from DAG.

---

1. Intermediate Code Generation

The intermediate code is useful representation when compilers are designed as two pass system, i.e. as front end and back end. The source program is made source language independent by representing it in intermediate form, so that the back end is filtered from source language dependence. The intermediate code can be generated by modifying the syntax-directed translation rules to represent the program in intermediate form. This phase of intermediate code generation comes after semantic analysis and before code optimization.

Benefits of intermediate code

- Intermediate code makes target code generation easier
- It helps in retargeting, that is, creating more and more compilers for the same source language but for different machines.
- As intermediate code is machine independent, it helps in machine-independent code optimization.

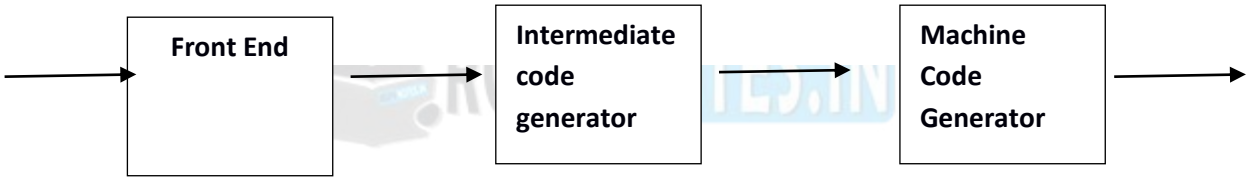

Figure 4.1:Intermediate code generation phase

A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

1.1 Intermediate code can be represented in the following four ways.

- Syntax trees
- Directed acyclic graph(DAG)
- Postfix notation
- Three address code

1.1.1 Syntax Trees

A syntax tree is a graphical representation of the source program. Here the node represents an operator and children of the node represent operands. It is a hierarchical structure that can be constructed by syntax rules. The target code can be generated by traversing the tree in post order form. For instance, consider an assignment statement a = b* – (c – d) + b* – (c – d) when represented using the syntax tree

The tree for the statement a = b* – (c – d) + b* – (c – d) is constructed by creating the nodes in the following order.

p_1 = mkleaf(id, c)
p_2 = mkleaf(id, d)
p_3 = mknode('–', p_1, p_2)
p_4 = mknode('U', p_3, NULL)

p_5 = mkleaf(id, b)

Production	Semantic Rule
S → id := E	S.nptr := mknode('assign', mkleaf(id, id.place), E.nptr)
E → E1 + E2	E.nptr := mknode('+', E1.nptr ,E2.nptr)
E → E1 * E2	E.nptr := mknode('* ', E1.nptr ,E2.nptr)
E → − E1	E.nptr := mkunode('uminus', E1.nptr)
E → (E1)	E.nptr := E1.nptr
E → id	E.nptr := mkleaf(id, id.place)

Table 4.1: Production in Syntax tree

1.1.2 Directed Acyclic Graph (DAG)

The tree that shows the same information with identified common sub-expression is called Directed Acyclic Graph (DAG). On examining the above example, it is observed that there are some nodes that are unnecessarily created. To avoid extra nodes these functions can be modified to check the existence of similar node before creating it. If a node exists then the pointer to it is returned instead of creating a new node. This creates a DAG, which reduces the space and time requirement.

1.1.3 Postfix Notation

Postfix notation is a linear representation of a syntax tree. This can be written by traversing the tree in the post order form. The edges in a syntax tree do not appear explicitly in postfix notation; only the nodes are listed. The order is followed by listing the parent node immediately after listing its left sub tree and its right sub tree. In postfix notation, the operators are placed after the operands.

1.1.4 Three Address Code

Three address code is a linear representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. Three address code is a sequence of statements of the form A = B *OP* C where A, B and C are the names of variables, constants or the temporary variables generated by the compiler. *OP* is any arithmetic operation or logical operation applied on the operands B and C. The name reflects that there are at most three variables where two are operands and one is for the result. In three address statement, only one operator is permitted; if the expression is large, then break it into a sequence of sub expressions using the BODMAS rules of arithmetic and store the intermediate results in newly created temporary variables. For example, consider the expression a + b * c;
this expression is expressed as follows:

T_1 = b * c
T_2 = a + T_1

Here T_1 and T_2 are compiler-generated temporary names. This simple representation of a complex expression in three address code makes the task of optimizer and code generator simple. It is also easy to rearrange the sequence for efficient code generation. Three address code for the statement a = b* – (c – d) + b* – (c – d) is as follows:

T_1 = c – d
T_2 = –T_1
T_3 = b * T_2
T_4 = c – d
T_5 = –T_4
T_6 = b * T_5

$T_7 = T_3 + T_6$
$a = T_7$

The code can also be written for DAG as follows:

$T_1 = c - d$
$T_2 = -T_1$
$T_3 = b * T_2$
$T_4 = T_3 + T_3$
$a = T_4$

Types of Three Address Statements

For expressing the different programming constructs, the three address statements can be written in different standard formats and these formats are used based on the expression. Some of them are as follows:

- *Assignment statements with binary operator.* They are of the form A := B op C where op is a binary arithmetic or logical operation.
- *Assignment statements with unary operator.* They are of the form A: = op B where op is a unary operation like unary plus, unary minus, shift, etc.
- *Copy statements.* They are of the form A: = B where the value of B is assigned to variable A.
- *Unconditional Jumps* such as goto L**:** The label **L** with three address statement is the next statement number to be executed.
- *Conditional Jumps* such as if X relop Y goto L**.** If the condition is satisfied, then this instruction applies a relational operator (<=,>=,<,>) to X and Y and executes the statement with label L else the statement following if X relop Y goto L is executed.
- *Functional calls:* The functional calls are written as a sequence of param A, call fun,n, and returnB statements, where A indicates one of the input argument in *n* arguments to be passed to the function fun that returns B. The return statement is optional.

2. Declarations

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an
activation record.

Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say offset, can keep track of the next available relative address.

In the translation scheme shown below:

- Non-terminal P generates a sequence of declarations of the form id : T.
- Before the first declaration is considered, offset is set to 0. As each new name is seen ,that name is entered in the symbol table with offset equal to the current value of offset,and offset is incremented by the width of the data object denoted by that name.
- The procedure enter(name, type, offset) creates a symbol-table entry for name, gives itstype and relative address offset in its data area.
- Attribute type represents a type expression constructed from the basic types integer and real by applying the type constructors pointer and array. If type expressions are represented by graphs, then attribute type might be a pointer to the node representing a type expression.
- The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4.

T $\rightarrow$ array [num] of T_1

T.type = array(num.val, T_1 .type)
T.width = num.val x T_1 .width
T $\rightarrow \uparrow T_1$
T.type = pointer(T_1 .type)
T.width = 4
This is the continuation of the example in the previous slide
Keeping Track of Scope Information:
When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:
P ->D
D ->D ; D | id : T | proc id ; D ; S
One possible implementation of a symbol table is a linked list of entries for names.
A new symbol table is created when a procedure declaration D proc id D1;S is seen, and entries for the declarations in D1 are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure enter is told which symbol table to make an entry in.
For example, consider the symbol tables for procedures readarray, exchange, and quicksort pointing back to that for the containing procedure sort, consisting of the entire program. Since partition is declared within quicksort, its table points to that of quicksort.

Whenever a procedure declaration Dproc id ; D1 ; S is processed, a new symbol table with a pointer to the symbol table of the enclosing procedure in its header is created and the entries for declarations in D1 are created in the new symbol table. The name represented by id is local to the enclosing procedure and is hence entered into the symbol table of the enclosing procedure.
Example

Program sort:

```
var a: array[1..n] of integer;
X:integer;
Procedure readarray;
var i: integer
…….
Procedure exchange(I,j:integers);
…….
Procedure quicksort(m,n : integer);
Var k,v:integer;

Function partition(x,y:integer):integer;
Var I,j:integer;
……
…….
Begin{main}
…….
end
```

For the above procedures, entries for x, a and b *quicksort*are created in the symbol table of *sort.* A pointer pointing to the symbol table of quicksort is also entered. Similarly, entries for k,v and partition are created

in the symbol table of quicksort. The headers of the symbol tables of quicksort and partition have pointers pointing to sort and quicksort respectively

This structure follows from the example in the previous slide.
Creating symbol table
mktable (previous)
create a new symbol table and return a pointer to the new table. The argument previous points to the enclosing procedure
enter (table, name, type, offset)
creates a new entry
addwidth (table, width)
records cumulative width of all the entries in a table
enterproc (table, name, newtable)
creates a new entry for procedure name. newtable points to the symbol table of the new procedure

The following operations are designed :

- mktable(previous): creates a new symbol table and returns a pointer to this table. *previous* is pointer to the symbol table of parent procedure.
- enter(table,name,type,offset): creates a new entry for *name* in the symbol table pointed to by *table* .
- addwidth(table,width): records cumulative width of entries of a table in its header.
- enterproc(table,name ,newtable): creates an entry for procedure *name* in the symbol table pointed to by *table* . *newtable* is a pointer to symbol table for *name* .

Creating symbol table.

P ⟶	{t=mktable(nil);
	push(t,tblptr);
	push(0,offset)}
D	
	{addwidth(top(tblptr),top(offset));
	pop(tblptr);
	pop(offset)}
D⟶ D ;	D

Table 4.2:Symbol table

The symbol tables are created using two stacks: tblptrto hold pointers to symbol tables of the enclosing procedures and *offset* whose top element is the next available relative address for a local of the current procedure. Declarations in nested procedures can be processed by the syntax directed definitions given below. Note that they are basically same as those given above but we have separately dealt with the epsilon productions.

3. Assignment Statements
Suppose that the context in which an assignment appears is given by the following grammar.
P ⟶M D
M ⟶ε
D ⟶D ; D | id : T | proc id ; N D ; S
N ⟶ε

Non-terminal P becomes the new start symbol when these productions are added to those in thetranslation scheme shown below.
Translation scheme to produce three-address code for assignments
S ⟶id : = E { p : = lookup (id.name);
if p ≠ nil then
emit(p ' : =' E.place)
else error }
E ⟶E1 + E2 { E.place : = newtemp;
emit(E.place ': =' E1.place ' + ' E2.place) }
E ⟶E1 * E2 { E.place : = newtemp;
emit(E.place ': =' E1.place ' * ' E2.place) }
E ⟶- E1 { E.place : = newtemp;
emit (E.place ': =' 'uminus' E1.place) }
E ⟶(E1) { E.place : = E1.place }
E ⟶id { p : = lookup (id.name);
if p ≠ nil then
E.place : = p
else error }
Reusing Temporary Names
The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.
Temporaries can be reused by changing newtemp. The code generated by the rules for E.E1 + E2 has the general form:
evaluate E1 into t1
evaluate E2 into t2
t : = t1 + t2
The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.
Keep a count c , initialized to zero. Whenever a temporary name is used as an operand,
decrement c by 1. Whenever a new temporary name is generated, use $c and increase c
by 1.

3.1 Addressing Array Elements
Arrays are stored in a block of consecutive locations
assume width of each element is w
ith element of array A begins in location base + (i - low) x w where base is relative address of A[low]the expression is equivalent to
i x w + (base-low x w)
⟶i x w + const
Elements of an array are stored in a block of consecutive locations. For a single dimensional array, if low is the lower bound of the index and base is the relative address of the storage allocated to the array i.e., the relative address of A[low], then the i th Elements of an array are stored in a block of consecutive locations
For a single dimensional array, if low is the lower bound of the index and base is the relative address of the storage allocated to the array i.e., the relative address of A[low], then the i th elements begins at the location: base + (I - low)* w. This expression can be reorganized as i*w + (base -low*w). The sub-expression base-low*wis calculated and stored in the symbol table at compile time when the array declaration is processed, so that the relative address of A[i] can be obtained by just adding ***i****wto it.
3.2 2-dimensional array
storage can be either row major or column major
in case of 2-D array stored in row major form address of $A[i_1 , i_2]$ can be calculated as
$base + ((i_1 - low_1) x n_2 + i_2 - low_2) x w$
where $n_2 = high_2 - low_2 + 1$

rewriting the expression gives
$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$
$((i_1 \times n_2) + i_2) \times w + constant$
this can be generalized for $A[i_1, i_2, .., i_k]$
Similarly, for a row major two dimensional array the address of A[i][j] can be calculated by the formula :
base + ((i-low_i)*n2 +j - low_j)*w where low_i and low_j are lower values of I and j and n2 is number of values j can take i.e. n2 = high2 - low2 + 1.
This can again be written as :
((i * n2) + j) *w + (base - ((low_i *n2) + low_j) * w) and the second term can be calculated at compile time.
In the same manner, the expression for the location of an element in column major two-dimensional array can be obtained. This addressing can be generalized to multidimensional arrays.

4. Boolean Expressions
Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.
Boolean expressions are composed of the boolean operators (and, or, and not) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E1 relop E2, where E1 and E2 are arithmetic expressions.
Here we consider boolean expressions generated by the following grammar:
E .E or E | E and E | not E | (E) | id relop id | true | false
Methods of Translating Boolean Expressions:
There are two principal methods of representing the value of a boolean expression. They are:
To encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
To implement boolean expressions by flow of control, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

4.1 Numerical representation
a or b and not c
t_1 = not c
t_2 = b and t_1
t_3 = a or t_2
relational expression a < b is equivalent to if a < b then 1 else 0
1. if a < b goto 4.
2. t = 0
3. goto 5
4. t = 1

Consider the implementation of Boolean expressions using 1 to denote true and 0 to denote false. Expressions are evaluated in a manner similar to arithmetic expressions.
For example, the three address code for a or b and not c is:
t1 = not c
t2 = b and t1
t3 = a or t2

4.2 Syntax directed translation of boolean expressions
$E \longrightarrow E_1$ or E_2
E.place := newtmp
emit(E.place ':=' E_1 .place 'or' E_2 .place)

E⟶ E_1 and E_2
E.place:= newtmp
emit(E.place ':=' E_1 .place 'and' E_2 .place)
E ⟶not E_1
E.place := newtmp
emit(E.place ':=' 'not' E_1 .place)
E ⟶(E_1) E.place = E_1 .place
The above written translation scheme produces three address code for Boolean expressions. It is continued to the next page**.**

4.3 Syntax directed translation of boolean expressions
E⟶ id1 relop id2
E.place := newtmp
emit(if id1.place relop id2.place goto nextstat+3)
emit(E.place = 0) emit(goto nextstat+2)
emit(E.place = 1)
E ⟶true
E.place := newtmp
emit(E.place = '1')
E ⟶false
E.place := newtmp
emit(E.place = '0')
In the above scheme, nextstat gives the index of the next three address code in the output sequence and emit increments nextstat after producing each three address statement.

Example:
Code for a < b or c < d and e < f

100: if a < b goto 103	if e < f goto 111
101: $t_1 = 0$	109: $t_3 = 0$
102: goto 104	110: goto 112
103: $t_1 = 1$	111: $t_3 = 1$
104:	112:
if c < d goto 107	$t_4 = t_2$ and t_3
105: $t_2 = 0$	113: $t_5 = t_1$ or t_4
106: goto 108	
107: $t_2 = 1$	
108:	

Table 4.3:Three address code for above example

A relational expression a < b is equivalent to the conditional statement if a < b then 1 else 0 and three address code for this expression is:
100: if a < b goto 103.
101: t = 0
102: goto 104
103: t = 1
104:
It is continued from 104 in the same manner as the above written block.

5. Case Statement

switch expression
begin
case value: statement
case value: statement
..
case value: statement
default: statement
end

evaluate the expression
find which value in the list of cases is the same as the value of the expression

Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression execute the statement associated with the value found.
There is a selector expression, which is to be evaluated, followed by ***n*** constant values that the expression can take. This may also include a *default* value which always matches the expression if no other value does. The intended translation of a switch case code to:

- evaluate the expression
- find which value in the list of cases is the same as the value of the expression.
- Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression. executethe statement associated with the value found
- Most machines provide instruction in hardware such that case instruction can be implemented easily. So, case is treated differently and not as a combination of if-then statements.

Translation

code to evaluate E into t	code to evaluate E into t
if t <> V1 goto L1	goto test
code for S1	L1: code for S1
goto next	goto next
L1 if t <> V2 goto L2	L2: code for S2
code for S2	goto next
goto next	..
L2: ..	Ln: code for Sn
Ln-2 if t <> Vn-l goto Ln-l	goto next

code for Sn-l	test: if t = V1 goto L1
goto next	if t = V2 goto L2
Ln-1: code for Sn	..
next:	if t = Vn-1 goto Ln-1
	goto Ln
	next:

Table 4.4: Translation table

Efficient for n-way branch

There are two ways of implementing switch-case statements, both given above. The above two implementations are equivalent except that in the first case all the jumps are short jumps while in the second case they are long jumps. However, many machines provide the n-way branch which is a hardware instruction. Exploiting this instruction is much easier in the second implementation while it is almost impossible in the first one. So, if hardware has this instruction the second method is much more efficient.

6. Backpatching

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.

To manipulate lists of labels, we use three functions:

- makelist(i) creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.
- merge(p1,p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.
- backpatch(p,i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions

E $\rightarrow$ E_1 or M E_2
| E_1 and M E_2
| not E_1
| (E_1)
| id $_1$ relop id $_2$
| true
| false M ? ε

Synthesized attributes truelist and falselist of non-terminal E are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by
E.truelist and E.falselist.
Consider production E ⟶E1 and M E2. If E1 is false, then E is also false, so the statements on E1.falselist become part of E.falselist. If E1 is true, then we must next test E2, so the target for the statements E1.truelist must be the beginning of the code generated for E2. This target is obtained using marker non-terminal M.
Attribute M.quad records the number of the first statement of E2.code. With the production M ⟶ε we associate the semantic action

{ M.quad : = nextquad }

The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched onto the E1.truelist when we have seen the remainder of the production E ⟶E1 and

M E2. The translation scheme is as follows:

(1) E ⟶E1 or M E2 { backpatch (E1.falselist, M.quad);
E.truelist : = merge(E1.truelist, E2.truelist);
E.falselist : = E2.falselist }

(2) E ⟶E1 and M E2 { backpatch (E1.truelist, M.quad);
E.truelist : = E2.truelist;
E.falselist : = merge(E1.falselist, E2.falselist) }

(3) E ⟶not E1 { E.truelist : = E1.falselist;
E.falselist : = E1.truelist; }

(4) E ⟶(E1) { E.truelist : = E1.truelist;
E.falselist : = E1.falselist; }

(5) E ⟶id1 relop id2 { E.truelist : = makelist (nextquad);
E.falselist : = makelist(nextquad + 1);
emit('if' id1.place relop.op id2.place 'goto_')
emit('goto_') }

(6) E⟶true { E.truelist : = makelist(nextquad);
emit('goto_') }

(7) E⟶false { E.falselist : = makelist(nextquad);
emit('goto_') }
(8) M ⟶ε { M.quad : = nextquad }

7. Procedure Calls
The procedure is such an important and frequently used programming construct that it isimperative for a compiler to generate good code for procedure calls and returns. The run-timeroutines that handle procedure argument passing, calls and returns are part of the run-time support package.
Let us consider a grammar for a simple procedure call statement
(1) S ⟶call id (Elist)
(2) Elist⟶Elist , E
(3) Elist ⟶E

Calling Sequences:
The translation for a call includes a calling sequence, a sequence of actions taken on entryto and exit from each procedure. The falling are the actions that take place in a calling sequence:

When a procedure call occurs, space must be allocated for the activation record of the called procedure.
The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
Environment pointers must be established to enable the called procedure to access data in enclosing blocks.
The state of the calling procedure must be saved so it can resume execution after the call.
Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.

Finally a jump to the beginning of the code for the called procedure must be generated. For example, consider the following syntax-directed translation.
(1) S⟶call id (Elist)
{ for each item p on queue do
emit (‘ param’ p);
emit (‘call’ id.place) }
(2) Elist ⟶Elist , E
{ append E.place to the end of queue }
(3) Elist ⟶E
{ initialize queue to contain only E.place }

Here, the code for S is the code for Elist, which evaluates the arguments, followed by a param p statement for each argument, followed by a call statement. Queue is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.

8. CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediaterepresentation of the source program and produces as output an equivalent target program. Thecode generation techniques presented below can be used whether or not an optimizing phaseoccurs before code generation.

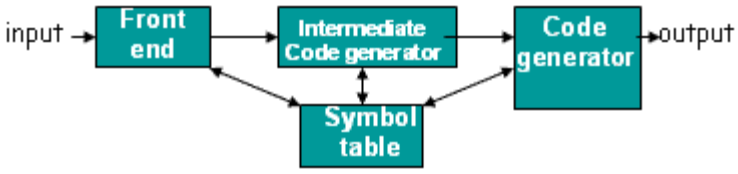

Figure 4.2: Code Generation in Compiler

8.1 Issues In The Design Of A Code Generator

The following issues arise during the code generation phase:
1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation

6. Evaluation order

Input to code generator:
The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be:
a. Linear representation such as postfix notation
b. Three address representation such as quadruples
c. Virtual machine representation such as stack machine code
d. Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

Target program:

The output of the code generator is the target program. The output may be:
a. Absolute machine language
It can be placed in a fixed memory location and can be executed immediately.
b. Reloadable machine language
It allows subprograms to be compiled separately.
c. Assembly language
Code generation is made easier.

Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions.

Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.

Register allocation

Instructions involving register operands are shorter and faster than those involving operands in emery.
The use of registers is subdivided into two sub problems:
Register allocation – the set of variables that will reside in registers at a point in the program is selected.
Register assignment – the specific register that a variable will reside in is picked.

Certain machine requires even-odd register pairs for some operands and results.
For example, consider the division instruction of the form: **D x, y**

Where, x – dividend even register in even/odd register pair
y – Divisor even register holds the remainder odd register holds the quotient

Evaluation order
The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

9. Basic Blocks And Flow Graphs
9.1 Basic Blocks
A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

The following sequence of three-address statements forms a basic block:
t1: = a * a
t2: = a * b
t3: = 2 * t2
t4: = t1 + t3
t5: = b * b
t6: = t4 + t5

Basic Block Construction:

Algorithm: Partition into basic blocks
Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:
a. The first statement is a leader.
b. Any statement that is the target of a conditional or unconditional goto is a leader.
c. Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not Including the next leader or the end of the program.

Consider the following source code for dot product of two vectors a and b of length 20

```
begin
prod :=0;
i:=1;
do begin
prod :=prod+ a[i] * b[i];
i :=i+1;
end
while i <= 20
end
```

The three-address code for the above source program is given as:

```
(1) prod := 0
(2) i := 1
(3) t1 := 4* i
(4) t2 := a[t1] /*compute a[i] */
(5) t3 := 4* i
(6) t4 := b[t3] /*compute b[i] */
(7) t5 := t2*t4
(8) t6 := prod+t5
(9) prod := t6
(10) t7 := i+1
(11) i := t7
(12) if i<=20 goto (3)
```

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without changing the set ofexpressions computed by the block. Two important classes of transformation are:

- Structure-preserving transformations
- Algebraic transformations

1. Structure preserving transformations:

a) Common sub expression elimination:

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

Since the second and fourth expressions compute the same expression, the basic block can betransformed as above.

b) Dead-code elimination:
Suppose x is dead, that is, never subsequently used, at the point where the statement x : =y + z appears in a basic block. Then this statement may be safely removed without changingthe value of the basic block.
c) Renaming temporary variables:
A statement t : = b + c (t is a temporary) can be changed to u : = b + c (u is a new temporary) and all uses of this instance of t can be changed to u without changing the value of the basic block.
Such a block is called a normal-form block.
d) Interchange of statements:
Suppose a block has the following two adjacent statements:
t1: = b + c
t2: = x + y
We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2.
2. Algebraic transformations:
Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:
i) x: = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set ofexpressions it computes.
ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

9.2 Flow Graphs
Flow graph is a directed graph containing the flow-of-control information for the set ofbasic blocks making up a program.

The nodes of the flow graph are basic blocks. It has a distinguished initial node.
Loops
A loop is a collection of nodes in a flow graph such that
1. All nodes in the collection are strongly connected.
2. The collection of nodes has a unique entry.

A loop that contains no other loops is called an inner loop.

9.3 Next-Use Information
If the name in a register is no longer needed, then we remove the name from the registerand the register can be used to store some other names

Input: Basic block B of three-address statements
Output: At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

Method: We start at the last statement of B and scan backwards.
1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to “not live” and “no next use”.
3. In the symbol table, set y and z to “live”, and next-uses of y and z to i.

10. The DAG Representation For Basic Blocks

A DAG for a basic block is a directed acyclic graph with the following labels on nodes:
1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

DAGs are useful data structures for implementing transformations on basic blocks.
It gives a picture of how the value computed by a statement is used in subsequent statements.
It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block
Output: A DAG for the basic block containing the following information:
1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.
Case (i) x : = y OP z
Case (ii) x : = OP y
Case (iii) x : = y

Method:
Step 1: If y is undefined then create node(y).
If z is undefined, create node(z) for case(i).
Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let n be this node.
For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.
For case(iii), node n will be node(y).
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Application of DAGs:
1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

11. Peephole Optimization
Target code often contains redundant instructions and suboptimal constructs.
Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence peephole is a small moving window on the target systems.
A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. 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. 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.

Peephole optimization examples.
Redundant loads and stores
Consider the code sequence
Move R_0 , a Move a, R_0
Instruction 2 can always be removed if it does not have a label.
Now, we will give some examples of program transformations that are characteristic of peephole optimization: Redundant loads and stores: If we see the instruction sequence
Move R_0 , a
Move a, R_0
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 is (2) has a label, we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).
Peephole optimization examples.

Unreachable code

Consider the following code

```
# define debug 0
If(debug) {
Print debugging info
}
This may be translated as
 If debug =1 goto L1
Goto L2

L1:print debugging info
L2:

Eliminate jump over iumps

If debug<> 1 goto L2
Print debugging information
L2:
```

Another opportunity for peephole optimization is the removal of unreachable instructions.

12. Generating Code from DAGs

The advantage of generating code for a basic block from its dag representation is that,from a dag we can easily see how to rearrange the order of the final computation sequence thanwe can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

t1: = a + b
t2: = c + d
t3: = e – t2
t4: = t1 – t3

Generated code sequence for basic block:

```
MOV a , R0
ADD b , R0
MOV c , R1
ADD d , R1
MOV R0 , t1
MOV e , R0
SUB R1 , R0
MOV t1 , R1
SUB R0 , R1
MOV R1 , t4
```

Rearranged basic block:

Now t1 occurs immediately before t4.

t2: = c + d
t3: = e – t2

t1: = a + b
t4: = t1 – t3

Revised code sequence:
MOV c , R0
ADD d , R0
MOV a , R0
SUB R0 , R1
MOV a , R0
ADD b , R0
SUB R1 , R0
MOV R0 , t4
In this order, two instructions MOV R0 , t1 and MOV t1 , R1 have been saved.

A Heuristic ordering for DAGS
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.
The algorithm shown below produces the ordering in reverse.

Algorithm:
1) while unlisted interior nodes remain do begin
2) select an unlisted node n, all of whose parents have been listed;
3) list n;
4) while the leftmost child m of n has no unlisted parents and is not a leaf do
begin
5) list m;
6) n : = m
end
end

Code generation phase generates the code based on the numbering assigned to each node T. All the registers available are arranged as a stack to maintain the order of the lower register at the top. This makes an assumption that the required number of registers cannot exceed the number of available registers. In some cases, we may need to spill the intermediate result of a node to memory. This algorithm also does not take advantage of the commutative and associative properties of operators to rearrange the expression tree.
It first checks whether the node T is a leaf node; if yes, it generates a load instruction corresponding to it as load top (), T. If the node T is an internal node, then it checks the left *l* and right *r* sub tree for the number assigned. There are three possible values, the number on the right is 0 or greater than or less than the number on the left. If it is 0 then call the generate () function with left sub tree *l* and then generate instruction op top (),r. If the numbering on the left is greater than or equal to right, then call generate () with left sub tree, get new register by popping the top, call generate () with right sub tree, generate new instruction for OP R, top (), and push back the used register on to the stack.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in