Compiler Design Unit 4and5

Compiler Design (SRM Institute of Science and Technology)

MODULE-4 INTERMEDIATE CODE GENERATION

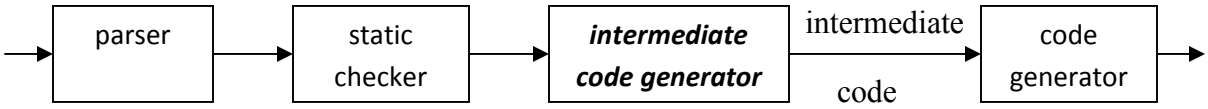
INTRODUCTION

The front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Position of intermediate code generator



INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

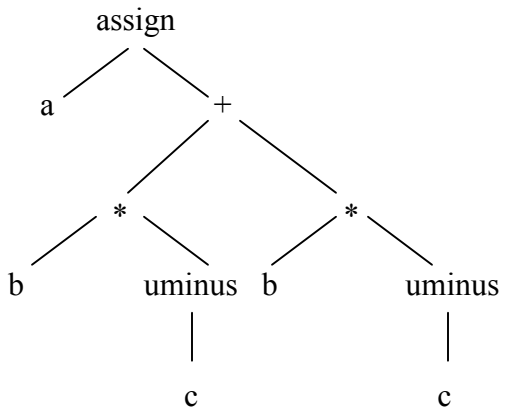
- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

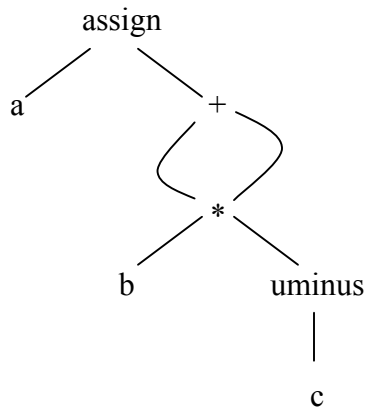
Graphical Representations:

Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A **dag (Directed Acyclic Graph)** gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement **a : = b * - c + b * - c** are as follows:



(a) Syntax tree



(b) Dag

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a : = b * - c + b* - c.

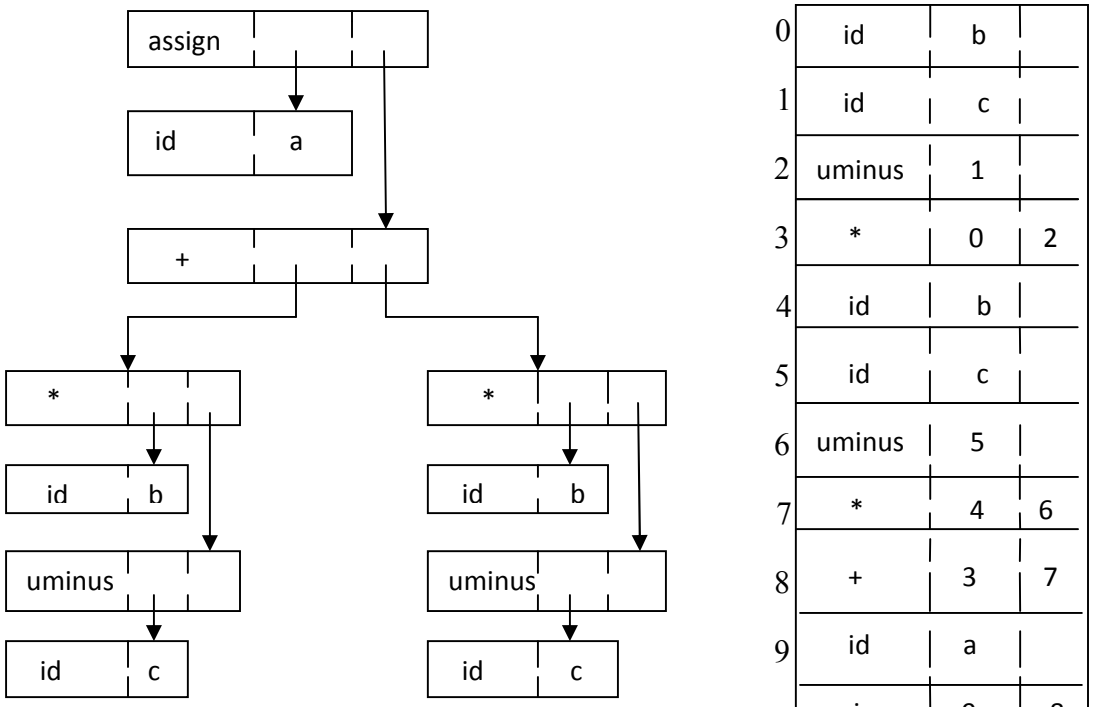
PRODUCTION	SEMANTIC RULE
S → id : = E	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
E → E_1 + E_2	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
E → E_1 * E_2	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)
E → - E_1	E.nptr : = mknode('uminus', E_1.nptr)
E → (E_1)	E.nptr : = E_1.nptr
E → id	E.nptr : = mkleaf(id, id.place)

Syntax-directed definition to produce syntax trees for assignment statements

The token **id** has an attribute *place* that points to the symbol-table entry for the identifier. A symbol-table entry can be found from an attribute **id**.*name*, representing the lexeme associated with that occurrence of **id.** If the lexical analyzer holds all lexemes in a single array of characters, then attribute *name* might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.

Two representations of the syntax tree



(a) (b)

Three-Address Code:

Three-address code is a sequence of statements of the general form

$$x := y \; op \; z$$

where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into a sequence

$$t_1 := y * z$$
$$t_2 := x + t_1$$

where t_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

$t_1 := - c$

$t_2 := b * t_1$

$t_3 := - c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

(a) Code for the syntax tree

$t_1 := -c$

$t_2 := b * t_1$

$t_5 := t_2 + t_2$

$a := t_5$

(b) Code for the dag

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form **x : = y *op* z**, where ***op*** is a binary arithmetic or logical operation.
2. Assignment instructions of the form **x : = *op* y**, where ***op*** is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. *Copy statements* of the form **x : = y** where the value of *y* is assigned to *x*.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as **if *x relop y* goto L**. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation

relop to y. If not, the three-address statement following if *x relop y* goto L is executed next, as in the usual sequence.

6. *param x* and *call p, n* for procedure calls and *return y*, where y representing a returned value is optional. For example,

```
param x1
param x2
  . . .
param xn
call p,n
```

generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$.

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Syntax-Directed Translation into Three-Address Code:

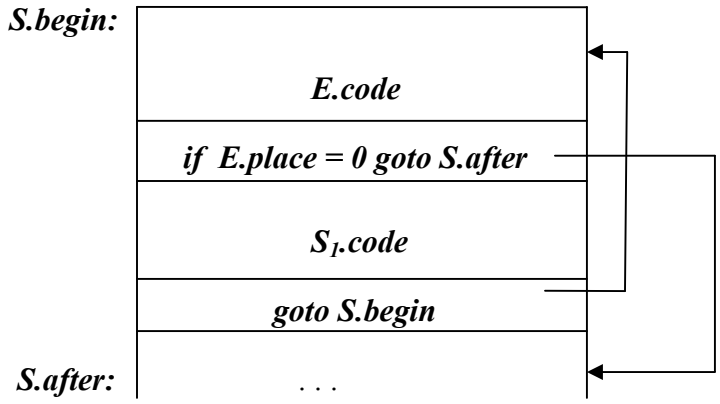
When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example, **id : = *E*** consists of code to evaluate ***E*** into some temporary t, followed by the assignment **id**.*place* : = **t.**

Given input a : = b * - c + b * - c, the three-address code is as shown above. The synthesized attribute *S.code* represents the three-address code for the assignment *S.*
The nonterminal *E* has two attributes :
1. *E.place*, the name that will hold the value of *E* , and
2. *E.code*, the sequence of three-address statements evaluating *E*.

Syntax-directed definition to produce three-address code for assignments

PRODUCTION	SEMANTIC RULES
S → id : = E	***S.code : = E.code || gen(id.place ':=' E.place)***
E → E_1 + E_2	***E.place := newtemp;*** ***E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)***
E → E_1 * E_2	***E.place := newtemp;*** ***E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)***
E → - E_1	***E.place := newtemp;*** ***E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)***
E → (E_1)	***E.place : = E_1.place;*** ***E.code : = E_1.code***
E → id	***E.place : = id.place;*** ***E.code : = ' '***

Semantic rules generating code for a while statement



PRODUCTION	SEMANTIC RULES
S → while *E* do S_1	***S.begin := newlabel;*** ***S.after := newlabel;*** ***S.code := gen(S.begin ':')*** ‖ ***E.code*** ‖ ***gen ('if' E.place '=' '0' 'goto' S.after)***‖ S_1***.code*** ‖ ***gen ('goto' S.begin)*** ‖ ***gen (S.after ':')***

- The function *newtemp* returns a sequence of distinct names $t_1, t_2, \ldots$ in response to successive calls.
- Notation *gen(x ':=' y '+' z)* is used to represent three-address statement x := y + z. Expressions appearing instead of variables like *x, y* and *z* are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- Flow-of–control statements can be added to the language of assignments. The code for ***S* → while *E* do S_1** is generated using new attributes *S.begin* and *S.after* to mark the first statement in the code for *E* and the statement following the code for S, respectively.
- The function *newlabel* returns a new label every time it is called.
- We assume that a non-zero expression represents true; that is when the value of ***E*** becomes zero, control leaves the while statement.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are, ***op, arg1, arg2*** and ***result.***
- The *op* field contains an internal code for the operator. The three-address statement **x : = y op z** is represented by placing *y* in *arg1*, *z* in *arg2* and *x* in *result.*
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: *op, arg1* and *arg2.*
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as *triples*.

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	: =	t_3		a

(a) Quadruples

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Quadruple and triple representation of three-address statements given above

A ternary operation like x[i] : = y requires two entries in the triple structure as shown as below while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

(a) x[i] : = y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

(b) x : = y[i]

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

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:

- Nonterminal 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 its type *type* 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.

Computing the types and relative addresses of declared names

P → D	***{ offset : = 0 }***
D → D ; D	
D → id : T	***{ enter(id.name, T.type, offset);*** ***offset : = offset + T.width }***
T → integer	***{ T.type : = integer;*** ***T.width : = 4 }***
T → real	***{ T.type : = real;*** ***T.width : = 8 }***
T → array [num] of T_1	***{ T.type : = array(num.val, T_1.type);*** ***T.width : = num.val X T_1.width }***
T → ↑ T_1	***{ T.type : = pointer (T_1.type);*** ***T.width : = 4 }***

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:

$P \rightarrow D$

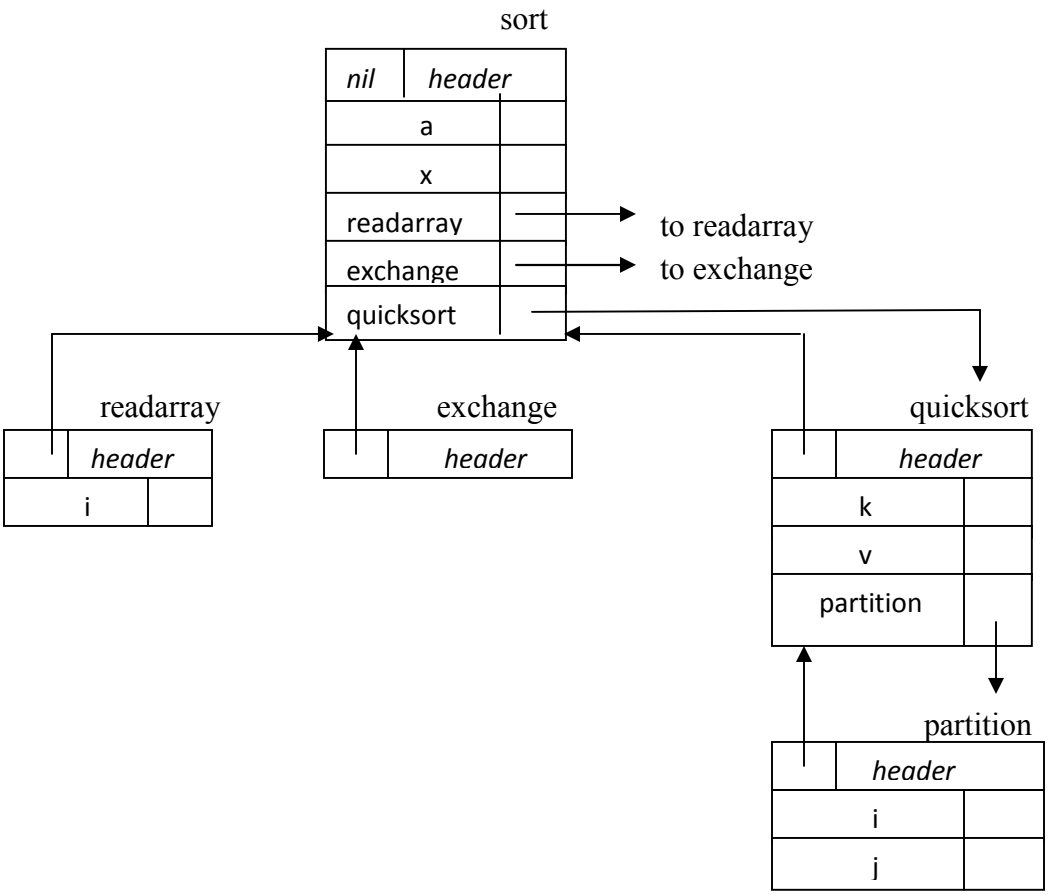
$D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T \mid \textbf{proc id}\ ;\ D\ ;\ S$

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 $D \rightarrow \textbf{proc id}\ D_1;S$ is seen, and entries for the declarations in D_1 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*.

Symbol tables for nested procedures



The semantic rules are defined in terms of the following operations:

1. *mktable(previous)* creates a new symbol table and returns a pointer to the new table. The argument *previous* points to a previously created symbol table, presumably that for the enclosing procedure.

2. *enter(table, name, type, offset)* creates a new entry for name *name* in the symbol table pointed to by *table.* Again, *enter* places type *type* and relative address *offset* in fields within the entry.

3. *addwidth(table, width)* records the cumulative width of all the entries in table in the header associated with this symbol table.

4. *enterproc(table, name, newtable)* creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

Syntax directed translation scheme for nested procedures

P → M D	***{ addwidth (top(tblptr) , top (offset));*** ***pop (tblptr); pop (offset) }***
M → ε	***{ t : = mktable (nil);*** ***push (t,tblptr); push (0,offset) }***
D → D_1 ; D_2	
D → proc id ; N D_1 ; S	***{ t : = top (tblptr);*** ***addwidth (t, top (offset));*** ***pop (tblptr); pop (offset);*** ***enterproc (top (tblptr), id.name, t) }***
D → id : T	***{ enter (top (tblptr), id.name, T.type, top (offset));*** ***top (offset) := top (offset) + T.width }***
N → ε	***{ t := mktable (top (tblptr));*** ***push (t, tblptr); push (0,offset) }***

- The stack *tblptr* is used to contain pointers to the tables for **sort, quicksort,** and **partition** when the declarations in **partition** are considered.

- The top element of stack *offset* is the next available relative address for a local of the current procedure.

- All semantic actions in the subtrees for B and C in

  A → BC {$action_A$}

  are done before $action_A$ at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.

- The action for nonterminal M initializes stack *tblptr* with a symbol table for the outermost scope, created by operation *mktable(nil).* The action also pushes relative address 0 onto stack offset.

- Similarly, the nonterminal N uses the operation *mktable(top(tblptr))* to create a new symbol table. The argument *top(tblptr)* gives the enclosing scope for the new table.

- For each variable declaration **id:** T, an entry is created for **id** in the current symbol table. The top of stack offset is incremented by T.width.

- When the action on the right side of $D \rightarrow proc\ id;\ ND_1;\ S$ occurs, the width of all declarations generated by D_1 is on the top of stack offset; it is recorded using *addwidth.* Stacks *tblptr* and *offset* are then popped.
  At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar.

$P \rightarrow M\ D$

$M \rightarrow \varepsilon$

$D \rightarrow D\ ;\ D \mid \mathbf{id}\ :\ T \mid \mathbf{proc\ id}\ ;\ N\ D\ ;\ S$

$N \rightarrow \varepsilon$

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

$S \rightarrow id := E$	{ p : = lookup (**id**.name); **if** p ≠ nil **then** emit(p ‘ : =’ E.place) **else** error }
$E \rightarrow E_1 + E_2$	{ E.place : = newtemp; emit(E.place ‘: =’ E_1.place ‘ + ‘ E_2.place) }
$E \rightarrow E_1 * E_2$	{ E.place : = newtemp; emit(E.place ‘: =’ E_1.place ‘ * ‘ E_2.place) }
$E \rightarrow - E_1$	{ E.place : = newtemp; emit (E.place ‘: =’ ‘uminus’ E_1.place) }
$E \rightarrow (E_1)$	{ E.place : = E_1.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 → $E_1 + E_2$ has the general form:

evaluate E_1 into t_1
evaluate E_2 into t_2
t : = t_1 + t_2

- 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.
- For example, consider the assignment x := a * b + c * d – e * f

Three-address code with stack temporaries

statement	*value of c*
	0
$0 := a * b	1
$1 := c * d	2
$0 := $0 + $1	1
$1 := e * f	2
$0 := $0 - $1	1
x := $0	0

Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is *w*, then the *i*th element of array A begins in location

base + (i – low) x w

where *low* is the lower bound on the subscript and *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of A*[low]*.

The expression can be partially evaluated at compile time if it is rewritten as

$$i \text{ x } w + (base - low \text{ x } w)$$

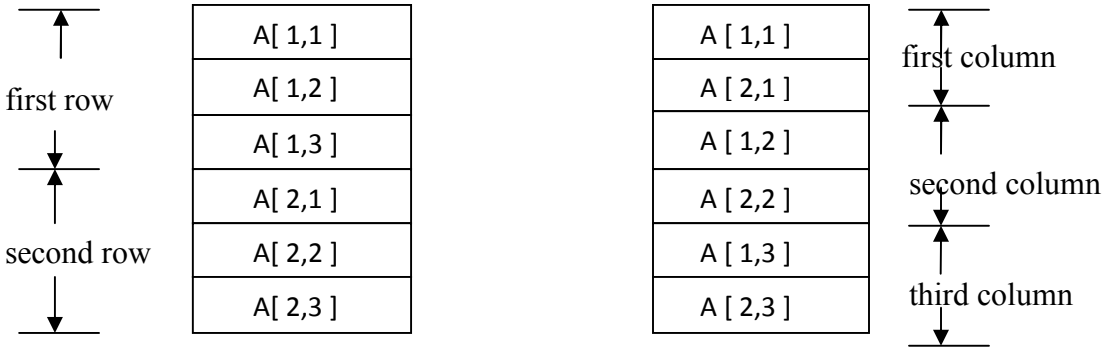
The subexpression $c = base - low \text{ x } w$ can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of A[i] is obtained by simply adding i x w to c.

Address calculation of multi-dimensional arrays:

A two-dimensional array is stored in of the two forms :

- Row-major (row-by-row)
- Column-major (column-by-column)

Layouts for a 2 x 3 array



(a) ROW-MAJOR **(b) COLUMN-MAJOR**

In the case of row-major form, the relative address of A[i_1 , i_2] can be calculated by the formula

$$base + ((i_1 - low_1) \text{ x } n_2 + i_2 - low_2) \text{ x } w$$

where, low_1 and low_2 are the lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take. That is, if $high_2$ is the upper bound on the value of i_2, then $n_2 = high_2 - low_2 + 1$.

Assuming that i_1 and i_2 are the only values that are known at compile time, we can rewrite the above expression as

$$((i_1 \text{ x } n_2) + i_2) \text{ x } w + (base - ((low_1 \text{ x } n_2) + low_2) \text{ x } w)$$

Generalized formula:

The expression generalizes to the following expression for the relative address of A[$i_1,i_2,...,i_k$]

$((\ldots((i_1n_2 + i_2)n_3 + i_3)\ldots)n_k + i_k)$ x w + $base - ((\ldots((low_1n_2 + low_2)n_3 + low_3)\ldots)n_k + low_k)$ x w

for all j, $n_j = high_j - low_j + 1$

The Translation Scheme for Addressing Array Elements :

Semantic actions will be added to the grammar :

(1) $S \rightarrow L := E$
(2) $E \rightarrow E + E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow L$
(5) $L \rightarrow Elist\ ]$
(6) $L \rightarrow$ **id**
(7) $Elist \rightarrow Elist\ ,\ E$
(8) $Elist \rightarrow$ **id** [E

We generate a normal assignment if L is a simple name, and an indexed assignment into the location denoted by L otherwise :

(1) $S \rightarrow L := E$ { **if** *L.offset* = **null then** /* *L is a simple* **id** */
emit (*L.place* ': =' *E.place*) ;
else
emit (*L.place* '[' *L.offset* ']' ': =' *E.place*) }

(2) $E \rightarrow E_1 + E_2$ { *E.place : = newtemp;*
emit (*E.place* ': =' E_1*.place* ' +' E_2*.place*) }

(3) $E \rightarrow (E_1)$ { $E.place := E_1.place$ }

When an array reference L is reduced to E , we want the *r*-value of L. Therefore we use indexing to obtain the contents of the location *L.place* [*L.offset*] :

(4) $E \rightarrow L$ { **if** *L.offset* = **null then** /* *L is a simple* **id** */
E.place : = L.place
else begin
E.place : = newtemp;
emit (*E.place* ': =' *L.place* '[' *L.offset* ']')
end }

(5) $L \rightarrow Elist\ ]$ { *L.place : = newtemp;*
L.offset : = newtemp;
emit (*L.place* ': =' *c(Elist.array));*
emit (*L.offset* ': =' *Elist.place* '*' *width (Elist.array*)) }

(6) $L \rightarrow$ **id** { *L.place :=* **id***.place;*
L.offset := **null** }

(7) $Elist \rightarrow Elist_1\ ,\ E$ { *t := newtemp;*
$m := Elist_1.ndim + 1$;
emit (*t* ': =' $Elist_1$*.place* '*' *limit* (*$Elist_1$.array,m*));
emit (*t* ': =' *t* '+' *E.place*);
$Elist.array := Elist_1.array$;

Elist.place : = t;
Elist.ndim : = m }

(8) *Elist* → **id** [*E* { *Elist.array : =* **id**.*place;*

Elist.place : = E.place;
Elist.ndim : = 1 }

Type conversion within Assignments :

Consider the grammar for assignment statements as above, but suppose there are two types – real and integer , with integers converted to reals when necessary. We have another attribute *E.type*, whose value is either *real* or *integer*. The semantic rule for *E.type* associated with the production $E \rightarrow E + E$ is :

$E \rightarrow E + E$ { *E.type* : =
if E_1.*type* = *integer* **and**
E_2.*type* = *integer* **then** *integer*
else *real* }

The entire semantic rule for $E \rightarrow E + E$ and most of the other productions must be modified to generate, when necessary, three-address statements of the form x : = inttoreal y, whose effect is to convert integer y to a real of equal value, called x.

Semantic action for $E \rightarrow E_1 + E_2$

E.place := newtemp;
if E_1.*type* = *integer* **and** E_2.*type* = *integer* **then begin**
emit(E.place ': =' E₁.place 'int +' *E₂.place);*
E.type : = integer
end
else if E_1.*type* = *real* **and** E_2.*type* = *real* **then begin**
emit(E.place ': =' E₁.place 'real +' *E₂.place);*
E.type : = real
end
else if E_1.*type* = *integer* **and** E_2.*type* = *real* **then begin**
u : = newtemp;
emit(u ': =' 'inttoreal' *E₁.place);*
emit(E.place ': =' u ' real +' *E₂.place);*
E.type : = real
end
else if E_1.*type* = *real* **and** E_2.*type* =*integer* **then begin**
u : = newtemp;
emit(u ': =' 'inttoreal' *E₂.place*);
emit(E.place ': =' E₁.place ' real +' *u);*
E.type : = real
end
else
E.type : = type_error;

For example, for the input x : = y + i * j
assuming *x* and *y* have type *real*, and i and j have type *integer*, the output would look like

t_1 : = i int* j
t_3 : = inttoreal t_1
t_2 : = y real+ t_3
x : = t_2

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 E_1 **relop** E_2, where E_1 and E_2 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.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for
  a **or** b **and** **not** c
  is the three-address sequence
  t_1 : = **not** c
  t_2 : = b **and** t_1
  t_3 : = a **or** t_2
- A relational expression such as a < b is equivalent to the conditional statement
  if a < b then 1 else 0

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```
100 :   if a < b goto 103
101 :   t : = 0
102 :   goto 104
103 :   t : = 1
104 :
```

Translation scheme using a numerical representation for booleans

$E \rightarrow E_1$ **or** E_2	*{ E.place : = newtemp;* *emit(E.place ': =' E_1.place '***or***' E_2.place) }*
$E \rightarrow E_1$ **and** E_2	*{ E.place : = newtemp;* *emit(E.place ': =' E_1.place '***and***' E_2.place) }*
$E \rightarrow$ **not** E_1	*{ E.place : = newtemp;* *emit(E.place ': =' '***not***' E_1.place) }*
$E \rightarrow (E_1)$	*{ E.place : = E_1.place }*
$E \rightarrow$ **id$_1$ relop id$_2$**	*{ E.place : = newtemp;* *emit('if'* **id$_1$**.*place* **relop**.*op* **id$_2$**.*place '***goto***' nextstat + **3**);* *emit(E.place ': =' '***0***');* *emit('***goto***' nextstat +**2**);* *emit(E.place ': =' '***1***') }*
$E \rightarrow$ **true**	*{ E.place : = newtemp;* *emit(E.place ': =' '***1***') }*
$E \rightarrow$ **false**	*{ E.place : = newtemp;* *emit(E.place ': =' '***0***') }*

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called "**short-circuit**" or "**jumping**" code. It is possible to evaluate boolean expressions without generating code for the boolean operators **and, or,** and **not** if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

```
100 :  if  a  <  b goto  103          107 :  t2 : = 1
101 :   t1 : = 0                      108 :  if  e <  f  goto  111
102 :  goto  104                      109 :  t3 : = 0
103 :  t1 : = 1                       110 : goto 112
104 :  if  c <  d  goto  107          111 :  t3 : = 1
105 :  t2 : = 0                       112 :  t4 : = t2 and t3
106 :  goto  108                      113 :  t5 : = t1 or  t4
```

Flow-of-Control Statements

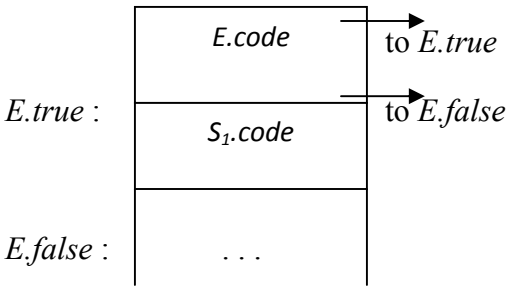
We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S → **if** E **then** S_1
| **if** E **then** S_1 **else** S_2
| **while** E **do** S_1

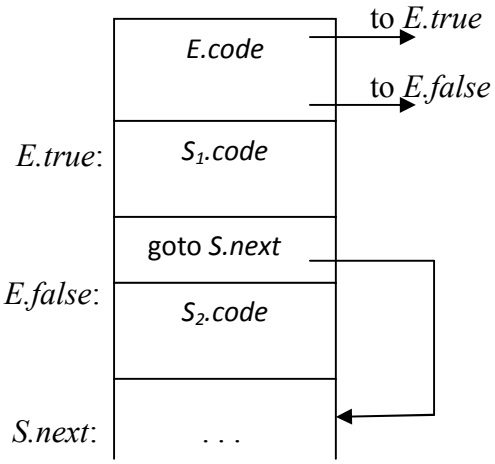
In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

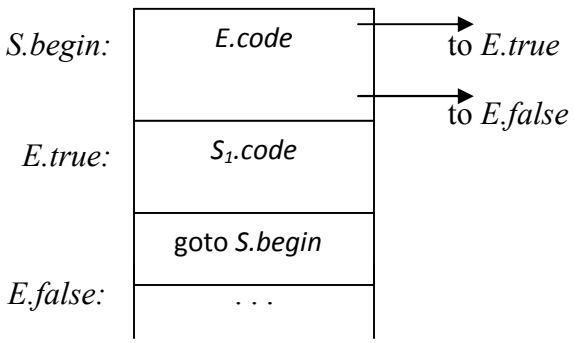
Code for if-then , if-then-else, and while-do statements



(a) if-then



(b) if-then-else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen('\textbf{goto}'\ S.next) \mid\mid$ $gen(E.false\ ':') \mid\mid S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin\ ':') \mid\mid E.code \mid\mid$ $gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen('\textbf{goto}'\ S.begin)$

Control-Flow Translation of Boolean Expressions:

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ **or** E_2	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.false\ ':') \mid\mid E_2.code$
$E \rightarrow E_1$ **and** E_2	$E.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.true\ ':') \mid\mid E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (\ E1\ )$	$E_1.true := E.true;$

	$E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow$ **id$_1$ relop id$_2$**	$E.code := gen($‘**if**’ **id$_1$**.*place* **relop.***op* **id$_2$**.*place* ‘**goto**’ *E.true)* || *gen*(‘**goto**’ *E.false*)
$E \rightarrow$ **true**	*E.code : = gen(‘***goto**’ *E.true)*
$E \rightarrow$ **false**	*E.code : = gen(‘***goto***’ E.false)*

CASE STATEMENTS

The “switch” or “case” statement is available in a variety of languages. The switch-statement syntax is as shown below :

Switch-statement syntax

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

There is a selector expression, which is to be evaluated, followed by *n* constant values that the expression might take, including a default “value” which always matches the expression if no other value does. The intended translation of a switch is code to:

1. Evaluate the expression.
2. Find which value in the list of cases is the same as the value of the expression.
3. Execute the statement associated with the value found.

Step (2) can be implemented in one of several ways :

- By a sequence of conditional **goto** statements, if the number of cases is small.
- By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
- If the number of cases s large, it is efficient to construct a hash table.
- There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say i_{min} to i_{max}, and the number of different values is a reasonable fraction of i_{max} - i_{min}, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset j - i_{min} and the label for the default in entries not filled otherwise. To perform switch,

evaluate the expression to obtain the value of j , check the value is within range and transfer to the table entry at offset $j-i_{min}$.

Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

switch *E*
begin
case V_1 : S_1
case V_2 : S_2
. . .
case V_{n-1} : S_{n-1}
default : S_n
end

This case statement is translated into intermediate code that has the following form :

Translation of a case statement

code to evaluate *E* into t
goto test
L_1 : code for S_1
goto next
L_2 : code for S_2
goto next
. . .
L_{n-1} : code for S_{n-1}
goto next
L_n : code for S_n
goto next
test : if t = V_1 goto L_1
if t = V_2 goto L_2
. . .
if t = V_{n-1} goto L_{n-1}
goto L_n
next :

To translate into above form :

- When keyword **switch** is seen, two new labels **test** and **next,** and a new temporary **t** are generated.
- As expression *E* is parsed, the code to evaluate *E* into **t** is generated. After processing *E* , the jump **goto test** is generated.
- As each **case** keyword occurs, a new label L_i is created and entered into the symbol table. A pointer to this symbol-table entry and the value V_i of case constant are placed on a stack (used only to store cases).

- Each statement **case** $V_i : S_i$ is processed by emitting the newly created label L_i, followed by the code for S_i , followed by the jump **goto next**.

- Then when the keyword **end** terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form

```
case   V1   L1
case   V2   L2
       . . .
case   Vn-1 Ln-1
case    t   Ln
label  next
```

where t is the name holding the value of the selector expression E, and L_n is the label for the default statement.

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, a 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 :

1. *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.
2. *merge*(p_1,p_2) concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.
3. *backpatch(p,i)* inserts i as the target label for each of the statements on the list pointed to by *p.*

Boolean Expressions:

We now construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. The grammar we use is the following:

(1) $E \rightarrow E_1$ **or** $M\ E_2$
(2) | E_1 **and** $M\ E_2$
(3) | **not** E_1
(4) | (E_1)
(5) | **id$_1$ relop id$_2$**
(6) | **true**
(7) | **false**
(8) $M \rightarrow \varepsilon$

Synthesized attributes *truelist* and *falselist* of nonterminal *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 $E \rightarrow E_1$ **and** $M\ E_2$. If E_1 is false, then *E* is also false, so the statements on E_1*.falselist* become part of *E.falselist*. If E_1 is true, then we must next test E_2, so the target for the statements E_1*.truelist* must be the beginning of the code generated for E_2. This target is obtained using marker nonterminal *M*.

Attribute *M.quad* records the number of the first statement of E_2*.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 E_1*.truelist* when we have seen the remainder of the production $E \rightarrow E_1$ **and** $M\ E_2$. The translation scheme is as follows:

(1) $E \rightarrow E_1$ **or** $M\ E_2$ { *backpatch* (E_1*.falselist*, *M.quad*);
E.truelist : = *merge*(E_1*.truelist,* E_2*.truelist*);
E.falselist : = E_2*.falselist* }

(2) $E \rightarrow E_1$ **and** $M\ E_2$ { *backpatch* (E_1*.truelist, M.quad*);
E.truelist : = E_2*.truelist*;
E.falselist : = *merge*(E_1*.falselist,* E_2*.falselist*) }

(3) $E \rightarrow$ **not** E_1 { *E.truelist* : = E_1*.falselist*;
E.falselist : = E_1*.truelist*; }

(4) $E \rightarrow$ (E_1) { *E.truelist* : = E_1*.truelist*;
E.falselist : = E_1*.falselist*; }

(5) $E \rightarrow$ **id₁ relop id₂** { *E.truelist* : = *makelist* (*nextquad*);
E.falselist : = *makelist*(*nextquad + 1*);
emit(‘if’ **id₁**.*place* **relop**.*op* **id₂**.place ‘**goto_**’)
emit(‘**goto_**’) }

(6) $E \rightarrow$ **true** { *E.truelist* : = *makelist*(*nextquad*);
emit(‘**goto_**’) }

(7) $E \rightarrow$ **false** { *E.falselist* : = *makelist*(*nextquad*);
emit(‘**goto_**’) }

(8) $M \rightarrow \varepsilon$ { *M.quad* : = *nextquad* }

Flow-of-Control Statements:

A translation scheme is developed for statements generated by the following grammar :

(1) $S \rightarrow$ **if** E **then** S
(2) | **if** E **then** S **else** S
(3) | **while** E **do** S
(4) | **begin** L **end**
(5) | A
(6) $L \rightarrow L$; S
(7) | S

Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

Scheme to implement the Translation:

The nonterminal E has two attributes *E.truelist* and *E.falselist*. *L* and *S* also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes *L..nextlist* and *S.nextlist*. *S.nextlist* is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and *L.nextlist* is defined similarly.

The semantic rules for the revised grammar are as follows:

(1) $S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$
{ *backpatch* (*E.truelist*, M_1*.quad*);
backpatch (*E.falselist*, M_2*.quad*);
S.nextlist : = *merge* (S_1*.nextlist*, *merge* (*N.nextlist*, S_2*.nextlist*)) }

We backpatch the jumps when *E* is true to the quadruple M_1*.quad*, which is the beginning of the code for S_1. Similarly, we backpatch jumps when *E* is false to go to the beginning of the code for S_2. The list *S.nextlist* includes all jumps out of S_1 and S_2, as well as the jump generated by *N*.

(2) $N \rightarrow \varepsilon$ { *N.nextlist* : = *makelist*(*nextquad*);
emit('**goto** _') }

(3) $M \rightarrow \varepsilon$ { *M.quad* : = *nextquad* }

(4) $S \rightarrow$ **if** E **then** $M\ S_1$ { *backpatch*(*E.truelist*, *M.quad*);
S.nextlist : = *merge*(*E.falselist*, S_1*.nextlist*) }

(5) $S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$ { *backpatch*(S_1*.nextlist*, M_1*.quad*);
backpatch(*E.truelist*, M_2*.quad*);
S.nextlist : = *E.falselist*
emit('**goto**' M_1*.quad*) }

(6) $S \rightarrow$ **begin** L **end** { *S.nextlist* : = *L.nextlist* }

(7) $S \rightarrow A$ { *S.nextlist* : = **nil** }

The assignment *S.nextlist* : = **nil** initializes *S.nextlist* to an empty list.

(8) $L \rightarrow L1 \ ; M \ S$ { *backpatch*(L_1.*nextlist*, *M.quad*);
L.nextlist : = *S.nextlist* }

The statement following L_1 in order of execution is the beginning of *S*. Thus the *L1.nextlist* list is backpatched to the beginning of the code for *S*, which is given by *M.quad*.

(9) $L \rightarrow S$ { *L.nextlist* : = *S.nextlist* }

PROCEDURE CALLS

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns. The run-time routines 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) $S \rightarrow$ **call id** (*Elist*)
(2) *Elist* $\rightarrow$ *Elist , E*
(3) *Elist* $\rightarrow E$

Calling Sequences:

The translation for a call includes a calling sequence, a sequence of actions taken on entry to 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) $S \rightarrow$ **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* }

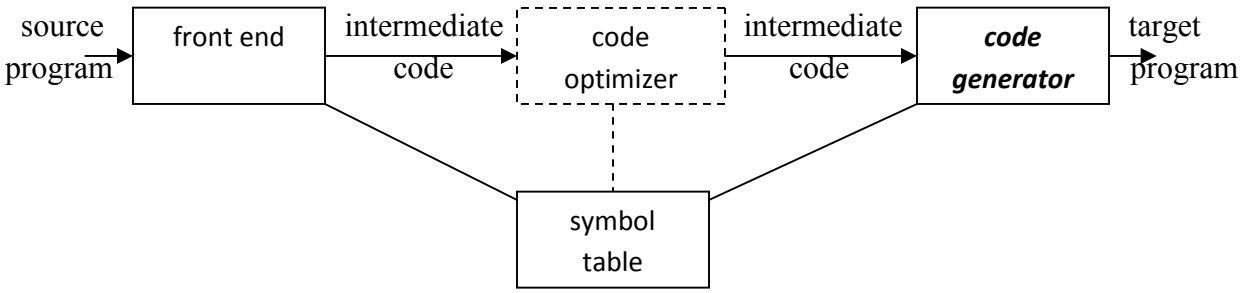
- Here, the code for S is the code for *Elist*, which evaluates the arguments, followed by a **param** *p* statement for each argument, followed by a **call** statement.

- *queue* is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.

MODULE-4 CODE GENERATION

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

Position of code generator



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

1. 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.

2. 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. Relocatable machine language
   - It allows subprograms to be compiled separately.

c. Assembly language
   - Code generation is made easier.

3. 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. For example,
  j : **goto** *i* generates jump instruction as follows :
  - if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
  - if $i > j$, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

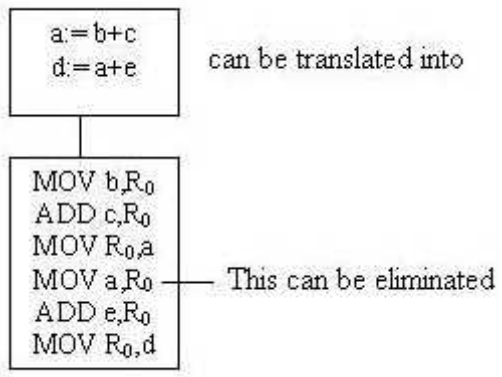
4. Instruction selection:

- 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.

- The former statement can be translated into the latter statement as shown below:



5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.

- The use of registers is subdivided into two subproblems :
  - ***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

6. 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.

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n* general-purpose registers, R_0, R_1, . . . , R_{n-1}.
- It has two-address instructions of the form:

  op source, destination

  where, *op* is an op-code, and *source* and *destination* are data fields.

- It has the following op-codes :

  MOV (move *source* to *destination*)
  ADD (add *source* to *destination*)
  SUB (subtract *source* from *destination*)

- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	FORM	*ADDRESS*	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	*c*(R)	*c*+*contents*(R)	1
indirect register	*R	*contents* (R)	0
indirect indexed	**c*(R)	*contents*(*c*+ *contents*(R))	1
literal	#*c*	*c*	1

- For example : MOV R_0, M stores contents of Register R_0 into memory location M ;
  MOV 4(R_0), M stores the value *contents*(4+*contents*(R_0)) into M.

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.
  For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

  i) MOV b, R_0
  ADD c, R_0 cost = 6
  MOV R_0, a

  ii) MOV b, a
  ADD c, a cost = 6

  iii) Assuming R_0, R_1 and R_2 contain the addresses of a, b, and c :
  MOV *R_1, *R_0
  ADD *R_2, *R_0 cost = 2

- In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

RUN-TIME STORAGE MANAGEMENT

- Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure.
- The two standard storage allocation strategies are:
  1. Static allocation
  2. Stack allocation
- In static allocation, the position of an activation record in memory is fixed at compile time.
- In stack allocation, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.
- The following three-address statements are associated with the run-time allocation and deallocation of activation records:
  1. Call,
  2. Return,
  3. Halt, and
  4. Action, a placeholder for other statements.
- We assume that the run-time memory is divided into areas for:
  1. Code
  2. Static data
  3. Stack

Static allocation

Implementation of call statement:

The codes needed to implement static allocation are as follows:

MOV *#here* + 20, *callee.static_area* /*It saves return address*/

GOTO *callee.code_area* /*It transfers control to the target code for the called procedure */

where,
callee.static_area – Address of the activation record
callee.code_area – Address of the first instruction for called procedure
#here + 20 – Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure *callee* is implemented by :

GOTO **callee.static_area*

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

MOV *#stackstart* , SP /* initializes stack */

Code for the first procedure

HALT /* terminate execution */

Implementation of Call statement:

ADD *#caller.recordsize*, SP /* increment stack pointer */

MOV *#here* + 16, *SP /*Save return address */

GOTO *callee.code_area*

where,
caller.recordsize – size of the activation record
#here + 16 – address of the instruction following the **GOTO**

Implementation of Return statement:

GOTO *0 (SP) /*return to the caller */

SUB *#caller.recordsize*, SP /* decrement SP and restore to previous value */

BASIC BLOCKS AND FLOW GRAPHS

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:
  $t_1 := a * a$
  $t_2 := a * b$
  $t_3 := 2 * t_2$
  $t_4 := t_1 + t_3$
  $t_5 := b * b$
  $t_6 := t_4 + t_5$

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)
```

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

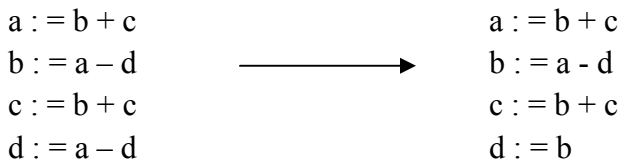
Transformations on Basic Blocks:

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

- Structure-preserving transformations
- Algebraic transformations

1. Structure preserving transformations:

a) Common subexpression elimination:



Since the second and fourth expressions compute the same expression, the basic block can be transformed 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 changing the 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

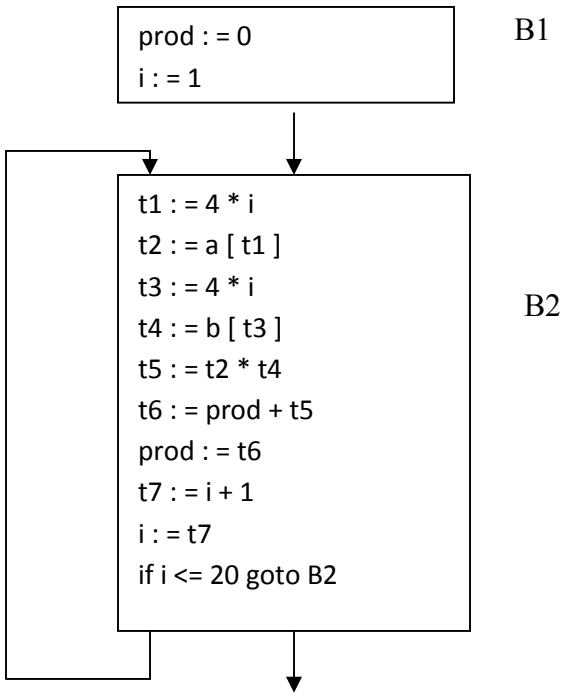
We can interchange the two statements without affecting the value of the block if and only if neither **x** nor **y** is $\mathbf{t_1}$ and neither **b** nor **c** is $\mathbf{t_2}$.

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 of expressions it computes.
ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:



- B_1 is the *initial* node. B_2 immediately follows B_1, so there is an edge from B_1 to B_2. The target of jump from last statement of B_1 is the first statement B_2, so there is an edge from B_1 (last statement) to B_2 (first statement).
- B_1 is the *predecessor* of B_2, and B_2 is a *successor* of B_1.

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.

NEXT-USE INFORMATION

- If the name in a register is no longer needed, then we remove the name from the register and 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.

Symbol Table:

Names	**Liveliness**	**Next-use**
x	not live	no next-use
y	Live	i
z	Live	i

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

- For example: consider the three-address statement **a := b+c**
  It can have the following sequence of codes:

ADD R_j, R_i Cost = 1 // if R_i contains b and R_j contains c

(or)

ADD c, R_i Cost = 2 // if c is in a memory location

(or)

MOV c, R_j Cost = 3 // move c from memory to Rj and add

ADD R_j, R_i

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The 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**,** 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.

2. Consult the address descriptor for y to determine y', the current location 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.

3. Generate the instruction **OP z' , L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

- 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 sequence for the example is:

Statements	**Code Generated**	**Register descriptor**	**Address descriptor**
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
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

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements
a : = b [i] and **a [i] : = b**

Statements	**Code Generated**	**Cost**
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments
a : = *p and ***p : = a**

Statements	**Code Generated**	**Cost**
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	**Code**
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */

x : = y +z
if x < 0 goto z
	MOV y, R_0
	ADD z, R_0
	MOV R_0,x
	CJ< z

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).

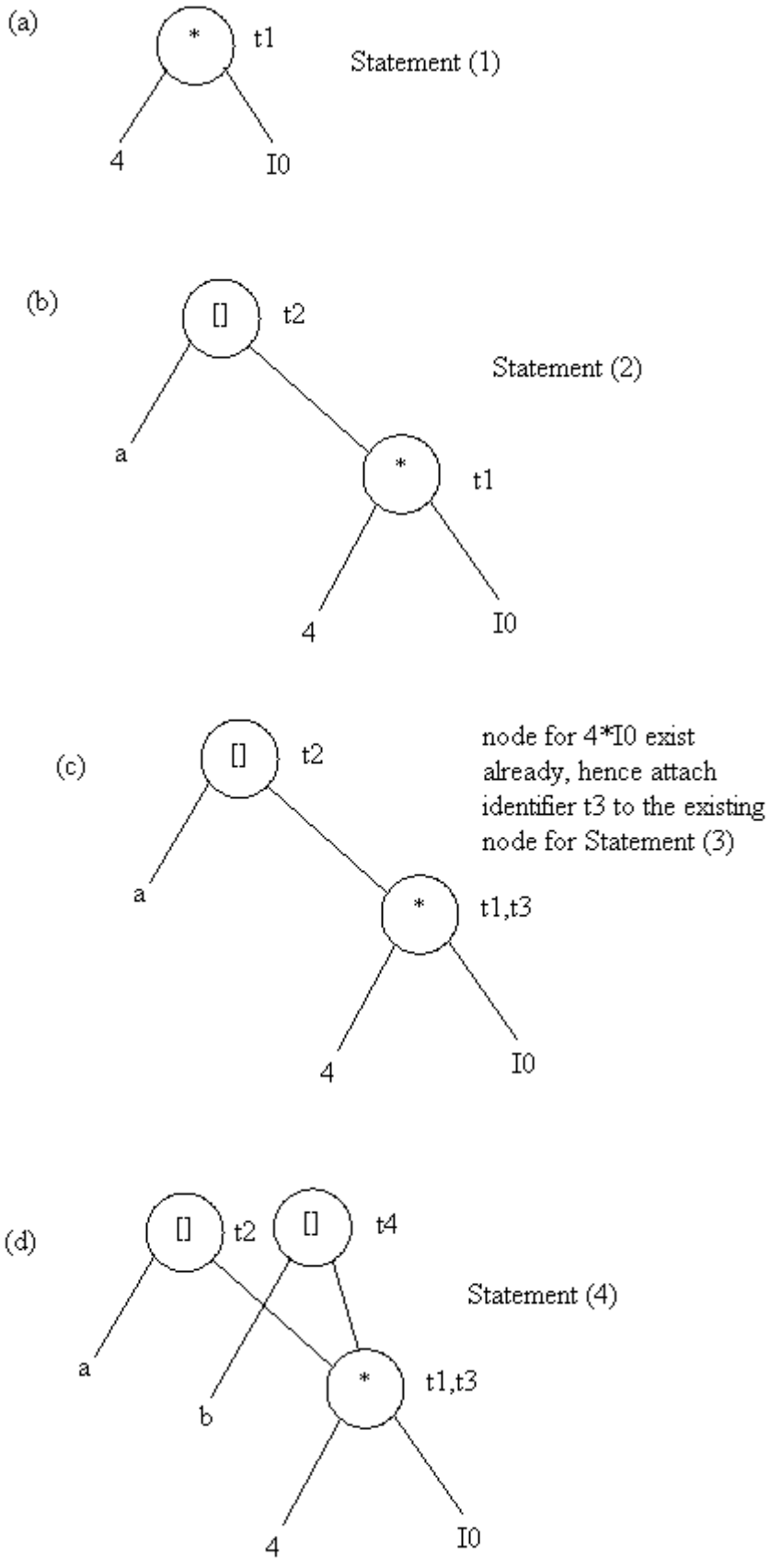
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

identifiers for the node n found in step 2 and set node(x) to n.

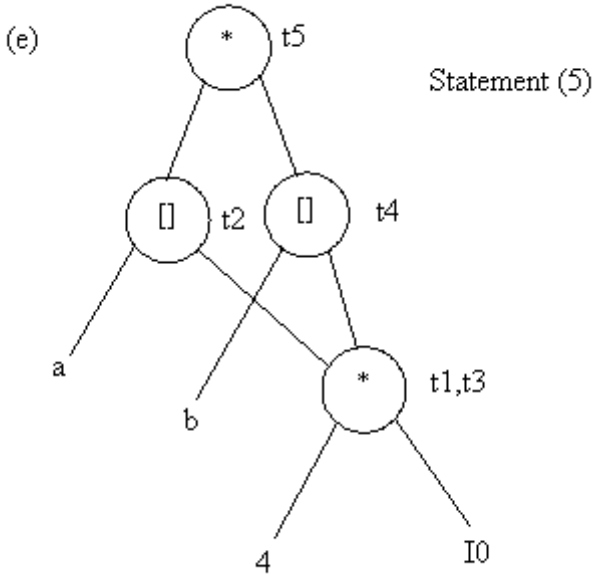
Example: Consider the block of three- address statements:

1. t_1 := 4* i
2. t_2 := a[t_1]
3. t_3 := 4* i
4. t_4 := b[t_3]
5. t_5 := t_2*t_4
6. t_6 := prod+t_5
7. prod := t_6
8. t_7 := i+1
9. i := t_7
10. if i<=20 goto (1)

Stages in DAG Construction

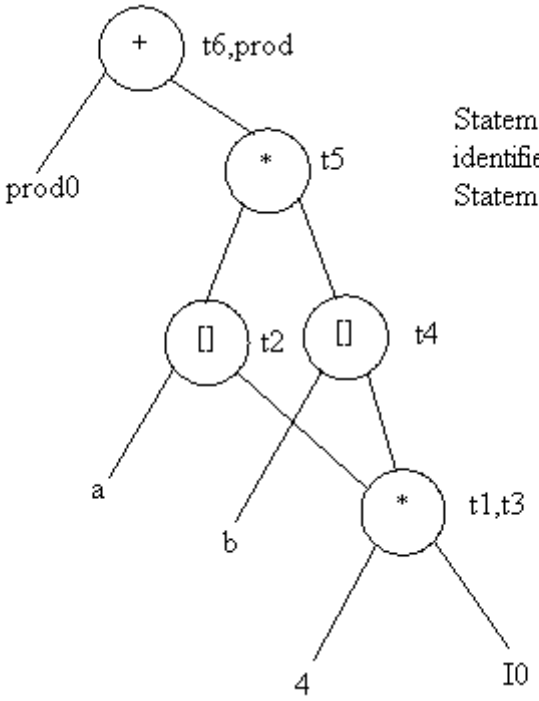


(e)



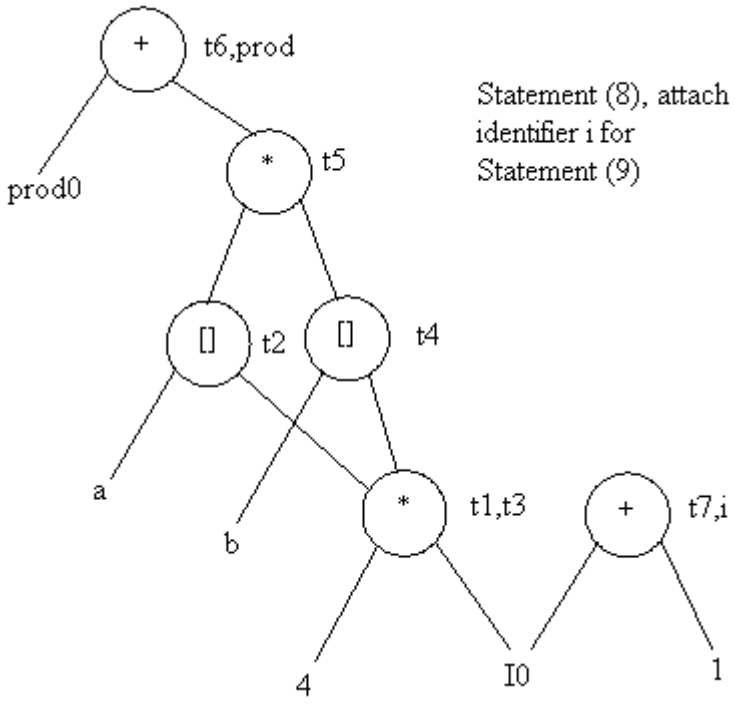
Statement (5)

(f)

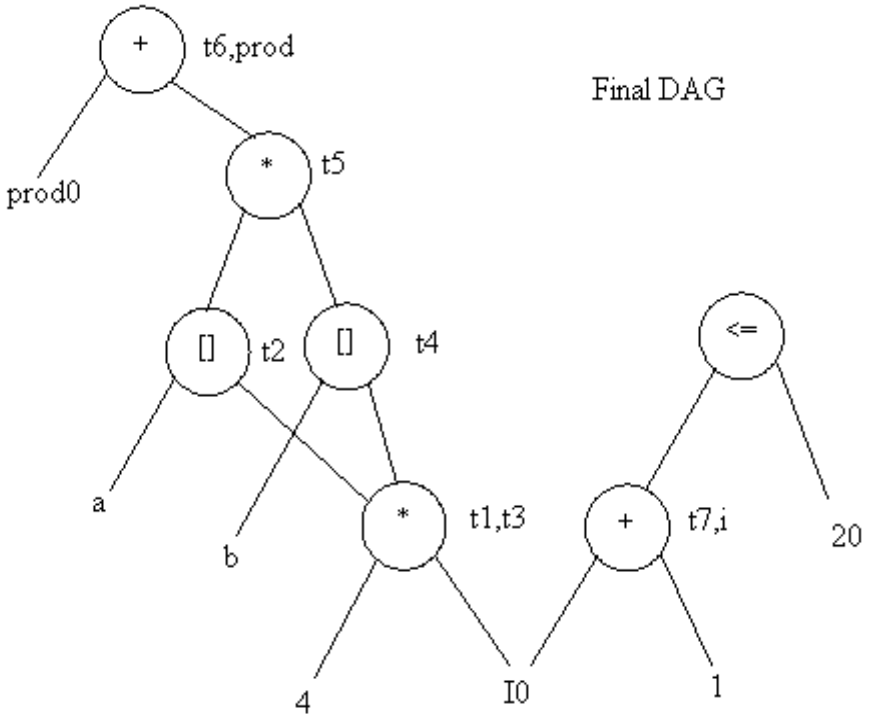


Statement (6), attach identifier prod for Statement (7)

(g)



(h)



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.

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 than we 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:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions **MOV R_0 , t_1** and **MOV t_1 , R_1** have been saved.

A Heuristic ordering for Dags

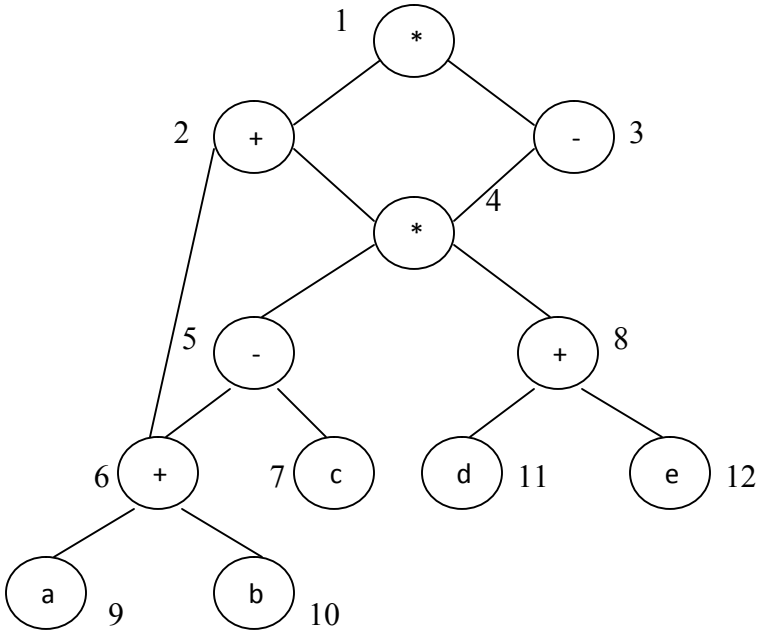
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

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
```

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

$t_8 := d + e$
$t_6 := a + b$
$t_5 := t_6 - c$
$t_4 := t_5 * t_8$
$t_3 := t4 - e$
$t_2 := t_6 + t_4$
$t_1 := t_2 * t_3$

This will yield an optimal code for the DAG on machine whatever be the number of registers.

MODULE-4 - CODE OPTIMIZATION

INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

- Optimizations are classified into two categories. They are
  - Machine independent optimizations:
  - Machine dependant optimizations:

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

- Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

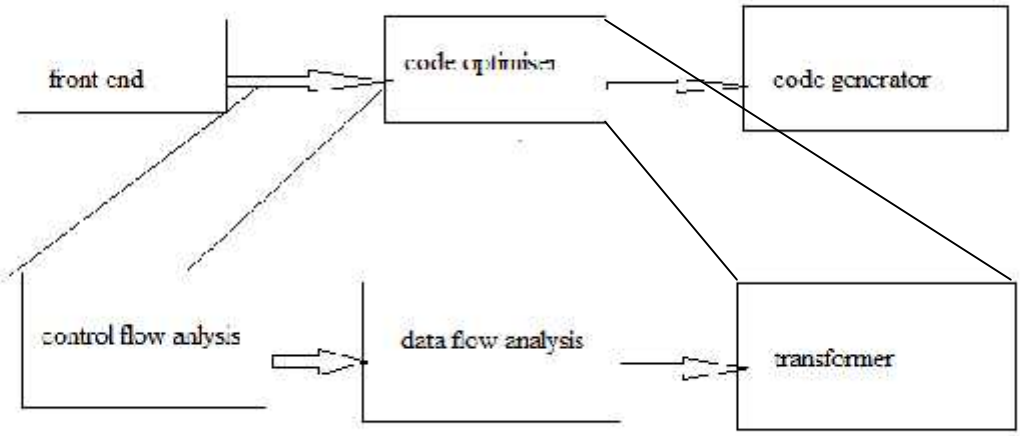
- ✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.

- ✓ The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

- ✓ A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.

- ✓ The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Organization for an Optimizing Compiler:



- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
  - control flow graph
  - Call graph
- Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

PRINCIPAL SOURCES OF OPTIMISATION

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- The transformations

  ✓ Common sub expression elimination,
  ✓ Copy propagation,
  ✓ Dead-code elimination, and
  ✓ Constant folding

  are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

➢ **Common Sub expressions elimination:**

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.
- For example

t_1: = 4*i
t_2: = a [t1]
t_3: = 4*j
t_4: = 4*i
t_5: = n
t_6: = b [t_4] +t_5

The above code can be optimized using the common sub-expression elimination as

t_1: = 4*i
t_2: = a [t_1]
t_3: = 4*j
t_5: = n
t_6: = b [t_1] +t_5

The common sub expression t_4: =4*i is eliminated as its computation is alre ady in t_1. And value of i is not been changed from definition to use.

➢ **Copy Propagation:**

- Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.
- For example:

```
x=Pi;
……
A=x*r*r;
```

The optimization using copy propagation can be done as follows:

```
A=Pi*r*r;
```

Here the variable x is eliminated

➢ **Dead-Code Eliminations:**

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute

values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.
Example:

```
i=0;
if(i=1)
{
 a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

- **Constant folding**:

  - We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

  - One advantage of copy propagation is that it often turns the copy statement into dead code.
  - ✓ For example,
    a=3.14157/2 can be replaced by
    a=1.570 there by eliminating a division operation.

- **Loop Optimizations:**

  - We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
  - Three techniques are important for loop optimization:

  - ✓ code motion, which moves code outside a loop;
  - ✓ Induction-variable elimination, which we apply to replace variables from inner loop.
  - ✓ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

- **Code Motion:**

  - An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

    while (i <= limit-2) /* statement does not change limit*/

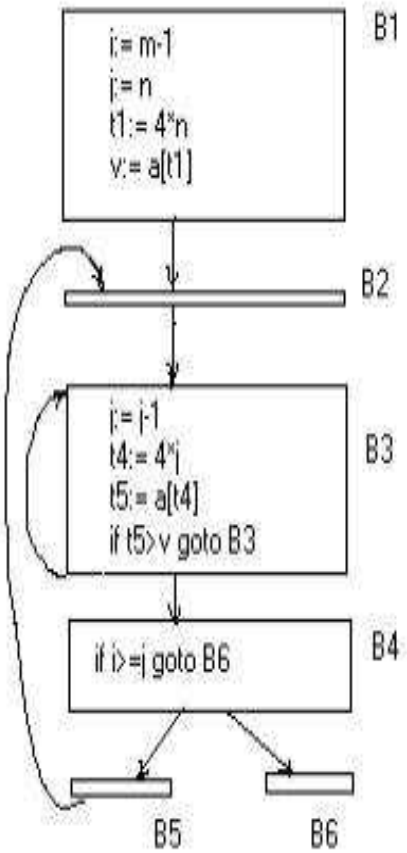
    Code motion will result in the equivalent of

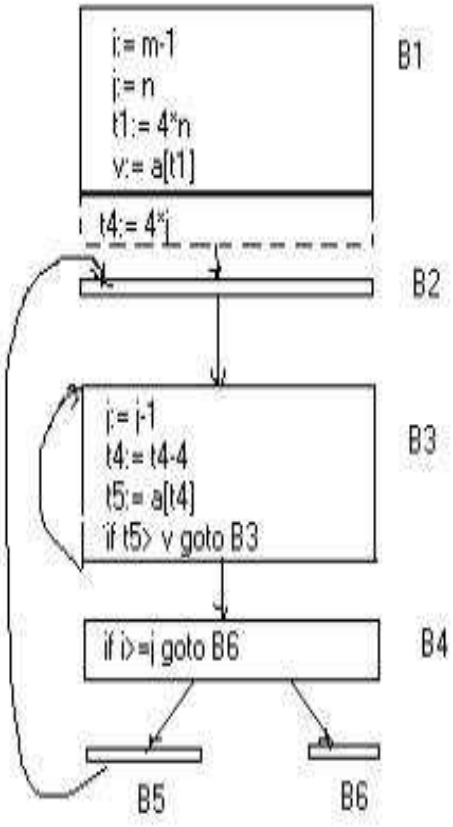
t= limit-2;
while (i<=t) /* statement does not change limit or t */

- **Induction Variables :**

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:
As the relationship t_4:=4*j surely holds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t_4:= 4*j-4 must hold. We may therefore replace the assignment t_4:= 4*j by t_4:= t_4-4. The only problem is that t_4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t_4=4*j on entry to the block B3, we place an initializations of t_4 at the end of the block where j itself is





before **after**

initialized, shown by the dashed addition to block B1 in second Fig.

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

➢ **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 example, 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 as multiplication by a constant, which may be cheaper.

OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- ✓ Structure-Preserving Transformations
- ✓ Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- ✓ Common sub-expression elimination
- ✓ Dead code elimination
- ✓ Renaming of temporary variables
- ✓ Interchange of two independent adjacent statements.

➢ **Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

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

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

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

- **Dead code elimination:**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

- **Renaming of temporary variables:**

  - A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
  - In this we can transform a basic block to its equivalent block called normal-form block.

- **Interchange of two independent adjacent statements:**

  - Two statements

    t_1:=b+c

    t_2:=x+y

    can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2.

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

  a :=b+c
  e :=c+d+b

  the following intermediate code may be generated:

  a :=b+c
  t :=c+d
  e :=t+b

- Example:

  x:=x+0 can be removed

  x:=y**2 can be replaced by a cheaper statement x:=y*y

- The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

LOOPS IN FLOW GRAPH

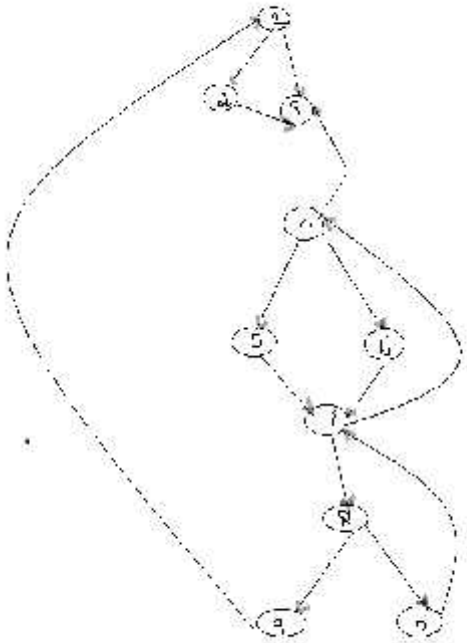
A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

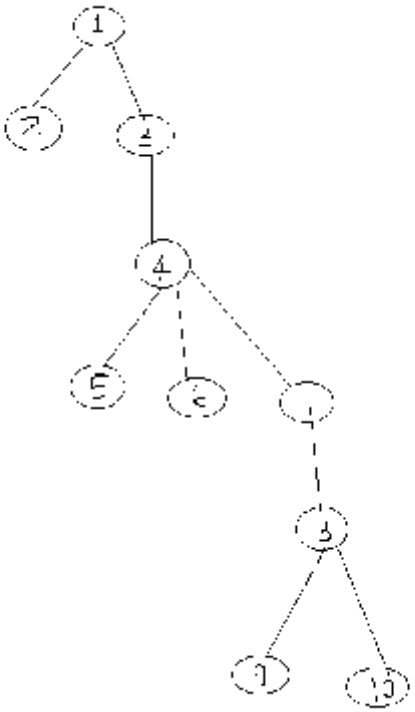
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

*In the flow graph below,
*Initial node,node1 dominates every node.
*node 2 dominates itself
*node 3 dominates all but 1 and 2.
*node 4 dominates all but 1,2 and 3.
*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
*node 7 dominates 7,8 ,9 and 10.
*node 8 dominates 8,9 and 10.
*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n.
- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.



D(1)={1}

D(2)={1,2}

D(3)={1,3}

D(4)={1,3,4}

D(5)={1,3,4,5}

D(6)={1,3,4,6}

D(7)={1,3,4,7}

D(8)={1,3,4,7,8}

D(9)={1,3,4,7,8,9}

D(10)={1,3,4,7,8,10}

Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.

- The properties of loops are

  - ✓ A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
  - ✓ There must be at least one way to iterate the loop(i.e.)at least one path back to the header.

- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

  - ✓ Example:

    In the above graph,

    7 → 4 4 DOM 7

    10 →7 7 DOM 10

    4 → 3

    8 → 3

    9 →1

- The above edges will form loop in flow graph.
- Given a back edge n → d, we define the natural loop of the edge to be d plus th e set of nodes that can reach n without going through d. Node d is the header of the loop.

Algorithm: Constructing the natural loop of a back edge.

Input: A flow graph G and a back edge n→d.

Output: The set loop consisting of all nodes in the natural loop n→d.

Method: Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m's predecessors are also placed in loop. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

Procedure insert(m);
if m is not in *loop* **then begin**
    loop := *loop* U {m};
    push *m* onto *stack*
end;

stack : = empty;

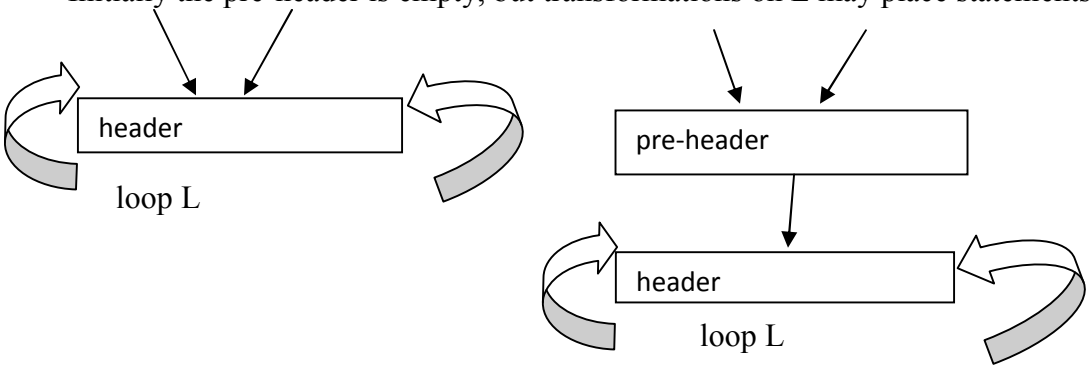
loop : = {*d*};
insert(*n*);
while *stack* is not empty **do begin**
    pop *m*, the first element of *stack*, off *stack*;
    for each predecessor *p* of *m* **do** *insert(p)*
end

Inner loop:

- If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Pre-Headers:

- Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheater.
- The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are not changed.
- Initially the pre-header is empty, but transformations on L may place statements in it.



(a) Before **(b) After**

Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

- The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

- ***Definition***:

  A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.

✓ The forward edges from an acyclic graph in which every node can be reached from initial node of G.

✓ The back edges consist only of edges where heads dominate theirs tails.

✓ Example: The above flow graph is reducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.

- The remaining edges are forward edges.

- If the forward edges form an acyclic graph, then we can say the flow graph reducible.

- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.

- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce 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.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving 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 contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:

  - ✓ Redundant-instructions elimination
  - ✓ Flow-of-control optimizations
  - ✓ Algebraic simplifications
  - ✓ Use of machine idioms
  - ✓ Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R_0,a

(2) MOV a,R_0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R_0.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).

Unreachable Code:

- Another opportunity for peephole optimizations is 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. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug 0

….

If ( debug  ) {

Print debugging information

}
```

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

```
If debug =1 goto L2

  goto L2

L1: print debugging information

L2:                                   ……(a)
```

- One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2

Print debugging information

L2:                                   ……(b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

If debug ≠0 goto L2

Print debugging information

L2: ……(c)

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The 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: gotoL2

by the sequence

goto L2

….

L1: goto L2

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 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

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

……..

L1: if a < b goto L2

L3: ……..(1)

- May be replaced by

If a < b goto L2

goto L3

…….

L3: ……(2)

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

x := x+0

Or

x := x * 1

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

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 example, x² 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 as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, 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 these 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.

i:=i+1 → i++

i:=i-1 → i- -

INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement."

- The details of how data-flow equations are set and solved depend on three factors.

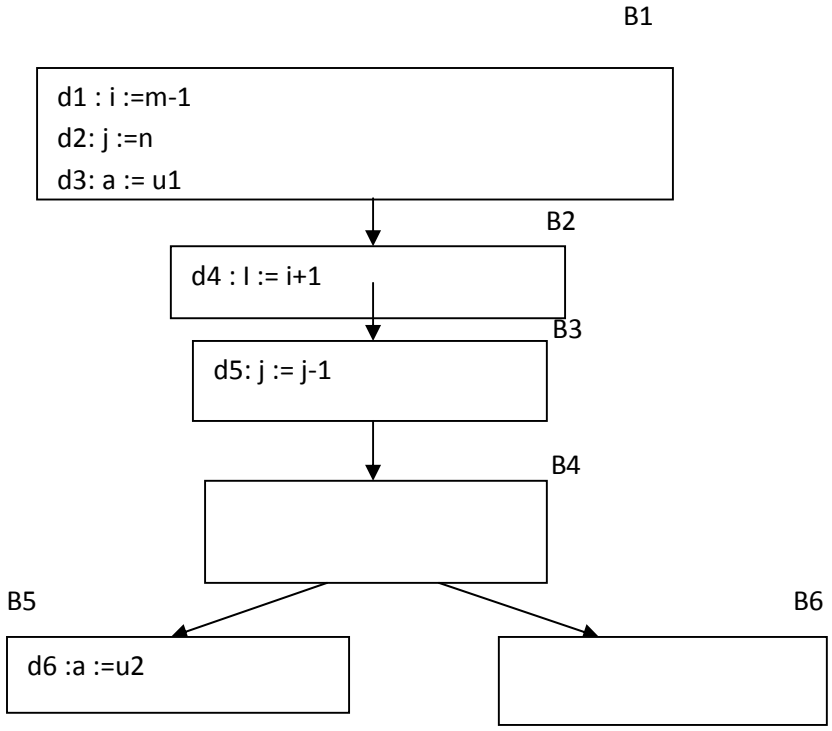
✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].

✓ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

✓ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



- Now let us take a global view and consider all the points in all the blocks. A path from p_1 to p_n is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each i between 1 and n-1, either

✓ P_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, or

✓ P_i is the end of some block and p_{i+1} is the beginning of a successor block.

Reaching definitions:

- A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.

- These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:

✓ A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.

✓ An assignment through a pointer that could refer to x. For example, the assignment *q: = y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Thus a point can be reached

by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

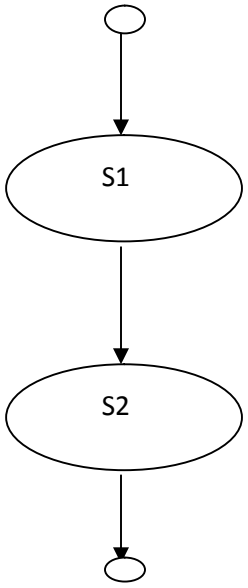
Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

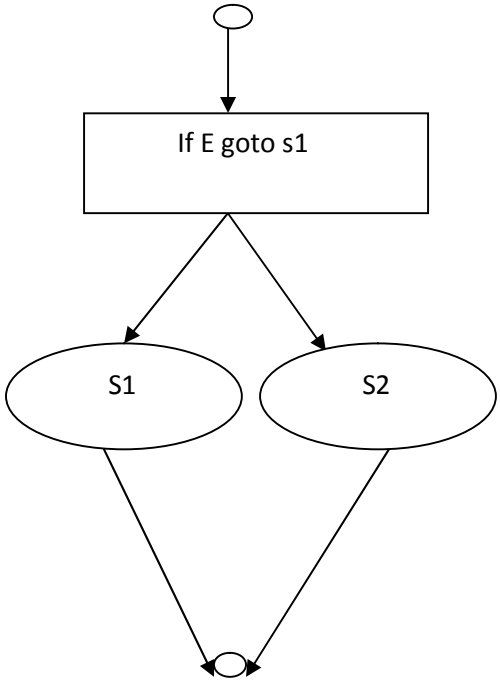
  S ⟶ id: = E| S; S | if E then S else S | do S while E

  E ⟶ id + id| id

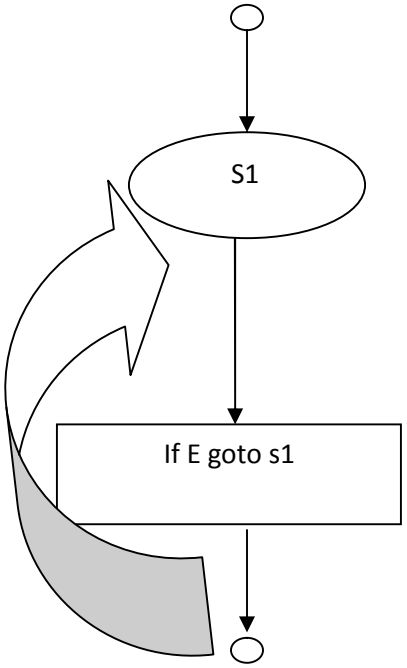
- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



S1 ; S2



IF E then S1 else S2

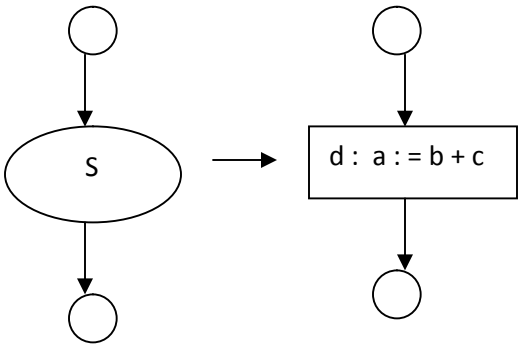


do S1 while E

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.
- **gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

i)



gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus

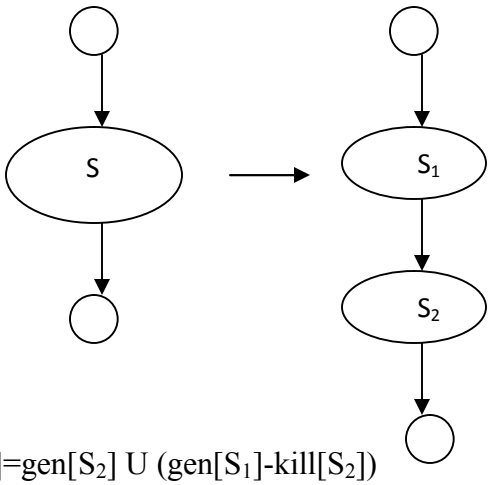
  Gen[S]={d}

- On the other hand, d "kills" all other definitions of a, so we write

  Kill[S] = D_a – {d}

  Where, D_a is the set of all definitions in the program for variable a.

ii)



gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])
Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

in [S_1] = in [S]
in [S_2] = out [S_1]
out [S] = out [S_2]

- Under what circumstances is definition d generated by $S=S_1; S_2$? First of all, if it is generated by S_2, then it is surely generated by S. if d is generated by S_1, it will reach the end of S provided it is not killed by S_2. Thus, we write

  $gen[S]=gen[S_2] \cup (gen[S_1]-kill[S_2])$

- Similar reasoning applies to the killing of a definition, so we have

  $Kill[S] = kill[S_2] \cup (kill[S_1] – gen[S_2])$

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that is, there exists inputs to the program that make their branches go either way.
- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.
- When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.
- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.
- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.
- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.

- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

- The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.

- Assuming we know in[S] we compute out by equation, that is

  Out[S] = gen[S] U (in[S] - kill[S])

- Considering cascade of two statements S_1; S_2, as in the second case. We start by observing in[S_1]=in[S]. Then, we recursively compute out[S_1], which gives us in[S_2], since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1. Now we can compute out[S_2], and this set is equal to out[S].

- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S_1 or S_2 exactly when it reaches the beginning of S.

  In[S_1] = in[S_2] = in[S]

- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

  Out[S]=out[S_1] U out[S_2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.

- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.

- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming

languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A .

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.
- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.
- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods

- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

CODE IMPROVIG TRANSFORMATIONS

- Algorithms for performing the code improving transformations rely on data-flow information. Here we consider common sub-expression elimination, copy propagation and transformations for moving loop invariant computations out of loops and for eliminating induction variables.
- Global transformations are not substitute for local transformations; both must be performed.

Elimination of global common sub expressions:

- The available expressions data-flow problem discussed in the last section allows us to determine if an expression at point p in a flow graph is a common sub-expression. The following algorithm formalizes the intuitive ideas presented for eliminating common sub-expressions.

❖ **ALGORITHM: Global common sub expression elimination.**

INPUT: A flow graph with available expression information.

OUTPUT: A revised flow graph.

METHOD: For every statement s of the form x := y+z[6] such that y+z is available at the beginning of block and neither y nor r z is defined prior to statement s in that block, do the following.

- ✓ **To discover the evaluations of y+z that reach s's block, we follow flow graph edges, searching backward from s's block. However, we do not go through any block that evaluates y+z. The last evaluation of y+z in each block encountered is an evaluation of y+z that reaches s.**
- ✓ **Create new variable u.**
- ✓ **Replace each statement w: =y+z found in (1) by**
  u : = y + z
  w : = u
- ✓ **Replace statement s by x:=u.**

- Some remarks about this algorithm are in order.

✓ The search in step(1) of the algorithm for the evaluations of y+z that reach statement s can also be formulated as a data-flow analysis problem. However, it does not make sense to solve it for all expressions y+z and all statements or blocks because too much irrelevant information is gathered.

- ✓ Not all changes made by algorithm are improvements. We might wish to limit the number of different evaluations reaching s found in step (1), probably to one.
- ✓ Algorithm will miss the fact that a*z and c*z must have the same value in

  a :=x+y c :=x+y

  vs

  b :=a*z d :=c*z
- ✓ Because this simple approach to common sub expressions considers only the literal expressions themselves, rather than the values computed by expressions.

Copy propagation:

- Various algorithms introduce copy statements such as x :=copies may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the dag construction. We may substitute y for x in all these places, provided the following conditions are met every such use u of x.
- Statement s must be the only definition of x reaching u.
- On every path from s to including paths that go through u several times, there are no assignments to y.
- Condition (1) can be checked using ud-changing information. We shall set up a new data-flow analysis problem in which in[B] is the set of copies s: x:=y such that every path from initial node to the beginning of B contains the statement s, and subsequent to the last occurrence of s, there are no assignments to y.

❖ **ALGORITHM: Copy propagation.**

INPUT: a flow graph G, with ud-chains giving the definitions reaching block B, and with c_in[B] representing the solution to equations that is the set of copies x:=y that reach block B along every path, with no assignment to x or y following the last occurrence of x:=y on the path. We also need ud-chains giving the uses of each definition.

OUTPUT: A revised flow graph.

METHOD: For each copy s : x:=y do the following:

- ✓ **Determine those uses of x that are reached by this definition of namely, s: x: =y.**
- ✓ **Determine whether for every use of x found in (1) , s is in c_in[B], where B is the block of this particular use, and moreover, no definitions of x or y occur prior to this use of x within B. Recall that if s is in c_in[B]then s is the only definition of x that reaches B.**

- ✓ **If s meets the conditions of (2), then remove s and replace all uses of x found in (1) by y.**

Detection of loop-invariant computations:

- Ud-chains can be used to detect those computations in a loop that are loop-invariant, that is, whose value does not change as long as control stays within the loop. Loop is a region consisting of set of blocks with a header that dominates all the other blocks, so the only way to enter the loop is through the header.
- If an assignment x := y+z is at a position in the loop where all possible definitions of y and z are outside the loop, then y+z is loop-invariant because its value will be the same each time x:=y+z is encountered. Having recognized that value of x will not change, consider v := x+w, where w could only have been defined outside the loop, then x+w is also loop-invariant.

❖ **ALGORITHM: Detection of loop-invariant computations.**

INPUT: A loop L consisting of a set of basic blocks, each block containing sequence of three-address statements. We assume ud-chains are available for the individual statements.

OUTPUT: the set of three-address statements that compute the same value each time executed, from the time control enters the loop L until control next leaves L.

METHOD: we shall give a rather informal specification of the algorithm, trusting that the principles will be clear.

- ✓ **Mark "invariant" those statements whose operands are all either constant or have all their reaching definitions outside L.**
- ✓ **Repeat step (3) until at some repetition no new statements are marked "invariant".**
- ✓ **Mark "invariant" all those statements not previously so marked all of whose operands either are constant, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked invariant.**

Performing code motion:

- Having found the invariant statements within a loop, we can apply to some of them an optimization known as code motion, in which the statements are moved to pre-header of the loop. The following three conditions ensure that code motion does not change what the program computes. Consider s: x: =y+z.
- ✓ The block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with a successor not in the loop.
- ✓ There is no other statement in the loop that assigns to x. Again, if x is a temporary assigned only once, this condition is surely satisfied and need not be changed.

- ✓ No use of x in the loop is reached by any definition of x other than s. This condition too will be satisfied, normally, if x is temporary.

❖ **ALGORITHM: Code motion.**

INPUT: A loop L with ud-chaining information and dominator information.

OUTPUT: A revised version of the loop with a pre-header and some statements moved to the pre-header.

METHOD:

- ✓ **Use loop-invariant computation algorithm to find loop-invariant statements.**
- ✓ **For each statement s defining x found in step(1), check:**
  - **i) That it is in a block that dominates all exits of L,**
  - **ii) That x is not defined elsewhere in L, and**
  - **iii) That all uses in L of x can only be reached by the definition of x in statement s.**
- ✓ **Move, in the order found by loop-invariant algorithm, each statement s found in (1) and meeting conditions (2i), (2ii), (2iii) , to a newly created pre-header, provided any operands of s that are defined in loop L have previously had their definition statements moved to the pre-header.**

- To understand why no change to what the program computes can occur, condition (2i) and (2ii) of this algorithm assure that the value of x computed at s must be the value of x after any exit block of L. When we move s to a pre-header, s will still be the definition of x that reaches the end of any exit block of L. Condition (2iii) assures that any uses of x within L did, and will continue to, use the value of x computed by s.

Alternative code motion strategies:

- The condition (1) can be relaxed if we are willing to take the risk that we may actually increase the running time of the program a bit; of course, we never change what the program computes. The relaxed version of code motion condition (1) is that we may move a statement s assigning x only if:

  1'. The block containing s either dominates all exists of the loop, or x is not used outside the loop. For example, if x is a temporary variable, we can be sure that the value will be used only in its own block.

- If code motion algorithm is modified to use condition (1'), occasionally the running time will increase, but we can expect to do reasonably well on the average. The modified algorithm may move to pre-header certain computations that may not be executed in the

loop. Not only does this risk slowing down the program significantly, it may also cause an error in certain circumstances.

- Even if none of the conditions of (2i), (2ii), (2iii) of code motion algorithm are met by an assignment x: =y+z, we can still take the computation y+z outside a loop. Create a new temporary t, and set t: =y+z in the pre-header. Then replace x: =y+z by x: =t in the loop. In many cases we can propagate out the copy statement x: = t.

Maintaining data-flow information after code motion:

- The transformations of code motion algorithm do not change ud-chaining information, since by condition (2i), (2ii), and (2iii), all uses of the variable assigned by a moved statement s that were reached by s are still reached by s from its new position.
- Definitions of variables used by s are either outside L, in which case they reach the pre-header, or they are inside L, in which case by step (3) they were moved to pre-header ahead of s.
- If the ud-chains are represented by lists of pointers to pointers to statements, we can maintain ud-chains when we move statement s by simply changing the pointer to s when we move it. That is, we create for each statement s pointer p_s, which always points to s.
- We put the pointer on each ud-chain containing s. Then, no matter where we move s, we have only to change p_s , regardless of how many ud-chains s is on.
- The dominator information is changed slightly by code motion. The pre-header is now the immediate dominator of the header, and the immediate dominator of the pre-header is the node that formerly was the immediate dominator of the header. That is, the pre-header is inserted into the dominator tree as the parent of the header.

Elimination of induction variable:

- A variable x is called an induction variable of a loop L if every time the variable x changes values, it is incremented or decremented by some constant. Often, an induction variable is incremented by the same constant each time around the loop, as in a loop headed by for i := 1 to 10.
- However, our methods deal with variables that are incremented or decremented zero, one, two, or more times as we go around a loop. The number of changes to an induction variable may even differ at different iterations.
- A common situation is one in which an induction variable, say i, indexes an array, and some other induction variable, say t, whose value is a linear function of i, is the actual offset used to access the array. Often, the only use made of i is in the test for loop termination. We can then get rid of i by replacing its test by one on t.
- We shall look for basic induction variables, which are those variables i whose only assignments within loop L are of the form i := i+c or i-c, where c is a constant.

❖ **ALGORITHM: Elimination of induction variables.**

INPUT: A loop L with reaching definition information, loop-invariant computation information and live variable information.

OUTPUT: A revised loop.

METHOD:

- ✓ **Consider each basic induction variable i whose only uses are to compute other induction variables in its family and in conditional branches. Take some j in i's family, preferably one such that c and d in its triple are as simple as possible and modify each test that i appears in to use j instead. We assume in the following tat c is positive. A test of the form 'if i relop x goto B', where x is not an induction variable, is replaced by**

  r := c*x /* r := x if c is 1. */

  r := r+d /* omit if d is 0 */

  if j relop r goto B

  where, r is a new temporary. The case 'if x relop i goto B' is handled analogously. If there are two induction variables i_1 and i_2 in the test if i_1 relop i_2 goto B, then we check if both i_1 and i_2 can be replaced. The easy case is when we have j_1 with triple and j_2 with triple, and $c_1=c_2$ and $d_1=d_2$. Then, i_1 relop i_2 is equivalent to j_1 relop j_2.

- ✓ **Now, consider each induction variable j for which a statement j: =s was introduced. First check that there can be no assignment to s between the introduced statement j :=s and any use of j. In the usual situation, j is used in the block in which it is defined, simplifying this check; otherwise, reaching definitions information, plus some graph analysis is needed to implement the check. Then replace all uses of j by uses of s and delete statement j: =s.**