4. COMPILERS AND INTERPRETERS

4.1 ASPECTS OF COMPILATION

A compiler bridges the semantic gap between a programming language domain and an execution domain. Two aspects of compilation are

1. Generate code to implement meaning of a source program in the execution domain.
2. Provide diagnostics for violations of programming language semantics in a source program

To understand the key issues involved in implementing these aspects, we first discuss features of programming language that contribute to the semantic gap between a programming language domain and an execution domain. These features are:

- Data types
- Data structures
- Scope rules
- Control structure

Data types

Definition of Data type: A data type is the specification of

(i) legal values for variables of the type and

(ii) legal operations on the legal values of the type.

Legal operations of a type typically include an assignment operation and a set of data manipulation operations. Semantics of a data type require a compiler to ensure that variables of a type are assigned or manipulated only through legal operations. The following tasks are involved

1. Checking legality of an operation for types of operands.
2. Use type conversion operations to convert values of one type into values of another type whenever necessary and permissible according to the rules of a programming language.

3. Use appropriate instruction sequence of the target machine to implement the operations of a type.

Example 1: Consider program segment

```
i: integer;
a, b: real
a := b + I
```

Instructions generated for program segment:

```
CONV_R AREG, I
ADD_R  AREG, B
MOVEM  AREG, A
```

Each instruction is a type specific i.e. it expects the operand to be of a specific type.

Data structures

A program may use a datra structure like an array, stack, records, list, etc. To compile a reference to an element of a data structure, the compiler must develop a memory mapping to access the memory word(s) allocated to the element.

Example 2: Consider the Pascal program segment

```
program example (input, output);
    type
        employee = record
            name : array [1..10] of character;
            sex : character;
            id : integer;
        end
        weekday = (mon, tue, wed, thu, fri, sat);
    var
        info : array [1..500] of employee;
        today : weekday;
        i, j : integer;
    begin
```

```
        today := mon;
        info[i].id := j;
        if today = tue then…
end
```

Here, info is an array of records. The reference info[i].id involves use of two different kinds of mapping.

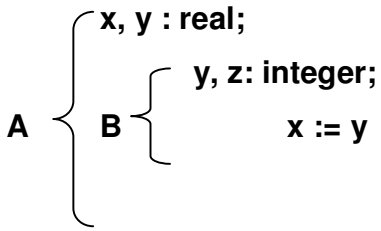
- The first one is the homogeneous mapping of an array reference. This is used to access info[i].
- The second mapping is used to access the field id within an element of info which is a data item of type employee.

The type weekday is user defined data types. The compiler must first decide how to represent different values of the type and then develop an appropriate mapping between the values mon,….,fri and their representations.

Scope rules

Scope rules determine the accessibility of variables declared in different block of a program. The scope of a program entity (e. g., a data item) is that part of a program in which the entity is accessible. In most languages the scope of a data item is restricted to the program block in which the data item is declared. It extends into an enclosed block unless the enclosed block contains a declaration of a variable with an identical name.

Example 3:



Variable x of block A is accessible in block A and also in the enclosed block B. However, variable of y block A is not accessible in block B because block B contains a declaration of another variable named y. Thus, the statement x: =y uses variable y of block Band variable x of block A. Note that variables y and z of block B are not accessible in block A.

The compiler performs operations called scope analysis and name resolution to determine which data item is designed by the use of specific place in the source program. The generated code simply implements results of these analyses.

Control structure

The control structure of a language is the collection of language features that can be used for altering the flow of control during execution of program. It includes statements for unconditional and conditional transfer of control, conditional execution, iteration control and procedure calls. The compiler must ensure that a source program does not violate the semantics of control structure.

Example 4: In the Pascal program segment

```
for i := 1 to 100 do
begin
        lab1: if i = 10 then..
end;
```

A control transfer to the statement bearing label lab1 from outside the loop is forbidden. Some languages also forbid assignments to the control variable of a for loop within the body of the loop.

4.2 MEMORY ALLOCATION

Memory allocation involves three important task.

1. Determine the amount of memory required to represent the values of a data item.
2. Use an appropriate memory allocation model to implement the lifetimes and scope of data items.
3. Determine appropriate memory mappings to access the values in a non-scalar data item.

4.2.1 Static and Dynamic Memory Allocation

Memory Binding is an association between the 'memory address' attribute of a data item and the address of a memory area.
Memory allocation is the procedure used to perform memory bindings. The binding ceases to exist when memory is deallocated. Memory bindings can be static or dynamic in nature giving rise to the static and dynamic memory allocation models.

Static Memory Allocation

In static memory allocation, memory is allocated to a variable before the execution of program begins. Static memory allocation is performed during compilation. No memory allocation or deallocation actions are performed during the execution of a program. Thus, variables remain permanently allocated; allocation to a variable exists even if the program unit in which it is defined is not active.

Dynamic Memory Allocation

In dynamic memory allocation, memory bindings are established and destroyed during the execution of a program.

Typical examples of the use of these memory allocation models are FORTRAN for static memory allocation and Block structured language like PL/1, Pascal, Ada, etc for dynamic memory allocation

Example 5: Figure 1 illustrates static and dynamic memory allocation to a program consisting of 3 program units A, B and C. Part (a) shows static memory allocation. Part (b) shows dynamic allocation when only program unit A is active. Part (c) shows the situation after A calls B, while Part (d) shows the situation after B returns to A and A calls C. C has been allocated part of memory deallocated from B. It is clear that static memory allocation allocates more memory than dynamic memory allocation except when all program units are active.

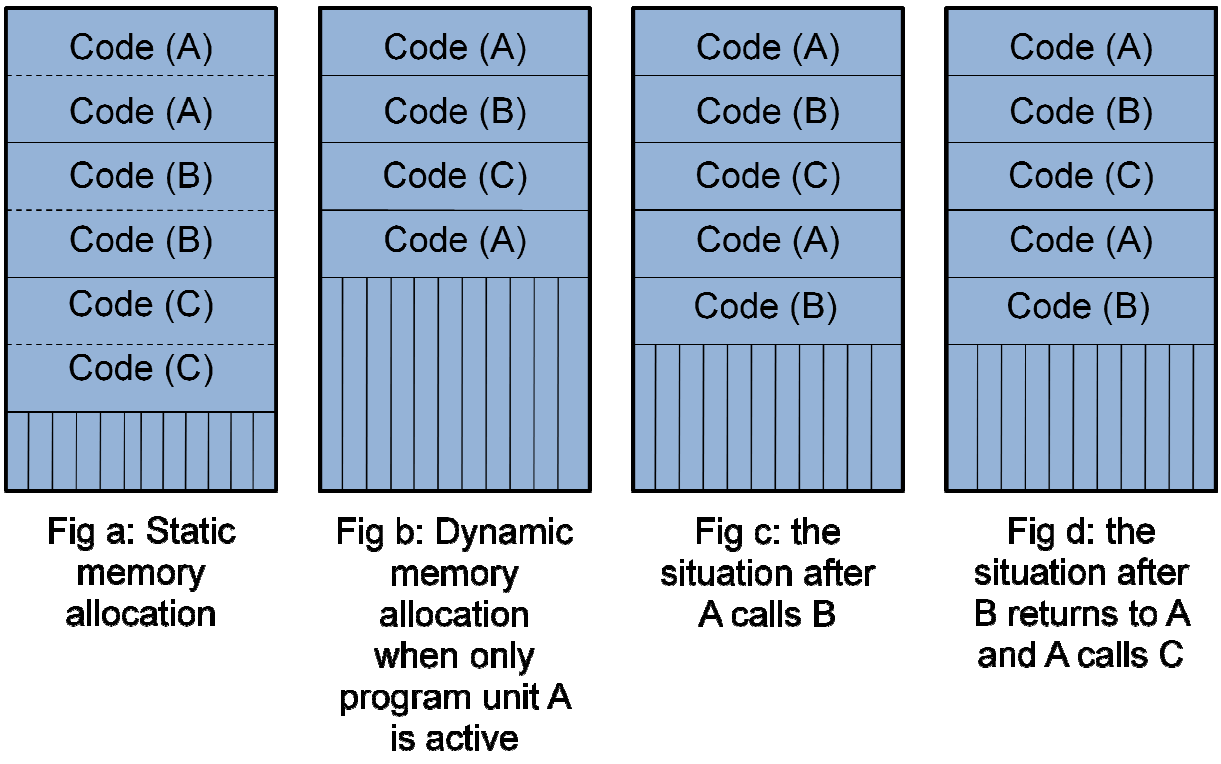


Figure 1: static and dynamic memory allocation

Dynamic memory allocation has two flavors-

- Automatic allocation implies memory binding performed at execution init time of a program unit. In automatic dynamic allocation, memory is allocated to the variables declared in a program unit when the program unit is entered during execution and is deallocated when the program unit is exited. Thus the same memory area may be used for the variables of different program units as in figure 1. It is also possible that different memory areas may be allocated to the same variable in different activations of a program.

- Program controlled allocation implies memory binding performed during the execution of a program unit. In program controlled dynamic allocation, a program can allocate or deallocated memory at arbitrary points during its execution.

In both automatic and program controlled allocation, address of the memory area allocated to a program unit cannot be determined at compilation time.

Dynamic memory allocation is implemented using stacks and heaps, thus necessitating pointer based access to variables. This tends to make it slower in execution than static memory allocation.

- Automatic dynamic allocation is implemented using a stack since entry and exit from program units is LIFO in nature. When a program unit is entered during the execution of the program, a record is created in the stack to contain its variables. A pointer is set to point to this record. Individual variables of the program unit are accessed using displacement from this pointer.
- Program controlled dynamic allocation is implemented using a heap. A pointer is now needed to point to each allocated memory area.

Dynamic allocation provides some significant advantages. Recursion can be implemented easily because memory is allocated when a program unit is entered during execution. This leads to allocation of a separate memory area for each recursive activation of a program unit. Dynamic allocation can also support data structures whose sizes are determined dynamically, e.g. an array declaration a[m, n] where m and n are variables.

4.2.2 Memory Allocation in Block Structured Languages

A block is a program unit which can contain data declarations. A program in a block structured language is a nested structure of bocks. A block structured language uses dynamic memory allocation.

e.g. Algol-60 is the first widely used block structured language. The concept of block structure is also used in PL/I, Algol-60, Pascal, Ada and many other languages.

Scope rules

A data declaration using a name $name_i$ creates a variable var_i and establishes a binding between $name_i$ and var_i. We will represent this binding as$(name_i, var_i)$,and call it the name-var binding. Variable var_i is visible at a place in the program if some binding $(name_i , var_i)$ is effective at that place. A visible variable can be accessed using its name, $name_i$ in the above case. It is possible for data declarations in many blocks of program to use a same

name, say $name_i$. This would establish many bindings of the form($name_i$, var_k) for different values of k. Scope rules determine which of these bindings is effective at a specific place in the program.

Consider a block b which contains a data declaration using the name $name_i$. This establishes a binding($name_i$, var_i) for a variable var_i. This binding only exists within *block b.* If a block b' nested inside block b also contains a declaration using $name_i$, a binding($name_i$, var_j) exists in b' for some variable var_j. This suppresses the binding($name_i$, var_i) over block b'. Thus variable var_i is not visible in b'. However, if b' did not contain a declaration using $name_i$, the binding ($name_i$, var_i) would have been effective over b' as well. Thus var_i would have been visible in b'. We summarize the rules governing the visibility of a variable in the following.

Scope of a variable

If a variable var_i is created with the name $name_i$ in a block b,

1. var_i can be accessed in any statement situated in block b.
2. var_i can be accessed in any statement situated in block b' which is enclosed in b, unless b' contains a declaration using the same name(i.e. using the same name $name_i$).

A variable declared in block b is called a local variable of block b. A variable of an enclosing block, that is accessible within block b, is called a nonlocal variable of block b. The following notation is used to differentiate between variables created using the same name in different blocks:

$name_{block\_name}$: variable created by a data declaration using the name *name* in block block_name

This $alpha_A$, $alpha_B$ are variables created using the namealpha in blocks A and B.

Example 6: Consider the block-structured program in Figure 2. The variables accessible within the various blocks are as follows:

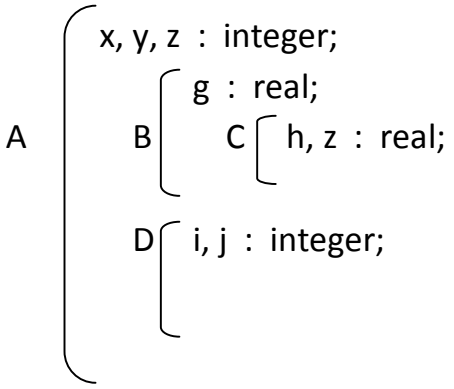


Figure 2: Block structured program

Block	Accessible variables	
	local	nonlocal
A	x_A, y_A, z_A	---
B	g_B	x_A, y_A, z_A
C	h_C, z_C	x_A, y_A, g_B
D	i_D, j_D	x_A, y_A, z_A

Variable z_A is not accessible inside block C since C contains a declaration using the name z. Thus z_A and z_C are two distinct variables. This would be true even if they had identical attributes, i.e. even if z of C was declared to be an integer.

Memory allocation and access

Automatic dynamic allocation is implemented using the extended stack model with a minor variation-each record in the stack has two reserved pointers instead of one as shown in figure 3. Each stack record accommodates the variables for one activation of a block, hence we call it an activation record (AR). The following notation is used to refer to the activation record of a block.

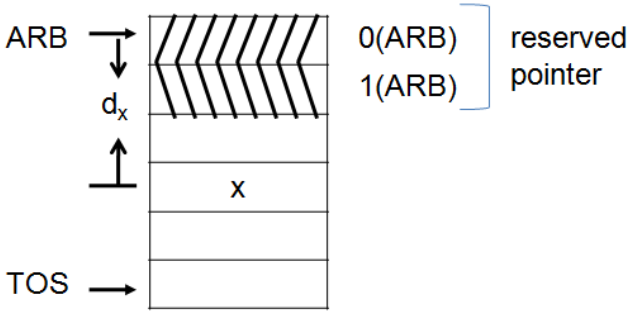


Figure 3: Stack record format

AR^i_A : Activation record for the i^{th} activation of A

wherein we omit the superscript i unless multiple activations of block exist. During the execution of a block structured program, a register called the activation record base (ARB) always points to the start address of the TOS record. This record belongs to the block which contains the statement being executed. A local variable x of this block is accessed using the address d_x(ARB), where d_x is the displacement of a variable x from the start of AR. The address may also be written as <ARB> +d_x , where <ARB> stands for the words 'contents of ARB'.

Dynamic pointer

The first reserved pointer in a block's AR points to the activation record of its dynamic pointer and has the address 0(ARB). The dynamic pointer is used for deallocating an AR. Actions at block entry and exit are analogous to those described in section 2.2.1.

Example 7: Figure 4 depicts memory allocation for the following program:

```
A   x : real;
    B   y : char;
        C   z,w : integer;

            z := 10;
            x := z;
```

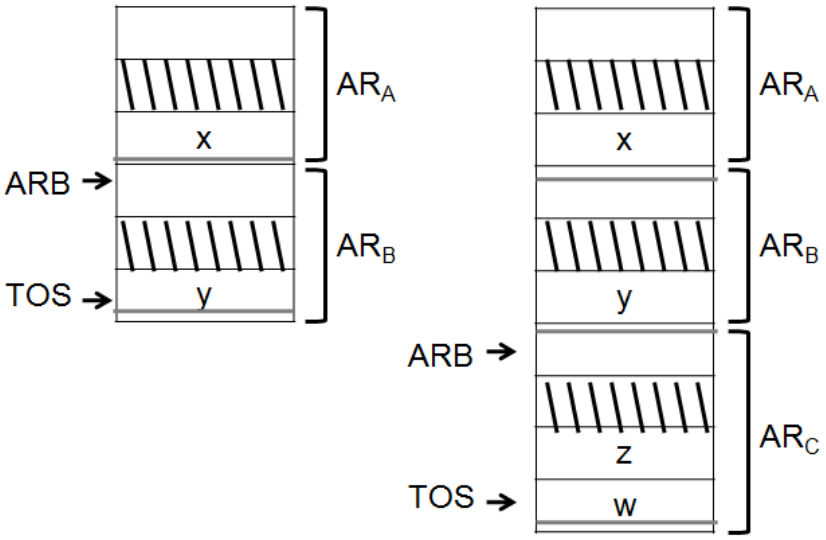


Figure 4: Dynamic allocation and access

where each block could be a begin block in Algol, PL/I or a procedure without parameters in some block structured language. It is assumed that the blocks are entered in the sequence A, B, C during the execution of the program.

Figure 6.4(a) shows the situation when blocks A and B are active. Figure 6.4(b) shows the situation after entry to C. Situation after exit from C would again be as shown in Fig 6.4(a).

Actions at entry

The actions at entry of block C are described in Table 1.

Table 1 Actions at block entry

No.			Statement
1.	TOS	:=	TOS + 1;
2.	TOS*	:=	ARB; {Set the dynamic pointer}
3.	ARB	:=	TOS;

4. TOS := TOS + 1;
5. TOS* := …..; {Set reserved pointer 2}
6. TOS := TOS + n;

where n is the number of memory words allocated to variables of block C, i.e. n = 2. After step 1. TOS points to the dynamic pointer of the AR being created. Step 2 sets the dynamic pointer to point to the previous activation record. Step 3 sets ARB to the start of new AR. Step 6 is the memory allocation step. Note that positions of local variables z and w in AR_C are determined during compilation using their type and length information. d_z and d_w are 2 and 3 assuming an integer to occupy 1 word. Thus the address of variable z is <ARB> + 2.

Actions at exit

The actions at entry of block C are described in Table 2.

Table 2 Actions at block exit

No.		Statement	
1.	TOS	:=	ARB – 1;
2.	ARB	:=	ARB*;

The first statement sets TOS to point to the last word of the previous activation record. The second statement sets ARB to point to the start of the previous activation record. These actions effectively deallocate the TOS AR in the stack.

4.3 COMPILATION OF EXPRESSIONS

The major issues in code generation for expressions are as follows:

1. Determination of an evaluation order for the operators in an expression.
2. Selection of instructions to be used in target program.
3. Use of registers and handling of partial results.

The evaluation order of operations depends on operator precedence in an obvious way-an operator which precedes its left and right neighbors must be evaluated before either of them. Hence, a feasible evaluation order is the order in which operators are reduced during a bottom up parse, or the reverse of the order in which operators are produced during a top down parse.

The choice of an instruction to be used in the target code depends on the following:

1. The type and length of each operand.
2. The addressability of each operand, i.e. where the operand is located and how it can be accessed

The addressability of an operand indicates where an operand is located and how it can be accessed.

A partial result is the value of some subexpression computed while evaluating an expression. In the interest of efficiency, partial results are maintained in CPU registers as far as possible. However, some of them have to be moved to memory if the number of results exceeds the number of available CPU registers. An improvement issue in the code generation is when and how to move partial results between memory and CPU registers, and how to know which partial result is contained in a register. We use a register descriptor to maintain information for the latter purpose.

Operand descriptors

An operand descriptor has the following fields:

1. **Attributes**: Contains the subfields type and length.
2. **Addressability**: Specifies where the operand is located, and how it can be accessed, It has the following two subfields:
    a) ***Addressability code***: This code takes the values 'M' (operand is in memory),and 'R' (operand in register).Other addressability such as address in a register('AR') and address in memory('AM') are also possible
    b) ***Address***: Address of a memory word or a CPU register.

An operand descriptor is built for every operand participating in an expression, i.e. for id's, constants and partial results. A descriptor is built for an id when the id is reduced during parsing. A partial result pr_i is the result of evaluating some operator op_j. A descriptor is built for pr_i immediately after code is generated for operator op_j. For simplicity we assume that all operand descriptors are stored in an array called Operand_descriptor. This enables us to designate a descriptor by its index in Operand_descriptor, e.g. descriptor #n is the descriptor in Operand descriptor[n].

Example 8: The code generated for the expression a * b is as follows:

```
MOVER    AREG, A
MULT     AREG, B
```

Three operand descriptors are used during code generation. Assuming a, b to be integers occupying 1 memory word, these are:

(int, 1)	M,addr(a)	Descriptor for a
(int, 1)	M,addr(b)	
(int, 1)	R, addr(AREG)	

A skeleton of the code generator is shown in Figure 5. Operand descriptor numbers are used as attributes of NT's. The routine build_descriptor called from the semantic action for the production <F>::= <id> builds a descriptor for <id> and returns its entry number in Operand_descriptor. The code generation routine generates code for an operator and builds an operand descriptor for the partial result. Semantic actions of other rules merely copy the operand descriptors as attributes of NT's.

```
%%
E   :   E + T  {$$ =  codegen( '+', $1, $3) }
    |   T      {$$ = $1}
    ;
T   :   T * F  {$$ = codegen( '*', $1, $3) }
    |   F      {$$ = $1}
```

```
        ;
F       :       id      {$$ = build_descriptor ($1) }
        ;
%%
Build_descriptor(operand)
{
        i= i + 1;
        operand_descr[i] = ((type), (addressability_code, address))
                                        of operand;
        return i;
}
```

Figure 5: Skeleton of the code generator

Register Descriptors

A register descriptor has two fields

1. Status: Contains the code *free* or *occupied* to indicate register status.
2. Operand descriptor #: If status = occupied, this field contains the descriptor # for the operand contained in the register.

Register descriptors are stored in an array called *Register_descriptor.* One register descriptor exists for each CPU register.

Example 9: The register descriptor for AREG after generating code for a*b in example 8 would be

Occupied	*#3*

This indicates that register AREG contains the operand described by descriptor #3.

Generating an instruction

When an operator op_i is reduced by the parser, the function codegen is called with op_i and descriptors of its operands as parameters.

- A single instruction can be generated to evaluate op_i if the descriptors indicate that one operand is in a register and the other is in memory.
- If both operands are in memory, an instruction is generated to move one of them into a register. This is followed by an instruction to evaluate op_i.

Saving partial results

If all registers are occupied (i.e. they contain partial results) when operator op_i is to be evaluated, a register r is freed by copying its contents into a temporary location in the memory. r is now used to evaluate operator op_i. For simplicity we assume that an array *temp* is declared in the target program(i.e. in the generated code) to hold partial results. A partial result is always stored in the next free entry of *temp*. Note that when partial result is moved to a temporary location, the descriptor of the partial result must change. The *operand descriptor #* field of the register descriptor is used to achieve this.

Example 10: Consider the expression a*b + c*d. After generating code for a*b, the operand and register descriptors would be as shown in Example 8 and 9. After the partial result a*b is moved to a temporary location, say temp [1], the operand descriptors must become

(int, 1)	M,addr(a)
(int, 1)	M,addr(b)
(int, 1)	M,addr(temp[1])

to indicate that the value of the operand described by operand descriptor #3 (viz. a*b) has been moved to memory location temp[1].

```
codegen(operator, opd1, opd2)
{
        if opd1.addressability_code = 'R' /* Code generation -- case 1 */
            if operator = '+' generate 'ADD AREG, opd2' ;
            /* Analogous code for other operators */
```

```
        else if opd2.addressability_code = 'R'  /* Code generation -- case 2 */
            if operator = '+' generate 'ADD AREG, opd1' ;
            /* Analogous code for other operators */
        else /* Code generation -- case 3 */
            if Register_descr.status = 'Occupied' /*save partial results */
            generate ( 'MOVEM AREG, Temp[j]' );
            j = j + 1;
            Operand_descr[Register_descr. Operand_descriptor#]
                = (<type>, (M, Addr (Temp[j])))
            /* Generate code */
            generate  'MOVER  AREG, opd1' ;
            if operator = '+' generate 'ADD AREG, opd2' ;
                /* Analogous code for other operators */
                /* common part – Create a new descriptor
                Saying operand value is in register AREG */
            i = i + 1;
            Operand_descr[i] = (<type>,('R', Addr(AREG)));
            Register_descr = ('Occupied', i);
            return i;
}
```

Figure 6: Code generation routine

Figure 6 shows the complete code generation routine. It first checks the addressability of its operands to decide whether one of them exists in a register. If so, it generates a single instruction to perform the operation. If none of the operands is in a register, it needs to move one operand into the register before performing the operation. For this purpose, it first frees the register if it is occupied and changes the operand descriptor of the result it contained previously. (Note how it uses register descriptor for this purpose.) It now moves one operand into the register and generates an instruction to perform the operation. At the end of code generation, it builds a descriptor for the partial result.

Example 11: Code generation steps for the expression a*b+c*d are shown in figure 7, where the superscript of an NT shows the operand descriptor # used as its attribute, and the notation $<id>_v$ represents the token <id> constructed for symbol v.

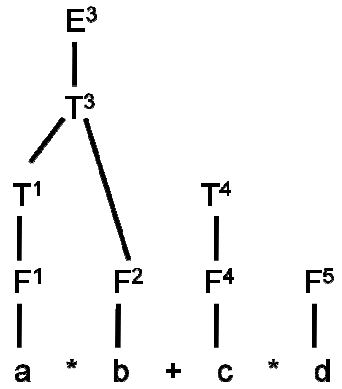
Figure 8(a) shows the parse tree and operand and register descriptor after step 8. Figure 8(b) shows the parse tree and descriptor after step 9. Note that operand descriptor #3 has been changed to indicate that the partial result a*b has been moved to the temporary location temp [1].

Step No.	Parsing action	Code generation action
1	$<id>_a \rightarrow F^1$	Build descriptor # 1
2	$F^1 \rightarrow T^1$	-
3	$<id>_b \rightarrow F^2$	Build descriptor # 2
4	$T^1 * F^2 \rightarrow T^3$	Generate MOVER AREG, A MULT AREG, B Build descriptor # 3
5	$T^3 \rightarrow E^3$	-
6	$<id>_c \rightarrow F^4$	Build descriptor # 4
7	$F^4 \rightarrow T^4$	-
8	$<id>_d \rightarrow F^5$	Build descriptor # 5

9	$T^4 * F^5 \rightarrow T^6$	Generate MOVEM AREG, TEMP_I
		MOVER AREG, C
		MULT AREG, D
		Build descriptor # 6

10	$E^3 + T^6 \rightarrow E^7$	Generate ADD AREG, TEMP_I

Figure 7: Code generation actions for a*b+c*d



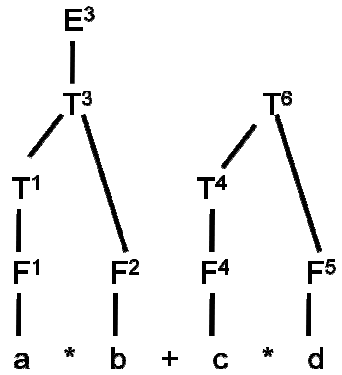
Operand Descriptor

1	(int, 1)	M, addr(a)	Descriptor for a
2	(int, 1)	M, addr(b)	Descriptor for b
3	(int, 1)	R, addr(AREG)	Descriptor for a*b
4	(int, 1)	M, addr(c)	Descriptor for c
5	(int, 1)	M, addr(d)	Descriptor for d

Register Descriptor

Occupied	#3

(a)



Operand Descriptor

1	(int, 1)	M, addr(a)	Descriptor for a
2	(int, 1)	M, addr(b)	Descriptor for b
3	(int, 1)	M, addr(temp[1])	Descriptor for a*b
4	(int, 1)	M, addr(c)	Descriptor for c
5	(int, 1)	M, addr(d)	Descriptor for d
6	(int, 1)	R, addr(AREG)	

Register Descriptor

Occupied	#6

(b)

Figure 8: Code generation actions

To see how temporary locations are used to hold partial results, consider the source string a*b+c*d*(e+f)+c*d. Figure 9 shows the expression tree for the string wherein operator numbers indicate the bottom up evaluation oder.

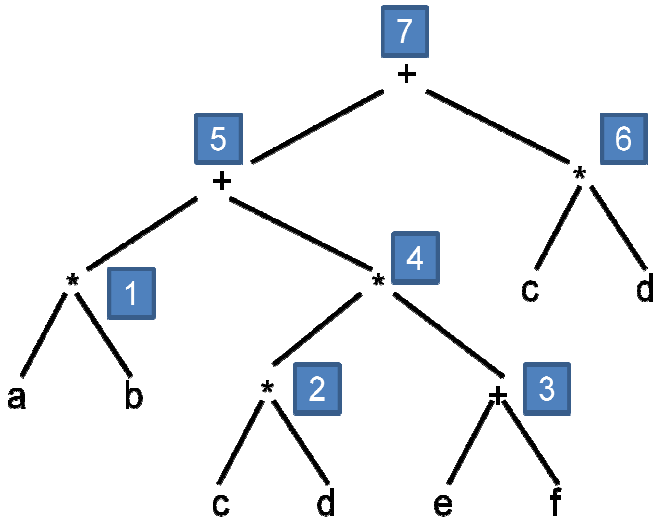
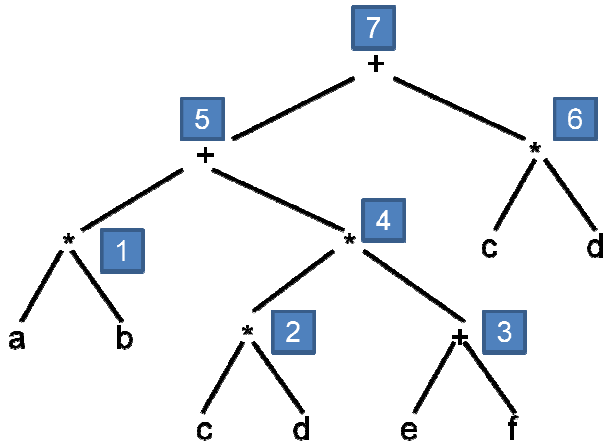


Figure 9: Expression tree

Fig. 10(a) shows the code generated for this string using the code generation routine of figure 6.

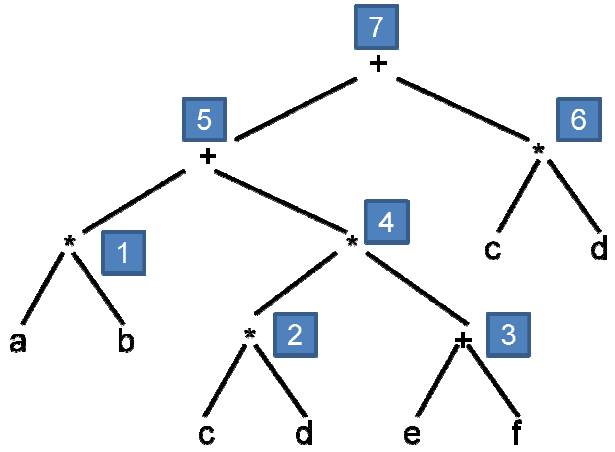
```
MOVER    AREG,    A
MULT     AREG,    B
MOVEM    AREG,    TEMP_1
MOVER    AREG,    C
MULT     AREG,    D
MOVEM    AREG,    TEMP_2
MOVER    AREG,    E
ADD      AREG,    F
MULT     AREG,    TEMP_2
ADD      AREG,    TEMP_1
MOVEM    AREG,    TEMP_3
MOVER    AREG,    C
MULT     AREG,    D
ADD      AREG,    TEMP_3
```



Temporary	Contents	Used in
temp[1]	Value of node 1	Evaluating node 5
temp[2]	Value of node 2	Evaluating node 4
temp[3]	Value of node 5	Evaluating node 7

(a)

```
MOVER   AREG,   A
MULT    AREG,   B
MOVEM   AREG,   TEMP_1
MOVER   AREG,   C
MULT    AREG,   D
MOVEM   AREG,   TEMP_2
MOVER   AREG,   E
ADD     AREG,   F
MULT    AREG,   TEMP_2
ADD     AREG,   TEMP_1
MOVEM   AREG,   TEMP_1
MOVER   AREG,   C
MULT    AREG,   D
ADD     AREG,   TEMP_1
```



Temporary	Contents	Used in
temp[1]	Value of node 1	Evaluating node 5
temp[2]	Value of node 2	Evaluating node 4
Temp[1]	Value of node 5	Evaluating node 7

(b)

Figure 10: Illustration for temporary location usage

Figure 10(b) shows the code for the expression resulting from the reuse of *temp[1]*.

To implement stack of temporaries, variable i of routine codegen is used as a stack pointer. Now, i need to be decremented whenever an operand of an instruction is a temporary location.

4.3.2 Intermediate Codes for Expressions

Postfix strings

In the postfix notation, each operator appears immediately after its last operand. Thus, a binary operator op_i appears after its second operand. If any of its operands is itself an expression involving an operator op_j, op_j must appear op_i. Thus operators can be evaluated in the order in which they appear in the string.

Example 12: Consider the following strings:

2 1 5 4 3

Source string: |- a + b * c + d * e + f -|

1 2 3 4 5

Postfix string: |- a b c * + d e f ↑ * + -|

The numbers appearing over the operators indicates their evaluation order. The second operand of the operator '+' marked 2 in the source string is the expression b*c. Hence its operands b, c and its operator '*' appear before '+' in the postfix string.

The postfix string is a popular intermediate code in non optimizing compilers due to ease of generation and use. We perform code generation from the postfix string using a stack of operand descriptors.

- Operand descriptors are pushed on the stack as operand appear in the string.
- When an operator with an arity *k* appears in the string *k* descriptors are popped off the stack.
- A descriptor for the partial result generated by the operator is now pushed on the stack.

Thus in example 12, The operand stack would contain descriptors for a, b and c when '*' is encountered. The stack contains the descriptors for a and the partial result b*c when '+' is encountered.

Triples and quadruples

A triple is a representation of an elementary operation in the form of pseudomachine instruction

Operator	Operand1	Operand2

Triples are numbered in some convenient manner. Each operand of triple is either a variable/constant or the result of some evaluation represented by another triple. In the latter case, the operand field contains that triples numbers.

Example 13: Figure 11 contains the triples for the expression string a + b * c + d * e + f
In the operand field of triple 2 indicates that the operand is the value of b* c represented by triple number 1.

	Operator	Operand1	Operand2
1	*	b	c
2	+	1	a
3	↑	e	f
4	*	d	3
5	+	2	4

Figure 11: Triples for a + b * c + d * e + f

A program representation called *indirect triples* is useful in optimizing compilers. In this representation, a table is built to contain all distinct triples in the program. A program statement is represented as a list of triple numbers. This arrangement is useful to detect the occurrences of identical expressions in a program. For efficiency reasons, a hash organization can be used for the table of triples. The indirect triples representation provides memory economy. It also aids in certain forms of optimization, viz. common subexpression elimination.

Example 14: Figure 12 shows the indirect triple representation for the program segment

z := a+b*c+d*e↑f;
y := x+b*c;

Use of b*c in both statements is reflected by the fact that triple number 1 appears in the list of triples for both statements.

	Operator	Operand1	Operand2
1	*	b	c
2	+	1	a
3	↑	e	f
4	*	d	3
5	+	2	4
6	+	x	1

Statement no.	Triple no.
1	1, 2, 3, 4, 5
2	1, 6

Figure 12: Indirect triples

A *quadruple* represents an elementary evaluation in the following format:

Operator	Operand1	Operand2	Result name

Here, result name designates the result of the evaluation. It can be used as the operand of another quadruple. This is more convenient than using a number to designate a subexpression.

	Operator	Operand1	Operand2	Result name
1	*	b	c	t1
2	+	1	a	t2
3	↑	e	f	t3
4	*	d	3	t4
5	+	2	4	t5

Figure 13: Quadruples

Example 15: Quadruples for the expression string a+b*c+d*e+f are shown in figure 13. $t_1,t_2,….t_5$ in figure 13 are not temporary locations for holding partial results. They are result names. Some of these become temporary locations when common subexpression elimination is implemented.

Expression trees

We have so far assumed that operators are evaluated in the order determined by a bottom up parser. This evaluation order may not lead to the most efficient code for an expression. Hence a compiler back end must analyze an expression to find the best evaluation order for its operators. An expression tree is an abstract syntax tree which depicts the structure of an expression. This representation simplifies the analysis of an expression to determine the best evaluation order.

Example 16: Figure 14 shows two alternative codes to evaluate the expression (a+b)/(c+d). The code in part (b) uses fewer MOVER/MOVEM instructions. It is obtained by deviating from the evaluation order determined by a bottom up parser.

```
MOVER    AREG, A          MOVER    AREG, C
ADD      AREG, B          ADD      AREG, D
MOVEM    AREG, TEMP_1     MOVEM    AREG, TEMP_1
MOVER    AREG, C          MOVER    AREG, A
ADD      AREG, D          ADD      AREG, B
 MOVEM   AREG, TEMP_2     DIV      AREG, TEMP_1
MOVER    AREG, TEMP_1
DIV      AREG, TEMP_2
         (a)                       (b)
```

Figure 14: Alternative codes for (a+b)/(c+d)

A two step procedure is used to determine the best evaluation order for the operations in an expression.

- The first step associates a *register requirement label* (RR label) with each node in the expression. It indicates the number of CPU registers required to evaluate the

subtree rooted at the node without moving partial result to memory. Labeling is performed in a bottom up pass of the expression tree.

- The second step, which consists of a top down pass, analyses the RR labels of the child nodes of a node to determine the order in which they should be evaluated.

Algorithm 1 (Evaluation order for operators)

1. Visit all nodes in an expression tree in post order (i.e., such that a node is visited *after* all its children).

For each node n_i

(a) If n_i is a leaf node then

if n_i is the left operand of its parents then $RR(n_i) := 1$;

else $RR(n_i) := 0$;

(b) If n_i is not a leaf node then

If $RR(l\_child_{ni}) \neq RR(r\_child_{ni})$ then

$RR(n_i) := \max (RR(r\_child_{ni})), RR(l\_child_{ni}))$;

Else $RR(n_i) := RR(l\_child_{ni}) + 1$;

2. Perform the procedure call *evaluation_order* (root) (see Fig 6.23), which prints a postfix form of the source string in which operators appears in the desired evaluation order.

```
Procedure evaluation_order(node);
    if node is not a leaf node then
        if RR(l_child_node) ≤ RR(r_child_node)
            evaluation_order(r_child_node);
            evaluation_order(l_child_node);
        else
            evaluation_order(l_child_node);
            evaluation_order(r_child_node);
    print node;
end evaluation_order;
```

Figure 15: Procedure evaluation_order

Let RR=q for the root node. This implies that the evaluation order can evaluate the expression without moving any particular result(s) to memory if q CPU registers are available. It thus provides the most efficient way to evaluate the expression. When the number of available registers < q, some partial results have to be saved in memory. However, the evaluation order still leads to the most efficient code.

Example 17: Figure 16 shows the expression tree for the string f+(x+y)*((a+b)/(c-d)). The boxed numbers indicate operator positions in the source string. The RR label of each node is shown as the superscript of the operand or operator at that node. The evaluation order according to algorithm 1 is 6,4,5,2,3,1. This evaluation order is determined as follows: When *evaluation_order* is called with the root node i.e. operator 1, as the parameter, it decides that the right child of the node should be evaluated before its left child since RR(3) > RR(node for f). This leads to recursive call with node(3) as the parameter . Here RR(5) > RR(2) leads to recursive call with node(5) as the parameter. At node(5) also decision is made to visit its right child before its left child. This leads to the postfix string cd-ab+/xy+*f+ in which operators appear in the evaluation order mentioned above.

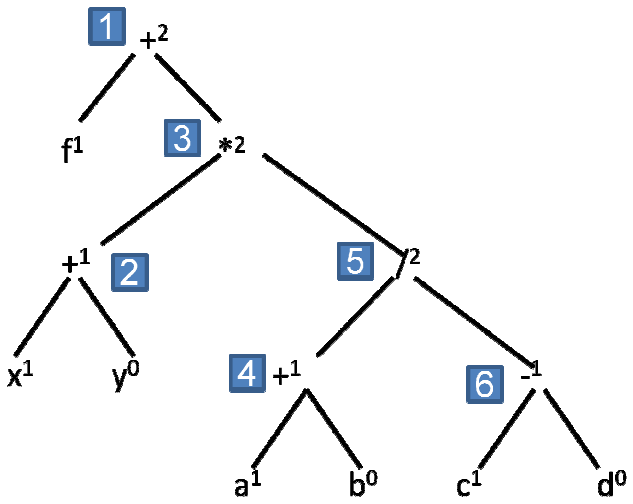


Figure 16: Register requirement labels

4.4 CODE OPTIMIZATION

Code optimization aims at improving the execution efficiency of a program. This is achieved in two ways:

1. Redundancies in a program are eliminated.
2. Computations in a program are rearranged or rewritten to make it execute efficiently.

It is axiomatic that code optimization must not change the meaning of a program. Two points concerning the scope of optimization should also be noted.

- First, optimization seeks to improve a program rather than the algorithm used in a program. Thus replacement of an algorithm by a more efficient algorithm is beyond the scope of optimization.
- Second, efficient code generation for a specific target machine (e.g., by fully exploiting its instruction set) is also beyond its scope; it belongs in the back end of a compiler. The optimization techniques are thus independent of both the PL and the target machine.

Figure 17 contains a schematic of an optimizing compiler. The front end generates an IR which could consist of triples, quadruples or ASTs. The optimization phase transforms this to achieve optimization. The transformed IR is input to the back end.

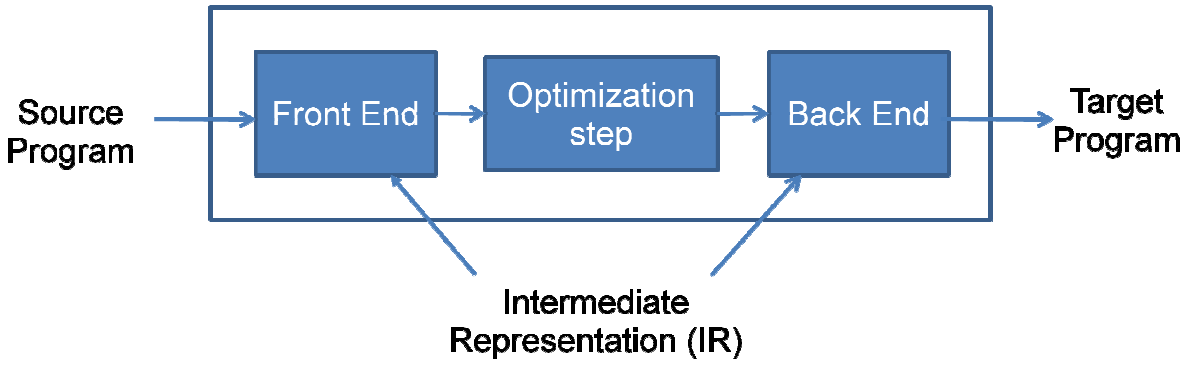


Figure 17: Schematic of an optimizing compiler

4.4.1 Optimizing Transformations

An Optimizing Transformation is a rule for rewriting a segment of a program to improve its execution efficiency without affecting its meaning. Optimizing transformations are classified into local and global transformations depending on whether they are applied over small segments of a program consisting of a few source statements or over larger segments consisting of loops or function bodies. The reason for this distinction is the difference in the costs and benefits of the optimizing transformations.

A few optimizing transformations commonly used in the compilers are discussed below.

- **Compile time evaluation**
- **Elimination of common subexpressions**
- **Dead code elimination**
- **Frequency reduction**
- **Strength reduction**

Compile time evaluation

Execution efficiency can be improved by performing certain actions specified in a program during compilation itself. This eliminates the need to perform them during execution of the program, thereby reducing the execution time of the program. Constant folding is the main optimization of this kind. When all operands in an operation are constants, the operation can be performed at compilation time. The result of the operation, also a constant, can replace original evaluation in the program.

Example 18: An assignment a:= 3.14157/2 can be replaced by a:= 1.570785, thereby eliminating a division operation.

An instance of compile time evaluation can be found in array address arithmetic.

Elimination of common subexpressions

Common Subexpressions are occurrences of expressions yielding the same value.

Let CS_i designate a set of common subexpressions. It is possible to eliminate an occurrence $e_j \in CS_i$ if, no matter how the evaluation of e_j is reached during the execution of the program, the value of some $e_k \in CS_i$ would have been already computed. Provision is made to save this value and use it at the place of occurrence of e_j.

Example 19:

```
a := b * c;
-----
x := b * c + 5.2;
```

After eliminating the common subexpression

```
t := b * c;
a := t;
-----
x := t + 5.2;
```

Here CS_i contains the two occurrences of b*c. The second occurrence of b*c can be eliminated because the first occurrence of b*c is always evaluated before the second occurrence is reached during execution of the program. The value computed at the first occurrence is saved in t. This value is used in the assignment to x.

Dead code elimination

Code which can be omitted from a program without affecting its results is called dead code. Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program.

Example 20: An assignment x := <exp> constitutes dead code if the value assigned to x is not used in the program, no matter how control flows after executing this assignment. Note that <exp> constitutes dead code only if its execution does not produce side effects, i.e. only if it does not contain function or procedure calls.

Frequency reduction

Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times. For example, the transformation of *loop optimization* moves loop invariant code out of a loop and places it prior to loop entry.

Example 21:

```
for I := 1 to 100 do
begin
        z := i;
        x := 25 * a;
        y := x + z;
end;
```

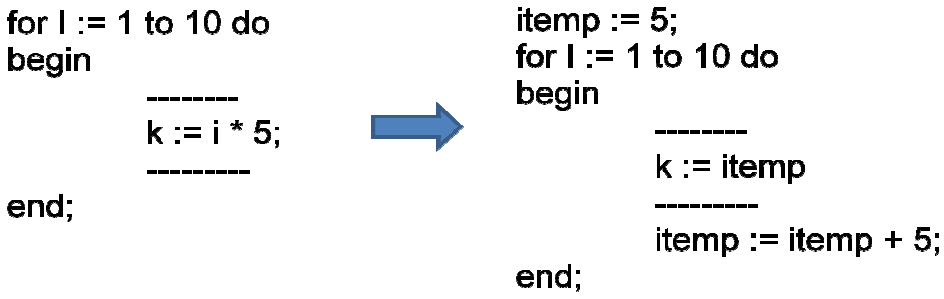
```
x := 25 * a;
for I := 1 to 100 do
begin
        z := i;
        x := 25 * a;
        y := x + z;
end;
```

Here x :=25*a; is loop invariant. Hence in the optimized program it is computed only once before entering the for loop. y := x+z; is not loop invariant. Hence it cannot be subjected to frequency reduction.

Strength reduction

The strength reduction optimization replaces the occurrences of a time consuming operation (a 'high strength' operation) by an occurrence of a faster operation(a 'low strength' operation), e.g. replacement of a multiplication by an addition.

Example 22: In the following example, the 'high strength' operator '*' in i*5 occuring inside the loop is replaced by a low strength operator '+' in temp+5.



Strength reduction is very important for array access occurring within program loops. For example, an array reference a[i, j] within a Pascal loop gives rise to the high strength computation i*n in the address arithmetic, where n is the number of rows in the array. Strength reduction optimizations is not performed on operations involving floating point operands because finite precision of floating point arithmetic cannot guarantee of results after strength reduction.

Local and global optimization

Optimization of a program is structured into the following two phases:

1. Local optimization: The optimizing transformations are applied over small segments of a program consisting of a few statements.

2. Global optimization: The optimizing transformations are applied over a program unit, i.e. over a function or a procedure.

Local optimization is preparatory phase for global optimization. It can be performed by the front end while converting the source program into the IR.

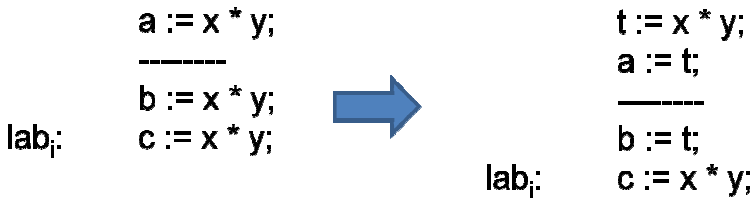
4.5.2 Local Optimization

Local optimization provides limited at a low cost. The scope of local optimization is a basic block which is an 'essentially sequential' segment in the source program. The cost of local optimization is low because the sequential nature of the basic block simplifies the analysis needed for optimization. The benefits are limited because certain optimizations, e.g. loop optimization, are beyond the scope of local optimization.

Basic block: A Basic block is a sequence of program statements ($s_1,s_2,….,s_n$) such that only s_n can be a transfer of control statement and only s_1 can be the destination of a transfer of control statement.

A basic block b is a program segment with a single entry point. If control reaches statement s_1 during program execution, all statement $s_1,s_2,….,s_n$ will be executed. The 'essentially sequential' nature of a basic block simplifies optimization.

Example 23: Consider the following program segment:



where lab_i is a label. Local optimization identifies two basic blocks in the program. The first block extends up to the statement b := x*y; Its optimization leads to elimination of the second occurrence of x*y. The third occurrence is not eliminated because it belongs to a different basic block. If the label *lab_i* did not exist, the entire program segment would

constitute a single basic block and the third occurrence x*y can also be eliminated during local optimization.

Value Numbers

Value numbers provide a simple means to determine if two occurrences of an expression in a basic block are equivalent. The value numbering technique is applied on the fly while identifying basic blocks in a source program. A value number vn_{alpha} is associated with variable alpha. It identifies the last assignment to alpha processed so far. Thus, the value number of a variable alpha changes on processing an assignment alpha := .... Now two expressions ei and ej are equivalent if they are congruent and their operands have the same value numbers.

For simplicity all statements of a basic block are numbered in some convenient manner. If statement n, the current statement being processed, is an assignment to alpha, we set vn_{alpha} to n. A new field is added to each symbol table entry to hold the value number of a variable. The IC for a basic block is a list of quadruples stored in a tabular form. Each operand field in s quadruple holds the pair(operand, value number). A boolean flag *save* is associated with each quadruple to indicate whether its value should be saved for use elsewhere in the program. The flag is initialized to *false* in every new quadruple entered in the table.

Consider the symbol table and the quadruple table during local optimization of the following program:

stmt no.	statement
14	g := 25.2;
15	x := z+2;
16	h := x*y+d;
..	...
34	w := x*y

Symbol	….	Value number
y		0
x		15
g		14
z		0
d		5
w		0

	operator	**Operand 1**		**Operand2**		**Result name**	**Use flag**
		operand	**Value no.**	**Operand**	**Value no.**		
20	:=	g	--	25.2	--	t20	*f*
21	+	z	0	2	--	t21	*f*
22	:=	x	0	t21	--	t22	*f*
23	*	x	15	y	0	t23	*f t*
24	+	t23	--	d	5	t24	*f*
	..						
57	:=	w	w	t23	--	t57	*f*

Figure 18: Local optimization using value numbering

Local optimization proceeds as follows: All variables are assumed to have the value numbers '0' to start with. Processing of statements 14 and 15 leads to generation of quadruples numbered 20-22 shown in the table. Value numbers of g, x and y at this stage are 14, 15 and 0, respectively. Quadruple x* y is generated next, and the value numbers of x and y are copied from the symbol table. This quadruple is assigned the result name t23, and its save flag is set to false. This quadruple is entered in entry number 23 of the quadruple table. When the statement w := x*y (i.e. statement 34) is processed, the

quadruple for x*y is formed using the value numbers found in the symbol table entries of x and y. Since these are 15 and 0, the new quadruple is identical with quadruple 23 in the table. Instead, the save flag of quadruple 23 is set to true. The only quadruple generated for this statement is therefore the assignment to w(quadruple number 57). While generating code for quadruple 23 its save flag indicates that the value of expression x*y needs to be saved in a temporary location for later use. t23 can itself become the name of this temporary location.

This schematic can be easily extended to implement constant propagation, which is the substitution of a variable *var* occurring in a statement by a constant const, and constant folding. When an assignment of the form *var* := const is encountered, we enter const into a table constants, say entry n, and associates the value number '-n' with var. Constant propagation and folding is implemented while generating a quadruple if each operand is either a constant or has a negative value number.

Example 24: In the following program, variable a is given a negative value number, say '-10', on processing the first statement.

```
a := 27.3;                    a := 27.3;
-----            ->           ------
b := a*3.0;                   b := 81.9;
```

This leads to the possibility of constant propagation and folding in the third statement. The value of a, viz. '27.3', is obtained from the 10^{th} entry of constants table. Its multiplication with 3.0 yields 81.9. This value is treated as a new constant and a quadruple for b := 81.9 is now generated.

4.4.3 Global optimization

Compared to local optimization, global optimization requires more analysis effort to establish the feasibility of an optimization. Consider global common subexpression elimination. If some expression x*y occurs in a set of basic blocks SB of program P, its occurrence in a block $b_j \in$ SB can be eliminated if the following two conditions are satisfied for every execution of P:

1. Basic block b_j is executed only after some block $b_k \epsilon$ SB has been executed one or more times.
2. No assignment to x or y have been executed after the last (or only) evaluation of x*y in block b_k.

} **GO**

Condition 1 ensures that x*y is evaluated before execution reaches block b_j, while condition 2 ensures that the evaluated value is equivalent to the value of x*y in block b_j. The optimization is realized by saving the value of x*y in a temporary location in all blocks b_k which satisfy condition 1.

To ensure that *every possible execution* of program P satisfies condition 1 and 2, the program is analysed using the techniques of control flow analysis and data flow analysis.

4.4.3.1 Program Representation

A program is represented in the form of a program flow graph.

Program flow graph(PFG): A program flow graph for a program P is directed graph $G_p = (N, E, n_0)$ where

N : set of basic blocks in P

E : set of directed edges (b_i, b_j) indicating the possibility of control flow from the last statement of b_i (the source node) to the first statement of b_j (the destination node)

n_0 : start node of P

A basic block, which is a sequence of statements $s_1,s_2,\ldots s_n$, is visualized as a sequence of *program points* $p_1,p_2,\ldots,p_n$ such that a statement s_i is said to exist at program point p_i. This notion is used to differentiate between different occurrences of identical statements. For example, an occurrence of *e* at program point p_i is distinct from the occurrence of *e* at program point p_j.

4.4.3.2 Control and Data Flow Analysis

The technique of control and data flow analysis is together used to determine whether the Conditions governing an optimizing transformation are satisfied in a program.

Control flow analysis

Control flow analysis analyses a program to collect information concerning its structure, e.g. presence and nesting of loops in the program. Information concerning structure is used to answer specific question of interest. The control flow concepts of interest are:

1. ***Predecessors and successors*** : If $(b_i, b_j) \in E$, b_i is a *predecessor* of b_j and b_j is *successor* of b_i.
2. ***Paths :*** A path is a sequence of edges such that the destination node of one edge is the source node of the following edge.
3. ***Ancestors and descendants :*** If a path exist from b_i to b_j, b_i is ancestor of b_j and b_j and b_j is a descendant of b_i.
4. ***Dominators and post-dominators:*** Block b_i is a dominator of block b_j if every path from n_0 to b_j passes through b_i. b_i is a post-dominator of b_j if every path from b_j to an exit node passes through b_i.

Data flow analysis

Data flow analysis techniques analyze the use of data in a program to collect information for the purpose of optimization. This information, called data flow information, is computed at the entry and exit of each basic block in G_p. It is used to decide whether an optimizing transformation can be applied to a segment of code in the program.

Design of the global optimization phase begins with the identification of an appropriate data flow concept to support the application of each optimizing transformation. The data flow information concerning a program entity—For example a variable or an expression—is now a Boolean value indicating whether the data flow concept is applicable to that entity. Following table contains a summary of important data flow concepts used in optimization.

Data flow concept	Optimization in which used
Available expression	Common subexpression elimination
Live variable	Dead code elimination
Reaching definition	Constants and variable propagation

The first two data flow concepts are discussed in details:

- **Available expressions**
- **Live variables**

Available expressions

The transformation of global common subexpression elimination can be defined as follows: Consider a subexpression x*y occurring at program point p_i in a basic block b_i. These occurrences can be eliminated if

1. Conditions 1 and 2 of (GO) are satisfied at entry to b_i.
2. No assignments to x or y precede the occurrence of x*y in b_i.

The data flow concept of available expressions is used to implement common subexpression elimination. An expression *e* is available at program point p_i if a value equivalent to its value is always computed before program execution reaches p_i. Thus the concept of available expressions captures the essence of Conditions 1 and 2 of (GO). The availability of an expression at entry or exit of basic block b_i is computed using the following rules.

1. Expression *e* is available at the exit of b_i if

   (i) b_i contains an evaluation of *e* which is not followed by assignments to any operands of *e*, or

   (ii) the value of *e* is available at the entry to b_i and b_i does not contain assignments to any operands of *e*.

2. Expression *e* is available at entry to b_i if it is available at the exit of each predecessor of b_i in G_p.

Available expression is termed a forward data flow concept because availability at the exit of node determines availability at the entry of its successor(s). It is an all paths concept because availability at entry of a basic block requires availability at the exit of all predecessors.

We use the Boolean variables $Avail\_in_i$ and $Avail\_out_i$ to represent the availability of an expression e at entry and exit of basic block b_i, respectively.

Example 25: Available expression analysis for the PFG of Fig. 6.31 gives the following results:

a*b : Avail_in = true for blocks 2,5,6,7,8,9
Avail_out = true for blocks 1,2,5,6,7,8,9,10
x+y : Avail_in = true for blocks 6,7,8,9
Avail_out = true for blocks 5,6,7,8,9

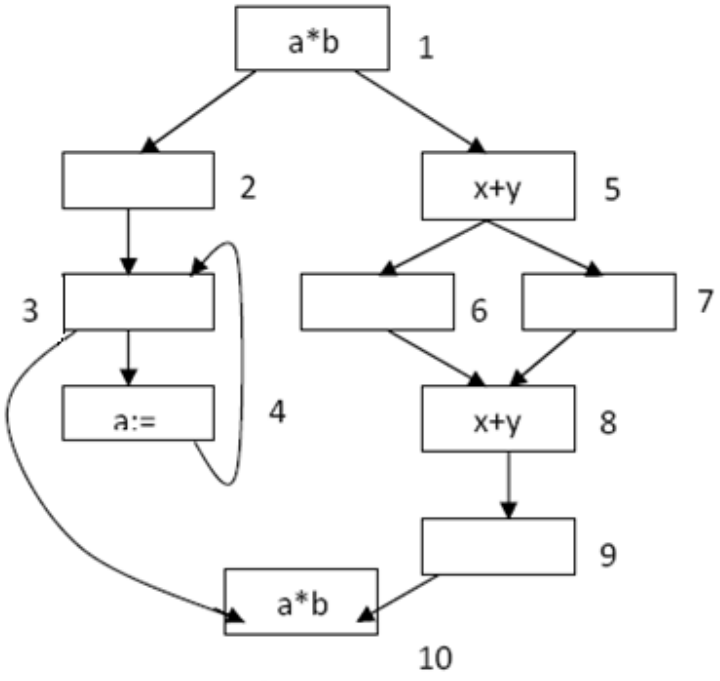


Figure 19: A PFG

The values of Avail_in and Avail_out for a*b can be explained as follows: The assignment a := .. in block 4 makes $Avail\_out_4$ = false. This makes $Avail\_in_3$ = false , which makes $Avail\_out_3$ =$Avail\_in_{10}$ =false. For x*y, $Avail\_in_1$ = false makes $Avail\_out_1$ = false, which makes Avail_in = Avail_out =false for blocks 2,3,4 and 10.

Live variables

A variable var is said to live at a program point p_i in basic block b_i if the value contained in it at p_i is likely to be used during subsequent execution of the program. If var is not live at program point which contains a definition var := .., the value assigned to var by this definition is redundant in the program. Such a definition constitutes dead code which can be eliminated from the program without changing its meaning.

The liveness property of a variable can be determined as follows:

1. Variable v is live at the entry of b_i if
    (i) b_i contains a use of e which is not preceded by assignment(s) to v,or
    (ii) v is ,ive at the exit of b_i and b_i does not contain assignment(s) to v.
2. v is live at exit of b_i if it is live at the entry of some successor of b_i in G_p.

Live variables are termed a backward data flow concept because availability at the entry of block determines availability at the exit of its predecessor(s). It is an any path concept because liveness at the entry of one successor is sufficient to ensure liveness at the exit of a block.

Example 26: In the PFG of Figure 19, variable b is live at the entry of all blocks, a is live at the entry of all blocks excepting block 4 and variables x, y are live at the entry of blocks 1,5,6,7 and 8.