UNIT-VI

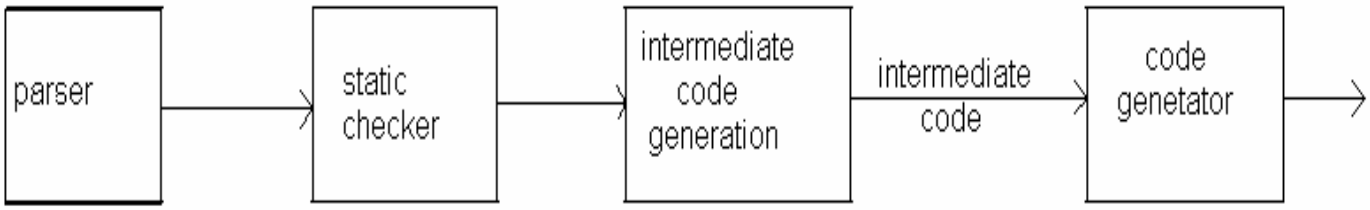
Intermediate Code Generation

Intermediate Code is generated using the Parse rule Producing a language from the input language.

Why do we need intermediate code?

-intermediate code has the following property – simple enough to be Translated to assembly code .

-complex enough to capture the complication of high level language.



Types Of Intermediate Representation:

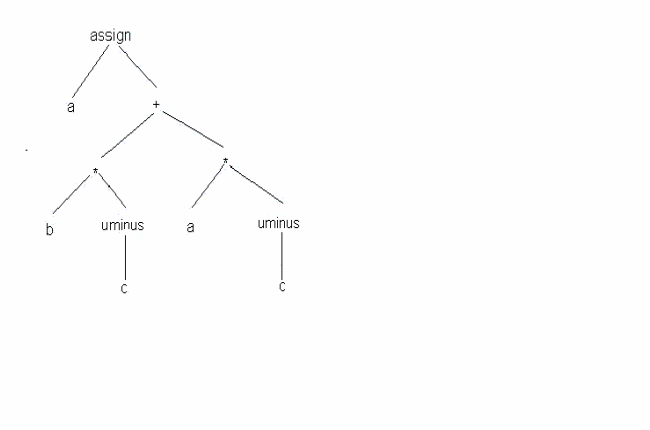
Syntax tree
Postfix notation
Three address Code

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

Graphical Representations

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 sub-expressions are identified. A syntax tree for
the assignment statement a:=b*-c+b*-c appear in the figure.

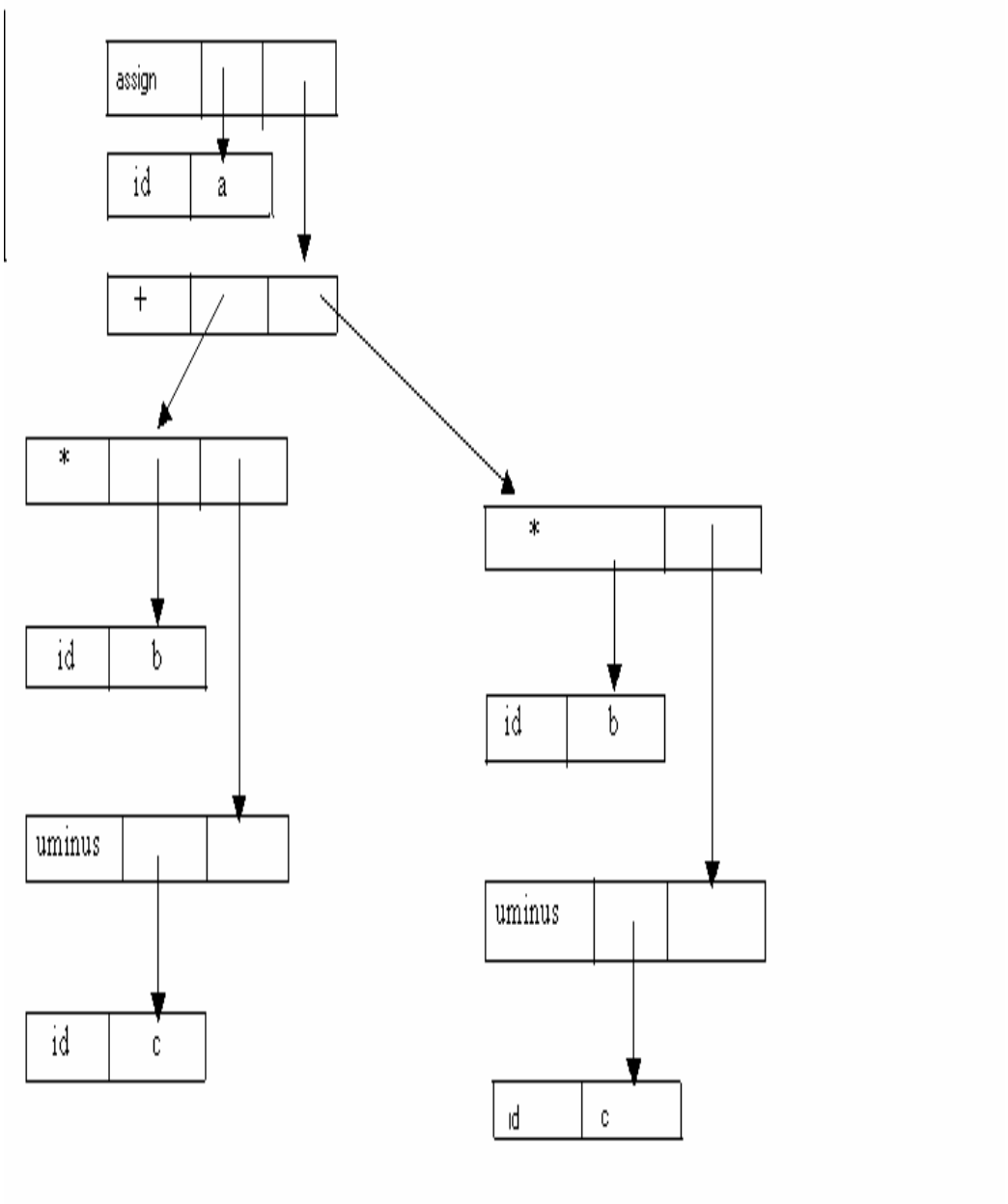
Syntax Tree:



Syntax tree for assignment statements are produced by the syntax directed definition

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

0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	……		



2 representations of the Syntax tree

Three Address Code Generation

What is a Three Address code?

Three address code is a sequence of statements of the general form **x = y op z** ,where x , y , z are names, constants and op stands for any operator such as fixed or floating point arithmetic operator or a logical operator on boolean valued data. Here three address refers to three addresses ie addresses of x, y and z.

Say for example we have a statement like a = b + c * d then we can make a three address code for it as follows:
t1 = c * d;
a = b+ t1;

Types of Three address statements:

There are different types of three address statements. Some of them are as follows :-

- Assignment statements. They are of the form **x := y op z** where op is a binary arithmetic or logical operation

- Assignment Instructions. They are of the form **x := op y** where op is an unary operation like unary plus, unary minus shift etc....

- Copy statements. They are of the form **x := y** where the value of y is assigned to x

- Unconditional Jump **goto L**. The three address statement with label L is the next to be executed.

• Conditional Jumps such as **if x relop y goto L**. This instruction applies a relational operator (<,>,<=,>=) to x and y and executes the statement with label L if the conditional statement is satisfied. Else the statement following if x relop y goto L is executed

• **param x** and **call p,n** for procedure calls and return y where y representing a returned value (optional).Three Address statements for it are as follows.

```
param x1
param x2
param x3
  .
  .
param xn
call p,n
```

generated as a part of the three address code for call of the procedure p(x1,x2,x3,....xn)
where n are the number of variables being sent to the procedure.

Implementation of Three-Address Statements:

Eg. a := b * -c + b * -c

Quadruples:(easy to rearrange code for global optimization, lots of temporaries)

#	Op	Arg1	Arg2	Res
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3

(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Triples: (temporaries are implicit, difficult to rearrange code)

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Indirect Triples: (temporaries are implicit & easier to rearrange code.

#	Stmt		#	Op	Arg1	Arg2
(0)	(14)	→	(14)	uminus	c	
(1)	(15)	→	(15)	*	b	(14)
(2)	(16)	→	(16)	uminus	c	
(3)	(17)	→	(17)	*	b	(16)
(4)	(18)	→	(18)	+	(15)	(17)
(5)	(19)	→	(19)	:=	a	(18)

Program container **Triple**

Consider the following grammar with its translation scheme for producing 3-address statements

$S \rightarrow$ id:= E	S.Code := E.Code ‖ gen(id.place '=' E.place)
$E \rightarrow E + E$	E.place := newtemp( ) , E.Code =E_1.Code ‖ E_2.Code ‖ gen(E.place '=' E_1.place + E_2.place)
$E \rightarrow E * E$	E.place = newtemp( ) , E.Code := E_1.Code ‖ E_2.Code ‖ gen(E.place '=' E_1.place * E_2.place)
$E \rightarrow -E$	E.place = newtemp( ), E.Code := E_1.Code ‖ gen (E.place '=' 'uni – 'E_1.place)
$E \rightarrow (E_1)$	E.place = E_1.place , E.Code = E_1.Code
$E \rightarrow$ id	E.place = id.place , E.Code := ' '

Consider the expression x := (y + z) * (-w + v)

Three-address code statements for the above expression will be as follows

- t1 = y +z
- t2 = -w
- t3 = t2 +v
- t4 = t1 * t3
- x = t4

Syntax directed translation for the above expression:

1. E→id | E.place := y
2. E→id | E.place := z
3. E→E + E | E.place := t1 E.Code := gen(t1 = y + z)
4. E→(E) | E.place := t1 E.Code := gen(t1 = y + z)
5. E→id | E.place := w
6. E→ -E | E.place := t2 E.Code := gen(t2 = -w)
7. E→id | E.place := v
8. E→E + E | E.place := t3 E.Code := gen(t2 = -w , t3 = t2 +v)
9. E →(E) | E.place := t3 E.Code := gen(t2 = -w , t3 = t2 + v)
10. E→E * E | E.place = t4 E.Code := gen(t1= y + z , t2 = -w , t3 = t2 + v , t4 = t1 * t3)
11. S→id:=E | S.Code := gen(t1= y + z , t2 = -w , t3 = t2 + v , t4 = t1 * t3 , x=t4)

Where

- *S.code* represents the three address code for assignment S
- Non terminal E has 2 attributes
- E.place, the name that will hold the value of E
- E.code, the sequence of three-address statements evaluating E
- The function *newtemp* returns a sequence of distinct names t1,t2 … in successive calls

THREE-ADDRESS CODE GENERATION

The Translation Scheme for Addressing Array Elements:

- Semantic actions will be added to the following grammar:

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

- The three-address code is produced by the *emit* procedure which is invoked in the semantic actions.
- Normal assignment is generated if L is a simple name.
- Indexed assignment is generated into the location denoted by L otherwise.

Grammar Rule	**Semantic Actions**
1. S → 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 → E1 + E2	*{ E.place : = newtemp;* *emit(E.place ‘: =’ E1.place ‘+’ E2.place) }*
3. E → (E1)	*{ E.place : = E1.place }*
4. E → 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 → Elist]	*{ L.place : = newtemp* *L.offset : = newtemp* *emit(L.place ‘: =’ c(Elist.array))* *emit(L.offset ‘: =’ Elist.place ‘*’ width(Elist.array)) }*
6. L → ***id***	*{ L.place : =* ***id****.place* *L.offset : =* ***null*** *}*
7. Elist → Elist1 , E	*{ t : = newtemp* *m : = Elist1.ndim + 1* *emit(t ‘: =’ Elist1.place ‘*’ limit(Elist1.array,m))* *emit(t ‘: =’ t ‘+’ E.place)* *Elist.array : = Elist1.array* *Elist.place : = t* *Elist.dim : = m }*

8. Elist → ***id*** *[E*	*{ Elist.array : = **id**.place* *Elist.place : = E.place* *Elist.dim : = 1 }*

Formula to remember
In more simple terms

1) *((i1*n2)+i2)*w*
2) *baseAddr(array)-n2*w*

Consider the expression :: X[i , j] = Y [i + j , k] + Z[k , l]

- d_1, d_2 are the dimensions of X ;
- d_3 , d_4 are the dimensions of Y ;
- d_5 , d_6 are the dimensions of Z.

Three-address code statements for the above expression will be as follows

For array Y

- t1 = i + j
- t2 = t1 * d4
- t3 = t2 + k
- t4 = t3 * w
- t5 =addr(y)- C where c= d4*4
- t6= t5[t4]

For array Z

- t7 = k*d6
- t8 = t7 + l
- t9 = t8 * w
- t10 =addr(z)- C where c=d6*4
- t11 = t10[t9]

For array X

- t12 = i * d2
- t13 = t12 + j
- t14 = t13 * w
- t15 =addr(x)- C where c=d2*4
- t16 = t15[t14]

    - t17 = t6 + t11
    - t16 = t17

t1,t2,….t17 are address locations and are generated using *newtemp* in syntax directed translation.

Example:

Consider the statement:

X [i , j] : = Y [i + j, k] + z.

The maximum dimensions of *X* are *[d1, d2]* and of *Y* are *[d3, d4].*

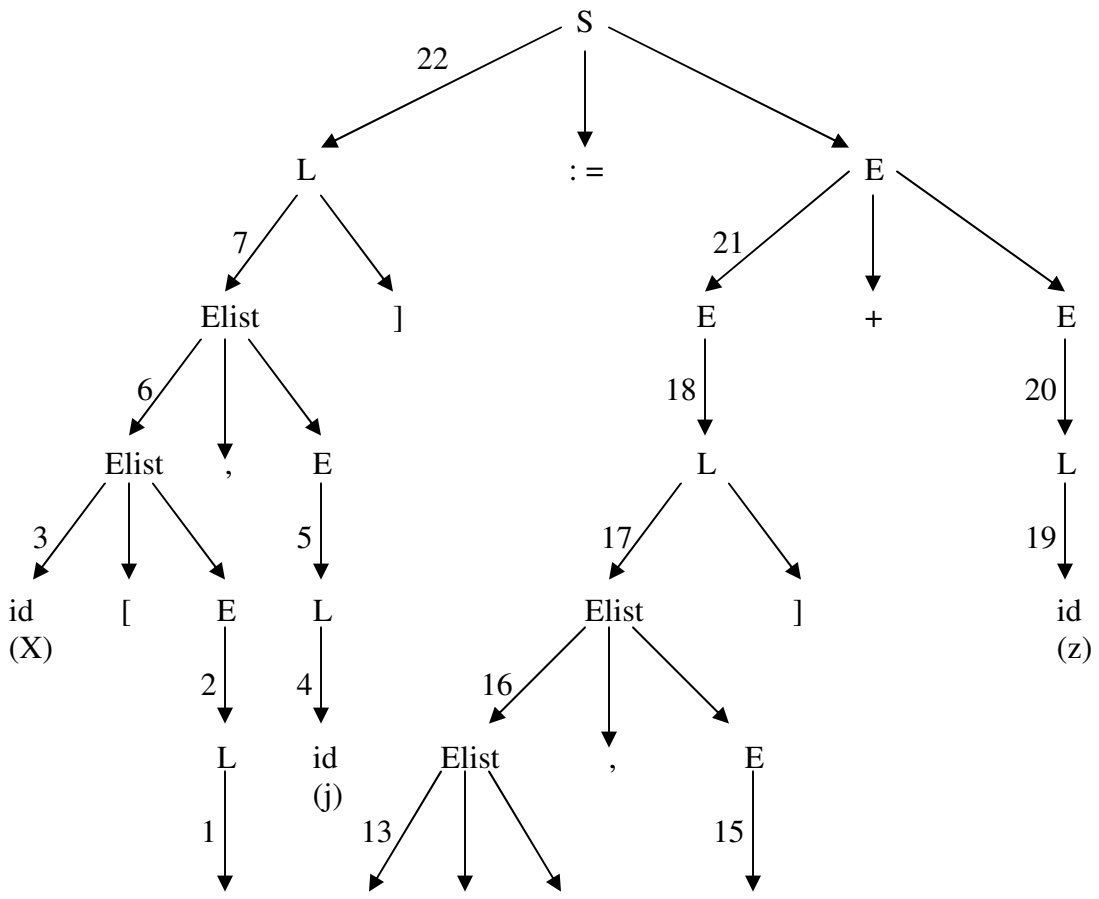
The intermediate three-address code for this statement can be written by intuition as follows:

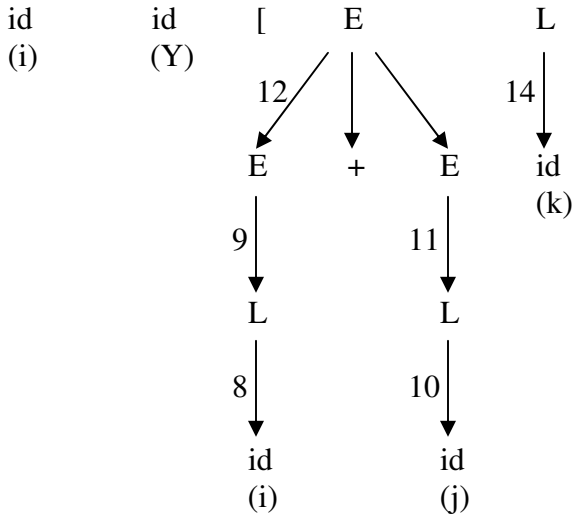
1. *t1 : = i * d1*
2. *t2 : = t1 + j*
3. *t3 : =addrX- c where c=d2*width(X)*
4. *t4 : = t2 * width(X)*
5. *t5 : = i + j*
6. *t6 : = t5 * d4*
7. *t7 : = t6 + k*
8. *t8 : = addrY-c where c=d2*width(Y)*
9. *t9 : = t7 * width(Y)*
10. *t10 : = t8[t9]*
11. *t11 : = t10 + z*

12. t3[t4] : = t11

Let's generate this intermediate code using the above method.

The **parse tree** for the statement is as follows:





NOTE: The number associated with each production is the **production rule number**.

The corresponding **semantic actions** and **three-address codes** generated are as follows:

	Semantic Action	**Three-Address Code**
1.	*L.place : = i*	
2.	*E.place : = i*	
3.	*Elist.array : = X* *Elist.place : = i* *Elist.dim : = 1*	
4.	*L.place : = j*	
5.	*E.place : = j*	
6.	*m : = 2* *Elist.array : = X* *Elist.place : = t1* *Elist.dim : = 2*	*t1 : = i * d2* *t1 : = t1 + j*
7.	*L.place : = t2* *L.place : = t3*	*t2 : = addr X-c* *t3 : = t1 * width(X)*
8.	*L.place : = i*	
9.	*E.place : = i*	
10.	*L.place : = j*	
11.	*E.place : = j*	
12.	*E.place : = t4*	*t4 : = i + j*
13.	*Elist.array : = Y* *Elist.place : = t4* *Elist.dim : = 1*	
14.	*L.place : = k*	
15.	*E.place : = k*	

16. m : = 2	*t5 : = t4 * d4*
Elist.array : = Y	*t5 : = t5 + k*
Elist.place : = t5	
Elist.dim : = 2	
17. L.place : = t6	*t6 : =addrY - c*
L.offset : = t7	*t7 : = t5 * width(Y)*
18. E.place : = t8	*t8 : = t6[t7]*
19. L.place : = z	
20. E.place : = z	
21. E.place : = t9	*t9 : = t8 + z*
22.	*t2[t3] : = t9*

NOTE: The three-address code generated by the above method is same as that which was generated by intuition. This proves the validity of the above method.

Boolean Functions:

Boolean expressions are composed of the Boolean operators (*and* , *or*, and *not*) applied to the elements that are Boolean variables or relational expressions.

Example:
a *or* b *and not* c

t_1 = *not* c
t_2 = b *and* t_1
t_3 = a *or* t_2

Example:
a<b

if *a<b* goto nextstat+3
t_1 := 0;
goto nextstat+2
t_1 :=1

E ⟶ E *or* E | E *and* E | *not* E | id_1 *relop* id_2 | true | false | (E)

Translation scheme for producting Three Address Codes for Boolean Expressions:

E ⟶ E_1 *or* E_2
{
E.place := newtemp();

emit (E.place ‘:=’ E_1.place ‘*or*’ E_2.place);
}

E ⟶ E_1 *and* E_2
{
E.place := newtemp();
emit (E.place ‘:=’ E_1.place ‘*and*’ E_2.place);
}

E ⟶ *not* E
{
E.place := newtemp();
emit (E.place ‘:=’ ‘*not*’ E.place);
}

E ⟶ (E_1)
{
E.place := E_1.place;
}

E ⟶ 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 ⟶ *true*
{
E.place := newtemp;
emit (E.place ‘:=’ ‘1’);
}

E ⟶ *false*
{
E.place = newtemp;
emit (E.place ‘:=’ ‘0’);
}

Short-Circuit Code

We can 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 is called *Short-Ciruit* or *Jumping* code.

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

100: if a<b goto 103
101: t_1 := 0
102: goto 104
103: t_1 := 1
104: ifc<d goto 107
105: t_2 := 0
106: goto 108
107: t_2 := 1
108: if e<f goto 111
109: t_3 := 0
110: goto 112
111: t_3 := 1

Type Conversions Within Assignments

In practice, there would be many different types of variables and constants, so the compiler must either reject certain mixed-type operations or generate appropriate coercion (type conversion) instructions. 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 introduce 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 → E+E     {E.type :=
                  if E1.type = integer and
                        E2.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. We must also include with the operator code an indication of whether fixed or floating-point arithmetic is intended. The complete semantic action for a production of the form $E \rightarrow E, + E.$ is listed in Fig. 8. 19.

```
E.place := newtemp;
if E1.type = integer and E2.type = integer then begin
      emit(E.place':=' E1.place 'int+' E2.place);
      E.type: = integer
```

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

TYPE CHECKING

14th Nov '05

Position of a type checker

(Parser) → syntax tree →(type checker)

(type checker)→syntax tree→(intermediate code generator)

(intermediate code generator) → intermediate representation

Type checking can be of two types:-

1. Static: Done during compilation time. This reduces the run time of the program and the code generation is also faster.

2. Dynamic: Done during run time. Due to this the code gets inefficient and it also slows down the execution time. But it adds to the flexibility of the program.

Types:

1. Basic Types: int, real, bool, char.
2. Arrays: as Array (length , type).
3. Function Arguments: as T1 x T2 x T3 x …… x Tn.
4. Pointer: as Pointer (T) .
5. Named Record :

If there is a structure defined as :-

```
struct  record
 {
      int length;
       char word[10];
  };
```

Then its type will be constructed as….

(length x integer) x (word x array(10,char))

6.Function: it maps element of one set to another i.e. from domain to range as

(D)→(R)

or

T1 → T2

Consider a simple C language:

1. P → D ; E
2. D → D ;D | id : T
3. T → char | integer | array[num] of T
4. E → literal | num | id | E mod E | E[E]

Corresponding Actions:

1. T → char : {T.type=char}
2. T → integer: {T.type = integer}

3. D→id : T : { AddEntry(id.entry,T.type)}
4. T→array[num] of T1 : {T.type = Array(num,T1.type)}
5. E→literal : { E.type = char }
6. E→num : { E.type = iniger }
7. E→id :{E.type = lookup(id.type)}
8. E→E1mod E2: {
   if(E1.type==int) && (E2.type == int)
   then E.type = int
   else type error
   }
9. E → E1 [E2] :{
   if(E2.type == int & E1.type == array(s,t))
   then E.type = t
   else type error
   }

CODE GENERATION

The final phase in our 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 requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.

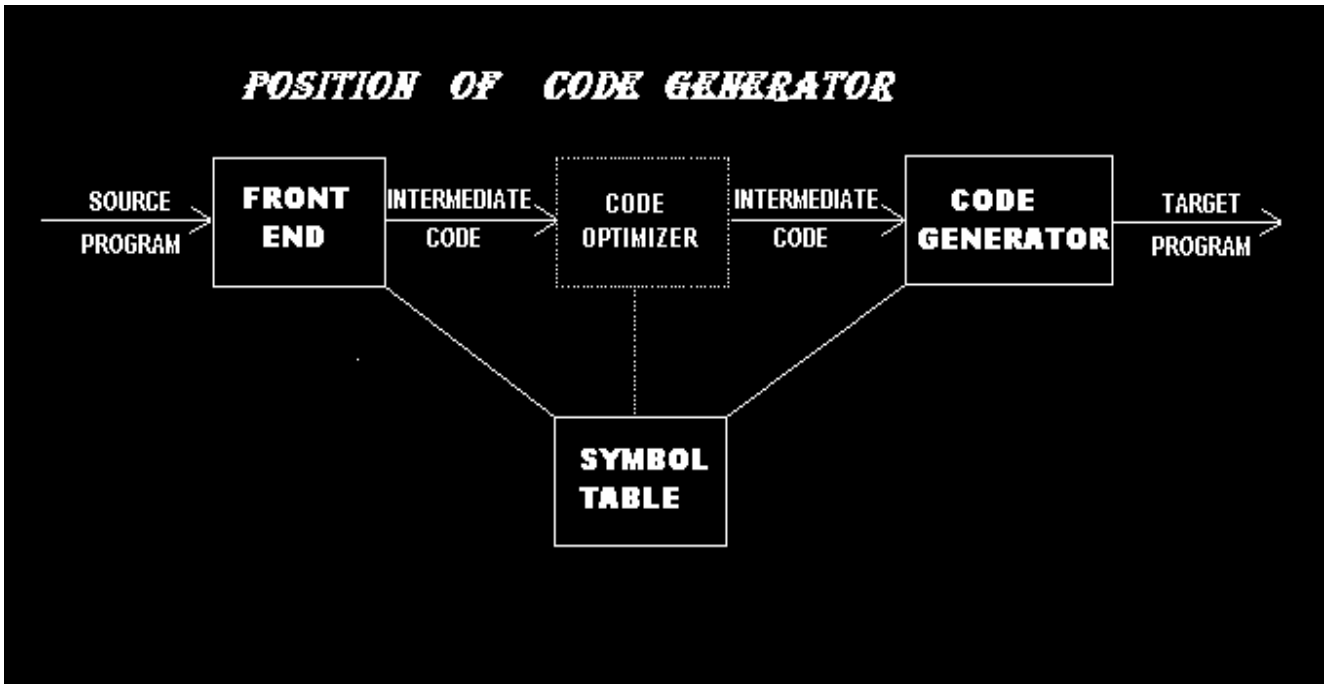


fig. 1

ISSUES IN THE DESIGN OF A CODE GENERATOR

1) INPUT TO THE CODE GENERATOR

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

We assume that prior to code generation the front end has scanned, parsed, and translated the source program into a reasonably detailed intermediate representation,The code generation phase can proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with code generation.

2) TARGET PROGRAMS

The output of the code generator is the target program. The output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.

3) MEMORY MANAGEMENT

Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name.

4) Instruction selection

For each type of three address statement, we can design a code skeleton that outlines the target code to be generated for that construct.

Eg.

x=y+z ➔

```
mov R1,z
add R1,y
mov x,R1
```

5) Register allocation

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of registers is particularly important in generating good code. The use of registers is often subdivided into two subproblems :

1. During register allocation , we select the set of variables that will reside in registers at a point in the program.
2. During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

6) Evaluation order

BASIC BLOCKS

IDENTIFYING 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 halt or possibility of branching except at the end.

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are the following :

1. The first statement is a leader.
2. Any statement that is the target of a conditional or unconditional goto is a leader.
3. 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.

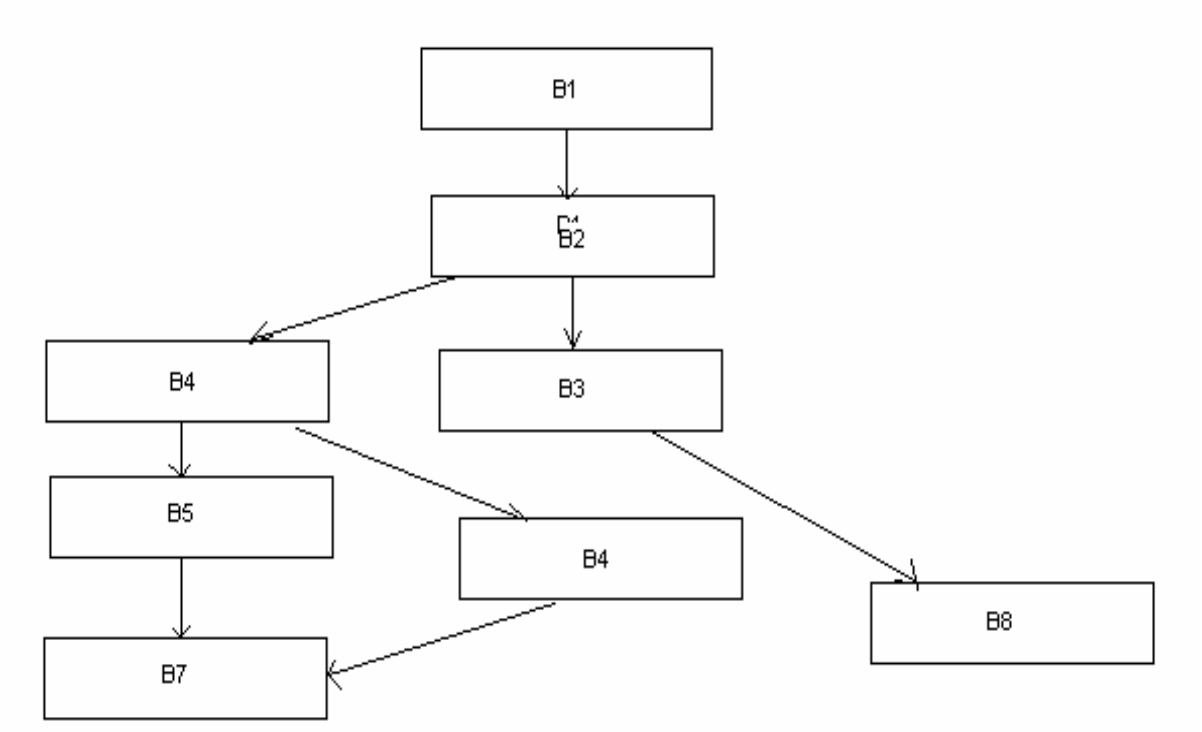
Egs.

1. location = -1
2. i=0
3. if (i<100) goto 5
4. goto 13
5. t1 = 4*1
6. t2 = A[t1]
7. if t2 = x goto 9
8. goto 10
9. location = i
10. t3 = i+1
11. i = t3
12. goto 3
13. ..

Leaders :

1,3,4,5,8,9,10,13

B1 = 1,2
B2 = 3
B3 = 4
B4 = 5,6,7
B5 = 8
B6 = 9
B7 = 10 , 11 , 12
B8 = 13



REGISTER ALLOCATION

The *getreg* function :

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

The algorithm for getreg:
1) If the name y is in a register , that holds the value of no other names (in other words no other names point to the same register as y does), and y is not live and has no next use after the execution of y = x *op* z, then
   a. return L.
   b. Update the address descriptor of y , so that y is no longer in L.
2) Failing (1), return an empty register for L if there is one.
3) Failing (2), if x has a next use in the block, or if *op* requires a register then
   a. find an occupied register R.
   b. MOV(R,M) if value of R is not in proper M. If R holds value of many variables, generate a MOV for each of the variables.
4) Failing (3), select the memory location of x as L.

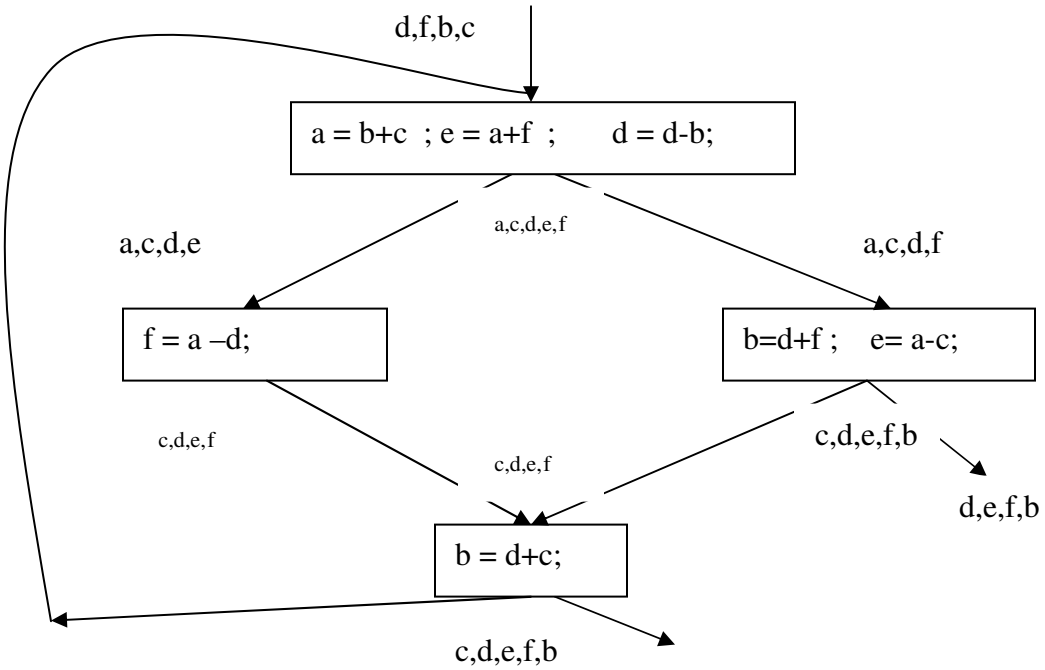
Example:
d := (a-b) + (a-c) + (a-c)

Statements	Code generated	Register descriptor	Address descriptor
t = a-b	MOV a, R0 SUB b,R0	R0 contains t	t in R0
u= a-c	MOV a, R1 SUB c,R1	R0 contains t R1 contains u	t in R0 u in R1
v= t+u	ADD R1,R0	R0 contains v R1 contains u	u in R1 v in R0
d= v+u	ADD R1,R0 MOV R0,d	R0 contains d	d in R0 d in R0 and memory

Note that cost of this code can be reduced from 12 to 11 by generating MOV R0,R1 immediately after the instruction MOV a,R1.

The guiding force behind a good program is the strategy of efficient allocation of registers.
A simple design is to keep fixed number of registers for each purpose like:
1) Base address
2) Stack
3) Arithmetic Operations
The simple design compensates for the inefficient utilization by this strategy.

Example of usage of registers across blocks:



Peephole Optimization

- A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target program.

- Here optimization is a misleading term because the most optimum code is not generated through the process called **peephole optimization**.

- Peephole optimization can be called a small but effective technique for locally improving the target code. It is called so because at any time optimization is done by looking at a small sequence of target instructions called the peephole and replacing the code by faster or shorter code whenever possible.

- The general practice being that each optimization results in spawning additional opportunities for code improvisation. And thus repeated passes over the code can be done to get the maximum benefit of this type of optimization.

❖ Some of the characteristic transformations in the code is listed below.

- Redundant-instruction optimization
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Removal of unreachable code

➢ Redundant-instruction optimization:-

Just look at the following code:-

MOV R0, a

MOV a, R0

This code can be improved by removing the second instruction which is redundant. Many such instructions may exist which are redundant and thus their removal can result in better code which is both shorter and faster.

➢ Flow of control optimization:-

Now take, for example, the following code

(n)Goto L1

.

.

(n+k)L1 : goto L2

This code can be replaced by the following code. Notice how one instruction is skipped when the flow of control reaches the line number (n).

(n)Goto L2

.

.

(n+k) goto L2

➢ Algebraic Simplifications :-

x = x + 0 ;

x = x * 1 ;

Pow(x, 2) ; /*can be replaced by "(x*x)"*/

Statements such as these can always be omitted or substituted as required for they end up doing no productive work and eat up the valuable CPU working time.

- Use of Machine Idioms :-

The target language may have hardware instructions to implement certain specific operations efficiently. And detecting situations such as these can result in a far more efficient code than the other ways optimizations. A good example can be that several hardwares support the single instruction INC x. Which serves to increment the value of x which otherwise would have taken at least 3 micro-instructions.