CODE GENERATION

INTERMEDIATE CODE

THREE ADDRESS CODE

A = b OP c

A→ destination

B and C → operands/ source

OP → operation

Add a, b → a = a+ b

Another intermediate code: P-code →very lengthy

Three-address codes:

Instructions:

Go to

Label

If-false

Jl → jump if less

Example:

A = b*c + (-d)/e;

Convert to three-address code

T1 = b * c

T = minus d

T2 = T / e

T3 = T1 + T2

A = T3

Define attributed grammar with code attribute

Code attribute → computed only for N-T symbols

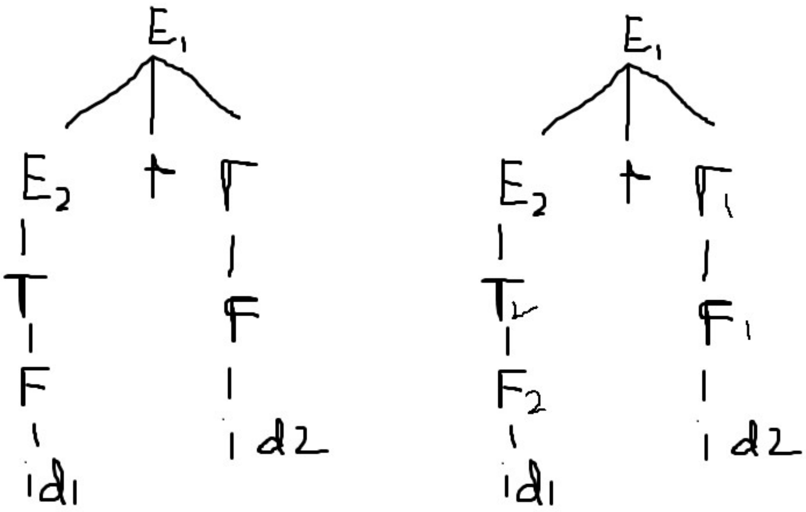
Code attribute → Synthesized attribute (child → Parent)

PRODUCTIONS	Semantic Rules (code attribute)
$E_1 \rightarrow E_2$ **+** T	E1.code = E2.code, T.code, E1.val = E2.val + T.val
E → T	E.code = T.code, E.val = T.val
$T_1 \rightarrow T_2$ ***** F	T1.code = T2.code, F.code, T1.val = T2.val * F.val
T → F	T.code = F.code, T.val = F.val
F → id	F.code = F.val = id. val
F → **(** E **)**	F.code = E.code, F.val = E.val

Example: id1 + id2

Parse Tree

Post-order traversal:



Id1 , F2 , T2, E2, +, id2, F1, T1, E1

Code will be generated for NT symbols.

F2.code → F2.val = id1.val { F2 → id1}

T2.code → F2.code   { T2 → F2}

T2.val = F2.val

T2.code → F2.val = id1.val

T2.val = F2.val

E2.code → T2.code

E2.val = T2.val

E2.code → F2.val = id1.val

T2.val = F2.val

E2.val = T2.val

F1.code → F1.val = id2.val

T1.code → F1.code

T1.val = F1.val

T1.code → F1.val = id2.val

T1.val = F1.val

E1.code → E2.code

T1.code

E1.val = E2.val + T1.val

E1.code → F2.val = id1.val

T2.val = F2.val

E2.val = T2.val

F1.val = id2.val

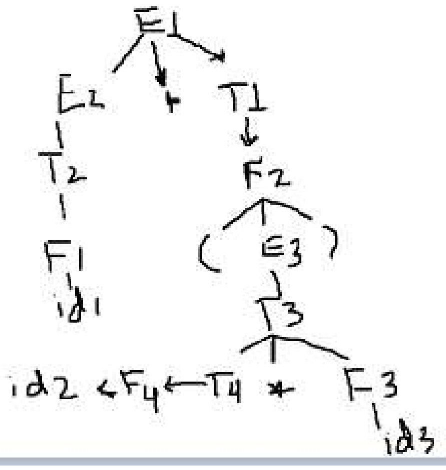
T1.val = F1.val

E1.val = E2.val + T1.val

So many temporary variables are used

Code is optimized in next phase

Expression : id1 + (id2* id3)



Id1, F1, T2, E2, +, (, id2, F4, T4, * ,id3, F3, T3, E3,), F2, T1, E1

F1.code→ F1.val = id1.val { F→ id}

T2.code → F1.val = id1.val //F1.code

T2.val = F1.val

E2.code → F1.val = id1.val

T2.val = F1.val //T2.code { E→ T}

E2.val = T2.val

F4.code → F4.val = id2.val

T4.code → F4.val = id2.val //F4.code

T4.val = F4.val

F3.code → F3.val = id3.val

T3.code → F4.val = id2.val

T4.val = F4.val //T4.code

F3.val = id3.val // F3.code

T3.val = T4.val * F3.val

E3.code → F4.val = id2.val

T4.val = F4.val

F3.val = id3.val

T3.val = T4.val * F3.val //T3.code

E3.val = T3.val

F2.code → F4.val = id2.val

T4.val = F4.val

F3.val = id3.val

T3.val = T4.val * F3.val

E3.val = T3.val //E3.code

F2.val = E3.val

T1.code → F4.val = id2.val

T4.val = F4.val

F3.val = id3.val

T3.val = T4.val * F3.val

E3.val = T3.val

F2.val = E3.val // F2.code

T1.val = F2.val

E1.code → F1.val = id1.val

T2.val = F1.val

E2.val = T2.val //E2.code

F4.val = id2.val

T4.val = F4.val

F3.val = id3.val

T3.val = T4.val * F3.val

E3.val = T3.val

F2.val = E3.val

T1.val = F2.val //T1.code

E1.val = E2.val + T1.val

Semantic Rules for Control Statements

If (E)

{ S }

If-false E goto Label1

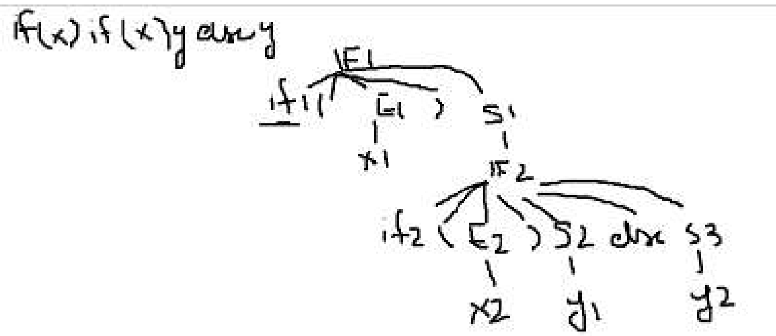
Statement S

Label1:

Productions	Semantic rules (Code)
IF → if (E) S	IF.code = E.code, If-false E.val goto L1 S.code Label L1:
IF → if (E) S1 else S2	IF.code = E.code, If-false E.val goto L1 S1.code, Goto L2, Label L1: S2.code, Label L2:
E → x	E.code = E.val = x.val
S → IF	S.code = IF.code
S → y (y is not a terminal)	S.code = y.code

Example:

if (x) if (x) y else y



if1, (, x1, E1,), if2, (, x2, E2,) y1, S2, else, y2, S3, IF2, S1, IF1

E1.code → E1.val = x1.val

E2.code → E2.val = x2.val

S2.code → y1.code

S3.code → y2.code

IF2.code → E2.code,

If-false E2.val goto L1

S2.code,

Goto L2,

Label 1: S3.code

Label L2:

IF2.code → E2.val = x2.val,

If-false E2.val goto L1

y1.code,

Goto L2,

Label 1: y2.code

Label L2:

S1.code → IF2.code

S1.code → E2.val = x2.val,

If-false E2.val goto L1

y1.code,

Goto L2,

Label 1: y2.code

Label L2:

IF1.code → E1.code,

If-false E1.val goto L3

S1.code

Label L3:

IF1.code → E1.val = x1.val

If-false E1.val goto L3

E2.val = x2.val,

If-false E2.val goto L1

y1.code,

Goto L2,

Label 1: y2.code

Label L2:

Label L3:

Repetition Structure:

Productions	Semantic Rules
FOR → for (E1 ; E2; E3) S	FOR.code = E1.code, Label L1: E2.code, If-false E2.val goto End S.code, E3.code, Goto L1 Label End:
WHILE → while (E) S	WHILE.code = Label L1:

	E.code, If-false E.val goto L2 S.code Goto L1 Label L2:

CODE OPTIMIZATION

2 TYPES:

1- Machine independent code optimization
    a. Front end (source code)
    b. Generic (no consideration of target machine)
2- Machine specific code optimization
    a. Backend
    b. Dependent on target machine
    c. Use target machine architecture, registers details etc

Machine independent code optimization techniques:

1. Dead Code elimination

```
a = 20;
if (a<10)
{   100 lines of statements;
} // never execute
```

Remove such code.

Memory efficient

2. Constant folding & Constant propagation

   a = 20; b = 10;

c = a*b; // constant propagation

Replaced by

c = 200;

x = 10+2*70; // constant folding

Replaced by

x =150;

Time and speed → both are saved

a = 3;

cin>>b;

c = b+ a *10;

c = b + 30;

c = a + b *10; // Not possible

3. Loop Unrolling
    - Creates overhead

```
for ( int i = 1; i<4 ; i++)
{
        Arr[i] = 0;
}
```

Arr[1] =0; Arr[2] = 0; Arr[3] =0;

Only works when no. of iterations is known

```
for (int a = 0; a< n; a++) // loop unrolling not possible
```

4. Loop Invariant:

```
for ( int i = 0; i<1000; i++)
{
…
a = b+c*d; //b,c,d values are never changing inside
the loop
….
}
```

```
Temp = b+c*d;
for ( int i = 0; i<1000; i++)
{
…
a =Temp;
….}
```

```
for ( int i = 0; i<1000; i++)
{
…
a = i+c*d; //c,d values are never changing inside the
loop→ partial invariant
….
}
```

```
Temp = c*d;
for ( int i = 0; i<1000; i++)
{
…
a = i+temp;
..}
```

5. Strength Reduction:

   Expensive operations→ *, /, %

   Cheaper operations→ +, -

   Replace expensive operations with cheaper operations wherever possible

   a *2 → a+ a

   a *10 → a+a+….+a_{10} --→ Not valid

6. Statement Rearrangement

```
A=b;
c=d;
e=a;
T1=b; a= t1;
T2=d;c=t2;
T3=a; e=t3;
```

```
A =b;
e = a;
c = d;
```

T1=b;

A = t1;

e = t1;

T2 = d;

c = t2;

7. Tail Recursion

This won't change your code

This is done by compiler internally

```
Void f1()
{
If( ) //base case;
f1(); // last statement of your function
}
Void main()
{ f1();
....}
```