UNIT-3

SYNTAX-DIRECTED TRANSLATION:

A Grammar symbols are associated with attributes to associate information with the programming language constructs that they represent. Values of these attributes are computed by the semantic rules associated with the grammar production rules. Evaluation of these semantic rules:

- ✓ may generate intermediate codes
- ✓ may put information into the symbol table
- ✓ may perform type checking
- ✓ may issue error messages
- ✓ may perform some other activities
- ✓ In fact, they may perform almost any activities.

An attribute may hold almost any thing. A string, a number, a memory location, a complex record.
Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.
There are two notations for associating semantic rules with grammar productions;

Syntax directed definition (SDD):

An SDD is also known as attribute grammar. An SDD are high level specifications for translatins. They hide many implementation details and free the user from having to specify explicitly the order in which translation takes place.
Recall that a syntax-directed definition (SDD) adds semantic rules to the productions of a grammar. For example to the production $T \rightarrow T / F$ we might add the rule

$T.code = T_1.code \parallel F.code \parallel '/'$ (if we were doing an infix to postfix translator)

Syntax directed translation schemes (SDT):

An SDT indicate the order in which semantic rules are to be evaluated.

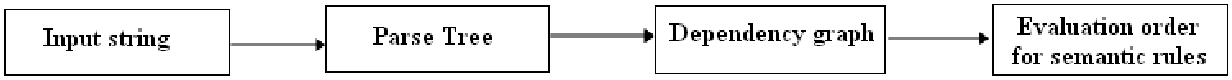

Figure: conceptual view of syntax directed translation.

Syntax directed definition:

Syntax directed definition is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes. An SDD partitioned the grammar attributes into two subset are as follows;

1. Synthesized attributes (An SDD with only synthesized attributes is called **S-attributed**.)
2. Inherited attributes

A Synthesized attribute for a non terminal A at a parse tree node N is defined by a semantic rule associated with the production at N. The value of a synthesized attribute at a node is computed from the value of attributes at the children of that node in the parse tree or by itself. Syntax directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition.

Production	Semantic Rules
L → E $	L.val = E.val
E → E_1 + T	E.val = E_1.val + T.val
E → E_1 - T	E.val = E_1.val - T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → T_1 / F	T.val = T_1.val / F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → num	F.val = num.lexval

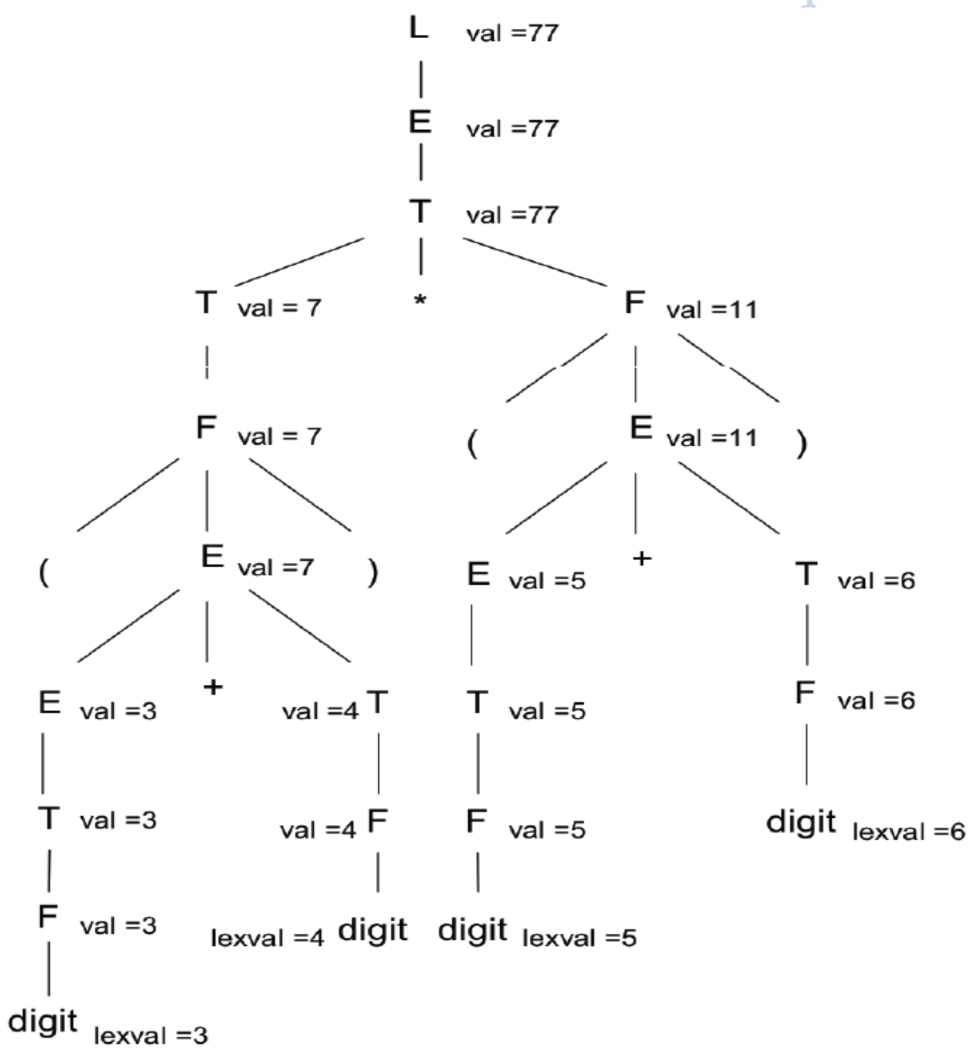

Figure: The annotated parse tree for the expression (3+4) * (5+6).

An inherited attribute for a non-terminal B at a parse tree node N is defined by a semantic rule associated with the production at the parent of N. the value of an inherited attributed is computed form the values of attributes at the sibling node, itself node and parent of that node.

Dependency Graphs:

Intermediate code generation:

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j. This approach to creating suite of compilers can save a considerable amount of effort: m*n compilers can be built by writing just m front ends and n back ends.

Although a source program can be translated directly into the target language, some benefits of using a machine independent intermediate form are;

1. Retargeting us facilitated: a compiler for a different machine can be created by attaching a back end for the new machine to an exiting front end.
2. A machine independent code optimizer can be applied to the intermediate representation.

An intermediate code can be point out as;

1. Intermediate code is closer to the target machine than the source language, and hence easier to generate code from.
2. Unlike machine language, intermediate code is (more or less) machine independent. This makes it easier to retarget the compiler.
3. It allows a variety of optimizations to be performed in a machine independent way.
4. Typically, intermediate code generation can be implemented via syntax-directed translation, and thus can be folded into parsing by augmenting the code for the parser.

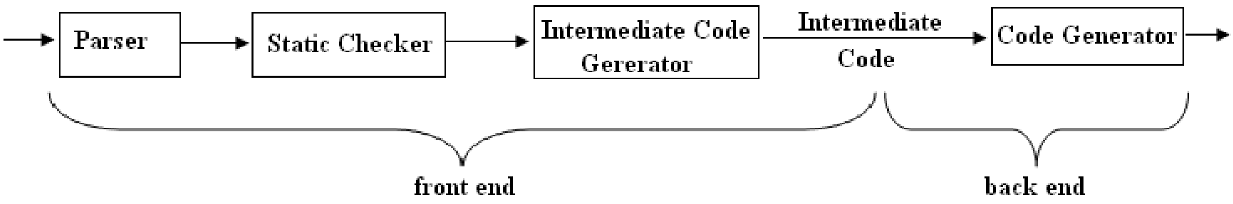

Figure: position of intermediate code generator

Translation to an intermediate representation:

Intermediate codes are machine independent codes, but they are close to machine instructions. Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language. In order to perform optimizations on a program and to generate code, it is convenient to produce an intermediate representation better suited to these than the parse tree representation. We will consider the following intermediate representations;

1) Abstract syntax trees can be used as an intermediate language.
2) postfix notation can be used as an intermediate language.

3) three-address code can be used as an intermediate language

we will use quadraples to discuss intermediate code generation quadraples are close to machine instructions, but they are not actual machine instructions.

Syntax Tree:

Syntax Tree is a variant of the Parse tree. Syntax trees are high level intermediat representation which depict the natural hierarchical structure of the source program. In a syntax tree each node (interior node) represent operators and the children of interrior node (leaf node) represents an operand.

Example:

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 = E1. Nptr
E → - E1	E. nptr = mkunode (‘uminus’ , E1. nptr)
E → id	E. nptr = mkleaf (id, id.place)

Tabel: Syntax directed defintion to produce syntax trees for assignment statements.

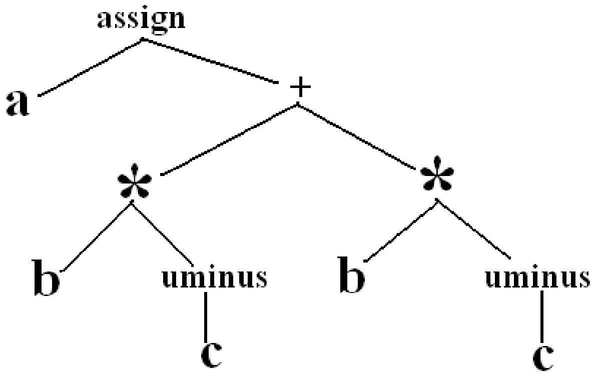

Figure: A syntax tree representation of a sentence a:= b * - c + b * - c

Postfix Notation:

Postfix Notation is another useful form of intermediate code if the language is mostly expressions. A postfix is a linear line of code which is more useful for code generation that for the optimization phases since it is difficult to do the sorts of transformations on it which are performed during the optimization phase. Sometimes a postfix notation is known as **Reverse Polish Notation (RPN)**, in which place each binary arithmetic operator after its two operands instead of between them.
There are following two reasons for using the postfix notation:

1. There is only one interpretation
2. We do not need parenthesis to disabiguate the grammar.

Production	Semantic Rule
E → E1 op E2	E.code := E1.code || E2.code || op

E → (E1)	E.code := E1.code
E → id	E.code := id, print. Id

Tabel: Syntax directed defination to produce postfix code for assignment statements.

Example 1: Traslate the following expression into postfix form:

a) a * (b + c)
b) (a + b) * c
c) (a + b) * (cc + d)

Solution:

a) The postfix code is: **abc+***
b) The postfix code is: **ab+c***
c) The postfix code is: **ab+cd+***

Example 2: Traslate the following expression into postfix form:

a) If e then x else y.
b) If a then if c-d then a+c else a*c else a+b.

Solution:

a) The postfix code is: exy?
b) The postfix code is: acd-ac+ac*?ab+?
(where ? using for ternary operator)

Three Address Code:

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like x+y*z might be translated into the sequence of three-address instructions.

t1 = y*z;
t2 = x + t1;

We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result). The most general kind of three-address code is:

x := y op z

where x, y and z are names, constants or compiler-generated temporaries; op is any operator. But we may also the following notation for quadraples (much better notation because it looks like a machine code instruction)

op y,z,x

apply operator op to y and z, and store the result in x.

Representation of three-address codes:

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-address code can be represented in various forms

1. Quadruples,
2. Triples
3. Indirect Triples

Quadruples:

A quadruple (or just "quad') has four fields, which we call **op, arg$_1$, arg$_2$,** and **result**. The op field contains an internal code for the operator. For instance, the three-address instruction x = y + x is represented by placing **+ in op, y in arg_1, z in arg_2, and x in result**. The following are some exceptions to this rule:

1. Instructions with unary operators like x = minusy or x = y do not use arg,. Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.
2. Operators like param use neither arg_2 nor result.
3. Conditional and unconditional jumps put the target label in result.

Example: Three-address code for the assignment **a = b *- c + b*- c;**

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

	Operator	arg_1	arg_2	Result
0	Minus	C		t_1
1	*	B	t_1	t_2
2	Minus	C		t_3
3	*	B	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

Triples:

Quadruples use a name, sometimes called a temporary name or 'temp', to represent the single operation. Triples are a form of three address code which do not use an extra temporary variable; when a reference to another triple's value is needed, a pointer to that triple is used. So a triple has only three fields, which we call op, arg_1, and arg_2.

Example: Three address code for the assignment of **a = b *- c + b*- c;**

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

	Operator	**arg_1**	**arg_2**
0	Minus	c	
1	*	b	(0)
2	Minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect triples:

An Indirect triples is a representaiton of three address code in which an additional array is used to list the pointers to the triples in the desired order. Therefore an Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

(0)	(35)
(1)	(36)
(2)	(37)
(3)	(38)
(4)	(39)
(5)	(40)

	Operator	**arg_1**	**arg_2**
0	minus	c	
1	*	b	(35)
2	minus	c	
3	*	b	(37)
4	+	(36)	(38)
5	=	a	(39)

Table : Indirect triples representation of three-address code

Production	Semantic Action
S → id := E	S.code := E.code || gen(id.place’: =’ E.place)

E → E1 + E2	E.place = newtemp();
	E.code = E1.code || E2.code || gen(E.place :‘=’ E1.place ‘+’ E2.place)

E → E1 * E2	E.place = newtemp();
	E.code = E1.code || E2.code || gen(E.place ‘:=’ E1.place ‘*’ E2.place)

E → - E1	E.place = newtemp();
	E.code = E1.code || gen(E.place ‘:=’ ‘uminus’ E1.place)

E → (E1)	E.place = E1.place;
	E.code = E1.code

E → id	E.place = id.place;
	E.code = null

Tabel: Syntax directed defination to produce Three address code for assignment statements.

Note: In the syntax directed translation o three address code, the nonterminal E has two attributes;

1. E.place, the name that will hold the value of E,
2. E,code, the sequence of three address statements evaluating E.

Example: A = -B * (C + D)

Three-Address code is as follows:

T_1 = - B
T_2 = C + D
T_3 = T_1 * T_2
A = T_3

Quadruple:

	Operator	Operator1	Operator2	Result
1.	-	B		T_1
2.	+	C	D	T_2
3.	*	T_1	T_2	T_3
4.	=	A	T_3	

Triple:

	Operator	Operator1	Operator2
1.	-	B	
2.	+	C	D
3.	*	1	2
4.	=	A	3

Indirect triple:

0. (56)
1. (57)
2. (58)

3. (59)

	Operator	Operator1	Operator2
(56)	-	B	
(57)	+	C	D
(58)	*	(56)	(57)
(59)	=	A	(58)

Translation of assignment statements:

Production	Tranlation scheme

$S \rightarrow id := E$	{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)}

$E \rightarrow - E$	E.place = newtemp();
	emit(E.place:=' 'uminus' E_1.place)}

$E \rightarrow (E_1)$	E.place = E_1.place;

$E \rightarrow id$	{p := lookup(id.name);
	if p≠nil then
	E.place := p)
	else error}

Table: Translation scheme to produce three address code for assignments.

Translation of boolean expressions:

Boolean expressions are composed of the boolean operators (which we denote &&, ||, and !, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions. A boolean expressions have two primary purposes.

1. A boolean expression is used to compute the logical values.
2. Boolean expression is used as conditional expressions in statements that alter the flow of control.

we consider boolean expressions generated by the following grammar:

E→E or E| E and E| not E | (E) | id rel.op id | true | false

We use the attribute rel.op to indicate which of the six comparison operators <, <=, =, ! =, >, or >= is represented by rel.

Note: We assume that || and && are left-associative, and that || has lowest precedence, then &&, then !.

There are two methods that are used to translating the boolean expressions;

1. **Numerical representation**
2. **Control flow representation**

Numerical representation:

In this boolean expression method encode true and false numerically. In this method TRUE is denoted by 1 and FALSE by 0. Expressions are evaluated from left to right, in a manner similar to arithmetic expressions.

Example 1: The translation for A or B and C is the three-address sequence:

T_1 := B and C

T_2 := A or T_1

Also, the translation of a relational expression such as A < B is the three-address sequence:

(1) if A < B goto (4)

(2) T := 0

(3) goto (5)

(4) T := 1

(5)

Therefore, a Boolean expression A < B or C can be translated as:

(1) if A < B goto (4)

(2) T_1 := 0

(3) goto (5)

(4) T_1 := 1

(5) T_2 := T_1 or C

Example 2: 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

therefore, a boolean expression **if a < b then 1 else 0;**

(0) if a < b goto (3)

(1) t := 0

(2) goto (4)

(3) t := 1

(4)

Example 3: The expression **a < b or c < d and e < f.**

Solution:

(0) if a < b goto (3)

(1) t1:=0

(2) goto (4)

(3) t1 := 1

(4) if c < d goto (7)

(5) t2 := 0

(6) goto (8)

(7) t2 := (1)

(8) if e < f goto (11)

(9) t3 := 0

(10) goto 12

(11) t3 := 1

(12) t4 := t2 and t3

(13) t5 := t1 or t4

Production	**Semantic Action**
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_1	E.place ‘:=’ newtemp; emit (E.place ‘:=’ ‘not’ E_1.place)
E → (E_1)	E.place := E_1.place;

E → id_1 relop id_2	{E.place ‘:=’ newtemp;
	emit(‘if’ id_1.place rel.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’)}

Table: Translation scheme using a numerical representation for booleans.

Control-Flow/ Positional Representation of Boolean Expressions:

1. If we evaluate Boolean expressions by program position, we may be able to avoid evaluating the entire expressions.
2. In A or B, if we determine A to be true, we need not evaluate B and can declare the entire expression to be true.
3. In A and B, if we determine A to be false, we need not evaluate B and can declare the entire expression to be false.
4. A better code can thus be generated using the above properties.

Production	**Semantic Action**
E → E_1 or E_2	E_1.true := E.true; E_1.false := newlable; E_2.true := E.true; E_2.false :=E.false; E.code := E_1..code || gen(E_1.false ‘:’) || E_2.code
E → E_1 and E_2	E_1.true := newlable; E_1.false := E.false; E_2.true := E.true; E_2.false :=E.false; E.code := E_1..code || gen(E_1.true ‘:’) || E_2.code
E → not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E_1.code
E → (E_1)	E_1.true := E.true;

	E_1.false := E.false; E.code := E_1.code
E → id_1 relop id_2	E.code := gen('if' id_1.place rel.op id_2.place 'goto' E.true ‖ gen('goto' E.true));
E → true	E.code ;= gen('goto' E.true));
E →false	E.code ;= gen('goto' E.false));

Table: Syntax directed defination to produce three address code for booleans.

Example: The statement if **(A<B ‖ C<D) x = y + z;** can be translated as

Solution:

```
(1) if A<B goto (4)
(2) if C<D goto (4)
(3) goto (6)
(4) T=y+z
(5) X=T
(6)
```

Here (4) is a true exit and (6) is a false exit of the Boolean expressions.

Short circuit code:

A short circuit code or jumping code is used to translate a boolean expression into three address code without generating code for any of the boolean operator and without having the code necessarily evaluate the entire expression.

Example: The statement

if(x<100 ‖ x> 200 && x != y) x=0;

might be translated into the code of **.** In this translation, the boolean expression is true if control reaches label L2. If the expression is false, control goes immediately to L1, skipping L2 and the assignment x = 0.

```
if x < 100 goto L2
if false x > 200 goto L1
if false x != y goto L1
L2        x = 0
L1
```

Figure Jumping code

Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of 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 these productions, nonterminal E represents a boolean expression and nonterminal S represents a statement.

Production	Semantic rules

S → if E then S_1	E.true := newlable;
	E.false := S.next
	S_1.next := S. next
	S.code := E.code ‖ gen(E.true':') ‖ S_1.code

S → if E then S_1 else S_2	E.true := newlable;
	E.false := newlable;
	S_1.false := S.next;
	S_2.next := S. next;
	S.code := E.code ǁ gen(E.true':') ǁ S_1.code gen('goto' S_1.next) ǁ gen(E.false ':') ǁ S_2.code

S → while E do S_1	S.begin := newlable;
	E.true := newlable;
	E.false := S.next;
	S_1.next := S.begin;
	S.code := gen(S.begin ':' ǁ E.true':' gen(E.true ':') ǁ S_1.code ǁ gen('goto' S.begin)

Table: Syntax directed defination for flow of control statements.

Backpatching:

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump. For example consider the boolean expression;
(aǁb) &&(c<d)&&(x>5)
If (aǁb) is false, then there is no need to compute the rest of the expression however, when we generate the code for (aǁb), we have no idea where to jump to if (aǁb) is false. The same problem exists for some flow-of-control statements.
As able to generate the code for this kind of structure in a single pass, we can generate code bottom up during parsing, this process is accomplished with backpatching.

Production	Semantic Action

E → 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;)

E → E_1 and E_2	{ backpatch(E_1.truelist, M.quad);
	E.truelist := E_2.truelist;
	E.falselist := merge(E_1.falselist, E_2.falselist;)

E → not E_1	E_1.truelist := E.falselist; E_1.falselist := E.truelist
E → (E_1)	E_1.truelist := E.truelist; E_1.falselist := E.faleslist

E → id_1 relop id_2	E.truelist := makelist(nextquad);
	E.falselist := makelist(nextquad +1);
	gen('if' id_1 relop id_2 'goto _ ')
	gen('goto _')

E → true	E.truelist :=makelist(nextquad);
	gen('goto_')

E →false	E.falselist :=makelist(nextquad);
	gen('goto_')

M → ϵ	M.quad := nextquad

Table:semantic rule for boolean expression using backpatching