UNIT: IV & V

Intermediate code Generation

Intermediate Languages – Intermediate Representation – Implementation of Three address statements – Declarations – Declarations in procedure – Declaration in Nested procedures – Assignment Statements – Boolean Expressions – Numerical Representation – Control flow translation of boolean expressions – Flow control statements – Backpatching.

Intermediate Languages:

The task of compiler is to convert the source program into machine program. This activity can be done directly but it is not always possible to generate such a machine code directly in one pass.

Then typically compilers generate an easy to represent form of source language which is called intermediate language. The generation of an intermediate language leads to efficient code generation.

From the intermediate language, the target machine code can be generated effectively

Scanned by CamScanner

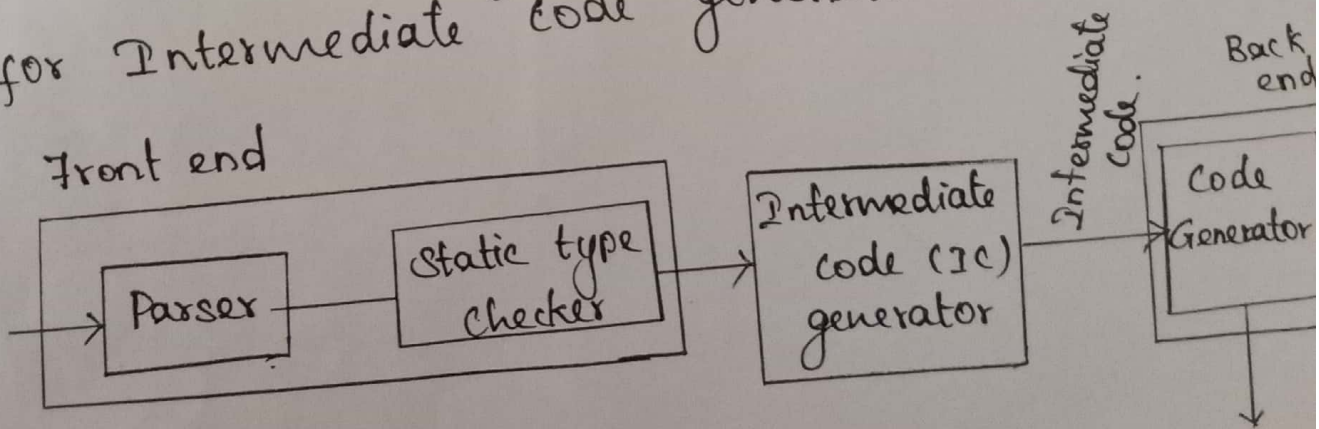
Benefits of Intermediate Code Generation:

There are certain benefits of generating machine independent intermediate code

* A compiler for different machines can be created by attaching different backend to the existing front ends of each Machine.
* A compiler for different source languages can be constructed With different front ends for corresponding source languages to existing end.
* A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

Intermediate code generator in compiler:-

The following is the block diagram for Intermediate code generator in compiler.



Scanned by CamScanner

Properties of Intermediate Language:.

* The intermediate language is an easy form of source language which can be generated efficiently by the compiler.

* The generator of Intermediate language should lead to efficient code generation.

* It should act as effective mediator.

* It should be flexible.

Intermediate Representation:.

There are mainly three type of IC representation.

1. Abstract Syntax tree.
2. Polish Notation.
3. Three address code.

Abstract Syntax tree:.

The natural hierarchical structure is represented by syntax trees. Directed Acyclic Graph or DAG is very much similar to syntax trees but they are in more compact form.

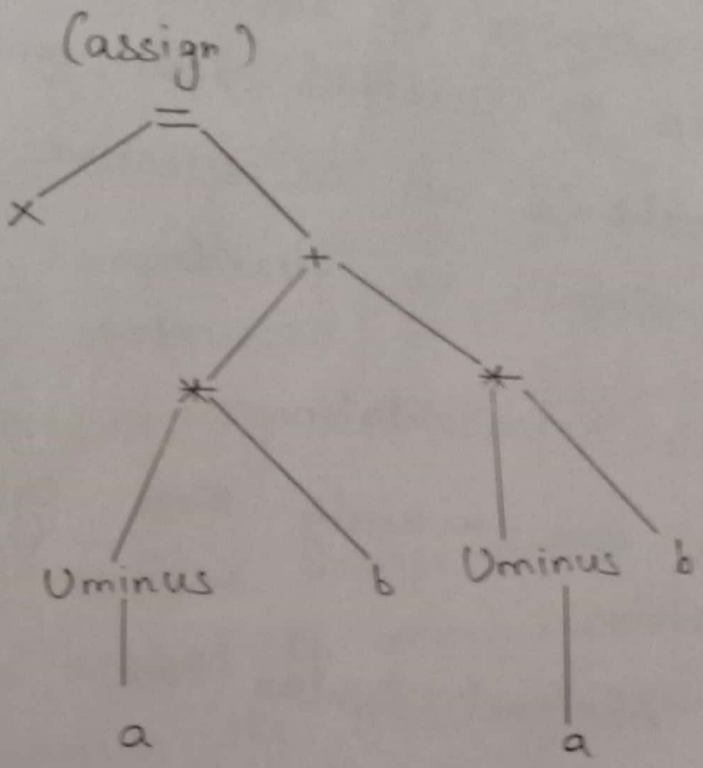
The code being generated as intermediate should be such that the remaining processing

Scanned by CamScanner

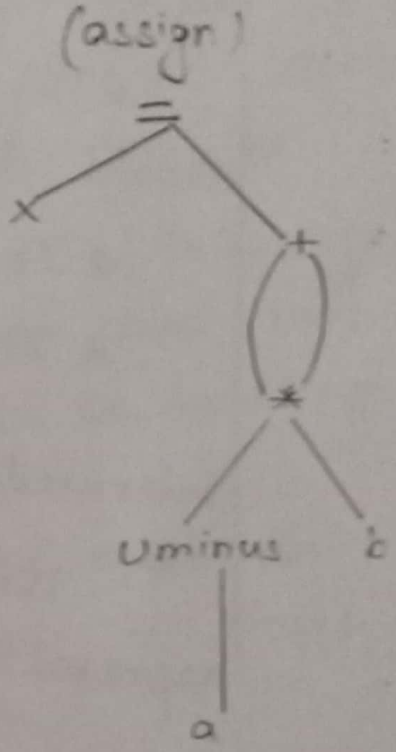
of the subsequent phases should be

eg: x:=-a*b+ -a*b

Syntax tree



DAG.



Polish Notation:-

Basically, the linearization of syntax trees is polish notation. In this representation, the operator can be easily associated with the corresponding operands. This is the most natural way of representation in expression evaluation.

The polish notation is also called as prefix notation in which the operator occurs first and then operands are arranged.

Scanned by CamScanner

eg: a+b ⟹ +ab.

Implementation of Three address code

Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields. There are three representation used for three address code such as quadruples, triples and indirect triples.

Quadruple representation:

The Quadruple is a structure with at the most four fields such as op, arg1, arg2, result.

The op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands used and result field is used to store the result of an expression.

eg: $x = -a * b + -a * b$

The three address code is

$t_1 := \text{uminus } a$

$t_2 := t_1 * b$

$t_3 := -a$

Scanned by CamScanner

$t_4 := t_3 * b$

$t_5 := t_2 + t_4$

$x := t_5$

Quadruple				
	op	Arg1	Arg 2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

Triples:.

In the triple representation the use of temporary variables is avoided by referring the pointers in the symbol table.

eg: $x := -a * b + -a * b$

Number	op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	x	(4)

Scanned by CamScanner

Indirect Triples:

In the indirect triple representation the listing of triples is been done and listing pointers are used instead of using statement

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(11)	b
(2)	uminus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	:=	x	(15)

	Stmt
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

Types of three address statement:

* Assignment statement
  - → x := y op z
  - → x := op y.
* Copy statement
  - → x := y
* Unconditional Jump
  - → goto L
* Conditional Jump
  - → if x relop y goto L
* Procedure calls

```
param x1
param x2
  :
param xn
call p,n
return y
```

* Array statement
  - x := y[i]
* Address & pointer
  - x := &y
  - x := *y
  - *x := y

Scanned by CamScanner

Declarative Statement :.

In the declaration declarative statement data items along with their data types are declared.

* Declaration in Procedure :.

eg: Production	Semantic action
$S \to D$	{ offset := 0 }
$D \to id : T$	{ enter_tab (id.name, T.type, offset); offset := offset + T.width) }
$T \to integer$	{ T.type := integer; T.width := 4 }
$T \to real$	{ T.type := real; T.width := 8 }
$T \to array [num]$ of T_1	{ T.type := array (num.val, T.type) T.width := num.val × T_1.width }
$T \to *T_1$	{ T.type := pointer (T.type) T.width := 4 }

The initialization of offset in the translation scheme is more evident if the first production appears as

$P \to M D$

$M \to \epsilon$ { offset := 0 }

Non terminal M, generating ϵ is called NP

Scanned by CamScanner

*Declarations in Nested Procedures:

In nested procedures, name local to each procedure can be assigned relative addresses using the previous approach. When a nested procedure is seen processing of declarations in the enclosing procedure is temporarily suspended

eg: This approach will be illustrated by adding semantic rules to the following language.

$P \rightarrow D$

$D \rightarrow D ; D \mid id : T \mid proc\ id ; D ; S$

A new symbol-table is created when a procedure declaration $D \rightarrow proc\ id\ D_1 ; S_2$ is seen and entries for the declarations in D are created in the new table. The new table points back to the symbol table of the enclosing procedure. The name represented by id itself is local to the enclosing procedure

eg:

$T \rightarrow record\ LD\ end$: {T.type := record(top(tblptr)); T.width := top(offset); pop(tblptr); pop(offset)}

$T \rightarrow \epsilon$: {t := mktable(nil); push(t, tblptr); push(0, offset)}

Scanned by CamScanner

Assignment statements:

The assignment statement mainly deals with the expressions. The expressions can be of type integer, real, array and record.

eg: $x := (a+b) * (c+d)$.

~~The three address code is~~

output:

$t_1 := a+b$

$t_2 := c+d$

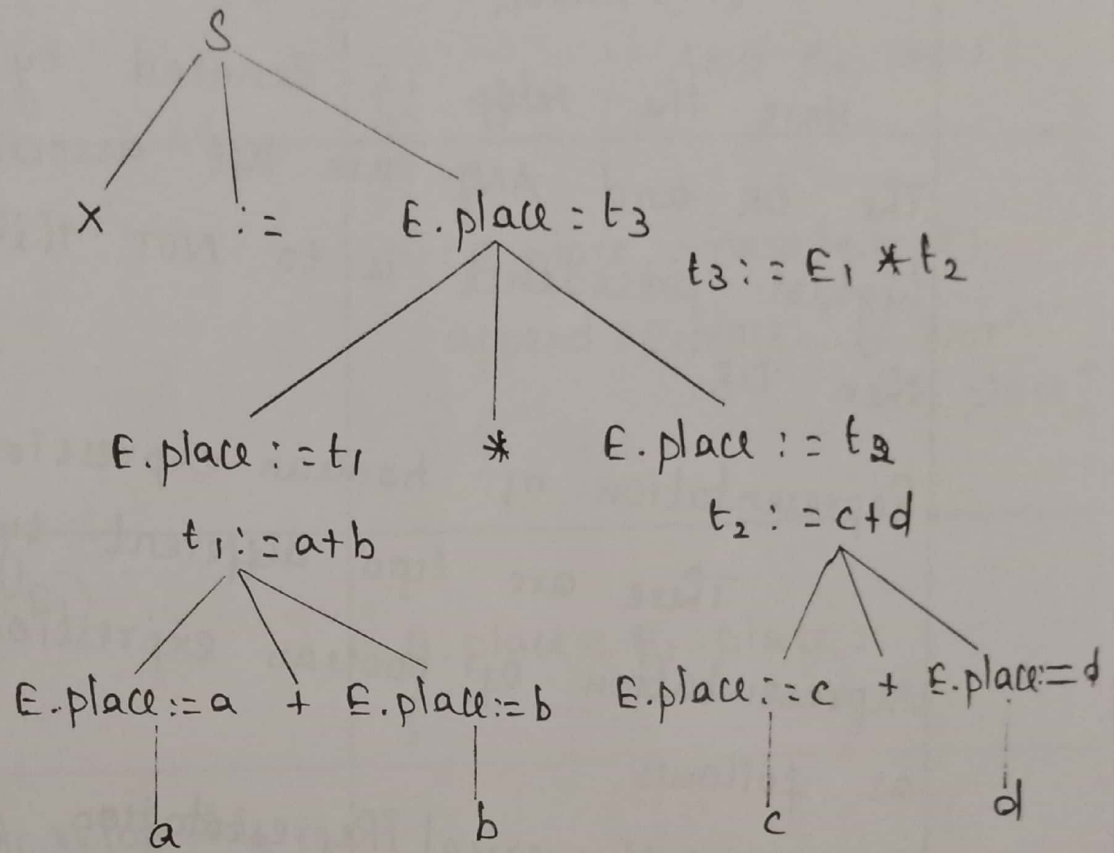
$t_3 := (a+b) * (c+d)$

$x := t_3$

Production rule	Semantic action	o/p
$E \rightarrow id$	E.place := a	
$E \rightarrow id$	E.place := b	
$E \rightarrow E_1 + E_2$	E.place := t_1	$t_1 := a+b$
$E \rightarrow id$	E.place := c	
$E \rightarrow id$	E.place := d	
$E \rightarrow E_1 + E_2$	E.place := t_2	$t_2 := c+d$
$E \rightarrow E_1 * E$	E.place := t_3	$t_3 := (a+b) (c+d)$
$S \rightarrow id := E$		$x := t_3$

Scanned by CamScanner

The annotated parse tree can be drawn as follows



Boolean Expressions:

Normally there are two types of boolean expressions used. They are as follows

1. For computing the logical values
2. In conditional expressions using if-then-else or while-do.

The following are the boolean expressions

E → E OR E

E → E AND E

E → NOT E

E → (E)

Scanned by CamScanner

E → id relop id

E → TRUE

E → FALSE

Here the relop is denoted by ≤, ≥, ≠, <, >, =. The OR and AND are left associate. The highest precedence is to NOT then AND and then OR.

Representation of boolean expression:-

There are two different types of representation of boolean expression. They are as follows

1. Numerical Representation
2. Control flow translation of boolean expression.

Numerical Representation:-

The translation scheme for boolean expressions having numerical representation is as follows.

Production Rule	Semantic Action.
$E \rightarrow E_1$ OR E_2	{ E.place := newtemp() append(E.place ':=' E_1.place 'OR' E_2.place) }

Scanned by CamScanner

$E \rightarrow E_1$ AND E_2	{
	E.place := newtemp()
	append (E.place ':=' E_1.place
	'AND' E_2.place)
	}

$E \rightarrow$ NOT E_1	{
	E.place := newtemp()
	append (E.place ':=' 'NOT'
	E_1.place
	}

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

$E \rightarrow id_1$ relop id_2	{
	E.place := newtemp()
	append ('if' id_1.place relop.op
	id_2.place 'goto' next_state+3)
	append (E.place ':=' '0');
	append ('goto' next_state +2);
	append (E.place := '1');
	}

$E \rightarrow$ True	{
	E.place := newtemp();
	append (E.place ':=' '1')
	}

$E \rightarrow$ False	{
	E.place := newtemp()
	append (E.place ':=' '0')
	}

Scanned by CamScanner

The function append generates the address code and newtemp() is for gen of temporary variables.

For the Semantic action for the rule : $E \rightarrow id_1$ relop id_2 contains next_stat which gives the index of next three address statement in the output sequence.

eg: P > Q AND r < s OR U > V

```
100: if P>q goto 103
101: t1 := 0
102: goto 104
103: t1 := 1
104: if r<s goto 107
105: t2 := 0
106: goto 108
107: t2 := 1
108: if U>V goto 111
109: t3 := 0
110: goto 112
111: t3 := 1
112: t4 := t1 AND t2
113: t5 := t4 OR t3.
```

In this three address code we have used goto to jump on some specific statement. This method of evaluation is called "Short - Circuit".

Scanned by CamScanner

Control Flow translation of Boolean Expressions :.

* Syntax - directed definition for boolean expression

Productions	Semantic Actions
$E \rightarrow E_1$ or E_2	{ E_1.true := E.true; E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.false ':') || E_2.code }
$E \rightarrow E_1$ and E_2	{ E_1.true := newlabel; 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}
$E \rightarrow$ not E_1	{ E_1.true := E.false E_1.false := E.true E.code := E_1.code }
$E \rightarrow (E_1)$	{ E_1.true := E.true E_1.false := E.false E.code := E_1.code }

Scanned by CamScanner

$E \rightarrow id_1$ relop id_2	{ E.code := gen ('if' id_1. relop.op id_2.place 'goto' gen ('goto' E.false) }
E → true	{ E.code := gen('goto' E.true) }
E → False	{ E.code := gen ('goto' E.false) }

In this, the attribute E.code gives the sequence of three-address statements evaluating E.

In this method, E is translated into a sequence of three-address statements evaluates E is a sequence of conditional and unconditional jumps to one of the two locations. ie

→ E.true → control flows if E is true

→ E.false → control flows if E is false

The function newlabel returns a new symbolic label each time it is called.

Scanned by CamScanner

Flow control statements:-

The control statements are if-then-else and while-do. The grammar for such statemer is as shown below

$S \rightarrow$ if E then S_1

| if E then S_1 else S_2

| while E do S_1

While generating three address code:-

* To generate new symbolic label the function new-label() is used.
* With the expression E.true and E.false are the labels associated.

S.code and E.code is for generating three address code.

If-then.

$S \rightarrow$ if E then s $\Rightarrow$ E.true := new_label()

E.false := S.next

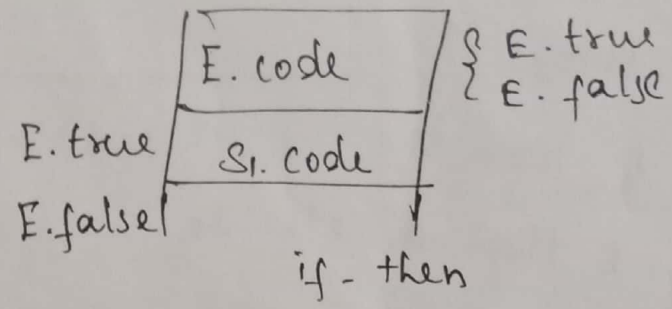
S_1.next := S.next

S.code := E.code || gen_code(E.true) || S_1.code

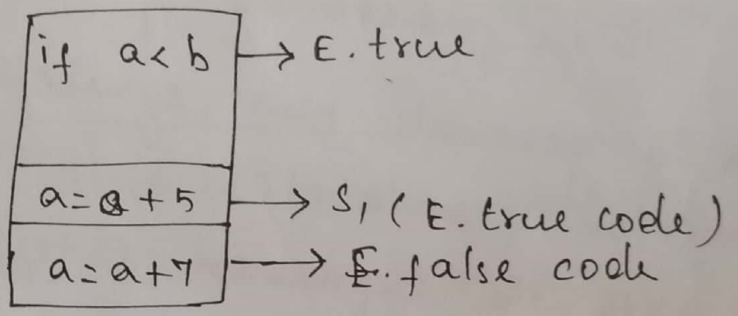
In the above translation scheme || is used to concant concatenate the strings. The function gen_code is used to evaluate the non quoted

arguments passed to it and to concaten complete string.

S.code ⇒ generate the three address



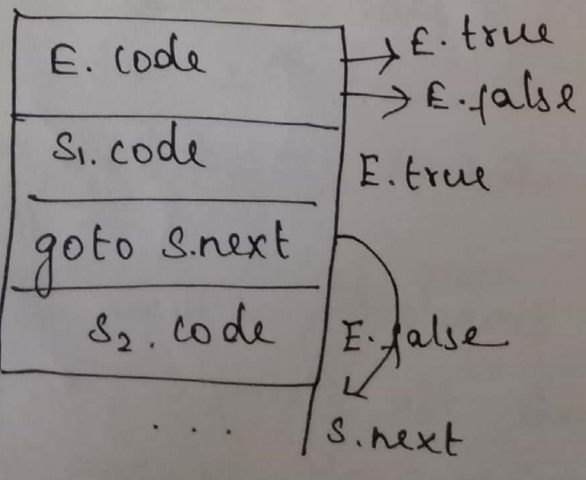
eg: if a<b then a=a+5 else a=a+7



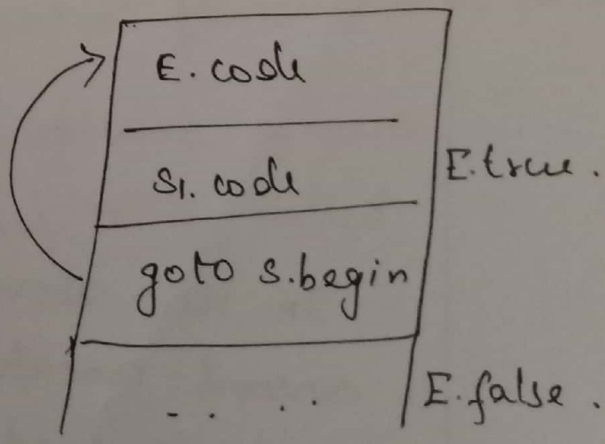
The three address code for if-then is

100: if a<b goto L1
101: goto 103
102: L1: a=a+5
103: a:=a+7

If-then



while-do



Scanned by CamScanner

$S \rightarrow$ if E then S_1 else S_2 $\Rightarrow$ E.true := new_label()
E.false := new_label()
S_1.next := S.next
S_2.next := S.next
S.code := E.code || gen_code (E.true ':') || S_1.code ||
gen_code ('goto', S.next) ||
gen_code (E.false ':') || S_2.code

$S \rightarrow$ while E do S_1 $\Rightarrow$ S.begin := new_label()
E.true := newlabel()
E.false := S.next
S.code := gencode (S.begin ':') ||
E.code || gen_code (E.true) || S_1.code
|| gen_code ('goto', S.begin)

Backpatching:

Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.

In general, the implementation of syntax directed definition using two passes is the most convenient method. If we decide to generate the three address code for given syntax directed definition using single pass

Scanned by CamScanner

only, then the main problem that occurs decision of addresses of the labels. The jump (goto) statements refer these label statements and in one pass it becomes difficult to know the locations of these label statements. If we use two passes instead of one pass then in one pass we can leave these addresses unspecified and in the second pass ~~then is one pass we can leave these addresses~~ this in-complete information can be filled up. To overcome the problem of processing the incomplete information in one pass the backpatching technique is used.

Functions used:.

To generate code using backpatching, In the semantic actions the following functions are used.

1. Mklist (i) :- Creates the new list. The index i is passed as an argument to this function.
2. Mergelist (p_1, p_2) :- This functions concatinates two lists pointed by P_1 and P_2
3. backpatch (p, i) :. Inserts i as target label for the statement pointed by p.

Scanned by CamScanner

Production Rule	Semantic Action.
$S \rightarrow$ if B then M S_1	{ backpatch (B.Tlist, M.state); S.next := merge (B.Flist, S_1.next) }
$S \rightarrow$ if B then M_1 S_1 N else M_2 S_2	{ Backpatch (B.Tlist, M_1.State); backpatch (B.Flist, M_2.state); S.next := merge (S_1.next, Merge (N.next, S_2.Next); }
$S \rightarrow$ while M_1 B then do M_2 S	{ backpatch (S_1.next, M_1.state); backpatch (B.Tlist, M_2.state); S.next = B.Flist; append ('goto' M_1.state); }
$S \rightarrow$ begin L end	{ S.next := L.next }
$S \rightarrow A$	{ S.next := Nulc }
$L \rightarrow L_1$; M S	{ Backpatch (L_1.next, M.state); L.next := S.next; }
$L \rightarrow S$	{ L.next := S.next; }
$M \rightarrow \epsilon$	{ M.state := Nextstate
$N \rightarrow \epsilon$	{ N.next := mklist (nextstate); append ('goto-');

Scanned by CamScanner

* The synthesized attributes Tlist and Flist are used to generate the jumping code for boolean expressions. For the true statement E.Tlist will be generated and for the false statement E.Flist is generated.

* The attribute 'state' will be associated with the M and that is used to record the number (address) of the statement. It is denoted as M.state. The nextstate will point to the next statement.

Backpatching using flow of control statement :-

The flow of control statements can be handled using backpatching techniques for the following grammar

$S \rightarrow$ if B then M S_1

$S \rightarrow$ if B then M_1 S_1 N else M_2 S_2

$S \rightarrow$ while M_1 B then do M_2 S

$S \rightarrow$ begin L end

$S \rightarrow A$

$L \rightarrow L_1$; M S

$L \rightarrow S$

$M \rightarrow \epsilon$

$N \rightarrow \epsilon$

B → boolean Exp

L → stmt list

A → Assignment stmt

Scanned by CamScanner

Implementation of backpatching:

* Backpatching using Boolean Expressions
* Backpatching using flow of control statem

Backpatching using boolean expressions:

Consider the grammar for boolean expressio

$E \rightarrow E_1$ OR $M\ E_2$

$E \rightarrow E_1$ AND $M\ E_2$

$E \rightarrow$ NOT E_1

$E \rightarrow (E_1)$

$E \rightarrow id_1$ relop id_2

$E \rightarrow$ True

$E \rightarrow$ False

$M \rightarrow \&.$

Here a new non terminal M is inserted as a marker nonterminal. The purpose of M is to marks the exact point where the semantic action is picked up.

Production Rule	Semantic Action
$E \rightarrow E_1$ OR $M\ E_2$	{ backpatch (E_1.Tlist, M.state); E.Tlist := merge (E_1.Tlist, E_2.Tlist); E.Flist := E_2.Flist }

Scanned by CamScanner

$E \rightarrow E_1$ AND M E_2	{ backpatch (E_1.Tlist, M.st); E.Tlist := E_2.Tlist; E.Flist := merge (E_1.Flist, E_2. }
$E \rightarrow$ NOT E_1	{ E.Tlist := E_1.Flist; E.Flist := E_1.Tlist; }
$E \rightarrow (E_1)$	{ E.Tlist := E_1.Tlist; E.Flist := E_1.Flist; }
$E \rightarrow id_1$ relop id_2	{ E.Tlist := Mklist (nextstati). E.flist := mklist (nextstati +1); append ('if' id_1.place relop . op id_2.place 'goto-') append ('goto _') }
E → TRUE E → False	{ E.Tlist := mklist (nextstate). append ('goto' -'); } { E.flist := mklist (nextstate); append ('goto-'); }
M → ε	{ m.state := nextstati; }

Scanned by CamScanner