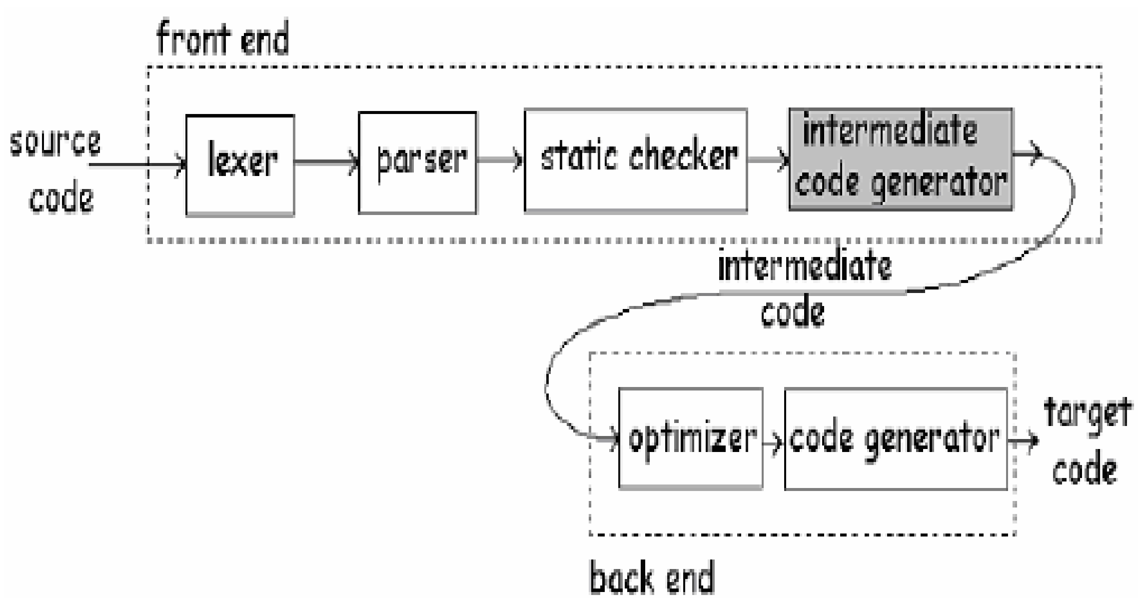
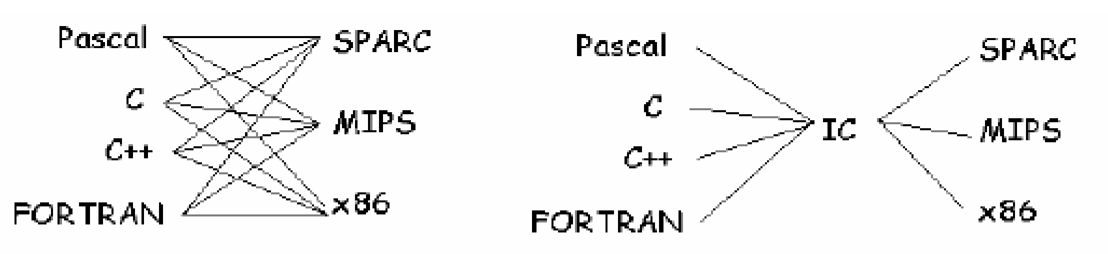
Intermediate code generation

- Most modern compilers are split into two:
  - the front end translates a source program to an intermediate representation
  - the back end then generates machine code for target architecture



Motivation

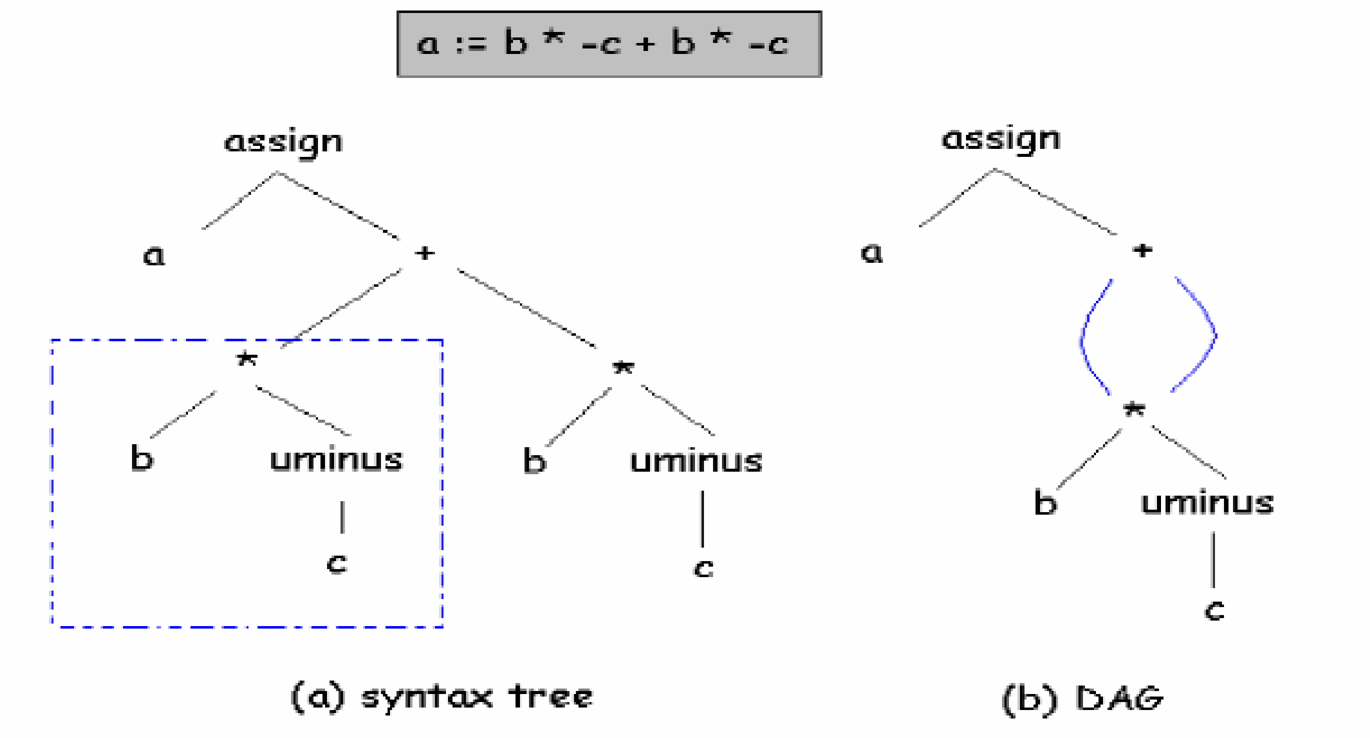
- Advantages: It is easier to write different back ends for different target machines. Optimization is machine independent.
- The disadvantage is that the compiling become a little slower (because of this intermediate step)



Various forms

- Various Intermediate representations include:
  - syntax tree, Directed Acyclic Graph (DAG)
  - Postfix
  - three address code
  - Control Flow Graphs (CFG), Program dependence Graph (PDG), Static Single Assignment Form (SSA)

Syntax-tree & Directed Acyclic Graph



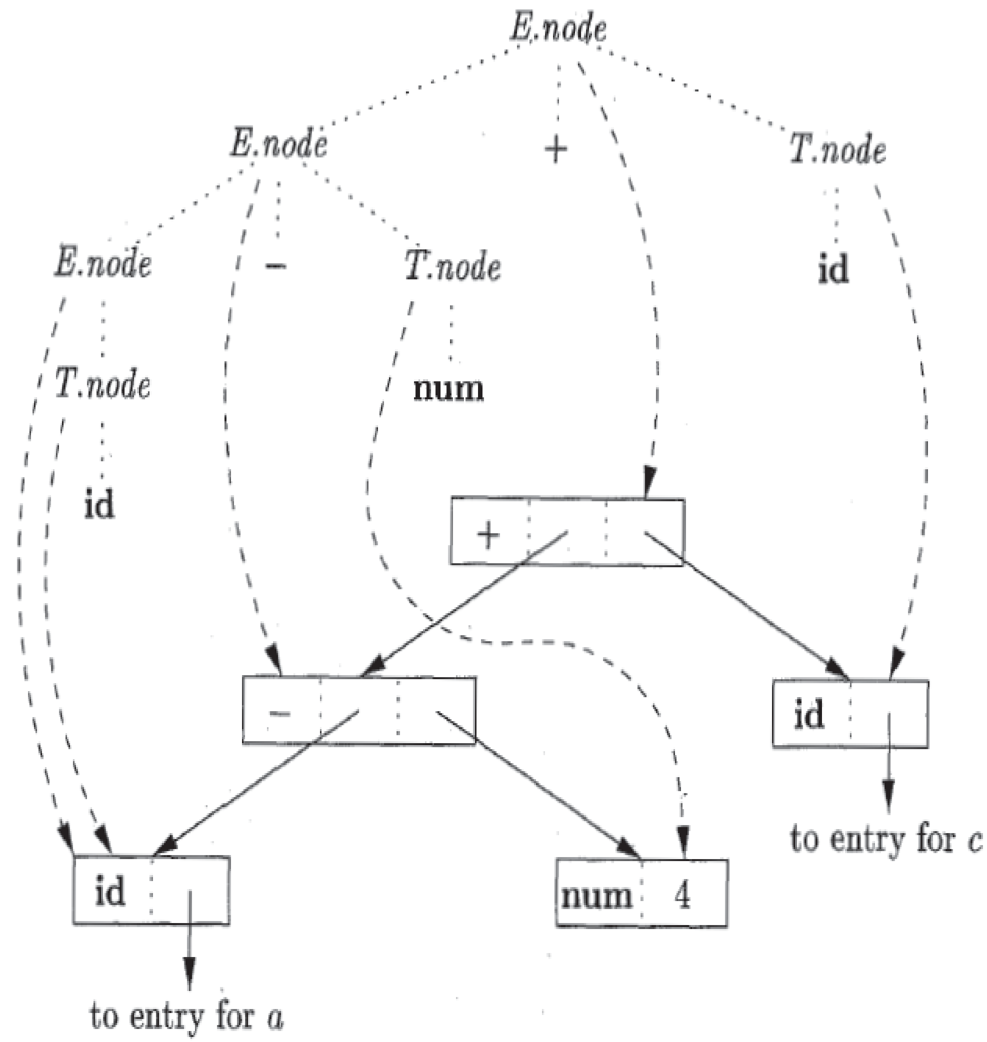
SDD to construct Syntax-tree

- The syntax tree is an abstract representation of the program constructs

	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Syntax tree from a-4+c

1) $p_1 = \textbf{new } Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new } Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new } Node('-', p_1, p_2);$
4) $p_4 = \textbf{new } Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new } Node('+', p_3, p_4);$



How to Generate DAG from Syntax-Directed Definition?

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

Functions such as ***Node and Leaf above*** check whether a node already exists. If such a node exists, a pointer is returned to that node.

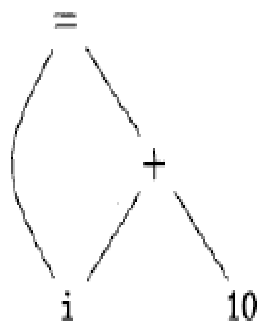
How to Generate DAG from Syntax-Directed Definition?

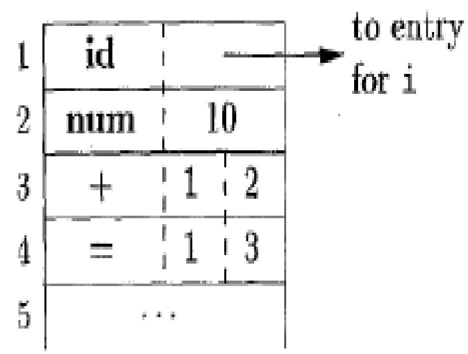
	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

```
a + a * (b - c) + (b - c) * d
```

DAG using data structure array

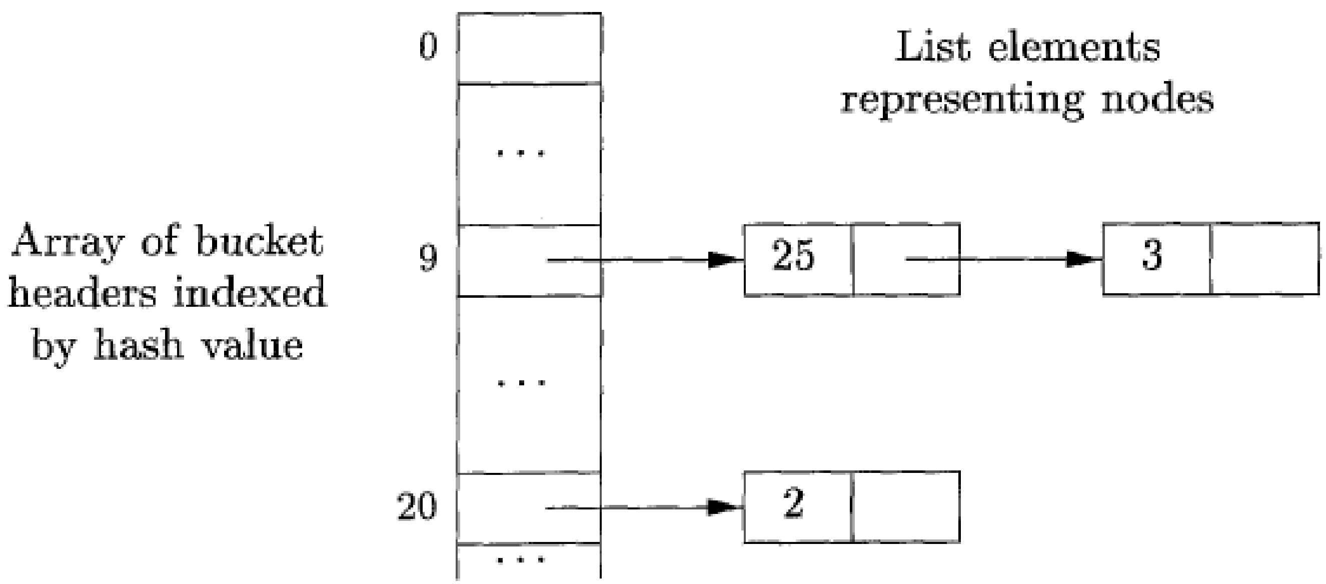




- Scanning the array each time a new node is needed, is not an efficient thing to do.

DAG using Hash Table

- Hash function = h(op, L, R)



Postfix form

- Postfix form for a=b*-c + b*-c is

a b c uminus * b c uminus * + assign

Grammar	Semantic Rules
$E \rightarrow E_1 + T$	$E.t := E_1.t \; \| \| \; T.t \; \| \| \; '+'$
$\quad \| \; T$	$E.t := T.t$
$T \rightarrow T_1 * F$	$T.t := T_1.t \; \| \| \; F.t \; \| \| \; '*'$
$\quad \| \; F$	$T.t := F.t$
$F \rightarrow (E)$	$F.t := E.t$
$\quad \| \; num$	$F.t := num.t$

Three addresses instructions

- Sequence of statements of the general form

$$x = y \text{ op } z$$

where x, y, z are names, constants or compiler generated temporaries, and op is operator (arithmetic, logical, shift, etc.) that takes *at most two operands*.

- **Example:**

```
i = 2*j + k - 1;
    ⇓
t1 = 2 * j
t2 = t1 + k
i = t2 - 1
```

Three addresses instructions

Assignment instructions of the form $x = y \ op \ z$

Assignments of the form $x = op \ y$

Copy instructions of the form $x = y$

An unconditional jump goto L

Conditional jumps of the form if x goto L and ifFalse x goto L

Conditional jumps such as if x *relop* y goto L

Procedure call such as p(x1, x2, …, xn) is implemented as:

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

Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$

Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$

Three addresses instructions

- Assume each array element takes 8 units of spaces.

```
do i = i+1; while (a[i] < v);
```

L: t_1 = i + 1		100:	t_1 = i + 1
i = t_1		101:	i = t_1
t_2 = i * 8	OR	102:	t_2 = i * 8
t_3 = a [t_2]		103:	t_3 = a [t_2]
if t_3 < v goto L		104:	if t_3 < v goto 100

Implementation of three-address statements

- How to present these instructions in a data structure?
  - Quadruples
  - Triples
  - Indirect triples

Implementation of three-address statements: Quadruples

- Quadruples: records with four fields

```
typedef struct {
  int op;
  SYM_TAB *arg1, *arg2, *result;
} QUAD;
```

- Unused field left blank/null

Implementation of three-address statements: Quadruples

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

	op	*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
		...		

- Disadvantage: temporary names have to be entered into each record.

Implementation of three-address statements: Triples

- Avoids entering temporary names into records.
- Use records with three fields: operator, arg1, arg2

Implementation of three-address statements: Triples

$t_1 := -c$
$t_2 := b * t_1$
$t_3 := -c$
$t_4 := b * t_3$
$t_5 := t_2 + t_4$
$a := t_5$

(a) code for syntax tree

	op	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

pointers to TAS struct

(b) Triples

- Ternary operation x[i] := y requires two entries in the triple structure and x := y[i] requires two operations.

	op	arg 1	arg 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] := y

	op	arg 1	arg 2
(0)	=[ ]	y	i
(1)	assign	x	(0)

(b) x := y[i]

More triple representations

Implementation of three-address statements: Triples

- In an optimizing compiler, instructions are often moved around.
- Moving of an instruction that computes temporary may require to change all the references to that result.
  - Benefit of quadruples over triples
  - Indirect triples solve this problem

Implementation of three-address statements: Indirect Triples

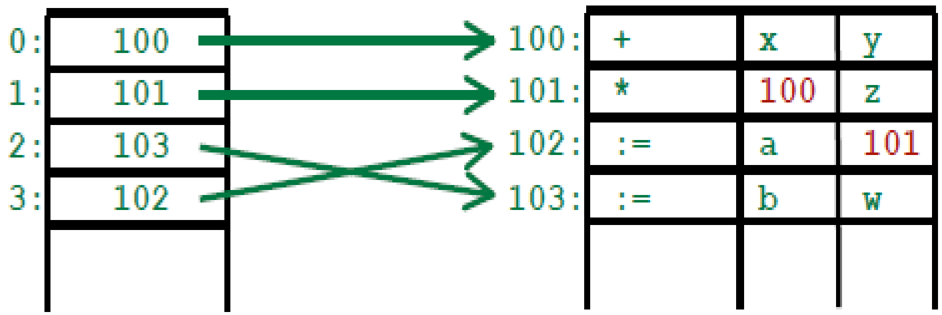
	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

List of pointers to triples

Optimizing complier can reorder instruction list, instead of affecting the triples themselves

Advantage of Indirect Triples



- Advantage of indirect triples over quadruples?

Static Single Assignment (SSA)

- Is an intermediate presentation
- Facilitates certain code optimizations
- All assignments are to variables with distinct names

$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$
(a) Three-address code.	(b) Static single-assignment form.

Static Single Assignment (SSA)

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1,x2);
```

Returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment statement containing the Ø-function

Assignment

Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

c) Triples.

d) Indirect triples.

SDD to generate three-address code for expressions

Address holding value of E
(e.g. tmp variable, name, constant)

Three-address code of E

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \Vert$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \Vert\ E_2.code \ \Vert$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \Vert$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Build an instruction

Get a temporary variable

Current symbol table

SDD to generate three-address code for expressions

Production	Semantic Rules
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \ \|\| \ gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \ \|\| \ E_2.code \ \|\| \ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \ \|\| \ gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

a = b + - c

```
t1 = minus c
t2 = b + t1
a = t2
```

SDD for Boolean expression to generate three-address code

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \ ! \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\texttt{'if'} \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ \texttt{'goto'} \ B.true)$ $\mid\mid gen(\texttt{'goto'} \ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen(\texttt{'goto'} \ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen(\texttt{'goto'} \ B.false)$

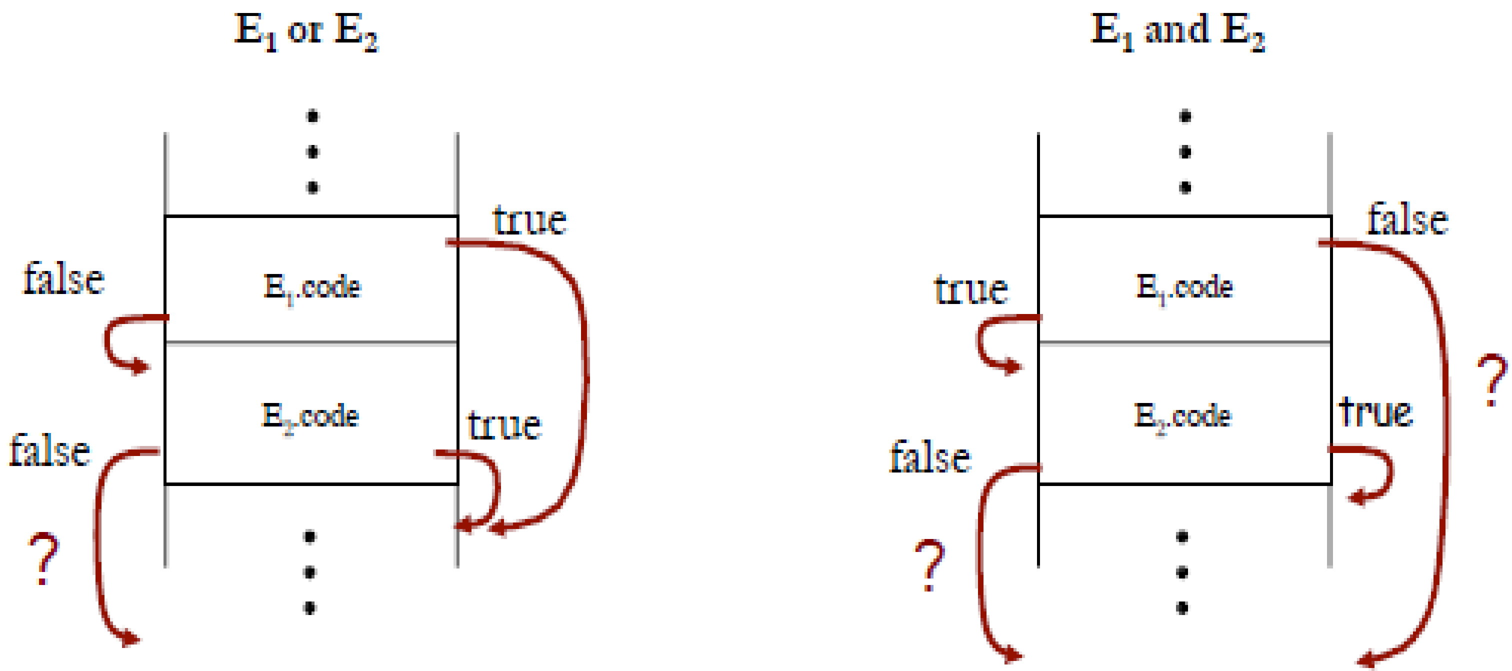
Three-address code for boolean

- E: a < b

  if a < b goto E.true

  goto E.false

Boolean expression: code outline



Three address code for boolean

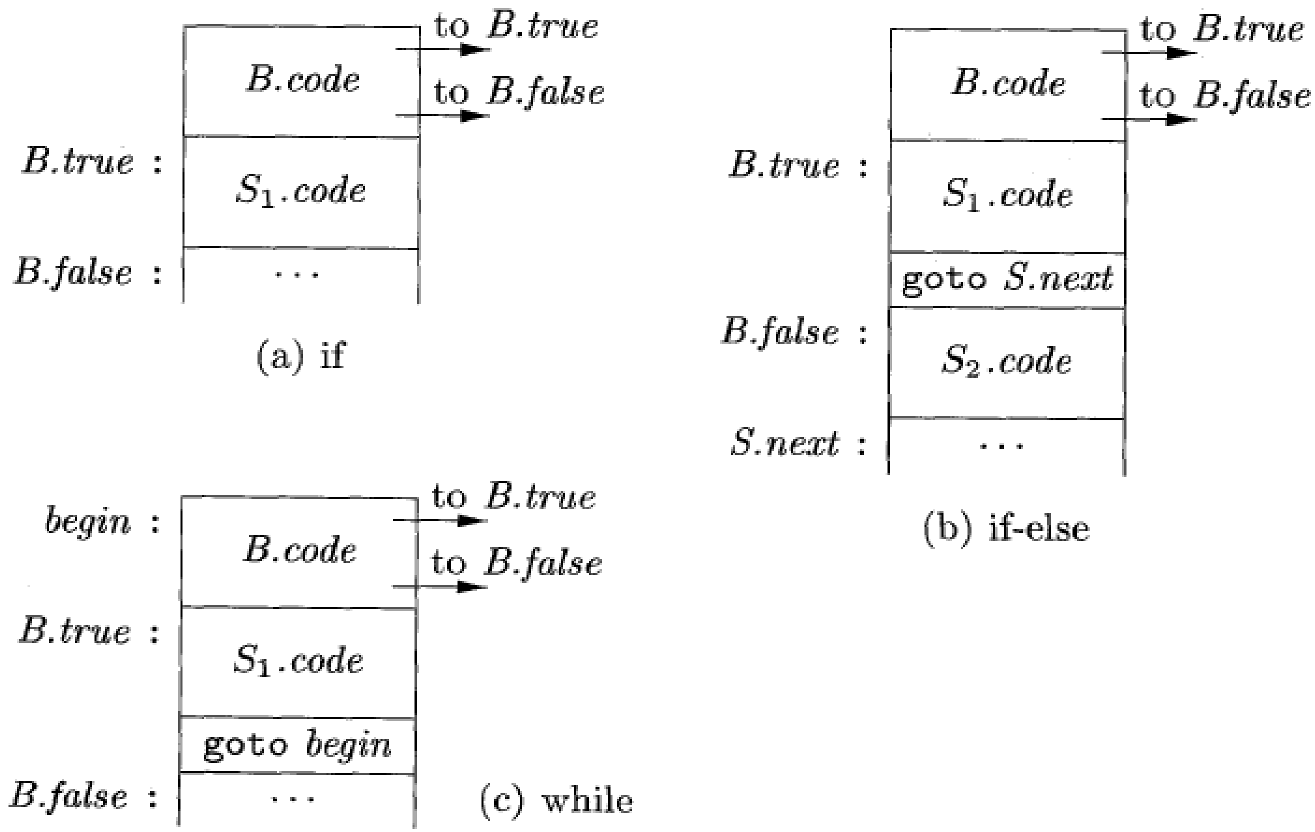
a < b || c < d && e < f

```
       if a < b goto Ltrue
       goto  L1
L1:    if c < d goto  L2
       goto  Lfalse
L2:    if e < f goto Ltrue
       goto  Lfalse
```

Ltrue resp. Lfalse are the targets if the entire expression evaluates to true resp. false

Translation of Flow-of-Control Statements

$$
\begin{array}{lll}
S & \rightarrow & \textbf{if} (B) S_1 \\
S & \rightarrow & \textbf{if} (B) S_1 \textbf{ else } S_2 \\
S & \rightarrow & \textbf{while} (B) S_1
\end{array}
$$

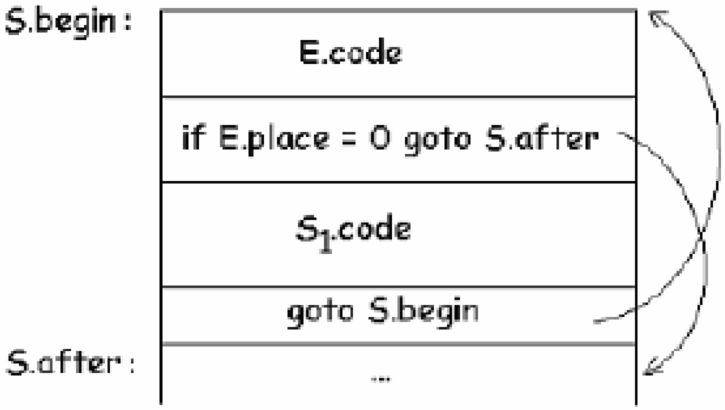


(a) if

(b) if-else

(c) while

SDD for control statement to generate three-address code



new S-attributes: begin, after are generated

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **while** E **do** E_1	S.begin := newlabel; S.after := newlabel; S.code := gen(S.begin ': ') || E.code || gen('if' E.place '= ' '0' 'goto' S.after) S_1.code || gen('goto' S.begin) || gen(S.after ': ')

Example

```
i := 2 * n + k
while i do
   i := i - k
```

```
     t1 := 2
     t2 := t1 * n
     t3 := t2 + k
     i  := t3
L1:  if i = 0 goto L2
     t4 := i - k
     i  := t4
     goto L1
L2:
```

SDD to generate three-address code

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\|\|\; label(B.true) \;\|\|\; S_1.code$ $\|\|\; gen(\texttt{'goto'}\ S.next)$ $\|\|\; label(B.false) \;\|\|\; S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\|\; B.code$ $\|\|\; label(B.true) \;\|\|\; S_1.code$ $\|\|\; gen(\texttt{'goto'}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \;\|\|\; label(S_1.next) \;\|\|\; S_2.code$

Backpatching

- Single Pass Solution to Code Generation?
  - No more symbolic labels - symbolic addresses instead
  - Emit code directly into an array of instructions
  - Actions associated with Productions
  - Executed when Bottom-Up Parser "Reduces" a production
- Problem
  - Need to know the labels for target branches before actually generating the code for them.
- Solution
  - Leave Branches undefined and patch them later
  - Requires: carrying around a list of the places that need to be patched until the value to be patched with is known.

Auxiliary functions

- Functions:
  - makelist(i): make a list with the label i
  - merge(p1,p2): creates a new list of labels with lists p1 and p2
  - backpatch(p,i): fills the locations in p with the address i
  - newAddr() : returns a new symbolic address in sequence and increments the value for the next call

- Array of Instructions
  - Linearly sequence of instructions
  - Function `emit` to generate actual instructions in the array
  - Symbolic Addresses

Boolean expression revisited

- Use Additional ε-Production
  - Just a Marker M
  - Label Value M.addr
- Attributes:
  - E.truelist: code places that need to be filled-in corresponding to the evaluation of E as "true".
  - E.falselist: same for "false"

(1) $E \rightarrow E_1$ or $M\ E_2$
(2) $\quad | \ E_1$ and $M\ E_2$
(3) $\quad | \ $ not E_1
(4) $\quad | \ (E_1)$
(5) $\quad | \ id_1$ relop id_2
(6) $\quad | \ $ true
(7) $\quad | \ $ false
(8) $M \rightarrow \varepsilon$

SDT for Boolean expressions

(1) $E \rightarrow E_1$ or $M\ E_2$	{ backpatch(E_1.falselist,M.Addr); E.truelist := merge(E_1.truelist,E_2.truelist); E.falselist := E_2.falselist; }
(2) $E \rightarrow E_1$ and $M\ E_2$	{ backpatch(E_1.truelist,M.Addr); E.truelist := E_2.truelist; E.falselist := merge(E_1.falselist, E_2.falselist); }
(8) $M \rightarrow \varepsilon$	{ M.Addr := nextAddr; }

SDT for Boolean expressions

(3) $E \rightarrow$ not E_1 { E.truelist := E_1.falselist; E.falselist := E_1.truelist; }

(4) $E \rightarrow (E_1)$ { E.truelist := E_1.truelist; E.falselist := E_1.falselist; }

(5) $E \rightarrow id_1$ relop id_2

```
{
  E.truelist := makelist(nextAddr());
  E.falselist := makelist(nextAddr());
  emit("if id1.place relop.op id2.place goto _");
  emit("goto _");
}
```

(6) $E \rightarrow$ true { E.truelist := makelist(nextAddr()); emit("goto _"); }

(7) $E \rightarrow$ false { E.falselist := makelist(nextAddr()); emit("goto _"); }

Control Flow and loops

- Add the `nextlist` attribute to S and N
  - denotes the set of locations in the S code to be patched with the address that follows the execution of S
  - Can be either due to control flow or fall-through

(1) $S \rightarrow$ if E then M_1 S_1 N else M_2 S_2 {
backpatch(E.truelist, M_1.addr);
backpatch(E.falselist,M_2.addr);
S.nextlist := merge(S_1.nextlist,merge(N.nextlist,S_2.nextlist));
}

(2) $N \rightarrow \varepsilon$ { N.nextlist := makelist(nextAddr();
emit("goto _"); }

(3) $M \rightarrow \varepsilon$ { M.quad := nextAddr; }

(4) $S \rightarrow$ if E then M S_1 {
backpatch(E.truelist, M.addr);
S.nextlist := merge(E.falselist,S_1.nextlist);
}

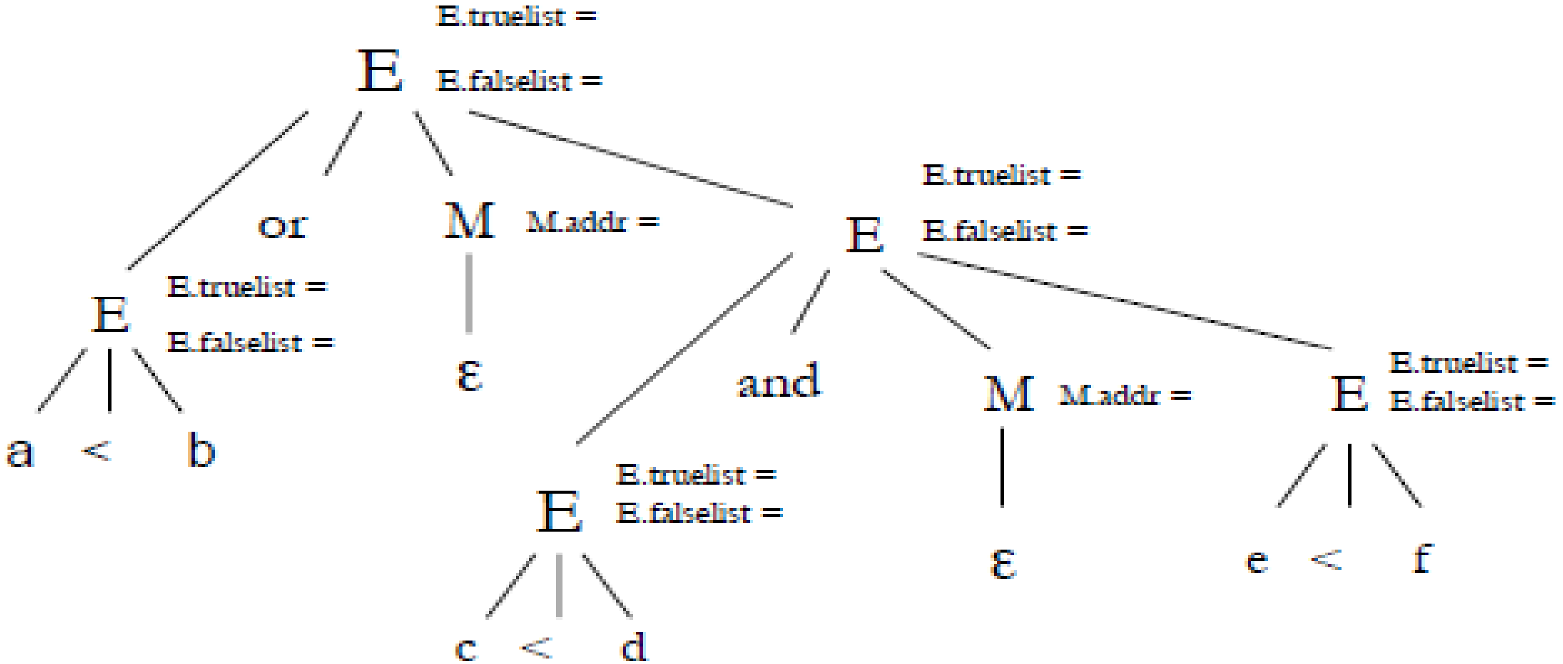
(5) $S \rightarrow$ while M_1 E do M_2 S_1 {
backpatch(S_1.nextlist, M_1.addr);
backpatch(E.truelist,M_2.addr);
S.nextlist := E.falselist;
emit("goto M_1.addr");
}

Example

Executing Action **Generated Code**

E E.truelist
E.falselist

M M.addr



Example

E E.truelist E.falselist

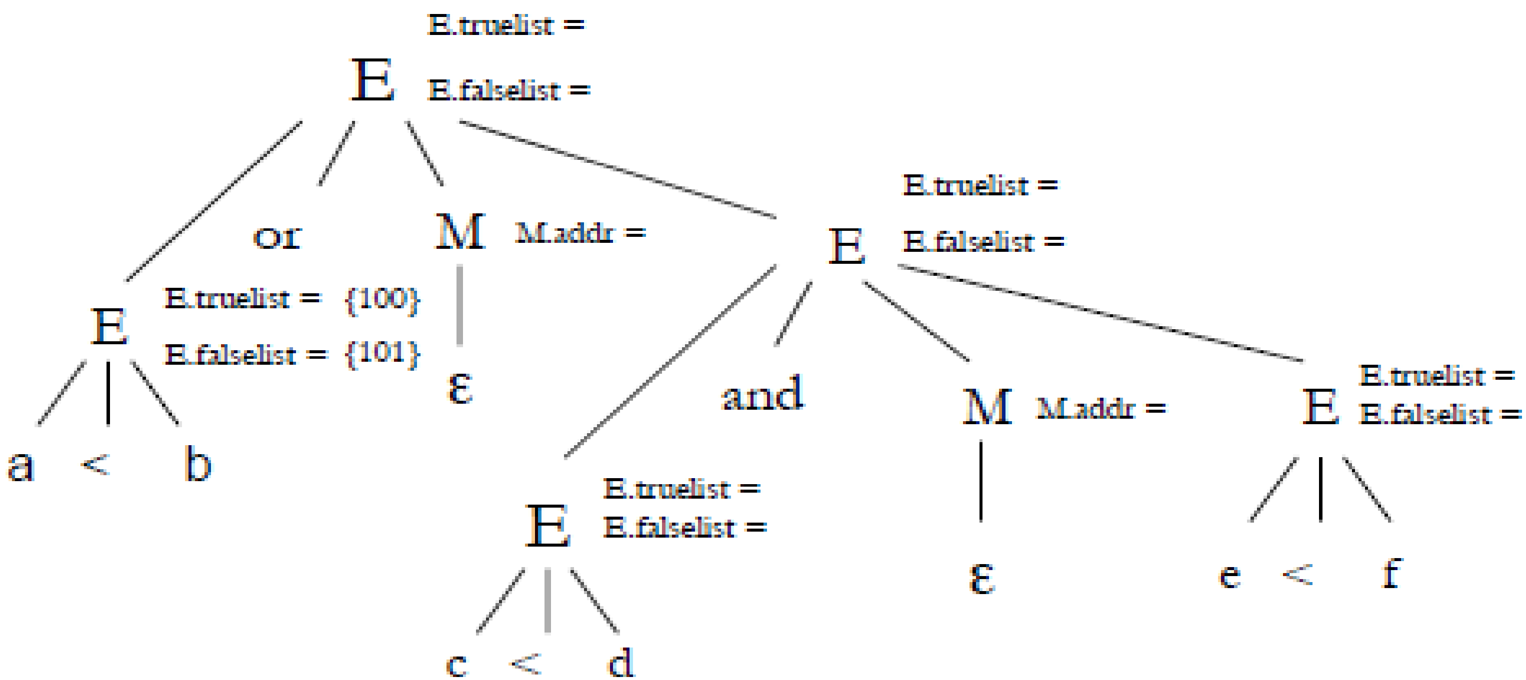
M M.addr

Executing Action

```
{ E.truelist := makelist(nextquad());
  E.falselist := makelist(nextquad());
  emit("if id1.place relop.op id2.place goto _");
  emit("goto _"); }
```

Generated Code

```
100: if a < b goto _
101: goto _
```



Example

Executing Action | **Generated Code**

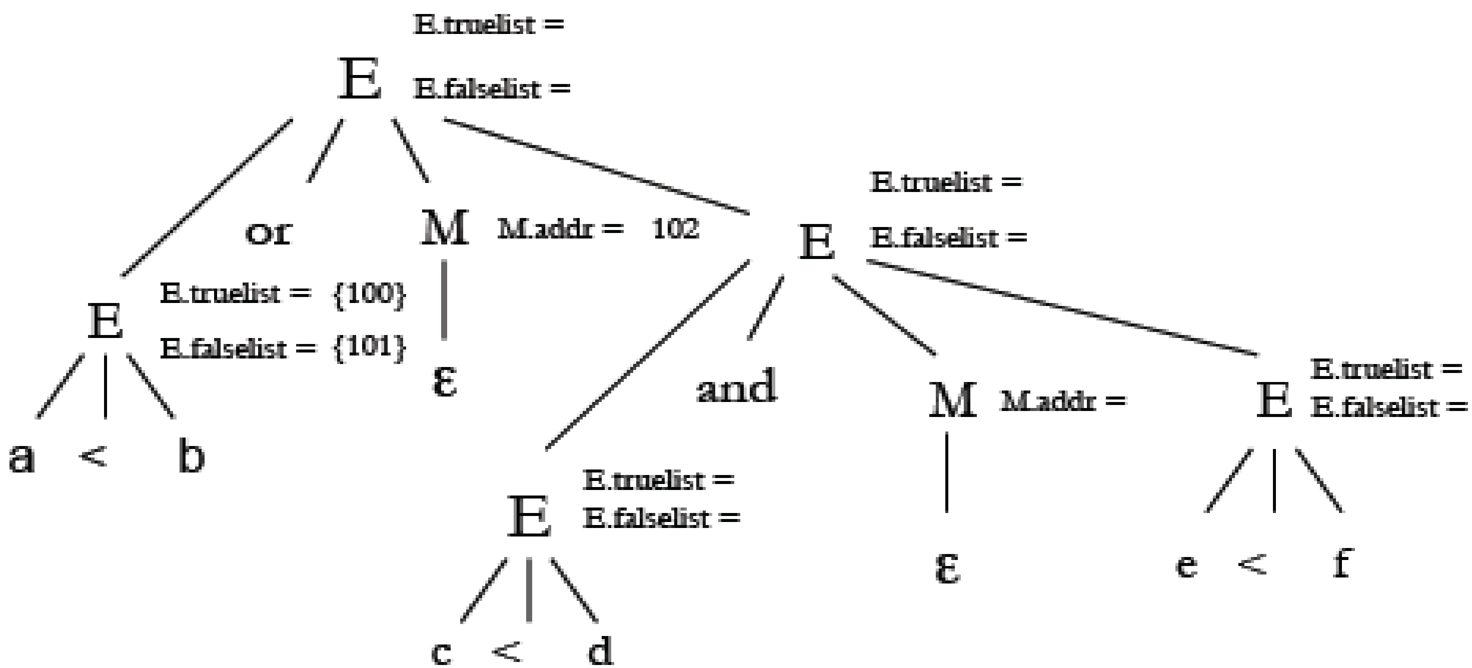
E E.truelist E.falselist

M M.addr

{ M.quad = nextquad(); }

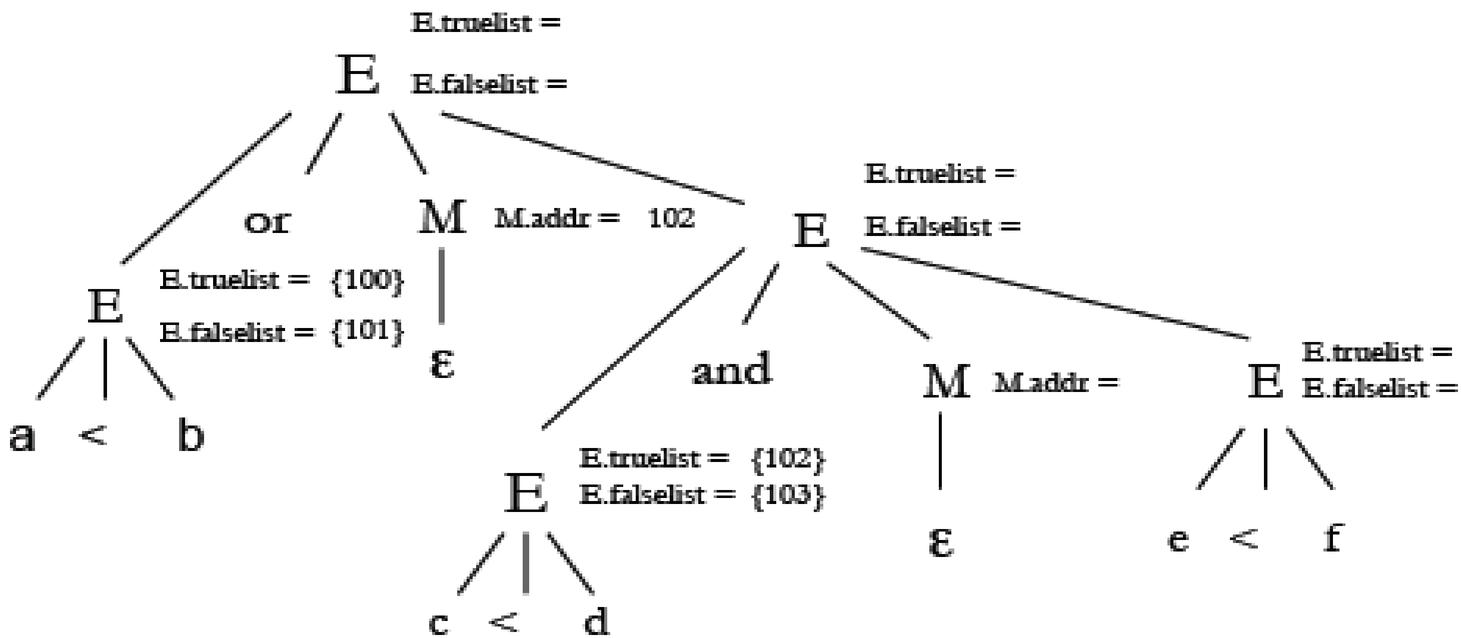
100: if a < b goto _

101: goto _



Example

	Executing Action	**Generated Code**
E E.truelist E.falselist	{ E.truelist := makelist(nextquad()); E.falselist := makelist(nextquad()); emit("if id1.place relop.op id2.place goto _"); emit("goto _"); }	100: if a < b goto _ 101: goto _ 102: if c < d goto _ 103: goto _
M M.addr		



Example

Executing Action | **Generated Code**

E E.truelist E.falselist

M M.addr

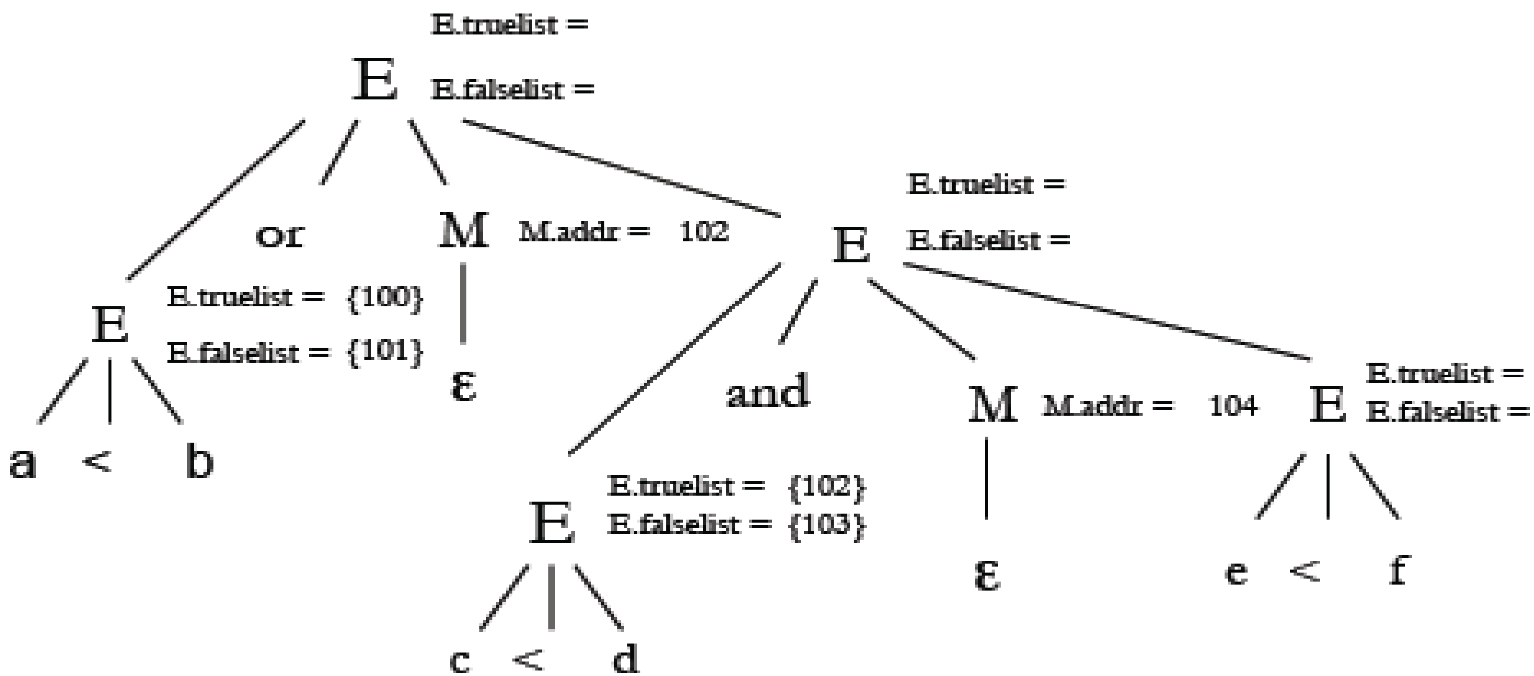
{ M.quad = nextquad(); }

100: if a < b goto _

101: goto _

102: if c < d goto _

103: goto _



Example

E: E.truelist, E.falselist

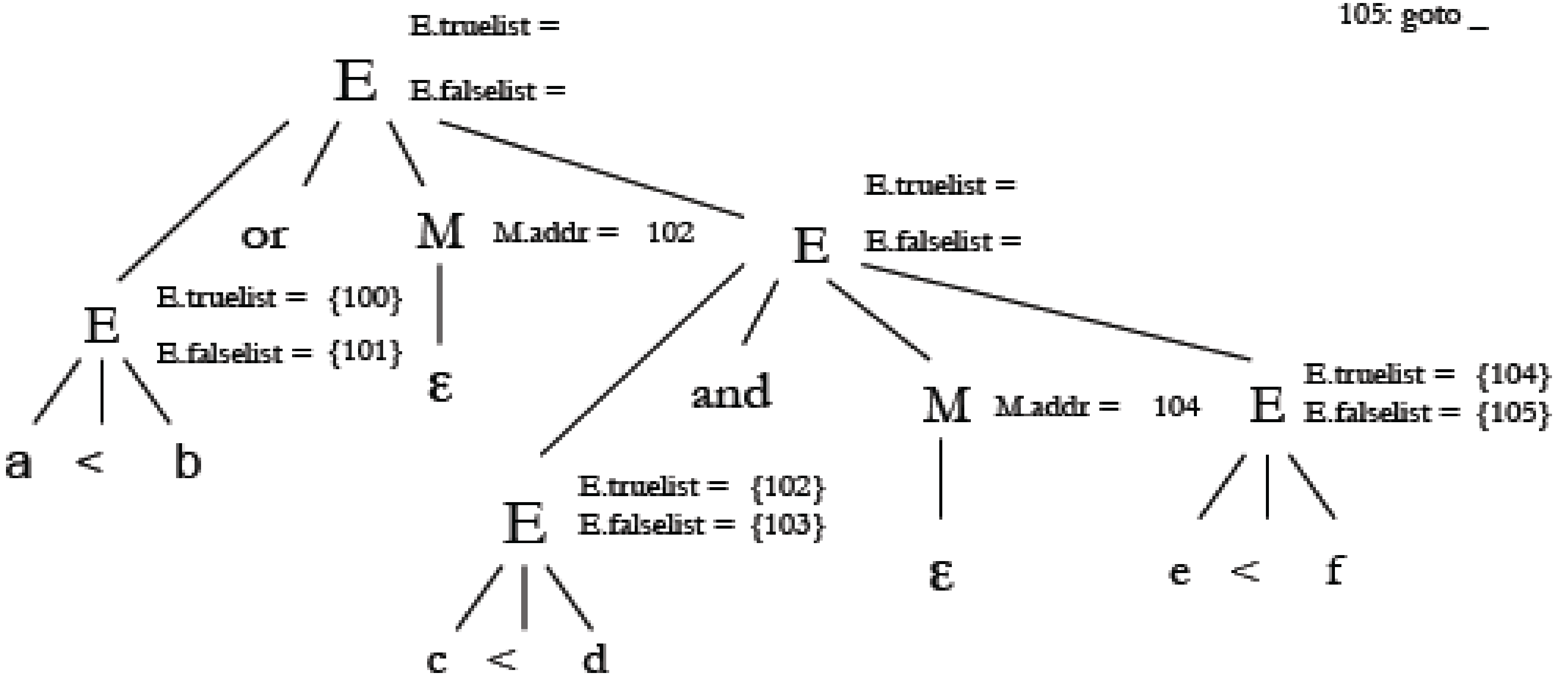
M: M.addr

Executing Action

{ E.truelist := makelist(nextquad());
E.falselist := makelist(nextquad());
emit("if id1.place relop.op id2.place goto _");
emit("goto _"); }

Generated Code

100: if a < b goto _
101: goto _
102: if c < d goto _
103: goto _
104: if e < f goto _
105: goto _



Example

Executing Action

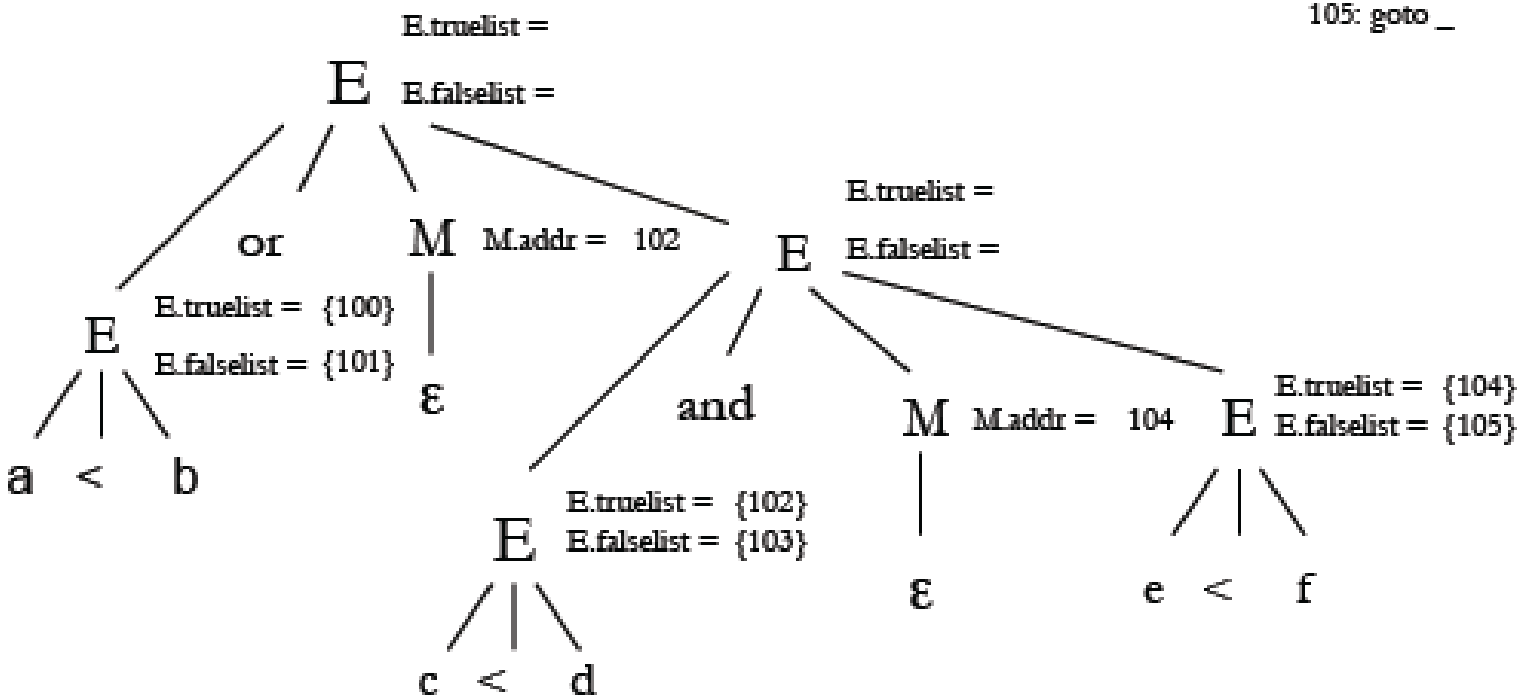
Generated Code

E E.truelist E.falselist

M M.addr

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

100: if a < b goto _
101: goto _
102: if c < d goto _
103: goto _
104: if e < f goto _
105: goto _



Example

Executing Action

E E.truelist E.falselist

M M.addr

{ backpatch(E_1.truelist,M.quad);

E.truelist := E_2.truelist;

E.falselist := merge(E_1.falselist,E_2.falselist; }

Generated Code

100: if a < b goto _

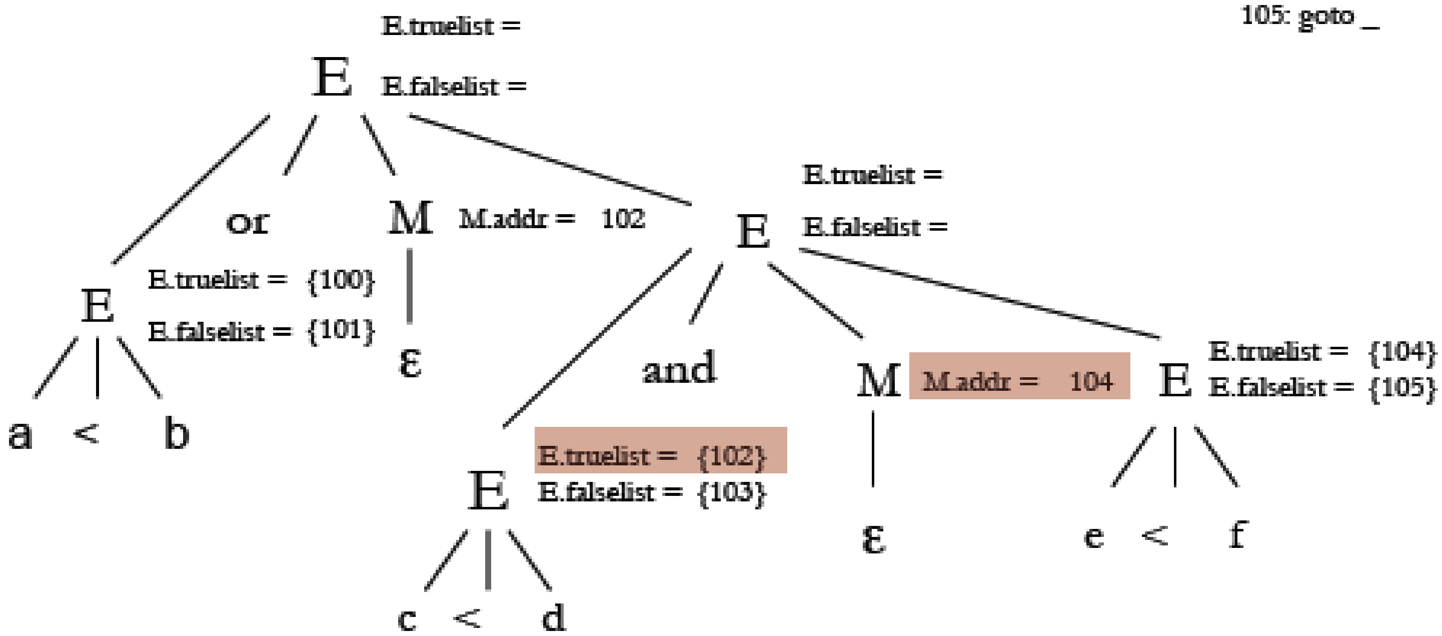
101: goto _

102: if c < d goto _

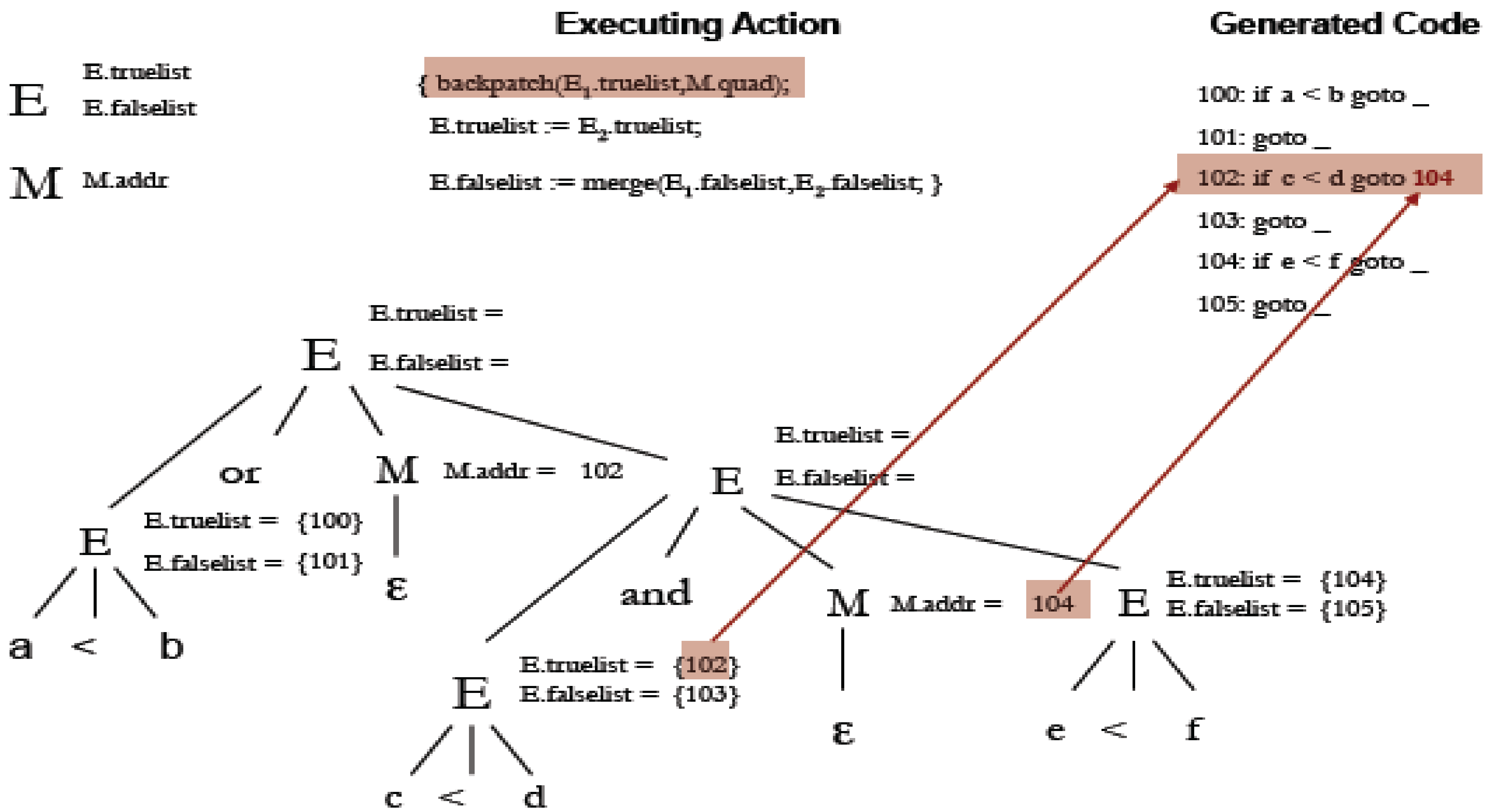
103: goto _

104: if e < f goto _

105: goto _



Example



Executing Action
Generated Code
E
E.truelist
E.falselist
M
M.addr
{ backpatch(E1.truelist,M.quad);
E.truelist := E2.truelist;
E.falselist := merge(E1.falselist,E2.falselist; }
100: if a < b goto _
101: goto _
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _
E.truelist =
E.falselist =
or
M.addr = 102
E.truelist = {100}
E.falselist = {101}
a < b
ε
and
M.addr = 104
E.truelist = {104}
E.falselist = {105}
E.truelist = {102}
E.falselist = {103}
c < d
e < f

Example

Executing Action	Generated Code

E E.truelist, E.falselist

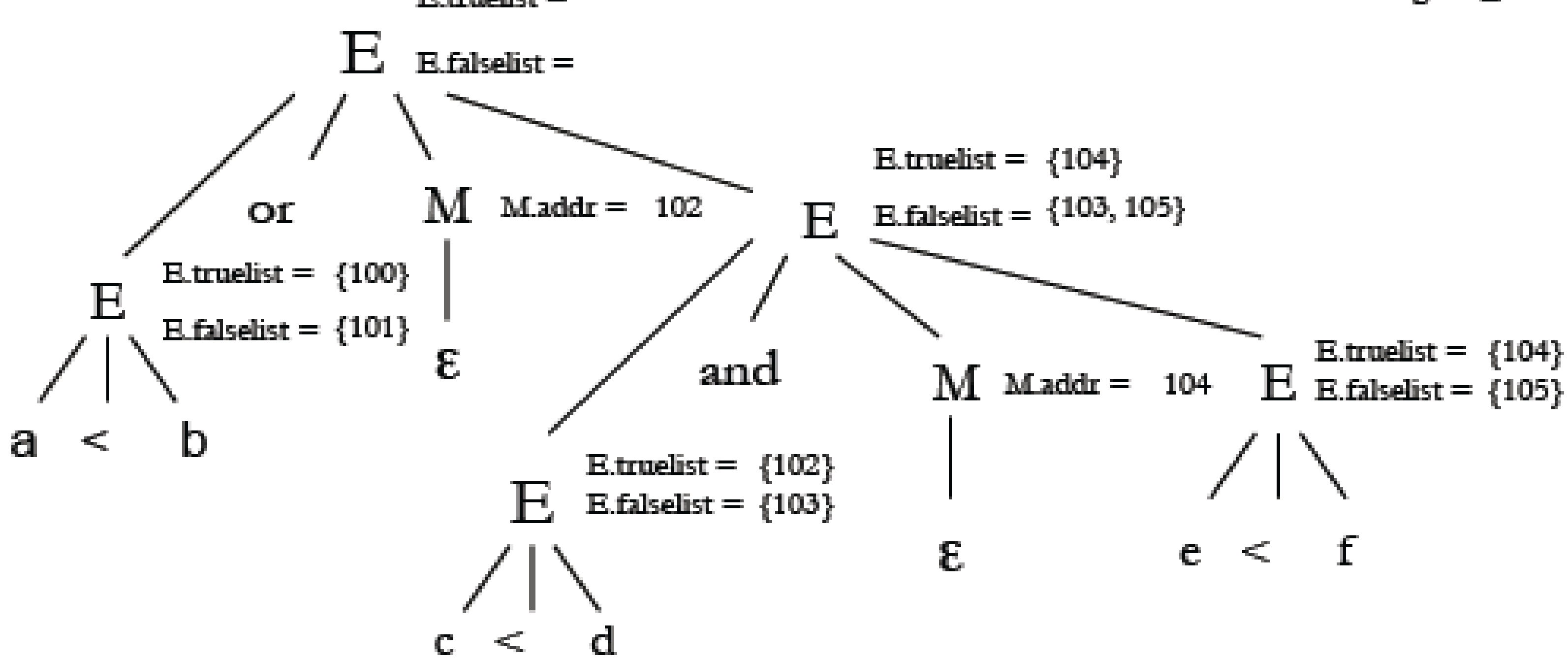
M M.addr

Executing Action

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

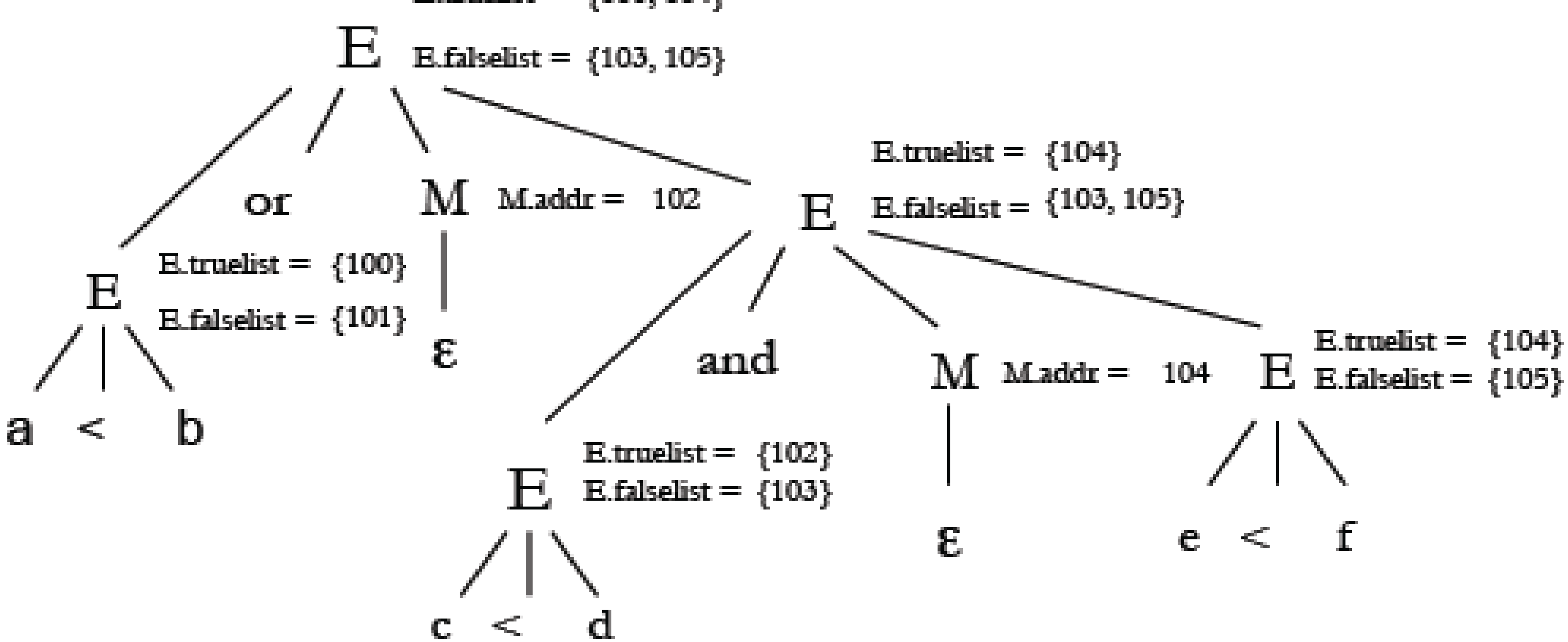
Generated Code

100: if a < b goto _
101: goto _
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _



Example

	Executing Action	**Generated Code**
E: E.truelist, E.falselist	{ backpatch(E_1.truelist,M.quad);	100: if a < b goto _
	E.truelist := E_2.truelist;	101: goto 102
M: M.addr	E.falselist := merge(E_1.falselist,E_2.falselist; }	102: if c < d goto 104
		103: goto _
		104: if e < f goto _
		105: goto _

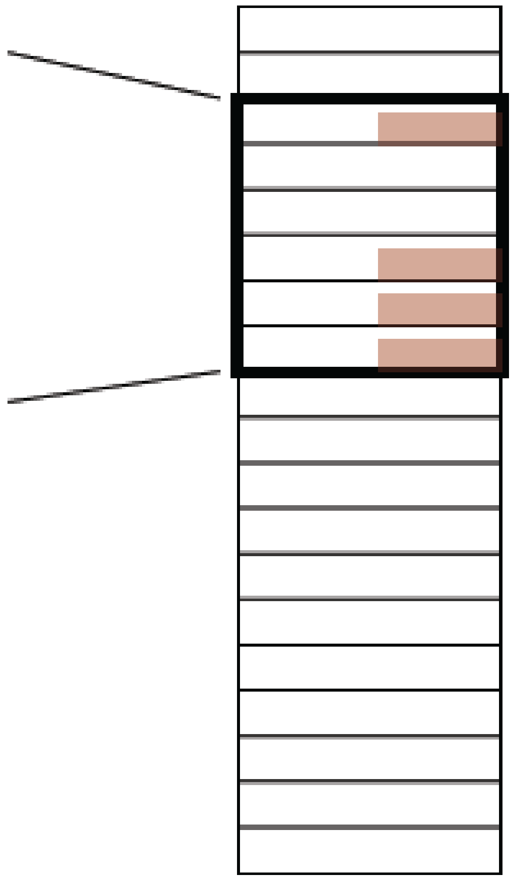


Example

$S \rightarrow$ while M_1 E do M_2 S_1 { backpatch(S_1.nextlist, M_1.addr);
backpatch(E.truelist,M_2.addr);
S.nextlist := E.falselist;
emit("goto M_1.addr");
}

E

```
100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _
```

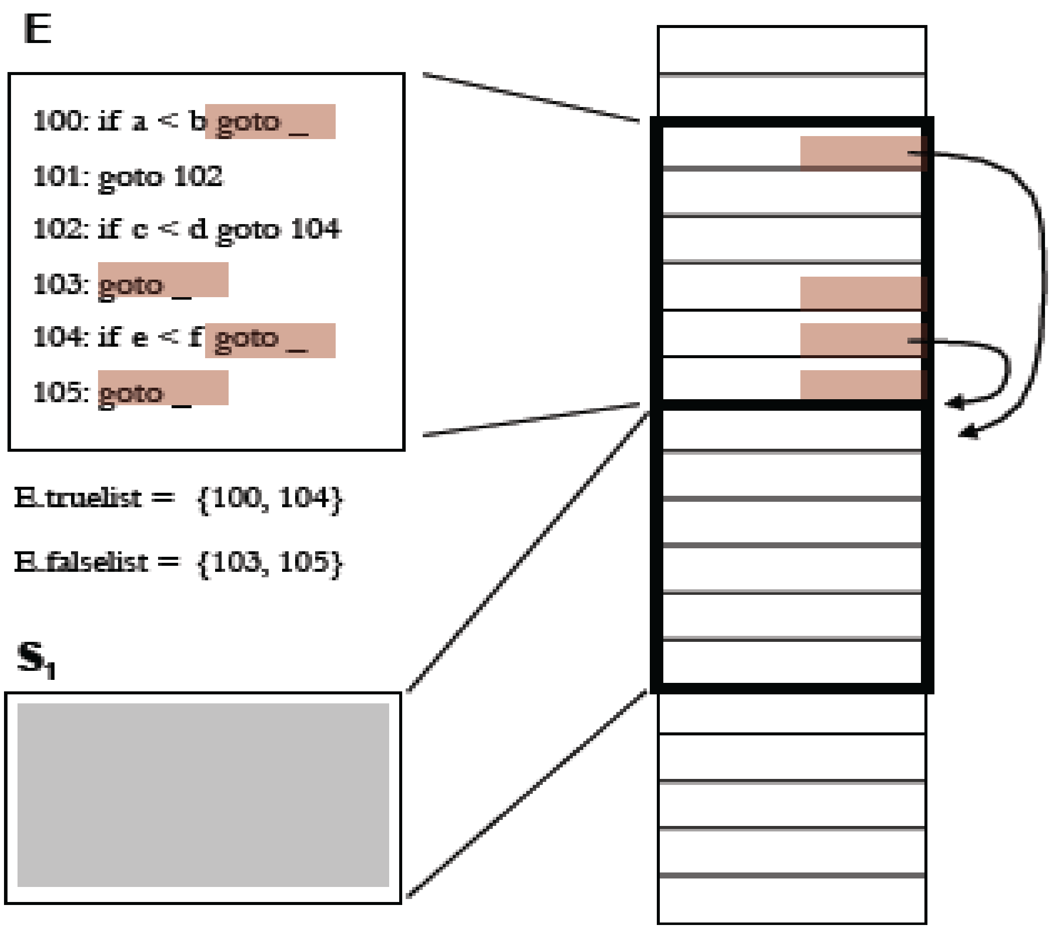


E.truelist = {100, 104}

E.falselist = {103, 105}

Example

$S \rightarrow$ while M_1 E do M_2 S_1 { backpatch(S_1.nextlist, M_1.addr);
backpatch(E.truelist,M_2.addr);
S.nextlist := E.falselist;
emit("goto M_1.addr");
}



Example

$S \rightarrow$ while M_1 E do M_2 S_1 { backpatch(S_1.nextlist, M_1.addr);
backpatch(E.truelist,M_2.addr);
S.nextlist := E.falselist;
emit("goto M_1.addr");
}