Backpatching and procedures

Backpatching introduction

- SDD is done in two passes
  - Construct the syntax tree
  - Walk in DFS to perform SDT
- In a single pass, labels may not be known hence, introduce a technique called backpatching

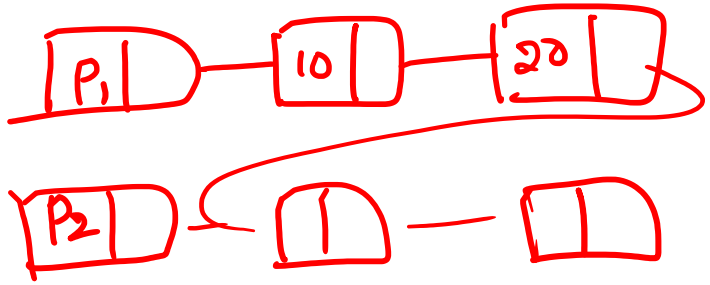
Backpatching

- Generate a series of branching statements with the target of jumps unspecified
- Put each statement in a list and use a second pass to fill the labels

Backpatching

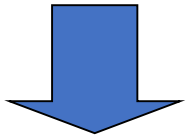
- *makelist*(*i*) creates a new list containing three-address location *i*, returns a pointer to the list
- *merge*(p_1, p_2) concatenates lists pointed to by p_1 and p_2, returns a pointer to the concatenates list
- *backpatch*(*p*, *i*) inserts *i* as the target label for each of the statements in the list pointed to by *p*

backpatch(P1, 25)



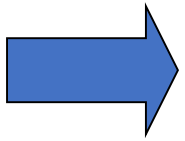
Backpatching - Example

a < b or c < d and e < f



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

backpatch



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

Backpatching - SDD

Production	Semantic Rule	Inference
M → ε	{ M.quad := nextquad() }	
E → E1 or M E2	{ backpatch(E_1.falselist, M.quad); E.truelist := merge(E_1.truelist, E_2.truelist); E.falselist := E_2.falselist }	Same as earlier.. Just list formed
E → E1 and M E2	{ backpatch(E_1.truelist, M.quad); E.truelist := E_2.truelist; E.falselist := merge(E_1.falselist, E_2.falselist); }	
E → not E1	{ E.truelist := E_1.falselist; E.falselist := E_1.truelist }	
E → (E1)	{ E.truelist := E_1.truelist; E.falselist := E_1.falselist }	

Production	Semantic Rule	Inference
E → id1 relop id2	{ E.truelist := makelist(nextquad()); E.falselist := makelist(nextquad() + 1); emit('**`if`**' **id**$_1$.place **relop**.op **id**$_2$.place '**`goto _`**'); emit('**`goto _`**') }	
E→ true	{ E.truelist := makelist(nextquad()); E.falselist := nil; emit('**`goto _`**') }	
E → false	{ E.falselist := makelist(nextquad()); E.truelist := nil; emit('**`goto _`**') }	

Backpatching: Grammar

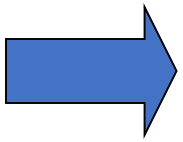
$S \rightarrow$ **if** E **then** S
| **if** E **then** S **else** S
| **while** E **do** S
| **begin** L **end**
| A

$L \rightarrow L$ **;** S
| S

S_1 **;** S_2 **;** S_3 **;** S_4 **;** S_4 …

Synthesized attributes:

S.nextlist	backpatch list for jumps to the next statement after S (or nil)
L.nextlist	backpatch list for jumps to the next statement after L (or nil)



```
100: Code for S1
200: Code for S2
300: Code for S3
400: Code for S4
500: Code for S5
```

Jumps out of S_1

backpatch(S_1.nextlist, 200)
backpatch(S_2.nextlist, 300)
backpatch(S_3.nextlist, 400)
backpatch(S_4.nextlist, 500)

Syntax directed definition for flow control

Production	Semantic Rules
$S \rightarrow A$	{ S.nextlist := nil }
$S \rightarrow$ **begin** L **end**	{ S.nextlist := L.nextlist }
$S \rightarrow$ **if** E **then** M S_1	{ backpatch(E.truelist, M.quad); S.nextlist := merge(E.falselist, S_1.nextlist) }
$L \rightarrow L_1$ **;** M S	{ backpatch(L_1.nextlist, M.quad); L.nextlist := S.nextlist; }
$L \rightarrow S$	{ L.nextlist := S.nextlist; }
$M \rightarrow \varepsilon$	{ M.quad := nextquad() }

Syntax directed definition for if-then

Production	Semantic Rules
$S \rightarrow$ **if** E **then** M_1 S_1 N **else** M_2 S_2	{ backpatch(E.truelist, M_1.quad); backpatch(E.falselist, M_2.quad); S.nextlist := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist)) }
$S \rightarrow$ **while** M_1 E **do** M_2 S_1	{ backpatch(S_1.nextlist, M_1.quad); backpatch(E.truelist, M_2.quad); S.nextlist := E.falselist; emit('`goto _`') }
$N \rightarrow \varepsilon$	{ *N*.nextlist := *makelist*(*nextquad*()); *emit*('`goto _`') }

Boolean expression

- a < b or c < d and e < f

100: if a < b goto ---

101: goto --- 102

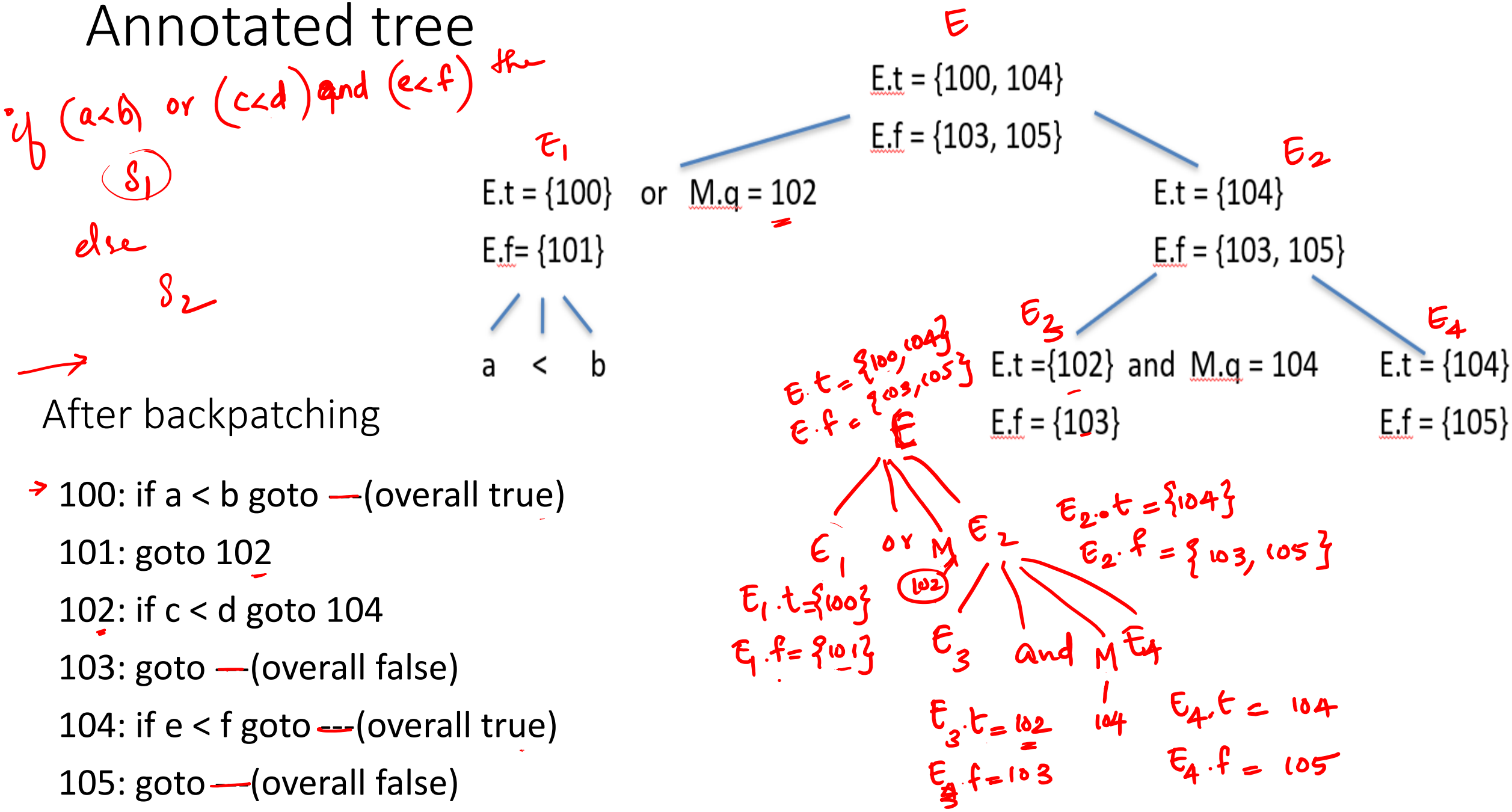
102: if c < d goto --- 104

103: goto ---

104: if e < f goto ---

105: goto ---

Annotated tree



After backpatching

100: if a < b goto —(overall true)

101: goto 102

102: if c < d goto 104

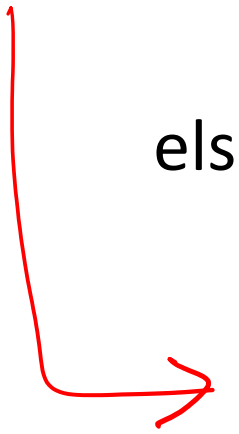
103: goto —(overall false)

104: if e < f goto —(overall true)

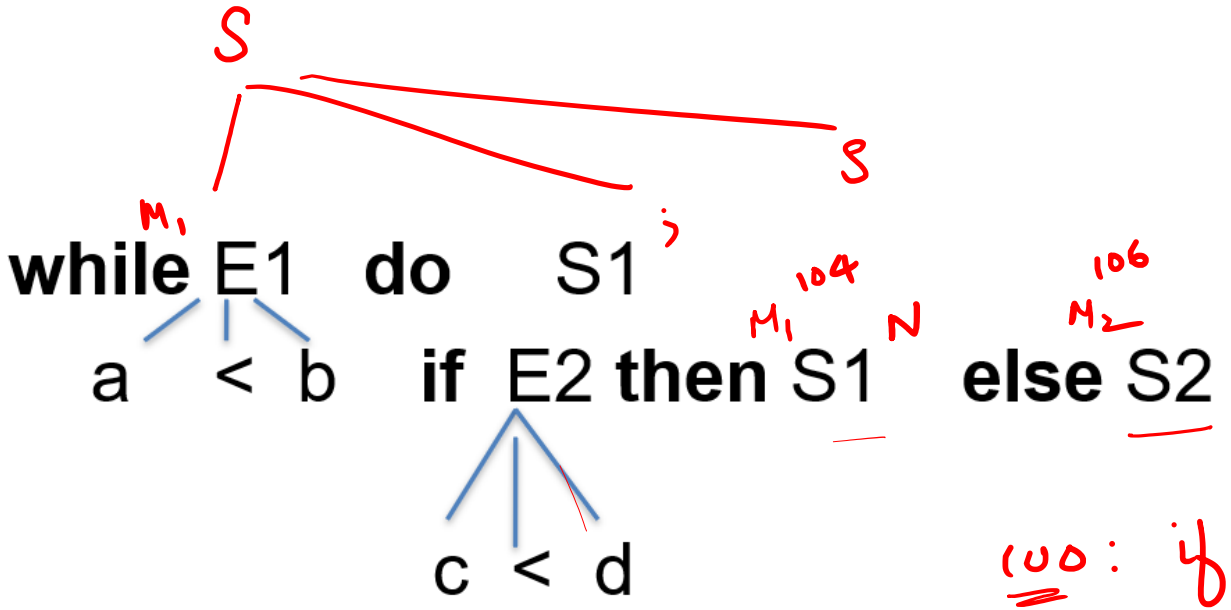
105: goto —(overall false)

Example while

```
while a < b do
        if c < d then
                        x : = y + z
   else
                        x : = y - z
```



Derivation



S1.nextlist = { N.nextlist, S1.nextlist, S2.nextlist }
= { 105,
N.nextlist = { 105 }

$M_2 = 102$

$M_1 = 100$

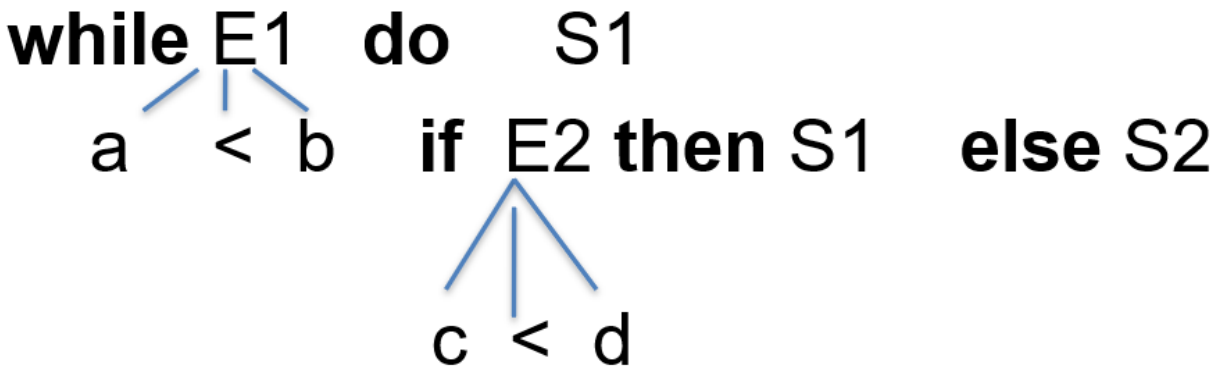
$E_1.t = \{100\}$
$E_1.f = \{101\}$

$E_2.t = \{102\}$
$E_3.f = \{103\}$

100 : if a<b goto 102
101 : goto ___ overall false
102 : if c<d goto 104
103 : goto 106
104 : x = y + z
105 : goto 100
106 : x = y - z
107 : goto 100

Three address code to start

- 100: if (a<b) goto ---
- 101: goto ---
- 102: if (c < d) goto ---
- 103: goto ---
- 104: x:= y + z
- 105: goto ---
- 106: x:= y – z
- 107: goto ---



Three address code after backpatch

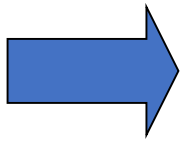
- 100: if (a<b) goto 102
- 101: goto 108 overall false
- 102: if (c < d) goto 104
- 103: goto 106
- 104: x:= y + z
- 105: goto 100
- 106: x:= y – z
- 107: goto 100

Translating Procedure Calls

$S \rightarrow$ **call id (** *Elist* **)**

Elist $\rightarrow$ *Elist* **,** *E* | *E*

foo(a+1, b, 7)



```
t1 := a + 1
t2 := 7
param t1
param b
param t2
call foo 3
```

Translating Procedure Calls

- $S \rightarrow$ **call id (** *Elist* **)**

  { **for** each item *p* on *queue* **do**
  emit('**`param`**' *p*);
  emit('**`call`**' **id**.place |*queue*|) }

  Elist $\rightarrow$ *Elist* **,** *E*

  { append *E*.place to the end of *queue* }

  Elist $\rightarrow$ *E*

  { initialize *queue* to contain only *E*.place }

Mixed mode Boolean expressions

- Boolean expressions can have arithmetic sub-expressions
- Boolean can be considered as arithmetic in languages where true is “1’ and false is “0”
- Short-circuit could still be used

Evaluation

- E → E+E | E and E | E relop E | id
- E+E could produce arithmetic result and could be “and” with another boolean expression’s result

Three address code for E → E1+E2

E.Type := arith;

if E1.type == arith and E2.type = arith

  E.place := newtemp

  E.code := E1.code || E2.code ||

                gen(E.place ‘:=‘ E1.place ‘+’ E2.place)

Three address code for E → E1+E2

```
else if E1.type = arith and E2.type = bool
  E.place := newtemp
  E2.true := newlabel
  E2.false := newlabel
  E.code := E1.code || E2.code ||
                gen(E2.true ‘:’ E.place ‘:=‘ E1.place +1)
                gen(‘goto’ nextstat +1)
                gen(E2.false ‘:’ E.place ‘:=‘ E1.place)
else.
```

Case statements

Switch expression

begin

case value: statement

case value: statement

…

default: statement

end

Switch

- Evaluate the expression
- Find which value in the list of cases is the same as the value of the expression
- Execute the statement associated with the value found

Translation

- Code to evaluate E into t
- goto test
- L1: code for S1
  goto next
- L2: code for S2
  goto next

  ….

Translation

- Ln: code for Sn
  goto next
- test: if t = V1 goto L1
  if t = V2 goto L2
  …
  if t = Vn-1 goto Ln-1
  goto Ln
- next:

Summary

- Control flow statements with backpatching
- Boolean expressions with backpatching
- three address code for Procedures and case statements