Lecture 09 – IR (Backpatching)

# THEORY OF COMPILATION

Eran Yahav

www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec09.pptx

Reference: Dragon 6.2,6.3,6.4,6.6

# Recap

- Lexical analysis
  - regular expressions identify tokens (“words”)
- Syntax analysis
  - context-free grammars identify the structure of the program (“sentences”)
- Contextual (semantic) analysis
  - type checking defined via typing judgments
  - can be encoded via attribute grammars
- Syntax directed translation (SDT)
  - attribute grammars
- Intermediate representation
  - many possible IRs
  - generation of intermediate representation
  - 3AC

# Journey inside a compiler

```
float position;
float initial;
float rate;
position = initial + rate * 60
```

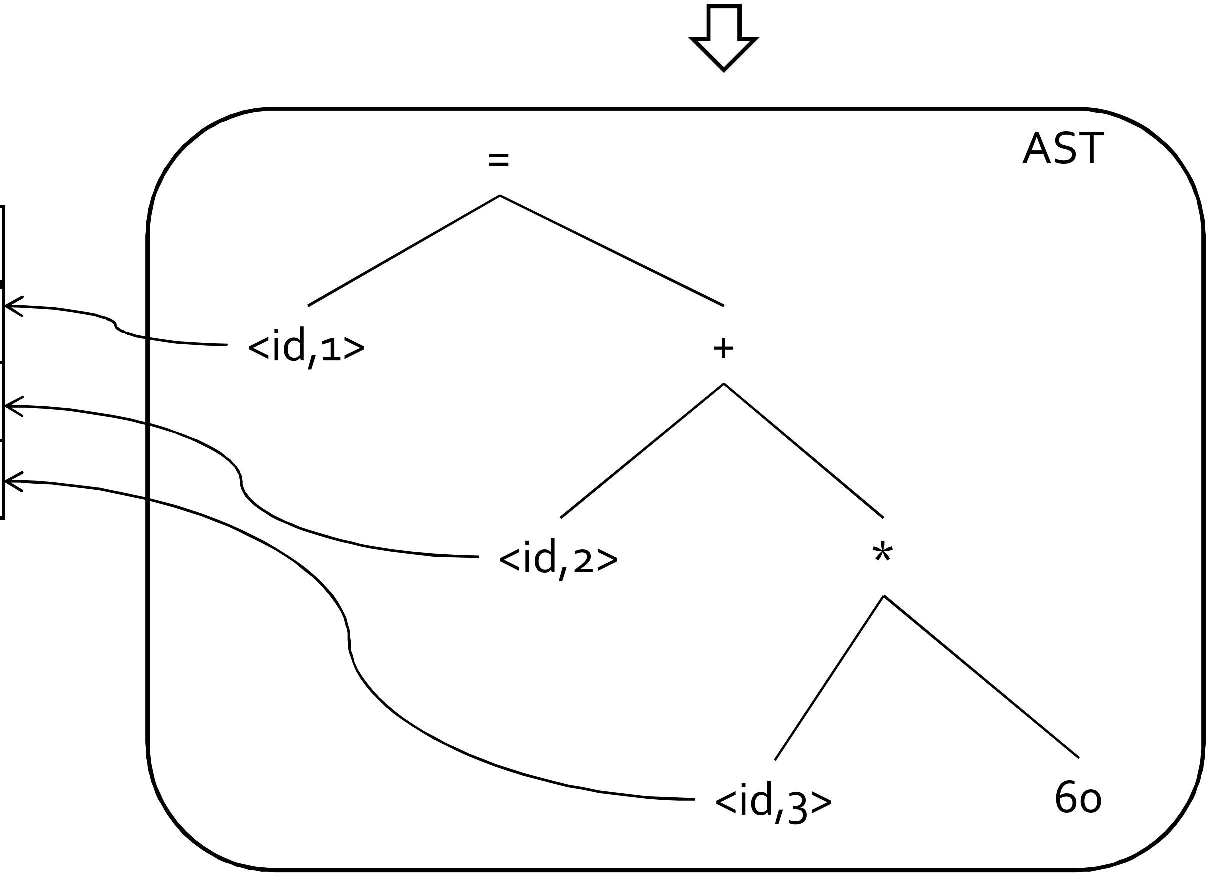
Token Stream

<float> <ID,position> <;> <float> <ID,initial> <;> <float> <ID,rate> <;> <ID,1> <=> <ID,2> <+> <ID,3> <*> <60>

Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

# Journey inside a compiler

<ID,1> <=> <ID,2> <+> <ID,3> <*> <60>



symbol table

| id | symbol | type | data |
|---|---|---|---|
| 1 | position | float | … |
| 2 | initial | float | … |
| 3 | rate | float | … |

S ⟶ ID = E
E ⟶ ID
 | E + E
 | E * E
 | NUM

Lexical Analysis

Syntax Analysis

Sem. Analysis

Inter. Rep.

Code Gen.

# Problem 3.8 from [Appel]

A simple left-recursive grammar:

$E \rightarrow E + id$

$E \rightarrow id$

A simple right-recursive grammar accepting the same language:

$E \rightarrow id + E$

$E \rightarrow id$

Which has better behavior for shift-reduce parsing?

# Answer

Input

id+id+id+id+id

left recursive

$E \rightarrow E + id$
$E \rightarrow id$

stack

id (reduce)
E
E +
E + id (reduce)
E
E +
E + id (reduce)
E
E +
E + id (reduce)
E
E +
E + id (reduce)
E

The stack never has more than three items on it. In general, with LR-parsing of left-recursive grammars, an input string of length O(n) requires only O(1) space on the stack.

# Answer

Input

id+id+id+id+id

right recursive

E → id + E
E → id

stack

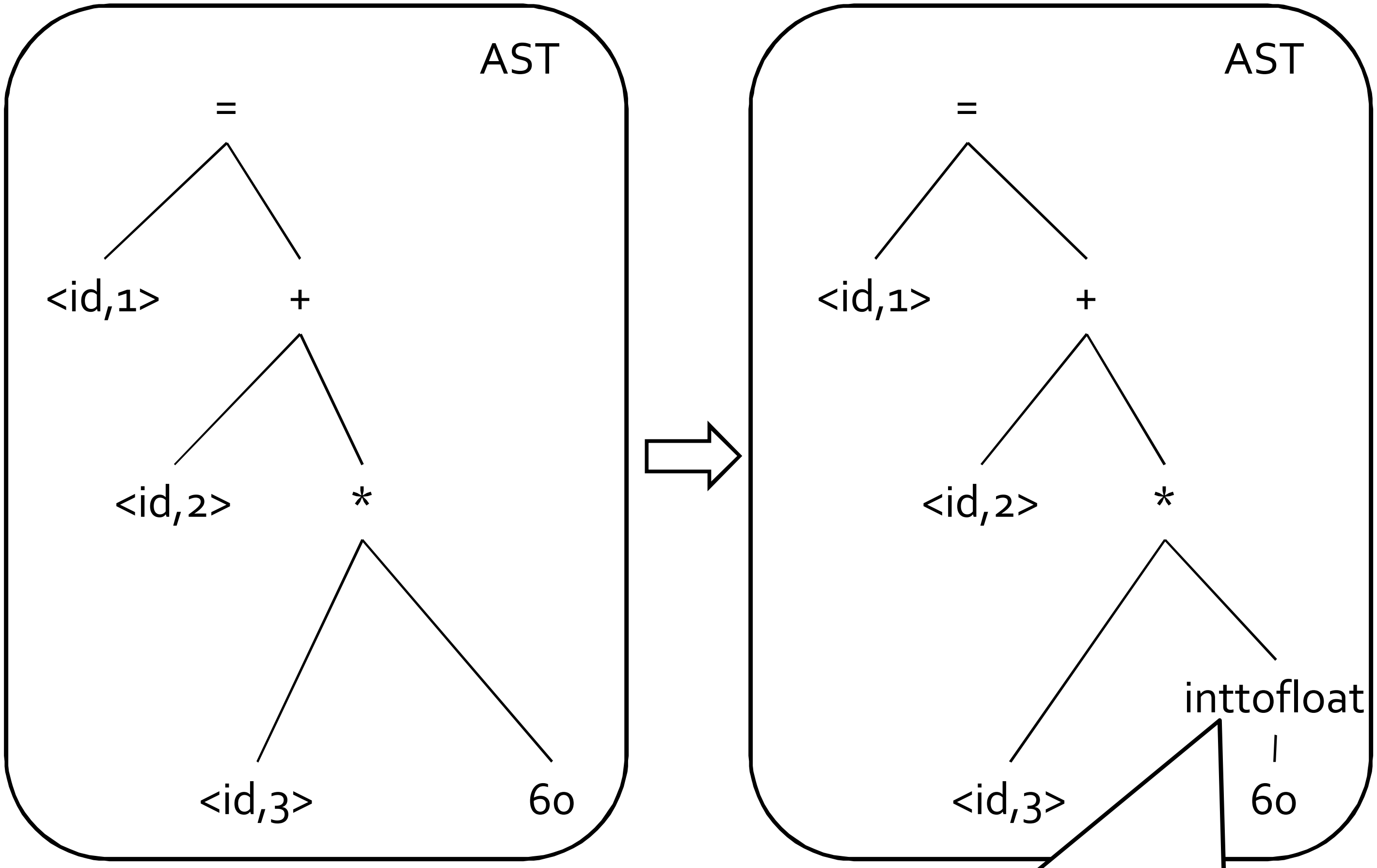
id
id +
id + id
id + id +
id + id + id
id + id + id
id + id + id + id
id + id + id + id +
id + id + id + id + id (reduce)
id + id + id + id + E (reduce)
id + id + id + E (reduce)
id + id + E (reduce)
id + E (reduce)
E

The stack grows as large as the input string. In general, with LR-parsing of right-recursive grammars, an input string of length O(n) requires O(n) space on the stack.

# Journey inside a compiler

symbol table

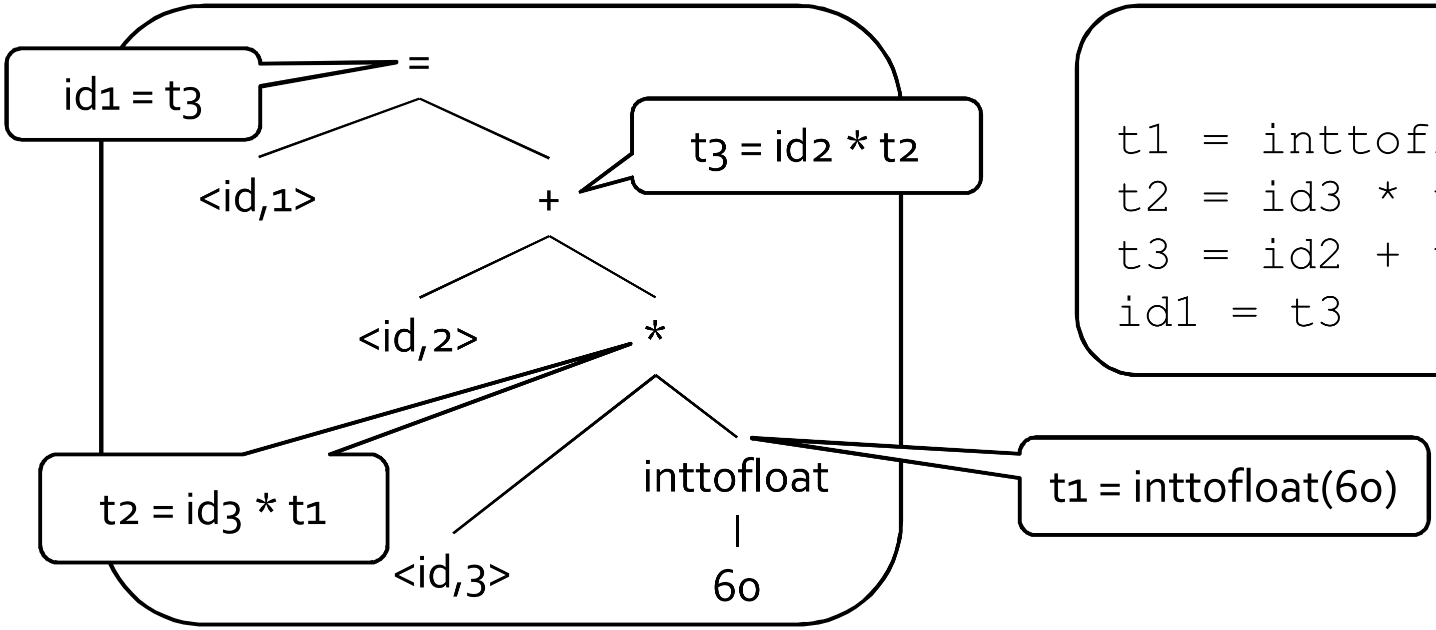
| id | symbol | type |
|---|---|---|
| 1 | position | float |
| 2 | initial | float |
| 3 | rate | float |



coercion: automatic conversion from int to float inserted by the compiler

Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

# Journey inside a compiler



3AC

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

| **production** | **semantic rule** |
|---|---|
| S → id = E | S.code := E. code \|\| gen(id.var ‘:=‘ E.var) |
| E → E1 op E2 | E.var := freshVar();<br>E.code = E1.code \|\| E2.code \|\| gen(E.var ‘:=‘ E1.var ‘op’ E2.var) |
| E → inttofloat(num) | E.var := freshVar();<br>E.code = gen(E.var ‘:=‘ inttofloat(num)) |
| E → id | E.var := id.var; E.code = ‘’ |

(for brevity, bubbles show only code generated by the node and not all accumulated “code” attribute)

note the structure:
translate E1
translate E2
handle operator

Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

# Journey inside a compiler

3AC

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Optimized

```
t1 = id3 * 60.0
id1 = id2 + t1
```

value known at compile time
can generate code with converted value

eliminated temporary t3

Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

# Journey inside a compiler

Optimized

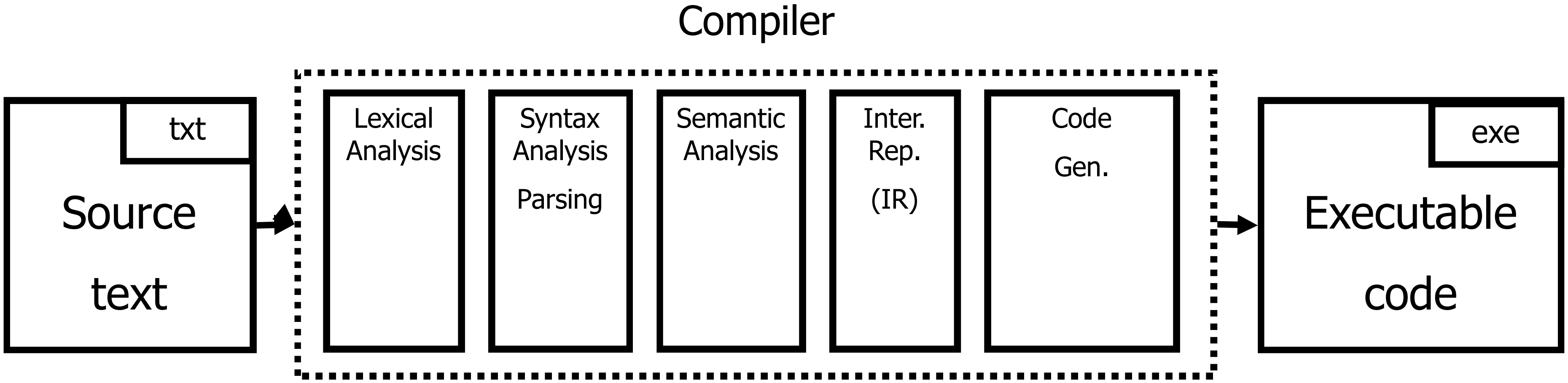
```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Gen

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1,R1,R2
STF id1,R1
```

Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

# You are here


Compiler

txt

Source text

Lexical Analysis

Syntax Analysis

Parsing

Semantic Analysis

Inter. Rep.

(IR)

Code

Gen.

exe

Executable code

# IR So Far…

- many possible intermediate representations
- 3-address code (3AC)
- Every instruction operates on at most three addresses
  - result = operand1 operator operand2
- gets us closer to code generation
- enables machine-independent optimizations
- how do we generate 3AC?

# Last Time: Creating 3AC

- Creating 3AC via syntax directed translation
- Attributes
    - code – code generated for a nonterminal
    - var – name of variable that stores result of nonterminal

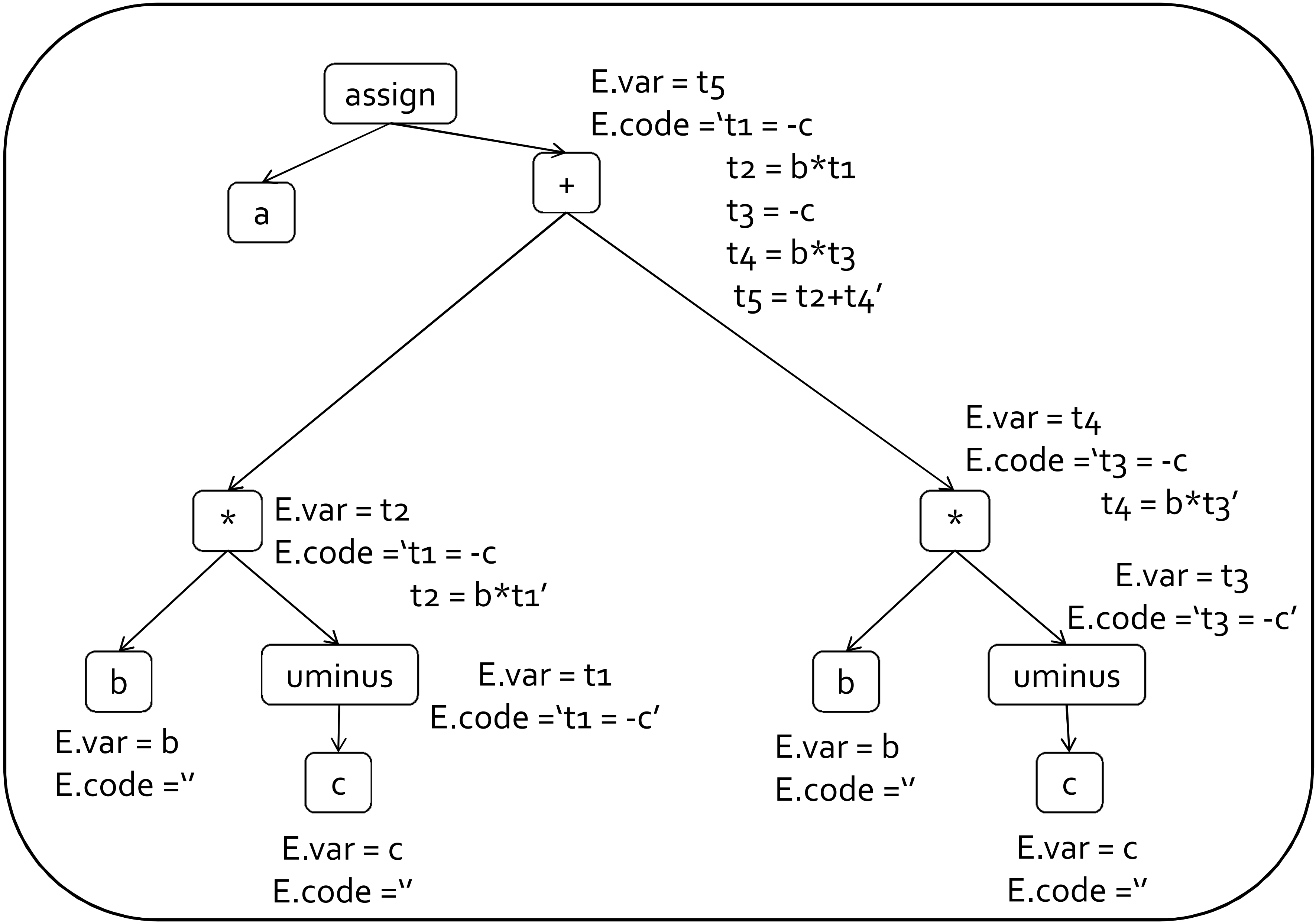
- freshVar() – helper function that returns the name of a fresh variable

# Creating 3AC: expressions

<table>
<tr><th>production</th><th>semantic rule</th></tr>
<tr><td>S → id := E</td><td>S.code := E. code || gen(id.var ‘:=‘ E.var)</td></tr>
<tr><td>E → E1 + E2</td><td>E.var := freshVar();<br>E.code = E1.code || E2.code || gen(E.var ‘:=‘ E1.var ‘+’ E2.var)</td></tr>
<tr><td>E → E1 * E2</td><td>E.var := freshVar();<br>E.code = E1.code || E2.code || gen(E.var ‘:=‘ E1.var ‘*’ E2.var)</td></tr>
<tr><td>E → - E1</td><td>E.var := freshVar();<br>E.code = E1.code || gen(E.var ‘:=‘ ‘uminu’ E1.var)</td></tr>
<tr><td>E → (E1)</td><td>E.var := E1.var<br>E.code = ‘(‘ || E1.code || ‘)’</td></tr>
<tr><td>E → id</td><td>E.var := id.var; E.code = ‘’</td></tr>
</table>

(we use || to denote concatenation of intermediate code fragments)

# example



E.var = t5
E.code =‘t1 = -c
t2 = b*t1
t3 = -c
t4 = b*t3
t5 = t2+t4’

E.var = t2
E.code =‘t1 = -c
t2 = b*t1’

E.var = t4
E.code =‘t3 = -c
t4 = b*t3’

E.var = t1
E.code =‘t1 = -c’

E.var = t3
E.code =‘t3 = -c’

E.var = b
E.code =‘’

E.var = c
E.code =‘’

E.var = b
E.code =‘’

E.var = c
E.code =‘’

# Creating 3AC: control statements

- 3AC only supports conditional/unconditional jumps
- Add labels

- Attributes
  - begin – label marks beginning of code
  - after – label marks end of code

- Helper function freshLabel() allocates a new fresh label

# Expressions and assignments

| **production** | **semantic action** |
|---|---|
| S → id := E | { p:= lookup(id.name); if p ≠ null then **emit**(p ‘:=‘ E.var) else error } |
| E → E1 op E2 | { E.var := freshVar(); **emit**(E.var ‘:=‘ E1.var op E2.var) } |
| E → - E1 | { E.var := freshVar(); **emit**(E.var ‘:=‘ ‘uminus’ E1.var) } |
| E → ( E1) | { E.var := E1.var } |
| E → id | { p:= lookup(id.name); if p ≠ null then E.var :=p else error } |

# Boolean Expressions

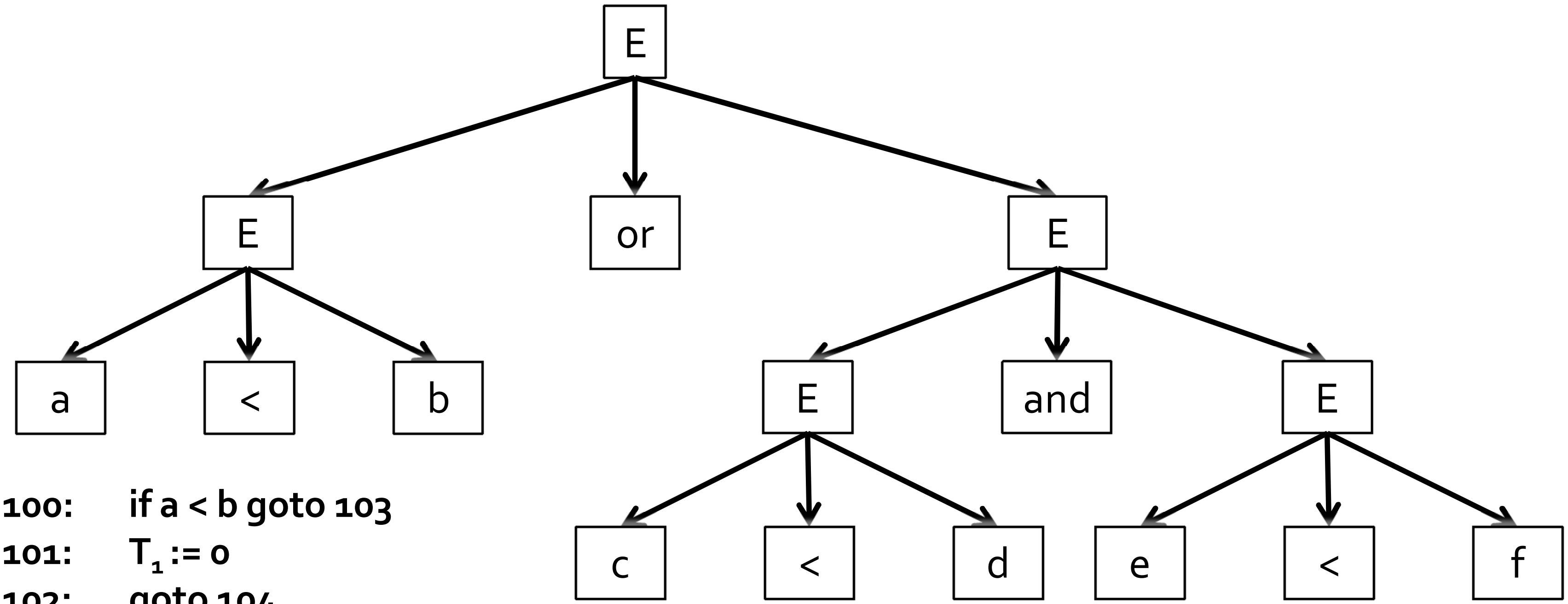
| **production** | **semantic action** |
|---|---|
| E → E1 op E2 | { E.var := freshVar(); **emit**(E.var ‘:=‘ E1.var op E2.var) } |
| E → not E1 | { E.var := freshVar(); **emit**(E.var ‘:=‘ ‘not’ E1.var) } |
| E → ( E1) | { E.var := E1.var } |
| E → true | { E.var := freshVar(); **emit**(E.var ‘:=‘ ‘1’) } |
| E → false | { E.var := freshVar(); **emit**(E.var ‘:=‘ ‘0’) } |

- Represent true as 1, false as 0
- Wasteful representation, creating variables for true/false

# Boolean expressions via jumps

| **production** | **semantic action** |
|---|---|
| E ⟶ id1 op id2 | {<br>E.var := freshVar();<br>**emit**(‘if’ id1.var relop id2.var ‘goto’ nextStmt+2);<br>**emit**( E.var ‘:=‘ ‘0’);<br>**emit**(‘goto ‘ nextStmt + 1);<br>**emit**(E.var ‘:=‘ ‘1’)<br>} |

# Example



```
100:    if a < b goto 103
101:    $T_1$ := 0
102:    goto 104
103:    $T_1$ := 1
```

```
104:    if c < d goto 107
105:    $T_2$ := 0
106:    goto 108
107:    $T_2$ := 1
```

```
108:    if e < f goto 111
109:    $T_3$ := 0
110:    goto 112
111:    $T_3$ := 1
112:    $T_4$ := $T_2$ and $T_3$
113:    $T_5$ := $T_1$ or $T_4$
```

# Short circuit evaluation

- Second argument of a Boolean operator is only evaluated if the first argument does not already determine the outcome

- (x and y) is equivalent to
  if x then y else false;
- (x or y) is equivalent to
  if x then true else y

# example

a < b or (c<d and e<f)

```
100: if a < b goto 103
101: $T_1$ := 0
102: goto 104
103: $T_1$ := 1
104: if c < d goto 107
105: $T_2$ := 0
106: goto 108
107: $T_2$ := 1
108: if e < f goto 111
109: $T_3$ := 0
110: goto 112
111: $T_3$ := 1
112: $T_4$ := $T_2$ and $T_3$
113: $T_5$ := $T_1$ and $T_4$
```

naive

```
100: if a < b goto 105
101: if !(c < d) goto 103
102: if e < f goto 105
103: T := 0
104: goto 106
105: T := 1
106:
```

Short circuit evaluation

# Control Structures

```
S→ if B then S1
  | if B then S1 else S2
  | while B do S1
```

- For every Boolean expression B, we attach two properties
  - falseLabel – target label for a jump when condition B evaluates to false
  - trueLabel – target label for a jump when condition B evaluates to true
- For every statement S we attach a property
  - next – the label of the next code to execute after S
- Challenge
  - Compute falseLabel and trueLabel during code generation

# Control Structures: next

| **production** | **semantic action** |
|---|---|
| P → S | S.next = freshLabel();<br>P.code = S.code \|\| label(S.next) |
| S → S1S2 | S1.next = freshLabel();<br>S2.next = S.next;<br>S.code = S1.code \|\| label(S1.next) \|\| S2.code |

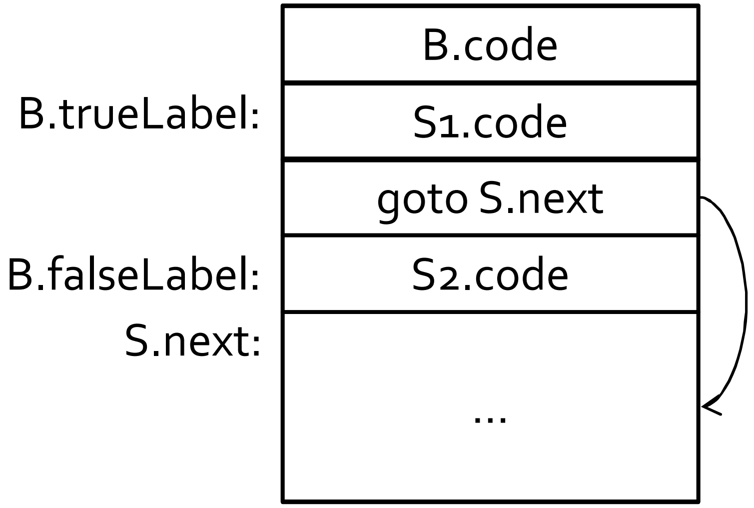
- The label S.next is symbolic, we will only determine its value after we finish deriving S

# Control Structures: conditional

| **production** | **semantic action** |
|---|---|
| S → if B then S1 | B.trueLabel = freshLabel();<br>B.falseLabel = S.next;<br>S1.next = S.next;<br>S.code = B.code \|\| gen (B.trueLabel ‘:’) \|\| S1.code |

# Control Structures: conditional

| **production** | **semantic action** |
|---|---|
| S → if B then S1 else S2 | B.trueLabel = freshLabel();<br>B.falseLabel = freshLabel();<br>S1.next = S.next;<br>S2.next = S.next;<br>S.code =<br>B.code \|\| gen(B.trueLabel ':') \|\| S1.code \|\| gen('goto' S.next)<br>\|\| gen(B.falseLabel ':') \|\| S2.code |



# Boolean expressions

| **production** | **semantic action** |
|---|---|
| B → B1 or B2 | B1.trueLabel = B.trueLabel;<br>B1.falseLabel = freshLabel();<br>B2.trueLabel = B.trueLabel;<br>B2.falseLabel = B.falseLabel;<br>B.code = B1.code \|\| gen (B1.falseLabel ‘:’) \|\| B2.code |
| B → B1 and B2 | B1.trueLabel = freshLabel();<br>B1.falseLabel = B.falseLabel;<br>B2.trueLabel = B.trueLabel;<br>B2.falseLabel = B.falseLabel;<br>B.code = B1.code \|\| gen (B1.trueLabel ‘:’) \|\| B2.code |
| B → not B1 | B1.trueLabel = B.falseLabel;<br>B1.falseLabel = B.trueLabel;<br>B.code = B1.code; |
| B → (B1) | B1.trueLabel = B.trueLabel; B1.falseLabel = B.falseLabel; B.code = B1.code; |
| B → id1 relop id2 | B.code=gen (‘if’ id1.var relop id2.var ‘goto’ B.trueLabel)\|\|gen(‘goto’ B.falseLabel); |
| B → true | B.code = gen(‘goto’ B.trueLabel) |
| B → false | B.code = gen(‘goto’ B.falseLabel); |

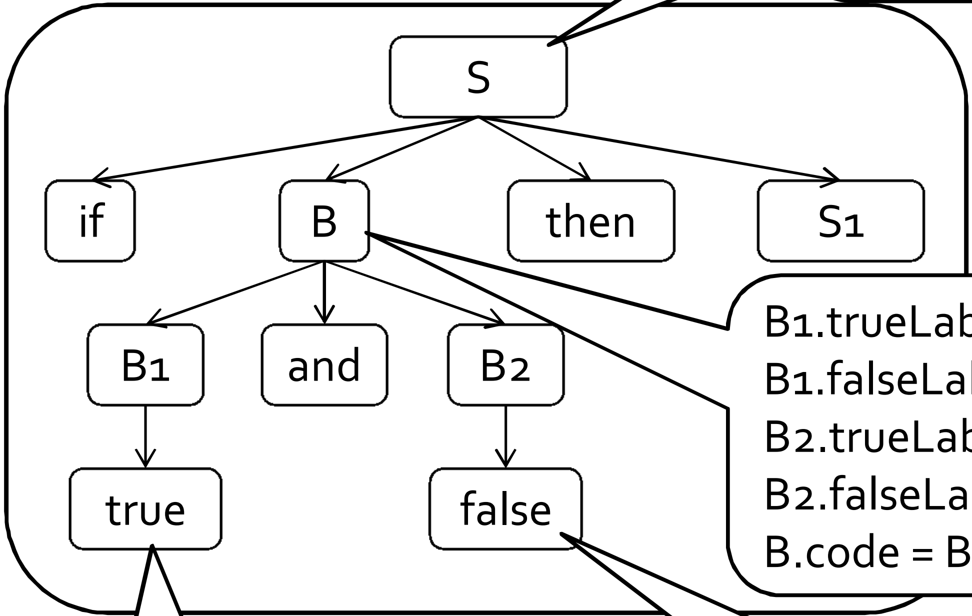
# Boolean expressions

| **production** | **semantic action** |
|---|---|
| B → B1 or B2 | B1.trueLabel = B.trueLabel;<br>B1.falseLabel = freshLabel();<br>B2.trueLabel = B.trueLabel;<br>B.falseLabel = B.falseLabel;<br>B.code = B1.code \|\| gen (B1.falseLabel ':') \|\| B2.code |

- How can we determine the address of B1.falseLabel?

- Only possible after we know the code of B1 and all the code preceding B1

# Example



B.trueLabel = freshLabel();
B.falseLabel = S.next;
S1.next = S.next;
S.code = B.code || gen (B.trueLabel ‘:’) || S1.code

B1.trueLabel = freshLabel();
B1.falseLabel = B.falseLabel;
B2.trueLabel = B.trueLabel;
B2.falseLabel = B.falseLabel;
B.code = B1.code || gen (B1.trueLabel ‘:’) || B2.code

B.code = gen(‘goto’ B.falseLabel)

B.code = gen(‘goto’ B.trueLabel)

# Computing addresses for labels

- We used symbolic labels
- We need to compute their addresses
- We can compute addresses for the labels but it would require an additional pass on the AST

- Can we do it in a single pass?

# Backpatching

- Goal: generate code in a single pass

- Generate code as we did before, but manage labels differently
- Keep labels symbolic until values are known, and then back-patch them

- New synthesized attributes for B
  - B.truelist – list of jump instructions that eventually get the label where B goes when B is true.
  - B.falselist – list of jump instructions that eventually get the label where B goes when B is false.

# Backpatching

- Previous approach does not guarantee a single pass
    - The attribute grammar we had before is not S-attributed (e.g., next), and is not L-attributed.
- For every label, maintain a list of instructions that jump to this label
- When the address of the label is known, go over the list and update the address of the label

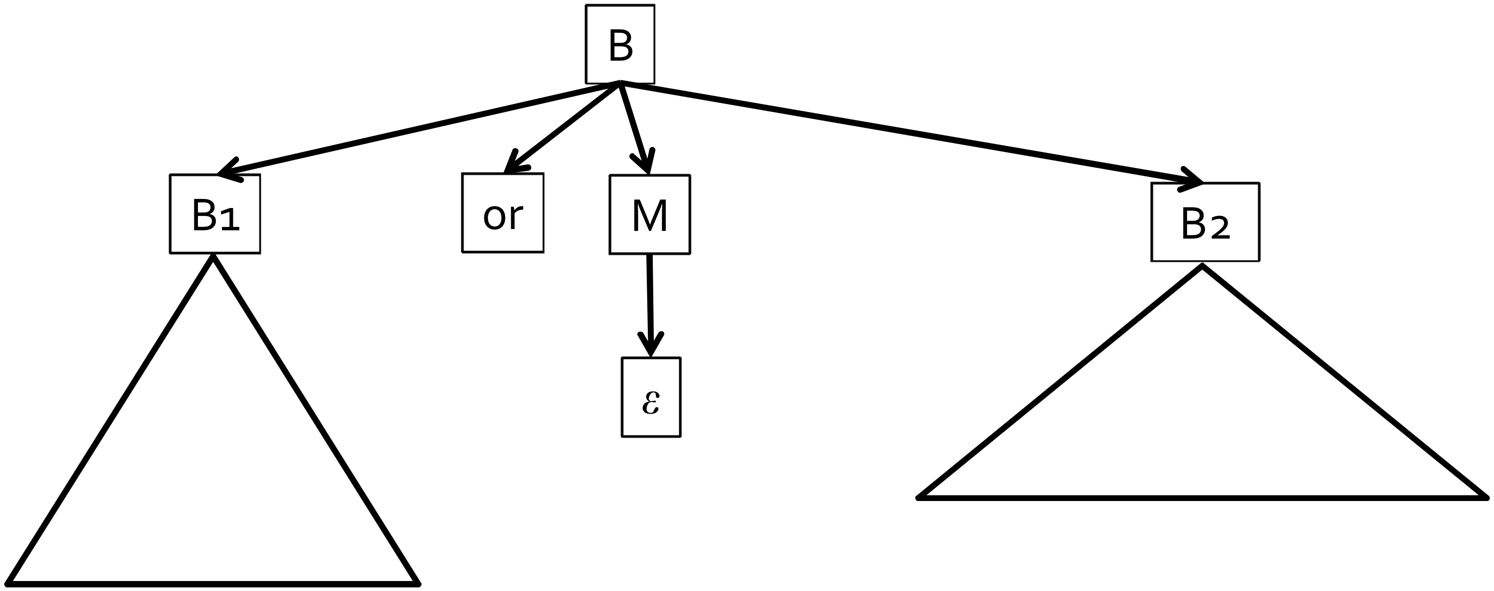
# Backpatching

- makelist(addr) – create a list of instructions containing addr
- merge(p1,p2) – concatenate the lists pointed to by p1 and p2, returns a pointer to the new list
- backpatch(p,addr) – inserts i as the target label for each of the instructions in the list pointed to by p

# Backpatching Boolean expressions

| **production** | **semantic action** |
|---|---|
| B → B1 or **M** B2 | backpatch(B1.falseList,M.instr);<br>B.trueList = merge(B1.trueList,B2.trueList);<br>B.falseList = B2.falseList; |
| B → B1 and **M** B2 | backpatch(B1.trueList,M.instr);<br>B.trueList = B2.trueList;<br>B.falseList = merge(B1.falseList,B2.falseList); |
| B → not B1 | B.trueList = B1.falseList;<br>B.falseList = B1.trueList; |
| B → (B1) | B.trueList = B1.trueList;<br>B.falseList = B1.falseList; |
| B → id1 relop id2 | B.trueList = makeList(nextInstr);<br>B.falseList = makeList(nextInstr+1);<br>emit (‘if’ id1.var relop id2.var ‘goto _’) \|\| emit(‘goto _’); |
| B → true | B.trueList = makeList(nextInstr);<br>emit (‘goto _’); |
| B → false | B.falseList = makeList(nextInstr);<br>emit (‘goto _’); |
| **M** → ε | M.instr = nextinstr; |

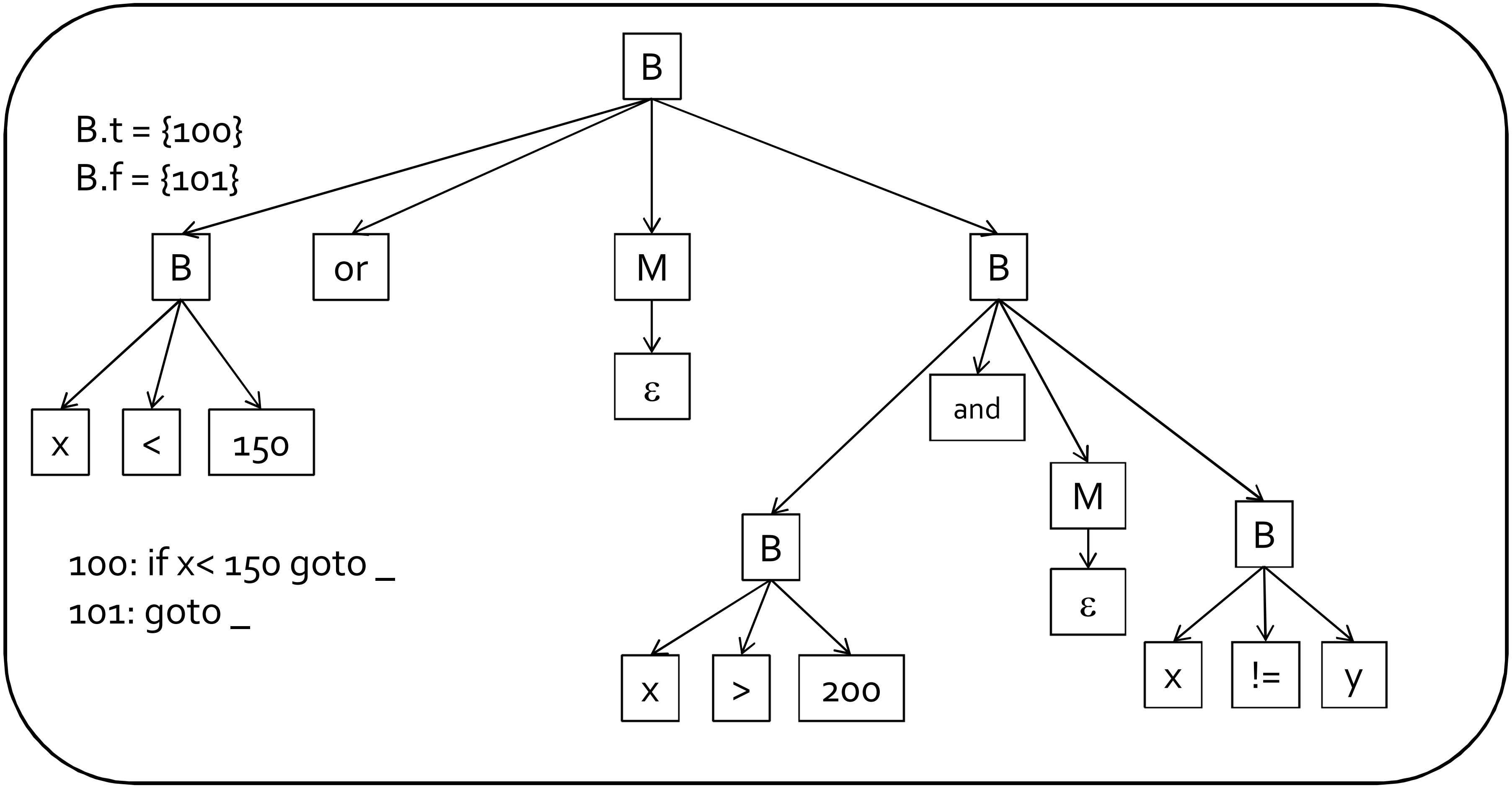
# Marker



- { M.instr = nextinstr;}
- Use M to obtain the address just before B2 code starts being generated

# Example

X < 150 or x > 200 and x != y



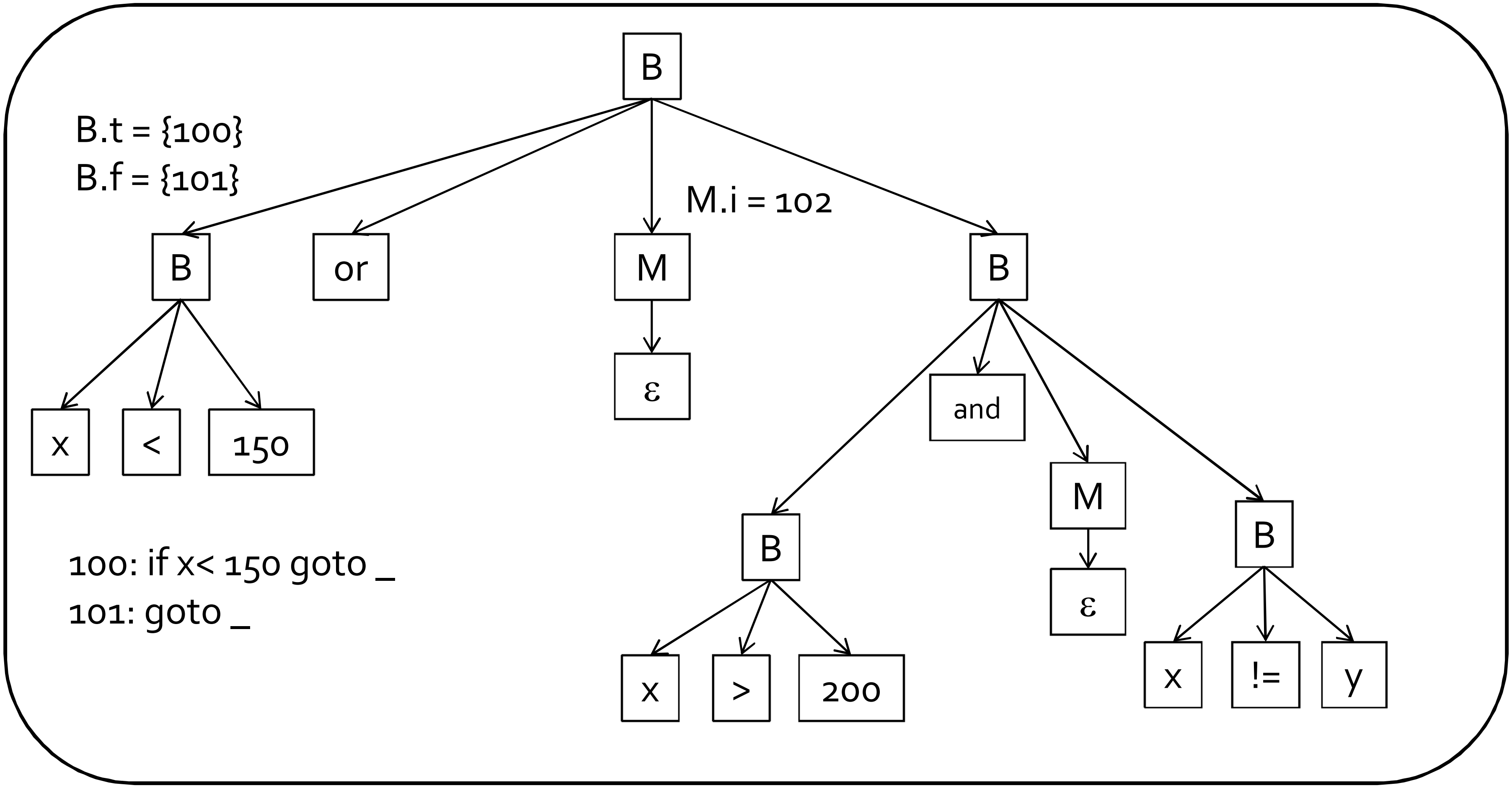
B.t = {100}
B.f = {101}

100: if x< 150 goto _
101: goto _

| | |
|---|---|
| **B → id1 relop id2** | **B.trueList = makeList(nextInstr);**<br>**B.falseList = makeList(nextInstr+1);**<br>**emit (‘if’ id1.var relop id2.var ‘goto _’) \|\| emit(‘goto _’);** |

# Example

X < 150 or x > 200 and x != y



B.t = {100}
B.f = {101}

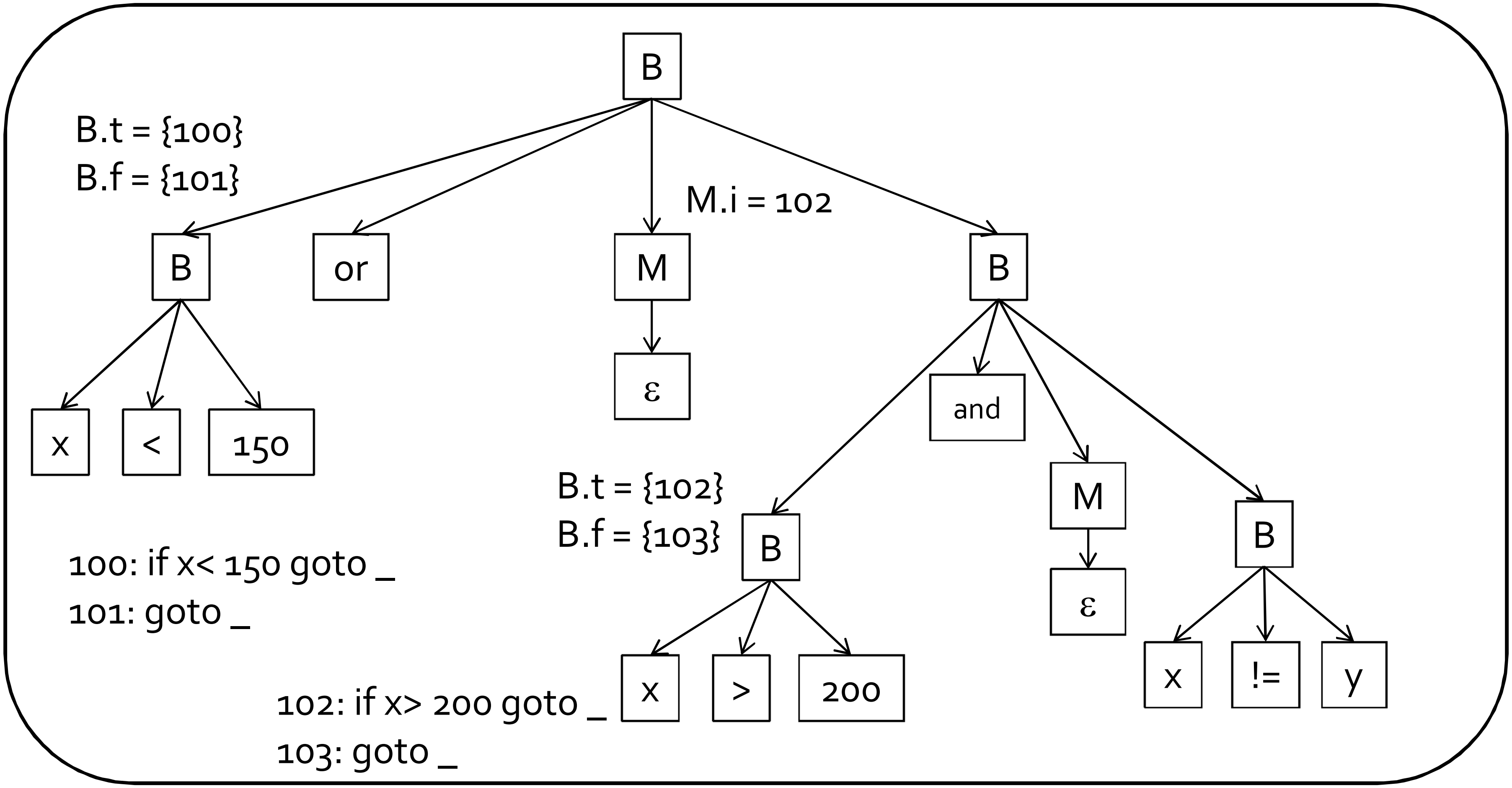
M.i = 102

100: if x< 150 goto _
101: goto _

| **M → ε** | **M.instr = nextinstr;** |
|---|---|

# Example

X < 150 or x > 200 and x != y



B.t = {100}
B.f = {101}

M.i = 102

B.t = {102}
B.f = {103}

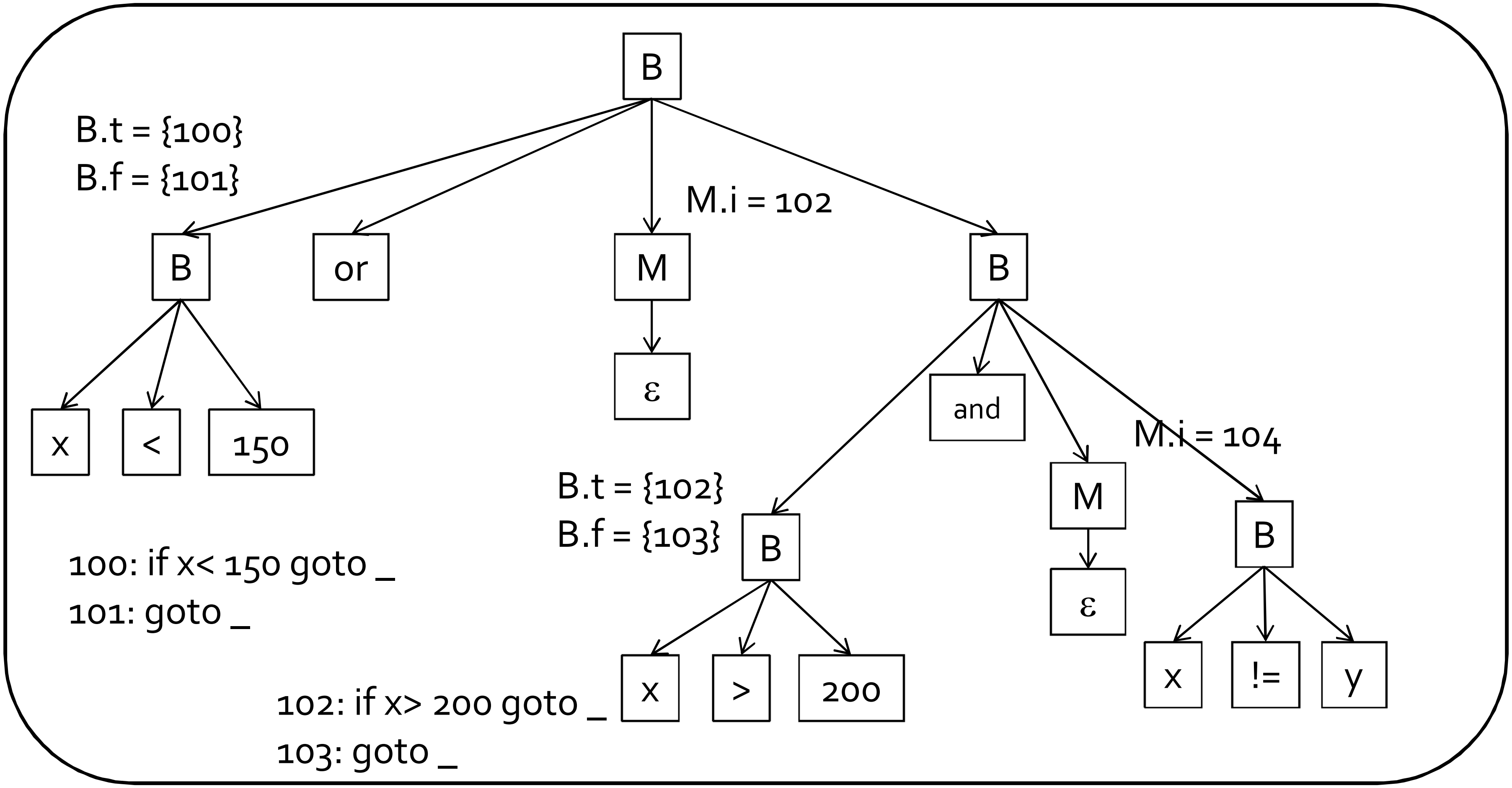
100: if x< 150 goto _
101: goto _

102: if x> 200 goto _
103: goto _

| | |
|---|---|
| **B → id1 relop id2** | **B.trueList = makeList(nextInstr);**<br>**B.falseList = makeList(nextInstr+1);**<br>**emit (‘if’ id1.var relop id2.var ‘goto _’) \|\| emit(‘goto _’);** |

# Example

X < 150 or x > 200 and x != y



B.t = {100}
B.f = {101}

M.i = 102

M.i = 104

B.t = {102}
B.f = {103}

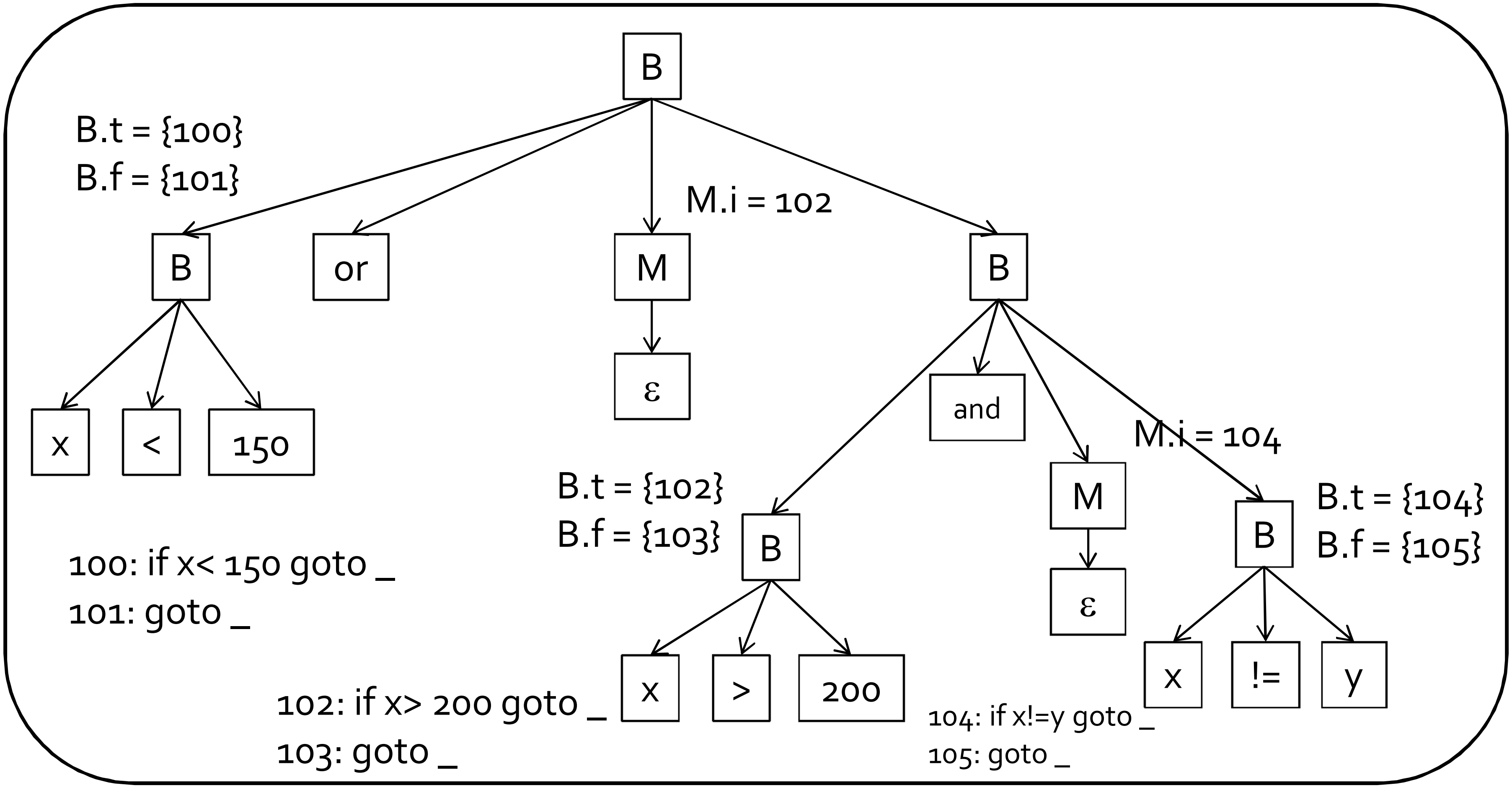
100: if x< 150 goto _
101: goto _

102: if x> 200 goto _
103: goto _

| | |
|---|---|
| **M → ε** | **M.instr = nextinstr;** |

# Example

X < 150 or x > 200 and x != y



100: if x< 150 goto _
101: goto _

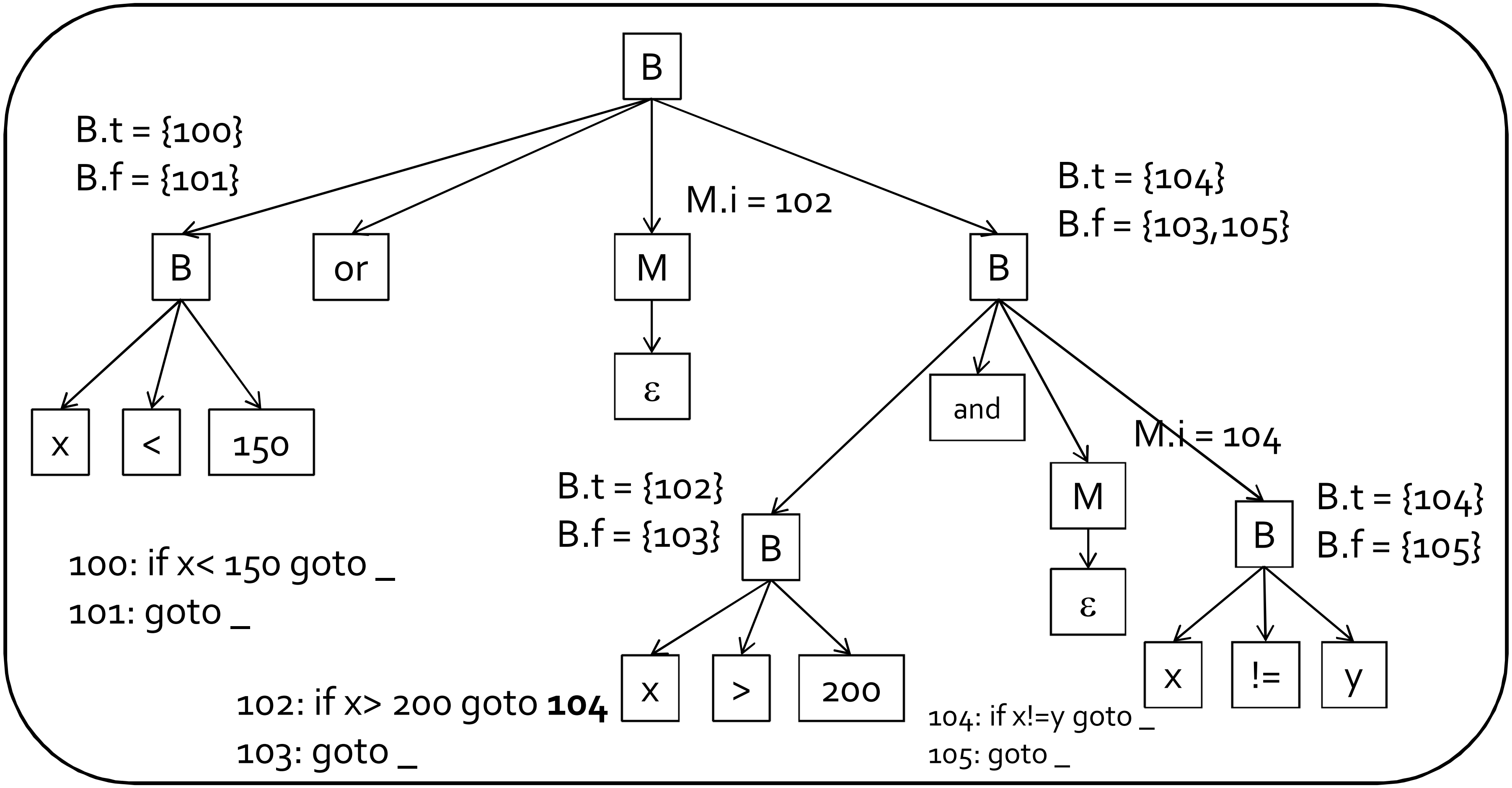
102: if x> 200 goto _
103: goto _

104: if x!=y goto _
105: goto _

| | |
|---|---|
| **B → id1 relop id2** | **B.trueList = makeList(nextInstr);**<br>**B.falseList = makeList(nextInstr+1);**<br>**emit ('if' id1.var relop id2.var 'goto _') \|\| emit('goto _');** |

# Example

X < 150 or x > 200 and x != y



B.t = {100}
B.f = {101}

M.i = 102

B.t = {104}
B.f = {103,105}

M.i = 104

B.t = {102}
B.f = {103}

B.t = {104}
B.f = {105}

100: if x< 150 goto _
101: goto _

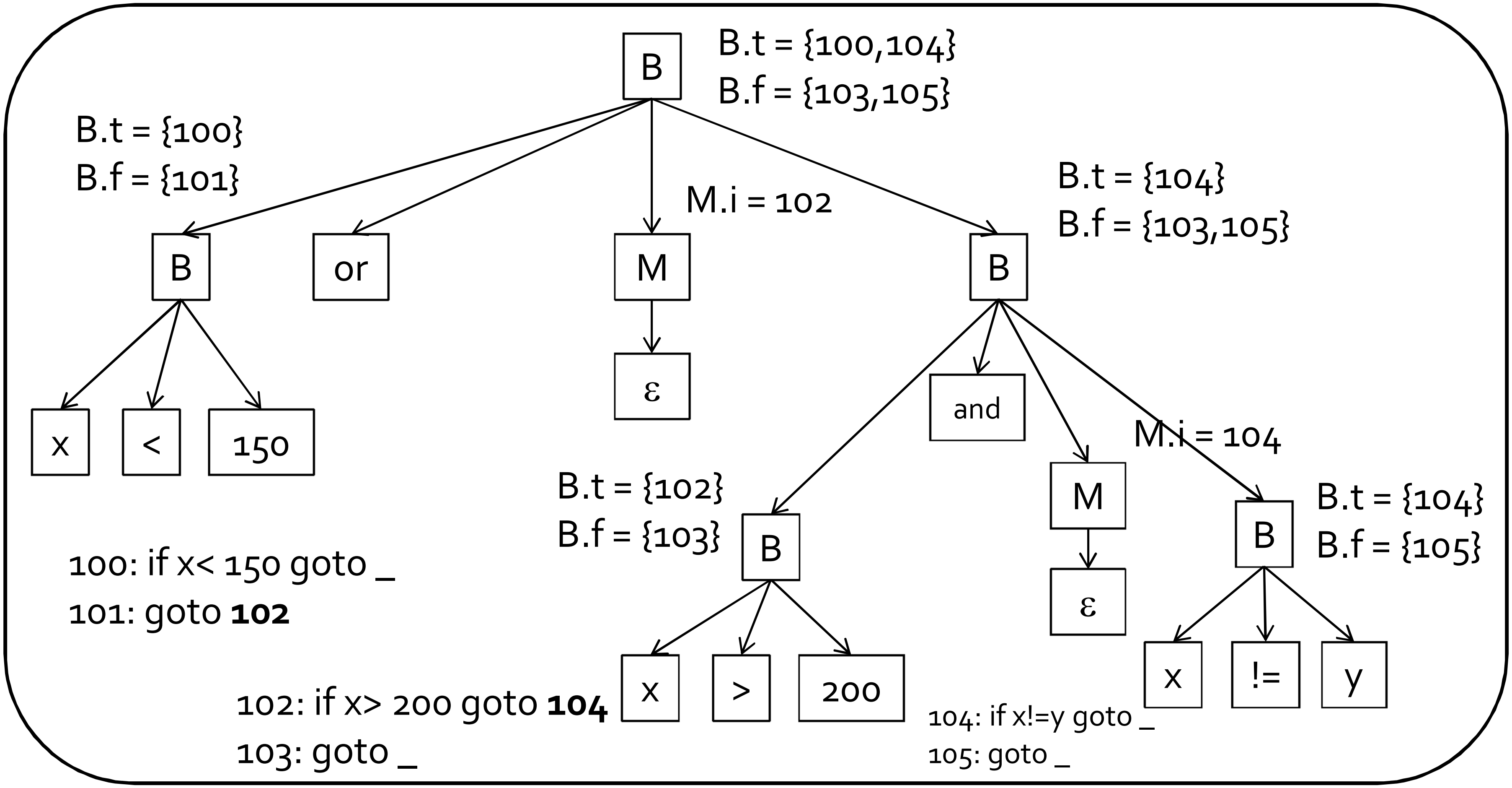
102: if x> 200 goto **104**
103: goto _

104: if x!=y goto _
105: goto _

| | |
|---|---|
| **B → B1 and M B2** | **backpatch(B1.trueList,M.instr);**<br>**B.trueList = B2.trueList;**<br>**B.falseList = merge(B1.falseList,B2.falseList);** |

# Example

X < 150 or x > 200 and x != y



100: if x< 150 goto _
101: goto **102**

102: if x> 200 goto **104**
103: goto _

104: if x!=y goto _
105: goto _

| | |
|---|---|
| **B → B1 or M B2** | **backpatch(B1.falseList,M.instr);**<br>**B.trueList = merge(B1.trueList,B2.trueList);**<br>**B.falseList = B2.falseList;** |

# Example

```
100: if x<150 goto _
101: goto _
102: if x>200 goto _
103: goto _
104: if x!=y goto _
105: goto _
```

Before backpatching

```
100: if x<150 goto _
101: goto _
102: if x>200 goto 104
103: goto _
104: if x!=y goto _
105: goto _
```

After backpatching
by the production
B → B1 and M B2

```
100: if x<150 goto _
101: goto 102
102: if x>200 goto 104
103: goto _
104: if x!=y goto _
105: goto _
```

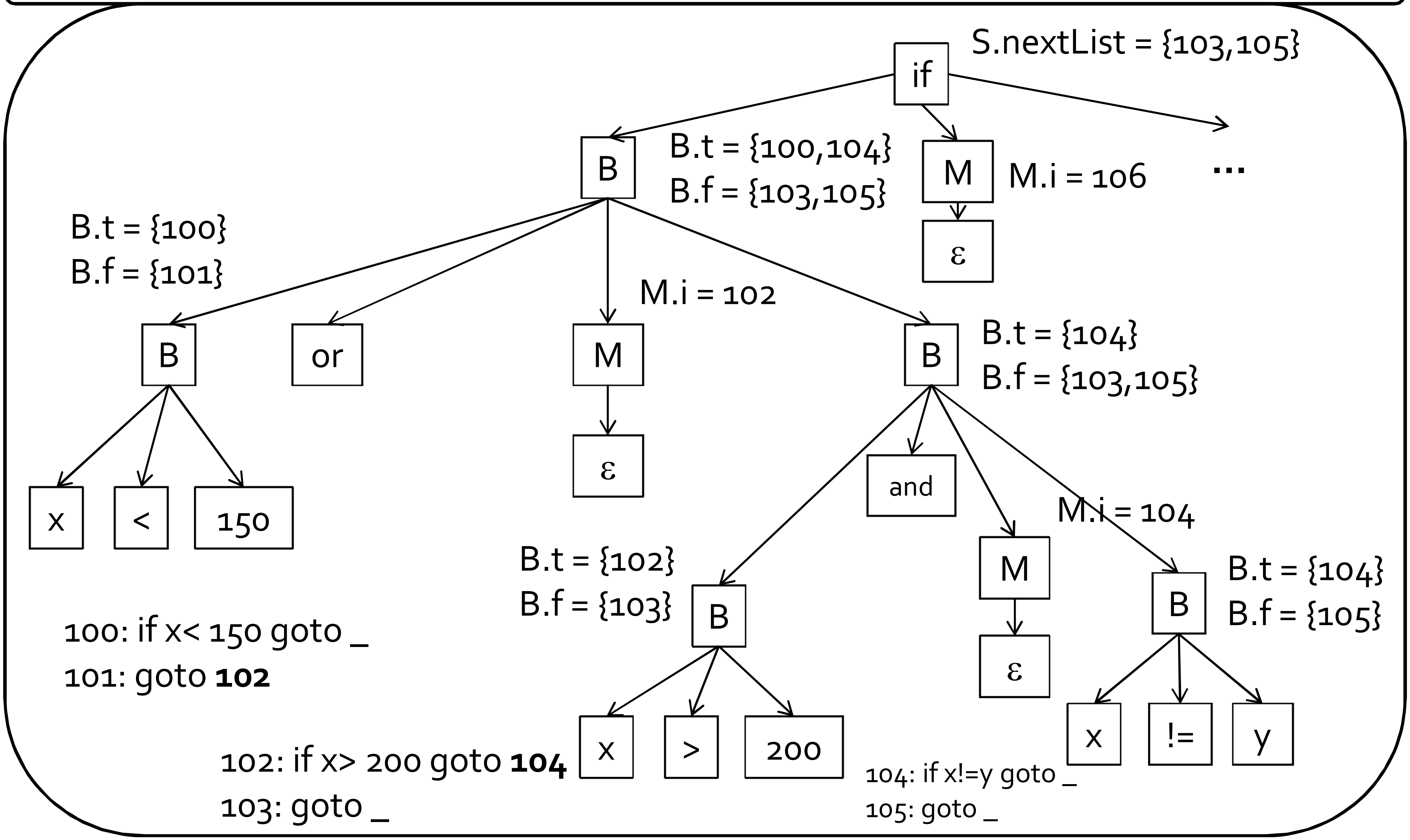
After backpatching
by the production
B → B1 or M B2

# Backpatching for statements

| **production** | **semantic action** |
|---|---|
| S → if (B) M S1 | backpatch(B.trueList,M.instr);<br>S.nextList = merge(B.falseList,S1.nextList); |
| S → if (B) M1 S1 N else M2 S2 | backpatch(B.trueList,M1.instr);<br>backpatch(B.falseList,M2.instr);<br>temp = merge(S1.nextList,N.nextList);<br>S.nextList = merge(temp,S2.nextList); |
| S → while M1 (B) M2 S1 | backpatch(S1.nextList,M1.instr);<br>backpatch(B.trueList,M2.instr);<br>S.nextList = B.falseList;<br>emit(‘goto’ M1.instr); |
| S → { L } | S.nextList = L.nextList; |
| S → A | S.nextList = null; |
| **M** → ε | M.instr = nextinstr; |
| **N** → ε | N.nextList = makeList(nextInstr); emit(‘goto _’); |
| L → L1 M S | backpatch(L1.nextList,M.instr);<br>L.nextList = S.nextList; |
| L → S | L.nextList = S.nextList |

# Example

if (x < 150 or x > 200 and x != y) y=200;



S.nextList = {103,105}

B.t = {100,104}
B.f = {103,105}

M.i = 106

B.t = {100}
B.f = {101}

M.i = 102

B.t = {104}
B.f = {103,105}

M.i = 104

B.t = {102}
B.f = {103}

B.t = {104}
B.f = {105}

100: if x< 150 goto _
101: goto **102**

102: if x> 200 goto **104**
103: goto _

104: if x!=y goto _
105: goto _

| | |
|---|---|
| **S → if (B) M S1** | **backpatch(B.trueList,M.instr);**<br>**S.nextList = merge(B.falseList,S1.nextList);** |

# Example

```
100: if x<150 goto _
101: goto 102
102: if x>200 goto 104
103: goto _
104: if x!=y goto _
105: goto _
106: y = 200
```

After backpatching
by the production
B → B1 or M B2

```
100: if x<150 goto 106
101: goto 102
102: if x>200 goto 104
103: goto _
104: if x!=y goto 106
105: goto _
106: y = 200
```

After backpatching
by the production
S → if (B) M S1

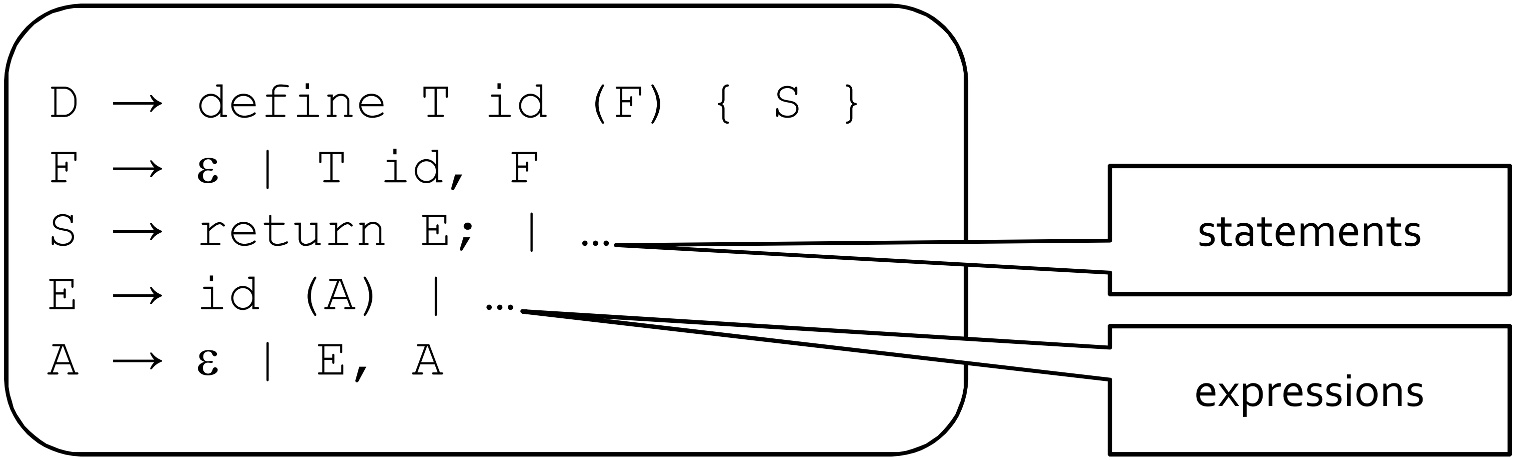
# Procedures

```
n = f(a[i]);
```

```
t1 = i * 4
t2 = a[t1] // could have expanded this as well
param t2
t3 = call f, 1
n = t3
```

- we will see handling of procedure calls in much more detail later

# Procedures



```
D → define T id (F) { S }
F → ε | T id, F
S → return E; | …
E → id (A) | …
A → ε | E, A
```

- type checking
  - function type: return type, type of formal parameters
  - within an expression function treated like any other operator
- symbol table
  - parameter names

# Summary

- pick an intermediate representation
- translate expressions
- use a symbol table to implement declarations
- generate jumping code for boolean expressions
  - value of the expression is implicit in the control location
- backpatching
  - a technique for generating code for boolean expressions and statements in one pass
  - idea: maintain lists of incomplete jumps, where all jumps in a list have the same target. When the target becomes known, all instructions on its list are “filled in”.

# Coming up next…

- Activation Records

# The End