Lecture 08 – Intermediate Representation

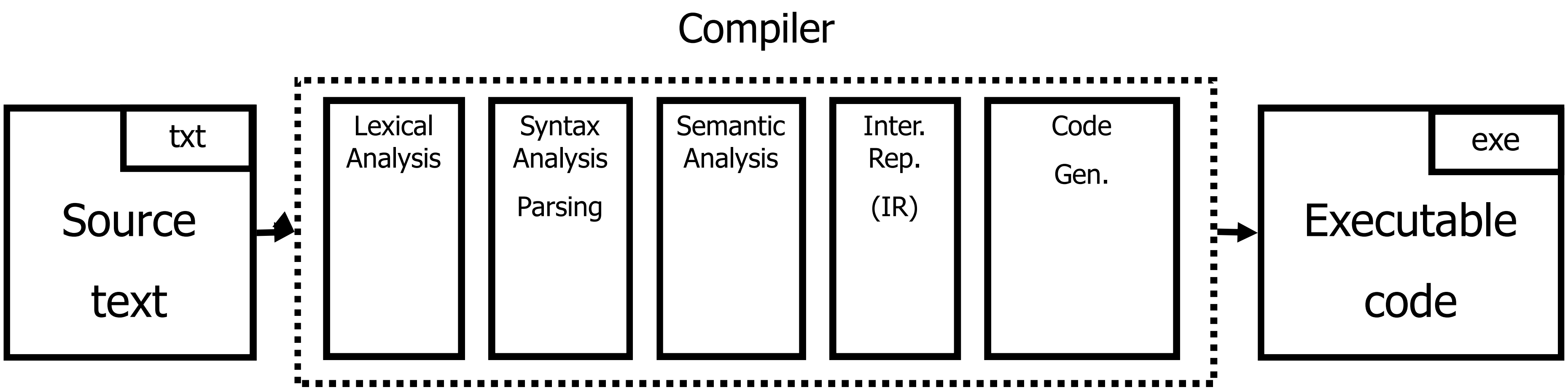
THEORY OF COMPILATION

Eran Yahav

www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec08.pptx

Reference: Dragon 6.2,6.3,6.4,6.6

You are here


Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
Inter.
Rep.
(IR)
Code
Gen.
exe
Executable
code

Last Week: Attribute Grammars

- Adding attributes + actions to a grammar
- Evaluating attributes
  - Build AST
  - Build dependency graph
  - Evaluation based on topological order
  - (works as long as there are no cycles)
- L-attributes, S-attributed grammars
  - Pre-determined evaluation order
  - Can be integrated into parsing

Last Week: Three Address Code (3AC)

- Every instruction operates on three addresses
  - result = operand1 operator operand2
- Close to low-level operations in the machine language
  - Operator is a basic operation
- Statements in the source language may be mapped to multiple instructions in three address code
- can be represented as "quads" (result, operand1,operator,operand2)

Last Week: Creating 3AC

- Assume bottom up parser
  - Covers a wider range of grammars
  - LALR sufficient to cover most programming languages

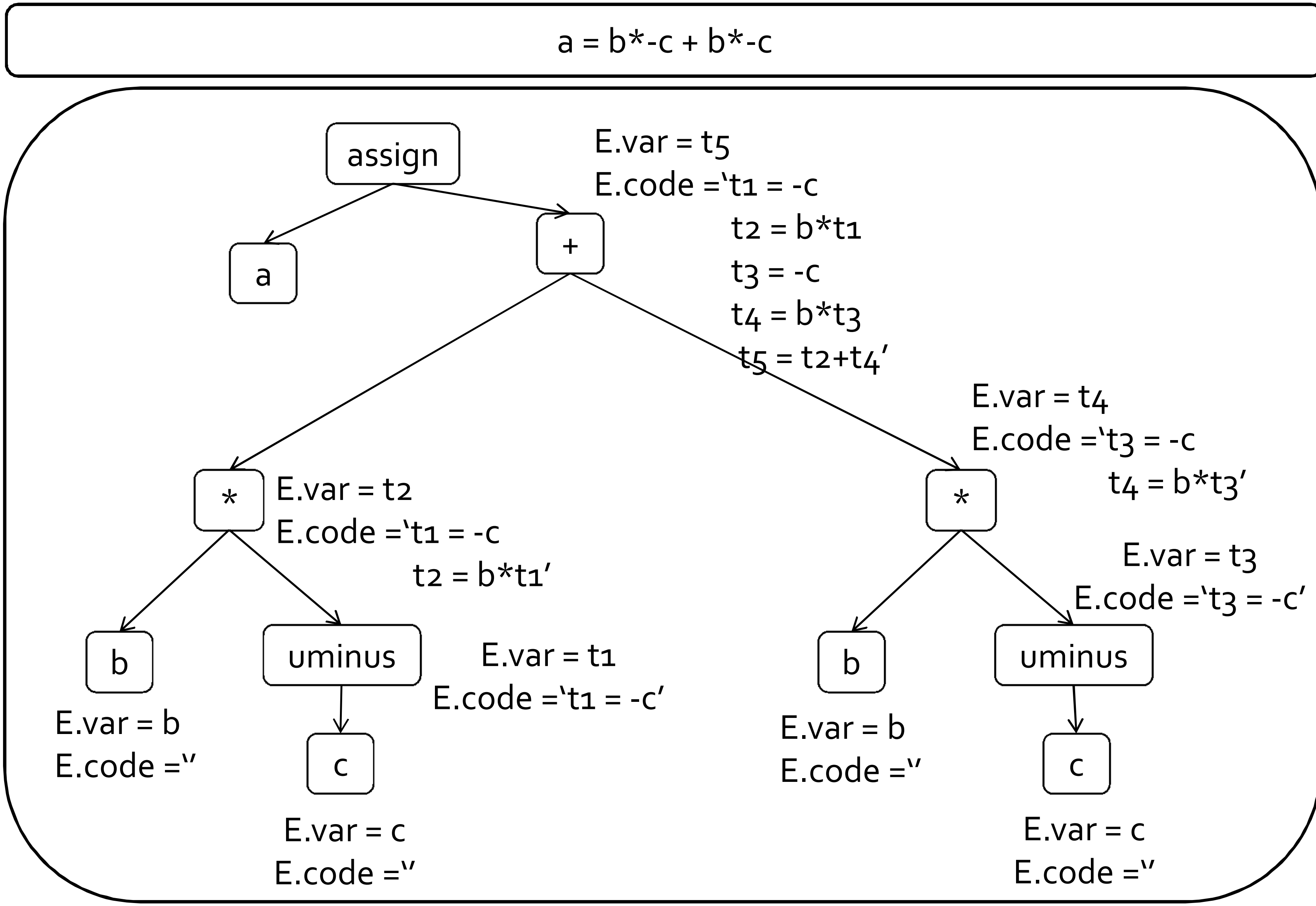
- Creating 3AC via syntax directed translation
- Attributes
  - code – code generated for a nonterminal
  - var – name of variable that stores result of nonterminal
- freshVar() – helper function that returns the name of a fresh variable

Creating 3AC: expressions

production	**semantic rule**
S → id := E	S.code := E. code || gen(id.var ':=' E.var)
E → E_1 + E_2	E.var := freshVar(); E.code = E_1.code || E_2.code || gen(E.var ':=' E_1.var '+' E_2.var)
E → E_1 * E_2	E.var := freshVar(); E.code = E_1.code || E_2.code || gen(E.var ':=' E_1.var '*' E_2.var)
E → - E_1	E.var := freshVar(); E.code = E_1.code || gen(E.var ':=' 'uminu' E_1.var)
E → (E_1)	E.var := E_1.var E.code = '(' || E_1.code || ')'
E → id	E.var := id.var; E.code = ''

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

example


a = b*-c + b*-c
assign
a
+
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'
b
E.var = b
E.code =''
uminus
E.var = t1
E.code ='t1 = -c'
c
E.var = c
E.code =''
b
E.var = b
E.code =''
uminus
E.var = t3
E.code ='t3 = -c'
c
E.var = c
E.code =''

Three address code: example

```
int main(void) {
  int i;
  int b[10];
  for (i = 0; i < 10; ++i)
    b[i] = i*i;
}
```

```
i := 0                        ; assignment
L1: if i >= 10 goto L2        ; conditional jump
    t0 := i*i
    t1 := &b                  ; address-of operation
    t2 := t1 + i              ; t2 holds the address of b[i]
    *t2 := t0                 ; store through pointer
    i := i + 1
    goto L1
L2:
```

(example source: wikipedia)

Static Single-Assignment Form (SSA)

- Every assignment writes to a distinct variable
- Every variable is only assigned once

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

SSA

```
if (f)
  x = 42;
else
  x = 73;
y = x * a;
```

```
if (f)
  x1 = 42;
else
  x2 = 73;
x3 = ϕ(x1,x2);
y = x3 * a;
```

- ϕ (phi) function combines different definitions
- ϕ returns the value of x1 if control passes through the true branch and the value of x2 if it passed through the false branch

SSA why should we care?

```
x = 42
x = 73
y = x
```

```
x1 = 42
x2 = 73
y = x2
```

- makes it easy to apply many optimizations
  - constant propagation, dead code elimination…

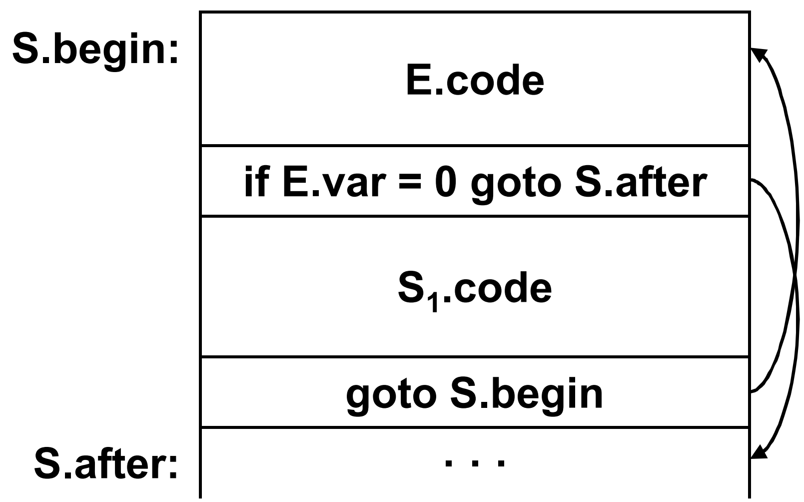
Creating 3AC: control statements

- 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

Creating 3AC: control statements

S → while E do S1



production	semantic rule
S → while E do S1	S.begin := freshLabel(); S.after = freshLabel(); S.code := gen(S.begin ':') || E.code || gen('if' E.var '=' '0' 'goto' S.after) || S1.code || gen('goto' S.begin) || gen(S.after ':')

Allocating Memory

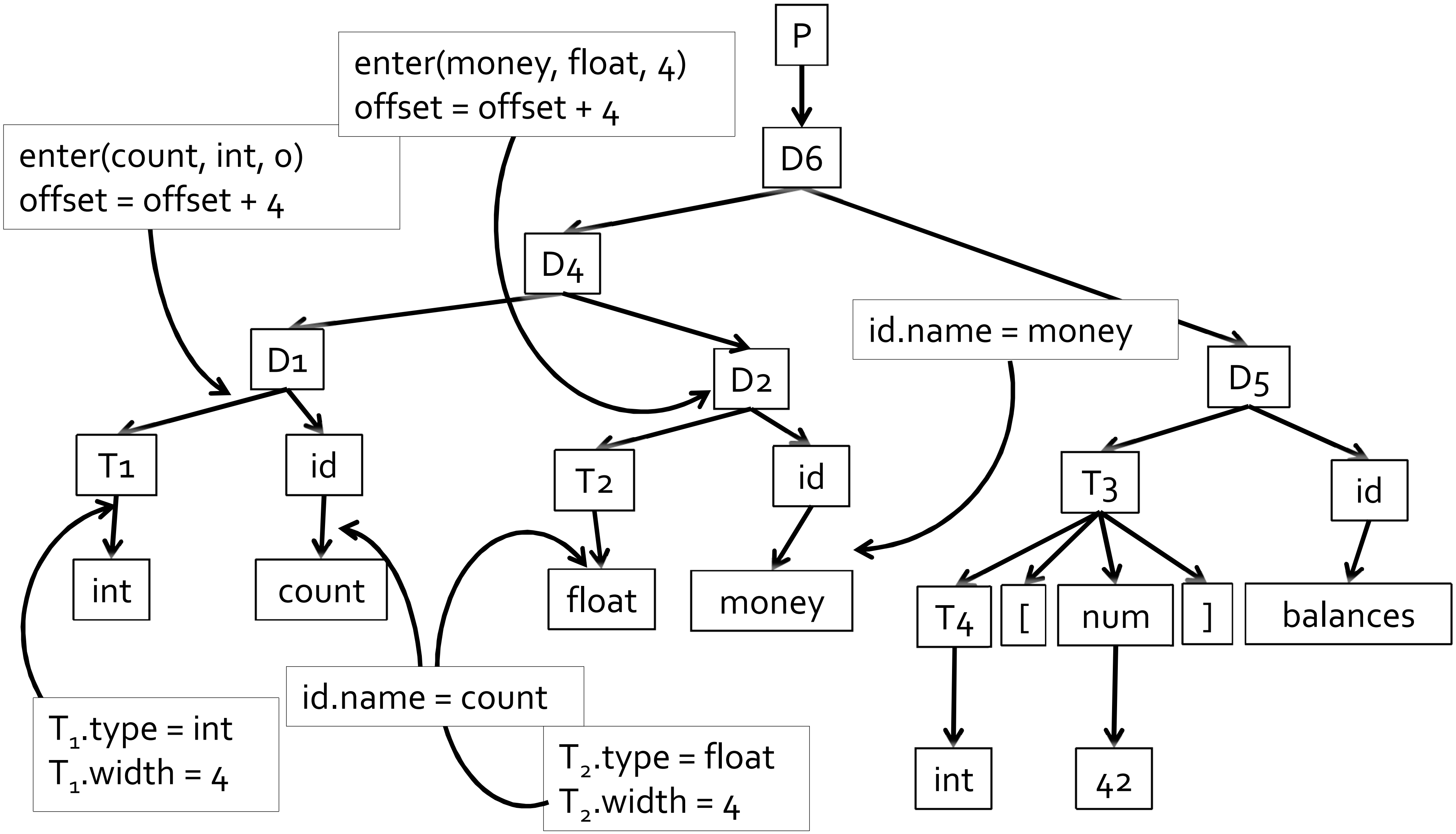
- Type checking helped us guarantee correctness
- Also tells us
  - How much memory allocate on the heap/stack for variables
  - Where to find variables (based on offsets)
  - Compute address of an element inside array (size of stride based on type of element)

Allocating Memory

- Global variable "offset" with memory allocated so far

production	semantic rule
P → D	{ offset := 0}
D → D D	
D → T id;	{ enter(id.name, T.type, offset); offset += T.width }
T → integer	{ T.type := int; T.width = 4 }
T → float	{ T.type := float; T.width = 8}
T → T1[num]	{ T.type = array (num.val,T1.Type); T.width = num.val * T1.width; }
T → *T1	{ T.type := pointer(T1.type); T.width = 4}

Allocating Memory



Adjusting to bottom-up

production	semantic rule
P → **M** D	
M → ε	**{ offset := 0}**
D → D D	
D → T id;	{ enter(id.name, T.type, offset); offset += T.width }
T → integer	{ T.type := int; T.width = 4 }
T → float	{ T.type := float; T.width = 8}
T → T1[num]	{ T.type = array (num.val,T1.Type); T.width = num.val * T1.width; }
T → *T1	{ T.type := pointer(T1.type); T.width = 4}

Generating IR code

- Option 1
  accumulate code in AST attributes

- Option 2
  emit IR code to a file during compilation
  - If for every production the code of the left-hand-side is constructed from a concatenation of the code of the RHS in some fixed order

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') }

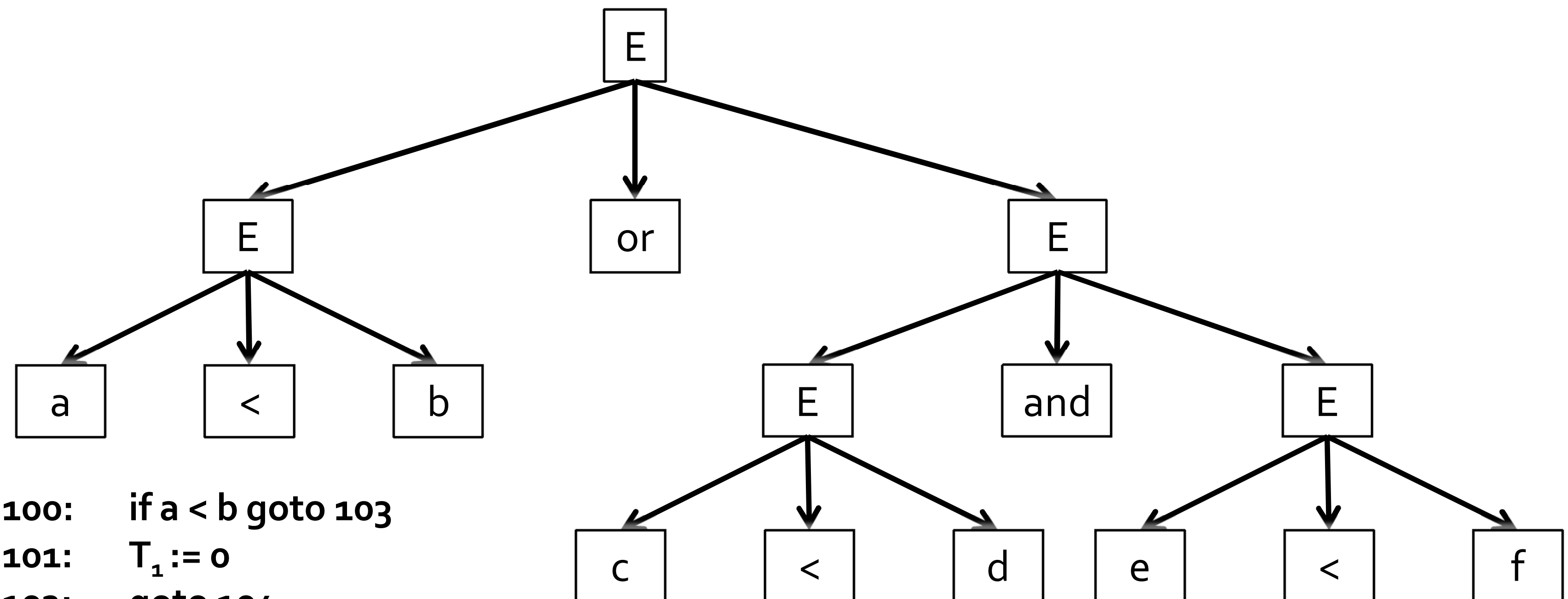
- Represent true as 1, false as 0
- Wasteful representation, creating variables for true/false

Boolean expressions via jumps

production	semantic action

E ⟶ id1 op id2	{
	E.var := freshVar();
	emit('if' id1.var relop id2.var 'goto' nextStmt+2);
	emit(E.var ':=' '0');
	emit('goto ' nextStmt + 1);
	emit(E.var ':=' '1')
	}

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

More examples

if(x ≠ null && x.val = 42)

```
int denom = 0;
if (denom && nom/denom) {
        oops_i_just_divided_by_zero();
}
```

```
int x=0;
if (++x>0 && x++) {
        hmmm();
}
```

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 we attach a property
  - S.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
$P \rightarrow S$	S.next = freshLabel(); P.code = S.code || label(S.next)
$S \rightarrow S_1 S_2$	S_1.next = freshLabel(); S_2.next = S.next; S.code = S_1.code || label(S_1.next) || S_2.code

- Is S.next inherited or synthesized?
- Is S.code inherited or synthesized?

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

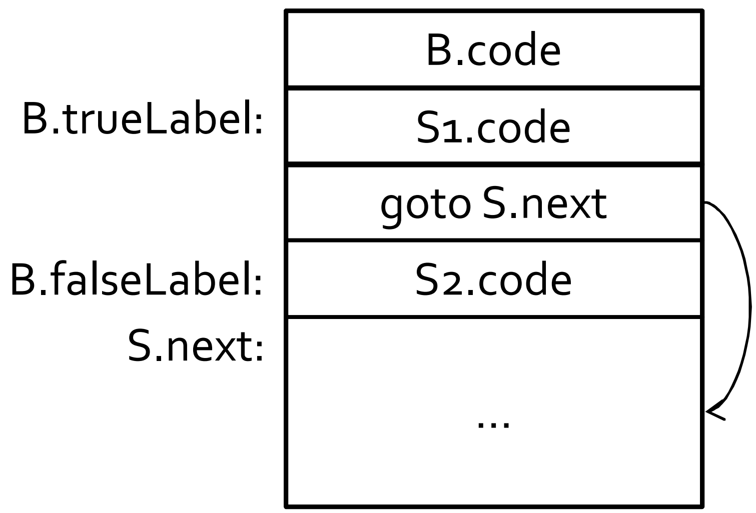
- Are S1.next, B.falseLabel inherited or synthesized?
- Is S.code inherited or synthesized?

Control Structures: conditional

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

- B.trueLabel and B.falseLabel considered inherited

Control Structures: conditional



B.trueLabel = freshLabel();
B.falseLabel = freshLabel();
S1.next = S.next;
S2.next = S.next;
S.code =
B.code || gen(B.trueLabel ‘:’) || S1.code || gen(‘goto’ S.next)
|| gen(B.falseLabel ‘:’) || S2.code

Boolean expressions

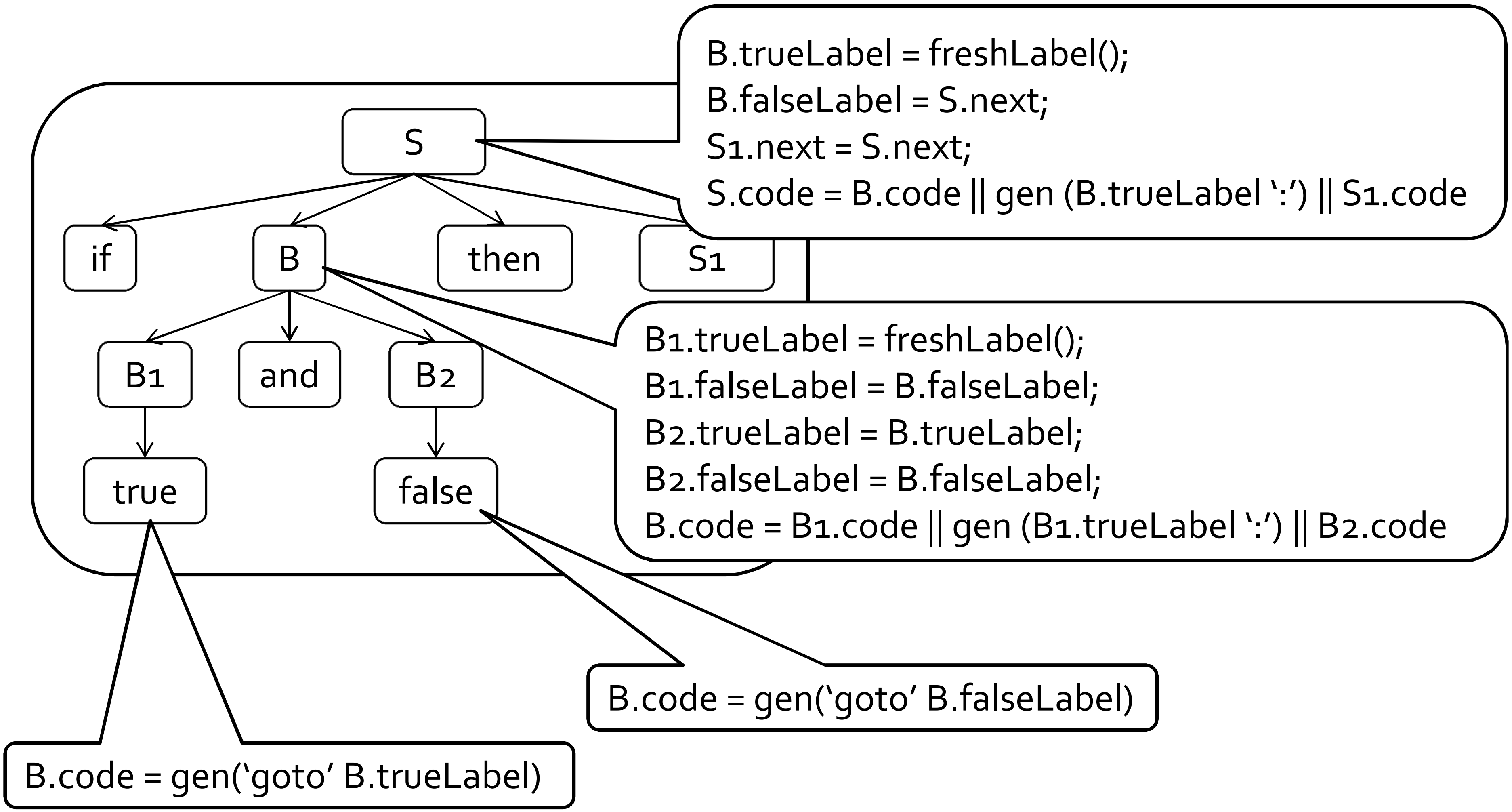
production	semantic action
B ⟶ B1 or B2	B1.trueLabel = B.trueLabel; B1.falseLabel = freshLabel(); B2.trueLabel = B.trueLabel; B2.falseLabel = B.falseLabel; B.code = B1.code || gen (B1.falseLabel ':') || B2.code
B ⟶ B1 and B2	B1.trueLabel = freshLabel(); B1.falseLabel = B.falseLabel; B2.trueLabel = B.trueLabel; B2.falseLabel = B.falseLabel; B.code = B1.code || gen (B1.trueLabel ':') || B2.code
B ⟶ not B1	B1.trueLabel = B.falseLabel; B1.falseLabel = B.trueLabel; 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; B1.falseLabel = freshLabel(); B2.trueLabel = B.trueLabel; B.falseLabel = B.falseLabel; 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
S
if
B
then
S1
B1.trueLabel = freshLabel();
B1.falseLabel = B.falseLabel;
B2.trueLabel = B.trueLabel;
B2.falseLabel = B.falseLabel;
B.code = B1.code || gen (B1.trueLabel ‘:’) || B2.code
B1
and
B2
true
false
B.code = gen(‘goto’ B.falseLabel)
B.code = gen(‘goto’ B.trueLabel)

Computing labels

- We can compute the values for the labels but it would require more than one 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

- 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

- Previous solutions do not guarantee a single pass
  - The attribute grammar we had before is not S-attributed (e.g., next), and is not L-attributed.

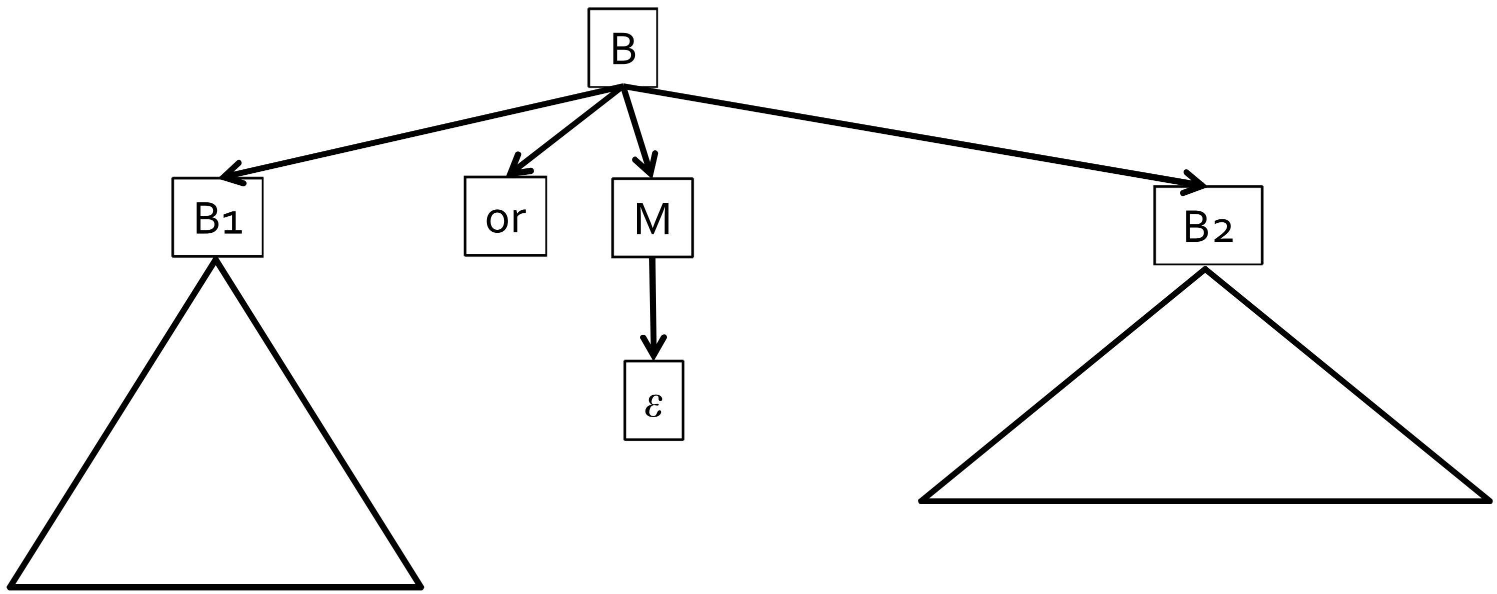
Backpatching

- 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
$B \rightarrow B1$ or **M** $B2$	backpatch(B1.falseList,M.instr); B.trueList = merge(B1.trueList,B2.trueList); B.falseList = B2.falseList;
$B \rightarrow B1$ and **M** $B2$	backpatch(B1.trueList,M.instr); B.trueList = B2.trueList; B.falseList = merge(B1.falseList,B2.falseList);
$B \rightarrow$ not B1	B.trueList = B1.falseList; B.falseList = B1.trueList;
$B \rightarrow$ (B1)	B.trueList = B1.trueList; B.falseList = B1.falseList;
$B \rightarrow$ id1 relop id2	B.trueList = makeList(nextInstr); B.falseList = makeList(nextInstr+1); emit ('if' id1.var relop id2.var 'goto _') || emit('goto _');
$B \rightarrow$ true	B.trueList = makeList(nextInstr); emit ('goto _');
$B \rightarrow$ false	B.falseList = makeList(nextInstr); emit ('goto _');
M $\rightarrow \varepsilon$	M.instr = nextinstr;

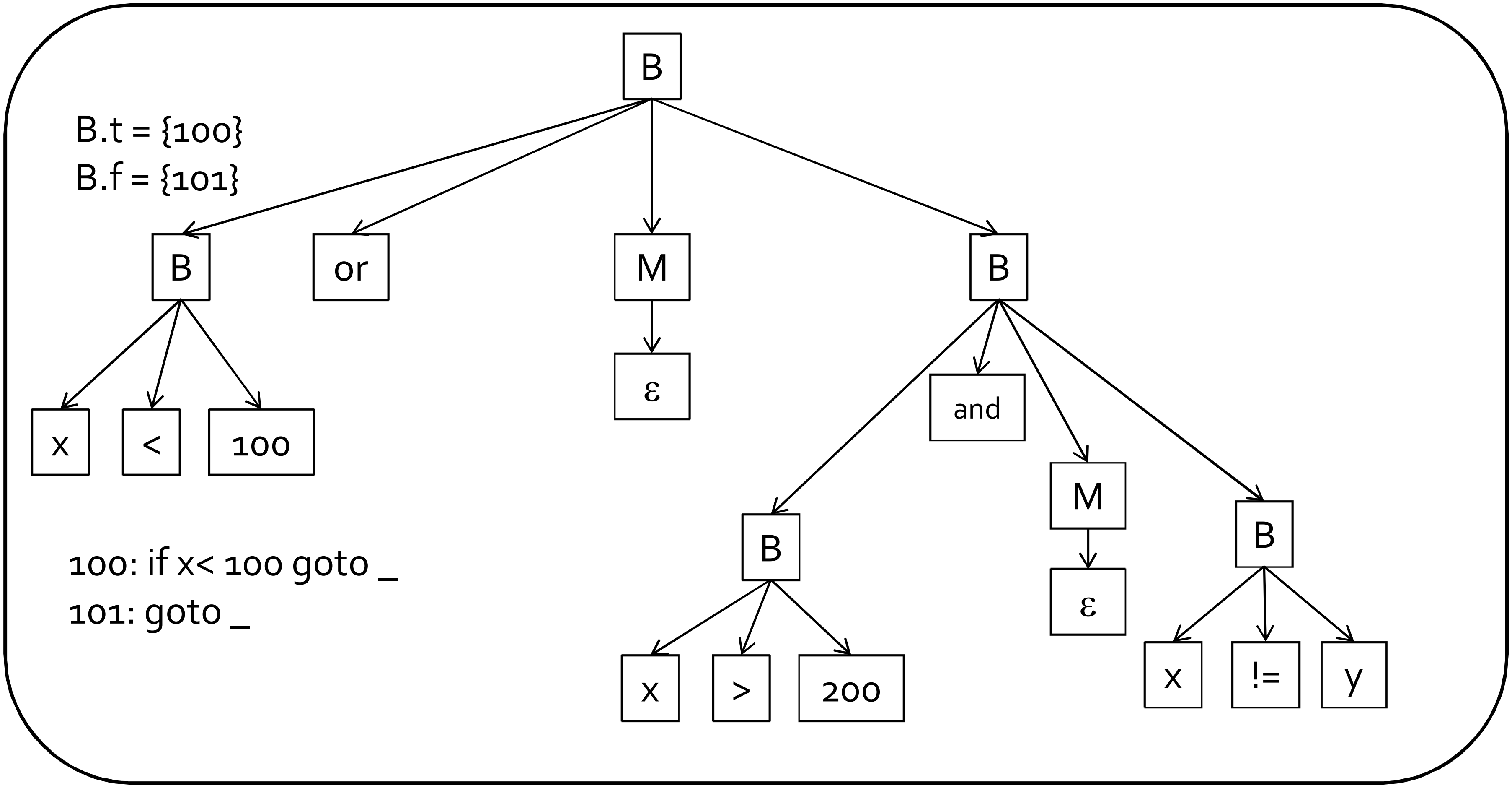
Marker



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

Example

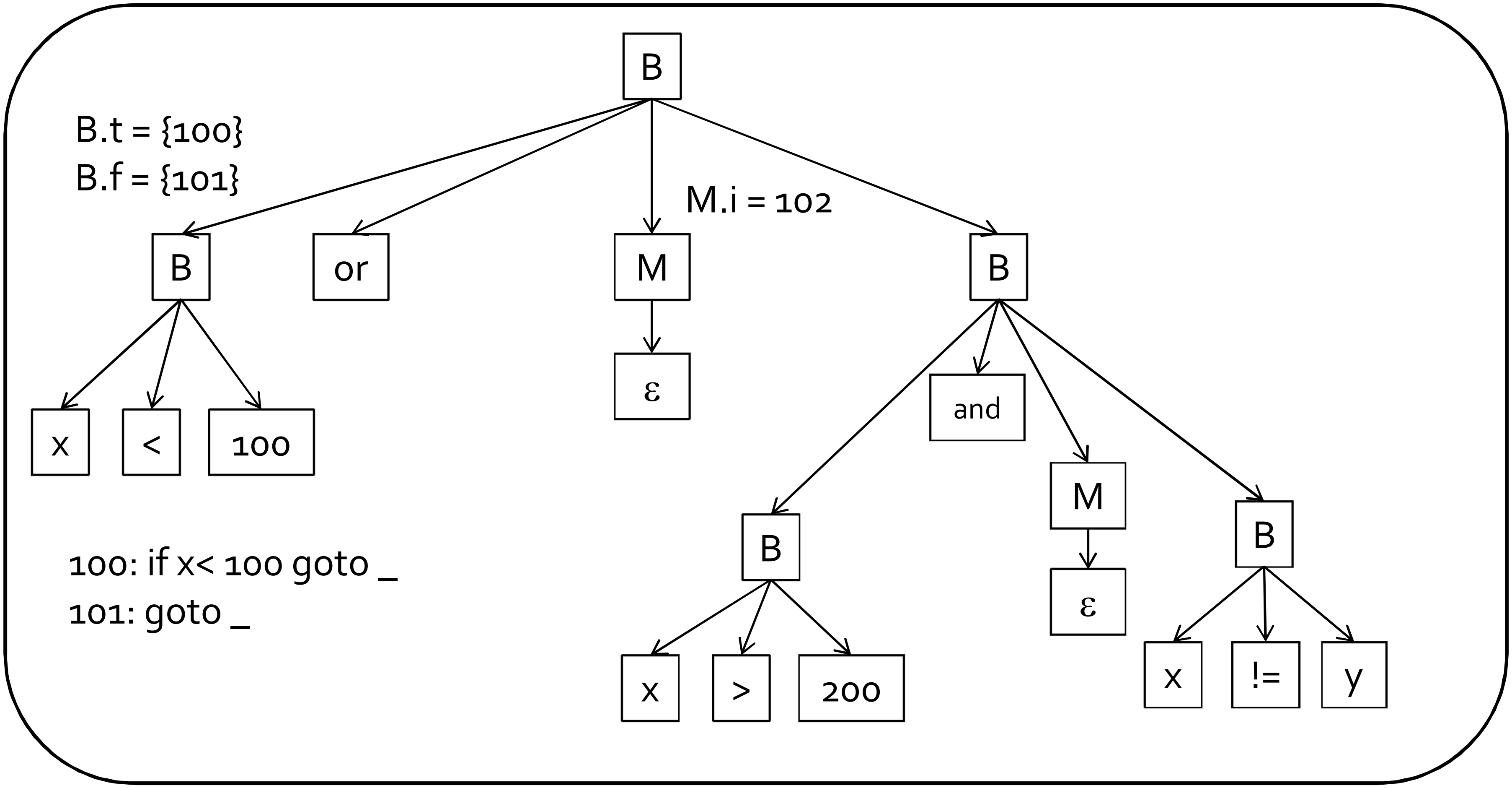
X < 100 or x > 200 and x != y



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

Example

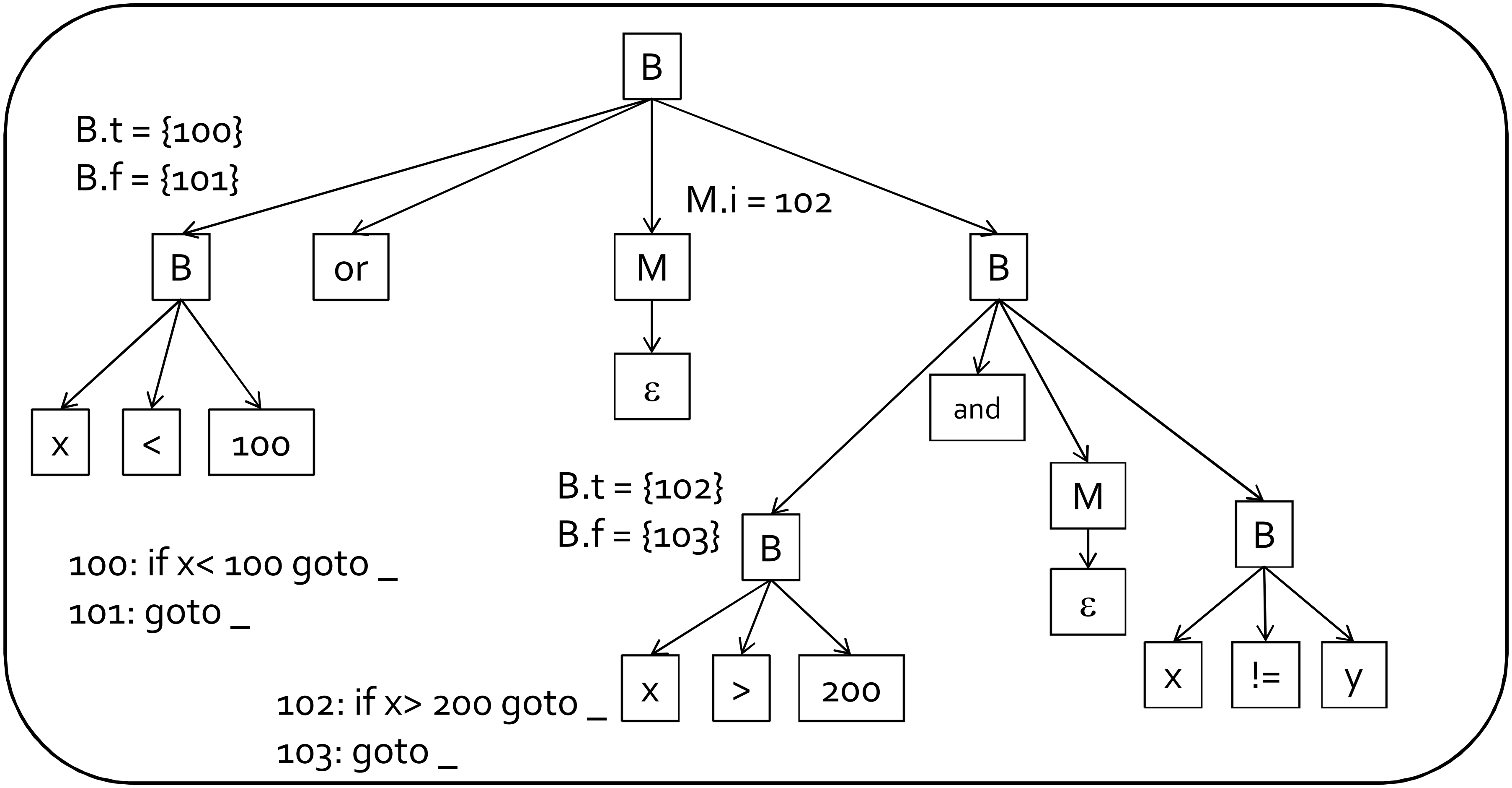
X < 100 or x > 200 and x != y



M → ε	**M.instr = nextinstr;**

Example

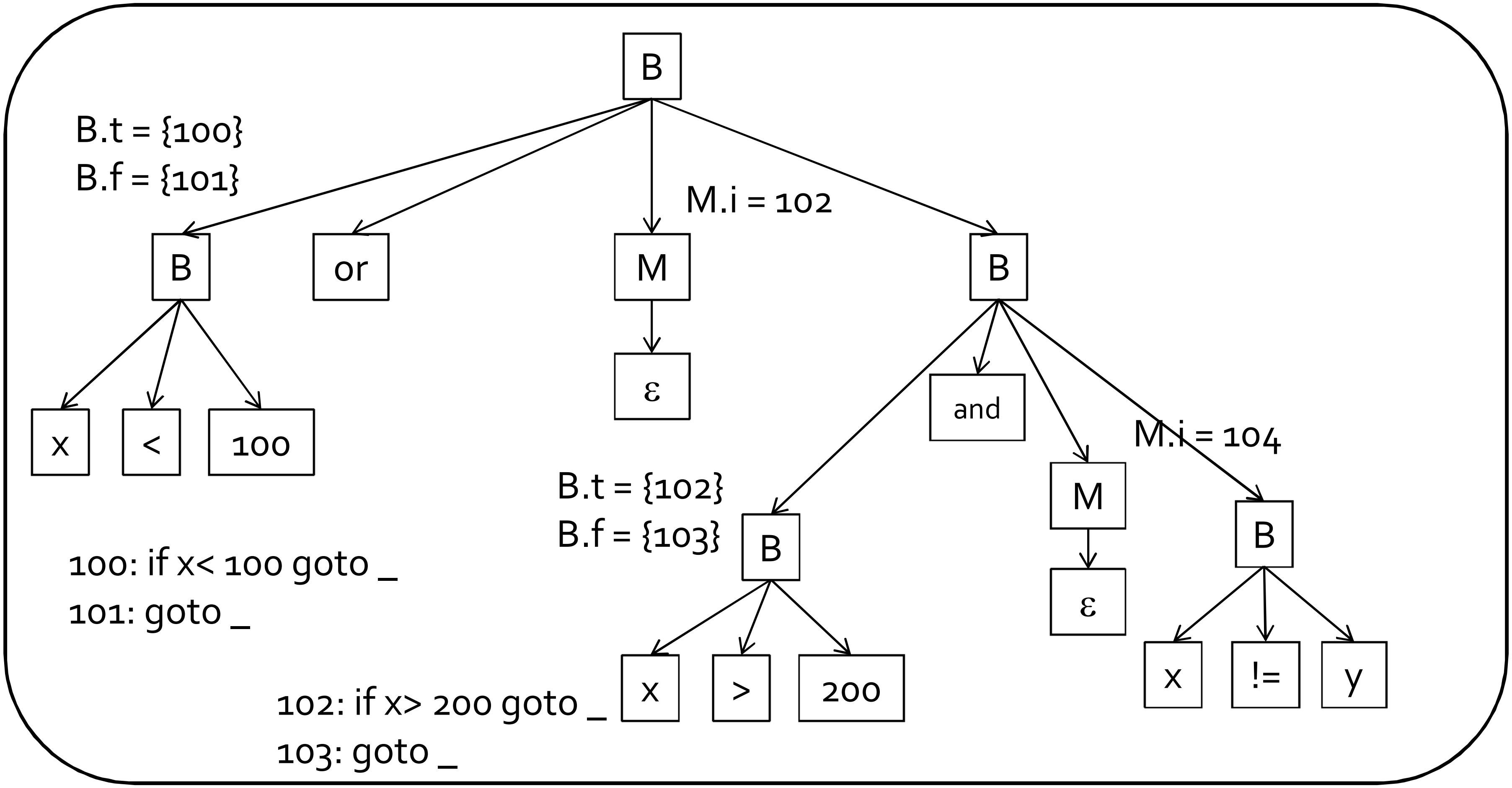
X < 100 or x > 200 and x != y



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

Example

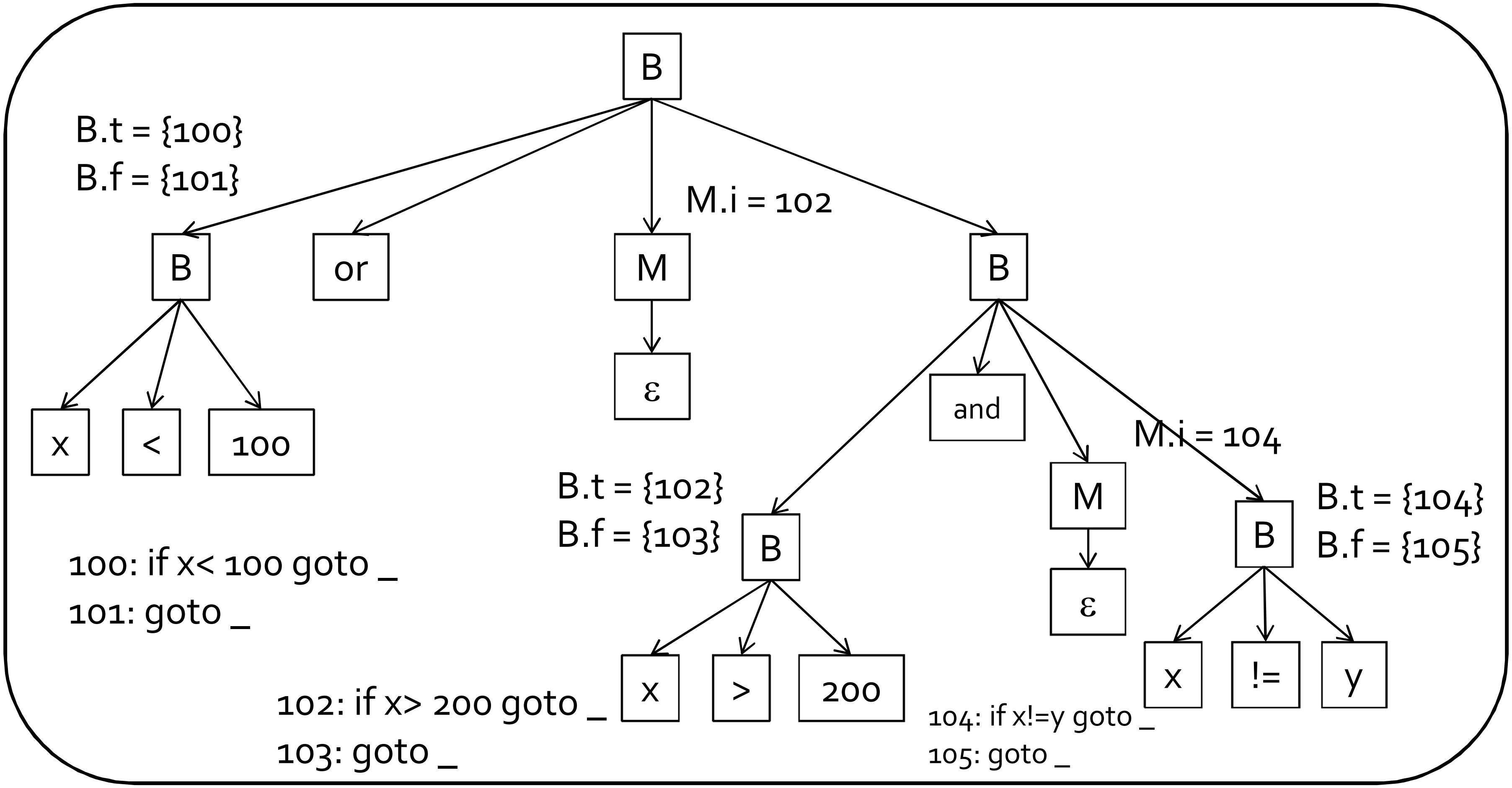
X < 100 or x > 200 and x != y



M → ε	**M.instr = nextinstr;**

Example

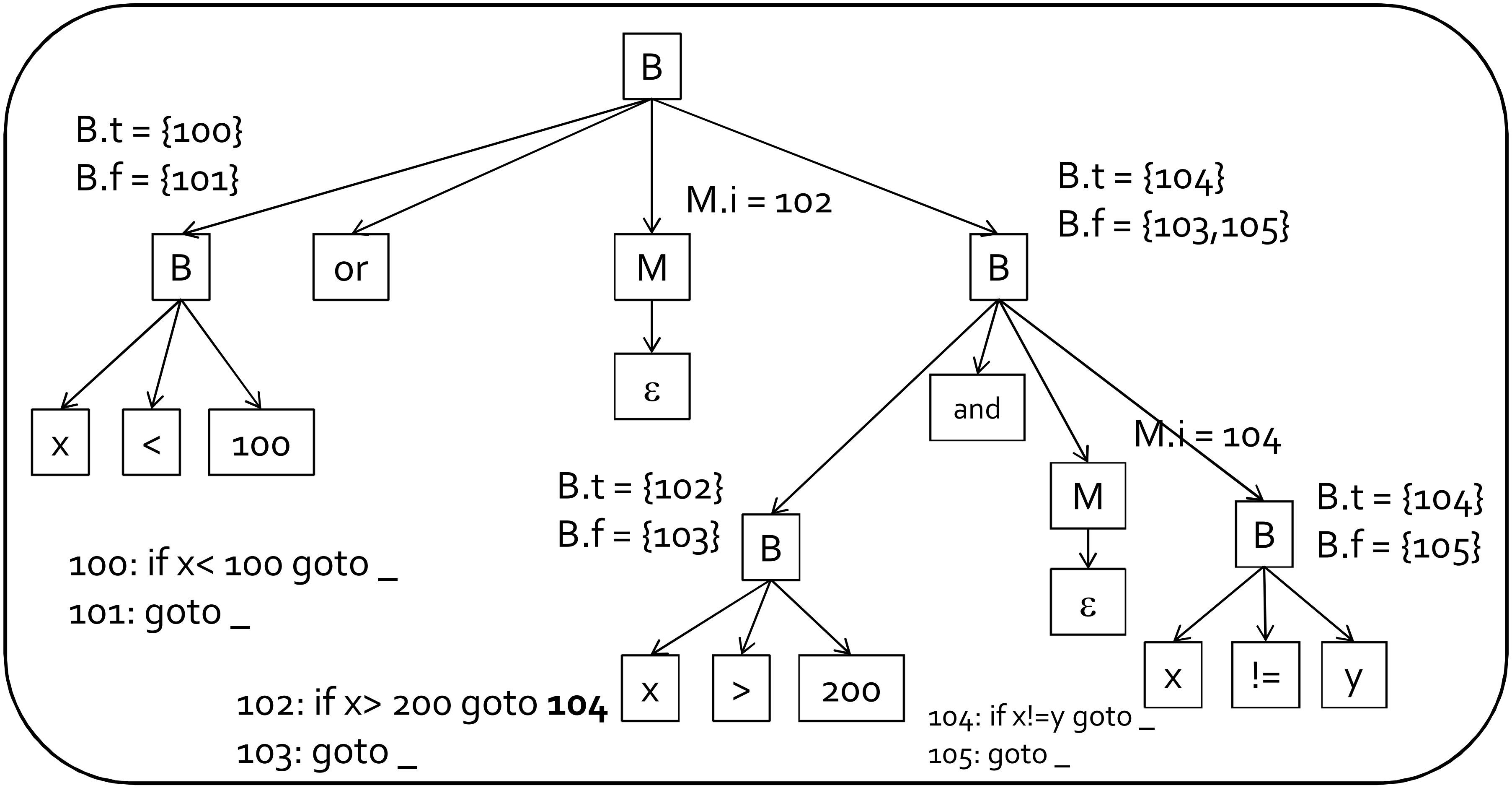
X < 100 or x > 200 and x != y



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

Example

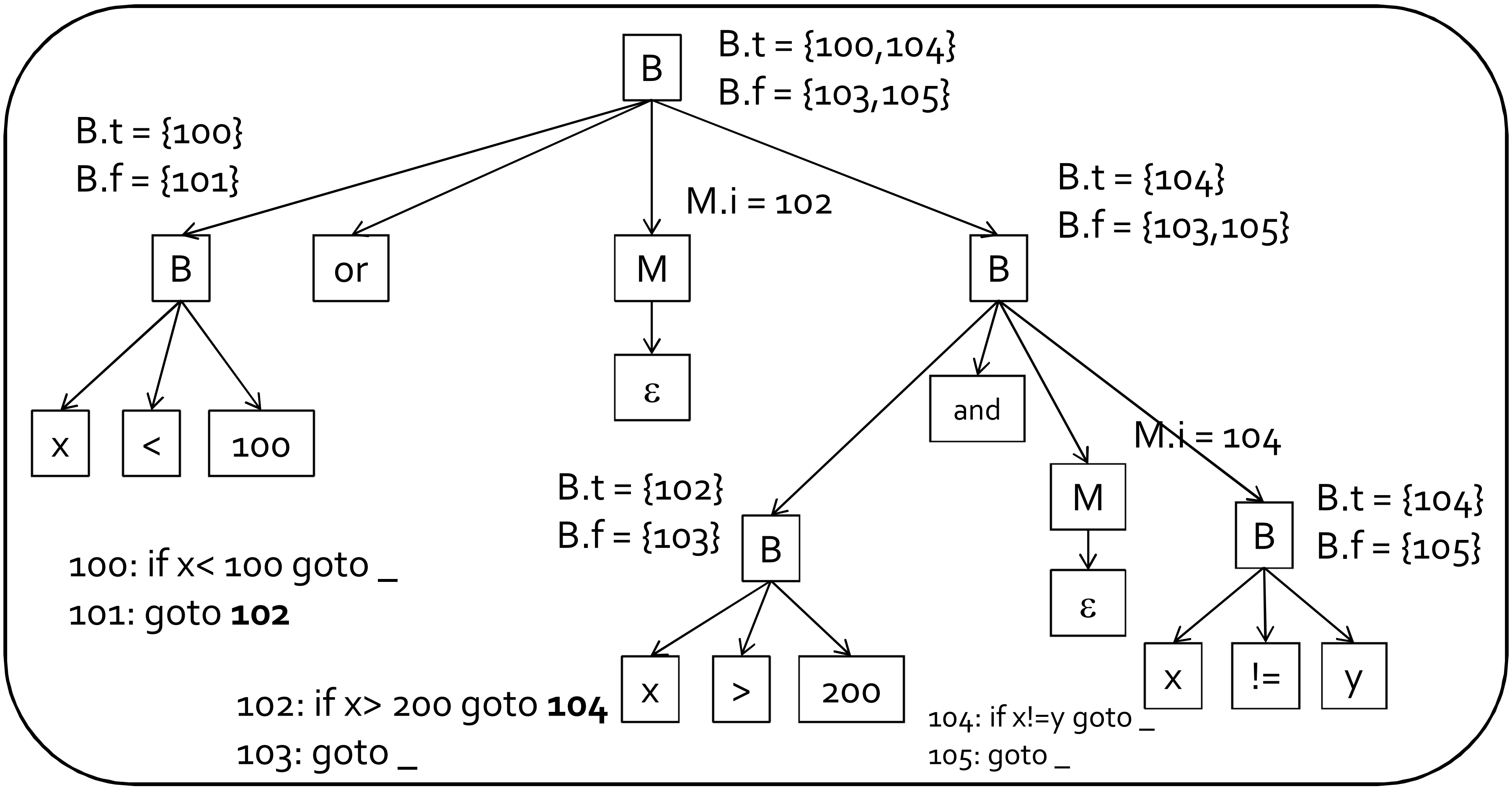
X < 100 or x > 200 and x != y



B → B1 and M B2	**backpatch(B1.trueList,M.instr);** **B.trueList = B2.trueList;** **B.falseList = merge(B1.falseList,B2.falseList);**

Example

X < 100 or x > 200 and x != y



B → B1 or M B2	**backpatch(B1.falseList,M.instr);** **B.trueList = merge(B1.trueList,B2.trueList);** **B.falseList = B2.falseList;**

Example

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

Before backpatching

100: if x<100 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<100 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); S.nextList = merge(B.falseList,S1.nextList);

S → if (B) M1 S1 N else M2 S2	backpatch(B.trueList,M1.instr);
	backpatch(B.falseList,M2.instr);
	temp = merge(S1.nextList,N.nextList);
	S.nextList = merge(temp,S2.nextList);

S → while M1 (B) M2 S1	backpatch(S1.nextList,M1.instr);
	backpatch(B.trueList,M2.instr);
	S.nextList = B.falseList;
	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); L.nextList = S.nextList;
L → S	L.nextList = S.nextList

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; | …        (statements)
E → id (A) | …           (expressions)
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

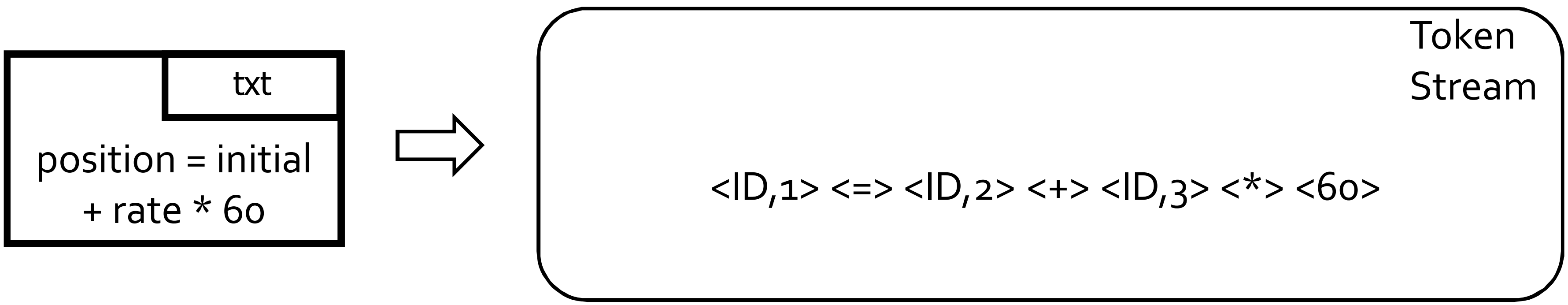
Summary

- pick an intermediate representation
- translate expressions
- use a symbol table to implement delcarations
- 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".

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 judgements
  - can be encoded via attribute grammars
- Syntax directed translation
  - attribute grammars
- Intermediate representation
  - many possible IRs
  - generation of intermediate representation

Journey inside a compiler

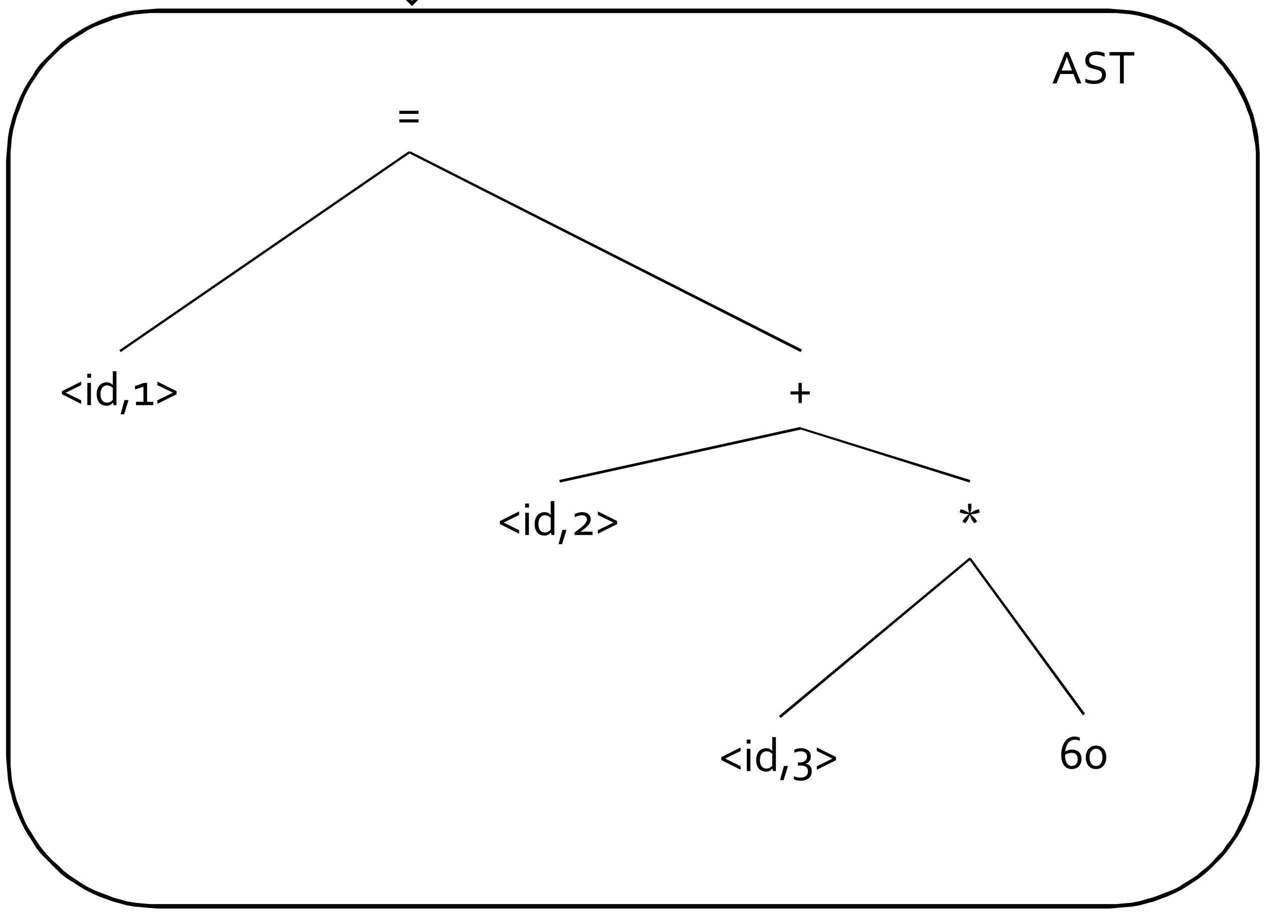


Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

Journey inside a compiler

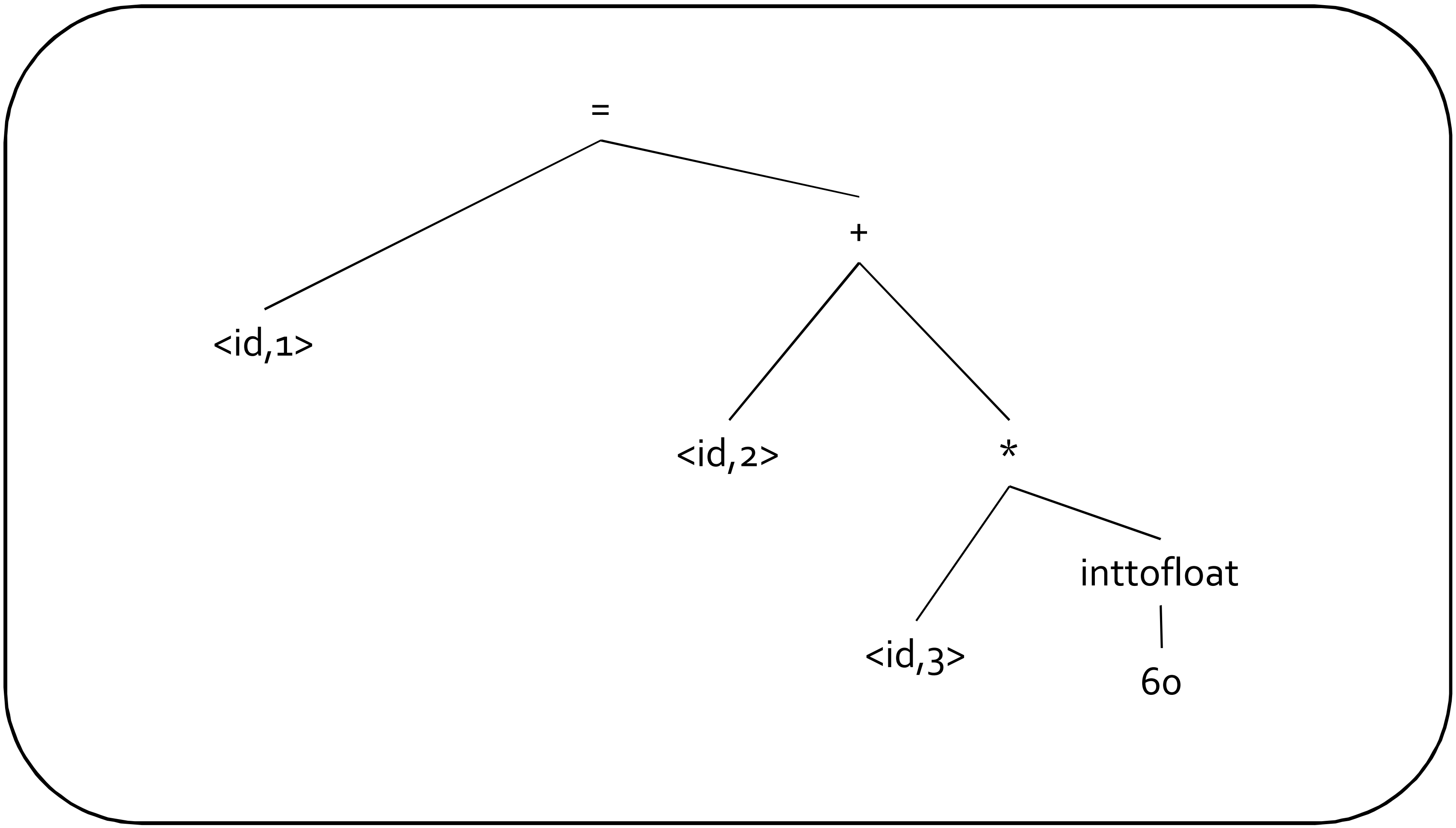
<ID,1> <=> <ID,2> <+> <ID,3> <*> <60>

symbol	type	data
position	float	...
initial	float	...
rate	float	...



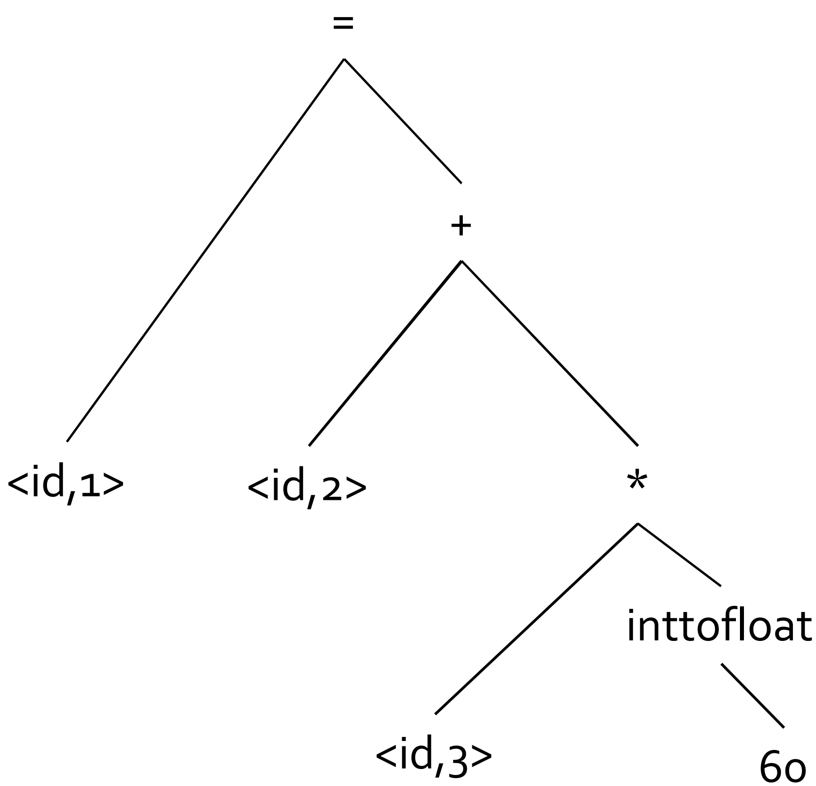
Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

Journey inside a compiler



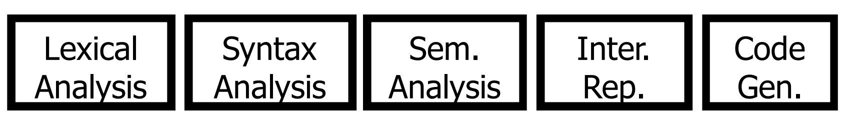
Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

Journey inside a compiler



Intermediate Representation

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



Journey inside a compiler

Intermediate Representation

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

Optimized

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

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.

Next time

- Runtime Environments

The End