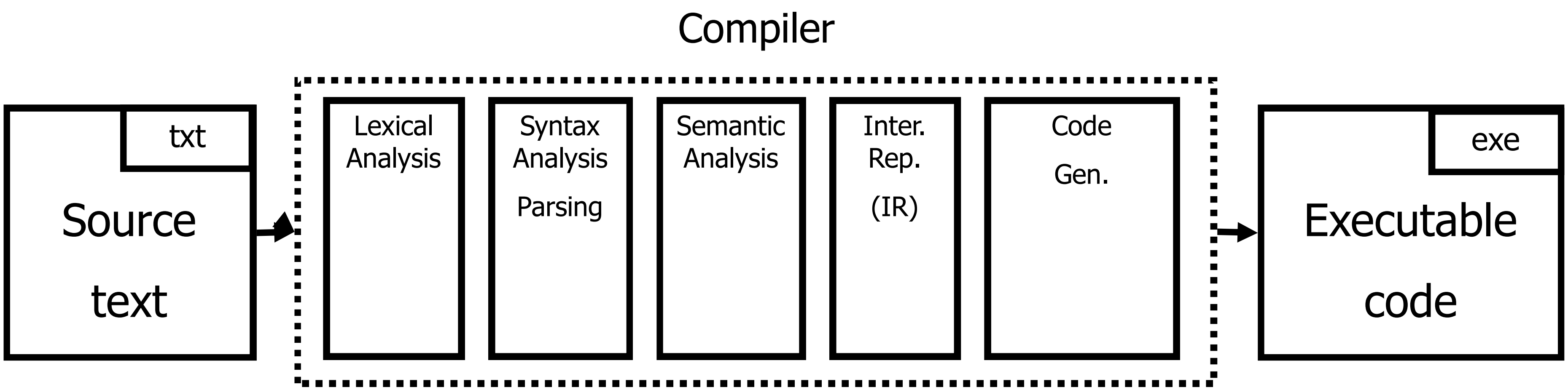
Lecture 07 – attribute grammars + intro to IR

# THEORY OF COMPILATION

Eran Yahav

# You are here



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

# Last Week: Types

- What is a type?
  - Simplest answer: a set of values
  - Integers, real numbers, booleans, …

- Why do we care?
  - Safety
    - Guarantee that certain errors cannot occur at runtime
  - Abstraction
    - Hide implementation details
  - Documentation
  - Optimization

# Last Week: Type System

- A type system of a programming language is a way to define how "good" program behave
  - Good programs = well-typed programs
  - Bad programs = not well typed

- Type checking
  - Static typing – most checking at compile time
  - Dynamic typing – most checking at runtime
- Type inference
  - Automatically infer types for a program (or show that there is no valid typing)

# Strongly vs. weakly typed

- Coercion
- Strongly typed
  - C, C++, Java
- Weakly typed
  - Perl, PHP

- (YMMV, not everybody agrees on this classification)

perl

```perl
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

Output: 73

C

```c
main() {
 int a=31;
 char b[3]="42x";
 int c=a+b;
}
```

warning: initialization makes integer from pointer without a cast

Java

```java
public class… {
 public static void main() {
 int a=31;
 String b ="42x";
 int c=a+b;
 }
}
```

error: Incompatible type for declaration. Can't convert java.lang.String to int

## Last week: how does this magic happen?

- We probably need to go over the AST?
- how does this relate to the clean formalism of the parser?

# Syntax Directed Translation

- The parse tree (syntax) is used to drive the translation
- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - How to update the attributes when a production is used in a derivation
- Attribute grammars

# Attribute grammars

- Attributes
  - Every grammar symbol has attached attributes
    - Example: Expr.type
- Semantic actions
  - Every production rule can define how to assign values to attributes
    - Example:
      Expr → Expr + Term
      Expr.type = Expr1.type when (Expr1.type == Term.type)
      Error otherwise

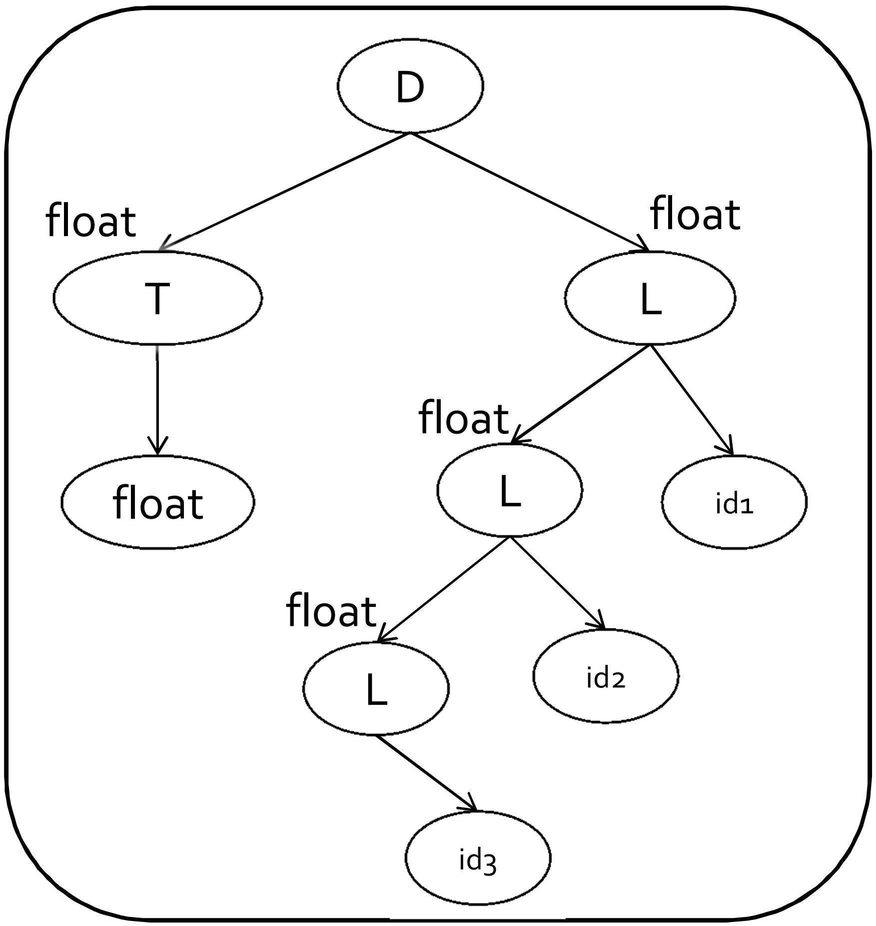
# Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions

- $Expr \longrightarrow Expr + Term$
  Becomes
  $Expr \longrightarrow Expr_1 + Term$

# Example

float x,y,z




| **Production** | **Semantic Rule** |
|---|---|
| D → T L | L.in = T.type |
| T → int | T.type = integer |
| T → float | T.type = float |
| L → $L_1$, id | $L_1$.in = L.in<br>addType(id.entry,L.in) |
| L → id | addType(id.entry,L.in) |

# Attribute Evaluation

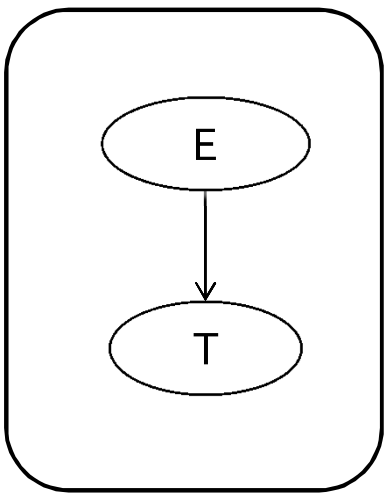
- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before its available
    - Each attribute will get a value only once

# Dependencies

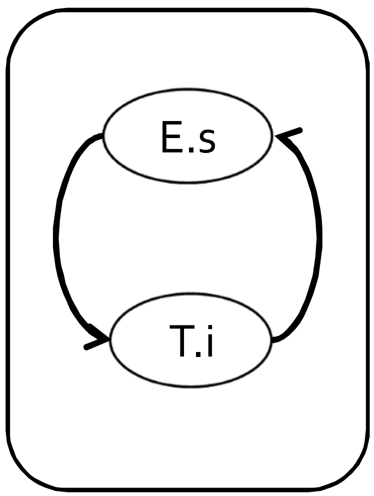
- A semantic equation $a = b_1,\ldots,b_m$ requires computation of $b_1,\ldots,b_m$ to determine the value of a
- The value of a depends on $b_1,\ldots,b_m$
  - We write $a \leftarrow b_i$

# Cycles

- Cycle in the dependence graph
- May not be able to compute attribute values




AST




Dependence graph

E.S = T.i
T.i = E.s + 1

# Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering

- Works as long as there are no cycles

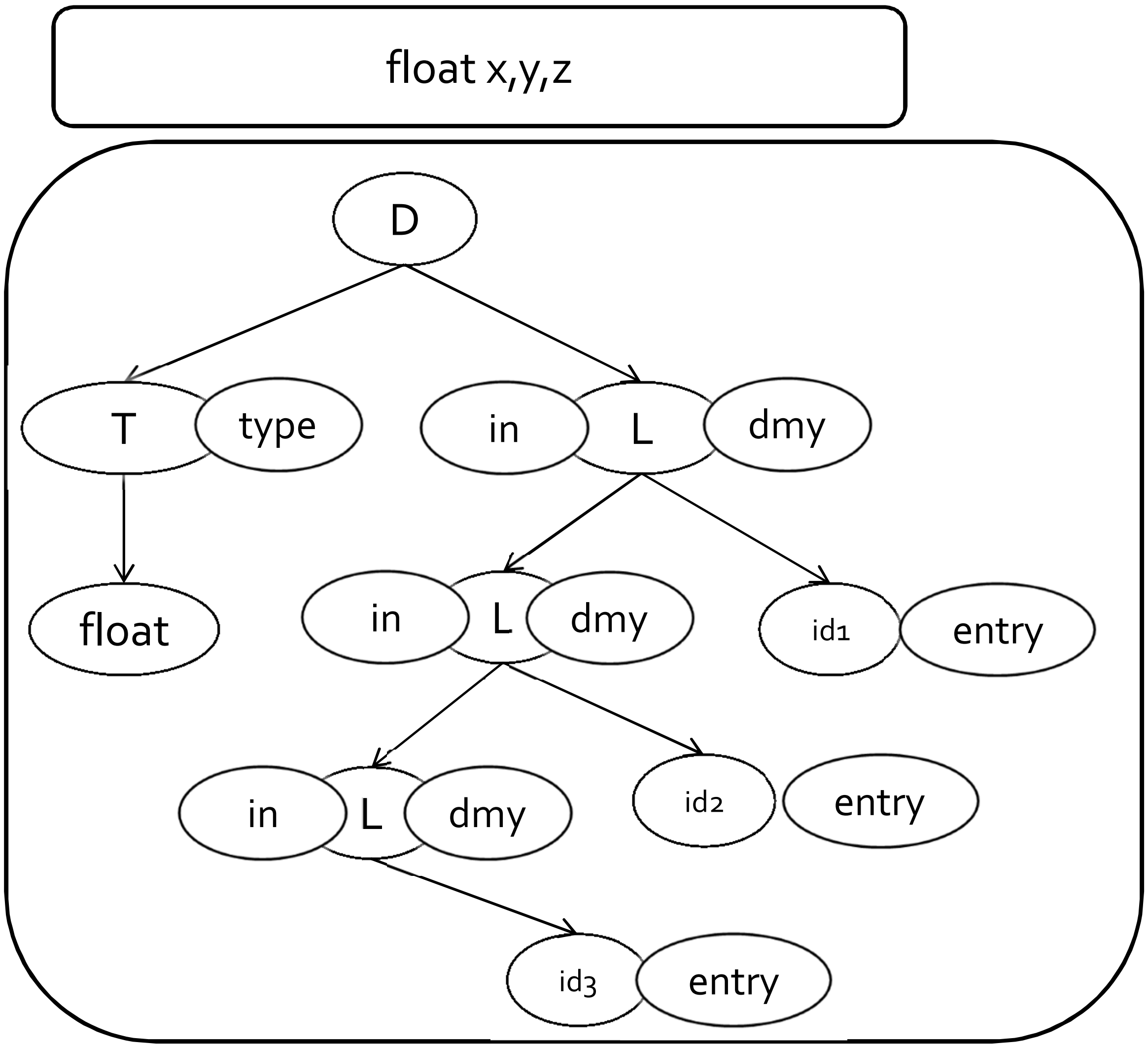
# Building Dependency Graph

- All semantic equations take the form

  attr1 = func1(attr1.1, attr1.2,…)
  attr2 = func2(attr2.1, attr2.2,…)

- Actions with side effects use a dummy attribute
- Build a directed dependency graph G
  - For every attribute a of a node n in the AST create a node n.a
  - For every node n in the AST and a semantic action of the form b = f(c1,c2,…ck) add edges of the form (ci,b)

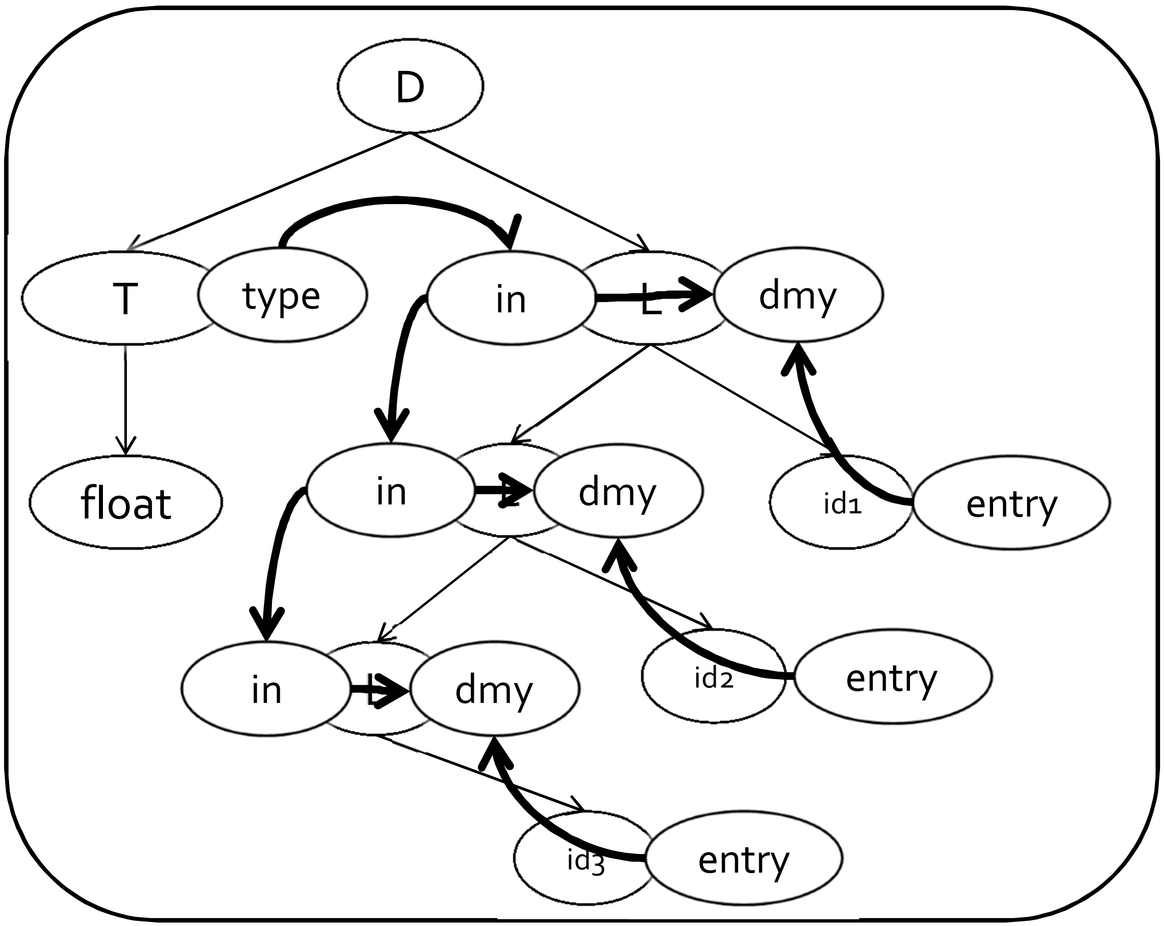
# Example




| Prod. | Semantic Rule |
|---|---|
| D → T L | L.in = T.type |
| T → int | T.type = integer |
| T → float | T.type = float |
| L → $L_1$, id | $L_1$.in = L.in<br>addType(id.entry,L.in) |
| L → id | addType(id.entry,L.in) |

# Example

float x,y,z

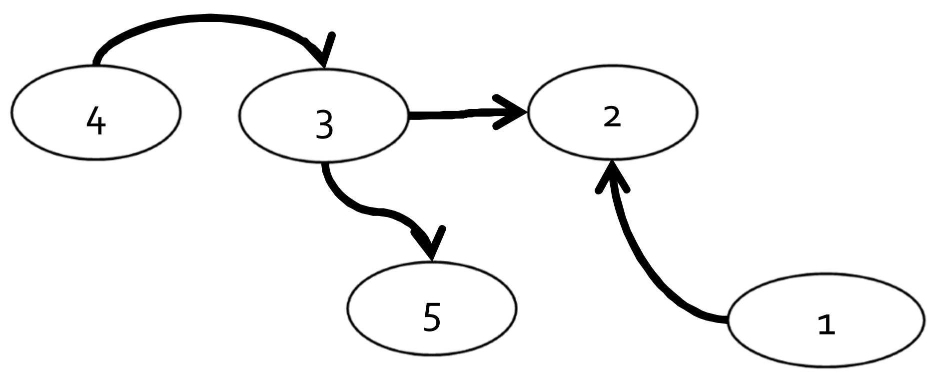



| Prod. | Semantic Rule |
|---|---|
| D → T L | L.in = T.type |
| T → int | T.type = integer |
| T → float | T.type = float |
| L → L1, id | L1.in = L.in<br>addType(id.entry,L.in) |
| L → id | addType(id.entry,L.in) |

# Topological Order

- For a graph G=(V,E), |V|=k

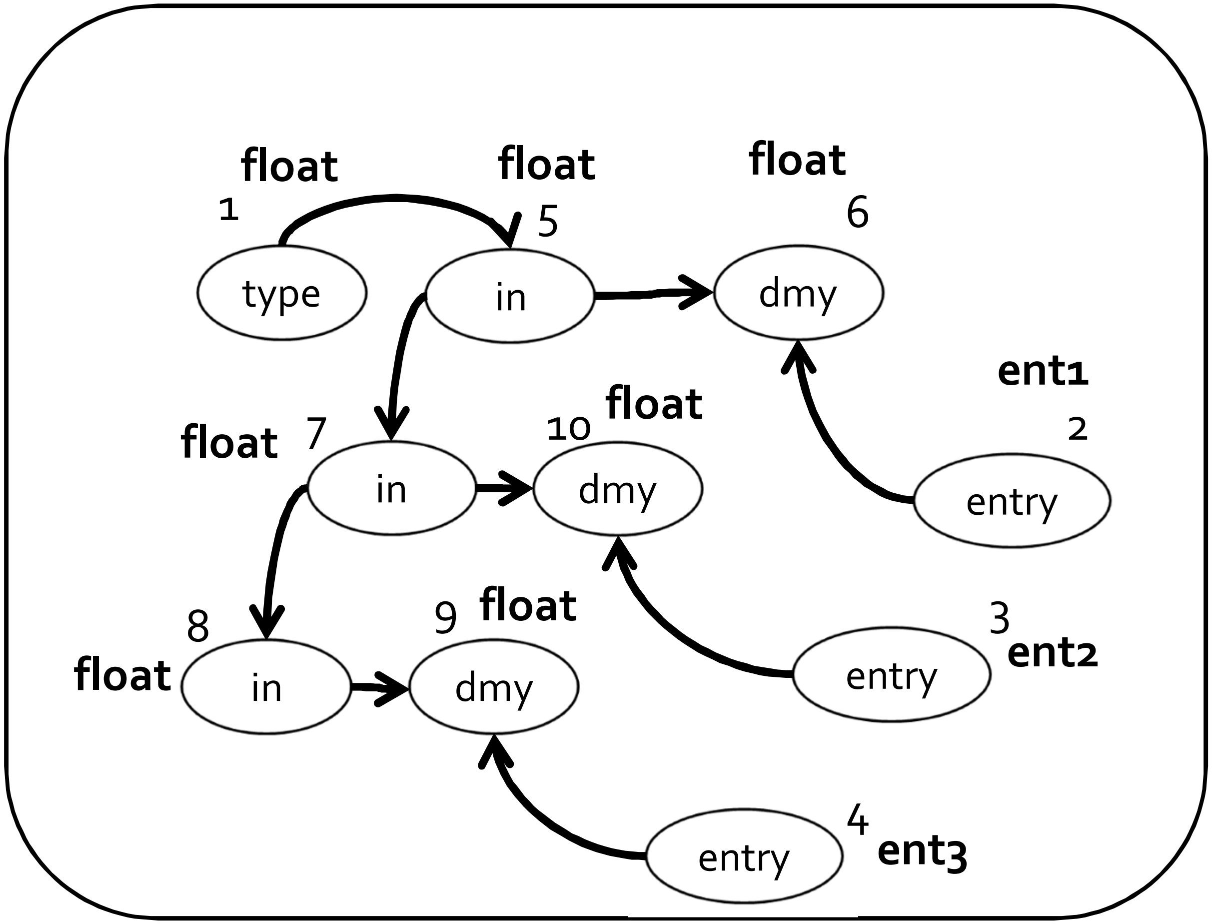
- Ordering of the nodes $v_1, v_2, \ldots v_k$ such that for every edge $(v_i, v_j) \in E$, $i < j$




Example topological orderings: 1 4 3 2 5, 4 1 3 5 2

# Example

float x,y,z



# But what about cycles?

- For a given attribute grammar hard to detect if it has cyclic dependencies
  - Exponential cost

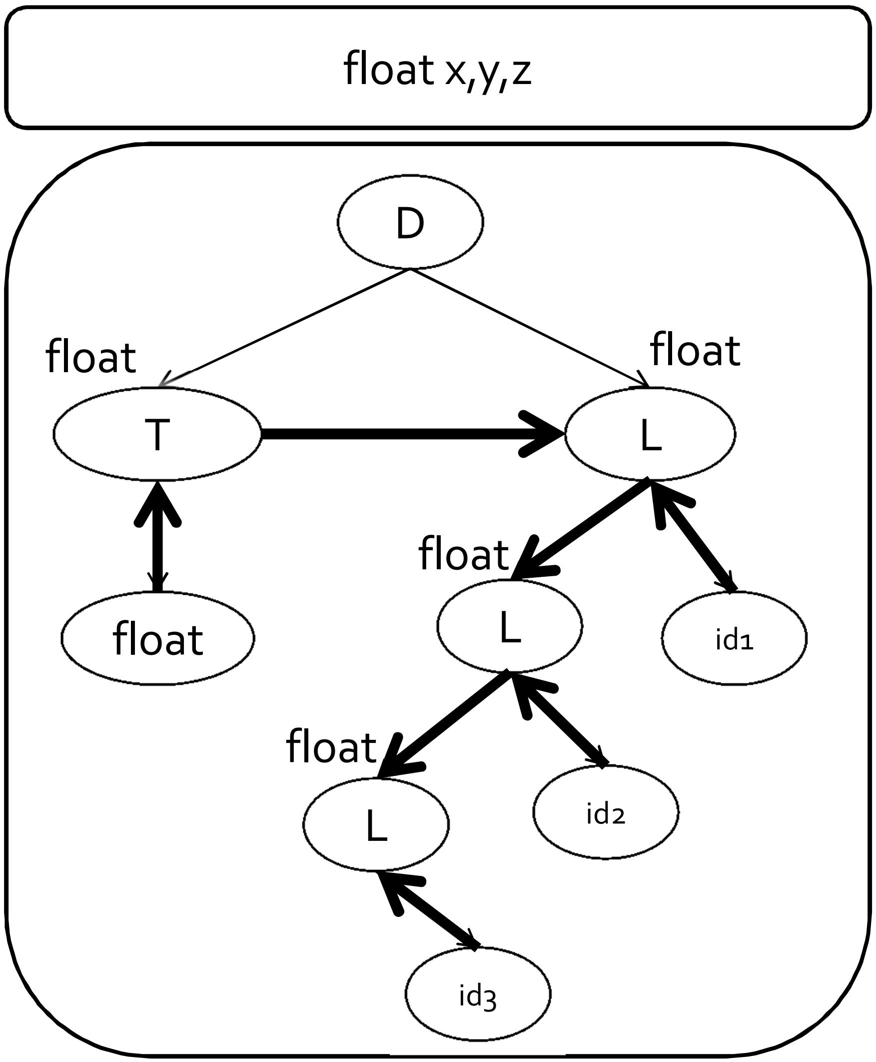
- Special classes of attribute grammars
  - Our “usual trick”
  - sacrifice generality for predictable performance

# Inherited vs. Synthesized Attributes

- Synthesized attributes
  - Computed from children of a node
- Inherited attributes
  - Computed from parents and siblings of a node

- Attributes of tokens are technically considered as synthesized attributes

# example




| Production | Semantic Rule |
|---|---|
| D → T L | L.in = T.type |
| T → int | T.type = integer |
| T → float | T.type = float |
| L → $L_1$, id | $L_1$.in = L.in<br>addType(id.entry,L.in) |
| L → id | addType(id.entry,L.in) |

inherited

synthesized

# S-attributed Grammars

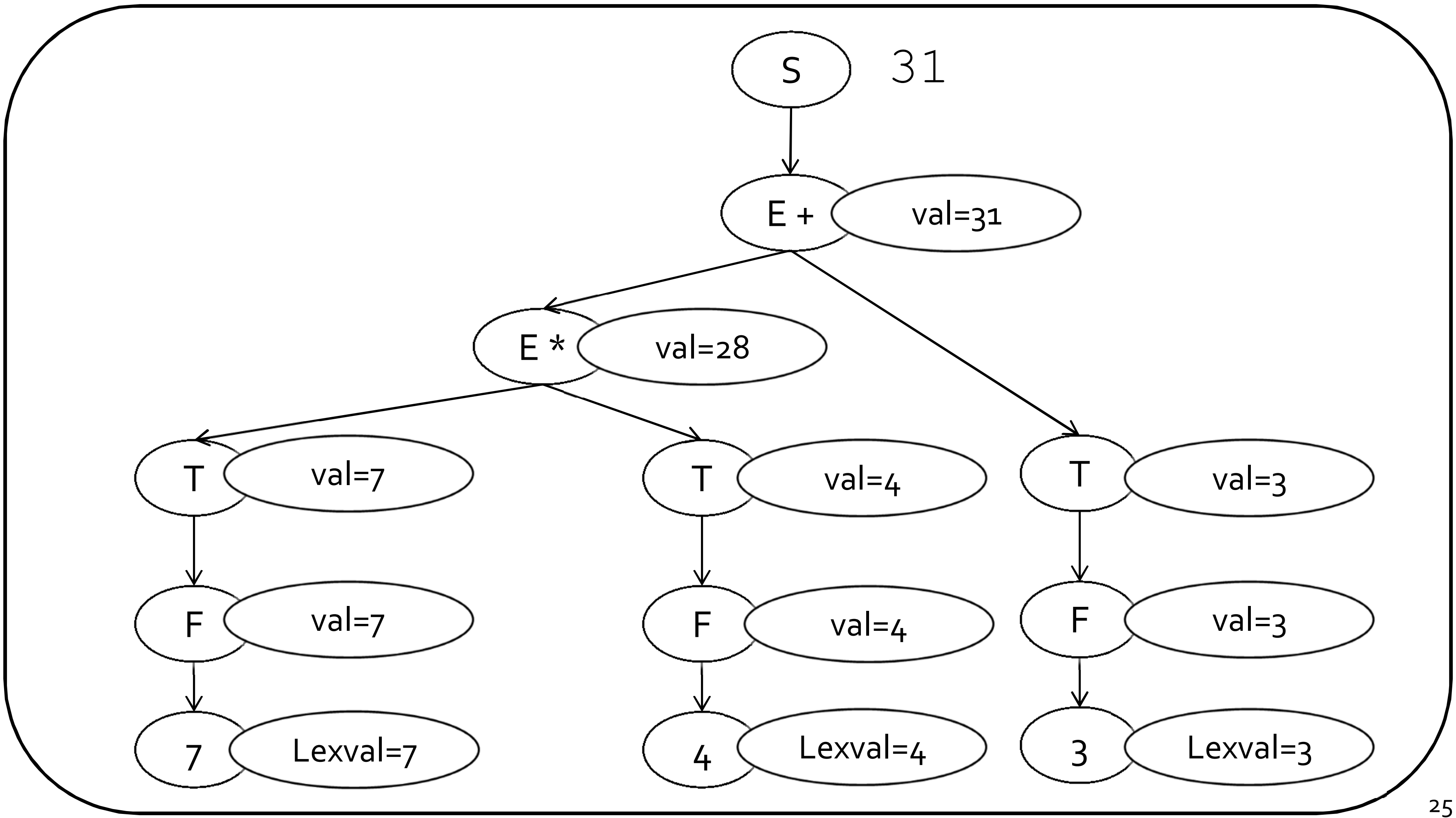
- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes

- Can be computed by any bottom-up parser **during parsing**
- Attributes can be stored on the parsing stack
- Reduce operation computes the (synthesized) attribute from attributes of children

# S-attributed Grammar: example

| **Production** | **Semantic Rule** |
|---|---|
| S→ E ; | print(E.val) |
| E → E1 + T | E.val = E1.val + T.val |
| E → T | E.val = T.val |
| T → T1 * F | T.val = T1.val * F.val |
| T → F | T.val = F.val |
| F → (E) | F.val = E.val |
| F → digit | F.val = digit.lexval |

# example


S
31
E +
val=31
E *
val=28
T
val=7
T
val=4
T
val=3
F
val=7
F
val=4
F
val=3
7
Lexval=7
4
Lexval=4
3
Lexval=3

# L-attributed grammars

- L-attributed attribute grammar when every attribute in a production A → $X_1 \ldots X_n$ is
  - A synthesized attribute, or
  - An inherited attribute of $X_j$, $1 <= j <= n$ that only depends on
    - Attributes of $X_1 \ldots X_{j-1}$ to the left of $X_j$, or
    - Inherited attributes of A

# Example: typesetting


- Vertical geometry
  - pointsize (ps) – size of letters in a box. Subscript text has smaller point size of 0.7p.
  - baseline
  - height (ht) – distance from top of the box to the baseline
  - depth (dp) – distance from baseline to the bottom of the box.

# Example: typesetting

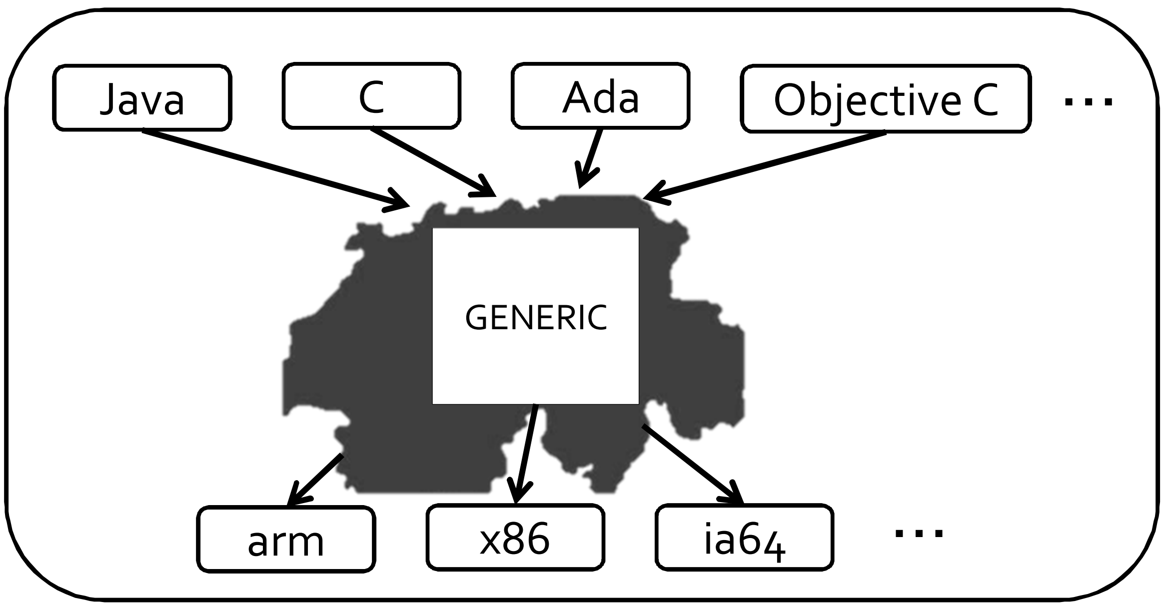
| production | semantic rules |
|---|---|
| S → B | B.ps = 10 |
| B → B1 B2 | B1.ps = B.ps<br>B2.ps = B.ps<br>B.ht = max(B1.ht,B2.ht)<br>B.dp = max(B1.dp,B2.dp) |
| B → B1 sub B2 | B1.ps = B.ps<br>B2.ps = 0.7*B.ps<br>B.ht = max(B1.ht,B2.ht – 0.25*B.ps)<br>B.dp = max(B1.dp,B2.dp– 0.25*B.ps) |
| B → text | B.ht = getHt(B.ps,text.lexval)<br>B.dp = getDp(B.ps,text.lexval) |

# Attribute grammars: summary

- Contextual analysis can move information between nodes in the AST
  - Even when they are not “local”
- Attribute grammars
  - Attach attributes and semantic actions to grammar
- Attribute evaluation
  - Build dependency graph, topological sort, evaluate
- Special classes with pre-determined evaluation order: S-attributed, L-attributed

# Intermediate Representation

- “neutral” representation between the front-end and the back-end
  - Abstracts away details of the source language
  - Abstract away details of the target language
- A compiler may have multiple intermediate representations and move between them
- In practice, the IR may be biased toward a certain language (e.g., GENERIC in gcc)



# Intermediate Representation(s)

- Annotated abstract syntax tree
- Three address code
- …

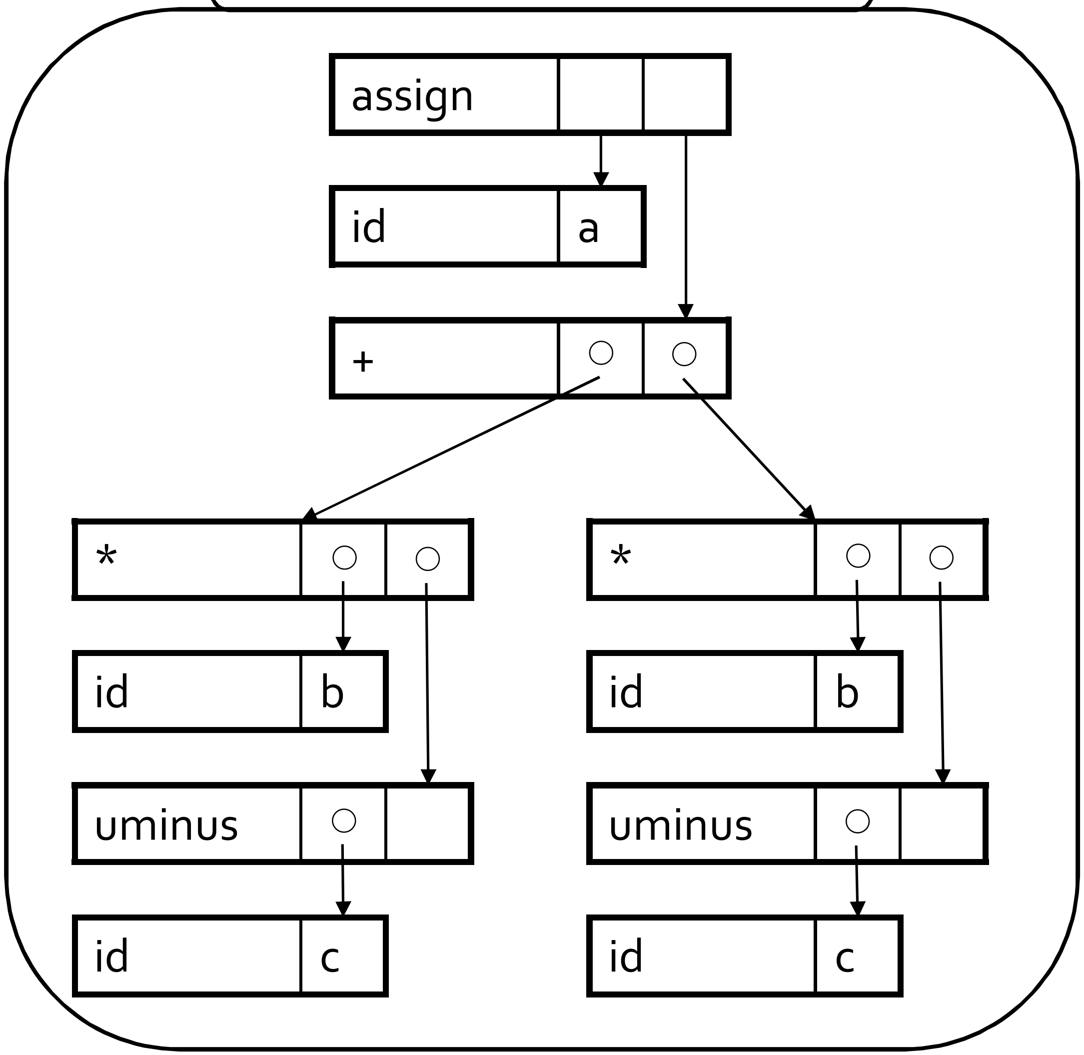
# Example: Annotated AST

| production | semantic rule |
|---|---|
| S → id := E | S.nptr = makeNode('assign', makeLeaf(id,id.place), E.nptr) |
| E → E1 + E2 | E.nptr = makeNode('+',E1.nptr,E2.nptr) |
| E → E1 * E2 | E.nptr = makeNode('*',E1.nptr,E2.nptr) |
| E → -E1 | E.nptr = makeNode('uminus',E1.nptr) |
| E → (E1) | E.nptr = E1.nptr |
| E → id | E.nptr = makeLeaf(id,id.place) |

- makeNode – creates new node for unary/binary operator
- makeLeaf – creates a leaf
- id.place – pointer to symbol table

# Example

a = b * -c + b* -c

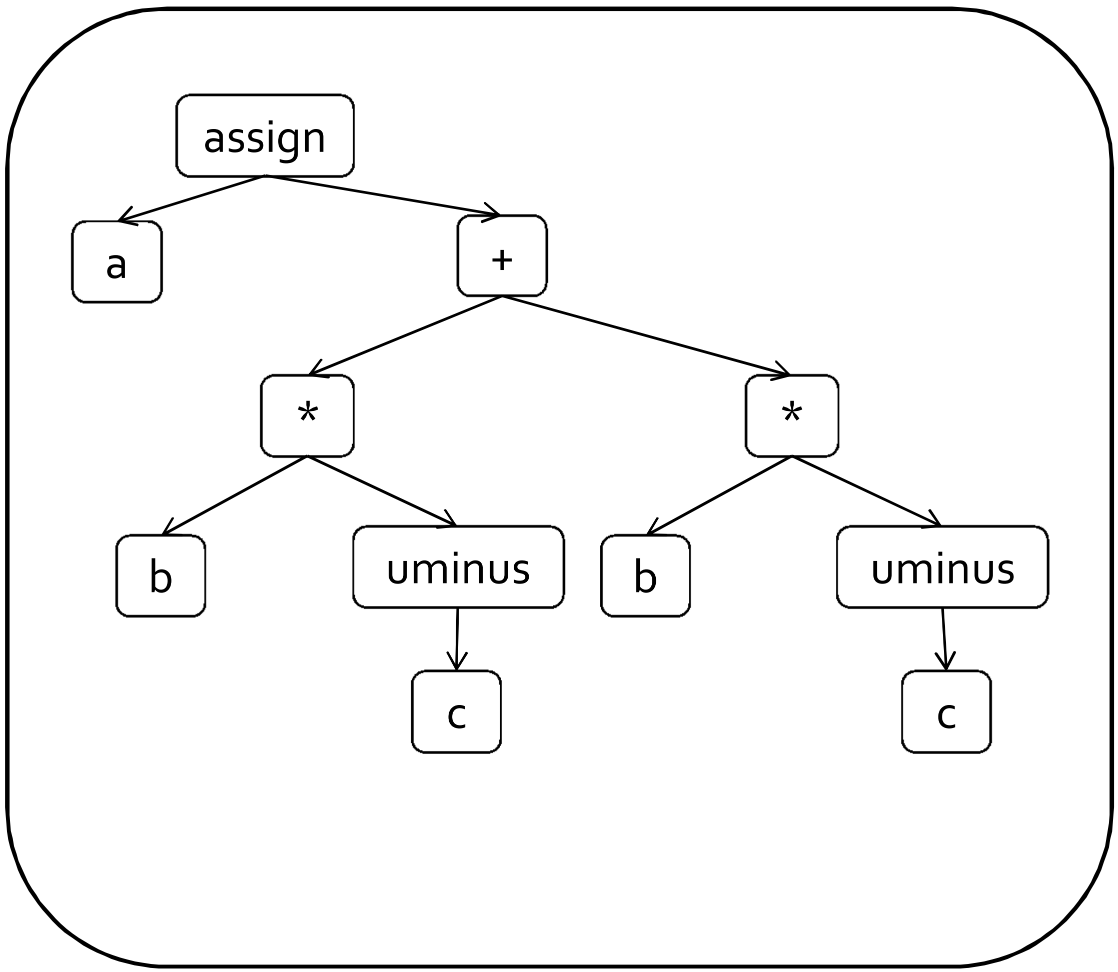



| | | | |
|---|---|---|---|
| 0 | id | b | |
| 1 | id | c | |
| 2 | uminus | 1 | |
| 3 | * | 0 | 2 |
| 4 | id | b | |
| 5 | id | c | |
| 6 | uminus | 5 | |
| 7 | * | 4 | 6 |
| 8 | + | 3 | 7 |
| 9 | id | a | |
| 10 | assign | 9 | 8 |
| 11 | . . . | | |

# 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

# Three address code: example




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

# Three address code: example instructions

| **instruction** | **meaning** |
|---|---|
| x := y op z | assignment with binary operator |
| x := op y | assignment unary operator |
| x:= y | assignment |
| x := &y | assign address of y |
| x:=*y | assignment from deref y |
| *x := y | assignment to deref x |

| **instruction** | **meaning** |
|---|---|
| goto L | unconditional jump |
| if x relop y goto L | conditional jump |

# Array operations

- Are these 3AC operations?

x := y[i]

```
t1 := &y       ; t1 = address-of y
t2 := t1 + i   ; t2 = address of y[i]
X  := *t2      ; value stored at y[i]
```

x[i] := y

```
t1 := &x       ; t1 = address-of x
t2 := t1 + i   ; t2 = address of x[i]
*t2:= y        ; store through pointer
```

# 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)

# Three address code

- Choice of instructions and operators affects code generation and optimization

- Small set of instructions
  - Easy to generate machine code
  - Harder to optimize
- Large set of instructions
  - Harder to generate machine code

- Typically prefer small set and smart optimizer

# Creating 3AC

- Assume bottom up parser
  - Why?

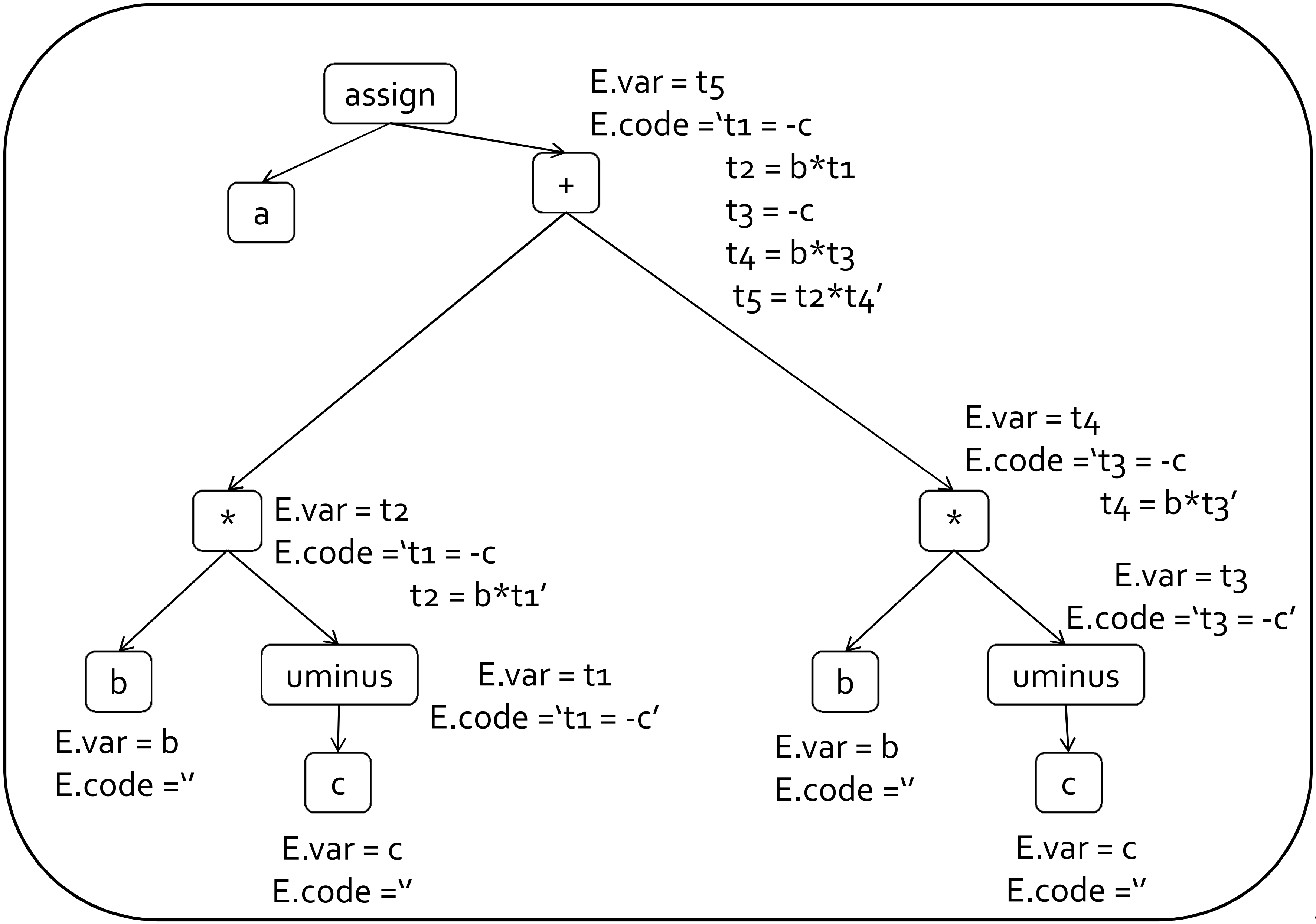
- Creating 3AC via syntax directed translation
- Attributes
  - code – code generated for a nonterminal
  - var – name of variable that stores result of nonterminal
- freshVar – helper function that returns the name of a fresh variable

# 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 → $E_1$ + $E_2$</td><td>E.var := freshVar();<br>E.code = $E_1$.code || $E_2$.code || gen(E.var ‘:=‘ $E_1$.var ‘+’ $E_2$.var)</td></tr>
<tr><td>E → $E_1$ * $E_2$</td><td>E.var := freshVar();<br>E.code = $E_1$.code || $E_2$.code || gen(E.var ‘:=‘ $E_1$.var ‘*’ $E_2$.var)</td></tr>
<tr><td>E → - $E_1$</td><td>E.var := freshVar();<br>E.code = $E_1$.code || gen(E.var ‘:=‘ ‘uminu’ $E_1$.var)</td></tr>
<tr><td>E → ($E_1$)</td><td>E.var := $E_1$.var<br>E.code = ‘(‘ || $E_1$.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



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
uminus
E.var = t1
E.code ='t1 = -c'
b
uminus
E.var = t3
E.code ='t3 = -c'
E.var = b
E.code =''
c
E.var = c
E.code =''
E.var = b
E.code =''
c
E.var = c
E.code =''

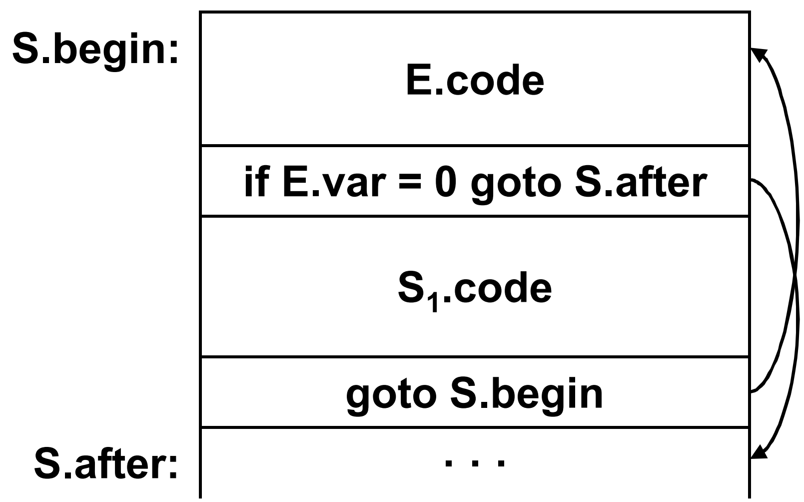
# Creating 3AC: control statements

- 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

# Creating 3AC: control statements

S → while E do S1




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

# Representing 3AC

- Quadruple (op,arg1,arg2,result)
- Result of every instruction is written into a new temporary variable
- Generates many variable names
- Can move code fragments without complicated renaming
- Alternative representations may be more compact

```
t1 = - c
t2 = b * t1
t3 = - c
t4 = b * t3
t5 = t2 * t4
a = t5
```

| | op | arg 1 | arg 2 | result |
|---|---|---|---|---|
| (0) | uminus | c | | $t_1$ |
| (1) | * | b | $t_1$ | $t_2$ |
| (2) | uminus | c | | $t_3$ |
| (3) | * | b | $t_3$ | $t_4$ |
| (4) | + | $t_2$ | $t_4$ | $t_5$ |
| (5) | := | $t_5$ | | a |

# Allocating Memory

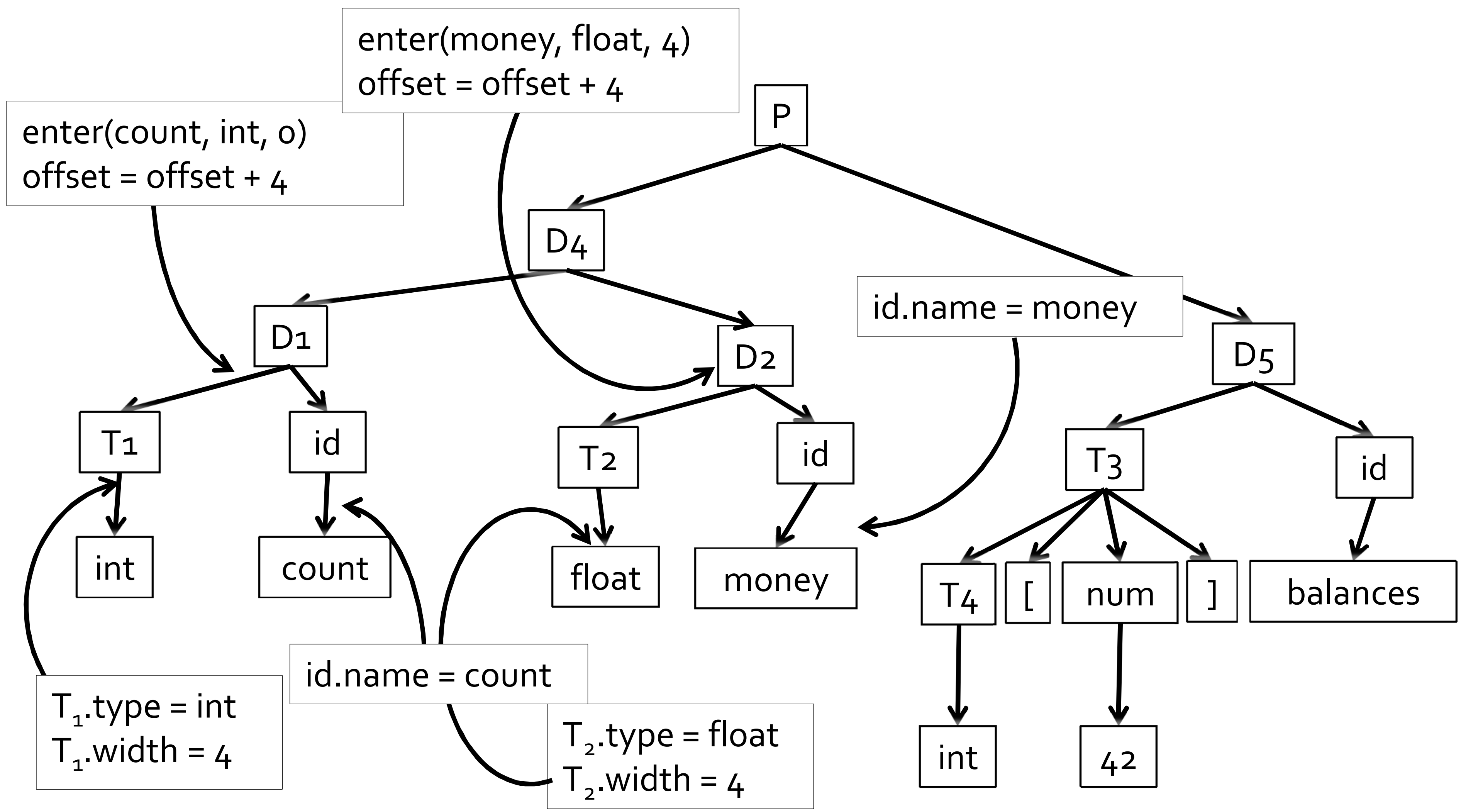
- Type checking helped us guarantee correctness
- Also tells us
  - How much memory allocate on the heap/stack for varaibles
  - Where to find variables (based on offsets)
  - Compute address of an element inside array (size of stride based on type of element)

# 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

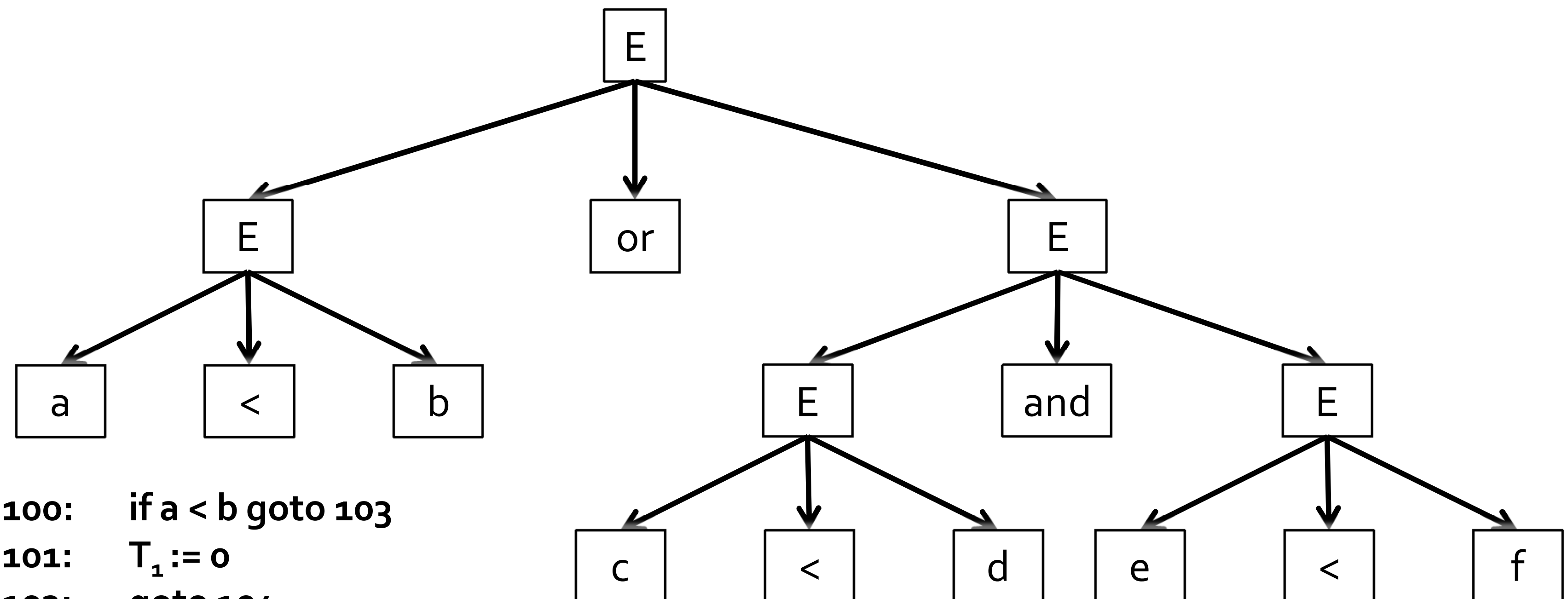
| **production** | **semantic action** |
|---|---|
| E → E1 op E2 | { E.var := freshVar(); **emit**(E.var ':=' E1.var op E2.var) } |
| E → not E1 | { E.var := freshVar(); **emit**(E.var ':=' 'not' E1.var) } |
| E → ( E1) | { E.var := E1.var } |
| E → true | { E.var := freshVar(); **emit**(E.var ':=' '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 | {<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

# More examples

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

```
int x=0;
if (++x>0 && x==1) {
        hmmm();
}
```

# Summary

- Three address code (3AC)
- Generating 3AC
- Boolean expressions
- Short circuit evaluation

# Next time

- Generating IR for control structures
  - While, for, if
- backpatching

# The End