Lecture 07 – attribute grammars + intro to IR

THEORY OF COMPILATION

Eran Yahav
You are here

Compiler

Source text

Lexical Analysis
Syntax Analysis
Semantic Analysis
Inter. Rep. (IR)
Code Gen.

Executable code

exe

txt
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” programs 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
$a=31;
$b="42x";
$c=$a+$b;
print $c;
```

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

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

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

Output: 73

warning: initialization makes integer from pointer without a cast
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:
      \[
      \text{Expr} \rightarrow \text{Expr} + \text{Term}
      \]
      \[
      \text{Expr.type} = \text{Expr1.type when (Expr1.type == Term.type)} \\
      \text{Error otherwise}
      \]
Indexed symbols

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

- $\text{Expr} \rightarrow \text{Expr} + \text{Term}$
  Becomes
- $\text{Expr} \rightarrow \text{Expr}_1 + \text{Term}$
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 → L1, id | L1.in = L.in
            | 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 $a = b_1, ..., b_m$ requires computation of $b_1, ..., b_m$ to determine the value of $a$

- The value of $a$ depends on $b_1, ..., b_m$
  - We write $a \leftarrow b_i$
Cycles

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

\[ E.S = T.i \]
\[ T.i = E.s + 1 \]
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
  
  \[ \text{attr1} = \text{func1} (\text{attr1.1}, \text{attr1.2}, \ldots) \]
  
  \[ \text{attr2} = \text{func2} (\text{attr2.1}, \text{attr2.2}, \ldots) \]

- 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 \( b = f(c_1, c_2, \ldots, c_k) \) add edges of the form \((c_i, b)\)
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
 | addType(id.entry, L.in)
L → id | addType(id.entry, L.in)
Example

float x,y,z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in</td>
</tr>
<tr>
<td></td>
<td>addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
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

float type

float in

dmy

float float

ent1

ent2

ent3

float float

float float

float float

float float
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 → L1, id         L1.in = L.in
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

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → E;</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>E → E₁ + T</td>
<td>E.val = E₁.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>T → T₁ * F</td>
<td>T.val = T₁.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>F → (E)</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>F → digit</td>
<td>F.val = digit.lexval</td>
</tr>
</tbody>
</table>
example
L-attributed grammars

- L-attributed attribute grammar when every attribute in a production $A \rightarrow X_1 \ldots X_n$ is
  - A synthesized attribute, or
  - An inherited attribute of $X_j$, $1 \leq j \leq n$ 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

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → B</td>
<td>B.ps = 10</td>
</tr>
<tr>
<td>B → B₁ B₂</td>
<td>B₁.ps = B.ps</td>
</tr>
<tr>
<td></td>
<td>B₂.ps = B.ps</td>
</tr>
<tr>
<td></td>
<td>B.ht = max(B₁.ht, B₂.ht)</td>
</tr>
<tr>
<td></td>
<td>B.dp = max(B₁.dp, B₂.dp)</td>
</tr>
<tr>
<td>B → B₁ sub B₂</td>
<td>B₁.ps = B.ps</td>
</tr>
<tr>
<td></td>
<td>B₂.ps = 0.7*B.ps</td>
</tr>
<tr>
<td></td>
<td>B.ht = max(B₁.ht, B₂.ht – 0.25*B.ps)</td>
</tr>
<tr>
<td></td>
<td>B.dp = max(B₁.dp, B₂.dp– 0.25*B.ps)</td>
</tr>
<tr>
<td>B → text</td>
<td>B.ht = getHt(B.ps, text.lexval)</td>
</tr>
<tr>
<td></td>
<td>B.dp = getDp(B.ps, text.lexval)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id := E</td>
<td>S.nptr = makeNode('assign', makeLeaf(id, id.place), E.nptr)</td>
</tr>
<tr>
<td>E → E1 + E2</td>
<td>E.nptr = makeNode('+', E1.nptr, E2.nptr)</td>
</tr>
<tr>
<td>E → E1 * E2</td>
<td>E.nptr = makeNode('*', E1.nptr, E2.nptr)</td>
</tr>
<tr>
<td>E → -E1</td>
<td>E.nptr = makeNode('uminus', E1.nptr)</td>
</tr>
<tr>
<td>E → (E1)</td>
<td>E.nptr = E1.nptr</td>
</tr>
<tr>
<td>E → id</td>
<td>E.nptr = makeLeaf(id, id.place)</td>
</tr>
</tbody>
</table>

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

\[ a = b \ast -c + b\ast -c \]

```
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

\[
\begin{align*}
    t_1 & := - c \\
    t_2 & := b \times t_1 \\
    t_3 & := - c \\
    t_4 & := b \times t_3 \\
    t_5 & := t_2 + t_4 \\
    a & := t_5
\end{align*}
\]
Three address code: example instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := y op z</td>
<td>assignment with binary operator</td>
</tr>
<tr>
<td>x := op y</td>
<td>assignment unary operator</td>
</tr>
<tr>
<td>x := y</td>
<td>assignment</td>
</tr>
<tr>
<td>x := &amp;y</td>
<td>assign address of y</td>
</tr>
<tr>
<td>x := *y</td>
<td>assignment from deref y</td>
</tr>
<tr>
<td>*x := y</td>
<td>assignment to deref x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto L</td>
<td>unconditional jump</td>
</tr>
<tr>
<td>if x relop y goto L</td>
<td>conditional jump</td>
</tr>
</tbody>
</table>
Array operations

- Are these 3AC operations?

\[
x := y[i]
\]

\[
t1 := &y \quad ; \quad t1 = \text{address-of } y \\
t2 := t1 + i \quad ; \quad t2 = \text{address of } y[i] \\
X := *t2 \quad ; \quad \text{value stored at } y[i]
\]

\[
x[i] := y
\]

\[
t1 := &x \quad ; \quad t1 = \text{address-of } x \\
t2 := t1 + i \quad ; \quad t2 = \text{address of } x[i] \\
*t2 := y \quad ; \quad \text{store through pointer}
\]
Three address code: example

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

```plaintext
i := 0                      ; assignment
L1: if i >= 10 goto L2      ; conditional jump
t0 := i*i                   ; address-of operation
t1 := &b
    t2 := t1 + i
    *t2 := t0
    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>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} := E$</td>
<td>$S.\text{code} := E.\text{code} | \text{gen(id.var '} := ' E.\text{var})$</td>
</tr>
</tbody>
</table>
| $E \rightarrow E_1 + E_2$ | $E.\text{var} := \text{freshVar}();$
| | $E.\text{code} = E_1.\text{code} \| E_2.\text{code} \| \text{gen(E.var '} := ' E_1.\text{var} '+' E_2.\text{var})$ |
| $E \rightarrow E_1 * E_2$ | $E.\text{var} := \text{freshVar}();$
| | $E.\text{code} = E_1.\text{code} \| E_2.\text{code} \| \text{gen(E.var '} := ' E_1.\text{var} '*' E_2.\text{var})$ |
| $E \rightarrow - E_1$ | $E.\text{var} := \text{freshVar}();$
| | $E.\text{code} = E_1.\text{code} \| \text{gen(E.var '} := ' \text{uminu' E_1.\text{var})$ |
| $E \rightarrow (E_1)$ | $E.\text{var} := E_1.\text{var}$
| | $E.\text{code} = '(' || E_1.\text{code} || ')$'$ |
| $E \rightarrow \text{id}$ | $E.\text{var} := \text{id.var}; E.\text{code} = ''$ |

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

```
assign

a

+

t2 = b*t1

E.var = t5
E.code = 't1 = -c

t2 = b*t1

E.var = t4
E.code = 't3 = -c
t4 = b*t3

t5 = t2*t4'

*  E.var = t2
E.code = 't1 = -c

t2 = b*t1

E.var = t1
E.code = 't1 = -c'

*  E.var = t4
E.code = 't3 = -c

t4 = b*t3

E.var = t3
E.code = 't3 = -c'

/  E.var = b
E.code = ''

b

uminus

c

E.var = c
E.code = ''

/  E.var = b
E.code = ''

b

uminus

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

\[ S \rightarrow \text{while } E \text{ do } S_1 \]

**simplified diagram**

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
</table>
| \[ S \rightarrow \text{while } E \text{ do } S_1 \] | 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 ':') |
Representing 3AC

- Quadruple \((\text{op}, \text{arg}_1, \text{arg}_2, \text{result})\)
- 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

\[
\begin{align*}
t_1 &= -c \\
t_2 &= b \times t_1 \\
t_3 &= -c \\
t_4 &= b \times t_3 \\
t_5 &= t_2 \times t_4 \\
a &= t_5
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>arg 1</th>
<th>arg 2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>uminus</td>
<td>c</td>
<td></td>
<td>(t_1)</td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
<td>b</td>
<td>(t_1)</td>
<td>(t_2)</td>
</tr>
<tr>
<td>(2)</td>
<td>uminus</td>
<td>c</td>
<td></td>
<td>(t_3)</td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
<td>b</td>
<td>(t_3)</td>
<td>(t_4)</td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
<td>(t_2)</td>
<td>(t_4)</td>
<td>(t_5)</td>
</tr>
<tr>
<td>(5)</td>
<td>:=</td>
<td>(t_5)</td>
<td></td>
<td>(a)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → D</td>
<td>{ offset := 0}</td>
</tr>
<tr>
<td>D → D D</td>
<td></td>
</tr>
<tr>
<td>D → T id;</td>
<td>{ enter(id.name, T.type, offset); offset += T.width }</td>
</tr>
<tr>
<td>T → integer</td>
<td>{ T.type := int; T.width = 4 }</td>
</tr>
<tr>
<td>T → float</td>
<td>{ T.type := float; T.width = 8 }</td>
</tr>
<tr>
<td>T → T1[num]</td>
<td>{ T.type = array (num.val,T1.Type); T.width = num.val * T1.width; }</td>
</tr>
<tr>
<td>T → *T1</td>
<td>{ T.type := pointer(T1.type); T.width = 4 }</td>
</tr>
</tbody>
</table>
Allocating Memory

enter(count, int, 0)
offset = offset + 4

enter(money, float, 4)
offset = offset + 4

T1: int
id.name = count
T1.type = int
T1.width = 4

T2: float
id.name = money
T2.type = float
T2.width = 4

T3: balances

T4: [num]
T4.num: int

T4.num: 42
\section*{Adjusting to bottom-up}

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \rightarrow M \ D )</td>
<td>[{\text{offset} := 0}]</td>
</tr>
<tr>
<td>( M \rightarrow \varepsilon )</td>
<td>{ enter(id.name, T.type, offset); offset += T.width }</td>
</tr>
<tr>
<td>( D \rightarrow D \ D )</td>
<td>( T \rightarrow \text{integer} ) { T.type := int; T.width = 4 }</td>
</tr>
<tr>
<td>( D \rightarrow T \ id; )</td>
<td>( T \rightarrow \text{float} ) { T.type := float; T.width = 8 }</td>
</tr>
<tr>
<td>( T \rightarrow T_1[num] )</td>
<td>( T \rightarrow *T_1 ) { T.type := pointer(T_1.type); T.width = 4 }</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id := E</td>
<td>{ p:= lookup(id.name); if p ≠ null then \textbf{emit}(p ':= ' E.var) else error }</td>
</tr>
<tr>
<td>E → E₁ op E₂</td>
<td>{ E.var := freshVar(); \textbf{emit}(E.var ':= ' E₁.var op E₂.var) }</td>
</tr>
<tr>
<td>E → - E₁</td>
<td>{ E.var := freshVar(); \textbf{emit}(E.var ':=' 'uminus' E₁.var) }</td>
</tr>
<tr>
<td>E → ( E₁)</td>
<td>{ E.var := E₁.var }</td>
</tr>
<tr>
<td>E → id</td>
<td>{ p:= lookup(id.name); if p ≠ null then E.var :=p else error }</td>
</tr>
</tbody>
</table>
Boolean Expressions

<table>
<thead>
<tr>
<th>production</th>
<th>semantic action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E₁ op E₂</td>
<td>{ E.var := freshVar(); <strong>emit</strong>(E.var ‘:=’ E₁.var op E₂.var) }</td>
</tr>
<tr>
<td>E → not E₁</td>
<td>{ E.var := freshVar(); <strong>emit</strong>(E.var ‘:=’ ‘not’ E₁.var) }</td>
</tr>
<tr>
<td>E → ( E₁ )</td>
<td>{ E.var := E₁.var }</td>
</tr>
<tr>
<td>E → true</td>
<td>{ E.var := freshVar(); <strong>emit</strong>(E.var ‘:=’ ‘1’) }</td>
</tr>
<tr>
<td>E → false</td>
<td>{ E.var := freshVar(); <strong>emit</strong>(E.var ‘:=’ ‘0’) }</td>
</tr>
</tbody>
</table>

- Represent true as 1, false as 0
- Wasteful representation, creating variables for true/false
### Boolean expressions via jumps

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow id_1 \text{ op } id_2$</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>E.var := freshVar();</td>
<td></td>
</tr>
<tr>
<td>emit('if' id1.var relop id2.var 'goto' nextStmt+2);</td>
<td></td>
</tr>
<tr>
<td>emit(E.var := '0');</td>
<td></td>
</tr>
<tr>
<td>emit('goto ' nextStmt + 1);</td>
<td></td>
</tr>
<tr>
<td>emit(E.var := '1')</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
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

- \((x \text{ and } y)\) is equivalent to if \(x\) then \(y\) else false;

- \((x \text{ or } y)\) is equivalent to if \(x\) then true else \(y\)
example

\[ a < b \text{ or } (c < d \text{ and } e < f) \]

**naive**

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: \( T_4 := T_2 \text{ and } T_3 \)
113: \( T_5 := T_1 \text{ and } T_4 \)

**Short circuit evaluation**

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:
More examples

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

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