THEORY OF COMPIILATION

LECTURE 06

INTERMEDIATE REPRESENTATION

You are here

Intermediate Representation

- "Neutral" representation between the front-end and the back-end
- Abstracts away details of the source language
- Abstracts 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 was created for C)

Intermediate Representation(s)

- Annotated abstract syntax tree
- Data dependence graph
- Three address code (3AC)
- ...
Example: Annotated AST

- new Node(op, operand1[, operand2]) – creates new node for unary/binary operator
- new Node('id', entry) – creates a leaf
- id.entry – pointer to symbol table

Production | Semantic Rule
---|---
S → id := E
E → E1 + E2
E → E1 * E2
E → – E1
E → ( E1 )
E → id

Example

```
a = b * -c + b
```

Three Address Code (3AC)

- Every instruction operates on (at most) 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 3AC

Three Address Code — Example

```
a = b * -c + b
```

Three Address Code: example instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a := y op z</td>
<td>assignment with binary operator</td>
</tr>
<tr>
<td>x := op y</td>
<td>assignment with unary operator</td>
</tr>
<tr>
<td>k := y</td>
<td>assignment (copy)</td>
</tr>
<tr>
<td>x := &amp;y</td>
<td>assign address of y</td>
</tr>
<tr>
<td>x := *y</td>
<td>assign value in address y (deref)</td>
</tr>
<tr>
<td>*a := y</td>
<td>assign into address a</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?

\[
x := y \{i\}
\]

Array O

\[
x[\{i\}] := y
\]

Three Address Code — Example

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

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

Example
Creating 3AC: Control Statements

- 3AC only supports conditional/unconditional jumps
  - Add labels to code
  - Store labels in attributes:
    - begin – marks beginning of code fragment
    - next – follows end of code fragment
- Helper function freshLabel() allocates a new unused label

Creating 3AC: control statements

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

production: semantic rule

\[ S \rightarrow \text{while } E \text{ do } S_1 \]
\[ S_1 \text{ begin } = \text{freshLabel}() \]
\[ S_1 \text{ next } = \text{freshLabel}() \]
\[ S_1 \text{ code } = (S_1 \text{ begin })' | E \text{ code } | (S_1 \text{ next })' \]

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)

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allocating Memory

- Type checking helped us guarantee correctness
- Also tells us
  - How much memory to 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

- Attribute “offset” with memory allocated before each declaration

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D \rightarrow T</td>
<td>T.width = 4</td>
</tr>
<tr>
<td>D \rightarrow T [num]</td>
<td>T.type = array(num.val, T1.Type); T.width = num.val \cdot T1.width</td>
</tr>
<tr>
<td>T \rightarrow D</td>
<td>T.type = pointer(T1.type); T.width = 4</td>
</tr>
<tr>
<td>T \rightarrow D</td>
<td>T.type = float; T.width = 4</td>
</tr>
<tr>
<td>T \rightarrow D</td>
<td>T.type = int; T.width = 4</td>
</tr>
<tr>
<td>D \rightarrow D, D2</td>
<td>D.offset = D2.offset + D.width</td>
</tr>
<tr>
<td>D \rightarrow D, D1</td>
<td>D.offset = D1.offset + D.width</td>
</tr>
<tr>
<td>D \rightarrow D</td>
<td>D.offset = 0</td>
</tr>
<tr>
<td>D \rightarrow D</td>
<td>D.offset = D1.offset; D.width = D1.width</td>
</tr>
<tr>
<td>D \rightarrow T</td>
<td>T.type = int; T.width = int</td>
</tr>
<tr>
<td>D \rightarrow T</td>
<td>T.type = float; T.width = float</td>
</tr>
<tr>
<td>D \rightarrow T</td>
<td>T.type = array(num.val, T1.Type); T.width = num.val \cdot T1.width</td>
</tr>
<tr>
<td>D \rightarrow T</td>
<td>T.type = pointer(T1.type); T.width = 4</td>
</tr>
</tbody>
</table>
Allocating Memory

- Global variable “offset” with memory allocated so far

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>offset = 0</td>
</tr>
<tr>
<td>0 → 0.0</td>
<td>offset = offset + 4</td>
</tr>
<tr>
<td>D → T.1</td>
<td>enter(&quot;money&quot;, T.type, offset); offset = T.width</td>
</tr>
<tr>
<td>f → f.0</td>
<td>Type = float, T.width = 4</td>
</tr>
<tr>
<td>f → f.1</td>
<td>Type = int, T.width = 4</td>
</tr>
<tr>
<td>f → f.2</td>
<td>Type = pointer(T.type), T.width = 4</td>
</tr>
</tbody>
</table>

Adjusting to bottom-up

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>offset = 0</td>
</tr>
<tr>
<td>0 → 0.0</td>
<td>offset = offset + 4</td>
</tr>
<tr>
<td>D → T.1</td>
<td>enter(&quot;money&quot;, T.type, offset); offset = T.width</td>
</tr>
<tr>
<td>f → f.0</td>
<td>Type = float, T.width = 4</td>
</tr>
<tr>
<td>f → f.1</td>
<td>Type = int, T.width = 4</td>
</tr>
<tr>
<td>f → f.2</td>
<td>Type = pointer(T.type), T.width = 4</td>
</tr>
</tbody>
</table>

Generating IR code

- Option 1: accumulate code in AST attributes
- Option 2 (incremental translation): emit IR code to a file during compilation
  -> Possible when for every production the code of the left-hand side is constructed from a concatenation of the code of the right-hand side in some fixed order

Expressions and Assignments

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → id</td>
<td>p = lookup(id.name); if p != null then emit(p ':=' E.var) else error</td>
</tr>
<tr>
<td>E → E1 op E2</td>
<td>E.var = freshVar(); emit(E.var ':=' E1.var op E2.var)</td>
</tr>
<tr>
<td>E → - E1</td>
<td>E.var = freshVar(); 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>

Semantic errors — these are not needed if we make a first pass of identification & type checking. But the two phases can be combined, and this is an example for that.
Boolean Expressions

- Production: \( E \)
  - Semantic action: \( E \rightarrow E_1 \text{ op } E_2 \)
    - \( E \) is replaced with freshVar
    - \( E \) is replaced with 'not' \( E_1 \)
    - \( E \) is replaced with \( (E_1) \)
    - \( E \) is replaced with \( \text{true} \)
    - \( E \) is replaced with \( \text{false} \)

Boolean Expressions via Jumps

- Production: \( E \)
  - Semantic action:
    - \( E \rightarrow id_1 \text{ op } id_2 \)
      - \( E \) is replaced with freshVar
      - Emit: 'if' \( id_1 \) \( \text{ var } \) \( \text{ op } \) \( id_2 \) \( \text{ var } \) 'goto' nextStmt+2)
      - Emit: \( E \) is replaced with '0')
      - Emit: 'goto' nextStmt+1)
      - Emit: \( E \) is replaced with '1')

Example

- Example:
  - \( a < b \) or \( c < d \) and \( e < f \)

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
Control Structures

For every Boolean expression B, we attach two attributes:

- 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

\[ P \rightarrow S \]
\[ \text{S.next = freshLabel();} \]
\[ \text{S.code = S.code} \]

\[ P \rightarrow S_1 S_2 \]
\[ S_1.next = S.next; \]
\[ S_2.next = S.next; \]
\[ S_1.code \mid (S_1.next ':') \mid S_2.code \]

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

(\Rightarrow \text{inherited attribute})

Control Structures: conditional

Production Semantic Action

\[ S \rightarrow \text{if } B \text{ then } S_1 \]
\[ B.trueLabel = freshLabel(); \]
\[ B.falseLabel = S.next; \]
\[ S_1.next = S.next; \]
\[ S_1.code \mid B.code \mid (B.trueLabel ':') \mid S_1.code \]

\[ S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2 \]
\[ B.trueLabel = freshLabel(); \]
\[ B.falseLabel = freshLabel(); \]
\[ S_1.next = S.next; \]
\[ S_2.next = S.next; \]
\[ S_1.code \mid B.code \mid (B.trueLabel ':') \mid S_1.code \]
\[ \mid (\text{goto } S_2) \mid (B.falseLabel ':') \mid S_2.code \]

Control Structures: conditional

Production Semantic Action

\[ S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2 \]
\[ B.trueLabel = freshLabel(); \]
\[ B.falseLabel = freshLabel(); \]
\[ S_1.next = S.next; \]
\[ S_2.next = S.next; \]
\[ S_1.code \mid B.code \mid (B.trueLabel ':') \mid S_1.code \]
\[ \mid (\text{goto } S_2) \mid (B.falseLabel ':') \mid S_2.code \]

\[ B.trueLabel: \]
\[ B.falseLabel: \]
\[ S.next: \]

\[ B.trueLabel = freshLabel(); \]
\[ B.falseLabel = freshLabel(); \]
\[ S_1.next = S.next; \]
\[ S_2.next = S.next; \]
\[ S_1.code \mid B.code \mid (B.trueLabel ':') \mid S_1.code \]
\[ \mid (\text{goto } S_2) \mid (B.falseLabel ':') \mid S_2.code \]

B.trueLabel

B.falseLabel

S.next

(\Rightarrow \text{inherited attribute})
### Boolean expressions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B \rightarrow B_1 \text{ or } B_2$</td>
<td>$B_1$.trueLabel = $B$.trueLabel; $B_1$.falseLabel = freshLabel(); $B_2$.trueLabel = $B$.trueLabel; $B_2$.falseLabel = $B$.falseLabel; $B$.code = $B_1$.code</td>
</tr>
<tr>
<td>$B \rightarrow B_1 \text{ and } B_2$</td>
<td>$B_1$.trueLabel = freshLabel(); $B_1$.falseLabel = $B$.falseLabel; $B_2$.trueLabel = $B$.trueLabel; $B_2$.falseLabel = $B$.falseLabel; $B$.code = $B_1$.code</td>
</tr>
<tr>
<td>$B \rightarrow \text{ not } B_1$</td>
<td>$B_1$.trueLabel = $B$.falseLabel; $B_1$.falseLabel = $B$.trueLabel; $B$.code = $B_1$.code;</td>
</tr>
<tr>
<td>$B \rightarrow (B_1)$</td>
<td>$B_1$.trueLabel = $B$.trueLabel; $B_1$.falseLabel = $B$.falseLabel; $B$.code = $B_1$.code;</td>
</tr>
<tr>
<td>$B \rightarrow \text{id}_1 \text{ relop id}_2$</td>
<td>$B$.code = ('if' $id_1$.var relop $id_2$.var 'goto' $B$.trueLabel)</td>
</tr>
<tr>
<td>$B \rightarrow \text{true}$</td>
<td>$B$.code = 'goto' $B$.trueLabel;</td>
</tr>
<tr>
<td>$B \rightarrow \text{false}$</td>
<td>$B$.code = 'goto' $B$.falseLabel;</td>
</tr>
</tbody>
</table>

### Example

![Example Diagram]

### 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
  - **Main idea**: keep labels symbolic until values are known, and then go back and patch them
- New synthesized attributes for $B$
  - $B$.trueList – list of jump instructions that will eventually get the label to jump to when $B$ is true.
  - $B$.falseList – list of jump instructions that will eventually get the label to jump to when $B$ is false.

### How can we determine the address of $B_1$.falseLabel?
- Only possible after we know the code of $B_1$ and all the code preceding $B_1$.

### 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
  - **Main idea**: keep labels symbolic until values are known, and then go back and patch them
- New synthesized attributes for $B$
  - $B$.trueList – list of jump instructions that will eventually get the label to jump to when $B$ is true.
  - $B$.falseList – list of jump instructions that will eventually get the label to jump to 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` is inherited), and not LL(1).

```
<table>
<thead>
<tr>
<th>Backpatching Boolean expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>production</strong></td>
</tr>
<tr>
<td>$B \rightarrow B_1 or M B_2$</td>
</tr>
<tr>
<td>$B \rightarrow B_1 and M B_2$</td>
</tr>
<tr>
<td>$B \rightarrow not B_1$</td>
</tr>
<tr>
<td>$B \rightarrow ( B_1$</td>
</tr>
<tr>
<td>$B \rightarrow id_1 relop id_2$</td>
</tr>
<tr>
<td>$B \rightarrow true$</td>
</tr>
<tr>
<td>$B \rightarrow false$</td>
</tr>
<tr>
<td>$M \rightarrow \epsilon$</td>
</tr>
</tbody>
</table>
```

Marker

- `{ M.instr = nextInstr; }
- Use $M$ to obtain the address just before $B_2$ code starts being generated

Example

```
B \rightarrow [addr]
- create a list of instructions containing addr

p_1 ++ p_2 -- concatenate the lists pointed to by p_1 and p_2, returns a pointer to the new list

backpatch(p, addr) -- inserts addr as the target label for each of the instructions in the list pointed to by p
```

Example

```
x < 100  or  (x > 200  and  x != y)
```

```
B \rightarrow [nextInstr]
B.trueList = [nextInstr];
B.falseList = [nextInstr+1];
emit('if ' id_1.var relop id_2.var 'goto _');
emit('goto _');
```
Example

\[ x < 100 \text{ or } (x > 200 \text{ and } x \neq y) \]

Before backpatching

\[ B \rightarrow \text{falseList}, \text{trueList}; \]
\[ B.t = \{100\}; \]
\[ B.f = \{101\}; \]
\[ M.i = 102 \]
\[ 100: \text{if } x < 100 \text{ goto } 102 \]
\[ 101: \text{goto } _{} \]

After backpatching by the semantics of

\[ B \rightarrow \text{falseList}; B \rightarrow \text{trueList}; \]
\[ B.t = \{100,104\}; \]
\[ B.f = \{103,105\}; \]
\[ M.i = 102 \]
\[ 102: \text{if } x > 200 \text{ goto } 104 \]
\[ 103: \text{goto } _{} \]

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
float initial, rate;

position = initial +
rate * 60

\[
\text{float } \text{initial}, \text{ rate;}
\]
\[
\text{position } = \text{initial } + \text{rate} \times 60
\]
Journey inside a compiler

Optimized
\[ t_1 = \text{id3} \times 60.0 \]
\[ \text{id1} = \text{id2} + t_1 \]

Assembly
LDI \text{R2}, \text{id3}
MULF \text{R2}, \text{R2}, \#60.0
LDI \text{R1}, \text{id2}
ADDF \text{R1}, \text{R1}, \text{R2}
STF \text{id1}, \text{R1}

Coming Up