**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

<table>
<thead>
<tr>
<th>Production</th>
<th>Annotated AST</th>
</tr>
</thead>
<tbody>
<tr>
<td>E = id := E</td>
<td>E.op = new Node('assign', new Node('id', id.entry), E.eptr)</td>
</tr>
<tr>
<td>E = E₁ + E₂</td>
<td>E.op = new Node('+', E₁.nptr, E₂.nptr)</td>
</tr>
<tr>
<td>E = E₁ * E₂</td>
<td>E.op = new Node('*', E₁.nptr, E₂.nptr)</td>
</tr>
<tr>
<td>E = – E₁</td>
<td>E.op = new Node('uminus', E₁.nptr)</td>
</tr>
<tr>
<td>E = (E₁)</td>
<td>E.op = E₁.nptr</td>
</tr>
<tr>
<td>E = id</td>
<td>E.op = new Node('id', id.entry)</td>
</tr>
</tbody>
</table>

- 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

Example

**Array representation**

<table>
<thead>
<tr>
<th>i</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>-c</td>
</tr>
<tr>
<td>3</td>
<td>b * t</td>
</tr>
<tr>
<td>4</td>
<td>t</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>6</td>
<td>t</td>
</tr>
<tr>
<td>7</td>
<td>-c</td>
</tr>
<tr>
<td>8</td>
<td>b * t</td>
</tr>
<tr>
<td>9</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>assign</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>AST</th>
<th>3AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>t₁ := – c</td>
</tr>
<tr>
<td>t₂ := b * t₁</td>
<td></td>
</tr>
<tr>
<td>t₃ := – c</td>
<td></td>
</tr>
<tr>
<td>t₄ := b * t₂</td>
<td></td>
</tr>
<tr>
<td>t₅ := t₂ + t₄</td>
<td></td>
</tr>
<tr>
<td>a := t₅</td>
<td></td>
</tr>
</tbody>
</table>
Three Address Code: example instructions

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

Array Operations

- Are these 3AC operations?

```
x := y[i]
```

```
S1 := ky  ; S1 = address of y
S2 := k1 + i  ; S2 = address of y[i]
S3 := *S2  ; load value from y[i]

x[i] := y
```

```
x := &y
```

```
S1 := k1 + i  ; S1 = address of y[i]
S2 := *S1  ; load value from y[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
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

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)

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>Declarative</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → D</td>
<td>D.offset = 0</td>
</tr>
<tr>
<td>D → D, D_1</td>
<td>D.offset = D.offset + D_1.width; D.width = D_1.width</td>
</tr>
<tr>
<td>D → T.id</td>
<td>enter(id.name, Type, D.offset); D.width = T.width</td>
</tr>
<tr>
<td>T → Inst</td>
<td>Type = Inst; T.width = 4</td>
</tr>
<tr>
<td>T → T[expr]</td>
<td>Type = array(T.type, T.width); T.width = T.width + 4</td>
</tr>
<tr>
<td>T → T1</td>
<td>Type = pointer(T.type); T.width = 4</td>
</tr>
</tbody>
</table>

Allocating Memory

- Allocating memory for variables

Adjusting to Bottom-up

- Global variable “offset” with memory allocated so far

<table>
<thead>
<tr>
<th>Production</th>
<th>Declarative</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → M</td>
<td>M.offset = 0</td>
</tr>
<tr>
<td>M → ε</td>
<td></td>
</tr>
<tr>
<td>D → D</td>
<td>D.offset = D.offset + D.width</td>
</tr>
<tr>
<td>D → T.id</td>
<td>enter(id.name, Type, offset); offset += T.width</td>
</tr>
<tr>
<td>T → Inst</td>
<td>Type = Inst; T.width = 4</td>
</tr>
<tr>
<td>T → T1[expr]</td>
<td>Type = array(T.type, T.width); T.width = T.width + 4</td>
</tr>
<tr>
<td>T → T1</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: 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><code>S → id := E</code></td>
<td>p := lookup(id).name; if p ≠ null then emit(p ':=' E.var) else error</td>
</tr>
<tr>
<td><code>E → E1 op E2</code></td>
<td>E.var := freshVar(); emit(E.var ':=' E1.var op E2.var)</td>
</tr>
<tr>
<td><code>E → – E1</code></td>
<td>E.var := freshVar(); emit(E.var ':=' 'uminus' E1.var)</td>
</tr>
<tr>
<td><code>E → (E1)</code></td>
<td>E.var := E1.var</td>
</tr>
<tr>
<td><code>E → id</code></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 passes can be combined, and this is an example for that.

Boolean Expressions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E → id</code></td>
<td>p := lookup(id.name); if p ≠ null then E.var := p else error</td>
</tr>
<tr>
<td><code>E → E1 op E2</code></td>
<td>E.var := freshVar(); emit(E.var ':=' E1.var op E2.var)</td>
</tr>
<tr>
<td><code>E → not E1</code></td>
<td>E.var := freshVar(); emit(E.var ':=' 'not' E1.var)</td>
</tr>
<tr>
<td><code>E → (E1)</code></td>
<td>E.var := E1.var</td>
</tr>
<tr>
<td><code>E → true</code></td>
<td>E.var := freshVar(); emit(E.var ':=' '1')</td>
</tr>
<tr>
<td><code>E → false</code></td>
<td>E.var := freshVar(); emit(E.var ':=' '0')</td>
</tr>
</tbody>
</table>

op ∈ {'+', '*'}

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
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: conditional

production | semantic action
---|---
\( S \rightarrow \text{if } B \text{ then } S_1 \) | \( B \).trueLabel = freshLabel();
\( S_1 \).next = S.next; \( S_1 \).code = \( B \).code \( \mid \) \( B \).trueLabel ': \( S_1 \).code
\( S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2 \) | \( B \).falseLabel = freshLabel();
\( S_1 \).next = \( B \).trueLabel; \( S_2 \).next = S.next;
\( S_{\text{code}} = B \).code \( \mid \) \( B \).falseLabel ': \( S_2 \).code \( \mid \) (\( B \).trueLabel ': \( S_1 \).code) \( \mid \) (\( B \).falseLabel ': \( S_2 \).code)

B.trueLabel:
B.falseLabel:
S.next:

---

Control Structures: next

production | semantic action
---|---
P := S | S.next = freshLabel();
S := \( S_1, S_2 \) | \( S_{\text{code}} = S_1 \).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.
(\( = \) inherited attribute)
Boolean expressions

production semantic action
B → B1 or B2
B1.trueLabel = B.trueLabel;
B1.falseLabel = freshLabel();
B.code = B1.code | (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 | (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 = ('if' id1.var relop id2.var 'goto' B.trueLabel) | ('goto' B.falseLabel);
B → true
B.code = 'goto' B.trueLabel
B → false
B.code = 'goto' B.falseLabel

Example

Computing labels

• We can compute the values for the labels but it would require more than one pass on the AST (and/or IR).

• 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 (unresolved) 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).

Backpatching

- `[addr]` – create a list of instructions containing a single element, `addr`
- `p1 ++ p2` – concatenate the lists pointed to by `p1` and `p2`, returns a new list
- `backpatch(p, addr)` – inserts `addr` as the target label for each of the goto instructions in the list of locations `p`

### Backpatching Boolean expressions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>B → B1 or M B2</code></td>
<td><code>backpatch(B1.falseList, M.instr); B.trueList = B1.trueList ++ B2.trueList; B.falseList = B2.falseList;</code></td>
</tr>
<tr>
<td><code>B → B1 and M B2</code></td>
<td><code>backpatch(B1.trueList, M.instr); B.trueList = B2.trueList; B.falseList = B1.falseList ++ B2.falseList;</code></td>
</tr>
<tr>
<td><code>B → not B1</code></td>
<td><code>B.trueList = B1.falseList; B.falseList = B1.trueList;</code></td>
</tr>
<tr>
<td><code>B → (B1)</code></td>
<td><code>B.trueList = B1.trueList; B.falseList = B1.falseList;</code></td>
</tr>
<tr>
<td><code>B → id</code></td>
<td><code>B.trueList = [nextInstr]; B.falseList = [nextInstr+1]; emit('if' id.var relop id2.var 'goto _'); emit('goto _');</code></td>
</tr>
<tr>
<td><code>M → ε</code></td>
<td><code>M.instr = nextInstr;</code></td>
</tr>
</tbody>
</table>

Marker

- `{ M.Instr = nextInstr; }`
- Use `M` to obtain the address just before `B2` code starts being generated

Example

```
if x < 100 goto _
if x > 200 goto _
if x != y goto _
```
Example

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

Example

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

Example

- Notice that the rule for not neither has to emit nor backpatch anything. Cool, right?

Backpatching for Statements

- 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

Recap
float initial, rate;
position = initial + rate * 60

AST

Intermediate Representation

Optimized

\[ t_1 = \text{inttofloat}(60) \]
\[ t_2 = \text{id}_3 \times t_1 \]
\[ t_3 = \text{id}_2 + t_2 \]
\[ \text{id}_1 = t_3 \]
Journey inside a compiler

Optimized
\[
\begin{align*}
    t_3 &= id_3 \times 60.0 \\
    id_1 &= id_2 + t_1
\end{align*}
\]

Assembly
\[
\begin{align*}
    \text{LDF} & \quad R2, \quad id_3 \\
    \text{MULF} & \quad R2, \quad R2, \quad #60.0 \\
    \text{LDF} & \quad R1, \quad id_2 \\
    \text{ADDF} & \quad R1, \quad R1, \quad R2 \\
    \text{STF} & \quad id_1, \quad R1
\end{align*}
\]

Coming Up