Announcements

- Be sure to go to recitation today - Amy has important feedback.
- We will institute team meetings with me to facilitate communication.
Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
Intermediate Representations
Directed Acyclic Graph (DAG)

- Similar to a syntax tree
- No repeated nodes: structure sharing
Ex. 6.1 [p 359]

\[ a + a \times (b - c) + (b - c) \times d \]
Ex. 6.1 [p 359]

\[ a + a \times (b - c) + (b - c) \times d \]
Ex. 6.1 [p 359]

\[ a + a \ast (b - c) + (b - c) \ast d \]
Revisiting 6.1

see construction steps in figure 6.5 [p. 360]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>id</td>
<td>→ to ST entry for a</td>
</tr>
<tr>
<td>2</td>
<td>id</td>
<td>→ to ST entry for b</td>
</tr>
<tr>
<td>3</td>
<td>id</td>
<td>→ to ST entry for c</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>id</td>
<td>→ to ST entry for d</td>
</tr>
<tr>
<td>8</td>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>+</td>
<td>6</td>
</tr>
</tbody>
</table>
Motivating three-address code

- The DAG does not say anything about how the computation should be carried out.

- For example, there could be one instruction to do this computation:

  \[ t_1 = x + y \times z \]
Motivating three-address code

- In three-address code instructions can have no more than one operator on the right of an assignment.

- \( x + y * z \) must be broken into two instructions:

\[
\begin{align*}
  t_1 &= y * z \\
  t_2 &= x + t_1
\end{align*}
\]
Three-address code representation

"Three-address code is a linearized representation of a DAG in which explicit names correspond to the interior nodes of the graph." [p. 363]

\[ t_1 = b - c \]
\[ t_2 = a \times t_1 \]
\[ t_3 = a + t_2 \]
\[ t_4 = t_1 \times d \]
\[ t_5 = t_3 + t_4 \]
Three address code instructions
(see 6.2.1, pages 364-5)

1. \( x = y \text{ op } z \)
2. \( x = \text{ op } y \)  \hspace{1cm} (treat i2r and r2i as unary ops)
3. \( x = y \)
4. \( \text{goto L} \)
5. \( \text{if } x \text{ goto L / ifFalse x goto L} \)
6. \( \text{if } x \text{ relop y goto L} \)
7. function calls:
   - param \( x \)
   - call \( p, n \)
   - \( y = \text{call } p \)
   - return \( y \)
8. \( x = y[i] \text{ and } x[i] = y \)
9. \( x = &y, x = *y, *x = y \)
Representation options

"The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure."

[p. 366]
Quadruples

Instructions have four fields: op, arg1, arg2, result

Example: \( t_3 = a + t_2 \) is represented as

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>a</td>
<td>( t_2 )</td>
<td>( t_3 )</td>
</tr>
</tbody>
</table>

Example: \( t_4 = -c \) is represented as

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>minus</td>
<td>c</td>
<td></td>
<td>( t_4 )</td>
</tr>
</tbody>
</table>
Quadruples

Identifiers would be pointers to symbol table entries. Compiler-introduced temporaries can be added to the symbol table.

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>→ entry for a</td>
<td>→ entry for $t_2$</td>
<td>→ entry for $t_3$</td>
</tr>
</tbody>
</table>
Triples

Instructions have three fields: op, arg1, arg2

Example:
\[ t_2 = \ldots \]
\[ t_3 = a + t_2 \] is represented as

<table>
<thead>
<tr>
<th>line</th>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>computation of ( t_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>a</td>
<td>(5)</td>
</tr>
</tbody>
</table>
Indirect triples

Because order matters (due to embedded references instead of explicit variables) it is more challenging to rearrange instructions with triples than with quadruples.

Indirect triples allow for easier reordering (see page 369).
Static Single Assignment (SSA)
an additional constraint on the three address code

1) Each variable is assigned to exactly once.

\[
\begin{align*}
x &= r + 1 \\
y &= s \times 2 \\
x &= 2 \times x + y \\
y &= y + 1
\end{align*}
\]

\[
\begin{align*}
x_1 &= r + 1 \\
y_1 &= s \times 2 \\
x_2 &= 2 \times x_1 + y_1 \\
y_2 &= y_1 + 1
\end{align*}
\]
Static Single Assignment (SSA)  
an additional constraint on the three address code

1) Each variable is assigned to exactly once.

2) Need $\phi$ function to merge split variables:

   if (e) then { $x = a$ } else { $x = b$ }  
y = $x$

With SSA:

   if (e) then { $x_1 = a$ } else { $x_2 = b$ }  
y = $\phi( x_1, x_2 )$
In $y = \phi(x_1, x_2)$ simply let $y$, $x_1$ and $x_2$ be bound to the same address.
Type equivalence

Name equivalence: two types are equivalent if and only if they have the same name.

Structural equivalence: two types are equivalent if and only if they have the same structure. A type is structurally equivalent to itself (i.e. int is both name equivalent and structurally equivalent to int)
int x = 3;
int y = 5;
int z = x * y;

The type of z is int.
The type of x * y is int.
The names of the types are the same, so the assignment is legal.
Structural equivalence

```c
struct S { int v; double w; };
struct T { int v; double w; };

int main() {
    struct S x;
    x.v = 1; x.w = 4.5;
    struct T y;
    x = y;
    return 0;
}
```

Under name equivalence the assignment is disallowed.

Under structural equivalence the assignment is permitted.

What does C do?

Under name equivalence the assignment is disallowed.

Under structural equivalence the assignment is permitted.
C does not allow the assignment

bash-3.2$ gcc type.c
type.c:9:5: error: assigning to 'struct S' from incompatible type 'struct T'
    x = y;
   ^ ~
1 error generated.
Structural equivalence

```c
struct S { int v; double w; };
struct T { int a; double b; };

int main() {
    struct S x;
    x.v = 1; x.w = 4.5;
    struct T y;
    x = y;
    return 0;
}
```

Should this be allowed?

Types and order of fields align, but names differ
Consider...

```c
struct Rectangular { double x; double y; }
struct Polar { double r; double theta; }

int main() {
    struct Rectangular p;
    p.x = 3.14; x.y = 3.14;
    struct Polar q;
    q = p;
    return 0;
}
```

Should this be allowed?
Interpretation matters

polar interpretation

rectangular interpretation
Our language
(use name equivalence)

- Primitive types: integer, real, Boolean, character, string
- User-defined types:
  - Record types have names
    - `type rec : ( real x y ; ) : ( x := 0 y := 0)`
  - Array types have names
    - `type arr : 2 -> string`
  - Function types have names
    - `type fun : ( real : x ) -> rec`
Recursive records

Recursive functions

A record type must allow a component to be of the same type as the type itself:

```
type Node: ( integer datum:=0 ; Node rest:=null )
```