Announcements

Homework 2 submissions due 3/17/23 @ 5:00
(my mistake - I put the wrong date in AutoLab)

Milestone 2 submissions due 3/15/23 @ 5:00
Submissions open 3/13/23 @ 5:00
No autograding

Homework 3 & Milestone 3 posted 3/16/23 @ 5:00

No class on Friday

No team meetings on Friday
Phases of a compiler

Figure 1.6, page 5 of text
Roadmap

We will revisit how the semantics of flow-of-control statements can be expressed in section 6.6.3 Flow-of-Control Statements.

At that point we will learn the backpatching approach, which you will implement in your compiler.
§6.3 Types and Declarations
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)
Name equivalence

```c
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

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;  
  y = x;  
  return 0;  
}

Under name equivalence the assignment is disallowed.

Under structural equivalence the assignment is permitted.

What does C do?

- Types, names and order of fields all align.
C does not allow the assignment

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

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;
    y = x;
    return 0;
}
Consider...

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

int main() {
    struct Rectangular p;
    p.x = 3.14; p.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)

- **built-in types:**
  - primitive types: integer, real, Boolean, character
  - non-primitive type: string

- **user-defined types:**
  - record types have names
    - type recType : [ real : x, y ]
  - array types have names
    - type arrType : 2 -> string
  - function types have names
    - type funType : real -> recType
Recursive records

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

```plaintext
type Node: [ integer datum ; Node rest ]
```
Recursive records

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

```plaintext
type Node: [ integer datum ; Node rest ]
```

Be careful how you process declaration: you need to ensure that the second occurrence of Node does not trigger an undefined name.
Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
IR → IR

machine independent optimizations

machine dependent optimizations

→ target 1

→ target 2

→ target n
HLL

Milestone 3

IR

target 1

Milestone 4

machine independent optimizations

Milestone 4 (maybe some simple ones)
Intermediate Representations
Ex. 6.1 [p 359]

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

\[ a + a \times (b - c) + (b - c) \times d \]
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]

\[a + a \times (b - c) + (b - c) \times d\]

Things can be more complicated if expressions have side effects.
<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $E \rightarrow E_1 + T$</td>
<td>$E$.node = new Node('+', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>2 $E \rightarrow E_1 - T$</td>
<td>$E$.node = new Node('-', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>3 $E \rightarrow E_1 \times T$</td>
<td>$E$.node = new Node('*', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>4 $E \rightarrow T$</td>
<td>$E$.node = $T$.node</td>
</tr>
<tr>
<td>5 $T \rightarrow (E)$</td>
<td>$T$.node = $E$.node</td>
</tr>
<tr>
<td>6 $T \rightarrow \text{id}$</td>
<td>$T$.node = new Leaf(id, id.entry)</td>
</tr>
<tr>
<td>7 $T \rightarrow \text{num}$</td>
<td>$T$.node = new Leaf(num, num.val)</td>
</tr>
</tbody>
</table>

Figure 6.4 in text (p. 360), corrected according to errata sheet.
SDT

Tree or DAG

- SDT produces a tree if each call to Node creates a new tree node.

- SDT produces a DAG if for each call to Node there is a check whether this node already exists, and if so it returns a reference to the existing node rather than returning a new node.
Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1 = \text{Leaf(id, entry-a)}$</td>
<td></td>
</tr>
<tr>
<td>$p_2 = \text{Leaf(id, entry-a)} = p_1$</td>
<td></td>
</tr>
<tr>
<td>$p_3 = \text{Leaf(id, entry-b)}$</td>
<td></td>
</tr>
<tr>
<td>$p_4 = \text{Leaf(id, entry-c)}$</td>
<td></td>
</tr>
<tr>
<td>$p_5 = \text{Node('-',p}_3,p_4)$</td>
<td></td>
</tr>
<tr>
<td>$p_6 = \text{Node('*',p}_1,p_5)$</td>
<td></td>
</tr>
<tr>
<td>$p_7 = \text{Node('+',p}_1,p_6)$</td>
<td></td>
</tr>
<tr>
<td>$p_8 = \text{Leaf(id, entry-b)} = p_3$</td>
<td></td>
</tr>
<tr>
<td>$p_9 = \text{Leaf(id, entry-c)} = p_4$</td>
<td></td>
</tr>
<tr>
<td>$p_{10} = \text{Node('-',p}_3,p_4) = p_5$</td>
<td></td>
</tr>
<tr>
<td>$p_{11} = \text{Leaf(id, entry-d)}$</td>
<td></td>
</tr>
<tr>
<td>$p_{12} = \text{Node('*',p}<em>5,p</em>{11})$</td>
<td></td>
</tr>
<tr>
<td>$p_{13} = \text{Node('+',p}<em>7,p</em>{12})$</td>
<td></td>
</tr>
</tbody>
</table>
Value-number method
Algorithm 6.3 [p. 361]

- Input: label $op$, node $l$, node $r$
- Output: The value number of a node in the array with signature <$op$,l,$r$>
- Method: Search the array for a node $M$ with signature <$op$,l,$r$>. If there is such a node, return the value number of $M$. If not, create in the array a new node $N$ with signature <$op$,l,$r$> and return its value number.
Algorithm 6.3 [p. 361]

Input: label $op$, node $l$, node $r$

Output: The value number of a node in the array with signature $<op,l,r>$

Method: Search the array for a node $M$ with signature $<op,l,r>$. If there is such a node, return the value number of $M$. If not, create in the array a new node $N$ with signature $<op,l,r>$ and return its value number.

Can use hash table for efficiency.
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>
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:
  \[ t_1 = x + y \cdot z \]
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:

\[ t_1 = y * z \]
\[ t_2 = x + t_1 \]
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]

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

1. \( x = y \text{ op } z \)
2. \( x = \text{ op } y \)  
   (treat i2r and r2i as unary ops)
3. \( x = y \)
4. goto L
5. if x goto L / ifFalse x goto L
6. if x relop y goto L
7. function calls:
   - param x
   - call p, n
   - y = call p
   - return y
8. \( x = y[i] \text{ and } x[i] = y \)
9. \( x = &y, x = *y, *x = y \)
"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:
  \textit{op}, arg1, arg2, result

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

\[
\begin{array}{|c|c|c|c|}
\hline
\text{op} & \text{arg1} & \text{arg2} & \text{result} \\
\hline
+ & a & t_2 & t_3 \\
\hline
\end{array}
\]

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

\[
\begin{array}{|c|c|c|c|}
\hline
\text{op} & \text{arg1} & \text{arg2} & \text{result} \\
\hline
\textbf{minus} & c & & t_4 \\
\hline
\end{array}
\]
Variables in representation

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 t₂</td>
<td>→ entry for t₃</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).
Indirect Triples

Instructions have three fields:
- op
- arg1
- arg2

Example:
- $t_2 = \ldots$
- $t_3 = a + t_2$

is represented as

<table>
<thead>
<tr>
<th>index</th>
<th>instruction</th>
<th>line</th>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>5</td>
<td>5</td>
<td>computation of $t_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>6</td>
<td>6</td>
<td>+</td>
<td>a</td>
<td>((72))</td>
</tr>
</tbody>
</table>

Rearranging instructions changes the instruction array contents, but the instructions themselves do not change.
Static Single Assignment (SSA)
an additional constraint on the three address code

"[SSA] is an intermediate representation that facilitates certain code optimizations." [p. 369]

1) Each variable is assigned to exactly once. Occurrences of the same variable are subscripted to make them unique.

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

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

1) Each variable is assigned to exactly once.

2) Need ϕ function to merge split variables:

\[
\text{if (e) then } \{ x = a \} \text{ else } \{ x = b \} \\
y = x
\]

With SSA:

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

- **Milestone 1:** char stream $\rightarrow$ LEXER $\rightarrow$ token stream

- **Milestone 2:** PARSER builds symbol table, checks for undefined or multiply defined names from token stream.

- **Milestone 3:** PARSER will also perform type checking and generate intermediate code.