CS415 Compilers
Intermediate representations

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Homework #7 is due on Friday, April 26
• Project 2 is due on Wednesday, April 24
• Sample solutions for Homeworks #5 and #6 have been posted
• Project 1 (code) grades have been posted; test cases will be posted soon on sakai/Resources
• Final exam: May 14, noon – 3:00pm, SEC-118 (our current room)

CONFLICTS?
If you have a conflict, please send me the details of your conflict:
class, email of instructor, time of scheduled exam
Intermediate Representations  
(EaC Chapter 5)

- Front end - produces an intermediate representation (IR)
- Middle end - transforms the IR into an equivalent IR that runs more efficiently
- Back end - transforms the IR into native code

- IR encodes the compiler's knowledge of the program
- Middle end usually consists of several passes
Intermediate Representations

- Decisions in IR design affect the speed and efficiency of the compiler

- Some important IR properties
  - Ease of generation
  - Ease of manipulation
  - Size
  - Level of abstraction

- The importance of different properties varies between compilers
  - Selecting an appropriate IR for a compiler is critical
Three major categories

• Structural
  → Graphically oriented
  → Heavily used in source-to-source translators
  → Tend to be large

• Linear
  → Pseudo-code for an abstract machine
  → Level of abstraction varies
  → Simple, compact data structures
  → Easier to rearrange

• Hybrid
  → Combination of graphs and linear code

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- Control-flow graph
• The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

• Two different representations of an array reference:

```plaintext
loadI 1 => r1
sub  r_j, r1 => r2
loadI 10 => r3
mult r_2, r_3 => r4
sub  r_i, r1 => r5
add  r_4, r_5 => r6
loadI @A => r7
Add  r_7, r_6 => r8
load  r_8 => r_{Aij}
```

High level AST:
Good for memory disambiguation

Low level linear code:
Good for address calculation
- Structural IRs are usually considered high-level
- Linear IRs are usually considered low-level
- Not necessarily true:

Low level AST

load

loadArray A,i,j

High level linear code

\[
\begin{align*}
&\text{Low level AST} \\
&\text{High level linear code}
\end{align*}
\]
An abstract syntax tree is the procedure’s parse tree with the nodes for most non-terminal nodes removed.

\[ \text{\textbf{x - 2 * y}} \]

- Can use linearized form of the tree
  - Easier to manipulate than pointers
    - in postfix form: \( x \ 2 \ y \ * \ - \)
    - in prefix form: \( - \ * \ 2 \ y \ x \)
- S-expressions are (essentially) ASTs (remember functional languages such as Scheme or Lisp!)
A directed acyclic graph (DAG) is an AST with a unique node for each value

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!
Stack Machine Code

Originally used for stack-based computers, now Java

- Example:
  \[ x - 2 \times y \] becomes
  
  ```
  push x
  push 2
  push y
  multiply
  subtract
  ```

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Useful where code is transmitted over slow communication links *(the net)*

Implicit names take up no space, where explicit ones do!
Several different representations of three address code

- In general, three address code has statements of the form:
  \[ x \leftarrow y \ op \ z \]
  With 1 operator (\( \text{op} \)) and, at most, 3 names (\( x, y, z \))

Example:

\[ z \leftarrow x \ - \ 2 \ * \ y \]

becomes

\[ t \leftarrow 2 \ * \ y \]
\[ z \leftarrow x \ - \ t \]

Advantages:

- Resembles many machines
- Introduces a new set of names
- Compact form
Naïve representation of three address code

- Table of $k \times 4$ small integers
- Simple record structure
- Easy to reorder
- Explicit names

RISC assembly code (not ILOC)

```
load r1, y
loadI r2, 2
mult r3, r2, r1
load r4, x
sub r5, r4, r3
```

Quadruples

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>loadI</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>load</td>
<td>y</td>
</tr>
<tr>
<td>(2)</td>
<td>loadI</td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
<td>(1)</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
<td>x</td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
<td>(4)</td>
</tr>
</tbody>
</table>

Implicit names take no space!
Control-flow Graph (CFG)

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example

```
if (x = y)

a ← 2
b ← 5

a ← 3
b ← 4

c ← a * b
```

Basic blocks — Maximal length sequences of straight-line code
### Static Single Assignment Form (SSA)

- **The main idea:** each name defined exactly once in program
- **Introduce** $\phi$-functions to make it work

#### Original

\[
x \leftarrow \ldots
\]
\[
y \leftarrow \ldots
\]
\[
\text{while (} x < k \text{)}
\]
\[
x \leftarrow x + 1
\]
\[
y \leftarrow y + x
\]

#### SSA-form

\[
x_0 \leftarrow \ldots
\]
\[
y_0 \leftarrow \ldots
\]
\[
\text{if (} x_0 > k \text{) goto next}
\]
\[
\text{loop:}
\]
\[
x_1 \leftarrow \phi(x_0,x_2)
\]
\[
y_1 \leftarrow \phi(y_0,y_2)
\]
\[
x_2 \leftarrow x_1 + 1
\]
\[
y_2 \leftarrow y_1 + x_2
\]
\[
\text{if (} x_2 < k \text{) goto loop}
\]
\[
\text{next:}
\]
\[
\ldots
\]

### Strengths of SSA-form

- Sharper analysis
- “minimal” $\phi$-functions placement is non-trivial
- (sometimes) faster algorithms
Work on the project!

Procedure abstraction