Intermediate Code & Runtime Storage Organization

Compiler Design

CSE 504
Intermediate Code

“Abstract” code generated from AST

Motivation for use: **Simplicity and Portability**

- Machine independent code.
- Enables common optimizations on intermediate code.
- Machine-dependent code optimizations postponed to last phase.
Intermediate Forms

- **Stack machine code:**
  Code for a “postfix” stack machine.

- **Two address code:**
  Code of the form “add $r_1, r_2$”

- **Three address code:**
  Code of the form “add $src_1, src_2, dest$”

**Quadruples and Triples:** Representations for three-address code.
Quadruples

Explicit representation of three-address code.
Example: \( a := a + b * -c; \)

<table>
<thead>
<tr>
<th>Instr</th>
<th>Operation</th>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>uminus</td>
<td>c</td>
<td></td>
<td>( t_1 )</td>
</tr>
<tr>
<td>(1)</td>
<td>mult</td>
<td>b</td>
<td>( t_1 )</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>(2)</td>
<td>add</td>
<td>a</td>
<td>( t_2 )</td>
<td>( t_3 )</td>
</tr>
<tr>
<td>(3)</td>
<td>move</td>
<td>( t_3 )</td>
<td></td>
<td>( a )</td>
</tr>
</tbody>
</table>
Representation of three-address code with implicit destination argument.

Example: \( a := a + b * -c; \)

<table>
<thead>
<tr>
<th>Instr</th>
<th>Operation</th>
<th>Arg 1</th>
<th>Arg 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>uminus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>mult</td>
<td>b</td>
<td>(0)</td>
</tr>
<tr>
<td>(2)</td>
<td>add</td>
<td>a</td>
<td>(1)</td>
</tr>
<tr>
<td>(3)</td>
<td>move</td>
<td>a</td>
<td>(2)</td>
</tr>
</tbody>
</table>
Static Single-Assignment Form is similar to three-address code, with the following differences:

1. Each assignment is to a distinct variable (single assignment)
2. Merge points of branches have a special $\phi$-function to combine values of variables computed on the different branches.

<table>
<thead>
<tr>
<th>Three-Address Code</th>
<th>SSA Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = a + b$</td>
<td>$a_1 = a + b$</td>
</tr>
<tr>
<td>$b = a + c$</td>
<td>$b_1 = a_1 + c$</td>
</tr>
<tr>
<td>$a = a \ast c$</td>
<td>$a_2 = a_1 \ast c$</td>
</tr>
<tr>
<td>$c = a + b$</td>
<td>$c_1 = a_2 + b_1$</td>
</tr>
</tbody>
</table>
if \((y < 0)\)
\[
x = -1;
\]
else
\[
x = 1;
\]
\[
y = x \ast y;
\]

<table>
<thead>
<tr>
<th>Three-Address Code</th>
<th>SSA Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (y &gt;= 0) goto L1 (x = -1) goto L2 L1: (x = 1) L2: (y = x \ast y)</td>
<td>if (y &gt;= 0) goto L1 (x_1 = -1) goto L2 L1: (x_2 = 1) L2: (x_3 = \phi(x_1, x_2)) (y_1 = x_3 \ast y)</td>
</tr>
</tbody>
</table>
Intermediate Forms

Choice depends on convenience of further processing

- Code for stack machines is simplest to generate for expressions.
- Quadruples are most general, permitting most optimizations including code motion.
- Triples permit optimizations such as *common subexpression elimination*, but code motion is difficult.
- SSA forms make several optimizations easier.
Runtime Storage Organization

Storage for code and data.

- **Code Area**: Procedures, functions, methods.
- **Static Data Area**: “Permanent” data with statically known size.
- **Stack**: Temporary Data with known lifetime.
- **Heap**: Temporary Data with unknown lifetime (dynamically allocated).
Issues in Storage Organization

- Recursion
- Block structure and nesting (*nested procedures*).
- Parameter passing (*by value, reference, name*).
- Higher order procedures (*procedures as parameters to other procedures*).
- Dynamic Storage Management (*malloc, free*).
Storage Areas

Storage Organization for a typical procedural language.
void qsort(int m, int n)
{
    int i;

    if (n > m) {
        i = part(m, n);
        qsort(m, i-1);
        qsort(i+1, n);
    }
}

Recursion
Activation Trees

```
qsort(1,9)
  part(1,9)
  qsort(1,3)
  qsort(5,9)
  part(1,3)
  qsort(1,0)
  qsort(2,3)
  part(5,9)
  qsort(5,5)
  qsort(7,9)
  part(2,3)
  qsort(2,1)
  qsort(3,3)
```
Activation Records

All information local to a single invocation of a procedure is kept in an Activation Record.

- Return Address
- Arguments
- Return Value
- Local variables
- Temporaries
- Other control information
Activation Records: An Example

- Local variables
- Low memory
- High memory
- Operand Area
- Old bp
- Old ep
- Argument 0
- Argument 1
- Argument n
- Operand Area (Temporaries)
- bp
- ep
- sp
- Compiler Design
- Abstract Machines
- CSE 504 15 / 18
Organizing Activation Records

Control information for accessing different areas in an activation record:

- **Base Pointer**: Beginning of activation record. Arguments are accessed as offsets from base pointer.
- **Environment Pointer**: Pointer to the most recent activation record. Usually a fixed offset from base pointer.
- **Stack Pointer**: Top of activation record stack. Temporaries are allocated on top of stack.
int m(int k)
{
    int i;

    i = k + 15 * n(3);
    return l(i);
}
Managing Activation Records (contd.)

_m:

    pushl %ebp
    movl %esp,%ebp

    .. code for m

    movl %ebp, %esp
    popl %ebp
    ret