Intermediate Representations

- Some compilers use the AST as the only intermediate representation. Optimizations (code improvements) are performed directly on the AST, and machine code is generated directly from the AST.

- The AST is OK for machine-independent optimizations, such as inlining (replacing a procedure call with the called procedure's code).

- The AST is a bit too high-level for machine code generation and machine-dependent optimizations.

- For this reason, some compilers generate a lower level (simpler, closer to machine code) representation from the AST. This representation is used during code generation and code optimization.

Intermediate Code I

Advantages of:

1. Fitting many front-ends to many back-ends,
2. Different development teams for front- and back-end,
3. Debugging is simplified,
4. Portable optimization.

Requirements:

1. Architecture independent,
2. Language independent,
3. Easy to generate,
4. Easy to optimize,
5. Easy to produce machine code from.

A representation which is both architecture and language independent is known as an UNCOL, a Universal Compiler Oriented Language.
Intermediate Code II

- UNCOL is the holy grail of compiler design – many have search for it, but no one has found it. Problems:
  1. Programming language semantics differ from one language to another,
- There are several different types of intermediate representations:
  1. Tree-Based.
  2. Graph-Based.
  3. Tuple-Based.
  4. Linear representations.
- All representations contain the same information. Some are easier to generate, some are easy to generate simple machine code from, some are easy to generate good code from.

Slide 3–4

Postfix Notation

Infix: \( b := (a \times 2) + (a \times 2) \)

Postfix: \( b\ a\ 2\ \times\ a\ 2\ \times\ +\ := \)

- Postfix notation is a parenthesis free notation for arithmetic expression. It is essentially a linearized representation of an abstract syntax tree.
- In postfix notation an operator appears after its operands.
- Very simple to generate, very compact, easy to generate straight-forward machine code from, difficult to generate good machine code from.

Slide 3–5

Tree & DAG Repr. I

- Trees make good intermediate representations. We can represent the program as a sequence of expression trees. Each assignment, procedure call, or jump becomes one individual tree in the forest.
- Common Subexpression Elimination (CSE): Even if the same (sub-) expression appears more than once in a procedure, we should only compute its value once, and save the result for future reference.
- One way of doing this is to build a graph representation, rather than a tree. In the following slides we see how the expression \( a \times 2 \) gets two subtrees in the tree representation and one subtree in the DAG representation.

Slide 3–6

Tree & DAG Repr. II

\( b := (a \times 2) + (a \times 2) \)

Linearized Tree:

<table>
<thead>
<tr>
<th>Nr</th>
<th>Op</th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ident</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mul</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>ident</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>mul</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>add</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>ident</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>assign</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

Slide 3–7
Tree & DAG Repr. III

\[
b := (a * 2) + (a * 2)
\]

assign

\[
b +
\]

a 2

Linearized DAG:

<table>
<thead>
<tr>
<th>Nr</th>
<th>OP</th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ident</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mul</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>add</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>ident</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>assign</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Three-Address Code I

- Another common representation is **three-address code**. It is akin to **assembly code**, but uses an infinite number of **temporaries** (registers) to store the results of operations.

- There are three common realizations of three-address code: **quadruples**, **triples** and **indirect triples**.

Types of 3-Addr Statements:

\[
\begin{array}{ll}
\text{x := y op z} & \text{Binary arithmetic or logical operation. Example: Mul, And.} \\
\text{x := op y} & \text{Unary arithmetic, conversion, or logical operation. Example: Abs, UnaryMinus, Float.} \\
\text{x := y} & \text{Copy statement.} \\
\text{goto L} & \text{Unconditional jump.}
\end{array}
\]

Three-Address Code II

- \text{if x relop y goto L} Conditional jump. \text{relop} is one of \langle,\rangle,\leq, \text{etc.}. If \text{x relop y} evaluates to \text{True}, then jump to label \text{L}. Otherwise continue with the next tuple.

- \text{param X \_ call P, n} Make \text{X} the next parameter; make a procedure call to \text{P} with \text{n} parameters.

- \text{x := y[i]} Indexed assignment. Set \text{x} to the value in the location \text{i} memory units beyond \text{y}.

- \text{x := ADDR(y)} Address assignment. Set \text{x} to the address of \text{y}.

- \text{x := IND(y)} Indirect assignment. Set \text{x} to the value stored at the address in \text{y}.

- \text{IND(x) := y} Indirect assignment. Set the memory location pointed to by \text{x} to the value held by \text{y}.

Three-Address Code III

- Many three-address statements (particularly those for binary arithmetic) consist of one operator and three addresses (identifiers or temporaries):

\[
b := (a * 2) + (a * 2)
\]

\[
t_1 := a \text{ mul } 2
\]

\[
t_2 := a \text{ mul } 2
\]

\[
t_3 := t_1 \text{ add } t_2
\]

\[
b := t_3
\]

- There are several ways of implementing three-address statements. They differ in the amount of space they require, how closely tied they are to the symbol table, and how easily they can be manipulated.

- During optimization we may want to move the three-address statements around.
Three-Address Code IV

- Quadruples can be implemented as an array of records with four fields. One field is the operator.
- The remaining three fields can be pointers to the symbol table nodes for the identifiers. In this case, literals and temporaries must be inserted into the symbol table.

\[ b := (a * 2) + (a * 2) \]

<table>
<thead>
<tr>
<th>Nr</th>
<th>Res</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>( t_1 )</td>
<td>mul</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(2)</td>
<td>( t_2 )</td>
<td>mul</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td>( t_3 )</td>
<td>add</td>
<td>( t_1 )</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>(4)</td>
<td>( t_1 )</td>
<td>assign</td>
<td>b</td>
<td>( t_3 )</td>
</tr>
</tbody>
</table>

Basic Blocks and Flow Graphs I

- We divide the intermediate code of each procedure into basic blocks. A basic block is a piece of straight line code, i.e. there are no jumps in or out of the middle of a block.
- The basic blocks within one procedure are organized as a flow graph.
- A flowgraph has
  - basic blocks \( B_1 \cdots B_n \) as nodes,
  - a directed edge \( B_1 \rightarrow B_2 \) if control can flow from \( B_1 \) to \( B_2 \).
- Code generation can be performed on a small or large piece of the flow graph at a time (small=easy, large=hard):
  - Local Within one basic block.
  - Global Within one procedure.
  - Inter-procedural Within one program.

Basic Blocks and Flow Graphs II

Basic Blocks and Flow Graphs III

Source Code:

\[
\begin{align*}
X &:= 20; \quad \text{WHILE } X < 10 \quad \text{DO} \\
& X := X-1; \ A[X] := 10; \\
& \quad \text{IF } X = 4 \quad \text{THEN } X := X - 2; \ \text{ENDIF;} \\
& \quad \text{ENDDO; } Y := X + 5;
\end{align*}
\]

Intermediate Code:

\[
\begin{align*}
(1) & \quad X := 20 \\
(2) & \quad \text{if } X \geq 4 \quad \text{goto } \quad \text{(7)} \\
(3) & \quad X := X-1 \\
(4) & \quad A[X] := 10 \\
(5) & \quad \text{if } X < 4 \quad \text{goto } \quad \text{(9)} \\
(6) & \quad X := X-2 \\
(7) & \quad \text{goto } \quad \text{(2)} \\
(8) & \quad Y := X + 5
\end{align*}
\]

Flow Graph:
Basic Blocks I

- How do we identify the basic blocks and build the flow graph?
- Assume that the input to the code generator is a list of tuples. How do we find the beginning and end of each basic block?

Algorithm:

1. First determine a set of leaders, the first tuple of basic blocks:
   (a) The first tuple is a leader.
   (b) Tuple L is a leader if there is a tuple \( \text{if ... goto L or goto L} \).
   (c) Tuple L is a leader if it immediately follows a tuple \( \text{if ... goto L or goto L} \).

2. A basic block consists of a leader and all the following tuples until the next leader.

---

Basic Blocks II

\[ P := 0; I := 1; \]

\[ \text{REPEAT} \]

\[ P := P + I; \]

\[ \text{IF } P > 60 \text{ THEN } P := 0; I := 5 \text{ ENDIF; } \]

\[ I := I * 2 + 1; \]

\[ \text{UNTIL } I > 20; \]

\[ K := P * 3 \]

Tuples:

1. \( P := 0 \) \( \leftarrow \) Leader (Rule 1.a)
2. \( I := 1 \)
3. \( P := P + I \) \( \leftarrow \) Leader (Rule 1.b)
4. \( \text{IF } P \leq 60 \text{ GOTO (7)} \)
5. \( P := 0 \) \( \leftarrow \) Leader (Rule 1.c)
6. \( I := 5 \)
7. \( T1 := I * 2 \) \( \leftarrow \) Leader (Rule 1.b)
8. \( I := T1 + 1 \)
9. \( \text{IF } I \leq 20 \text{ GOTO (3)} \)
10. \( K := P * 3 \) \( \leftarrow \) Leader (Rule 1.c)

---

Basic Blocks III

Block \( B_1 \): \([1] P:=0; [2] I:=1\]

Block \( B_2 \): \([3] P:=P+I; \]

\( \text{(4) IF } P \leq 60 \text{ GOTO } B_4 \)\]

Block \( B_3 \): \([5] P:=0; [6] I:=5\]

Block \( B_4 \): \([7] T1:=I*2; [8] I:=T1+1; \]

\( \text{(9) IF } I \leq 20 \text{ GOTO } B_2 \)\]

Block \( B_5 \): \([10] K:=P*3\]

```
Basic Blocks III

Block B1: [(1) P:=0; (2) I:=1]
Block B2: [(3) P:=P+I;
        (4) IF P<=60 GOTO B4]
Block B3: [(5) P:=0; (6) I:=5]
Block B4: [(7) T1:=I*2; (8) I:=T1+1;
        (9) IF I<=20 GOTO B2]
Block B5: [(10) K:=P*3]
```

---

Slide 3–16

Slide 3–17

Slide 3–18