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 \textit{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.

Slide 4–1

Intermediate Code

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 \textsc{UNCOL}, a Universal Compiler Oriented Language.

Mix-and-Match Compilers

Slide 4–2

Slide 4–3
**Intermediate Code...**

- 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.
- **IR** — Intermediate Representation.

---

**Postfix Notation**

Infix: \[ b := (a \ast 2) + (a \ast 2) \]

![Postfix Notation Diagram]

- 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.

---

**Tree & DAG Repr.**

- 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 \[ a \ast 2 \] gets two subtrees in the tree representation and one subtree in the DAG representation.

---

**Tree & DAG Repr...**

Infix: \[ b := (a \ast 2) + (a \ast 2) \]

![Tree & DAG Repr Diagram]

**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 4–4**

**Slide 4–5**

**Slide 4–6**

**Slide 4–7**
Tree & DAG Repr.

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

**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>

Sequence of Expression Trees

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

Three-Address Code

- 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:

- \( x := y \text{ op } z \): Binary arithmetic or logical operation. Example: \text{Mul}, \text{And}.
- \( x := \text{ op } y \): Unary arithmetic, conversion, or logical operation. Example: \text{Abs}, \text{UnaryMinus}, \text{Float}.
- \( x := y \): Copy statement.
- \text{goto } L\: Unconditional jump.

Three-Address Code...

- \text{if } x \text{ rel } y \text{ goto } L\: Conditional jump. \text{rel} is one of \(<\,\ge,\le\) etc. If \text{x rel y} evaluates to True, then jump to label \text{L}. Otherwise continue with the next tuple.
- \text{param } X; \text{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...**

- Many three-address statements (particularly those for binary arithmetic) consist of one operator and three addresses (identifiers or temporaries):
  \[ b := (a \times 2) + (a \times 2) \]
  \[ t_1 := a \quad \text{mul} \quad 2 \]
  \[ t_2 := a \quad \text{mul} \quad 2 \]
  \[ t_3 := t_1 \quad \text{add} \quad t_2 \]
  \[ b := t_3 \]

- 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...**

---

**Quadruples:**

- 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.

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

<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>(\text{mul})</td>
<td>(a)</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>(t_2)</td>
<td>(\text{mul})</td>
<td>(a)</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>(t_3)</td>
<td>(\text{add})</td>
<td>(t_1)</td>
<td>(t_2)</td>
</tr>
<tr>
<td>4</td>
<td>(b)</td>
<td>(\text{assign})</td>
<td>(t_3)</td>
<td></td>
</tr>
</tbody>
</table>

---

**Control Flow Graphs...**

- 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 (control) flow graph, or CFG.

- A flow-graph has
  - basic blocks \(B_1 \ldots B_n\) as nodes,
  - a directed edge \(B_1 \rightarrow B_2\) if control can flow from \(B_1\) to \(B_2\).
  - Special nodes [ENTER] and [EXIT] that are the source and sink of the graph.

- Inside each basic block can be any of the IRs we’ve seen: tuples, trees, DAGs, etc.
**Control Flow Graphs...**

Source Code:

```plaintext
X := 20; WHILE X < 10 DO
  X := X-1; A[X] := 10;
  IF X = 4 THEN X := X - 2; ENDIF;
ENDDO; Y := X + 5;
```

Intermediate Code:

(1) X := 20  
(2) X := X-1  
(3) A[X] := 10  
(4) IF X = 4 THEN X := X - 2; ENDIF;  
(5) IF X < 4 goto (7)  
(6) X := X-2  
(7) goto (2)  
(8) Y := X+5

**Flow Graph:**

```
ENTER

X := 20;  
B1

if x >= 10 goto B4  
B2

X := X-1;  
A[X] := 10;  
if X <> 4 goto B6  
B3

Y := X + 5;  
B4

X := X - 2;  
B5

goto B2  
B6

EXIT
```

Slide 4–16

**Basic Blocks**

- How do we identify the basic blocks and build the flow graph?
- Assume that the input 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 `if ... goto L` or `goto L`.
   (c) Tuple L is a leader if it immediately follows a tuple `if ... goto B` or `goto B`.
2. A basic block consists of a leader and all the following tuples until the next leader.

**Basic Blocks...**

P := 0; I := 1;
REPEAT
  P := P + I;
  IF P > 60 THEN P := 0; I := 5; ENDIF;
  I := I * 2 + 1;
UNTIL I > 20;
K := P * 3

**Tuples:**

(1) P := 0 ← Leader (Rule 1.a)
(2) I := 1
(3) P := P + I ← Leader (Rule 1.b)
(4) IF P <= 60 GOTO (7)
(5) P := 0 ← Leader (Rule 1.c)
(6) I := 5
(7) T1 := I * 2 ← Leader (Rule 1.b)
(8) I := T1 + 1
(9) IF I <= 20 GOTO (3)
(10) K := P * 3 ← Leader (Rule 1.c)

Slide 4–18

Slide 4–19
Basic Blocks...

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

Block $B_2$: [(3) P:=P+I;
(4) IF P<=60 GOTO $B_4$]

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

Block $B_4$: [(7) T1:=I*2; (8) I:=T1+1;
(9) IF I<=20 GOTO $B_2$]

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

1. The instruction following an instruction that can throw an exception is a leader. Note that exceptions can be thrown explicitly (throw in Java) or implicitly (null-pointer exception, for example).

2. Add an edge $u \rightarrow v$ if $u$ is a basic block that can throw an exception and $v$ is a handler block that could catch that exception.

CFGs and Exceptions

- Control-Flow analysis becomes much more difficult in the presence of exceptions.
- The algorithm for constructing a CFG must be amended:

3. Create a special basic block $\text{ABORT}$.

4. Add an edge $u \rightarrow \text{ABORT}$ if $u$ is a basic block that can throw an exception not caught by any handler in the procedure.

- In Java bytecode many instructions can throw exceptions. For this reason, a CFG constructed using the method above can be come very large, and the basic blocks very small.
- Various alternative CFG representations have been proposed to reduce the size of the graph. See the reference section.

Readings and References

- Louden:
  Intermediate Code 398–407
  Generating Intermediate Code 407–410
  Flow Graphs 475–477

- The Dragon book:
  Postfix notation 33
  Intermediate Languages 463–468, 470–473
  Basic Blocks 528–530
  Flow Graphs 532–534

- Jianjun Zhao, Analyzing Control Flow in Java Bytecode, citeseer.nj.nec.com/317884.html.

- Choi, Grove, Hind, Sarkar, Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs, citeseer.nj.nec.com/choi99efficient.html
Summary

- We use an intermediate representation of the program in order to isolate the back-end from the front-end.
- A high-level intermediate form makes the compiler retargetable (easily changed to generate code for another machine). It also makes code-generation difficult.
- A low-level intermediate form makes code-generation easy, but our compiler becomes more closely tied to a particular architecture.
- A basic block is a straight-line piece of code, with no jumps in or out except at the beginning and end.

Summary...

- A Control Flow Graph (CFG) is a graph whose nodes are basic blocks. There is an edge from basic block \( B_1 \) to \( B_2 \) if control can flow from \( B_1 \) to \( B_2 \).
- Control flows in and out of a CFG through two special nodes ENTER and EXIT.
- We construct a CFG for each procedure. This representation is used during code generation and optimization.
- Java bytecode is a stack-based IR. It was never intended as an UNCOL, but people have still built compilers for Ada, Scheme and other languages that generate Java bytecode. It is painful.
- Microsoft’s MSIL is the latest UNCOL attempt.

---

Exam Question

- Draw the control flow graph for the tuples:

```plaintext
int A[5], x, i, n;

for (i = 0; i < n; i++) {
    if (i <= 4) {
        if (i < 2)
            x = A[i];
        else
            while (i > 4) {
                x = x + 4 * A[i];
            }
        i = i + 2;
    } else
    x = x + 5;
}
```

---

Slide 4–24

Slide 4–25

Slide 4–26