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.

Intermediate Representations...

- 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.

Compiler Phases

We are here!
**UNCOL**

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

**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.

---

**Intermediate Code...**

- 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.

---

**Mix-and-Match Compilers**

---

- Ada
- Pascal
- Modula-2
- C++
- Sparc
- Mips
- 68000
- IBM/370
- Ada
- Mips
- Pascal
- 68k

---

**INTERMEDIATE REPRESENTATION**

---

- Ada
- Pascal
- Modula-2
- C++
- Sparc
- Mips
- 68000
- IBM/370
- Ada
- Mips
- Pascal
- 68k
Postfix Notation

- 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 Representation

- 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.
Building DAGs...

Example Expression:

\[(a + b) * c + \{(a + b) + e\} * (e + f)\] * \[(a + b) * c\]

Tree Representation:

```
+      *
|  +  / |
| a  b c |
```

Building DAGs...

\[(a + b) * c + \{(a + b) + e\} * (e + f)\] * \[(a + b) * c\]

DAG Representation:

```
+      *
|  +  / |
| a  b c |
```

Building DAGs

- From an expression/expression tree such as the one on the left we might generate the machine code (for some fictitious architecture) on the right:

```
a * (b + c)
```

```
LOAD b, r0
LOAD c, r1
ADD r0, r1, r2
LOAD a, r3
MUL r2, r3, r4
```

Building DAGs

- Can we generate better code from a DAG than a tree?
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:

- \( x := y \ op \ z \) Binary arithmetic or logical operation. Example: Mul, And.
- \( x := op \ y \) Unary arithmetic, conversion, or logical operation. Example: Abs, UnaryMinus, Float.
- \( x := y \) Copy statement.

Building DAGs...

- Generating machine code from the tree yields 21 instructions.

<table>
<thead>
<tr>
<th>Code from Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD a, r0 ; a</td>
</tr>
<tr>
<td>LOAD b, r1 ; b</td>
</tr>
<tr>
<td>ADD r0, r1, r2 ; a + b</td>
</tr>
<tr>
<td>LOAD c, r0 ; c</td>
</tr>
<tr>
<td>MUL r0, r2, r3 ; ((a + b) * c)</td>
</tr>
<tr>
<td>LOAD a, r0 ; a</td>
</tr>
<tr>
<td>LOAD b, r1 ; b</td>
</tr>
<tr>
<td>ADD r0, r1, r2 ; a + b</td>
</tr>
<tr>
<td>LOAD e, r0 ; e</td>
</tr>
</tbody>
</table>

Three-Address Code...

- \( \text{goto } L \) Unconditional jump.

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

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

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

Building DAGs...

- Generating machine code from the DAG yields only 12 instructions.

<table>
<thead>
<tr>
<th>Code from DAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD a, r0 ; a</td>
</tr>
<tr>
<td>LOAD b, r1 ; b</td>
</tr>
<tr>
<td>ADD r0, r1, r2 ; a + b</td>
</tr>
<tr>
<td>LOAD c, r0 ; c</td>
</tr>
<tr>
<td>MUL r0, r2, r3 ; ((a + b) * c)</td>
</tr>
<tr>
<td>MUL r4, r0, r4</td>
</tr>
<tr>
<td>MUL r0, r3, r0</td>
</tr>
</tbody>
</table>
Three-Address Code...

- 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.

<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>t₁</td>
<td>mul</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(2)</td>
<td>t₂</td>
<td>mul</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td>t₃</td>
<td>add</td>
<td>t₁</td>
<td>t₂</td>
</tr>
<tr>
<td>(4)</td>
<td>b</td>
<td>assign</td>
<td>t₃</td>
<td></td>
</tr>
</tbody>
</table>

Three-Address Code...

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

```
b := (a * 2) + (a * 2)
```

```
t₁ := a mul 2
t₂ := a mul 2
t₃ := t₁ add t₂
b := t₃
```
Three-Address Code — Triples

- Triples are similar to quadruples, but save some space.
- Instead of each three-address statement having an explicit result field, we let the statement itself represent the result.
- We don’t have to insert temporaries into the symbol table.

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

<table>
<thead>
<tr>
<th>Nr</th>
<th>Op</th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>mul</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(2)</td>
<td>mul</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td>add</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>(4)</td>
<td>assign</td>
<td>b</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Three-Address Code — Indirect Triples

- One problem with triples (“The Trouble With Triples?”) is that they cannot be moved around. We may want to do this during optimization. We can fix this by adding a level of indirection, an array of pointers to the “real” triples.

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

<table>
<thead>
<tr>
<th>Abs</th>
<th>Real</th>
<th>Nr</th>
<th>Op</th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(10)</td>
<td>(11)</td>
<td>mul</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(2)</td>
<td>(11)</td>
<td>(12)</td>
<td>mul</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td>(12)</td>
<td>(13)</td>
<td>add</td>
<td>(11)</td>
<td>(12)</td>
</tr>
<tr>
<td>(4)</td>
<td>(13)</td>
<td>(14)</td>
<td>:=</td>
<td>b</td>
<td>(13)</td>
</tr>
</tbody>
</table>

“This is a joke. It refers to the famous Star Trek episode “The Trouble With Tribbles.” OK, so it’s not funny.
Generating Quadruples...

- **NewTemp** generates a new temporary var.

```
PROCEDURE Stat (n: Node);
  IF n.Kind = Assign THEN
    Expr(n.Des);
    Expr(n.Expr);
    Emit(n.Des.Place ' := ' n.Expr.Place);
    Stat(n.Next);
  ENDIF
END;
```

Generating Quadruples...

```
PROCEDURE Expr (n: Node);
  IF n.Kind = Add THEN
    Expr(n.LOP);
    Expr(n.ROP);
    n.Place := NewTemp();
    Emit(n.Place ' := ' n.LOP.Place '+' n.ROP.Place);
  ELSIF n.Kind = VarRef THEN
    n.Place := n.Symbol;
  ENDIF
END;
```

Generating Quadruples...

```
PROCEDURE Program (n: Node);
  Decl(n.DeclSeq);
  Stat(n.StatSeq);
END;
```

Generating Quadruples...

- To generate quadruples from an AST we make an extra traversal of the tree after semantic analysis.
- Each AST node in an expression sub-tree is given an attribute \( \uparrow \text{Place.SymbolT} \) which represents the name of the identifier or temporary in which the value of the subtree will be computed.

```
PROCEDURE Decl (n: Node);
  IF n.Kind = ProcDecl THEN
    Decl(n.Locals);
    Decl(n.Next);
    Stat(n.StatSeq);
  ENDIF
END;
```
### 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 $B_1 \cdots B_n$ 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.

### Generating Quadruples...

#### Symbol Table Entry

- **Assign**: Des : Expr
- **Binary**: Place = Op

#### Quadruples

- **Assign**: Des = Expr
- **Binary**: Place = Op

### Attribute Grammar

- The tree-walker is better described using an attribute grammar notation.

**Assign** ::= Des : Expr | Expr : Expr

\{ Emit(n.Des.Place ':=' n.Expr.Place); \}

**Add** ::= LOP : Expr | ROP : Expr

\{ n.Place := NewTemp();
  Emit(n.Place ':=' n.LOP.Place '+=' n.ROP.Place);
\}

**Name** ::= Ident

\{ n.Place := n.Symbol; \}
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; ENDF;
ENDDO; Y := X + 5;
```

Intermediate Code:

1. X := 20
2. if X>=10 goto (8)
3. X := X-1
5. if X<>4 goto (7)
6. X := X-2
7. goto (2)
8. Y := X+5

Constructing Basic Blocks

- Assume that the input is a list of tuples. How do we find the beginning and end of each basic block?

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.
Readings and References

- Read Louden:
  Intermediate Code 398–407
  Generating Intermediate Code 407–410
  Flow Graphs 475–477
- Or, read the Dragon book:
  Postfix notation 33
  DAGs & Value Number Alg. 290–293
  Intermediate Languages 463–468, 470–473
  Assignment Statements 478–481
  Basic Blocks 528–530
  Flow Graphs 532–534

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 make 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.

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

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

Translate the program below into quadruples. Identify beginnings and ends of basic blocks. Build the control flow graph.

PROGRAM P;
VAR X : INTEGER; Y : REAL;
BEGIN
  X := 1; Y := 5.5;
  WHILE X < 10 DO
    Y := Y + FLOAT(X);
    X := X + 1;
    IF Y > 10 THEN Y := Y * 2.2; ENDIF;
  ENDDO;
END.

**Exam Question**

- Draw the control flow graph for the tuples.

```plaintext
int A[5],x,i,n;
for (i=1; i<=n; i++) {
  if (i<n) {
    x = A[i];
  } else {
    while (x>4) {
      x = x*2+A[i];
    };
  }
  x = x+5;
}

(1) i := 1
(2) IF i>n GOTO (14)
(3) IF i>n GOTO (6)
(4) x := A[i]
(5) GOTO (11)
(6) IF x<=4 GOTO (11)
(7) T1 := x*2
(8) T2 := A[i]
(9) x := T1+T2
(10) GOTO (6)
(11) x := x+5
(12) i := i+1
(13) GOTO (2)
(14) GOTO (2)
```

**Homework I**

- Translate the program below into quadruples, triples, and a 'sequence of expression trees.'

PROGRAM P;
VAR X : INTEGER; VAR Y : REAL;
BEGIN
  X := 1; Y := 5.5;
  WHILE X < 10 DO
    Y := Y + FLOAT(X);
    X := X + 1;
    IF Y > 10 THEN Y := Y * 2.2; ENDIF;
  ENDDO;
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.