University of Arizona

CSc 620

Security Through Obscurity

Christian Collberg
January 24, 2002

Program Representation

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.

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.

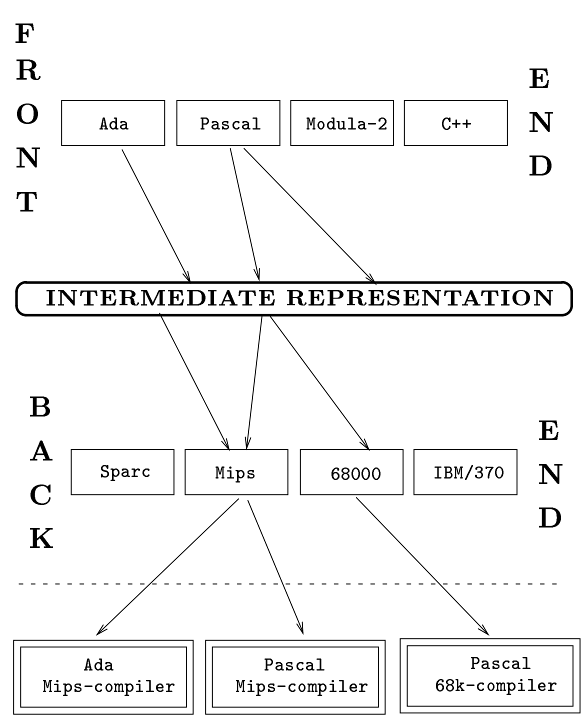
Requirements:

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 **UNCOL**, a **Universal Compiler Oriented Language**.

Slide 4–2

Mix-and-Match Compilers



Slide 4–3

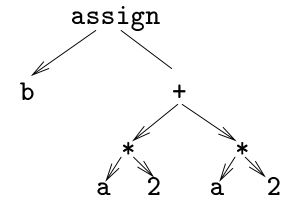
Intermediate Code...

- 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,
  2. Machine architectures differ.
- 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.

Slide 4–4

Postfix Notation

Infix: $\boxed{\text{b} := (a * 2) + (a * 2)}$



Postfix: $\boxed{\text{b a 2 * a 2 * + :=}}$

- 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 4–5

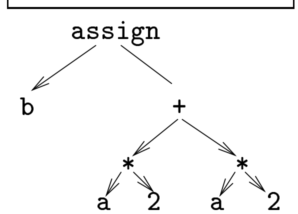
Tree & DAG Repr.

- 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 $a * 2$ gets two subtrees in the tree representation and one subtree in the DAG representation.

Slide 4–6

Tree & DAG Repr....

$\boxed{\text{b} := (a * 2) + (a * 2)}$



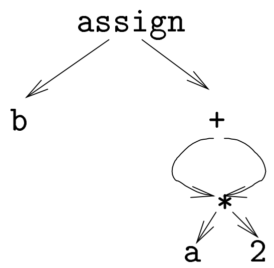
Linearized Tree:

Nr	Op	Arg_1	Arg_2
1	**ident**	a	
2	**int**	2	
3	**mul**	1	2
4	**ident**	a	
5	**int**	2	
6	**mul**	4	5
7	**add**	3	6
8	**ident**	b	
9	**assign**	8	7

Slide 4–7

Tree & DAG Repr. . .

$\boxed{\text{b} := (a * 2) + (a * 2)}$



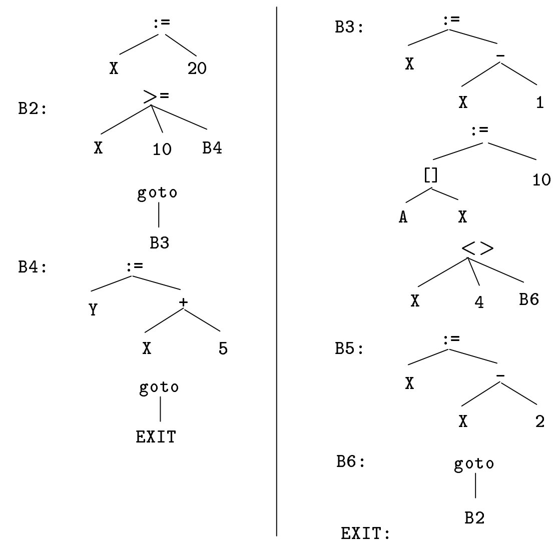
Linearized DAG:

NR	OP	ARG$_1$	ARG$_2$
1	**ident**	a	
2	**int**	2	
3	**mul**	1	2
4	**add**	3	3
5	**ident**	b	
6	**assign**	5	4

Slide 4–8

Sequence of Expression Trees

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



Slide 4–9

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.

goto L Unconditional jump.

Slide 4–10

Three-Address Code. . .

if x relop y goto L Conditional jump. `relop` is one of `<`,`>`,`<=`, etc. If `x relop y` evaluates to `True`, then jump to label L. Otherwise continue with the next tuple.

param X; **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`.

x := ADDR(y) Address assignment. Set `x` to the address of `y`.

x := IND(y) Indirect assignment. Set `x` to the value stored at the address in `y`.

IND(x) := y Indirect assignment. Set the memory location pointed to by `x` to the value held by `y`.

Slide 4–11

Three-Address Code...

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

$$\boxed{\mathrm{b} := (a * 2) + (a * 2)}$$

t_1	:=	a	**mul**	2
t_2	:=	a	**mul**	2
t_3	:=	t_1	**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.

Slide 4–12

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.

$$\boxed{\mathrm{b} := (a * 2) + (a * 2)}$$

NR	RES	OP	ARG$_1$	ARG$_2$
(1)	t_1	**mul**	a	2
(2)	t_2	**mul**	a	2
(3)	t_3	**add**	t_1	t_2
(4)	b	**assign**	t_3	

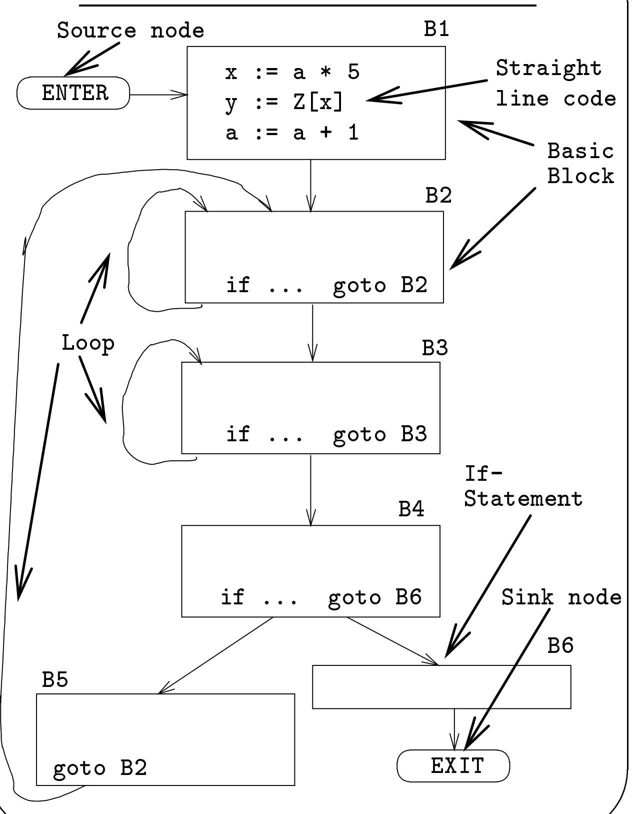
Slide 4–13

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 $B_1 \to B_2$ 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.

Slide 4–14

Control Flow Graphs...



Slide 4–15

Control Flow Graphs...

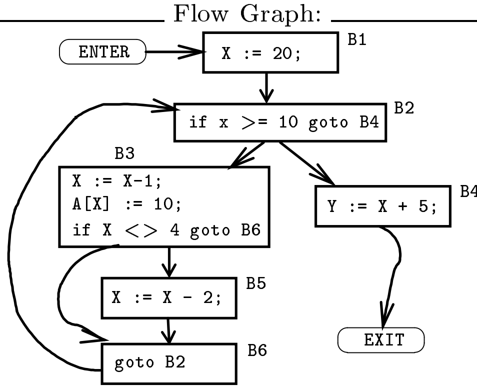
Source Code:

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

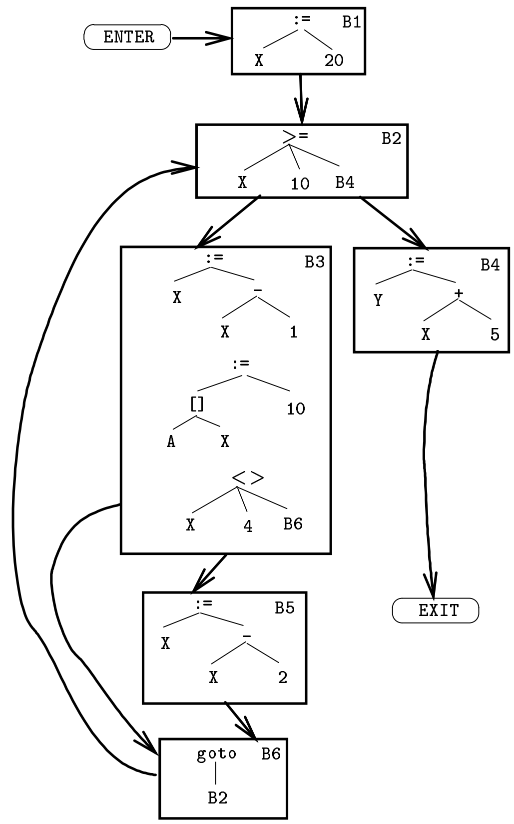
Intermediate Code:

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

Flow Graph:



Slide 4–16



Slide 4–17

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.

Slide 4–18

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
```

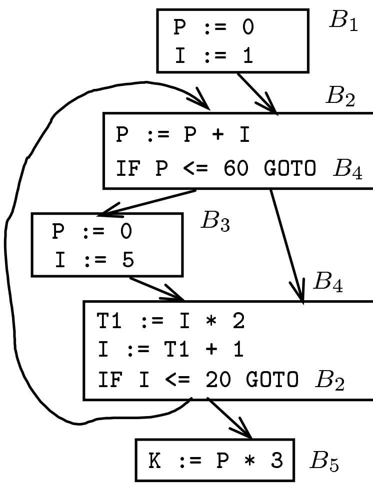
Tuples:

```
(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 4–19

Basic Blocks...

```
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 4–20

CFGs and Exceptions

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

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 $u \to v$ if u is a basic block that can throw an exception and v is a handler block that could catch that exception.

Slide 4–21

CFGs and Exceptions...

3. Create a special basic block $\boxed{\texttt{ABORT}}$.
4. Add an edge $u \to \boxed{\texttt{ABORT}}$ 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.

Slide 4–22

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`

Slide 4–23

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.

Slide 4–24

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.

Slide 4–25

Exam Question

- Draw the control flow graph for the tuples.

```
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)
```

Slide 4–26