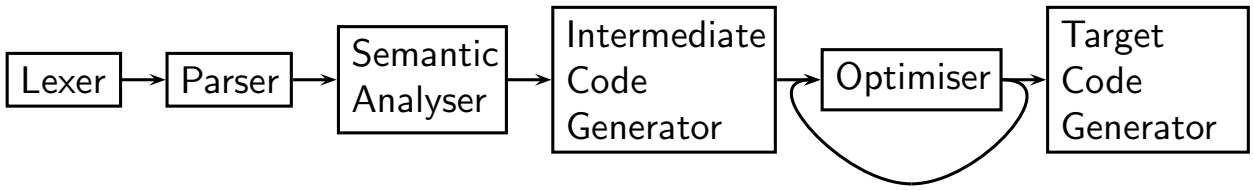
CA4003 - Compiler Construction

Intermediate Representation

David Sinclair

Why use an Intermediate Representation (IR)?

The structure of a typical compiler is:



There are many reasons for including an *Intermediate Code Generation* phase:

- keeping with good software engineering practices, it allows us to partition the compilation process and in particular separate the front-end from the back-end;
- simplifies changing the target code architecture without modifying the lexical, syntactic and semantic analysis;
- allows for multiple machine independent optimisation passes to be carried out.

Types of IRs

The goal of an *Intermediate Representation* (IR) is to allow multiple optimisation passes, where each pass transforms the current IR into an equivalent IR that runs more efficiently.

There are many different types of IRs:

- abstract syntax trees (ASTs)
- directed acyclic graphs (DAGs)
- control flow graphs
- 3-address code
  - single statement assignment (SSA)
- hybrids
  - combinations of all of the above (in practice this is what usually implemented)

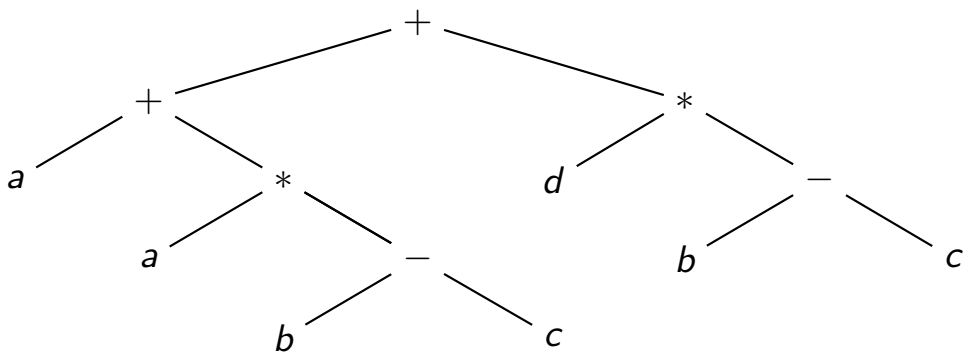
Directed Acyclic Graphs (DAGs)

Directed Acyclic Graphs (DAGs) are a variant of abstract syntax trees (ASTs) where nodes are not duplicated and any given node may have more than one parent. They are very efficient at representing expressions and hence generate efficient code for the expression.

Consider the following expression:

$$a + a * (b - c) + (b - c) * d$$

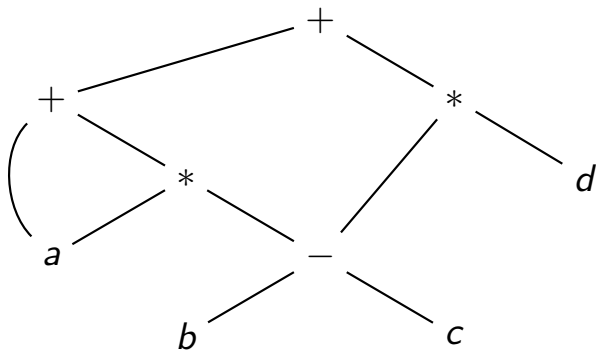
The AST for this expression is:



Directed Acyclic Graphs (DAGs) (2)

The DAG can be built from the AST by doing a post-order traversal, constructing each node and linking to an existing node if the constructed already exists in the DAG, otherwise add the constructed node to the DAG.

The corresponding DAG is:



3-address code

In *3-address code* there is at most one operator on the right-hand of an instruction. Hence, in 3-address code, the valid instructions for expressions are:

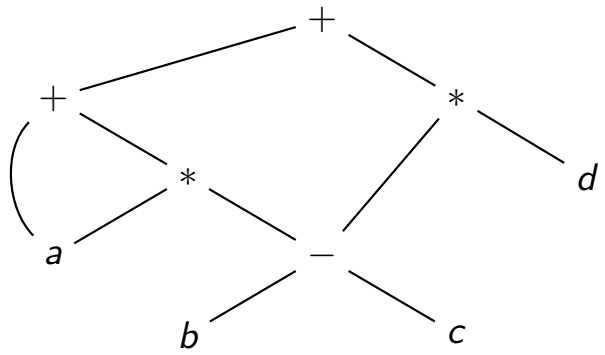
$$x = y \text{ op } z$$
$$x = \text{op } y$$
$$x = y$$

Complex expressions in the source language can be translated in a sequence of 3-address code instructions. For example $w = x + y - z$ would be translated to:

$$t_1 = y - z$$
$$w = x + t_1$$

3-address code (2)

3-address code is actually a linearised version of an AST or DAG.



$$t_1 = b - c$$
$$t_2 = a * t_1$$
$$t_3 = a + t_2$$
$$t_4 = d * t_1$$
$$t_5 = t_3 + t_4$$

3-address code Instructions

The intermediate representation using 3-address code are built on the concepts of addresses and instructions.

An *address* can be one of the following.

Name A *name* is a pointer to the symbol table entry that corresponds to a source programme name (variable). For simplicity sake in these notes we will just use the *name* rather than a pointer to its entry in the symbol table.

Constant There may be a need to type convert a constant.

Temporary These *compiler-generated temporaries* are created as needed. They may be removed by different optimisation passes. The remaining *temporaries* will be allocated to registers if possible.

3-address code Instructions (2)

The intermediate representation, based on 3-address code, that we will use has the following instructions.

$x = y\ op\ z$	Assignments where *op* is a binary arithmetic (e.g. +,-,*,) or binary logical (e.g. &, |) operation.
$x = op\ y$	Assignments where *op* is a unary operation such as unary minus, logical negation and type conversion operators.
$x = y$	Copy instructions
goto L	Unconditional jump to label *L*
if x goto L *ifFalse x goto L*	Conditional jumps where control is passed to label *L* is *x* is evaluated to **true** or **false** respectively. Otherwise control passes to the next instruction.

3-address code Instructions (3)

if x relop y goto L	Conditional jump where control is passed to label *L* if *x relop y* is **true** and *relop* is a binary relational operator (e.g. $>, >=, ==, <, <=$, etc.). Otherwise control passes to the next instruction.
param x_n ... *param* x_1 $y = call\ p, n$	Procedure calls where the *n* arguments to procedure *p* are defined by the *param* instructions. If there is no returned value from procedure *p*, the procedure is invoked by *call p, n*.
return	Passes control to the instruction following the *call* instruction that invoked the procedure.

3-address code Instructions (4)

$x = y[i]$ $x[i] = y$	Indexed copy instructions. In the first case x is set to the value that is i memory locations beyond location y. In the second case the memory location that is i memory locations beyond x is set to value in y.
$x = \&y$ $x = *y$ $*x = y$	Address and pointer assignments. $x = \&y$ sets (the r-value of) x to the location (l-value) of y. If y is an l-value, then $x = *y$, set x to the value in the location held in y. $*x = y$ sets the value of the location held in x to y.

This is not the only IR based on 3-address codes. The choice of IR is a trade-off between ease of translation from source language, ease of optimisation and ease of translation to target language.

Quadruples

Quadruples are a data structure for storing 3-address code instructions. Each *quadruple*, or simply *quad*, has 4 elements:

op, arg_1, arg_2 and *result*

In the case of instructions such as $x = y$ and $x = op\ y$, arg_2 is not used.

In the case of instructions such as *param* x_i, arg_2 and *result* are not used.

In the case of jump instructions, the target label is put into *result*.

Quadruples (2)

Consider the source language assignment:

```
a = b * -c + b * -c
```

$t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

3 -address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

Quadruples

Triples

With *quadruples* the *result* field is used to specify the *temporary* that holds the result of the instruction. *Triples* are an optimisation (in terms of space) where we use the instruction's position to specify the *temporary* that will hold the result.

Using the expression `a = b * -c + b * -c`, the corresponding set of *triples* is:

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

The bracketed numbers are pointers to the *temporaries* associated with each entry in the *triple* structure.

Triples (2)

In the case of copy instructions, arg_1 holds the receiving location and arg_2 holds the source location.

In the case of jump instructions, arg_1 holds the condition and arg_2 holds the target label.

Ternary operations, like `x[i] = y`, require 2 entries in the triple structure.

	op	arg_1	arg_2
0	[ ]	x	i
1	$=$	(0)	y

Indirect Triples

In the subsequent optimisation stage, *triples* cause complications as reordering the triples will result in a triple referring to the wrong *temporary*. The solution is to use another table to index into the *triple* structure.

25	(0)
26	(1)
27	(2)
28	(3)
29	(4)
30	(5)

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)

Now we can reorder the entries in the first table without messing up the references to *temporaries*.

Static Single Assignment (SSA)

An IR is in *static single assignment* (SSA) form if each variable (or temporary) is assigned exactly once and every variable (or temporary) is assigned a value before it is used.

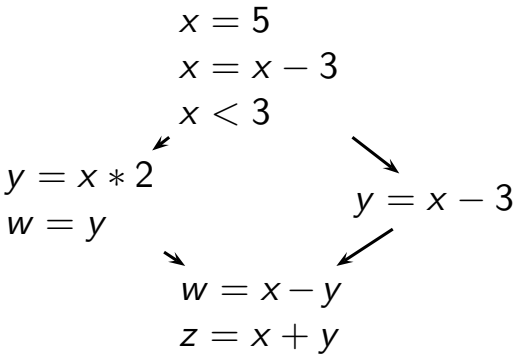
If a variable is used more than once then each use is assigned a separate variable, generally using subscripts to distinguish between them.

$$\begin{aligned} p &= a + b \\ q &= p - c \\ p &= q * d \\ p &= e - p \\ q &= p + q \end{aligned}$$

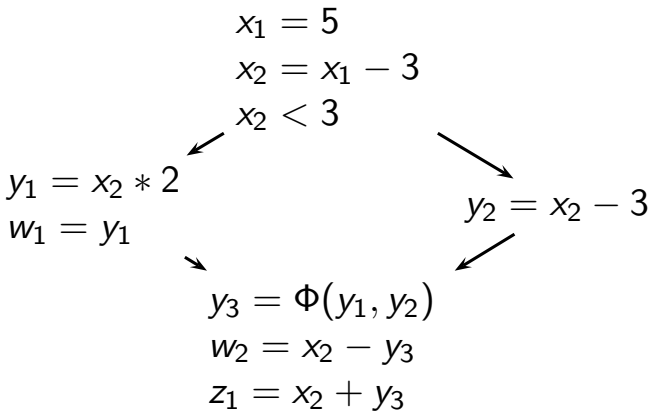
$$\begin{aligned} p_1 &= a + b \\ q_1 &= p_1 - c \\ p_2 &= q_1 * d \\ p_3 &= e - p_2 \\ q_2 &= p_3 + q_1 \end{aligned}$$

Static Single Assignment (SSA) (2)

Consider the following program fragment.



Which SSA version of y should we use in $w = x - y$ and $z = x + y$?



The Φ function selects the appropriate value of y depending on the control flow that occurred leading up to the assignment of y_3.

Type Expressions for Arrays

Arrays are best stored as a block of consecutive locations. The array name will be the *base* address of this block. There are 2 layouts for multidimensional arrays:

Row-major where A[2][3] is read as an array of 2 arrays, each array containing 3 elements. This is used in C, Java and related languages.

Column-major where A[2][3] is read as an array of 3 arrays, each array containing 2 elements. This is used in Fortran and related languages.

We will focus on *row-major* layouts. Before accessing an element of an array we need to know the size of each element and the size of each row. We get this from the array declaration and store it in the symbol table.

Type Expressions for Arrays (2)

To access $A[i]$ where each element is stored in w bytes and the index starts at 0, we access the location

$$A + (i * w)$$

To access a 2-dimensional array $A[i_1][i_2]$ where each element is stored in w bytes, each row contains n_2 elements and the index starts at 0, we access the location

$$A + (i_1 * n_2 + i_2) * w$$

To access a k-dimensional array $A[i_1] \ldots [i_k]$ where each element is stored in w bytes, row i contains n_i elements and the index starts at 0, we access the location

$$A + (((i_1 * n_2 + i_2) * n_3 + i_3) \ldots) * w$$

Translating Expressions

We will use a syntax-directed definition approach to build the 3-address code instructions for expressions for statement S and expression E. The attributes $S.code$, $E.code$ and $E.addr$ respectively represent the code for S, the code for E and the location that holds the value of E.

The function $gen(x\ '='y\ '+'\ z)$ will produce the 3-address code `x = y + z` where + is any binary operator **and** concatenates it with the previously generated code (incremental translation).

The function $get(id.lexeme)$ retrieve the address currently associated with the lexeme if id in the symbol table.

Translating Expressions (2)

For a simple expression language with arrays, $(L \rightarrow L[E] \mid \mathbf{id}[E])$, this could be:

Production	Semantic Rule
$S \rightarrow \mathbf{id} = E;$	$gen(get(\mathbf{id}.lexeme)\ '='\ E.addr);$
$S \rightarrow L = E;$	$gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$
$E \rightarrow \mathbf{id}$	$E.addr = get(\mathbf{id}.lexeme);$
$E \rightarrow L$	$E.addr = \mathbf{new}\ Temp();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$

Translating Expressions (3)

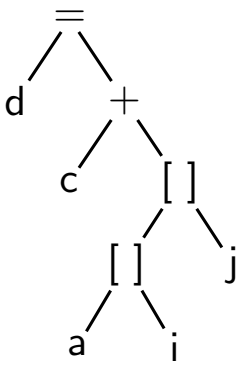
Production	Semantic Rule
$L \rightarrow \mathbf{id}[E]$	$L.array = get(\mathbf{id}.lexeme)$; $L.type = L.array.type.elem$; $L.addr = \mathbf{new}\, Temp()$; $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$;
$L \rightarrow L_1[E]$	$L.array = L_1.array$; $L.type = L_1.type.elem$ $t = \mathbf{new}\, Temp()$; $L.addr = \mathbf{new}\, Temp()$; $gen(t\ '='\ E.addr\ '*'\ L.type.width)$; $gen(L.addr\ '='\ L_1.addr\ '+'\ t)$;

Translating Expressions - Example

Consider the source language statement `d = c + a[i][j]` where a is a 2×3 array of integers and d, c, i and j are integers.

Assume the *width* of an integer is 4. The the type of a is $array(2, array(3, integer))$ and its *width* is 24. The type of $a[i]$ is $array(3, integer)$ and its *width* is 12. The type of $a[i][j]$ is *integer*.

The AST is



The translation to 3-address code is

$$t_1 = i * 12$$
$$t_2 = j * 4$$
$$t_3 = t_1 + t_2$$
$$t_4 = a[t_3]$$
$$t_5 = c + t_4$$
$$d = t_5$$

Translating Flow of Control

We will consider the classic control flow statements given by:

$$
\begin{aligned}
S &\rightarrow \textbf{if } (\ B\ )\ S_1 \\
S &\rightarrow \textbf{if } (\ B\ )\ S_1 \textbf{ else } S_2 \\
S &\rightarrow \textbf{while } (\ B\ )\ S_1
\end{aligned}
$$

The first step is to generate if the IR for the boolean condition B. In the following rules, $C_1 \| C_2$ concatenates the code for C_2 after the code for C_2. The functions *newlabel*() and *label*() generate and instantiate labels.

Production	Semantic Rule
$B \rightarrow B_1 \| \| B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B_1.code \| label(B_1.false) \| B_2.code$

Translating Flow of Control (2)

Production	Semantic Rule
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B_1.code \| label(B_1.true) \| B_2.code$
$B \rightarrow !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \| E_2.code$ $\| gen(\text{'if'}\ E_1.addr\ \textbf{rel}\ E_2.addr\ \text{'goto'}\ B.true)$ $\| gen(\text{'goto'}\ B.false)$
$B \rightarrow$ **true**	$B.code = gen(\text{'goto'}\ B.true)$
$B \rightarrow$ **false**	$B.code = gen(\text{'goto'}\ B.false)$

Translating Flow of Control (3)

Production	Semantic Rule
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \Vert label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \Vert label(B.true) \Vert S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \Vert label(B.true) \Vert S_1.code$ $gen(\text{'goto'}\ S.next) \Vert label(B.false) \Vert S_2.code$

Translating Flow of Control (4)

Production	Semantic Rule
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \Vert B.code$ $\Vert label(B.true) \Vert S_1.code \Vert gen(\text{'goto'}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel$ $S_2.next = S.next$ $S_1.code \Vert label(S_1.next) \Vert S_2.code$

Translating Flow of Control - Example

Consider the source-code fragment

```
sum = 0;
i = 0;
while (i < 10)
{
  sum = sum + a[i];
  i = i+ 1;
}
```

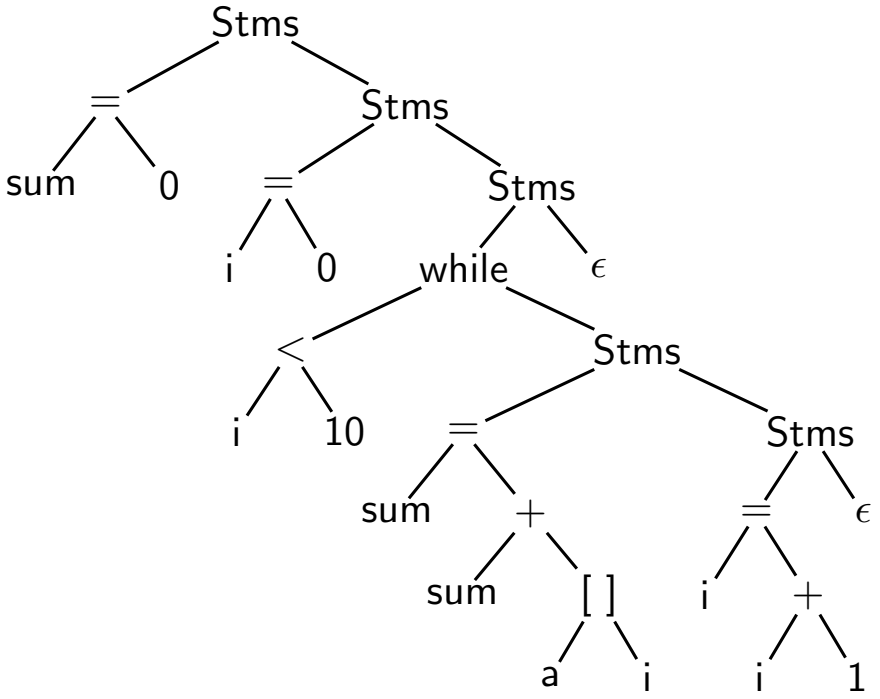
where the grammar represents a sequence of statements as

$$Stms \rightarrow Stm; Stms$$
$$Stms \rightarrow \epsilon$$

and *Stm* is single statement.

Translating Flow of Control - Example (2)

The AST is



Translating Flow of Control - Example (3)

Assuming all the variables are integers and an integer is stored in 4 bytes, then the generated 3-address code is:

$$
\begin{array}{ll}
 & sum = 0 \\
 & i = 0 \\
Lbl1: & \textbf{if } i < 10 \textbf{ goto } Lbl2 \\
 & \textbf{goto } Lbl3 \\
Lbl2: & t_1 = i * 4 \\
 & t_2 = a[t_1] \\
 & sum = sum + t_2 \\
 & i = i + 1 \\
 & \textbf{goto } Lbl1 \\
Lbl3: &
\end{array}
$$

IR for Procedure Calls

In 3-address code, a procedure call is decomposed into:

- preparing the parameters; and
- invoking the procedure.

For example, given the source language statement

```
n = f(a[1]);
```

the generated 3-address code would typically be

$$
\begin{array}{l}
t_1 = 1 * 4 \\
t_2 = a[t_1] \\
\textbf{param } t_2 \\
t_3 = \textbf{call } f, 1 \\
n = t_3
\end{array}
$$

Control Flow Graphs

A *control flow graph* is a data structure that represents the flow of control in an intermediate representation (IR), such as 3-address code. The first step in creating a *control flow graph* is to partition the IR into *basic blocks* that represent consecutive IR statements. These will be the nodes of the *control flow graph*. The conditional and unconditional jumps generate the edges in the *control flow graph*.

Control flow graph are very useful in optimising the IR and the generation of target code.

Control Flow Graphs (2)

A *basic block* is a sequence of IR statements where the flow of control is always from one statement to the next statement in the IR. When there are no jumps or labels, the flow of control always proceeds sequentially from one statement to the next statement.

Consider the following source code.

```
for (i = 1; i <= 10; i++)
{
  for(j = 1; j <= 10; j++)
  {
    a[i][j] = 0.0;
  }
}
for (i = 1; i < 10; i++)
{
  a[i][i] = 1.0;
}
```

Control Flow Graphs (3)

The resulting 3-address code could be (if a float requires 8 bytes):

1. $i = 1$
2. $L1:$ $j = 1$
3. $L2:$ $t_1 = i * 10$
4. $t_2 = t_1 + j$
5. $t_3 = t_2 * 8$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. $j = j + 1$
9. **if** $j <= 10$ **goto** $L2$
10. $i = i + 1$
11. **if** $i <= 10$ **goto** $L1$
12. $i = 1$
13. $L3:$ $t_5 = i - 1$
14. $t_6 = t_5 * 88$
15. $a[t_6] = 1.0$
16. $i = i + 1$
17. **if** $i <= 10$ **goto** $L3$

Control Flow Graphs (4)

To decompose this into *basic blocks* we first identify the *leaders*. *Leaders* are:

1. the 1^{st} instruction;
2. any instruction that is the target of a jump (conditional or unconditional); and
3. any instruction immediately following a jump.

For each *leader* its *basic block* are all the subsequent instructions until the next *leader*.

The jumps (conditional or unconditional) are always to leaders (i.e the start of a *basic block*) and these form the edges in the *control flow graph*.

Control Flow Graphs (5)

The resulting *control flow graph* is: