

**2 CONTROL STRUCTURES Control Structures Classes**

**3 Control Structures Classes-Details**

**4 Theorems On Control Structures-Definitions-Introduction**

**5 Definitions Functional and Semantic Equivalence Semantic Power**

**6 EXAMPLES Semantic and Functional Equivalence and Power**

**7 EXAMPLES Semantic and Functional Equivalence and Power**

**8 Theorems On Control Structures-Statements Boehm-Jacopini, Kosaraju, McCabe**

**9 About Proofs**

**10 Example: The Advantage of the Exit Command in REi**

**11 Exceptions**

**12 Exceptions in C++**

## **CONTENTS**

## CONTROL STRUCTURES

Control structures are all those commands in a language which cause a **change in the flow of control**. At the assembly language level there is the **transfer or goto** command which explicitly changes the flow of control. This may be **un-conditional or conditional** depending on a preset variable. In higher level languages the *goto* and condition testing are typically incorporated into a single **if c then B1 [else B2]** command. The need for the **goto** is reduced by allowing B1 and B2 to be any subprogram. In addition to the **if** command there are often other more specialized decision commands, ex, case statements, which provide convenience even though an equivalent effect can be achieved with simpler **if** commands. Since, after all, convenience helps justify the higher level languages, the specialized decision instructions can provide a real gain.

In addition to decision instructions there are a number of commands which **control looping** in a program. Again, looping is expressed in assembly language with simple *goto*'s. Such *goto*'s carry control backwards in a program and are usually executed conditionally. In the higher level languages the looping and the decision that controls it as well as the block of program through which the looping will occur are incorporated in one structure as, for example, **while c do B end** command.

### Control Structures Classes

The use of different sets of control structures provide different powers in creating a program. An exploration of how the ease of program specification is effected by the addition of more and more control structures will make this notion of power more precise. Following are several interesting classes of program constructs, all adequate for programming any function, but differing in their means and convenience for doing so. The first class, **D-S-structure**, has just enough commands to be universal. The classes that follow contain more and more control commands until the final one, **L-structure**. L-structure contains all the commands in other classes, and in addition the infamous **goto** command

### Comparisons

In addition to a fixed set of Control Structures a class will need the facility to do Basic Actions and to use a variety of Conditions to guide their control structures. We will use program schema which are abstract representations of programs. In which notation sufficient to distinguish differences between control structures, between conditions tested, and between actions are sufficient.

In Programs	In Schema
<b>Statements</b> consist of <b>Basic Actions</b> such as assignment and I/O statements ex. <code>x = y; printf("%d", x)</code>	<b>Statements</b> are represented by <code>s1, s2, ....</code> indicating different statements
<b>Conditions</b> are simple <b>Conditions</b> which are true or false ex. <code>x == y, x != y, etc.,</code> and Boolean expressions involving those conditions ex. <code>x == y &amp;&amp; x != y,</code>	<b>Conditions</b> are represented by <code>c1, c2, ....</code> indicating different conditions

Generally we will want to compare two program schema X and Y, each from a different class, and therefore each with **different Control Structures, but able to use the same actions and the same conditions**.

For the comparison to be focussed on these different control structures we will want:

X and Y to do the same thing (be **functionally equivalent**) and to do so using, as far as possible, the same set of statements (actions) and the same set of conditions.

If X and Y can be done with the same basic actions and conditions we say the X and Y are **semantically equivalent**. If X must use more conditions and or actions than Y then Y is **semantically more powerful** than X.

**D-S** 1. if statement  $s \in \mathbf{BA}$  then  $s \in \mathbf{D-S}$

2. if for  $i = 1$  to  $n$ ,  $s_i \in \mathbf{D-S}$  then the sequence  $s_1; ; ; ; s_n \in \mathbf{D-S}$

3. Conditional Structure

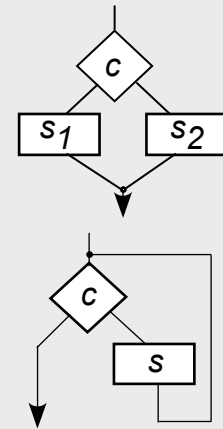
if  $c \in \mathbf{C}$  and  $s_1; s_2 \in \mathbf{D-S}$  then  $\text{if } c \text{ then } s_1 \text{ else } s_2; \in \mathbf{D-S}$

includes  $\text{if}(C)\{s_1; ; ; ; s_n\} \text{ else}\{s_1; ; ; ; s_m\}$

$\text{if}(C)\{s_1; ; ; ; s_n\}^1 s, 0$  brackets.

4. Iterative Structure: while

if  $c \in \mathbf{C}$ ,  $s \in \mathbf{D-S}$  then  $\text{while } c \text{ loop } s \text{ endloop}; \in \mathbf{D-S}$ .



In **D-S** every program has a single entry and a single exit (unlike programs with *goto*'s).

**D-S** structures have only one form of conditional expression(3) and one form of iterative structure (4). Will the addition of other forms of these structures effect the power of a language?

**D'-S** structures include all **D-S** structures, in addition to some alternative forms of conditional and iterative structures which are easily simulated by the structures in **D-S**.

**V** is a set of values.

**I** is an interval (e.i.,  $i := 1$  to  $100$ ,  $i := 1$  to  $100$  by  $2$ )

1. if  $s \in \mathbf{D-S}$  then  $s \in \mathbf{D'-S}$

2. Single Branch Conditional Structure

if  $c \in \mathbf{C}$  and  $s_j; \in \mathbf{D'-S}$  then  $\text{if } c \text{ then } s_1 \in \mathbf{D'-S}$ .

3. Case Conditional Structure

if  $exp$  is any expression that can appear on the right side of an assignment statement  $\in \mathbf{BA}$ , and  $s_1, \dots, s_n \in \mathbf{D'-S}$ , and  $v_1, \dots, v_n \in \mathbf{V}$ , then the following case statement is  $\in \mathbf{D'-S}$

case  $exp$  of  
when  $v_1 \Rightarrow s_1$

when  $v_n \Rightarrow s_n$   
endcase;

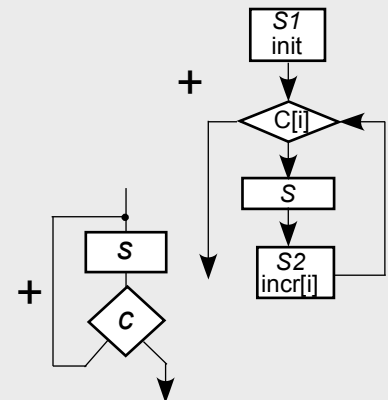
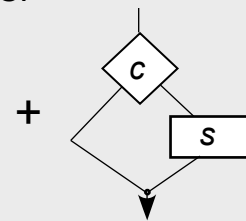
switch( $exp$ )  
{ case  $l_1$ :  $s_{11}; ; ; s_{1n}$  --[break]  
.  
case  $l_n$ :  $s_{n1}; ; ; s_{nn}$   
default  $s_{d1}; ; ; s_{dn}$

4. Iterative Structure Repeat-Until Statement

if  $c \in \mathbf{C}$ ,  $s \in \mathbf{D'-S}$  then  $\text{repeat } s \text{ until } c; \in \mathbf{D-S}$ .

5. Iterative Structure: For Statement

if  $i \in \mathbf{I}$ ,  $s \in \mathbf{D-S}$  then  $\text{for } i \text{ loop } s \text{ endfor}; \in \mathbf{D'-S}$ .



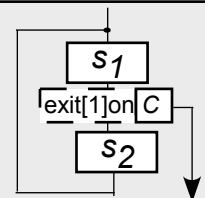
**RE<sub>j</sub>-Structures (RE<sub>j</sub>)**

1. if  $s \in \mathbf{D-S}$  then  $s \in \mathbf{RE}_j$

2. Iteration Structure-Using Exit

if  $c \in \mathbf{C}$  then  $\text{loop } s \text{ endloop}$  where  $s$  may be a sequence, and  $s$  contains the special commands  $\text{exit}(j)^*$  and  $\text{continue}$ ;

\* $\text{exit}(j)$  jumps out of a loops nested to a level  $j$ .if  $j >$  nested level jumps out to main after loop structure



**Free-Structures (L)**

1. if  $s \in$  any of the structure types given above  $s \in \mathbf{L}$ .

2. Furthermore labelled statements and  $\text{goto}(\text{label})$  may be used freely in  $\mathbf{L}$ .

goto label

## Control Structures Classes-Details

### 3 Theorems On Control Structures

#### Definitions

Two programs are **functionally equivalent** if for any input presented to both the output of both is the same.

Consider two programs S and T from different classes, implying that they may contain different control statements. If programs S and T are **functionally equivalent** and they both contain the **same** basic actions, **BA's**, the **same** basic conditionals **BCs** (for example  $c$  and  $\sim c$ , involve the **same** basic conditional, namely  $c$ ), and **same** variables **Vs**, perhaps arranged differently and using control statements which may or may not differ, then we say that S and T are **semantically equivalent**. On the other-hand if T contains the actions, conditions, and variables in S as well as some not in S we would say that S is **semantically more powerful** than T.

Usually we are interested in determining the relative power of the programs in different control classes. We say that class  $\Sigma$  is more powerful than class T if

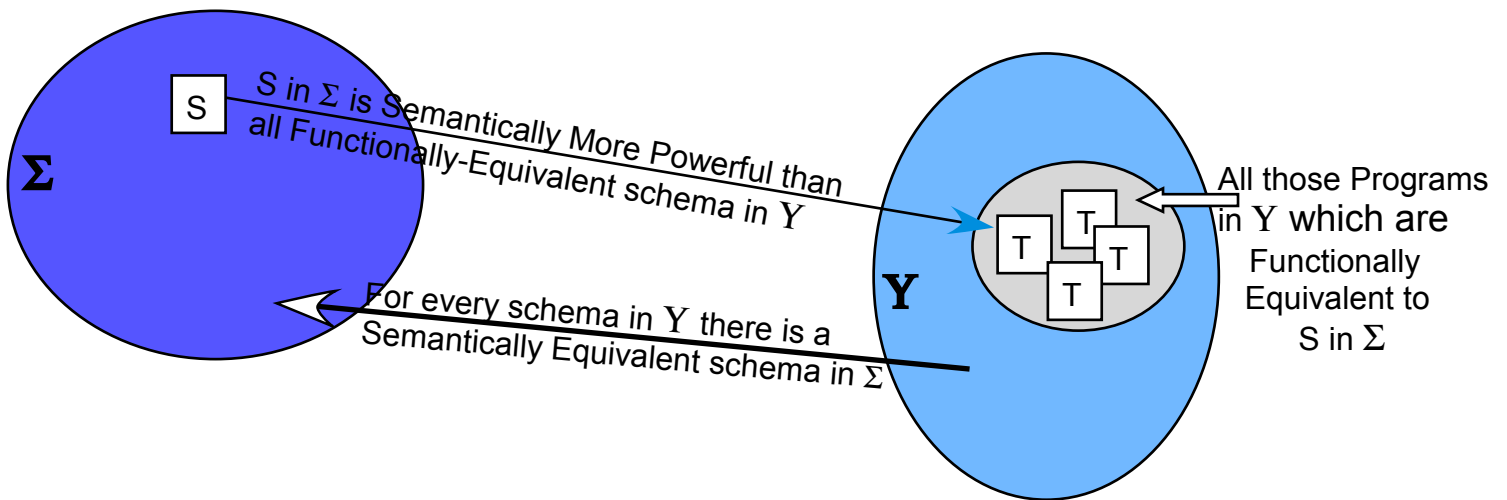
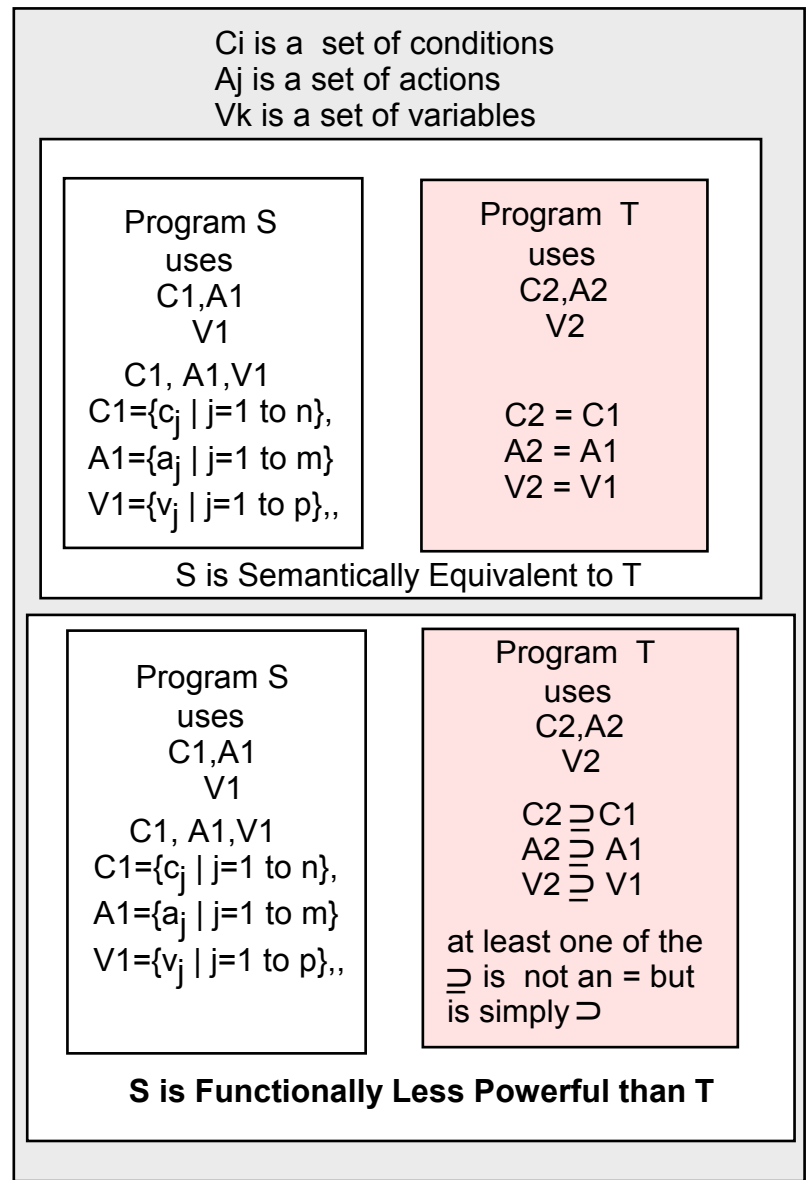
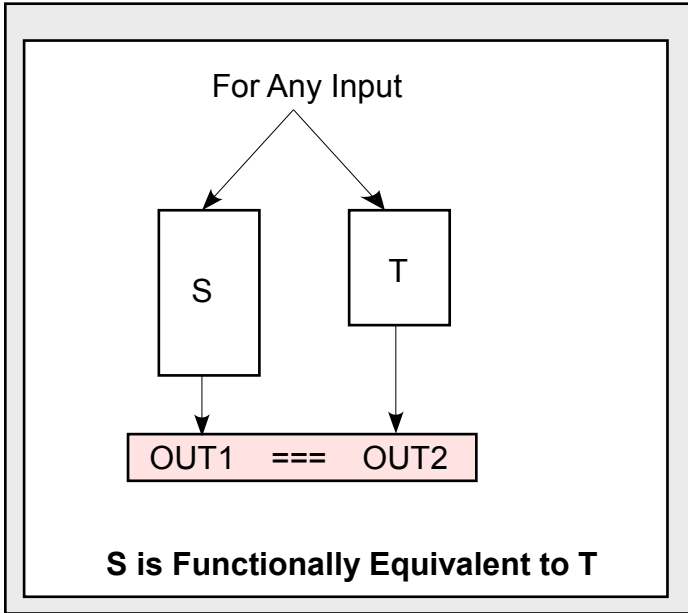
- 1 There is at least **one program S in  $\Sigma$  which is more powerful than all functionally equivalent programs in T**, and
- 2 **For every program T in T there is a semantically equivalent program in  $\Sigma$ .**

We say that class  $\Sigma$  and class T are semantically equivalent if

for every program in one there is a semantically equivalent program in the other.

(Note that this definition implies the existence of a series of classes in which each member of the series contains all the control structures of those appearing earlier in the series. such that any program written in one class of the series also can be written, identically, in all subsequent classes. So a class can never be less powerful than earlier classes in the series. Subsequently we will compare program schema from different languages to explore the relative power of those languages, according to these definitions of power. Program schema in which there are few concrete conditions or actions, are appropriate because we are only interested in the relation of conditions and actions in two schema, not the exact nature of such conditions and actions. (This may make the exploration problematic, since the exact nature of actions and conditions may be restricted so as to not be fairly represented by an abstract  $c$  which stands for any possible condition.))

**The question here is whether it is worthwhile to add a proposed control command to a language.** For example it is often necessary to search an array, say  $X[1..N]$ , for a member which satisfies some condition, say  $p(X[I])$ . using a *while* command. This may be done by making the *while* looping condition  $\sim p(X[I])$ , and decreasing the index, initially  $N$ , each time through the loop, and stopping the search when index  $I$  satisfies  $p(X[I])$ . However it will also be necessary to stop the loop when  $I == 0$  (assuming that  $1$  is the lowest array index), if the *while* condition is not met in the array. So perhaps the condition should be  $(\sim p(X[I]) \text{ 'or' } I == 0)$ . If this is done then at the end of the loop a test will be necessary to determine the reason the loop was ended. Another possibility is the "sentinal" approach, to use an array one larger than necessary to hold the data, say adding an  $X[0]$  and making  $p(X[0])$  true before entering the loop. This will provide the stopping condition when no element in the data array satisfies the condition. This change means that the  $I == 0$  condition need not be tested each time through the loop, but it does require an added command before entering the *while*. (this assignment may be made an integral part of a generalized *while*). A third way would be to provide a command which allows exit from a loop independent of the *while* condition: ex. an *if c then exit(1)*. The use of this removes need for the "or" operation in the *while* condition, but both tests are still done every time through the loop and the test is needed at the end of the loop. This addition would also apply to searches in multidimensional arrays, provided that the *exit* from inside a multi-nested *while* all the way to the main program, were allowed.



**$\Sigma$  is Semantically More Powerful than  $Y$**

**DEFINITIONS AGAIN**

### 3.1 Equivalent Programs From Different Classes

All the comparisons here are between Schema which differ in the control structures used

<pre>repeat   s until c;</pre>	Is FUNCTIONALLY and SEMANTICALLY EQUIVALENT to	<pre>s while ~c loop   s endloop;</pre>	<u>They implement the same function</u> <u>They use the same conditions and variables</u> <u>They Differ in the control functions used.</u>
--------------------------------	--	---	---

(a)

<b>P1</b> <pre>if c<sub>init</sub> then s<sub>1</sub> else s<sub>2</sub>;</pre>	Is FUNCTIONALLY EQUIVALENT but SEMANTICALLY MORE POWERFUL than:	<b>P2</b> <u>P2 uses the variable v not in P1</u> <pre>v = c<sub>init</sub>; if v then S1; if ~v then S2;</pre>
<div style="background-color: #e0ffe0; border: 1px solid black; padding: 10px;"> <p>The reason <math>c_{init}</math> should be not used directly is: if <math>c_{init}</math> is true then execution of <math>S_1</math> may change so as to make <math>\sim c_{init}</math> true als 0 and then <math>S_2</math> will also be executed.</p> <p>In more detail: The need for <math>v</math>: If initially <math>c_{init}</math> is the condition <math>(x &gt; 0)</math> and <math>x = 5</math> then <math>v = c_{init} = 1(\text{true})</math> and <math>\sim v = 0</math> (false) so, the two last lines could be</p> <pre>if 1 then {x=0;... if 0 then b2</pre> <p>If <math>c_{init}</math> is used directly:</p> <pre>if (x &gt; 0) then {x=0;... if ~(x &gt; 0) then {...</pre> <p>and both conditions are met-NG</p> </div>		
<pre>if c then s<sub>1</sub> else NOOP(s<sub>3</sub>); s<sub>2</sub>;</pre>	Functionally Equivalent But Semantically?	<pre>if c then s<sub>1</sub>; s<sub>2</sub>;</pre>

(b)

The next two schema, one from RE<sub>1</sub>, using an `exit(1)`, and the other, from L, using a `goto`, are FUNCTIONALLY EQUIVALENT. Both contain the BAs  $c_1, c_2, s_1, s_2,$  and  $s_3$

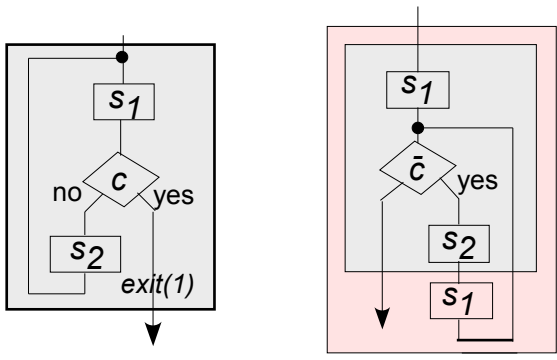
<pre>while c1 loop   if c2 then [s<sub>1</sub>; exit(1);]   else s<sub>2</sub>; endloop;</pre> <p style="text-align: center;">RE<sub>1</sub></p>	Is FUNCTIONALLY EQUIVALENT to	<pre>A: if c1 then( if c2 then s<sub>1</sub>; else (s<sub>2</sub>; goto A) )</pre> <p style="text-align: center;">L</p>
--	-------------------------------	---

Though it is FUNCTIONALLY EQUIVALENT to the schema below they are SEMANTICALLY MORE Powerful than the schema from D',S with neither the `exit(1)` or `goto`, shown below

<pre>c3 = true; while c1 &amp; c3 loop   if c2 then [s<sub>1</sub>; c3 = false;] else s<sub>2</sub>; endloop</pre>	
--	--

(c)

FUNCTIONALLY EQUIVALENT



```

loop
s1;
if c then exit(1);
s2;
endloop
    
```

RE<sub>1</sub>

```

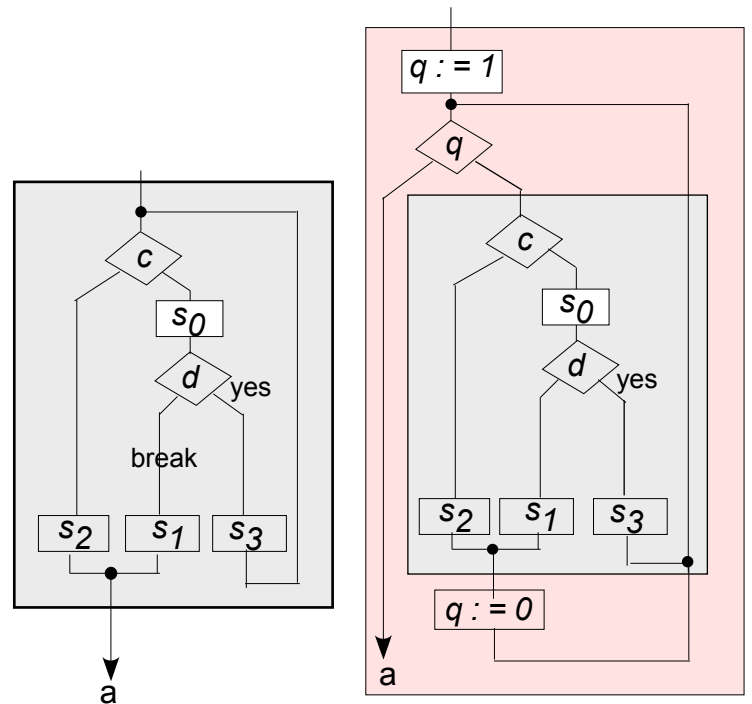
s1;
while c-bar loop
s2;
s1;
endloop
    
```

DS

SEMANTICALLY EQUIVALENT  
(But D-S requires an extra copy of s1)

(a)

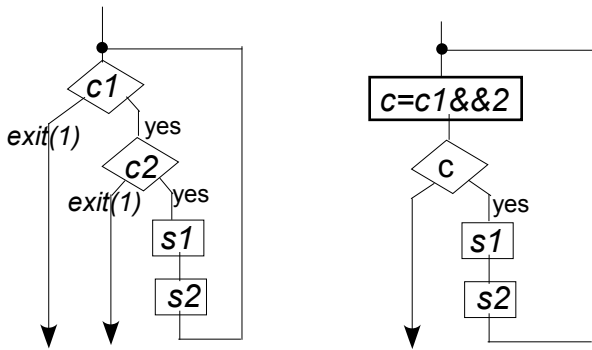
FUNCTIONALLY EQUIVALENT



RE<sub>1</sub>  $\Rightarrow$  DS'  
SEMANTICALLY MORE POWERFUL

(b)

FUNCTIONALLY EQUIVALENT



```

loop
if c1 then
{ if c2 { s1; s2 };
else exit(1); }
else exit(1);
endloop
    
```

RE<sub>1</sub>

```

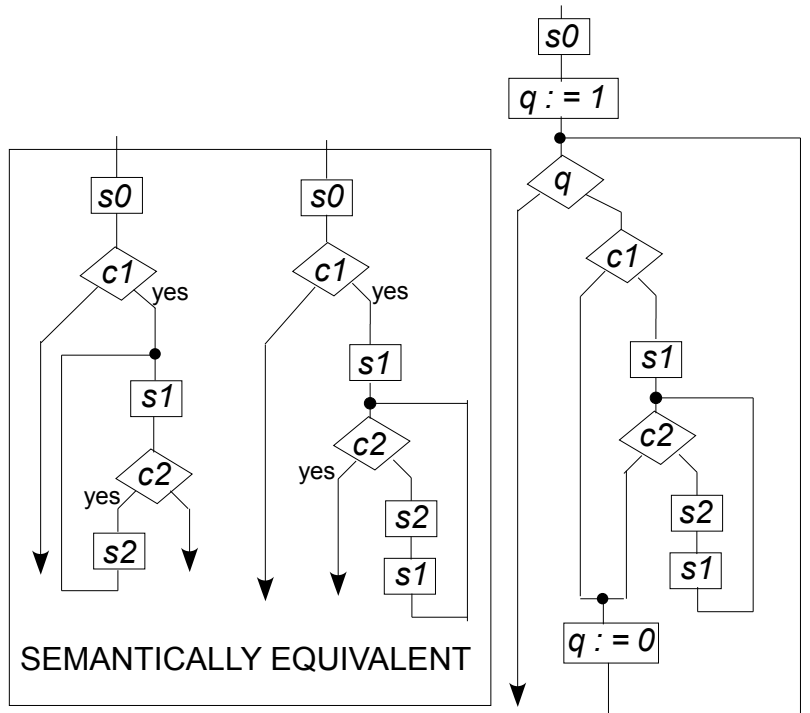
repeat
c=c1&&2
if c then { s1; s2 };
until ~c
endloop
    
```

DS'

SEMANTICALLY MORE POWERFUL



(c)



SEMANTICALLY EQUIVALENT

SEMANTICALLY MORE POWERFUL



(d)

### 3.2 Theorems

The basic results relating the different programming classes are given in a sequence of theorems.

#### Boehm-Jacopini Theorem: Functional Equivalence

For every proper program,  $S$ , one in which there is exactly one entrance and one exit, and in which there is a possible path from entry to exit involving each statement in the program, there is a **functionally equivalent**  $S$  in D-S. (You can do anything with D-S that can be done with any other class of control structures.)

#### Kosaraju's Theorem (1): Semantic Equivalence

A program  $P$  in L is **semantically equivalent** to one in  $D'$ -S iff  $P$  does not contain a loop with more than one exit. (You can do any program  $P$  in  $RE_1$  with a program,  $P'$ , in  $D'$ -S with  $P'$  having the same Actions and Conditions as  $P$ .)

#### Kosaraju's Theorem (2):

1. D-S and  $D'$ -S programs are **semantically equivalent**.
2. D-S and  $D'$ -S are **semantically less powerful** than  $RE_1$  programs.
3. For every program  $P$  in L there is an  $n$  such that  $P$  is a **semantically equivalent** to a member of  $RE_n$ .  $\forall P \in L, \exists n P \in R_n \ \& \ P = P'$  (pg 5 (c) pg 6 b,c,and d)
4. For every finite  $n$  there is a program  $P$  in L which has no **semantically equivalent** program in  $RE_n$ .  $\forall n \exists P \in L \ \& \ P \notin R_n$

#### McCabes Theorem:

A program that is not in D-S nor **semantically equivalent** to any member of D-S must contain at least two of the following:

- Branching out of a loop
- Branching into of a loop
- Branching out of a decision
- Branching into of a decision

### 3.3 About Proof

As a sample of the nature of the proof of some of the theorems we sketch the justification for showing that a loop *exit(i)* gives more power than not having one.

Consider a loop containing

$[b1 \ c \ b2]$  meaning a basic action,  $b1$ , followed by a decision based on condition  $c$  which, if positive, causes *exit* from the loop, and, if  $c$  is negative, followed by basic action  $b2$ ) Such a loop is directly  $RE_1$  as it stands also not in  $D'$ -S. But is there an equivalent in  $D'$ -S?

We can represent the result of running this loop as follows:

$b1 \ c \ b2 \ b1 \ c \ b2 \ b1 \ c \ b2 \ b1 \ c \ b2 \ \dots\dots$

this sequence is interpreted to mean:

*do b1 then test c and if it fails do b2, etc.*

If  $c$  succeeds at any time jump to the location at the end of the loop. Note that the repetitive part of this loop can be implemented by repeating the first three instructions  $b1 \ c \ b2$ , or the next three  $c \ b2 \ b1$ , or finally the third three,  $b2 \ b1 \ c$ , with the provision in the case that  $c \ b2 \ b1$  is the repetitive part that the loop is preceded by one execution of  $b1$  (Notice that such a structure is in D-S) or in the case that the repetitive part is  $b2 \ b1 \ c$  that the loop containing this be in the "yes" branch of a decision based on  $c$  and that decision be preceded by  $b1$ . (Although this is not in D-S the loop can be implemented as an *until* loop so it is in  $D'$ -S.)

Any other implementation of the repetitive part using just  $b1$ ,  $b2$ , and  $c$  part must involve a multiple of three instructions i.e., any sequence of 6 consisting of the 6 starting with the first the second or the third instruction, any sequence of 9, etc). (See figure 2a).

Consider another, slightly more complex loop with two conditions both of which can result in exit from the loop as allowed in RE<sub>1</sub>:

As above, the loop contains three instructions- this time  $c1 \ b \ c2$ . We can represent the function of such a loop as follows:

$c1 \ b \ c2 \ c1 \ b \ c2 \ c1 \ b \ c2 \ c1 \ b \ c2 \ \dots$

We interpret this sequence to mean test condition  $c1$ , if it fails do  $b$ , then test  $c2$  and if it fails do  $c1$  and then if it fails do  $b$  again, etc. If either  $c1$  or  $c2$  succeed at any time jump to the location at the end of the cycle. Now this is implemented in RE<sub>1</sub> by the loop containing  $c1 \ b \ c2$ . But by the type of analysis developed above: notice that it can be implemented by a loop containing  $b, \ c2, \ c1$  or  $c2, \ c1, \ b$  provided in the loop is in the "yes" branch of a decision based on condition  $c1 \ \& \ c2$ . A single execution of  $b \ c1$  and or

These repetitive parts can also be implemented without an exit command and are in D'-s. (see figure 2 b)

Finally consider the sequence  $b1 \ c1 \ b2 \ c2$  this requires two exits for its implementation. The alternatives  $c1 \ b2 \ c2 \ b1, \ b2 \ c2 \ b1 \ c1, \ \text{and} \ c2 \ b1 \ c1 \ b2$  all require an exit as illustrated in figure 2 c.

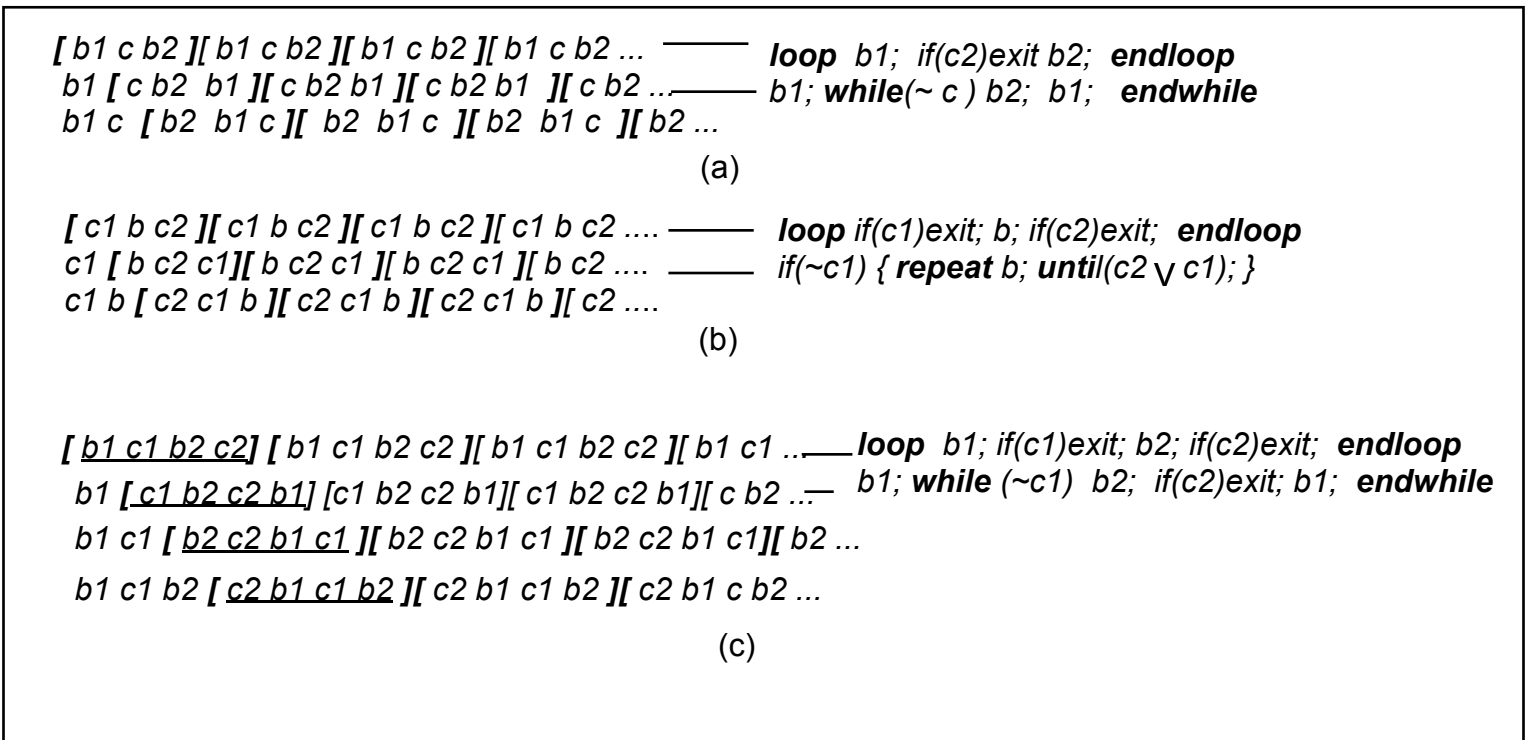


figure 2

This illustrates why the given the program with its exits has no equivalent program using a *while* and the same conditions,  $c1$  and  $c2$  employed in the *while-with-exits* implementation. However if we allow other conditions and/or boolean functions on  $c1$  and  $c2$  and the added conditions, an equivalent without exits. The following shows one way this can be done for a slightly more complex case in which the repeated sequence is  $C \ b1 \ C1 \ b2 \ C2 \ b3$ .

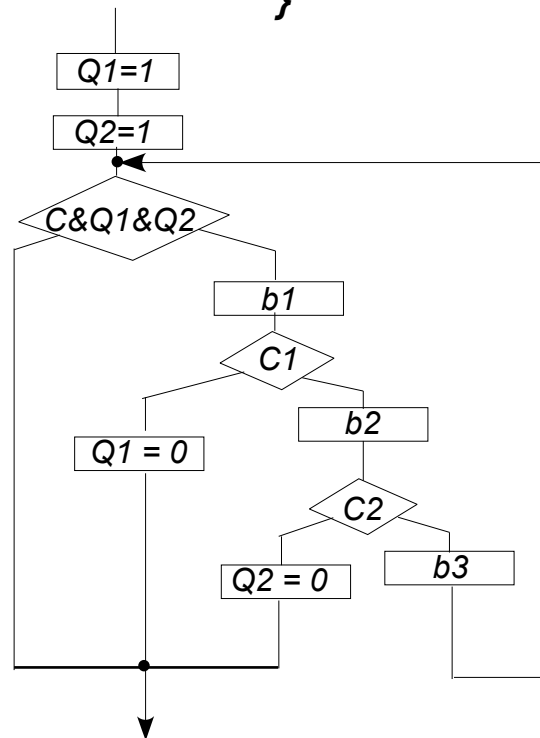
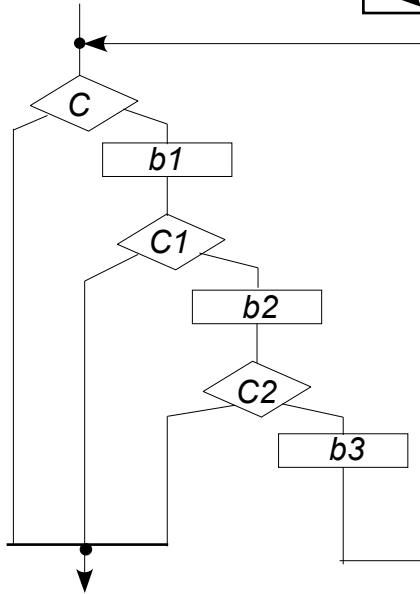
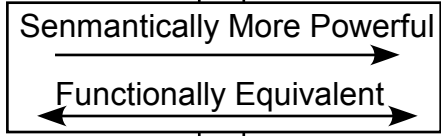
:  
 For any program that uses **gotos** and/or no **exits** there is a **functionally equivalent** one in D-S' so D-S'. However RE<sub>1</sub> a more powerful than D'-S.

```

while (C)
{ b1;
  if (~C1) then exit(1);
  b2;
  if (~C2) then exit(1) ;
  b3
}
  
```

```

Q1=1; Q2=1;
while (C&Q1&Q2)
{ b1;
  if(~C1) then Q1 = 0 ;
  else
  { b2;
    if(~C2) then Q2 = 0;
    else b3; };
}
  
```



In RE<sub>1</sub>

In D'-S

This **while** will only start if  $C \& Q1 \& Q2$  is true-otherwise it **exits**.  
 If it starts it does *b1* and will only continue if *C1* is true, otherwise it sets *Q1* to 0, arrives at the end of the **while** returning to the test of  $C \& Q1 \& Q$  which is 0 so it **exits**.  
 If it contnues it does *b2* and will only continue if *C1* is true, otherwise it sets *Q1* to 0, arrives at the end of the **while** returning to the test of  $C \& Q1 \& Q$  which is 0 so it **exits**.  
 If it contnues it does *b3* and return to *x*

Generally More Efficent Compiled Code  
 (Machine Language Is In L)

MORE COMPLICATED EXAMPLE SHOWS BIG ADVANTAGE FOR RE  
 EVEN FOR SHALLOW NESTING WITH MANY REASONS TO EXIT

## Control Structures Are Extended With the Existence of Function Calls, Returns and Exceptions(Throws and Catches):

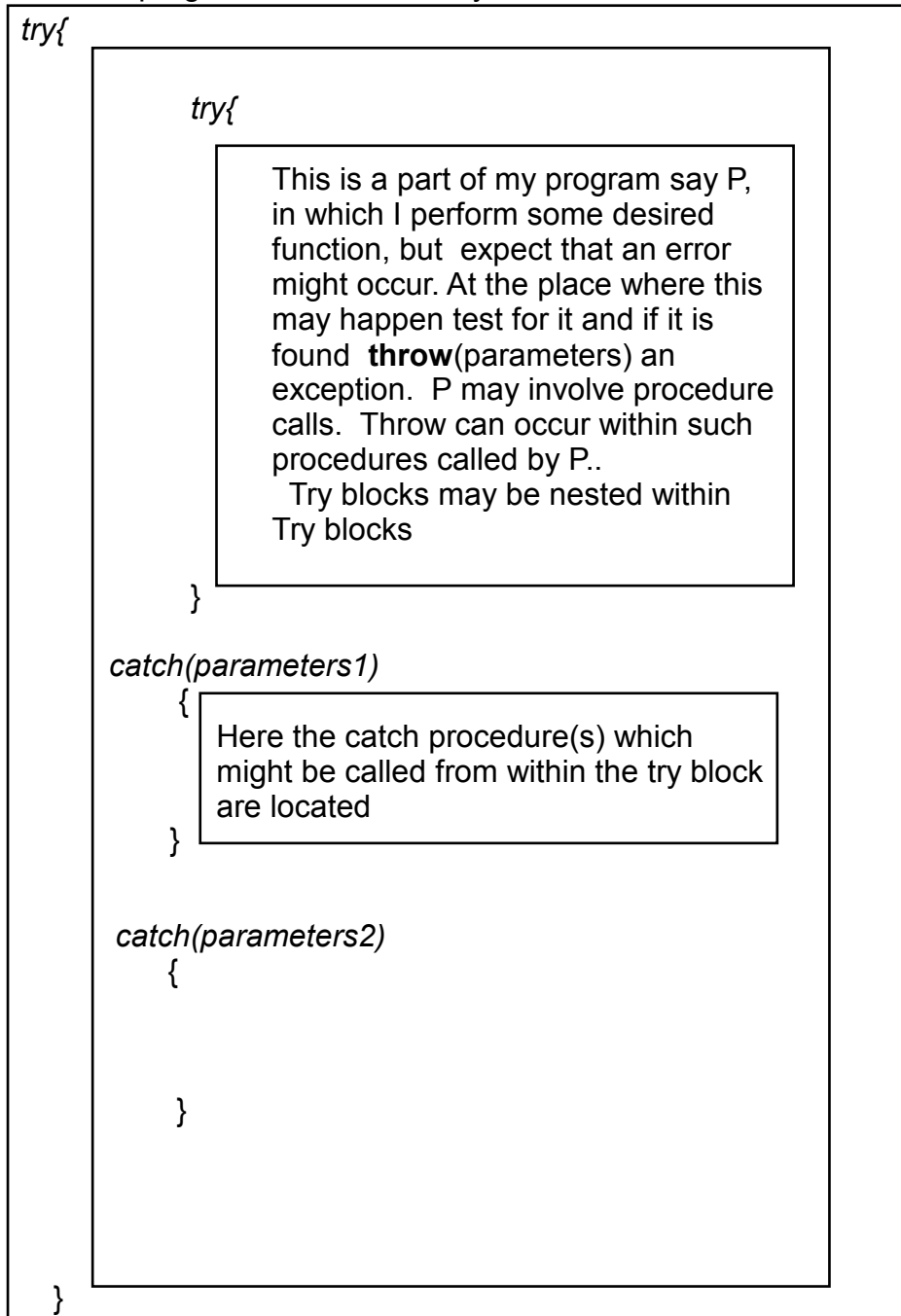
Transfers : Conditional **if (C) then {} else{}**, **if (C) then {}**, None-Conditional **exit(i)**, **goto**

Loops“ **while(C){,}; repeat{ } until( C), do{ }while( C)** Boolean C&/+C

Subroutines: **calls and return**

Exceptions: decl exception, **throws(raises)**, **catches(handlers)**

Language facilities for Exception Handling allow the programmer to organize important parts of error checking. Certain errors can occur at many places in a program Exception handling allows one to design one “error handler” to take the appropriate action when that error occurs, no matter where. For some errors the handler may terminate the program for others it may continue.



try blocks can be nested. A throw within a catch -throws the exception it receives to the next outer try block. A try may be in procedure Fj+1 called by procedure Fj if the corresponding catch is not found in Fj+1 it will be sought in Fj. In general the stack will be unwound until it is found-if never found entire program terminates

## Exceptions In C++

A program **throws** an exception at the point at which it is first detected. This can only happen within a **try** block. This causes a C++ program to search for a block of code called an **exception handler**, contained in a **catch block** which responds to the exception. This is called **catching** an exception. If there is no handler found the program terminates. **catch blocks** must be placed immediately after a try block.

Example C++

```
const int n = 50;
    int a[n];

insert(int i, int value)
{ if (i < 1) throw(1); // Detecting an exception index wrong
  if (value > 1000) {p = new int[n];
                    if(p==0) throw("NO SPACE IN HEAP") // Detecting an exception no space left
                    p[i] = value; }
  else a[i] = value;
}

void test()
{ int j;
  int anint;
  cout << "Enter Index and Value";
  cin >> j >> anint;
  try{ // Contains throws or calls functions which contain throws.
    insert(j, anint);
  }
  catch ( int r )
  {
    if r==1 cout << "rangerror in insert ( )" // Responds to throw at any level from try block.
    else cout << "other error">> ;
    abort;
  }
  catch ( const char* error ) {.....}
  {
  }
```

Termination (continue immediately after block or expression in which handler is found)  
and Resumption (pick up where you left off)

```
while(true)
  try
  {x = new X;
  break;}
  catch(bad_alloc)
  {collect_garbage(); }
```

Note goto's to labels outside the current subroutine can be used to locate handling problems outside the current subroutine. setjump and longjmp in C