SEQUENCE CONTROL

SEQUENCE CONTROL

Control Structure in a PL provides the basic framework within which operations and data are combined into a program and sets of programs.

Sequence Control -> Control of the order of execution of the operations
Data Control -> Control of transmission of data among subprograms of program

Sequence Control may be categorized into four groups:

1) **Expressions – They form the building blocks for statements.**
   An expression is a combination of variable constants and operators according to syntax of language. Properties as precedence rules and parentheses determine how expressions are evaluated
1) **Statements – The statements (conditional & iterative) determine how control flows from one part of program to another.**
2) **Declarative Programming – This is an execution model of program which is independent of the program statements. Logic programming model of PROLOG.**
3) **Subprograms – In structured programming, program is divided into small sections and each section is called subprogram. Subprogram calls and co-routines, can be invoked repeatedly and transfer control from one part of program to another.**

IMPLICIT AND EXPLICIT SEQUENCE CONTROL

Implicit Sequence Control

Implicit or default sequence-control structures are those defined by the programming language itself. These structures can be modified explicitly by the programmer.

eg. Most languages define physical sequence as the sequence in which statements are executed.

Explicit Sequence Control

Explicit sequence-control structures are those that programmer may optionally use to modify the implicit sequence of operations defined by the language.

eg. Use parentheses within expressions, or goto statements and labels

Sequence Control Within Expressions

Expression is a formula which uses operators and operands to give the output value.

i) Arithmetic Expression –

An expression consisting of numerical values (any number, variable or function call) together with some arithmetic operator is called "Arithmetic Expression".

Evaluation of Arithmetic Expression

Arithmetic Expressions are evaluated from left to right and using the rules of precedence of operators.

If expression involves parentheses, the expression inside parentheses is evaluated first

ii) Relational Expressions –

An expression involving a relational operator is known as "Relational Expression". A relational expression can be defined as a meaningful combination of operands and relational operators.

(a + b) > c c < b

Evaluation of Relational Expression

The relational operators <, >, <=, >= are given the first priority and other operators (== and !=) are given the second priority

The arithmetic operators have higher priority over relational operators.

The resulting expression will be of integer type, true = 1, false = 0

Sequence Control Within Expressions

iii) Logical Expression –

An expression involving logical operators is called 'Logical expression". The expression formed with two or more relational expression is called logical expression.

Ex. a > b && b < c

Evaluation of Logical Expression

The result of a logical expression is either true or false.

For expression involving AND (&&), OR (||) and NOT(!) operations, expression involving NOT is evaluated first, then the expression with AND and finally the expression having OR is evaluated.

Sequence Control Within Expressions

1. Controlling the evaluation of expressions

a) Precedence (Priority)

If expression involving more than one operator is evaluated, the operator at higher level of precedence is evaluated first

b) Associativity

The operators of the same precedence are evaluated either from left to right or from right to left depending on the level

Most operators are evaluated from left to right except

+ (unary plus), - (unary minus) ++, --, !, &

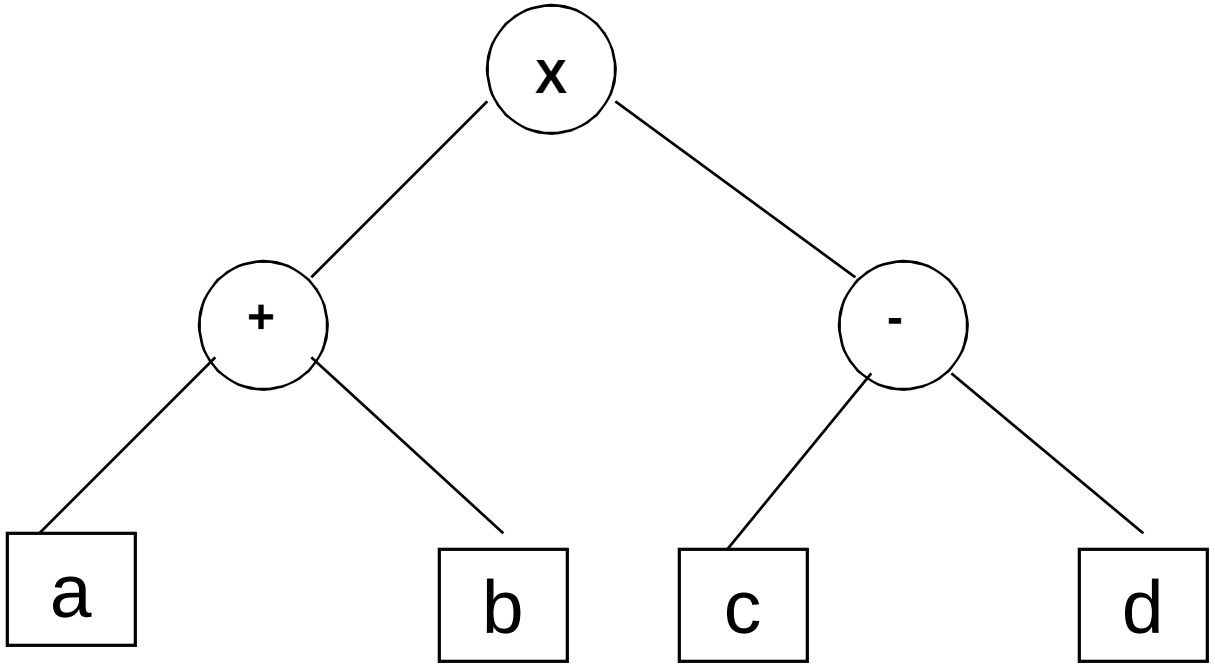
Assignment operators = , +=, *=, /=, %=

Sequence Control Within Expressions

2. Expression Tree

An expression (Arithmetic, relational or logical) can be represented in the form of an “expression tree”. The last or main operator comes on the top (root).

Example: (a + b) * (c – d) can be represented as



Sequence Control Within Expressions

3. Syntax for Expressions

a) Prefix or Polish notation

Named after polish mathematician Jan Lukasiewicz, refers to notation in which operator symbol is placed before its operands.

***XY, -AB, /*ab-cd**

Cambridge Polish - variant of notation used in LISP, parentheses surround an operator and its arguments.

(/(*ab)(-cd))

b) Postfix or reverse polish

Postfix refers to notation in which the operator symbol is placed after its two operands.

AB*, XY-

c) Infix notation

It is most suitable for binary (dyadic) operation. The operator symbol is placed between the two operands.

Sequence Control Within Expressions

4. Semantics for Expressions

Semantics determine the order of expression in which they are evaluated.

a) Evaluation of Prefix Expression

If P is an expression evaluate using stack

i) If the next item in P is an operator, push it on the stack. set the arguments count to be number of operands needed by operator.

(if number is n, operator is n-ary operator).

ii) If the next item in P is an operand, push it on the stack

iii) If the top n entries in the stack are operand entries needed for the last n-ary operator on the stack, apply the operator on those operands. Replace the operator and its n operands by the result of applying that operation on the n operands.

b) Evaluation of Postfix Expression

If P is an expression evaluate using stack

i) If the next item in P is an operand, push it on the stack.

ii) If the next item in P is an n-ary operator, its n arguments must be top n items on the stack. Replace these n items by the result of applying this operation using the n items as arguments.

Sequence Control Within Expressions

c) Evaluation of Infix Expression

Infix notation is common but its use in expression cause the problems:

i) Infix notation is suitable only for binary operations. A language cannot use only infix notation but must combine infix and postfix (or prefix) notations. The mixture makes translation complex.

ii) If more than one infix operator is in an expression, the notation is ambiguous unless parentheses are used.

Sequence Control Within Expressions

5. Execution-Time Representation:

Translators evaluate the expression using a method so as to get efficient result (optimum value at optimum time with optimum use of memory and processor).

Translation is done in two phases –

In first phase the basic tree control structure for expression is established. In next stage whole evaluation process takes place.

The following methods are used for translation of expression –

a) Machine code sequences

Expression can be translated into machine code directly performing the two stages (control structure establishment and evaluation) in one step.

The ordering of m/c code instructions reflect the control sequence of original expression.

b) Tree Structure

The expressions may be executed directly in tree structure representation using a software interpreter.

This kind of evaluation used in SW interpreted languages like LISP where programs are represented in the form of tree during execution

c) Prefix or postfix form

Problems with Evaluation of Expressions

1. Uniform Evaluation Code

Eager Evaluation Rule – For each operation node, first evaluate each of the operands, then apply the operation to the evaluated operands.

The order of evaluations shouldn't matter.

In C: A + (B = 0 ? C : C/B) --------- Problem

Lazy Evaluation Rule – Never evaluate operands before applying the operation. Pass the operands unevaluated and let the operation decide if evaluation is needed.

It is impractical to implement the same in many cases as it requires substantial software simulation.

LISP, Prolog use lazy rule.

In general, implementations use a mixture of two techniques.

LISP – functions split into two categories

SNOBOL – programmer-defined operations always receive evaluated operands

language-defined operations receive unevaluated operands

2. Side Effects

The use of operations may have side effects in expressions

c / func(y) + c

r-value of c must be fetched and func(y) must be evaluated before division.

If fun(y) has the side effect of modifying the value of c, the order of evaluation is critical.

Problems with Evaluation of Expressions

3. Short-circuit Boolean Expression

```
If ((X == 0) || ( Y/X < Z)  {……..}
do {……} while (( I > UB) && (A[I] < B))
```

Evaluation of second operand of Boolean expression may lead to an error condition (division by zero, subscript range error).

In C -- The left expression is evaluated first and second expression is evaluated only when needed.

In many languages, both operands are evaluated before boolean expression is Evaluated

ADA includes two special Boolean operations

and then , or else

```
if ( X = 0) or else (Y/X > Z) then
        can't fail
```

Sequential Control within Statement

1. Basic Statements

i) Assignment Statement

Assignment operator (=), compound assignment operator (+=)

MOVE A TO B. - COBOL

ii) Input and Output Statement

printf, scanf

iii) Declaration Statement

int age;

iv) GoTo statement

Explicit sequence control statement. Used to branch conditionally from one point to another in the program

```
int a, b;
Read:
    scanf ("%d", &a);
    if (a == 0) goto Read;
    y = sqrt(x);
    prinf("%d", y);
    goto Read;
```

Sequential Control within Statement

1. Basic Statements

v) Break Statement

An early exit from a loop can be accomplished by using break statement.

2. Statement Level Sequence Control

i) Implicit Sequence Control

The natural or default programming sequence of a PL is called implicit sequence. They are of 3 types.

a) Composition Type

Standard form of implicit sequence. Statements placed in order of execution.

b) Alternation Type

There are two alternate statement sequence in the program, the program chooses any of the sequence but not both at same type

c) Iteration Type

Here normal sequence is given to statements but the sequence repeats itself for more than one time.

ii) Explicit Sequence Control

The default sequence is altered by some special statements

a) Use of Goto statement **b) Use of Break Statement**

Sequential Control within Statement

3. Structured Sequence Control

a) Compound Statement

Collection of two or more statements may be treated as single statement.

```
begin          /* ----- Pascal        {                    /* C
    ……..                                  ……..
end                                     }
```

b) Conditional Statements

```
if (conditional exp)  then …….statements  endif
 if (conditional exp)  then …….statements  else  …..statements endif
 if (conditional exp)  then …….statements
    elseif (conditional exp) then … statements
    else  …. statements …endif
switch (exp) { case val1:  …statements break;
                           val2: ….statetments break;
                          default:  statements break;}
```

c) Iteration Statements

```
do {…….} while (conditional exp)
while (conditional exp) { ……….}
for (initialization; test condition; increment) { …….}
```

Subprogram Sequence Control

Subprogram sequence control is related to concept:

How one subprogram *invokes* another and called subprogram returns to the first.

Simple Call-Return Subprograms

- **Program is composed of single main program.**
- **During execution It calls various subprograms which may call other subprograms and so on to any depth**
- **Each subprogram returns the control to the program/subprogram after execution**
- **The execution of calling program is temporarily stopped during execution of the subprogram**
- **After the subprogram is completed, execution of the calling program resumes at the point immediately following the call**

Copy Rule

The effect of the call statement is the same as would be if the call statement is replaced by body of subprogram (with suitable substitution of parameters)

We use subprograms to avoid writing the same structure in program again and again.

Subprogram Sequence Control

Simple Call-Return Subprograms

The following assumptions are made for simple call return structure

i) Subprogram can not be recursive

ii) Explicit call statements are required

iii) Subprograms must execute completely at call

iv) Immediate transfer of control at point of call or return

v) Single execution sequence for each subprogram

Implementation

1. There is a distinction between a subprogram definition and subprogram activation.

Subprogram definition – The written program which is translated into a template.

Subprogram activation – Created each time a subprogram is called using the template created from the definition

2. An activation is implemented as two parts

   Code Segment – contains executable code and constants

   Activation record – contains local data, parameters & other data items

3. The code segment is invariant during execution. It is created by translator and stored statically in memory. They are never modified. Each activation uses the same code segment.

4. A new activation record is created each time the subprogram is called and is destroyed when the subprogram returns. The contents keep on changing while subprogram is executing

Subprogram Sequence Control

Two system-defined pointer variables keep track of the point at which program is being executed.

Current Instruction Pointer (CIP)

The pointer which points to the instruction in the code segment that is currently being executed (or just about to be) by the hardware or software interpreter.

Current Environment Pointer (CEP)

Each activation record contains its set of local variables. The activation record represents the "referencing environment" of the subprogram.

The pointer to current activation record is Current Execution Pointer.

Execution of Program

First an activation for the main program is created and CEP is assigned to it. CIP is assigned to a pointer to the first instruction of the code segment for the subprogram.

When a subprogram is called, new assignments are set to the CIP and CEP for the first instruction of the code segment of the subprogram and the activation of the subprogram.

To return correctly from the subprogram, values of CEP and CIP are stored before calling the subprogram. When return instruction is reached, it terminates the activation of subprogram, the old values of CEP and CIP that were saved at the time of subprogram call are retrieved and reinstated.

Recursive Subprograms

Recursive Subprograms

Recursion is a powerful technique for simplifying the design of algorithms.

Recursive subprogram is one that calls itself (directly or indirectly) repeatedly having two properties

a) It has a terminating condition or base criteria for which it doesn't call itself

b) Every time it calls itself, it brings closer to the terminating condition

In Recursive subprogram calls A subprogram may call any other subprogram including A itself, a subprogram B that calls A or so on.

The only difference between a recursive call and an ordinary call is that the recursive call creates a second activation of the subprogram during the lifetime of the first activation.

If execution of program results in chain such that 'k' recursive calls of subprogram occur before any return is made. Thus 'k+1' activation of subprogram exist before the return from k^{th} recursive call.

Both CIP and CEP are used to implement recursive subprogram.

Exception and Exception Handlers

Type of Bugs -

Logic Errors – Errors in program logic due to poor understanding of the problem and solution procedure.

Syntax Errors – Errors arise due to poor understanding of the language.

Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing.

eg. Divide by zero, access to an array out of bounds, running out of memory or disk space

When a program encounters an exceptional condition, it should be Identified and dealt with effectively.

Exception and Exception Handlers

Exception Handling –

It is a mechanism to detect and report an 'exceptional circumstance" so that appropriate action can be taken. It involves the following tasks.

- **Find the problem (Hit the exception)**
- **Inform that an error has occurred (Throw the exception)**
- **Receive the error information (catch the expression)**
- **Take corrective action (Handle the exception)**

```
main()
{  int x, y;
    cout << "Enter values of x and y";
    cin >>x>>y;
   try {
          if (x != 0)
             cout "y/x is ="<<y/x;
          else
             throw(x);
      }
  catch (int i)  {
     cout << "Divide by zero exception caught";
    }
}
```

Exception and Exception Handlers

try – Block contains sequence of statements which may generate exception.

throw – When an exception is detected, it is thrown using throw statement

catch – It's a block that catches the exception thrown by throw statement and handles it appropriately.

catch block immediately follows the try block.

The same exception may be thrown multiple times in the try block.

There may be many different exceptions thrown from the same try block.

There can be multiple catch blocks following the same try block handling different exceptions thrown.

The same block can handle all possible types of exceptions.

```
catch(…)
{
        // Statements for processing all exceptions
}
```

Exception and Exception Handlers

```
procedure sub1()
  divide_zero exception;
  wrong_array_sub exception;
 ---------------- other exceptions
begin
   -----------
   if x = 0 then raise divide_zero;
   -----------
exception
    when divide_zero =>
         ---------------- handler for divide-zero
    when array_sub =>
        ----------------- handler for array sub
end;
```

Exception and Exception Handlers

Propagating an Exception –

If an handler for an exception is not defined at the place where an exception occurs then it is propagated so it could be handled in the calling subprogram. If not handled there it is propagated further.

If no subprogram/program provides a handler, the entire program is terminated and standard language-defined handler is invoked.

After an exception is handled –

What to do after exception is handled?

Where the control should be transferred?

Should it be transferred at point where exception was raised?

Should control return to statement in subprogram containing handler after it was propagated?

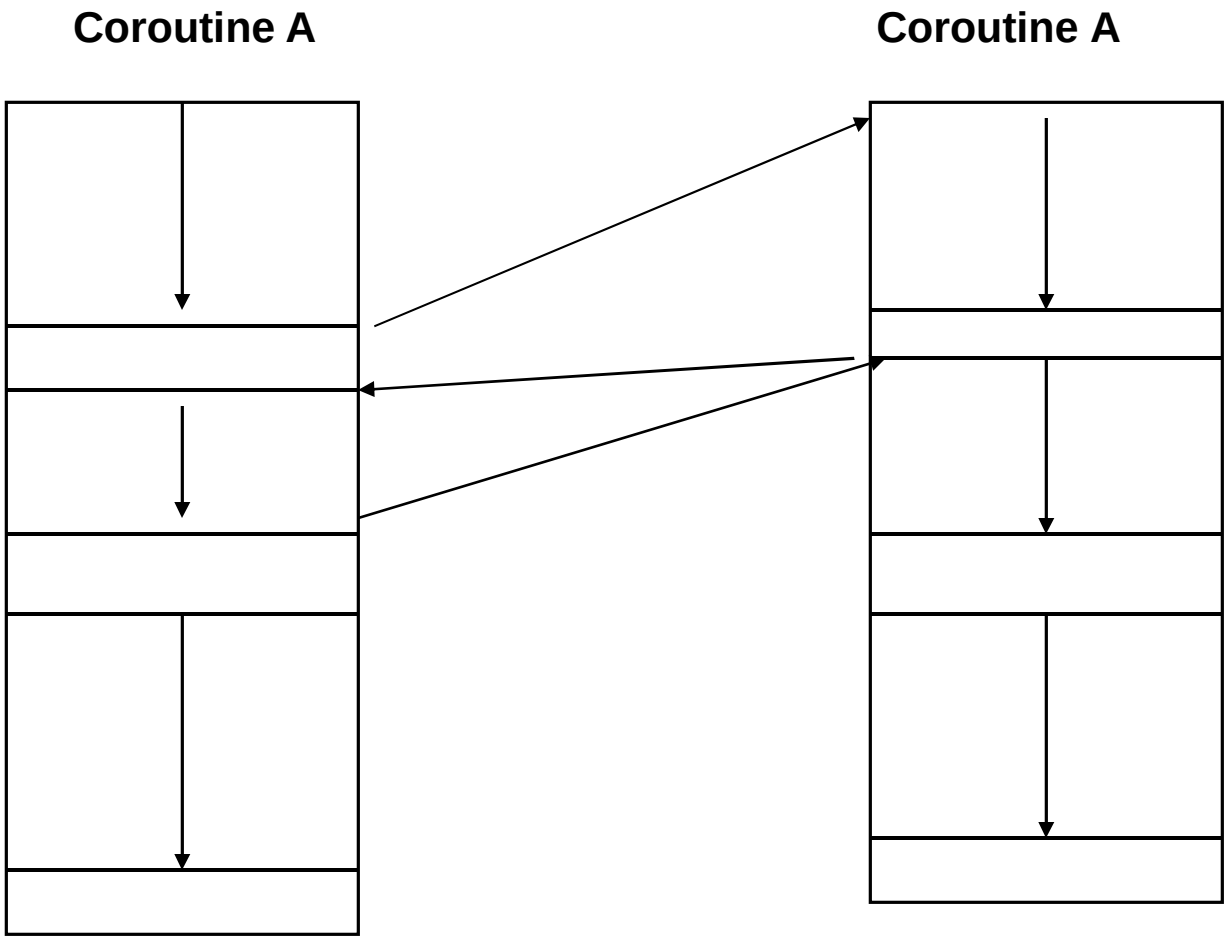
Should subprogram containing the handler be terminated normally and control transferred to calling subprogram? – ADA

Depends on language to language

COROUTINES

COROUTINES –

Coroutines are subprogram components that generalize subroutines to allow multiple entry points and suspending and resuming of execution at certain locations.



COROUTINES

Comparison with Subroutines

1. The lifespan of subroutines is dictated by last in, first out (the last subroutine called is the first to return); lifespan of coroutines is dictated by their use and need,
2. The start of the subroutine is the only point entry. There might be multiple entries in coroutines.
3. The subroutine has to complete execution before it returns the control. Coroutines may suspend execution and return control to caller.

Example: Let there be a consumer-producer relationship where one routine creates items and adds to the queue and the other removes from the queue and uses them.

```
var q := new queue
coroutine produce
  loop
     while q is not full
          create some new items
          add item to q
          yield to consume
```

```
coroutine consume
  loop
     while q is not empty
          remove some items from q
          use the items
     yield to produce
```

COROUTINES

Implementation of Coroutine

Only one activation of each coroutine exists at a time.

A single location, called *resume point* is reserved in the activation record to save the old ip value of CIP when a resume instruction transfer control to another subroutine.

Execution of resume B in coroutine A will involve the following steps:

- **The current value of CIP is saved in the resume point location of activation record for A.**

The ip value in the resume point location is fetched from B's activation record and assigned to CIP so that subprogram B resume at proper location

SCHEDULED SUBPROGRAMS

Subprogram Scheduling

Normally execution of subprogram is assumed to be initiated immediately upon its call

Subprogram scheduling relaxes the above condition.

Scheduling Techniques:

1. Schedule subprogram to be executed before or after other subprograms.

call B after A

2.Schedule subprogram to be executed when given Boolean expression is true

call X when Y = 7 and Z > 0

3. Schedule subprograms on basis of a simulated time scale.

call B at time = Currenttime + 50

4. Schedule subprograms according to a priority designation

call B with priority 5

Languages : GPSS, SIMULA