Refers to the flow of control of execution. Every programming language provides several mechanisms: Sequencing, selection, repetition/iteration, procedures/functions, recursion, concurrency, exception handling, non-determinacy.

**Expressions**
- Literals: 32767, true, -4.02, "Hello"
- Basic expressions: \( a + b, a \cdot b, a \times b, \frac{a}{b} \), \( a \leq b, \frac{a}{b}, a \oplus b \), \( a \oplus b \)

Infix expr: \( operand1 \text{ operator } operand2 \) binary
- \( operand \text{ operator } operand \) unary

Prefix expr: \( \text{operator} \ operand1 \ operand2 \)
- \( \text{operator} \ operand1 \ operand2 \)

Postfix expr: \( \text{operand1 \ operand2 \ operator} \)
- \( \text{operand1 \ operand2 \ operator} \)

For example, in LISP:
- \((+\ a\ b)\)
- \((\times\ (+\ 1\ 3)\ 2)\) = \((1+3)\times2\)

**Precedence Rules**
- C: ++, --, post, ++, i, --
- Java: !
- Python: **, *, /, %, +, -
- C++: , <, <=, ==, !=, &&, ||
- Java: , <, <=, ==
- Python: not and or lambda
\[ \| f \|_{L^1} = \int f(x) \, dx \]

\[ \mathcal{W} \]
Control Flow - Conditional Operator (?:)

**Left associative** → \( a + b + c \)

**Right-associative** → \( x ** y * * y \)

\((2^4)^2 \approx 2^{(4*2)} \approx 2^8 \)

\( (x ** y) ** y \approx x ** (y ** y) \)

**Conditional Operator** ?:: \( a ? b : c \)

**Ternary operator**

\[
\begin{align*}
\text{if } (a > b) & \quad \text{max} = a \\
\text{else} & \quad \text{max} = b
\end{align*}
\]

**Imperative PLs**

- Computation is carried out via expressions + assignment statements

**C, C++, Java, C#, Javascript, Swift**

\[
\text{max} = \text{if} a > b ? a : b \text{ else } b
\]

**Python, Go, Haskell, Common Lisp**

\[
\begin{align*}
\text{max} &= \text{if} (\text{cond} \ (> \ a \ b) \ a \ b) \\
&= \text{if} (> \ a \ b) \ a \ b
\end{align*}
\]

**Rust**

\[
\text{let max} = \text{if} a > b \Rightarrow \text{else other}
\]
7 Control Flow - Assignment (l-values and r-values)

Assignment:
\[
<\text{var}> \triangleq <\text{expression}>
\]
\[
<\text{var}> \leftarrow <\text{expression}>
\]
\[
<\text{var}> := <\text{expression}>
\]

LHS: var. name refers to its location
RHS: var. name refers to its value

Two Models of Variables

\[
\begin{align*}
\text{int} & \quad b = 2; \\
\text{int} & \quad c = b; \\
\text{int} & \quad a = c + b;
\end{align*}
\]

Value Model: variable name is a named container for its value
Reference Model: variable name is a named reference to its value. Every name is an l-value

\[
c = a;
\]
\[
c = 3;
\]

Java: value model for all built-in types, reference model for all class objects

C#: class objects - reference, struct - value

C/C++:

\[
a = c + b \quad \text{| Explicit dereferencing} \quad = &a;
\]
Implicit Derivative
\[ a = c + b \]

Explicit Derivative
\[ \frac{d}{dx} a = 8a \]
Java

```java
int [] a = \{3, 1, 2, 3\};

c for (int i : a):
    a.add(x);
```

```java
// This is OK!
ArrayList<Integer> A = new ArrayList<Integer>;

for (int x : a):
    A.add((Integer) x);
    A.add(new Integer(x));
```

**Orthogonality**: Design goal for all DLS.

Any feature can be used in combination with any other feature.

<table>
<thead>
<tr>
<th>ARM orthogonal</th>
<th>Instruction Set</th>
<th>Small</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ ADD A, B, C</td>
<td>Direct DADD</td>
<td>ADD</td>
</tr>
<tr>
<td>ADD @A, @B, @C</td>
<td>Indirect ADDI</td>
<td>A1B1C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addressing Modes</th>
<th>CISC processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>huge</td>
</tr>
</tbody>
</table>
small

RISC Processor (Reduced Instruction Set Computer) | CISC processor is complex