Goal of Talk

• The goal of this talk is to talk about expressions and the flow of programs
Control Flow

- **Control flow** is the order in which a program executes.
- For imperative languages (e.g., Java), this is fundamental.
- For other programing paradigms (e.g., functional), the compilers/interpreters take care of ordering.
Expression Evaluation

• An **expressions** consist of a **simple object** (e.g., a variable), an **operator**, or a **function** applied to a collection of objects and/or operators.

• Expression evaluation is a **crucial** component of **functional** languages.
Functional languages are very “math-like” and in math a primary concept is evaluating expressions.

Expression evaluation is a crucial component of functional languages.
Operators

- Operators are used in
  - **Prefix** notation: operators come first
    - \((* (+ 1 3) 2)\)
  - **Infix** notation: operators in middle
    - \((1+3)*2\)
  - **Postfix** notation: operators last
    - \(a++\)
Operators-Precedence

- **Precedence** rules specify the order in which operators of **different** precedence levels are evaluated.
  - e.g. Multiplication before addition.

- Precedence in **boolean** expressions **very** important
  - The phrase “if A<B and C<D” can be read as:
    - if (A<B) and (C<D)
    - if (A< (B and C)) <D
Operators--Associativity

• **Associativity** rules specify the order in which operators of the **same** precedence level are evaluated.
  • Usually they are evaluated “left-to-right”

• In Fortran, ** associates from **right-to-left**
  • $x^{y} = x^y$
  • Thus $2^{3^4}$ is read as $2^{(3^4)}$ rather than $(2^3)^4$.

• Also assignment in C
  • $a = b = c$
Assignment

- The basic operation language is **assignment**.
- An assignment places a **value** into a **specific memory location**.

![Diagram showing the effect of an assignment operation on memory]

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036

```
a = 2;
```

Before | After
--- | ---
2036 | 2036

Before | After
--- | ---
2036 | 2036
As a result, assignments have longevity and can exist beyond their original context.

```
a = 2;
```

Before:

<table>
<thead>
<tr>
<th>a</th>
<th>2036</th>
</tr>
</thead>
</table>

After:

<table>
<thead>
<tr>
<th>a</th>
<th>2036</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
Context

• To see the difference between context consider the two following statements.

Imperative

```c
int sum(int n){
    int val=0;
    for(int i=0,i<=n;i++){
        val+=i;
    }
    return val;
}
```

Functional

```c
int sum(int n){
    if (n<=0) then
        return 0
    else
        return n+sum(n-1)
}
```
To see the difference between context consider the two following statements.

```cpp
int sum(int n){
    int val=0;
    for(int i=0;i<=n;i++){
        val+=i;
    }
    return val;
}
```

```cpp
int sum(int n){
    if (n<=0) then
        return 0
    else
        return n+sum(n-1)
}
```

In the imperative code the value of `val` changes within the context of `sum`.
To see the difference between context consider the two following statements.

**Imperative**

```c
int sum(int n){
    int val=0;
    for(int i=0;i<=n;i++){
        val+=i;
    }
    return val;
}
```

**Functional**

```c
int sum(int n){
    if (n<=0) then
        return 0
    else
        return n+sum(n-1)
}
```

In the functional code the value of \( n \) changes but only between contexts of \( \text{sum} \).
Variables

• Two ways to model variables:
  • Value model
  • Reference model
Value Model

• Under the **value model** variables on the **left-hand side** (called **l-values**) of equations denote **references**, and variables on the **right-hand side** (called **r-values**) denote **values**.

```
b = 2;
a = b;
mem(1024) = 2;
mem(2036) = 2;
```
Under the value model, variables on the left-hand side (called l-values) of equations denote references, and variables on the right-hand side (called r-values) denote values.

Pascal and C use this model.
Reference Model

- Under the **reference model** variables on both the left- and right-hand side are references.

\[
b = 2; \\
a = b; \\
mem(1024) = loc(2); \\
mem(2036) = mem(1024);
\]
Lisp, Clu use this model.

```
b = 2;
a = b;
mem(1024) = loc(2);
mem(2036) = mem(1024);
```
Expressions: Initialization

• Variable initialization can be **implicit** or **explicit**.
  
  • **Implicit**: variables are initialized as they are used (e.g., Perl).
    
    ```
    $a += 3;
    ```

  • **Explicit**: variables are initialized by the programmer (e.g., C).
    
    ```
    int a = 0;
    a += 3;
    ```

• Java, C# require **definite assignment**

  • Variables must be assigned a value before they are used in expressions
Expressions: Orthogonality

- **Orthogonality** means that features can be used in any combination and the meaning is consistent regardless of the surrounding features.
  - Good idea in principle, but requires careful thought.
  - e.g. assignment as an expression.
    - unfortunate when combined with poor syntactic choices, as in C:

```c
if(a=b){ }
```

```c
if(a==b){ }
```
Expressions: Complication

- Execution ordering within expressions is **complicated by side effects** (and code improvements)
  - e.g., in C

```c
b=1;
int inc(int a) {
    b+=1;
    return a+1;
}
c = (3*b) * inc(b);
```

- If inc(b) is evaluated before (3*b), the final value of c is 12. If the (3*b) is evaluated first, then the value is c is 6.
Expressions: Short-Circuit

• Expressions may be executed using short-circuit evaluation

```c
p = my_list;
while (p && p->key != val)
    p = p->next
```
Expressions: Short-Circuit

• Expressions may be executed using short-circuit evaluation

```c
p = my_list;
while (p && p->key != val)
    p = p->next
```

"if \( p = \text{null} \), then \( p->\text{key} \) is never checked. Thus, it is "short-circuited"."
Expressions: Short-Circuit

- Expressions may be executed using short-circuit evaluation

```pascal
p = my_list;
while (p && p->key != val)
p = p->next
```

```pascal
p := my_list;
while (p<>nil) and (p^.key <> val) do
p := p^.next
```

Since Pascal does not have short circuiting, this will check both. Thus, if `p=nil`, then `p^.key` will return an error.