

CSC 347 - Concepts of Programming Languages

Statements and Expressions

Instructor: James Riely



Learning Objectives

- ❓ What should be the basic building blocks of computations?
 - Identify different ways of expressing computations
 - Identify the difference between statement sequences and compound expressions



Does it Translate?

Scala

```
1 def f (x: Int) : Int =  
2   val y: Int = if x!=0 then 1 else 2  
3   y  
4 end f
```

- Conditionally set a variable

Is this Java/C?

```
1 int f (int x) {  
2   int y = if (x!=0) 1 else 2;  
3   return y;  
4 }
```

- ⚠ In Java/C, `if ... else` is statement language, not expression language



Does it Translate?

Java/C

```
1 int f (int x) {  
2     int y = x!=0 ? 1 : 2;  
3     return y;  
4 }
```

- Conditionally set a variable

Is this Scala?

```
1 def f (x: Int) : Int =  
2     var y: Int = (x!=0 ? 1 : 2)  
3     y  
4 end f
```

⚠ Ternary operator not in Scala expression language

💡 `if ...` in Scala expression language!



Does it Translate?

Scala

```
1 def f (x: Int) : Int =  
2   val y: Int = {  
3     var i = x  
4     var z=0  
5     while i>0 do {i=i-1; z=z+1}  
6     z  
7   }  
8   y  
9 end f
```

- Set value of `y` with a compound expression

Is this Java/C?

```
1 int f (int x) {  
2   int y = {  
3     int i=x;  
4     int z=0;  
5     while (i>0) {i--; z++;}  
6     return z;  
7   }  
8   return y;  
9 }
```

-  Loops and `;` are not in the Java/C expression language
- How can we make it Java/C?



Does it Translate?

Scala

```
1 def g (x: Int) : Int =
2   var i = x
3   var z = 0
4   while i>0 do { i=i-1; z=z+1 }
5   z
6 end g
7
8 def f (x: Int) : Int =
9   val y = g(x)
10  y
11 end f
```

Java/C

```
1 int g (int x) {
2   int i=x;
3   int z=0;
4   while (i>0) { i--; z++; }
5   return z;
6 }
7
8 int f (int x) {
9   int y = g(x);
10  return y;
11 }
```

- Functions are in the expression language of Scala, Java, and C



Statements and Expressions

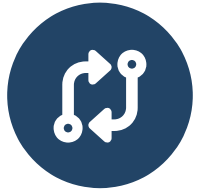
-  Assembly language consists of *statements*

```
1 mov  eax, 5  
2 add  eax, 6  
3 mov  ebx, eax
```

-  *Expressions* are a more abstract way of expressing computations

```
1 5+6
```

- Many imperative PL distinguish statement language from expression language
- Functional languages tend to emphasize expressions (Scala has only expressions)



Pure vs Side-Effecting Expressions

- A mathematical function takes arguments and gives results
- An expression is *pure* if that is all it does
- Pure expression does not modify store ξ
$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle e_1 + e_2, \xi_0 \rangle \Downarrow \langle v_1 + v_2, \xi_2 \rangle} \text{ (Add)}$$
- Side-effecting expression modifies store ξ
$$\frac{\langle e, \xi_0 \rangle \Downarrow \langle v, \xi_1 \rangle}{\langle x := e, \xi_0 \rangle \Downarrow \langle v, \xi_1 \{x \mapsto v\} \rangle} \text{ (:=)}$$
- Anything else is a *side effect*
 - Assignment to a variable
 - Change of control (goto)
 - I/O (console, network)
 - etc.



Expressions

- Literals (boolean, character, integer, string)
- Operators (arithmetic, bitwise, logical)
- Function calls

```
1 f(1 + 2 * "hello".length)
```

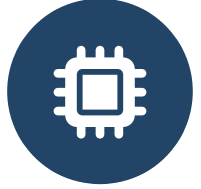
- Assignment

```
1 var y = 0  
2 val x = y = 5
```

```
1 int y = 0;  
2 int x = y = 5;
```

❓ Type and value of `x` ?

💡 In Scala, everything is an expression!



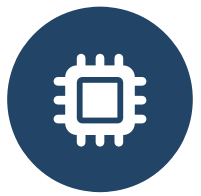
Statements in C

- Expression statements (including assignment)

```
1 printf("hello");  
2 ^^^^^^^^^^^^^^^  
3 ^^^^^^^^^^^^^^^ expression  
                    statement
```

- Return statements

```
1 return 1+x;  
2      ^^^  
3 ^^^^^ expression  
        statement
```



Statements in C

- Selection statements (if-then-else; switch-case)
- Iteration statements (while; do-while; for)

```
1 int count = 0;
2 while (1) {
3     int ch = getchar();
4     switch (ch) {
5         case -1: return count;
6         case 'a': count = count + 1;
7         default: continue;
8     }
9 }
```

⚠ Cannot use statements verbatim as part of expressions

💡 Use functions to turn statements into expressions



Side-Effecting Expressions in C

```
1 int x = 1;
2 printf ("%d\n", ++x);    // pre increment, prints 2
3 // value of x is now 2
```

```
1 int x = 1;
2 printf ("%d\n", x++);    //
3 //
```

```
1 x = 1 + (y = 5);        //
```

```
1 int x = 1;
2 printf ("%d\n", (x = x + 1) + x); //
```



Side-Effecting Expressions in C

```
1 int x = 1;
2 printf ("%d\n", ++x);    // pre increment, prints 2
3 // value of x is now 2
```

```
1 int x = 1;
2 printf ("%d\n", x++);    // post increment, prints 1
3 // value of x is now 2
```

```
1 x = 1 + (y = 5);        //
```

```
1 int x = 1;
2 printf ("%d\n", (x = x + 1) + x); //
```



Side-Effecting Expressions in C

```
1 int x = 1;
2 printf ("%d\n", ++x);    // pre increment, prints 2
3 // value of x is now 2
```

```
1 int x = 1;
2 printf ("%d\n", x++);    // post increment, prints 1
3 // value of x is now 2
```

```
1 x = 1 + (y = 5);         // assigns 5 to y and 6 to x
```

```
1 int x = 1;
2 printf ("%d\n", (x = x + 1) + x); //
```



Side-Effecting Expressions in C

```
1 int x = 1;
2 printf ("%d\n", ++x);    // pre increment, prints 2
3 // value of x is now 2
```

```
1 int x = 1;
2 printf ("%d\n", x++);    // post increment, prints 1
3 // value of x is now 2
```

```
1 x = 1 + (y = 5);         // assigns 5 to y and 6 to x
```

```
1 int x = 1;
2 printf ("%d\n", (x = x + 1) + x); // no "sequence point", undefined!
```

💡 *Sequence point*: A point in the execution of a C program at which all previous side effects are guaranteed to be complete.



Side-Effects in OOP

- Side effects are common in object-oriented programming

```
1 class C {  
2     private int x = 0;  
3     private int y = 0;  
4     public int f(int z) {  
5         x = x-z;  
6         return x;  
7     }  
8     public int g() {  
9         y = y+5;  
10        return y;  
11    }  
12 }  
13 C c = new C();  
14 c.f(c.g()); // same as x -= (y += 5)
```

- Results often depend on object state → potentially on entire execution history



Sequencing in C Expressions

- Operator `,` creates a sequence of expressions: last expression provides value

Statement sequence

```
1 int main () {  
2     int x = 5;  
3     x *= 2;  
4     printf ("%d\n", x);  
5 }
```

Expression sequence

```
1 int main () {  
2     int x = 5;  
3     printf ("%d\n", (x *= 2, x));  
4 }
```



Summary

Statements

- Change memory
- Are executed in sequence

Expressions

- Pure vs. side-effecting
- Sequencing by operator in expression language, e.g., C `e1, e2, ... en`
- Conditional by operator in expression language, e.g., C `e1 ? e2 : e3`