

CSC 347 - Concepts of Programming Languages

Statements and Expressions

Instructor: Stefan Mitsch



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



Is this Accepted Syntax?

Is this Scala?

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

- Enter in REPL to find out

Is this C?

```
1 int f (int x) {  
2   int y;  
3   if (x!=0) y=1; else y=2;  
4   return y;  
5 }
```

- Compile to find out



Is this Accepted Syntax?

Is this Scala?

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

Is this C?

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

⚠ Not accepted in C, `if ... else` is statement language, not expression language



Is this Accepted Syntax?

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!

Is this C?

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



Is this Accepted Syntax?

Is this Scala?

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

Is this C?

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

-  Loops and variable declarations are not in the C expression language
- How can we make it C?



Is this Accepted Syntax?

Is this Scala?

```
1 def g (x: Int) : Int =  
2   var xx = x  
3   var z = 0  
4   while xx>0 do { xx=xx-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
```

Is this C?

```
1 int g (int x) {  
2   int xx=x;  
3   int z=0;  
4   while (xx>0) { xx--; 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 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
- Anything else is a *side effect*
 - Assignment to a variable
 - Change of control (goto)
 - I/O (console, network)
 - etc.

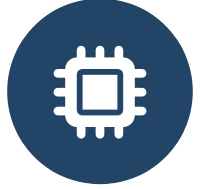


Expressions

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

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

💡 Scala does not have statements, everything is an expression!



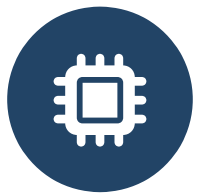
Statements in C

- 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

- ? Name some side-effecting expressions in C

- Post-increment `x++`
- Add and assign `x += 2`
- Assignment `x = (y = 5)`
- Combined `x -= (y += 5)`

- Side effects common in OOP

```
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



Side-Effecting Expressions in C

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

```
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);    //  
3
```



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);    //
3
```



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);    //
3
```



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);    //
3
```



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!
3 // in OOP often disguised: o.f() + o.g()
```



Sequencing in Expressions

- Scala `{ e1; e2; ...; en }`

```
1 {  
2   e1  
3   e2  
4   ...  
5   en  
6 }
```

- `e1` ... `en-1` executed for side effect
- Result is the value of `en`

- C `( e1, e2, ..., en )`

```
1 (  
2   e1,  
3   e2,  
4   ...  
5   en  
6 )
```

- ? Why have statement and expression sequences?
- A C [example](#):

```
1 string s;  
2 while(read_string(s), s.len() > 5) {  
3   // do something  
4 }
```



Sequencing in C Expressions

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

```
1 int main () {  
2     int x = 5;  
3     printf ("%d\n", (x *= 2, x));  
4     // behavior defined because comma operator introduces sequence point  
5 }
```



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`