CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (with slight modifications): Ira Woodring

Expressions and Assignment Statements (Follows the Sebesta Text Chapter 7)

Overview

Expressions are the way we specify computations in a programming language.

Issues that affect expressions (and therefore need to be addressed by language creators) are associativity and precedence, type mismatches, coercions, and short-circuit evaluations.

Functional Languages

It is important to note that functional languages are similar but not the same as imperative languages when it comes to expressions and assignments.

An "assignment" in a functional language is really a **declaration** that binds a value to a name (functional languages don't have variables; once a value is assigned it stays).

Additionally, the evaluation of an expression in a functional language can not have side effects.

Arithmetic Expressions

One of the original reasons computers were created was for the automatic evaluation of statements. We wanted to be able to solve math, science, and other types of problems without the slow human element.

Arithmetic Expressions

To that extent, many expressions in Computer Science mirror expressions from Mathematics:

- operators
- operands
- parentheses
- functions

Operators

Operators are labeled based on the number of operands they function on.

Unary operators have only one operand.

Binary operators have two.

Ternary operators have three.

Operators

Some Unary Operators:

```
int x = 41;
x++;

x--;

y = &x;
```

Operators

Binary are more what we are used to:

```
int x = 21;
int y = 21;

int z = x + y;

z << 1;
```

Operators

Ternary:

```
value = a.height < b.height ? x : y
```

Which is a substitute for:

```
if(a.height < b.height){
  value = x;
} else {
  value = y;
}
```

A Computer Science oddity... Some languages have a binary operator that works much like the ternary operator. This operator is called the Elvis Operator and works as follows:

```
value = funcOne() ?: funcTwo()
```

This sets value equal to the result from funcOne() if funcOne()'s result is a true value, or sets it equal to funcTwo()'s value otherwise.

Operators

In most programming languages binary operators are **infix**, meaning the operator falls between its operands:

```
a + b
```

However, some languages use **prefix** notation (Lisp for instance):

```
(+ a b)
```

Evaluation Order

Evaluation order affects the value an expression evaluates to.

```
a + b * c
```

Can mean

```
(a + b) * c

a + (b * c)
```

Evaluation Order

There is no rule that says we must follow mathematical rules - though languages that do not will likely not be adopted as quickly.

Mathematicians developed the hierarchies of operands long ago and people are used to them. We should mimic them as much as we can.

Operators

Therefore, every language designer must develop operator precedence rules for their language. We can view these in the language documentation:

https://docs.python.org/2/reference/expressions.html#operator-precedence

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

You'll notice they are largely the same.

Java precedence

```
Operators             Precedence
postfix               expr++ expr--
unary                 ++expr --expr +expr -expr ~ !
multiplicative        * / %
additive              + -
shift                 << >> >>>
relational            < > <= >= instanceof
equality              == !=
bitwise AND           &
bitwise exclusive OR  ^
bitwise inclusive OR  |
logical AND           &&
logical OR            ||
ternary               ? :
assignment            = += -= *= /= %= &= ^= |= <<= >>= >>>=
```

Operators

In some programming languages (such as Ruby), operators are implemented as methods. This means that:

```
a + b
```

Is really this:

```
a.add(b)
```

Operators

This also means we can do this:

```
class Fixnum
  def +(other)
    self - other
  end
end

3 + 4
=> -1
```

Operators

Many functional languages define operators as functions as well.

```
(+ a (* b c))
```

Is two function calls.

Operators

Side effects are when a function changes either one of its parameters or a global variable.

If a function does not have side effects, then the order of evaluation of an expression's operands does not matter. (We'll see what this means in a moment)

With side effects, odd things can happen

No side effects

```
def fun(ref pa):
    b = *pa + *pa
    return b

a = 10;
b = a + fun(&a)  // Here fun(a) does not change a
```

If we read `a = 10` and then evaluate `fun(&a) = 20`, b is 30

If we evaluate `fun(&a) = 20` and then read `a = 10`, and b is still 30

With side effects

```
def fun(ref pa):
    *pa = *pa + *pa
    return *pa

a = 10;
b = a + fun(&a)  // Here fun(a) **does** change a
```

If we read `a = 10` and then evaluate `fun(&a) = 20`, b is 30

If we evaluate `fun(&a) = 20`, we will then read `a = 20`, and b is 40

Operators

To address these issues, language designers must decide to either not allow functional side effects, or provide a strict order of operations.

In general, imperative languages don't block functional side effects as it is hard to accomplish and it harms writability.

Operators

Referential Transparency is a property of expressions that exists in a program if any two expressions with the same value in the program may be swapped for one another anywhere in the program, without changing the programs output:

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

If this code is referentially transparent then `result1` and `result2` will be the same.

If `fun()` has side effects, such as changing a, b, or c, `result2` might be different because `fun()` was called just once

Operators

Referential transparency is desirable as it makes programs more readable; this is largely due to the fact that it makes programs behave more like mathematical statements (which we are used to).

Functional languages are referentially transparent, as they lack variables and state.

Operators

Operator Overloading allows programmers to use arithmetic operators for multiple purposes. For instance, instead of using the + operator solely on integer addition, the programmer may be able to use it to add `floats`, `strings`, or even instances of custom types.

Operator overloading is a bit of a mixed-bag; it aids in writability and can aid in readability, but can also harm readability, writability, and reliability. We must be very careful how we use it.

Operators

Consider:

```
class Car:
    def __init__(self):
        self.name = ""
        self.gas_used = 0
    def __add__(self, other):
        return self.gas_used + other.gas_used

a = Student()
a.gas_used = 3.7

b = Student()
b.gas_used = 4.0

print(a + b)
```

This may or may not make sense...

- What does it mean to add two cars together?
- Would you expect adding two cars to return the total gas they had used?
- Is there another way we might want to combine two cars?
- Might some other programmer assume we are adding other fields?
- Can we guess what is happening without looking at the class code?

Operators

This makes more sense:

```
String one = "Hello";
String two = " World.";
System.out.println(one + two);
```

Most programmers would correctly guess (without looking at the String class code) that adding two Strings together is shortcut for concatenation.

Operators

Consider what it can do for writability though. If we have a class that holds a matrix, we could write custom functionality for that class so it could add or multiply matrices. This means that code that looks like this:

```
MatrixAdd(MatrixMult(A, B), MatrixMult(C,D))
```

Can become this:

```
A * B + C * D
```

Operators

Java does not allow operator overloading. Of the languages we have discussed in here, Python, Ruby, C#, and C++ do (although the particular operators each language can overload vary).

In C++ you can overload the following operators:

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	!=	<<	>>	<<=	>>=	==	!=
<=	>=	&&	||	++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

Type Conversions

There are two type conversions that can be performed, a

narrowing conversion which cannot store all of the values of the original type, and a

widening conversion which can store at least approximations of the original type.

Type Conversions

Example of narrowing conversion is converting double to float

Maximum value of each type depends on system. On my local machine, FLT_MAX is about `3e38` and DBL_MAX is about `1e308`

Some values of double simply cannot be approximated by float because they are outside possible range

Type Conversions

```c
double f1 = 1e100;
float f2 = f1;
printf("double: %g\n", f1);
printf("float: %g\n", f2);
```

```
double: 1e+100
float: inf
```

Type Conversions

We must be careful to remember that widening conversions are not always perfectly safe (though they usually are); we can lose a bit of accuracy. For instance:

```
int a = 123456789;
float b = a;
printf("%f\n", b);

...

123456792.000000
```

Coercion

Language designers need to decide if their languages will allow **mixed-mode expressions**. These occur when we allow operators to have operands of different types. For instance:

```
int a = 21;
float b = 21.0;
float c = a + b;
```

Here some coercion must occur (in this case the int will be interpreted as a float).

Errors

Type checking exists to prevent most errors in expressions, but other kinds of errors can occur.

Overflow and **underflow** errors occur when we cannot store the result of an expression in the allocated memory.

In C for instance:

```c
int a = 2938479187407210497;  // overflow
float c = 1e-50;                // underflow
```

Errors

Division by 0 is also an error common in expressions. Though it is not defined mathematically it is often found in expressions.

All of these errors are Run-Time errors. We call run-time errors **Exceptions**.

Relational Expressions

A **Relational Expression** is an expression that compares operands and returns a Boolean value (unless Booleans are not part of the language...).

Coercion is important in regards to relational expressions. For instance, consider the == operator in Javascript:

```
"42" == 42

True
```

This yields True because coercion is allowed; the String on the left-side is interpreted as, or coerced into an int.

The === operator disallows coercion:

```
"42" === 42

False
```

Relational Expressions

Relational operators should have lower precedence levels than arithmetic expressions. Otherwise statements like (from book):

```
a + 1 > 2 * b
```

will not evaluate correctly.