Computer Science 340
Programming Languages
Siena College
Fall 2023

Topic Notes: Expressions and Assignment Statements

We next take a brief look at the very common programming language constructs of assignments and the expressions used as part of those assignments.

Expressions

An *expression* is a sequence of *operands* and *operators* that is evaluated by a programming language.

Arithmetic Expressions

We first look at *arithmetic expressions*, which typically consist of operators, operands, parentheses, and function calls.

Some issues we need to consider:

- precedence rules and order of operations
- associativity rules: normally left to right
- unary operators
- ++ and −− and their *side effects*, *e.g.*, what happens with

  ```
  y = x++ + ++x;
  ```

  and the common beginning programmer error:

  ```
  x = x++;
  ```

- mixed mode arithmetic

Other side effects of concern are from functions:

```
y = x + f(x);
```

vs.

```
y = f(x) + x;
```

What if the function `f` modifies the value of `x`? What appear at first to be two ways to write the same expression can have different results.

If side effects such as this are permitted, some possible compiler optimizations that rely on taking advantage of operator associativity might need to be disallowed. Forbidding such side effects might be part of a language's design.

How many times have you made this mistake?

```
// you have a Java Scanner called s
while (s.hasNextInt()) {
   if (s.nextInt() < 0) {
      // do what you need to do for a negative number
   }
   else if (s.nextInt() == 0) {
      // do what you need to do for 0
   }
   else {
      // do what you need to do for a positive number
   }
```

Since `nextInt` consumes a value from the `Scanner`, you are using different numbers for the two comparison, and you consume 2 numbers from the input when a non-negative is encountered.

Another example:

```
x = (f(a) + b) * (f(a) + c);
temp = f(a);
y = (temp + b) * (temp + c);
```

If `f` has no side effects, we know that `x` must equal `y` after these statements execute. Otherwise, we cannot be sure.

A program for which we can be sure that `x` and `y` obtain the same values in the above can be said to have *referential transparency*.

This is one of the big advantages of pure functional languages. There are no state variables that can be modified by a function call. The value of any function depends only on its parameters.

Overloaded Operators

An operator that is used for more than one purpose is called an *overloaded operator*.

Some are common:

- + being used for addition of various numeric types (and for string catenation in Java)
- + and − for unary and binary operations
- `*` in C/C++ to indicate either multiplication or dereferencing of a pointer (potential danger as the meanings are unrelated)

Some languages (*e.g.*, C++) allow user-defined overloaded operators. For example, see:

Overloading Operators from cplusplus.com at `https://cplusplus.com/doc/tutorial/templates/`

This can be a great aid to readability and writability when used properly.

However, there is nothing stopping someone from overloading an operator in a way that is confusing or even nonsensical.

Type Conversions

Programming languages do automatic *type conversions* (or *coercions* when implicit).

We can categorize these as

- *narrowing* – conversion to a type that cannot represent all of the values of the original. *e.g.*, `float` to `int`
- *widening* – conversion to a type that can represent at least approximations to all of the values of the original. *e.g.*, `int` to `float`

These are often needed in *mixed-mode expressions*. Most languages implicitly coerce using widening conversions.

```
int x, y;
// put something in x and y
double z = x + y;
```

Narrowing conversions are more troublesome, so an explicit conversion, such as a *cast*, may be required.

```
double x, y;
// put something in x and y
int z = (int)(x + y);
```

In many cases, such casts are needed to avoid a compiler warning or error.

Other errors may be detectable only at run time.

- division by zero (NaN a possible result)
- arithmetic overflow/underflow

Some languages have run time systems that catch such errors, others may allow them to fail silently.

See Example:
`overflow` in `https://github.com/SienaCSISProgLang/expressions`

Boolean Expressions

With boolean expressions, we add *relational operators* (comparisons) and *logical operators*.

While languages vary greatly in the symbols used for some relational expressions (`!=`, `/=`, `~=`, `.NE.`, `<>`, `#`), there is a lot of commonality in the operators supported.

A few languages have some unusual variants:

- JavaScript and PHP have special relational operators, === and `!==`, which are the same as the more traditional == and `!=`, except that they do not coerce their operands.
- Ruby uses == for its equality relation operator that uses coercions and `eql?` for those that do not.

Recall that in C, `int` values are used for boolean expressions, with 0 representing `false`, everything else representing `true`.

Consider this very odd example in C:

See Example:
`doublelt` in `https://github.com/SienaCSISProgLang/expressions`

It evaluates the condition that would be evaluated with this `if` statement:

```
if (a < b < c)
```

It might be nice if this was interpreted as `((a < b) && (b < c))` (as many a beginning programmer has tried), but it instead evaluates `a < b`, which becomes a 0 or 1, then compares that result with `c`. Not likely what a programmer intended...

Short Circuit Evaluation

Many programming languages use *short circuit evaluation* to evaluate an expression without evaluating all operations.

It works by observing that an AND evaluates to false if any one of its operands is false, and an OR evaluates to true if any one of its operands is true.

This is done in the C/Java languages and others, but is not in BASIC, Pascal, or Fortran.

This can have the obvious efficiency advantage: we don't need to evaluate terms once the expression's overall value has been determined.

However, it also leads to some programming convenience:

```
int *a = NULL;
//
// ...some code happens...
//
// now, a may or may not be given a value
if (a && (*a < 0))
```

If `a` is still `NULL`, the comparison after the `&&` is not evaluated. If we did evaluate it, a memory error would occur.

Similarly for checking for values in an array:

```
index = 0;
while ((index < length) && (a[index] != val))
  index++;
```

The main problem that can arise with short circuit evaluation has to with side effects:

```
if ((x < 10) || (x++ > 5))
```

If `x < 10` evaluates to true, the rest is not evaluated at all, so the `x++` does not take place.

This can happen more subtlely if there is a function call with side effects as part of the expression that might not be evaluated due to short circuit evaluation.

Assignment Statements

Imperative languages depend on assignment statements for much of their functionality. We all know what they look like.

In C and friends:

```
x = 10;
```

In Pascal and Ada:

```
x := 10;
```

Languages usually use a different symbol for assignment and the equality relational operator (== in C and friends, = in Pascal).

We have also seen that there are additional shortcut operators in many languages: +=, −=, *etc.*.

These are generally harmless, but again side effects combined with shortcut assignment operators can be problematic. We would expect that

```
x += a;
```

and

```
x = x + a;
```

to be equivalent ways to express exactly the same computation.

However, consider

```
x[a++] += a;
```

and

```
x[a++] = x[a++] + a;
```

What happens? Try it:

See Example:
`sideeffects` in `https://github.com/SienaCSISProgLang/expressions`

Some languages allow us to use the result of an assignment statement as an expression whose value we can use.

```
x = y = z = 7;
```

Here, each assignment evaluates to the value assigned, giving the end result of all 3 variables being assigned 7.

This can be helpful or problematic in some cases:

```
if (flag = true)
```

This always evaluates to `true`. In Java, this can only happen with `boolean` variables (and what beginning or not-so-beginning Java programmer hasn't made this mistake?) since the compiler will flag erroneous assignments of other types as an invalid boolean expression for the condition.

However, since C/C++ allow any expression to be interpreted as a boolean expression, we could mistakenly write

```
if (x = 8)
```

and not realize we have both caused `x` to become 8, and forced that condition to evaluate to true every time. Not to mention that we set `x` to 8 when we weren't intending to modify its value.

It can be a useful construct, however:

```
while (c = getc())
```

will continue to assign `c` the values returned by `getc` and will reenter the loop until that value was a `\0` character.

More unusual assignment constructs exist in Perl (and in some cases, other languages):

- conditional targets:

  ```
  ($flag ? $total : $subtotal) = 0
  ```

  which is effectively:

  ```
  if ($flag){
    $total = 0
  } else {
    $subtotal = 0
  }
  ```

- multiple assignments

  ```
  ($first, $second, $third) = (20, 30, 40);
  ```

  which can also be used to accomplish a swap:

  ```
  ($x, $y) = ($y, $x);
  ```