Expressions

Programming Languages

William Killian

Millersville University

Outline

- Purpose
- Associativity, Precedence, and Evaluation Order
- Side Effects
- Categories of Expressions
  - Arithmetic
  - Operators / Function Calls
  - Casts
  - Relational and Boolean
- Assignment

Purpose / Role of Expressions

- Expressions are how we represent **computation**
- The fundamental reason for the creation of computers and programming languages

- Expressions:
  - Describe what actions a computer needs to do
  - *Semantics* defines the order the actions are done
  - *Syntax* defines the supported actions

Purpose / Role of Expressions

- From a **functional** programming view:
  - Everything is an expression!

- From an **imperative** programming view:
  - Assignment expressions are necessary!

Arithmetic Expressions

Case Study: Arithmetic Expressions

These are a programmer's bread and butter

Operators

- Symbols that define mathematical operations (e.g. addition, multiplication)

Operands

- Variables and/or numbers that are acted upon in a computational context

Parentheses

- Symbols to change/force evaluation order

Function calls

- Abstraction of a user- or library-defined operation. Function calls can accept variable arguments.

Arithmetic Expressions

Classes of Operators

Unary Operator

An operator that accepts exactly one argument

`~a` `!cond` `-val` `+cool`

Binary Operator

An operator that accepts exactly two arguments

`a + b` `true || false`

Ternary Operator

An operator that accepts exactly three arguments

`( condition ? value_if_true : value_if_false )`

Arithmetic Expressions

- Evaluation Order
  - The order in which subcomponents of an expression are evaluated

```
cout << (foo (--a, b++) - a) -> c
```

Questions:

1. What should be evaluated first?
2. Function argument evaluation order?
3. Which should come first: `<<` or `->` ?

Arithmetic Expressions

- Operator Precedence
  - Determines the evaluation order of each operator when given a sequence of operands with **operators with different precedence**

  - Typical Precedence in Mathematics
    - Parentheses
    - Unary operators (e.g. +/-)
    - Function Calls
    - Exponentiation and Logarithms
    - Multiplication and Division
    - Addition and Subtraction

Arithmetic Expressions

- Operator Associativity
  - Determines the evaluation order of each operator when given a sequence of operands with **operators with the same precedence**

- Typical Precedence in Programming Languages
  - Left-to-Right: most/all arithmetic operators
  - Right-to-Left: Assignment

Arithmetic Expressions

Operand Evaluation Order

- *Variables*
  - Retrieve value from memory
- *Constants*
  - Retrieve value from memory (usually embedded as part of the instruction)
- *Parenthesized expressions*
  - Evaluate the inner expression first
- *Function Calls?*

Function Calls

Function Calls

- Name
- Parameters / Arguments

- When we encounter a function call, how should it be evaluated?

```
doSomething (funX(), varY, funZ (varY))
```

Operators as Function Calls

Languages can let you define your own operators

OCaml

```
let rec (^^) b = function
  | 0                   -> 1
  | 1                   -> b
  | e when e < 0        -> 0
  | e when e mod 2 = 0 -> (b * b) ^^ (e / 2)
  | e                   -> b * (b ^^ (e - 1))
```

Operators as Function Calls

Languages can let you define your own operators

C++

```
MyInt& operator+= (MyInt& x, MyInt const& y) {
    x.value += y.value;
    return *this;
}

MyInt operator+ (MyInt x, MyInt const& y) {
    return x += y;
}
```

Operators as Function Calls

Overloading

When you have an **existing** operator and want to repurpose it for your own type(s)

Defining

When you want to **create** a new operator that uses a custom sequence of symbols

Turns out, this is exactly how function calls work, too!

Operators as Function Calls: C++

```
// Given the following code
cout << foo (a + 1);

// C++ automatically transforms it into
operator<< (cout,
            foo ( operator+ (a, 1)))
```

Dangers of Operator Overloading

- A comma can be overloaded in C++

```
R operator,(T const& lhs, U const& rhs)
```

- Operators can be rewritten in OCaml

```
let (+) (a:int) (b:int) =
  failwith "no addition for you"
```

- What other strange things have you seen?

Side Effects

Functional Side Effects

- When a function changes a program's state
- Parameter modification or updating a non-local variable

```
int myAdd (int& a, int b) {
   a += b;
   return a;
}

int a = 2;
int c = myAdd (a, 4);
int d = myAdd (a, 4);
// does c == d ?
```

“Preventing” Side Effects (1/2)

Disallow Functional Side Effects

- No references
- No non-local variable access
- **Advantages**:
  - it works
- **Disadvantages**:
  - inflexibility

“Preventing” Side Effects (2/2)

Define the language’s operator evaluation order

- This means all programs have *well-defined behavior*
- Example: `add (++x, x, x--, x)`
- **Advantages**:
  - programmers will expect behavior
- **Disadvantages**:
  - prevents some compiler optimization

Referential Transparency

- Given a program and any two expressions that have the same value
- When one expression is substituted for the other *anywhere in the program*
- Then the behavior of the program is not affected

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

Referential Transparency

Advantage

Semantics are much easier to understand

- All *pure-functional* programming languages are referentially transparent

Casting

Casting

- All values have types
- Some types may be compatible with one-another

  `int ⇔ float ⇔ double`

Casting: Converting from one type to another

Two Possible "Modes"

1. **Implicit**
   - language will automatically perform the conversion
2. **Explicit**
   - the programmer must specify the conversion

Implicit Casting (Coercion)

- Automatic Type Conversion by the Compiler
  - C, C++, Java, Python all support implicit casting
  - OCaml has no implicit casting
- **Type Promotion**
  - compiler expands the precision of a datatype
    - `bool --> char --> short --> int --> long`
    - `float --> double`
- *Can happen during*
  - Expression operands (including assignment!)
  - Function calls (parameters)
  - Function calls (return values)

Implicit Casting Example: C++

```
double add(double a, double b) {
  return a + b;
}

int result = add(1, 2.0);

    // Where are there implicit casts?
```

Explicit Casting

- When the programmer must state in their program that a type conversion to occur

- New type of expression

  Cast expression

```
(NewType) expr // Java, C, C++
NewType (expr) # python, F#
```

Will explicitly cast from *`expr`*'s old type to `NewType`

Explicit Casting Example: C++

`static_cast<To> (From)`

Converts only using implicit/user-defined conversions

`dynamic_cast<To> (From)`

Safely cast up/down/sideways along inheritance structure

`const_cast<To> (From)`

Removes const/volatile modifiers (doesn’t emit instruction)

`reinterpret_cast<To*> (From*)`

Reinterprets underlying bits (doesn’t emit instruction)

Casting

Three Possible Types:

1. **Narrowing**
   - Information loss will happen
   - Same “type” (integral, floating point) but shrinking size
2. **Widening**
   - No information loss
   - Same “type” (integral, floating point) but increasing size
3. **User-defined / Custom**
   - The type will change classes (perhaps even custom)
   - May or may not lose information

User-Defined Casts: C++

- C++ allows programmers to define their own casting function

```
class Foo {
  operator Bar() { // enables implicit
    return ...
  }
  explicit operator int() { // explicit only
    return ...
  }
};
```

Relationals and Booleans

Relational and Booleans

Two Classes of Relational Operators

1. **Equality**

   Used to determine equivalence of values

   Usually some form of == for equality

   Usually some form of != for inequality

   Other operators: <> ~= # /=

2. **Ordering/Comparison**

   Used to sort meaningful values

   Uses symbols like > and < to express

Relational expressions evaluate to a Boolean

Equality

Loose Equality (with Coercion)

```
"1" == 1                   true (JavaScript)
                           false (C/C++, Java, Python)
 2  == 2.0                 true (C/C++, Java)
```

Strict Equality

```
 1  === 1                  true  (JavaScript)
"1" === 1                  false (JavaScript)
[1; 2; 3] = [1; 2; 3]      true (OCaml)
```

Strong(est?) Equality

```
[1; 2; 3] == [1; 2; 3]     false (OCaml)
        a == a             true (OCaml)
```

Ordering and Comparisons

Most Languages require one of two options:

1. Implement all operators (<, >, <=, >=)
   - *This interface returns a Boolean (true/false)*
     - C++ (pre- C++20) and Python take this approach
     - C# can do this or do (2) with `IComparable<E>`
2. Implement one operator/interface (<=>)
   - *This interface usually returns one of three possible categories of values (less, equal, greater)*
     - **`Java`** `– Comparable<E> via compareTo()`
     - **`JavaScript`** `– define a lambda function`
     - **`OCaml`** `– lambda or overload`
     - **`C++20`** `– define` **`operator<=>`**

Expression Evaluation

Short Circuit Evaluation

- *When we can determine the value of an expression without evaluating all parts*

```
0 * ...
true || ...
false && ...
```

- Logic expressions are short-circuit evaluated in most languages
- Arithmetic expressions often are *not*

Assignment

Assignment

name <assign_op> *Expr*

Assignment operator can vary

- `=` in most languages
- `=:` in ADA
- `<-` for reference assignment in OCaml / F#

Conditional Assignment

Perl

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

C++

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

C

```
*(flag ? &total : &subtotal) = 0
```

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

Compound Assignment

Assignment expressions often take the form:

a = a *op* b

Some languages support a shorthand syntax:

a *op*= b

Can be overloaded in C++, Python

Unary Assignment Statements

Defined by unary symbols of `++` and `--`

- `++var` or `--var`
  - Returns the new value
- `var++` or `var--`
  - Returns the old value

```
int x = 4;
int y = ++x;
int z = x++;
// y == z?
```

```
int a = 2;
++a++;
// valid?
```

Multiple Assignment

In Some Languages

```
# Perl
($first, $second) = ($second, $first)

# Ruby, Python
first, second = second, first

# JavaScript
[first, second] = [second, first]

# OCaml
let (first, second) = (second, first) in ...
```

Multiple Assignment

In (Some) Compiled Languages

```
// Swift
(first, second) = (second, first)

// C++17 -- cheating with std::tuple functions
std::tie(first, second) = std::tuple(second, first)

// C++17 -- declaration + assignment
auto [first, second] = std::tuple(1, 2);
```