Arithmetic Expressions

- Typically consists of operators, operands, parentheses, and possibly function calls.
- Operators may be unary, binary, or even ternary.
- Operators may be infix, prefix, or postfix.
- Issues:
  - precedence
  - associativity
  - order of operand evaluation
  - side effects from operand evaluation
  - user-defined operator overloading
  - type mixing

Precedence

- Define different precedence levels for operators
- Unary operators are often at the top, though parentheses may be needed. e.g. `a + (-b) * c`
- Parentheses can always be used to override order

Associativity

- Associativity defines the order of operations when two adjacent occurrence of operators at the same level of precedence
- Usually it is either left-to-right (left associative) or right-to-left (right associative).
- Most operators are left associative, but = and exponentiation are often right associative
- Note that changing associativity may affect results even for mathematical operations that are normally associative (overflow, precision, etc.)

Conditional Expressions

- In C-based languages there is a conditional ternary expression:
  `expression1 ? expression2 : expression 3`
- This means if `expression1` is true, the value of the expression is that of `expression2`. Otherwise, the value is `expression3`.
- This is also available in Perl, JavaScript, and Ruby.

Operand Evaluation Order

- In an arithmetic expression, there are operands.
- These operands may in turn require evaluation. e.g.
  `f(1) + f(2) + f(3)`
- Does it matter what order the operands are evaluated?

Operand Evaluation Order

- If evaluating the operands does not have any side effects, the evaluation order of the operands is irrelevant
- If there are side effects (e.g. changing global variables, printing to screen, static variables, etc.), the order can be important
- Having side effects make it more difficult for compiler to optimize code

Side Effects

- Changing global variables
- Changing static variables
- Interacting with shared resources (files, screen, etc.)
- Functions defined in the mathematical sense do not have side effects
- Pure functional languages (e.g. Lisp, Scheme, Racket) do not have variables and functions cannot have side effects (not exactly)

Referential Transparency

- A program has the property of referential transparency if any two expressions in the program with the same value can be substituted for one another anywhere in the program, without affecting the action of the program.
- i.e. a referential transparent function depends only on the value of its parameters, not the order in which they are evaluated
- Semantics is much easier to understand
- Functions in pure functional languages are referentially transparent

Overloaded Operators

- A single operator may have many different meaning depends on context (e.g. operand type)
- Many languages have bulit-in operator overloading (e.g. + can be used for both integer and floating-point additions)
- Some languages allow for user-defined operator overloading
- It can be used to aid readability (e.g. addition for fraction objects)
- But there is no way to prevent misuse (e.g. define + to do multiplication)
- Precedence of operators usually cannot be changed

Type Conversions

- Narrowing conversion: converts a value to a type that cannot store even approximations of all of the values of the original type
- Widening conversion: converts a value to a type that can include approximations of all of the values of the original type (e.g. int to float)
- Widening conversions are almost always "safe", but can result in reduced accuracy.
- Coercion: implicit type conversion, common when operands do not have the required types (e.g. float + int)
- Many languages also have explicit type conversion (e.g. casting in C++)

Mixed-mode Expressions

- Some languages allow mixed-mode expressions, some do not
- Rules must be defined for implicit operand conversions
- Language designers need to balance flexibility vs. type checking

Errors in Expressions

- Some possible errors in evaluation of expressions are: overflow, underflow, divide by zero.
- Some language detect these as exceptions. Some may not.

Relational Expressions

- A relational operator compares the values of operands. Usually there are equality tests, and greater/less comparisons for certain types
- Relational operators should have lower precedence than arithmetic operators. What about assocativity? Does it matter?

Boolean Expressions

- Common to have AND, OR, NOT, possibly XOR
- Precedence among AND and OR?
- Perl and Ruby provides two sets of AND and OR with different precedence (e.g. `&&` vs. `and`)

Short-circuit Expressions

- Can be used to shorten code if used wisely
- Possibly more difficult to read
- Must also be careful about side effects: if the second part has side effects it may or may not execute
- Some languages such as Ada allows user to choose (`and then` and `or else`)

Assignment Statements

- Most use `=`, some use `:=`
- Some languages allow conditional target (l-value):
  `($flag ? $count1 : $count2) = 0` in Perl
- Many languages allow compound assignment operators: `a += b`
- Some have unary compound operators (e.g. `++` and `--`). Prefix vs. Postfix, side effects, what if there are multiple occurrences?
- In some languages assignment statements are expressions (have values). Can be used inside loop conditions or to chain assignments (right associative).

Assignment Statements

- Some languages allow multiple assignments: e.g. `(a,b,c) = (1,2,3)`
- Swap could be done without using any temporaries: e.g. `(a,b) = (b,a)`
- Is it simultaneous or sequential (conceptually)?

Functional Languages

- In functional languages, assignments are simply to names (not variables).
- e.g. `let` in Lisp-style languages