Expressions and Assignment

- Precedence, associativity, evaluation order, side effects
- Overloading
- Type conversions
- Relational and Boolean expressions
- Assignment statements

Expressions and Assignment Statements

- Common in all imperative languages
  - often included in functional languages (SET!) and declarative languages (is)
- <target location> <assign operator> <source expression>
  - “l value” and “r value”
- Language design issues
  - operators and precedence
  - associativity
  - order of evaluation
  - evaluating mixed mode expressions
  - short-circuit evaluation of Boolean expressions

Operators and Precedence

- First issue is operators to provide (and symbols)
  - +, -, *, / pretty much universal
  - div, mod, ** fairly common
    - Pascal has no **
  - <, >, =, /=, etc., and, or, not pretty much universal
  - special operators appropriate for language purpose
    - APL array manipulation (transpose, invert, etc.)
    - SNOBOL pattern matching operations
    - ++, +=, etc. in C

Operators and Precedence (continued)

- Then determine precedence among operators
  - determines which operations are done first
  - typical is
    - exponentiation
    - multiplicative
    - additive
    - relational
    - logical
- APL has all operators at same level
- Parentheses can always be used to override operator precedence

Associativity

- Determines order in which consecutive operators of equal precedence are evaluated
  - left to right (left associativity)
    - A + B + C means (A + B) + C
  - right to left (right associativity)
    - A + B + C means A + (B + C)
- Most operators in most languages are left associative
  - Fortran **; C ++, --, unary +- are right associative
  - ALL operators in APL are right associative
  - Ada ** is non-associative
    - programmer must explicitly use parentheses to determine order of operations

Side Effects and Evaluation Order

- Operand evaluation order
  - variables
    - just fetch the value
  - constants
    - sometimes a fetch from memory
    - sometimes the constant is in the machine language instruction
  - parenthesized expressions
    - evaluate all operands and operators first
  - function references
    - the case of most interest!
    - order of evaluation is crucial

Side Effects and Evaluation Order (cont.)

- A side effect is when a function or operator does more than just return a va
  - change a parameter or global variable
    - usually considered a bad programming practice
      - Ada disallows changes to function parameters
- If side effects are possible, then the order in which an operator's operands are evaluated can have an impact on the result
  - in C
    - x = 3; z = x * ++x; /* 12 or 16?*/
    - x = 3; a[x] = x++; /* assign 3 to a[3] or a[4]? */
- evaluation order by itself can have impact
  - for expression A + B + C order of additions shouldn't matter
  - but if A and C have very large positive values, and B has very large negative value, but order is A + C + B, error

Side Effects and Evaluation Order (cont.)

- One solution: write the language definition to disallow functional side effects
  - no two-way parameters in functions
    - Ada
  - no nonlocal references in functions
    - FORTRAN
  - advantage
    - it works!
  - disadvantage
    - programmers want the flexibility of two-way parameters and nonlocal references
- Another solution: write the language definition to demand that operand evaluation order be fixed
  - disadvantage
    - limits some compiler optimizations

Operator Overloading

- Overloading means using the same name to represent two or more different things
  - + means both integer and real addition in most languages
  - also applies to more than operators (functions, procedures)
- Most modern languages (Ada, C++, Fortran 90) allow user-defined operator overloading
  - Ada
    - function “+” (left, right: MYTYPE) returns MYTYPE is
  - then can use + as an infix operator for things of appropriate type
    - type MYTYPE is ...;
    - A, B, C: MYTYPE;
    - C := A + B;
  - compiler determines which + to use based on types of operands

Problems With Operator Overloading

- Loss of compiler error detection
  - omission of an operand should be a detectable error
    - * in C and C++
  - avoid by introduction of new symbols
    - Pascal's div
- C++ and Ada allow user-defined overloaded operators
  - users can define nonsense operations
  - readability may suffer

Type Conversions

- Often want to write expressions that are mixed mode (contain operands of more than one type)
  - real + integer
- This implies a need to convert one type to another so the expression can be evaluated
  - coercion is implicit conversion, done automatically by the compiler
    - implies semantics that define rules for determining type to convert to from operands
    - problem is loss of error detection
  - casts are explicit conversions specified by the programmer
    - can lead to very clumsy expressions if doing a lot of mixed mode expressions
    - Ada: float (Index);
    - C: (int) speed;

Type Conversions (continued)

- Whether implicit or explicit, conversions can be
  - widening
    - convert to a type with a greater representation range
      - although perhaps with a loss of precision
    - integer to real
  - narrowing
    - convert to a type with a more restricted range
    - double precision to real
  - promoting
    - convert to a type with additional semantic information
    - integer to character (Pascal chr function)
  - demoting
    - strip away semantic information
    - character to integer (Ada val attribute)

Type Conversions (continued)

- Most languages provide some coercions and allow casts
- PL/I allows coercion between almost any types
  - DCL A, B, C INT;
  - if (A <= B <= C) then ...
    - A <= B yields a Boolean value, which is a single bit 0 or 1
    - convert bit string to integer to compare to C
    - bit <= C is true for any positive values of C
- Ada allows no coercion
  - all conversions must be casts
  - conversions are allowed between all numeric types
  - other conversions only allowed between derived types that share an ancestor
    - type foo is new Boolean; type bar is new Boolean;
    - A: foo; B: bar;
    - A := foo(b);

Relational and Boolean Expressions

- Relational expression consists of 2 operands (which may themselves be expressions) and a relational operator
  - <, <=, >, >=, /=, in
  - result is a Boolean value, which is usually implemented as a 0 or non-zero integer value
- Boolean expression consists of Boolean variables, relational expressions, and Boolean operators
  - and, or, not, xor
  - result is a Boolean value
- Design issue for languages is whether to implement lazy or strict evaluation of Boolean expressions
  - strict evaluation evaluates both operands, then gets result
  - lazy (or short-circuit) evaluation stops evaluating as soon as truth or falsehood of expression can be determined

Relational and Boolean Expressions (continued)

- Suppose A is a 100-element array, then
  - while (scan < 101) and (A[scan] <> key)
  - works with short-circuit evaluation
  - runtime error with strict evaluation
- Most languages perform strict evaluation
- C, C++ and Java perform short-circuit evaluation
- Ada provides operators to allow programmer to control which occurs
  - and, or are strict evaluation
  - and then, or else are lazy evaluation
- C oddity
  - A < B < C is legal
  - so is A == B == C

Assignment

- Fairly standard in all imperative languages
  - major design choice is whether assignment functions as a statement or a function
  - operator used to indicate assignment varies
    - disambiguate assign and equality
    - PL/I A = B = C assigns to A the result of B = C as comparison
    - what is A = B = C in C
- As a statement, usually require a reference (location) and a value
  - usually single target for assignment
  - PL/I allows multiple A, B = 200;
- As a function, returns a value
  - APL, C, C++

Assignment (continued)

- C and C++ provide a rich diversity of assignment operators and choices
- Conditional targets in C++ and Java
  - (X > Y) ? A : B = 24.3;
  - A gets value if X > Y, else B does
- Conditional sources
  - AVG = (COUNT == 0) ? 0 : SUM / COUNT;
  - if COUNT is 0, AVG gets 0, else AVG gets SUM / COUNT
- Compound assignment operators
  - total += value; /* total = total + value */
  - also in Algol 68

Assignment (continued)

- Unary assignment operators
  - count++; /* count = count + 1 */
  - sum = ++count; /* count = count + 1; sum = count */
  - right associative
    - - count ++ /* - (count ++) */
- Assignment as operator
  - while ((ch = getchar()) != EOF) {
  - read input, assign to ch; while compares to EOF
  - this is another example of an expression side effect
  - means less error checking at compile time
    - suppose meant while ((ch == getchar())

Mixed-mode Assignment

- FORTRAN, C, C++
  - Any numeric value can be assigned to any numeric scalar variable, all necessary coercions are done
- Pascal
  - Integers assigned to reals, not other way
- Java
  - Only widening conversions allowed
- Ada
  - No coercion