

Express Yourself

- Expressions: combining named entities
 - Expression notation
 - Operators and operands
 - Functions
 - Precedence and associativity
- Assignment and variables
 - Functional vs. side-effect
 - Value vs. reference
 - Initialization
 - Assignment operators
- Evaluating expressions
 - Evaluation order
 - Short-circuiting

Expression Notation

- Constants, literals, or variables
- Operators, functions
 - One and the same in some languages: Ada, C++
- Notation refers to where the function or operator is placed relative to its arguments or operands
 - Prefix
 - Function calls in many languages
 - `function_name(arg1, arg2, arg3)`
 - `"+(a, b)` (Ada)
 - `!"cancel".equals(inputStr)` (Java)
 - `(* (+ 2 5) 10)` (Lisp)
 - `twice multiply 3` (ML)
 - Infix
 - Binary operators in many languages — `a + b`, `x ** y`, `p << 2`, `flags & SETTING_BIT`
 - `fileDialog openOnDirectory: dir withFileTypes: types` (Smalltalk, Objective-C)
 - `x = if j > 0 then x + 1 else x - 1` (Algol family)
 - `x = (j > 0) ? x + 1 : x - 1` (C and relatives)
 - Postfix
 - `pointer^` (Pascal)
 - `z++` (C and relatives)
 - `1 inch 1 inch moveto` (PostScript)

Precedence

- Infix notation can lead to ambiguity on what should be done first
 - In Fortran, is $a + b * c ** d ** e / f \dots$
 - $((((a + b) * c) ** d) ** e) / f ?$ or
 - $a + (((b * c) ** d) ** (e / f)) ?$ or
 - $a + ((b * (c ** (d ** e))) / f) ?$
 - In Smalltalk or Objective-C, is `shape containing: aShape intersects: anotherShape...`
 - `[shape containing: aShape] intersects: anotherShape ?` or
 - `shape containing: [aShape intersects: anotherShape] ?` or
 - `shape containing: aShape intersects: anotherShape ?`
- Typical conventions:
 - Multiplication/division precede addition/subtraction
 - Boolean operators group after arithmetic operators
 - Unary operators precede binary operators
 - Some details in Scott's Figure 6.1
- Precedence can be implied by the grammar and thus the parser, though this is not absolutely necessary

Associativity

- Mostly left-to-right: $2 + 3 - 4 - 5$ is...
 - $((2 + 3) - 4) - 5 = -4$, not
 - $2 + (3 - (4 - 5)) = 6$
- Exponentiation ($**$)
 - Fortran: right-associative — $4**3**2 = 4**(3**2)$
 - Ada: *not* associative — parentheses required
- Assignment
 - C, which allows concatenation of assignments, $a = b = a + c$, is right-associative
 - Calculate $a + c$ first, assign to b , then assign to a
 - Speaking of assignment...

Assignment

- Functional vs. imperative languages
 - Functional minimizes the need for state: computation is done by evaluation of an expression at a given time; recursion is frequently used
 - Imperative languages iterate more, and so require *side effects* — changes to values that persist across the program
 - Side effects introduce a distinction between *functions* — constructs that return a value — and *statements* — constructs whose value lie in their side effects
 - Imperative programming as “computing by means of side effects”
 - That is where variables, and assignments to variables, come in
- Anatomy of an assignment
 - *lvalue* *<assignment_op>* *rvalue*
 - An *l-value* is any expression to which an expression can be assigned
 - The *assignment_op* is the token for assignment (typically “=” or “:=”)
 - If you’re a stickler for these things, an assignment $a = b$ should be read “ a gets b ,” not “ a equals b ”
 - An *r-value* is any expression that can be assigned to an *l-value*

L-value and R-value Examples

$x = y + z$; (most languages where “=” is the assignment token)
 $x := x + 2$; (most languages where “:=” is the assignment token)
`a[5] = "Item".substring(2);` (Java)
`buf[i][calculateColumn(id)] = buf[i - 1][priorColumn] << 4;` (C et. al.)
`val currentTuple = (5, 7);` (ML)
`val (x, y, z) = (10, 9, "center");` (ML)
`a := Array fromCollection: c.` (Smalltalk...note the period)
`my ($arg1, $arg2) = @_;` (Perl)
`f[4]->id.tail = g[2]->postfix;` (C et. al.)
`@list = qw/Tom Dick Harry/;` (Perl)
`byte b[] = { 0, 5, (byte)0x23, (byte)0xaa };` (Java)

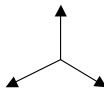
Variables: Values vs. References

- Note Scott's Figure 6.2
- “Copy” vs. “point”
- Distinction is often implicit
 - Java: based on primitive type vs. class
 - Smalltalk: it's *always* by reference
 - ML: distinction is required

```
val x = 5;
val xref = ref x;
val x = x + 2;    (* x is now 7; !xref is still 5 *)
xref := x + 3;   (* note no declaration; !xref is now 10 *)
```
 - C and C++ can derive references from values when applicable

```
int x = 5;
int *xref = &x;
x = x + 2;    /* x is now 7; so is *xref */
xref = x + 3; /* Watch out!!! In most compilers this is a warning. */
```

Orthogonality

- Introduced as a design goal in Algol 68
 - Orthogonal features...
 - can be used in any combination
 - are meaningful in all combinations of those features
 - have consistent meaning regardless of how they are combined
 - Directly related to geometric orthogonality
- 
- Examples
 - Allowing all “statements” to be used as expressions
 - *if..then..else*
 - nested statement blocks
 - giving assignments a value (usually the r-value)

Initialization

- Combining declaration and assignment

```
int x = 5;
val xref = ref (x + y);
void *buf = malloc(1024);
```
- Aggregates: a language's ability to specify literals for more complicated types (arrays, structures)

```
String[] names = { "Tom", "Dick", "Harry" };

my @commands = qw/Open Close Save Quit/;

struct SubRec { double x; double y; }
struct Rec { int a; SubRec sr; int b; }
Rec r = { 5, { 5.0, -1.5}, 10 };
```
- Constructors: parameterized and encapsulated initialization for aggregate types (including object-oriented classes)

```
MyClass c = new MyClass(5, 7, false);
```
- Implicit initialization?
 - C always sets values to zero; Java specifies detection of assignment
 - Explicitly undefined values: *null*, *NIL*, *NaN* (not-a-number)

Combination Assignments

- Simultaneous “operate-and-assign”

```
x += 5;
i /= calcDivisor() + 3;
j++;
s += " thought".substring(3);
```
- Not just syntactic sugar!
 - Avoids repeated (and possibly incorrect) function calls and/or memory accesses and dereferencing

```
tokens[getIndex()] = tokens[getIndex()] + " private";
r.a[i].value = r.a[i].value * alpha;
```
- C has pre- and post- combos, ++ and --
 - Assign the new value either before or after the expression is evaluated
 - `a[++index]` vs. `a[index++]`
- ML, Perl, and Clu allow tuple-like assignments

```
my ($x, $y) = (5, 3); ($x, $y) = ($y, $x);
```

Evaluation Order

- Not the same as determining precedence or associativity:

```
my $j = 0;
sub getIndex { $j++; return $j; }
my $result = $j - getIndex() + $j;           # $result gets 1.
```

- Watch out for side effects
 - Danger zone: rearranging floating point arithmetic
- May affect code improvement
 - Register allocation
 - Instruction scheduling
 - Memory accesses
- Typically, evaluation order is undefined
 - Exception: Java — evaluation is left-to-right
 - What if the above fragment were written in Java?

Short-Circuit Booleans

- Sometimes we know the result of a boolean expression before we evaluate the entire expression; short-circuiting bails out once we know the result

```
if ((x != 0) && (y == 5) || ("".equals(response))) ...
```

- May improve performance and avoid otherwise erroneous states

```
if (unlikely_condition && expensive_function()) ...
while ((p != null) && (p.getCount() > 0)) ...
for (Iterator it = aList.iterator(); it.hasNext() && !"".equals(it.next()); ) ...
```

- Watch out for when short-circuiting results in incorrect computation: Scott, Figure 6.3
 - Clu, Ada, C, C++, and Java allow explicit short-circuiting (or not):
 - Clu: *and/or* vs. *cand/cor*
 - Ada: *and/or* vs. *and then/or else*
 - C, C++, Java: *&& / ||* vs. *& / |* (on booleans in Java)