Express Yourself

- Expressions allow programming languages to combine simple objects (e.g., literals, variables, etc.) into more complex ones — in fact, the need for well-formed expressions (i.e., matched parentheses) is one factor that takes us from regular to context-free grammars
- Simple values or objects (e.g., literals, variables) make up the simplest expressions
- Such expressions may then serve as *operands* to an *operator* or function, which would itself be an expression, which could then be another operand, etc.

Expression Notation

- Start with tokens for constants, literals, or variables
- Add tokens for operators and functions
  - In some languages, operator == function (*Ada*, *C++*)
- Three ways to write operators
  - *Prefix*: operator first, then operands
  - *Infix*: operator between operands
  - *Postfix*: operands first, then operator last
- Typical pattern: infix for binary operators, prefix for most unary operators, otherwise postfix

Expression Notation Examples

- Prefix
  - Function calls: `function_name(arg1, arg2, arg3, ...)`
  - When operators and functions are identical, operators can look like functions: `"+"(a, b)` (*Ada*)
  - `!("cancel".equals(inputStr))` (*Java*)
  - `(* (+ 2 5) 10)` (*Lisp*, with the operator *inside* the parentheses — this is *Cambridge Polish* notation)
  - `twice multiply 3` (*ML*)

- Infix — mostly binary operators in many languages: `a + b`, `x ** y`, `p << 2`, `flags & SETTINGS_BIT`
  - *Functions* can be infix too: `fileDialog openOnDirectory: dir withFileTypes: types` (*Smalltalk*, *Objective-C*)
  - Conditional expressions are also a form of infix:
    - `x := if  j > 0 then x + 1 else x - 1;` (*Algol*)
    - `x = (j > 0) ? x + 1 : x - 1;` (*C* and its relatives)
  - Infix with > 2 operands is sometimes called "mixfix"

- Postfix: used for selected operators
  - `pointer^` (*Pascal* pointer dereference)
  - `i++` (post-increment and -decrement in *C* et. al.)
  - `1 inch 1 inch moveto` (functions in *PostScript*)

Precedence

Infix notation can have ambiguous operator order:

- Take the *ForTran* expression `a + b * c ** d ** e / f`
  = `((((a + b) * c) ** d) ** e) / f` ?
  = `a + (((b * c) ** d) ** (e / f))` ?
  = `a + ((b * (c ** (d ** e))) / f)` ? (and many more…)
- Also happens with mixfix functions: `shape1 containing: aShape intersects: shape2` (*Smalltalk*, *Objective-C*)
  = `[shape1 containing: aShape] intersects: shape2` ?
  = `shape1 containing: [aShape intersects: shape2]` ?
  = `shape1 containing: aShape intersects: shape2` ?

- To disambiguate operation order, many languages define *precedence rules*; typical conventions include:
  - Multiply/divide before you add/subtract
  - Perform arithmetic before boolean operators
  - Perform unary before binary (e.g., `-a + b`)
  - See Scott Figure 6.1 for detailed rules
- When in doubt, look at the language specification, or just use parentheses
- Some languages ditch the whole idea of precedence rules and simply *require* parentheses all the time (*APL*, *Smalltalk*) — would you say this is a cop-out or an elegant simplification?

Associativity

- *Associativity* refers to operators of equal precedence (e.g., addition/subtraction, multiplication/division)
  - `2 + 3 - 4 - 5`

    = `((2 + 3) - 4) - 5 = -4` if left-associative

    = `2 + (3 - (4 - 5)) = 6` if right-associative
  - In general, left associativity is used — is it because many languages read from left-to-right? Or because the computer reads character streams for left to right? Or both?

- Even where left-associativity is the norm, there are some exceptions
  - In deference to mathematical convention, exponentiation is typically right associative
    - *ForTran:* `4 ** 3 ** 2 = 4 ** (3 ** 2) = 262144`
    - *Ada* chooses not to deal with it: it requires parentheses for exponents
  - Some languages allow assignments inside expressions — right associativity applies here as well
    - *C:* `a = b = a + c` calculates `a + c` first, then assigns that to `b`, and finally assigns that to `a`
- Speaking of assignments…

Assignment

- Functional vs. imperative languages
  - Functional languages minimize the need for state: computation is done by evaluation of an expression at a given time; recursion is used frequently
  - 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 lies in their side effects
- Imperative programming can be thought of as "computing by means of side effects" — that's where variables, and assignments to variables, come in

Anatomy of an Assignment

l-value <*assignment_op*> *r-value*

- An *l-value* is any expression to which an expression can be assigned — note, this isn't always just a variable!
- The *assignment_op* token represents the string used to denote assignment — typically "=" or ":="
  - If you're a stickler for these things, the assignment *a* = *b* should be read "*a* gets *b*," not "*a* equals *b*"
- An *r-value* is any expression whose value can be assigned to an *l-value*

L-value and R-value Examples

- `x = y + z;` (in most languages where "=" is the assignment token)
- `x := x + 2;` (in most languages where ":=" is the assignment token)
- `a[5] = "Item".substring(2);` (*Java*)
- `buf[i][calculateColumn(id)] = buf[i – 1][priorColumn] << 4;` (*C* and its syntactic relatives)
- `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*)
- `var f = function(x, y) { return x + y; };` (*JavaScript*)

- While *ML* has an assignment-like construct, note that "=" truly means "equality" — "this symbol *is* this value"

```
val x = 5;

fun f y = x + y;

f 2; (* val it = 7 : int *)

val x = 10; (* Note how val is required even though x has
been "defined" previously. *)

f 2; (* val it = 7 : int! *)
```

- Other *ML* examples:
  - `val currentTuple = (5, 7);` (*ML*)
  - `val (x, y, z) = (10, 9, "center");` (*ML*)
- For exact assignment semantics, *ML* defines the distinct ":=" operator, used for placing values in references

Variables: Values vs. References

- Note Scott's Figure 6.2: "copy" vs. "point"
- Language approaches vary
  - *Java*: Built-in types (primitives) = value, user-defined types (classes) = reference
  - *C#, Eiffel*: User-defined types may be either value or reference; for example, in *C#*, *classes* use the reference model while *structs* use the value model
  - *Smalltalk*, *Clu*: Always by reference (no such thing as "primitive" types here!)
  - *ML*: Always explicit

```
val x = 5;
val xref = ref x;
val x = x + 2; (* x is now 7; !xref is still 5 *)
xref := x + 3; (* !xref is now 10; note no declaration *)
```

  - *C* and *C++* can derive references from values when applicable, and vice-versa ("&" and "*")

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

  - *Perl*: Similar in feel to *C* and *C++* via "\" and "{ }"

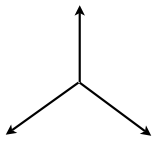
```
my @x = (7, 'y', 28);
my $xref = \@x;
$x[3] = 'n';    # print @{$xref} shows "7y28n"
```

Boxing and Unboxing

- In languages that differentiate "primitive" from "object" types (e.g., *Java*, *C#*), there is sometimes a need to use a primitive type in a context that expects an object — causing an issue with value vs. reference models
  - In Java, the primitive *int* has a corresponding *Integer* class, so that integers may be used, say, in a *Map* data structure (*Map* instances require objects for both keys and values)
- Explicit conversion used to be required in Java — e.g., *new Integer(28)* or *int i = integerObject.intValue()*
- These days, the conversion is automatic, and is called *boxing* and *unboxing*

Orthogonality

- Introduced as a design goal in Algol 68
- Features are *orthogonal* if…
  - They can be used in any combination
  - They are meaningful in all combinations of those features
  - They have consistent meaning regardless of how they are combined



- Related to geometric idea of orthogonality
- Example in expressions include the ability to use "statements" as expressions: *if…then…else,* nested statement blocks, having assignments "return" a value

Combination Assignments

- Simultaneous "operate-then-assign" constructs
  - `x += 5;`
  - `i /= calcDivisor() + 3;`
  - `j++;`
  - `s += " thought".substring(3);`
- Not just syntactic sugar — avoids repeated (and possibly incorrect) function calls and/or dereferencing
  - `tokens[getIndex()] = tokens[getIndex()] + " private";`
  - `r.a[i].value = r.a[i].value * alpha;` (possible side effects)
- C et. al. have prefix and postfix combinations: performs operation before or after evaluation, respectively

Multiway Assignments

- *ML*, *Perl*, *Clu*, *Python*, *Ruby* (to name a few) allow tuple-like assignments:
  - `val (x, y) = (5, 3); (* ML: val x = 5; val y = 3; *)`
  - `a, b := c, d; % Clu: a := c; b := d;`
- Not just a shortcut, since you can do this:
  - `($x, $y) = ($y, $x); # Perl: $x = $y; ......?`
- Further, when available, multiway assignments allow functions to return tuples:
  - `x, y := getCoordinates(mouseX, mouseY, scale); % Clu`

Initialization

- Frequently, it is very natural to combine declaration and assignment
  - `int x = 5; // C, C++, Java, JavaScript`
  - `val xref = ref (x + y); (* ML *)`
  - `void *buf = malloc(1024); // C`
- Beyond simple types, many languages allow *aggregates* for initializing arrays, maps, user-defined types, etc.
  - `String[] names = { "Tom", "Dick", "Harry" }; // Java`
  - `my @commands = qw/Open Close Save Quit/; # Perl`
  - `var p = { name: "Ed", age: 21, single: true }; // JavaScript`

- In the absence of explicit initialization, many languages provide for *default values*, frequently whatever is represented by zero-filled bits for the variable's type
  - In *C*'s type system, this is frequently just 0
  - Other languages have explicit literals for uninitialized values: *null*, *NIL*, *NaN* (not-a-number)
- Object-oriented languages provide *constructors*, for parameterized and encapsulated initialization; beyond assigning default values automatically, constructors can also invoke code for fancier initialization logic
- In many languages, uninitialized variables may lead to dynamic semantic errors; *Java* and *C#* move this to the realm of static semantic errors by mandating *definite assignment* — detection of whether all possible paths to an expression assign values to all of its variables

Evaluation Order

- A "loophole" of sorts that isn't taken care of by precedence and associativity rules: in what order are the *operands* of an expression evaluated?
  - ```
    # In Perl...
    my $j = 0;
    sub getIndex { ++$j; }
    my $result = $j - getIndex() + $j;  # $result gets 1!
    ```
- Two key issues: *side effects* and *code improvement*
  - Evaluation of operands (particularly functions) may affect the values of other operands
  - It is frequently desirable to rearrange evaluation order for speed
- With few exceptions, evaluation order is left undefined

Short-Circuit Evaluation

- We sometimes know the result of a boolean expression without having to evaluate the whole thing: *short-circuit evaluation* takes advantage of this:
  - `if ((x != 0) && ((y == 5) || ("".equals(response))))...`

Why bother with this when x == 0?

- Short-circuiting may improve performance and avoid erroneous states…except for when the short-circuited expression might have side effects
  - For this reason, some languages provide both short-circuiting and non-short-circuiting boolean operators (`and` / `or` vs. `cand` / `cor` in *Clu*, `and` / `or` vs. `then` / `or else` in *Ada*, && / || vs. & / | in *C*, *C++*, *Java*, and *JavaScript*)
    ```