Expressions and Assignment

COS 301: Programming Languages

Outline

Introduction

- Expressions: fundamental means of specifying computations
  - Imperative languages: usually RHS of assignment statements
  - Functional languages: just the function evaluation
- Need to understand order of operator, operand evaluation
  - Maybe only partially specified by associativity, precedence
  - If not completely specified → maybe different results in different implementations

Introduction

- Other issues: type mismatches, coercion, short-circuit evaluation
- For imperative languages: dominant role of assignment to memory cells

Outline

- Introduction
- Arithmetic expressions
- Infix/prefix/postfix
- Overloaded operators
- Type conversion
- Relational & Boolean expressions
- Short-circuit evaluation
- Assignment statements
- Other assignment mechanisms

Arithmetic expressions

- Arithmetic expression evaluation — primary motivation for first programming languages
- Parts:
  - operators
  - operands
  - parentheses (grouping)
  - function calls

Design issues for arithmetic

- Operator precedence
- Associativity
- Operand evaluation order
- Operand evaluation side effects?
- Overloading?
- Type mixing

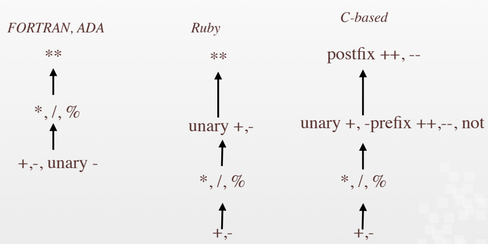
Operator *arity*

- Unary operators
- Binary operators
- Ternary operators
- n-ary operators

Operator precedence rules

- Define order in which adjacent operators are evaluated
- Typical *precedence levels:*
  - parentheses
  - unary operators
  - ** (if present)
  - *, /
  - +, -
  - relational operators

Operator precedence



Associativity

- *Operator associativity rules* → order adjacent operators at same precedence level are eval'd
- Typical:
  - Left to right, except for **
  - Unary ops: sometimes right→left (e.g., FORTRAN)
- APL: all operators have same precedence, all associate right→left
- Smalltalk: binary methods ("operators") have same precedence, left associativity
- Parentheses: can override precedence, associativity

Operator associativity

- Left-assoc relational ops: a < b < c okay…
  - but in C ⟹ if (a<b) then (1<c) else (0<c)
  - not (a<b) && (b<c)
- Non-assoc ops: things like a < b < c are illegal

Lang	+,-,*,/	Unary -	**	!=, ==,<,…
C-like	L	R	n/a	L
Ada	L	non-assoc	non-assoc	non-assoc
FORTRAN	L	R	R	L
VB	L	R	L	non-assoc

Operator associativity

- Optimizing compilers may reorder associative +,*
- E.g.: x * y * z * w can be evaluated in any order
- If x, z very large, y, w very small — makes a difference!
  - Floating point: lose precision, produce infinities
  - Integers: overflow, wraparound
- Can always override with parentheses

Expressions in Ruby, Smalltalk

- No operators per se
  - All "operators" implemented as methods of objects
  - Includes arithmetic, relational, assignment operators
  - Includes array indexing, shifts, bitwise ops
- Can override all within application programs

Ternary conditional operator

- *Conditional ternary operators:* in most C-like languages
- Format:

  condition ? then-stmt : else-stmt
- Ex:

  average = count == 0 ? 0 : sum/count;
- Same as:

```
if (count == 0)
   average = 0;
else
   average = sum/count;
```


Ternary conditional operator

- E.g., VB's `IIf()` function:

```
PF = IIf(grade >= 60, "Pass", "Fail")
```

- Similar to FORTRAN's arithmetic IF statement

```
IF (I-3) 10, 20, 30
```

  - More general
  - Also → a value (r-value)

Ternary conditional operator

- Not just r-values, but also l-values in some languages
- E.g., C, C++, Java (but not JavaScript):

  ((x == y) ? a : b) = 1;

Operand evaluation order

- If a variable → fetch from memory
- If a constant:
  - Statically-bound — already in code
  - Dynamic → fetch from memory
- Parentheses affect order, of course
- Evaluation order:
  - Generally irrelevant…
  - …**except** when operand is a function with side effects

Operand evaluation order

- Example:

```
int foo(int* val) {
  *val = *val * *val;
  return (*val);
}
…
a = 10;
b = a * foo(&a);
```

- If a eval'd first, then b = 10 * 100 = 1000;
- If foo(a) eval'd first, then b = 100 * 100 = 10,000!

Side effects

- In general:
  - A subroutine that returns a value is a **function**
  - A subroutine that does not is a **procedure**
- Functions should not have side-effects
- One opinion: Procedures really shouldn't have any side-effects other than modifying one or more arguments (not as widely-accepted)
- Most languages: no way to enforce this

Possible solutions

1. Write language to disallow side-effects
   - No pass by reference to function
   - No non-local references allowed in function
   - Advantage: works!
   - Disadvantage: inflexible
2. Write language to demand that operand order be fixed
   - Disadvantage: eliminates some compiler optimizations
   - Java, Lisp: Operands eval'd from left→right
   - C, C++: no fixed order

Referential transparency

- Expression is **referentially transparent** if it can be replaced by its value without changing the program

```
ans1 = (fun(a) + b) / (fun(a) + c);
temp = fun(a);
ans2 = (temp + b) / (temp + c);
```

- Referentially transparent if ans2 = ans1
- Absence of functional side-effects is necessary (but not sufficient) for referential transparency
- More in functional languages

Outline

- Introduction
- Arithmetic expressions
- Infix/prefix/postfix
- Overloaded operators
- Type conversion
- Relational & Boolean expressions
- Short-circuit evaluation
- Assignment statements
- Other assignment mechanisms

Relationship of operators to operands

- Most languages: **infix notation**
  - what we use in arithmetic
  - operators between operands
  - e.g., 3 + 4
- Some languages: **prefix notation**
  - operators first, then operands
  - e.g., + 3 4
- Some languages: **postfix notation**
  - operators last, after operands
  - e.g., 3 4 +

Infix expressions

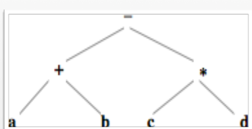
- Infix — inherently ambiguous without defined associativity and precedence
- Different parse trees from different precedence, associativity as specified in grammar
- E.g., a + b - c * d
  - Usually means (a + b) - (c * d)
  - Smalltalk: ((a + b) - c) * d
  - APL: a + (b - (c * d))

Prefix & postfix

- Both prefix and postfix are unambiguous
  - Infix: (a + b) - (c * d)
  - Prefix: - + a b * c d
  - Postfix: a b + c d * -
- Postfix
  - also known as Reverse Polish Notation (RPN)
  - introduced by Polish mathematician Jan Lukasiewicz in early 20th century

Obtaining postfix/prefix

- Consider expression tree for intended meaning:



- Prefix: **preorder** traversal
- Postfix: **postorder** traversal
- Infix: **inorder** traversal (and use depth for precedence, associativity from language definition)

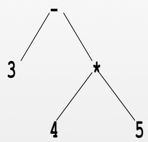
Evaluating postfix

```
read token;
while (not EOF) {
   if token is an operand then
      push token onto stack;
   else // for n-ary operators
      pop top n operands from stack;
      perform operation;
      push result onto stack;
   endif
   read token;
}
pop result from stack;
```

Postfix (RPN) example

```
Input: 3 4 5 * -   (Infix 3 - 4 * 5)

Token Action        Stack
3  Push               (3)
4  Push               (3 4)
5  Push               (3 4 5)
*  Pop 5, Pop 4;
   Push 4*5 = 20      (3 20)
-  Pop 20, Pop 3
   Push 3-20 = -17    (-17)
EOF Pop and return -17
```



Postfix example

Input:

2 3 * 12 3 / + 5 3 * 6 - +

```
Token     Action                 Stack
  2  Push                        (2)
  3  Push                        (2 3)
  *  Pop 3, Pop 2;
     Push 2 * 3 = 6              (6)
  12 Push                        (6 12)
  3  Push                        (6 12 3)
  /  Pop 3, Pop 12;
     Push 12/3 = 4               (6 4)
  +  Pop 4, Pop 6
     Push 6+4 = 10               (10)
  5  Push                        (10 5)
  3  Push                        (10 5 3)
  *  Pop 3, Pop 5;
     Push 5*3 = 15               (10 15)
  6  Push                        (10 15 6)
  -  Pop 6, Pop 15
     Push 15-6 = 9               (10 9)
  +  Pop 9, Pop 10
     Push 10+9 = 19              (19)
  EOLPop and return 19
```


Languages using postfix

- RPN calculators
- Forth
  - e.g., a square function in Forth:

    : squareit dup * ;

    3 squareit .

    9
- Postscript (and PDF)

Unary operators in postfix or prefix

- Can't have same operator be both unary and n-ary
- Don't know how many to take off stack, e.g. (postfix)
- One solution: use different operators
- Another solution: Cambridge Polish notation
  - Parenthesized
  - E.g., for Cambridge Polish prefix notation:

    (+ a b c d) = a + b + c + d

Languages using prefix notation

- Logo
- Lisp & Scheme — pretty much Cambridge Polish prefix notation
- E.g.:
  - Infix: 3 + (4 * 5) - (-23)
  - Prefix: - + 3 * 4 5 ~ 23
  - Lisp:
    - (- (+ 3 (* 4 5) -23)) or
    - (- (+ 3 (* 4 5) (- 23)))

Outline

- Introduction
- Arithmetic expressions
- Infix/prefix/postfix
- Overloaded operators
- Type conversion
- Relational & Boolean expressions
- Short-circuit evaluation
- Assignment statements
- Mixed-mode assignment

Overloaded operators

- Using operator for more than one thing → **operator overloading**
- Common: + for int & float (e.g.)
- Problematic:
  - E.g., * in C/C++
  - Can't really detect missing operator:

```
a = *foo * 2;
a = foo * 2;
```


User-defined overloading

- Some languages allow user-defined overloads (C++, C#, Ada…)
- E.g., Ada:

```
function "+" (a,b : complex) return complex;
```

User-defined overloading

- Functional languages
  - E.g., Lisp

```
(defvar *old+* #'+)

(defmethod + ((a number) &rest args)
  (apply *old+* (cons a args)))

(defmethod + ((a string) &rest args)
  (apply #'concatenate (cons 'string (cons a args))))
```

- OO languages
  - Ruby
  - Smalltalk

Problematic overloading

- JavaScript, Python: + is addition & concatenation
- E.g., JavaScript

```
var x = "10";
var y = x + 5;      // y = 105
var z = x – 3;      // z = 7
```

- E.g., Python

```
x = '10'
y = x + 'hi' # '10hi'
z = x + 3    # error
z = eval(x) + 3 # 13
```


Outline

- Introduction
- Arithmetic expressions
- Infix/prefix/postfix
- Overloaded operators
- Type conversion
- Relational & Boolean expressions
- Short-circuit evaluation
- Assignment statements
- Other assignment mechanisms

Type conversions: Reprise

- **narrowing conversion:**
  - long integer → integer
  - float → integer
- **widening conversion:**
  - integer → long integer
  - integer → float (i.e., float can represent (at least approximately) all integers in range
  - considered widening if conversion retains magnitude, even if it loses precision

Mixed mode conversions

- Operands of different types — **mixed-mode expression**
- Requires **coercion** by language
- Decreases error detection ability of compiler
- Most languages:
  - all numeric types are automatically coerced in expressions
  - use *widening* conversions
- Ada: virtually no coercion in expressions, however: forces user to do *casting*
- Coercion and casting inefficient — require runtime code

Casting

- Explicit type conversions
- E.g., C: **(int) angle**
- E.g., Ada: **Float (Sum)**
- Ada, many other languages: look like function call

Outline

- Introduction
- Arithmetic expressions
- Infix/prefix/postfix
- Overloaded operators
- Type conversion
- Relational & Boolean expressions
- Short-circuit evaluation
- Assignment statements
- Mixed-mode assignment

Relational expressions

- **Relational expressions:**
  - Relational operators, operands of various types
  - Evaluate to Boolean
- Operators vary: e.g., !=, /=, ~=, .NE., <>, #…
- JavaScript, PHP:
  - Two additional operators: === and !==
  - Like == and !=, but no coercion of operands

Boolean expressions

- Operands, operators are Boolean
- Examples:

FORTRAN 77	FORTRAN 90	C	Ada
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not
			xor

Languages without Booleans

- C
  - uses int: 0 = false, nonzero = true
  - reprise: 5 < 3 < 4
    - legal expression, but odd
    - left-associative ⟹ (5 <3) < 4⟹ 0 < 4 ⟹ 1
    - 3 < 5 < 4 ⟹ 1 < 4 ⟹ 1

Languages without Booleans

- Python:
  - Originally no True/False: 0, ‘’, (), [], {} nil, other true
  - Now: 0, 1 — but also True/False
- Perl: 0, ‘0’, (), etc. ⟹ false; else true
- Lisp: nil, t — anything not nil ⟹ true

- Introduction
- Arithmetic expressions
- Infix/prefix/postfix
- Overloaded operators
- Type conversion
- Relational & Boolean expressions
- Short-circuit evaluation
- Assignment statements
- Other assignment mechanisms

Short-circuit evaluation

- **Short-circuit evaluation:** stop executing the (Boolean) expression when some condition met
- For "and" expressions: stop when false encountered
- For "or" expressions: stop when true encountered
- E.g.:

```
Node p = head;
while (p != null && p.info != key)
        p = p.next;
if (p == null) // not in list
        ...
else // found it
        ...
```

  - If `p` null ⟹ doesn't try to eval `p.info`

Short-circuit evaluation

- Without short-circuit, would have to do, e.g.:

```
boolean found = false;
while (p != null && ! found) {
        if (p.info == key)
                found = true;
        else
                p = p.next;
}
```


Short-circuit evaluation

- Not all languages support it
- C, C++, Java: yes, for && and ||
- Ada, VB (.Net):
  - Programmer can specify
  - Use: `and then`, or `else`

Short-circuit evaluation

- Lisp:
  - or, and

```
(or (and (numberp a) (> a b))
    (and (stringp a)…)
```

  - Often used in lieu of conditionals
- Can use to prevent some problems:

```
if (count != 0 && total/count > 10) …
```

Short-circuit evaluation

- Potential problem: not calling functions whose side-effects you want:

```
if f(a, b) && g(y) {
    /* do something */
}
```

- Never calls `g()` if `f()` returns false
- E.g.,

```
if ((a > b) || (b++ / 3)){ }
```

If you were counting on b++…

Outline

- Introduction
- Arithmetic expressions
- Infix/prefix/postfix
- Overloaded operators
- Type conversion
- Relational & Boolean expressions
- Short-circuit evaluation
- Assignment statements
- Other assignment mechanisms

Assignment statements

- General syntax

  target assign-op expression

- Assignment operator:
  - FORTRAN, BASIC, C-based languages:

    ```
    foo = bar * baz;
    ```

  - ALGOL, Pascal, Ada:

    ```
    foo := bar * baz;
    ```


Assignment statements

- APL:

  ```
  a ← 2 3 4 5
  ```

- Lisp — assignments are function-like

  ```
  (set 'a 3)
  (setq a 3)
  (setf a 3)
  (setf (cdr b) 4)
  ```

Targets

- Usually a variable: **l-value** — address
- Can also be a **conditional assignment target** in some languages
- E.g., recall C's ternary operator, also in Perl:

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

  - Either $total or $subtotal ← 0, depending on $flag

Compound assignment

- Shorthand for commonly-needed assignment *idioms*

  a = a + b; // replace by:

  a += b;

- Introduced in ALGOL, adopted by all later C-based languages & VB
- Can be used with almost any binary operator:

a += b; a -= b; a /= b; a &&= b; a ||= b;

a *= b; // careful! easy to confuse with *a = b

y <<= 1;

$s .= "PHP string concatenation";

Multiple assignments

- Some languages: multiple assignments/statement
- Simple:
  - C: `a = b = 0`
  - Python: `a, b = b, a`
  - Lisp: `(setf a 3 b 4…)`

Outline

- Introduction
- Arithmetic expressions
- Infix/prefix/postfix
- Overloaded operators
- Type conversion
- Relational & Boolean expressions
- Short-circuit evaluation
- Assignment statements
- Other assignment mechanisms

Unary assignment

- Most C-based languages: pre- and post-operators ++ and --
- Assignment operators of a sort: change the value of a variable
- Derived from INC, DEC machine instructions
- Examples:

```
sum = ++count;  //inc count then add to sum
sum = count++;  //add to sum then inc count
count--;        //dec count, same as --count;
n = -count++;   // same as – (count++)
x = *p++;       // inc pointer p after dereference
x = (*p)++;     // dereference then inc value
```


Assignment expressions

- C, C++, Java: assignment statement returns a value

```
while ((ch = getchar()) != EOF) {…}
void strcopy (char *s, char *t) {
 while (*s++ = *t++);
 }
```

- Functional languages (of course)

```
(cond
  ((and (setq ch (read-char t nil :EOF))
            (not (eq :EOF ch)))
  …
```


Assignment expressions

- Assignment operator usually has low precedence ⟹ need parentheses in assignment expressions
- Assignment expressions — a type of side-effect ⟹ readability issues

a = b + (c = d / b) - 1;

- Useful for multiple assignment:

total = subtotal = count = 0;

- Note: assignment has to be right-associative for this to work like this!

List assignments

- Perl, Ruby:

($first, $second, $third) = (20, 30, 40);

- Swapping variables:

($first, $second) = ($second, $first);

- Lisp:

```
(destructuring-bind (a b)
     '(2 3)
   (list b a))
```