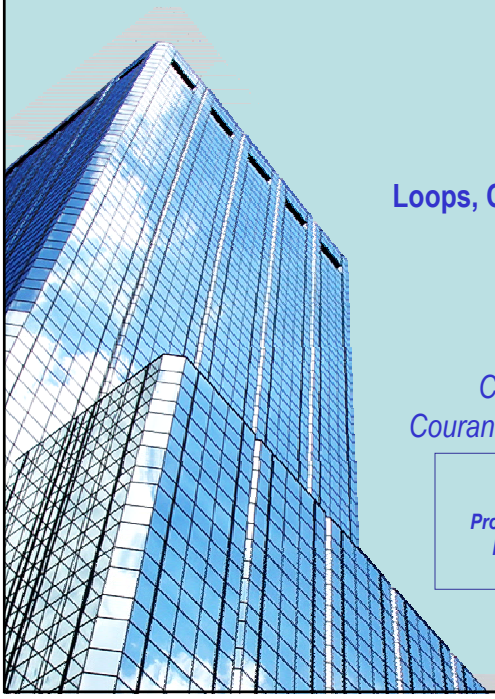


Programming Languages

Session 3 – Main Theme
Control Structures:
Loops, Conditionals, and Case Statements

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

What is the course about?

- Course description and syllabus:
  - http://www.nyu.edu/classes/jcf/g22.2110-001
  - http://www.cs.nyu.edu/courses/fall10/G22.2110-001/index.html
- Textbook:
  - ***Programming Language Pragmatics (3rd Edition)***

Michael L. Scott
Morgan Kaufmann
ISBN-10: 0-12374-514-4, ISBN-13: 978-0-12374-514-4, (04/06/09)

- Additional References:
  - Osinski, Lecture notes, Summer 2008
  - Barrett, Lecture notes, Fall 2008
  - Gottlieb, Lecture notes, Fall 2009
  - Grimm, Lecture notes, Spring 2010

Session Agenda

- Session Overview
- Control Structures: Loops, Conditionals, and Case Statements
- Conclusion

Icons / Metaphors


Information


Common Realization


Knowledge/Competency Pattern


Governance


Alignment


Solution Approach

Session 2 Review

- Use of Types
- Name, Scope, and Binding
- Names
- Binding
- Early vs. Late Binding Time Advantages Detailed
- Lifetimes
- Lifetime and Storage Management
- Garbage Collection
- Scopes
- Scope Rules
- Scope Rules – Example: Static vs. Dynamic
- The Meaning of Names within a Scope
- Bindings of Referencing Environments
- Separate Compilation
- Conclusions

Agenda

1. Session Overview
2. Control Structures: Loops, Conditionals, and Case Statements
3. Conclusion

Control Structures: Loops, Conditionals, and Case Statements

- Control Flow
- Control Structures
- Statement Grouping
- Expression Evaluation
- Sequencing
- Semicolons
- Selection
- Lists / Iteration
- Recursion
- Conclusions

Control Flow (1/3)

- Basic paradigms for control flow:
  - Sequencing
  - Selection
  - Iteration
  - Procedural Abstraction
  - Recursion
  - Concurrency
  - Exception Handling and Speculation
  - Nondeterminacy

Control Flow (2/3)

- Structured vs. Unstructured Flow
  - Early languages relied heavily on unstructured flow, especially goto's.
  - Common uses of goto have been captured by structured control statements.
    - Fortran had a DO loop, but no way to exit early except goto
    - C uses break for that purpose

Control Flow (3/3)

- The Infamous Goto
  - In machine language, there are no if statements or loops
  - We only have branches, which can be either unconditional or conditional (on a very simple condition)
  - With this, we can implement loops, if statements, and case statements. In fact, we only need
    - 1. increment
    - 2. decrement
    - 3. branch on zero
    - to build a universal machine (one that is Turing complete).
  - We don't do this in high-level languages because unstructured use of the goto can lead to confusing programs. See "Go To Statement Considered Harmful" by Edgar Dijkstra

Control Structures (1/2)

- A control structure is any mechanism that departs from the default of straight-line execution.
  - selection
    - if statements
    - case statements
  - iteration
    - while loops (unbounded)
    - for loops
    - iteration over collections
  - other
    - goto
    - call/return
    - exceptions
    - continuations

Control Structures (2/2)

- In assembly language, (essentially) the only control structures are:
  - Progression: Move to the next statement (increment the program counter).
  - Unconditional jump:

    JMP A Jump to address A
  - Conditional jump:

    JMZ R,A If (R==0) then jump to A

  Possible forms of conditions and addresses vary.

Statement Grouping

- Many languages provide a way to group several statement together
- PASCAL introduces begin-end pair to mark sequence
- C/C++/JAVA abbreviate keywords to { }
- ADA dispenses with brackets for sequences, because keywords for the enclosing control structure are sufficient
- **for** J **in** 1..N **loop** ... **end loop**
  - More writing but more readable
- Another possibility – make indentation significant (e.g., ABC, PYTHON, HASKELL)

Expression Evaluation (1/15)

- Languages may use various notation:
  - » prefix : (+ 1 2) – Scheme
  - » postfix : 0 0 moveto – Postscript
  - » infix : 1 + 2 – C/C++, Java
- Infix notation leads to some ambiguity:
  - » associativity : how operators of the same precedence are grouped
    - – x + y - z = (x + y) - z or x + (y - z) ?
  - » precedence : the order in which operators are applied
    - – x + y * z = (x + y) * z or x + (y * z) ?

Expression Evaluation (2/15)

- Infix, prefix operators
- Precedence, associativity (see Figure 6.1)
  - » C has 15 levels - too many to remember
  - » Pascal has 3 levels - too few for good semantics
  - » Fortran has 8
  - » Ada has 6
    - Ada puts *and* & *or* at same level
  - » **Lesson**: when unsure, use parentheses!

Expression Evaluation (3/15)

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /=, <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Figure 6.1 Operator precedence levels in Fortran, Pascal, C, and Ada. The operator s at the top of the figure group most tightly.

Expression Evaluation (4/15)

- Ordering of operand evaluation (generally none)
- Application of arithmetic identities
  - » distinguish between commutativity, and (assumed to be safe)
  - » associativity (known to be dangerous)

  `(a + b) + c` works if `a~=maxint` and `b~=minint` and `c<0`

  `a + (b + c)` does not
  - » inviolability of parentheses

Expression Evaluation (5/15)

- Short-circuiting
  - » Consider `(a < b) && (b < c)`:
    - If `a >= b` there is no point evaluating whether `b < c` because `(a < b) && (b < c)` is automatically false
  - » Other similar situations

```
if (b != 0 && a/b == c) ...
if (*p && p->foo) ...
if (f || messy()) ...
```


Expression Evaluation (6/15)

- Variables as values vs. variables as references
  - » value-oriented languages
    - C, Pascal, Ada
  - » reference-oriented languages
    - most functional languages (Lisp, Scheme, ML)
    - Clu, Smalltalk
  - » Algol-68 kinda halfway in-between
  - » Java deliberately in-between
    - built-in types are values
    - user-defined types are objects - references

Expression Evaluation (7/15)

- Expression-oriented vs. statement-oriented languages
  - » expression-oriented:
    - functional languages (Lisp, Scheme, ML)
    - Algol-68
  - » statement-oriented:
    - most imperative languages
  - » C kinda halfway in-between (distinguishes)
    - allows expression to appear instead of statement

Expression Evaluation (8/15)

- Orthogonality
  - » Features that can be used in any combination
    - Meaning is consistent

```
if (if b != 0 then a/b == c else false) then ...
if (if f then true else messy()) then ...
```

- Aggregates
  - » Compile-time constant values of user-defined composite types

Expression Evaluation (9/15)

- Initialization
  - » Pascal has no initialization facility (assign)
  - » Assignment statements provide a way to set a value of a variable.
  - » Language may not provide a way to specify an initial value. This can lead to bugs.
  - » Some languages provide default initialization.
    - C initializes external variables to zero
  - » System may check dynamically if a variable is uninitialized
    - IEEE floating point uses special bit pattern (NaN)
    - Requires hardware support and expensive software checking
  - » Compiler may statically check – Java, C#
    - May be overly conservative
  - » OO-languages use constructors to initialize dynamically allocated variables

Expression Evaluation (10/15)

- Assignment
  - » statement (or expression) executed for its side effect
  - » assignment operators (+=, -=, etc)
    - handy
    - avoid redundant work (or need for optimization)
    - perform side effects exactly once
  - » C --, ++
    - postfix form

Expression Evaluation (11/15)

- Side Effects
  - » often discussed in the context of functions
  - » a side effect is some permanent state change caused by execution of function
    - some noticable effect of call other than return value
    - in a more general sense, assignment statements provide the ultimate example of side effects
      - – they change the value of a variable
      - – **Side effects change the behavior of subsequent statements and expressions.**

Expression Evaluation (12/15)

- SIDE EFFECTS ARE FUNDAMENTAL TO THE WHOLE VON NEUMANN MODEL OF COMPUTING
- In (pure) functional, logic, and dataflow languages, there are no such changes
  - » These languages are called SINGLE-ASSIGNMENT languages

Expression Evaluation (13/15)

- Several languages outlaw side effects for functions
  - » easier to prove things about programs
  - » closer to Mathematical intuition
  - » easier to optimize
  - » (often) easier to understand
- But side effects can be nice
  - » consider rand()

Expression Evaluation (14/15)

- Side effects are a particular problem if they affect state used in other parts of the expression in which a function call appears
  - » It's nice not to specify an order, because it makes it easier to optimize
  - » Fortran says it's OK to have side effects
    - they aren't allowed to change other parts of the expression containing the function call
    - Unfortunately, compilers can't check this completely, and most don't at all

Expression Evaluation (15/15)

- There is a difference between the container for a value (”memory location”) and the value itself.
  - » l-value refers to the locations. (They are on the left hand side.)
  - » r-value refers to the values.
    - 3 = x + 1 – Illegal! ”3” Can’t be an l-value
    - x = x + 1 – x is both an l-value and an r-value
- Imperative languages rely on side effects
  - » Some languages introduced assignment operators.
  - » Consider a[f(i)] += 4
    - More convenient than a[f(i)] = a[f(i)] + 4
    - Ensures that f(i) is evaluated once
- Some languages allow multiway assignment:
  - » a,b,c = getabc() – Python, Perl

Sequencing

- Sequencing
  - » specifies a linear ordering on statements
    - one statement follows another
  - » very imperative, Von-Neuman

Sequencing

- Pascal: **begin** … **end**
- C, C++, Java: { … }
- Ada: Brackets for sequence are unnecessary. Keywords for control structures suffice.

  for J in 1 .. N loop … end loop
- ABC, Python: Indicate structure by indentation.

Semicolons

- Pascal: Semicolons are separators
- C etc.: Semicolons are terminators

```
begin X := 1;          { X = 1;
      Y := 2             Y = 2;
 end                   }
```

Selection (1/13)

- Selection
  - » sequential if statements

```
if ... then ... else
if ... then ... elsif ... else
(cond
      (C1)  (E1)
      (C2)  (E2)
      ...
      (Cn)  (En)
            (T)   (Et)
)
```


Selection (2/13)

- **if** Condition **then** Statement – PASCAL, ADA
- **if** (Condition) Statement – C/C++, JAVA
- To avoid ambiguities, use end marker: **end if**, "}"
- To deal with multiple alternatives, use keyword or bracketing:

if Condition **then**
    Statements
elsif Condition **then**
    Statements
else
    Statements
end if;

Selection (3/13)

- Nesting and the infamous “dangling else” problem:

```
if Condition1 then
    if Condition2 then
      Statements1
else
    Statements2
```

- The solution is to use end markers. In Ada:

```
if Condition1 then
    if Condition2 then
      Statements1
    end if;
else
    Statements2
end if;
```


Selection (4/13)

- Selection
  - » Fortran computed gotos
  - » jump code
    - for selection and logically-controlled loops
    - no point in computing a Boolean value into a register, then testing it
    - instead of passing register containing Boolean out of expression as a synthesized attribute, pass inherited attributes INTO expression indicating where to jump to if true, and where to jump to if false

Selection (5/13)

- Jump is especially useful in the presence of short-circuiting
- **Example** (section 6.4.1 of book):

```
if ((A > B) and (C > D)) or (E <> F) then
   then_clause
  else
   else_clause
```


Selection (6/13)

- Code generated w/o short-circuiting (Pascal)

```
            r1 := A                    -- load
            r2 := B
            r1 := r1 > r2
            r2 := C
            r3 := D
            r2 := r2 > r3
            r1 := r1 & r2
            r2 := E
            r3 := F
            r2 := r2 $<>$ r3
            r1 := r1 $|$ r2
            if r1 = 0 goto L2
L1:         then_clause        -- label not actually used
            goto L3
L2:         else_clause
L3:
```

Selection (7/13)

- Code generated w/ short-circuiting (C)

```
        r1 := A
        r2 := B
        if r1 <= r2 goto L4
        r1 := C
        r2 := D
        if r1 > r2 goto L1
L4:     r1 := E
        r2 := F
        if r1 = r2 goto L2
L1:     then_clause
        goto L3
L2:     else_clause
L3:
```


Selection (8/13)

- Short-Circuit Evaluation

  if (x/y > 5) { z = ... } // what **if** y == 0?

  if (y == 0 || x/y > 5) { z = ... }

- But binary operators normally evaluate both arguments. Solutions:
  - » a lazy evaluation rule for logical operators (LISP, C)

    C1 && C2 // don't evaluate C2 if C1 is false

    C1 || C2 // don't evaluate C2 if C1 is true
  - » a control structure with a different syntax (ADA)

    -- don't evaluate C2

    if C1 **and then** C2 **then** -- if C1 is false

    if C1 **or else** C2 **then** -- if C1 is true

Selection (9/13)

- Multi-way Selection
  - » Case statement needed when there are many possibilities “at the same logical level” (i.e. depending on the same condition)

```
case Next_Char is
    when 'I' => Val := 1;
    when 'V' => Val := 5;
    when 'X' => Val := 10;
    when 'C' => Val := 100;
    when 'D' => Val := 500;
    when 'M' => Val := 1000;
    when others => raise Illegal_Roman_Numeral;
end case;
```

- Can be simulated by sequence of if-statements, but logic is obscured

Selection (10/13)

- Ada Case Statement:
  - » no flow-through (unlike C/C++)
  - » all possible choices are covered
    - mechanism to specify default action for choices not given explicitly
  - » no inaccessible branches:
    - no duplicate choices (C/C++, ADA, JAVA)
  - » choices must be static (ADA, C/C++, JAVA, ML)
  - » in many languages, type of expression must be discrete (e.g. no floating point)

Selection (11/13)

- Implementation of Case:
  - A possible implementation for C/C++/JAVA/ADA style case (if we have a finite set of possibilities, and the choices are computable at compile-time):
    - build table of addresses, one for each choice
    - compute value
    - transform into table index
    - get table element at index and branch to that address
    - execute
    - branch to end of case statement
  - This is not the typical implementation for a ML/HASKELL style case

Selection (12/13)

- Complications

```
case (x+1) is
   when integer'first..0 ) Put_Line ("negative");
   when 1 ) Put_Line ("unit");
   when 3 | 5 | 7 | 11 ) Put_Line ("small prime");
   when 2 | 4 | 6 | 8 | 10 ) Put_Line ("small even");
   when 21 ) Put_Line ("house wins");
   when 12..20 | 22..99 ) Put_Line ("manageable");
   when others ) Put_Line ("irrelevant");
end case;
```

- Implementation would be a combination of tables and if statements

Selection (13/13)

- Unstructured Flow (Duff's Device)

```
void send (int *to, int *from, int count) {
    int n = (count + 7) / 8;
    switch (count % 8) {
                case 0: do { *to++ = *from++;
                case 7: *to++ = *from++;
                case 6: *to++ = *from++;
                case 5: *to++ = *from++;
                case 4: *to++ = *from++;
                case 3: *to++ = *from++;
                case 2: *to++ = *from++;
                case 1: *to++ = *from++;
    } while (--n > 0);
}
```


Iteration / Loops (1/14)

- Enumeration-controlled
  - » Pascal or Fortran-style for loops
    - scope of control variable
    - changes to bounds within loop
    - changes to loop variable within loop
    - value after the loop

Iteration / Loops (2/14)

- Indefinite Loops
  - » All loops can be expressed as while-loops
    - good for invariant/assertion reasoning
  - » condition evaluated at each iteration
  - » if condition initially false, loop is never executed

while condition **loop** ... **end loop**;

is equivalent to

if condition **then**

while condition **loop** ... **end loop**

end if;

if condition has no side-effects

Iteration / Loops (3/14)

- Executing While at Least Once
  - » Sometimes we want to check condition at end instead of at beginning; this will guarantee loop is executed at least once.
    - **repeat** ... **until** condition; (PASCAL)
    - **do** { ... } **while** (condition); (C)
  - » while form is most common can be simulated by while + a boolean variable:

```
first := True;
while (first or else condition) loop
  ...
  first := False;
end loop;
```

Iteration / Loops (4/14)

- Breaking Out
  - » A more common need is to be able to break out of the loop in the middle of an iteration.
    - **break** (C/C++, JAVA)
    - **last** (PERL)
    - **exit** (ADA)

```
loop
   ... part A ...
   exit when condition;
   ... part B ...
end loop;
```


Iteration / Loops (5/14)

- Breaking Way Out
  - » Sometimes, we want to break out of several levels of a nested loop
    - give names to loops (ADA, PERL)
    - use a goto (C/C++)
    - use a break + lable (JAVA)

```
Outer: while C1 loop ...
   Inner: while C2 loop ...
     Innermost: while C3 loop ...
       exit Outer when Major_Failure;
       exit Inner when Small_Annoyance;
       ...
     end loop Innermost;
   end loop Inner;
end loop Outer;
```

Iteration / Loops (6/14)

- Definite Loops
  - » Counting loops are iterators over discrete domains:
    - **for** J **in** 1..10 **loop** ... **end loop**;
    - **for** (**int** i = 0; i < n; i++) { ... }
  - » Design issues:
    - evaluation of bounds (only once, since ALGOL 60)
    - scope of loop variable
    - empty loops
    - increments other than 1
    - backwards iteration
    - non-numeric domains

Iteration / Loops (7/14)

- Evaluation of Bounds

```
for J in 1..N loop
   ...
   N := N + 1;
end loop; -- terminates?
```

» Yes – in ADA, bounds are evaluated once before iteration starts. Note: the above loop uses abominable style. C/C++/JAVA loop has hybrid semantics:

```
for (int j = 0; j < last; j++) {
   ...
   last++; -- terminates?
}
```

» No – the condition “j < last” is evaluated at the end of each iteration

Iteration / Loops (8/14)

- The Loop Variable
  - » is it mutable?
  - » what is its scope? (i.e. local to loop?)

- Constant and local is a better choice:
  - » constant: disallows changes to the variable, which can affect the loop execution and be confusing
  - » local: don't need to worry about value of variable after loop exits

```
Count: integer := 17;
   ...
for Count in 1..10 loop
   ...
end loop;
... -- Count is still 17
```


Iteration / Loops (9/14)

- Different Increments

ALGOL 60:

```
for j from exp1 to exp2 by exp3 do ...
```

- » too rich for most cases; typically, exp3 is +1 or -1.
- » what are semantics if exp1 > exp2 and exp3 < 0?

C/C++:

```
for (int j = exp1; j <= exp2; j += exp3) ...
```

ADA:

```
for J in 1..N loop ...
for J in reverse 1..N loop ...
```

Everything else can be programmed with a while loop

Iteration / Loops (10/14)

- Non-Numeric Domains

ADA form generalizes to discrete types:

```
for M in months loop ... end loop;
```

Basic pattern on other data types:

- define primitive operations: first, next, more_elements
- implement for loop as:

```
iterator = Collection.Iterate();
element thing = iterator.first;
for (element thing = iterator.first;
  iterator.more_elements();
  thing = iterator.next()) {
  ...
}
```


Iteration / Loops (11/14)

- List Comprehensions

- PYTHON calls them “generator expressions”
- Concise syntax for generating lists
- Example:

```
l = [1,2,3,4]
t = 'a', 'b'
c1 = [x for x in l if x % 2 == 0]
c2 = [(x,y) for x in l if x < 3 for y in t]
print str(c1) # [2,4]
print str(c2) # [(1, 'a'),(1, 'b'),(2, 'a'),(2, 'b')]
```

- Shorthand for:

```
c2 = [ ]
for x in l:
        if x < 3:
                for y in t:
                        c2.append((x,y))
```

Iteration / Loops (12/14)

Pre- and Post-conditions

How can we prove that a loop does what we want? *pre-conditions* and *post-conditions*:

$$\{P\}\ S\ \{Q\}$$

If proposition P holds before executing S, and the execution of S terminates, then proposition Q holds afterwards.

Need to formulate:

- pre- and post-conditions for all statement forms
- syntax-directed rules of inference

$$\frac{\{P \text{ and } C\}\ S\ \{P\}}{\{P \text{ and } C\}\ \text{while } C \text{ do } S \text{ endloop } \{P \text{ and not } C\}}$$

Iteration / Loops (13/14)

- Efficient Exponentiation

```
function Exp (Base: Integer;
              Expon: Integer) return Integer is
    N: Integer := Expon; -- successive bits of exponent
    Res: Integer := 1; -- running result
    Pow: Integer := Base; -- successive powers: Base2I
begin
    while N > 0 loop
        if N mod 2 = 1 then
            Res := Res * Pow;
        end if;
        Pow := Pow * Pow;
        N := N / 2;
    end loop;
    return Res;
end Exp;
```

Iteration / Loops (14/14)

Adding invariants

```
function Exp (Base: Integer;
              Expon: Integer) return Integer is
 N: Integer := Expon;   -- successive bits of exponent
 Res: Integer := 1;     -- running result
 Pow: Integer := Base;  -- successive powers: Base^2^I
begin                   {i = 0} -- count iterations
 while N > 0 loop       {i := i + 1}
  if N mod 2 = 1 then   -- ith bit of Expon from left
   Res := Res * Pow;    {Res := Base^(Expon mod 2^i)}
  end if;
  Pow := Pow * Pow;     {Pow := Base^2^i}
  N := N / 2;           {N := Expon/(2^i)}
 end loop;
 return Res;            {i = lg Expon; Res = Base^Expon; N = 0}
end Exp;
```


Recursion (1/3)

- Recursion
  - » equally powerful to iteration
  - » mechanical transformations back and forth
  - » often more intuitive (sometimes less)
  - » *naïve* implementation less efficient
    - no special syntax required
    - fundamental to functional languages like Scheme

Recursion (2/3)

- Tail recursion
  - » No computation follows recursive call
    - In this case we do **not** need to keep multiple copies of the local variables since, when one invocation calls the next, the first is finished with its copy of the variables and the second one can reuse them rather than pushing another set of local variables on the stack. This is very helpful for performance.

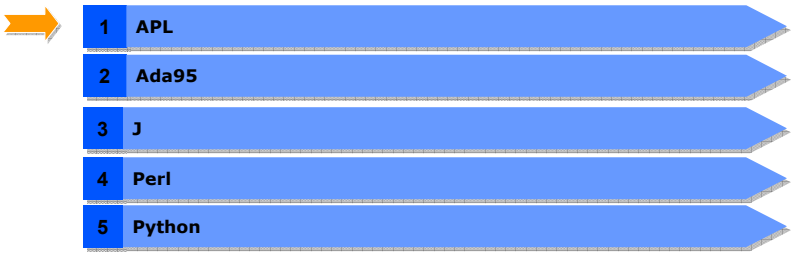
```
int gcd (int a, int b) {
   /* assume a, b > 0 */
        if (a == b) return a;
   else if (a > b) return gcd (a - b,b);
   else return gcd (a, b – a);
}
```


Recursion (3/3)

- Iterative version of the previous program:

```
int gcd (int a, int b) {
/* assume a, b > 0 */
start:
  if (a == b) return a;
  if (a > b) {
        a = a-b;
        goto start;
  }
  b = b-a;
  goto start;
}
```

Appendix



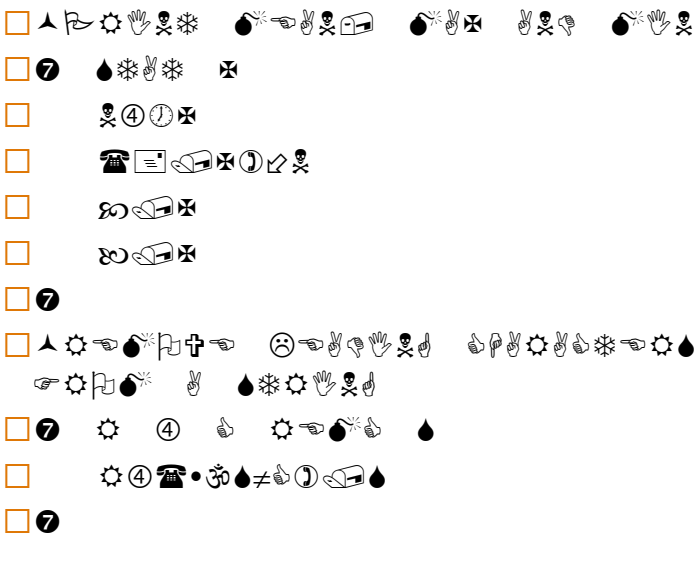
History

- Developed by Kenneth Iverson in the early 1960's
- Tool for mathematicians
  - » Tool for thought
  - » Way of thinking
  - » Very high level language for matrix manipulation
- Widely used by actuaries in Insurance
- Use restricted by special character set including greek letters and other symbols

Typing and Scope

- Dynamic Scope
- Two Types – Numbers and Characters
  - » Automatic conversion between floating point and integer
  - » Strings are character vectors
  - » Boolean values are 0 and 1
- Type associated with Values, not names
  - » Tagged types
  - » Run-time checking

Examples



Syntax

- Simple syntax
  - » Right to left evaluation
  - » infix operators and functions
  - » modifiers (verbs and adverbs)
    - Modifiers are operators that modify the operation of other operators
  - » Can be parsed with only 3 states (Zaks)
- Expression Language
  - » No selection or looping statements
  - » Only goto
- Scalar operators automatically extend to matrices
  - » Loops are unusual in APL

Operations on numbers

Monadic

- ❽, ❾ -- grade up/down

  X ④X [❽X] returns indices of elements in sorted order
- ⌈, ⌊ -- ceiling/floor

  ⌈3.4 = 4

Dyadic

- ⌈, ⌊ -- max, min

  x ⌈ y returns maximum of x or y 2 ⌈ 3 = 3

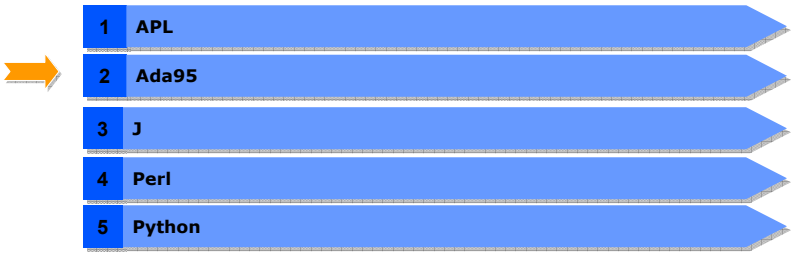
Operations on Arrays

- ⍳ -- interval
  ⍳n returns a vector of integers from origin to n
  ⍳4 = 1 2 3 4
- ⍴ -- size ⍴0 1 2 3 = 4
- Dyadic
  ⍴ -- shape
  reshapes an array
  2 2⍴0 1 2 3 creates a 2 x 2 array
- ⍉ -- Transpose
  Rotates an array along the major diagonal
- ⌹ -- Domino
  Does matrix inversion and division

Operators on Operators

- ∘.+ -- outer product
  1 2 ∘.+ 3 4
  4 5
  5 6
- +.× -- inner product
  1 2 +.× 3 4 – matrix multiplication
  7 14
- +/ -- reduction +/2 3 4 = 9
  equivalent to 2 + 3 + 4
- +\ -- scan +\2 3 4 = 2 5 9
  like reduction, but with intermediate results
  ^\ 0 0 1 0 1 = 0 0 1 1 1 -- turns on all bits after first 1
- Any dyadic operator can be used for + or ×

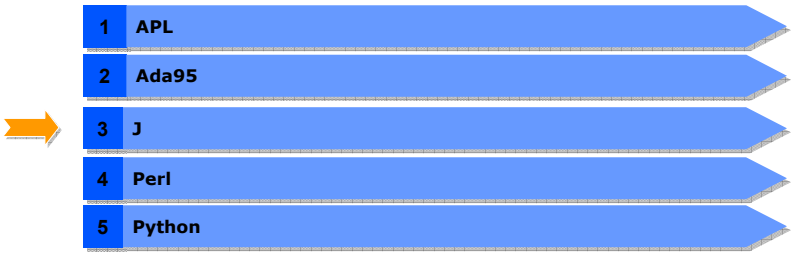
Appendix



Ada95

- Overview of Ada95
  - » http://cs.nyu.edu/courses/fall01/G22.2110-001/pl.lec3.ppt
- Ada Summary
  - » http://www.nyu.edu/classes/jcf/g22.2110-001/handouts/AdaIntro.html
- Notes on Ada
  - » http://www.nyu.edu/classes/jcf/g22.2110-001/handouts/AdaNotes.html
- Syntax of Ada95:
  - » http://www.cs.nyu.edu/courses/fall05/G22.2110-001/RM-P.html

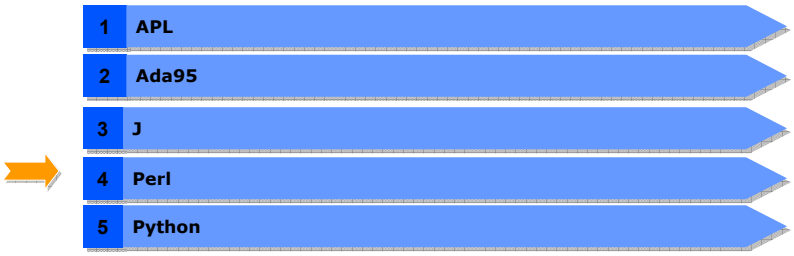
Appendix



J

- See
  - » http://www.nyu.edu/classes/jcf/g22.2110-001/handouts/JDictionary.pdf

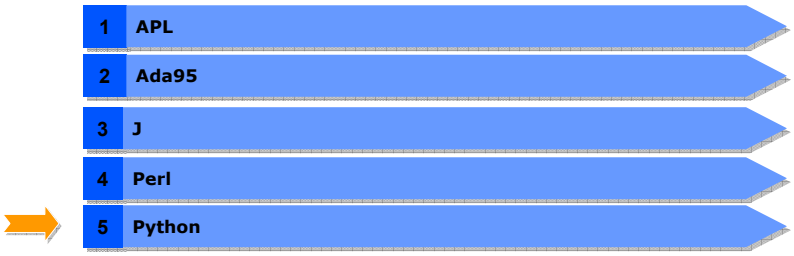
Appendix



Perl

- See
  - » http://www.nyu.edu/classes/jcf/g22.2110-001/handouts/PrototypingInPerl.pdf

Appendix



Python

- Introduction to Python
  - » http://www.nyu.edu/classes/jcf/g22.2110-001/handouts/PythonIntro.pdf
- Python Summary
  - » http://www.nyu.edu/classes/jcf/g22.2110-001/handouts/PythonSummary.pdf
- Notes on Python
  - » http://www.nyu.edu/classes/jcf/g22.2110-001/handouts/PythonNotes.html

Assignments & Readings

- Readings
  - Chapter Sections 6.1-6.5
- Programming Assignment:
  - See Programming Assignment #1 posted under “handouts” on the course Web site
  - Due on March 3, 2011

Next Session:

- Subprograms:
  - Functions and Procedures
  - Parameter Passing
  - Nested Procedures
  - First-Class and Higher-Order Functions