**Operations**

|  |  |
| --- | --- |
| **Characteristics of Operations**  syntax notation (prefix, infix)  precedence  number of operands  extensibility (we will discuss later in the semester)  operations which are undefined for certain inputs  explicit vs implicit operations  explicit vs implicit operands  explicit vs implicit results  invariant vs generic |  |
| **Syntax Notation**  Infix - operand operator operand  prefix - operator operand operand  postfix - operand operand operator  Shortcomings of infix:   * Fixed number of operands * Precedence rules or associativity rules (left-to-right, right-to-left) are needed * Translation is complicated by dealing with infix and precedence * Infix is insufficient since operations requiring more operands require a different syntax   If the number of operands can vary, we need a syntax which marks the end of the list of operands. Lisp uses parentheses. | **Example 1: infix, prefix, and postfix syntax in various languages**  Infix notation in C:  perimeter = 2 \* width + 2 \* length; // rectangle with precedence  perimeter = l1 + l2 + l3 + l4; // four-sided polygon  Infix in APL uses right-to-left associativity:  2 X 4 + 5  result is 18  Prefix notation in Lisp:  (setf perimeter (+ (\* 2 width) (\* 2 length)))  (setf perimeter (+ length1 length2 length3 length4))  Postfix notation in Forth:  2 width \* 2 length \* + perimeter !  As stated in the Programming Languages Overview, languages that use infix abandon infix notation when using functions:  printf("%s, Are you really going to eat that %s? "  , szName, szFood); |
| **Execution-Time Representations**   * Translation from infix to execution code (e.g., machine code, byte code) is fairly complex when dealing with precedence * Therefore, limit the translation and do it prior to execution-time | **Example 2: translation result for infix (A + B) \* D**  FETCH value of A  FETCH value of B  ADD value of A and value of B, producing Accum  FETCH value of D  MULTIPLY Accum by D  **Example 3: translation result for infix A + B \* D**  FETCH value of B  FETCH value of D  MULTIPLY value of B by value of D, producing Accum  FETCH value of A  ADD value of A to Accum  **Example 4: translation result for infix if-then-else**  if (A > B)  C = D;  else  C = E;  Translated code:  FETCH value of A  FETCH value of B  TEST > value of A with value of B, FALSE JUMP To FALSE\_PART  FETCH location of C  FETCH value of D  STORE value of D in location of C  JUMP TO AFTER\_FALSE  FALSE\_PART:  FETCH location of C  FETCH value of E  STORE value of E in location of C  AFTER\_FALSE: |
|  |  |
| **Execution-Time Representations continued - Prefix**   * Prefix notation is widely used in interpretive languages. * The execution-time representation may use a linked list data structure (as in LISP). * Execution may require stacking the operators which isn't a capability directly available on most machine architectures. Some virtual machines have two stacks for evaluating expressions:   + Value stack for stacking the values   + Operation stack for stacking the operations | **Example 5: LISP representation of code as lists**  (setf perimeter (+ (\* 2 width) (\* 2 length))) |
| Evaluating a prefix expression using an operation stack and a value stack for the expression:  (setf perimeter (+ (\* 2 width) (\* 2 length))) | **Example 6: Evaluation of a prefix expression using two stacks**  Evaluating that expression (assuming width is 5 and length is 6)  1. Since we have a list, we must evaluate the setf function  1.1 Push setf and value stack position onto the operation stack  1.2 Push the variable perimeter (not its value) since this is setf  1.3 Evaluate the 2nd argument   |  |  | | --- | --- | | Operator  V S Pos | SETF  0 |  |  |  | | --- | --- | | Value | PERIMETER |   2. Since second parameter is a list, we must evaluate the + function  2.1 Push + and value stack position onto the operation stack  2.2 Evaluate each of +'s arguments   |  |  | | --- | --- | | Operator  V S Pos | SETF +  0 1 |  |  |  | | --- | --- | | Value | PERIMETER |   3. Evaluate +'s first argument which is a list, we must evaluate the \* function  3.1 Push \* and value stack position onto the operation stack  3.2 Evaluate each of \*'s arguments   |  |  | | --- | --- | | Operator  V S Pos | SETF + \*  0 1 1 |  |  |  | | --- | --- | | Value | PERIMETER |   4. Evaluate 2: since it is a constant, push it onto the value stack   |  |  | | --- | --- | | Operator  V S Pos | SETF + \*  0 1 1 |  |  |  | | --- | --- | | Value | PERIMETER 2 |   5. Evaluate \*'s second argument: since it is a variable, push its value onto the value stack   |  |  | | --- | --- | | Operator  V S Pos | SETF + \*  0 1 1 |  |  |  | | --- | --- | | Value | PERIMETER 2 5 |   6. Reached the end of the (\* 2 width) list:  6.1 Pop the operator stack, giving \*  6.2 Apply \* to the elements of the value stack beginning with position 1 and push the result   |  |  | | --- | --- | | Operator  V S Pos | SETF +  0 1 |  |  |  | | --- | --- | | Value | PERIMETER 10 |   7. Evaluate +'s second argument which is a list, we must evaluate the \* function  7.1 Push \* and value stack position onto the operation stack  7.2 Evaluate each of \*'s arguments   |  |  | | --- | --- | | Operator  V S Pos | SETF + \*  0 1 2 |  |  |  | | --- | --- | | Value | PERIMETER 10 |   8. Evaluate 2: since it is a constant, push it onto the value stack   |  |  | | --- | --- | | Operator  V S Pos | SETF + \*  0 1 2 |  |  |  | | --- | --- | | Value | PERIMETER 10 2 |   9. Evaluate \*'s second argument: since it is a variable, push its value onto the value stack   |  |  | | --- | --- | | Operator  V S Pos | SETF + \*  0 1 2 |  |  |  | | --- | --- | | Value | PERIMETER 10 2 6 |   10. Reached the end of the (\* 2 length) list:  10.1 Pop the operator stack, giving \*  10.2 Apply \* to the elements of the value stack beginning with position 2 and push the result   |  |  | | --- | --- | | Operator  V S Pos | SETF +  0 1 |  |  |  | | --- | --- | | Value | PERIMETER 10 12 |   11. Reached the end of the (+ … …) list:  11.1 Pop the operator stack, giving +  11.2 Apply + to the elements of the value stack beginning with position 1 and push the result (22)   |  |  | | --- | --- | | Operator  V S Pos | SETF  0 |  |  |  | | --- | --- | | Value | PERIMETER 22 |   12. Reached the end of the (setf perimeter …) list:  12.1 Pop the operator stack, giving setf  12.1 Pop two elements from the stack: variable, value  12.2 Assign the value to the variable perimeter |
| **Operations which are Undefined for Certain Inputs** | **Example 7: operations that are undefined for certain inputs**   * x / y where y is 0 * array[i] where i is outside the bounds of the array * Python / operator not defined for string values. |
| **Explicit vs Implicit Operations**  Many operations are done explicitly by the programmer. Some operations are implicit (done in behalf of programmer). | **Example 8: implicit operations**   * Saving return address prior to branching to called function. * Allocating memory from the run-time memory stack for automatic variables * Garbage collection * Increasing the size of an unbounded array when necessary |
| **Explicit vs Implicit Operands**  **explicit operands -** list of operands is specified  **implicit operands** - arguments are obtained which aren't listed | **Example 9: implicit operands**   * In some languages (e.g., Python), operations like + implicitly access the data types of each operand. * Automatic memory implicitly accesses a stack pointer when allocating memory from the run-time memory stack. |
| **Invariant Operations and Generic Operations**  **invariant operations -** always produce the same type of results  **generic operations -** type of result based on the operands | **Example 10: invariant vs generic**  Invariant:   * C logical operators (e.g., &&, ||, !) always produce int values.   Generic:   * C arithmetic operators have a result data type based on the operands (could be short, int, long, float, double) |
| **Explicit vs Implicit Results and Side Effects**  **Explicit Results -** well defined by the operation  **Implicit Results -** other results which may not be apparent. These are often known as side effects. | **Example 11-1: side effects**  // Assume C-like syntax and static scope where iA is a non-local for func  iA = 1;  iX = 2;  iY = iA \* func(iX) + iA;  // Given this definition of func, what happens? what else do we need to know?  int func(int iX)  {  iA += iX;  return iX;  }  //What are some possible results?  Approach #1: Access iA before calling func (note: numbers represent order)     * Access iA for left iA reference and right iA reference (value is 1) * 1 \* func(iX) + 1 * Evaluate func(iX) * iA changes to 3. * func() returns 2 * 1 \* 2 + 1 * Evaluate \*: 2 + 1 * Evaluate +: 3 |
| // same code, repeated for viewing convenience  iA = 1;  iX = 2;  iY = iA \* func(iX) + iA;  // Given this definition of func, what happens? what else do we need to know?  int func(int iX)  {  iA += iX;  return iX;  } | **Example 11-2: side effects**  Approach #2: Invoke func first     * Invoke func first * iA changes to 3 * func() returns 2 * Access iA for both references * 3 \* 2 + 3 * Evaluate \*: 6 + 3 * Evaluate +: 9 |
| // same code, repeated for viewing convenience  iA = 1;  iX = 2;  iY = iA \* func(iX) + iA;  // Given this definition of func, what happens? what else do we need to know?  int func(int iX)  {  iA += iX;  return iX;  } | **Example 11-3: side effects**  What evaluation order would cause it to return 5?    Explain that evaluation order:   * Access left iA which is 1 * Evaluate func() * Change iA to 3 * Access right iA which is 3 * 1\*2 + 3 * 2+3 = * 5 |
| // same code, repeated for viewing convenience  iA = 1;  iX = 2;  iY = iA \* func(iX) + iA;  // Given this definition of func, what happens? what else do we need to know?  int func(int iX)  {  iA += iX;  return iX;  } | **Example 11-4: side effects**  What evaluation order would cause it to return 7?  Access operands right to left  Explain that evaluation order:   * Right iA we get 1 * Evaluate func() * Change iA to 3 * Left iA = 3 * 3 \* 2 + 1 * = 7 |
| **What are the disadvantages of side effects?** | * It makes it difficult to prove code correctness * Side effects may cause us to be more specific in a language definition |
| **Short Circuiting**  Many languages recognize situations where executing an entire expression isn't necessary. This must be specified in the language definition so that results are understood.  C provides short circuiting for && and ||. If the first operand for && is false, it doesn't execute the second operand. If the first operand for || is true, it doesn't execute the second operand.  PL/I always executes both sides of logical operators. | **Example 12: short circuiting in C**  C:  if (p == NULL || p->pLeft == pFind)  If p is NULL, the right side isn't invoked. This makes it convenient to use a non-NULL value of p on the right side.  Some languages (e.g., PL/I) don't support short circuiting. |
| **Assignment Categories:**  **simple** assign a simple value to a variable  **compound** combine another operation with an assignment; these are also called augmented assignment  **unary** involves only the variable  (continued below) | **Example 13: examples of some assignment categories**  Simple:  C: ix = iy;  iPerimeter = 2 \* iWidth + 2 \* iLength;  ALGOL: x := 2 \* width + 2 \* length;  Compound:  C: dSum += dCost; // Add dCost to dSum  ulBitMap |= 1 << k; // set a bit in position k  COBOL: ADD COST TO SUM.  Unary:  C: i++; |
| **Assignment Categories (continued)**  **conditional** conditional target | **Example 14: conditional assignments (and conditional expressions)**  Conditional:  Perl: $a > $b ? $x : $y = $result;  If $a > $b, $result is assigned to $x; otherwise, it is assigned to $y.  C: (included since some literature refers to this as a conditional assignment, but it is really just a conditional expression)  iMax = ix > iy ? ix : iy; |
| **Assignment Categories (continued)**  **chain** assign the same value to multiple targets | **Example 15: chain assignments in various languages.**  C: w = x = y = z; // Assign z to y, x, and w  Java: w = x = y = z; // Assign z to y, x, and w  Python: w = x = y = z # Assign z to y, x, and w  PL/I: w, x, y = z; // Assign z to y, x, and w  This is not a chain assignment in PL/I:  w = x = y; /\* x = y is a comparison \*/ |
| **Assignment Categories (continued)**  **multi-target** multiple targets receiving different values | **Example 16: chain assignments in various languages.**  Perl: ($x, $y, $z) = (1, 2, 3); # Assign 1 to $x,  # assign 2 to $y, and  # assign 3 to $z.  ($x, $y) = ($y, $x); # Interchange $x and $y  Python: x, y = y, x # Interchange x and y  # Suppose fruit is a list  fruit = ["orange", "grape", "apple"]  o, g, a = fruit # unpacks the list and  # assigns "orange" to o,  # "grape" to g, and  # "apple" to a  o, \*rest = fruit # The \* says to give anything else  # to the variable rest. Assigns  # "orange" to o and it assigns  # ["grape", "apple"] to rest  fruit = ["orange", "grape", "apple", "clark"]  one, \*two, three = fruit  # Assigns "orange" to one,  # ["grape", "apple"] to two,  # "clark" to three |
| **Assignment Categories (continued)**  **structure** the target can be structure or part of a structure  PL/I and COBOL provide a mechanism to assign attributes from different structure types based on the attribute names. | **Example 17: structure assignments**  Structure to structure:  C: Employee employee, manager;  employee = manager;  Partial structure based on common attribute names:  PL/I: DCL 1 EMPLOYEE,  2 NAME,  3 FIRST CHAR(30),  3 MIDDLE\_INITIAL CHAR(1),  3 LAST CHAR(30),  2 SSN CHAR(9),  2 ADDRESS,  3 STREET CHAR(30),  3 CITY CHAR(30),  3 ZIPCODE CHAR(5),  1 STUDENT,  2 NAME,  3 FIRST CHAR(30),  3 LAST CHAR(30),  2 ABC123 CHAR(6),  2 ADDRESS,  3 STREET CHAR(30),  3 CITY CHAR(30),  3 ZIPCODE CHAR(5);  STUDENT = EMPLOYEE, BY NAME;  /\* Assigns:  STUDENT.NAME.FIRST = EMPLOYEE.NAME.FIRST;  STUDENT.NAME.LAST = EMPLOYEE.NAME.LAST;  STUDENT.ADDRESS.STREET = EMPLOYEE.ADDRESS.STREET;  STUDENT.ADDRESS.CITY = EMPLOYEE.ADDRESS.CITY;  STUDENT.ADDRESS.ZIPCODE = EMPLOYEE.ADDRESS.ZIPCODE;  \*/ |
| **Assignment Categories (continued)**  **array** the target can be a type of array and the source might be a scalar or an array  Array assignments:   * Assign scalar to an entire array * Copy array contents to another array * Copy a reference to an array. You can then change an element, affecting the references. * Assign values to a slice of an array | **Example 18: array assignments**  Scalar Assignment to an array:  PL/I: DCL A(5) FIXED BIN,  X FIXED BIN INIT(5);  A = X; /\* assign the value of X to each element of A \*/  Copy Array Contents to another Array:  PL/I: DCL A(5) FIXED BIN,  B(5) FIXED BIN INIT(1,2,3,4,5);  A = B; /\* copies B into A \*/  Copying a Reference to an Array:  Python: fruit = ["orange", "grape", "apple", "clark"]  profs = fruit  profs[0] = "maynard"  print(fruit) # ["maynard", "grape", "apple", "clark"]  print(profs) # ["maynard", "grape", "apple", "clark"]  Assignment to a slice of an array:  Python: L = [100, 101, 102, 103, 104, 105, 106]  L[2:5] = [22,33,44] # replace items 2, 3, and 4  print (L) # [100, 101, 22, 33, 44, 105, 106]  L[2:5] = [] # remove items 2, 3, and 4  print (L) # [100, 101, 105, 106] |
|  |  |

**©2018 Larry W. Clark,** UTSA CS students may make copies for their personal use