CS323 Lecture: Variables, Constants, Expressions and Assignment 2/9/09

I. Variables and Constants

A. One of the most familiar concepts to most programmers is the notion of a "variable". In fact, it is hard to conceive of how one would structure a programming language without variables. We now want to formalize this familiar concept. What do we mean when we use the term "variable"? What is a variable?

1. ASK CLASS

2. It is tempting to equate a variable with its name. Indeed, we do this all the time. (We speak of "the variable x" or the like.) However, this is not an adequate definition of a variable for two reasons:

   a. It is possible, in most programming languages, for two different variables to have the same name.

   b. It is possible, in most programming language, for one variable to have two or more different names (aliases).

   c. In some languages, it is possible to have ANONYMOUS VARIABLES - variables with no name.

      Example: the allocator new in C/C++/Ada creates a new node that is referred to by a pointer variable or field in another node. The node itself has no name, but is referred to indirectly through another node or a pointer

3. Recall that we earlier considered the question of "what is a data type?" and found that the answer was that a data type can be formally characterized by two sets. In a similar vein, a variable can be characterized as a FOUR-TUPLE:

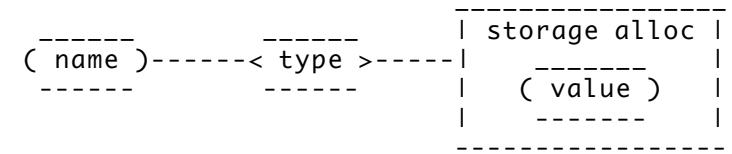
   a. A name

   b. A type

   c. A storage allocation (sometimes called a "reference")

   d. A value

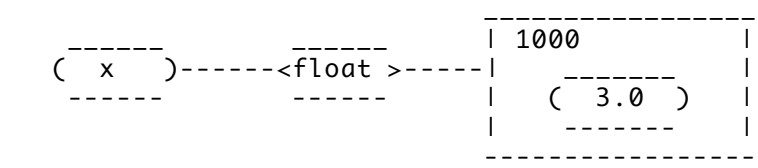
4. We can use a diagram like the following to record these four items of information about a given variable:



Example: suppose we have the following C program fragment, and suppose further that the variable x is stored at memory address 1000:

```
int x;
...
x = 3.0;
```

Then, following the execution of the assignment statement, the variable whose name is x can be described as follows:



5. Of course, for a given variable, it may not be the case that all of these "slots" can be filled in.

   a. As we have noted, some languages allow anonymous variables - variables without names of their own. A linked structure in many languages consists of a interconnected collection of such anonymous variables (often called nodes), pointed to by a single named variable whose value is the address of one of them.

   b. It is sometimes possible for a variable to exist without having a definite type.

      i. This is can occur in dynamically-typed languages such as LISP, between the time a variable is introduced and the time it is first given a value by setq.

      ii. It is also possible in statically-typed languages. For example, consider the Pascal variant record:

```
var
    v: record
         case t: integer of
           0: (r: real);
           1: (i: integer)
    end;
```

      The type of v is not fully determined until the tag field t is filled in. (The type of v is always constrained to be either a record with two integer slots or a record with one integer slot and one real slot, but which case holds is not determined until t is assigned a value.)

c. It is possible for a variable to not have a storage allocation. For example, this is the case with LOCAL VARIABLES of procedures and functions in block-structured languages when the procedure or function is not active.

   Example:

```
int f()
 {
  int i;

  ...
 }

main()
 {
   ...
   f();
   ...
 }
```

   The variable i has no storage allocation during the time period before f is called and again during the time period after f returns.

d. Unless the variable is explicitly initialized when it is declared, it usually has no value until it is first assigned one.

6. Likewise, for a given variable, the values in some of these "slots" can change over time.

   a. The name is not subject to change.

   b. In statically-typed languages, the type cannot change; but in dynamically-typed languages (such as LISP) it can.

   c. The storage allocation of a local variable may be different each time the routine belongs to is called.

   d. The fact that the value of a variable can be changed by assignment is the reason we call it a "variable" in the first place!

B. There are a number of subtleties we need to consider when using this model of a variable as a four-tuple, though.

   1. First, in languages that require that variables be declared, each declaration is normally associated with one variable. This is not always true, though; in particular, in the case of recursion, the same declaration can be responsible for creating several variables.

      Example:

```
(defun fact (x)
   (if (<= x 1)
       1
       (* x (fact (1- x))
   )
)
```

a. If we invoke this by (fact 3), then when we reach the base case of the recursion there will be three incarnations of the parameter x.

b. A case can be made for regarding each of these as distinct variables. Our model of a variable as a four-tuple supports this, since each incarnation will have a different storage allocation and (normally) a different value.

c. However, sometimes it is easier to think of these as different INCARNATIONS of the SAME variable, since they come from the same declaration.

d. Each point of view has its advantages, so long as we can avoid confusion.

2. Another subtlety we must consider is that while a given variable can (at any one time) have only one type, storage allocation and value, it is sometimes possible for a given variable to have two or more names.

   a. The most common way this occurs is through the use of reference (var) parameters.

      Example:

```
int a;

void p(int & b)
{
    ...
}

...
a = 3;
p(a);
```

      Inside p, when it has been called by p(a), the names a and b both refer to the same variable.

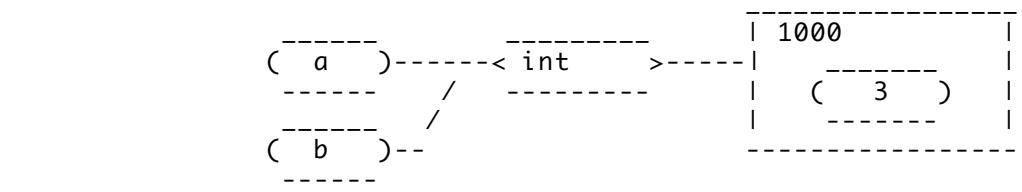
   b. A similar situation can occur with pointers and arrays:

```
int a[10], *b;

main()
  {
    b = a;
    ...
  }
```

   c. We call this situation ALIASING, and we say - in each case - that a and b are ALIASES.

   d. We can depict this using our diagramming technique as follows. (We use the declarations of the first example, and assume a is allocated storage at 1000).



e. Because aliasing makes correctness proofs hard to construct, there is some impetus for language designs to forbid aliasing. For example, Ada does not use call-by-reference for parameters, but rather a similar mechanism known as call-by-value-result that gives the same effect without the possibility of aliases.

   When a parameter is declared in, a copy of the value of the actual parameter is made and referred to by the name of the formal when the procedure is entered. Conversely, for an out parameter, the value of the formal is copied back to the actual when the procedure exits.

3. A third subtlety arises when we consider what a variable's NAME STANDS FOR when we use it in a program.

   a. Normally, when we use the name of a variable in an expression, we mean for it to stand for the VALUE of the variable - e.g.

      System.out.println(x + 3);

      x is used in the println call to stand for the current value of the variable whose name is x.

   b. However, when we use the name of a variable on the left-hand side of an assignment statement, it refers to the storage allocation of the variable - e.g.

      x = 17;

      here x stands for a memory address where the value 17 is to be stored.

   c. In the documentation for C, these two ways of using a variable's name are distinguished by using the terms LVALUE and RVALUE.

      i. The LVALUE of a variable's name is a reference to the storage allocation of that variable; the RVALUE of a variable's name is the variable's value.

      ii. The terms lvalue and rvalue are derived from the fact that when a variable name appears on the left-hand side of an assignment statement it is used as a reference to a storage allocation (lvalue); when it is used on the right-hand side of an assignment statement it stands for the variable's value (rvalue). (However, certain C operators require that their operands be lvalues - e.g. operators like ++.)

iii. Thus, suppose we have

```
int x = 3;
```

and suppose that the variable x is stored at memory address 1000.

- The lvalue of x is 1000.
- The rvalue of x is 3.

iv. Notice that every name has an rvalue, but some to not have an lvalue (e.g. symbolic constants defined by #define in C/C++).

v. In general, expressions have rvalues but not lvalues - e.g. x+3 does not have an lvalue because there is no memory location where the value of x+3 is stored. But certain expressions do have lvalues - e.g. if p is a pointer, then the expressions

```
*p
```

and

```
p -> info
```

have both an lvalue and an r value.

vi. This gives us another way of looking at the distinction between reference and value parameters to a procedure.

- When a variable is passed to a value parameter, what is passed to the procedure is its rvalue.

- When a variable is passed to a reference parameter, what is passed is its lvalue.

- In Java, primitive types, when passed as parameters, are always passed as lvalues; while reference types (Objects and arrays) are always passed as rvalues.

d. There are some contexts in which both the lvalue and the rvalue of a variable are used.

Examples?

ASK

i. The operators ++ and --

ii. Operators like +=, -=, *=, /= etc.

e. Programming language implementations generally handle lvalues and rvalues as follows:

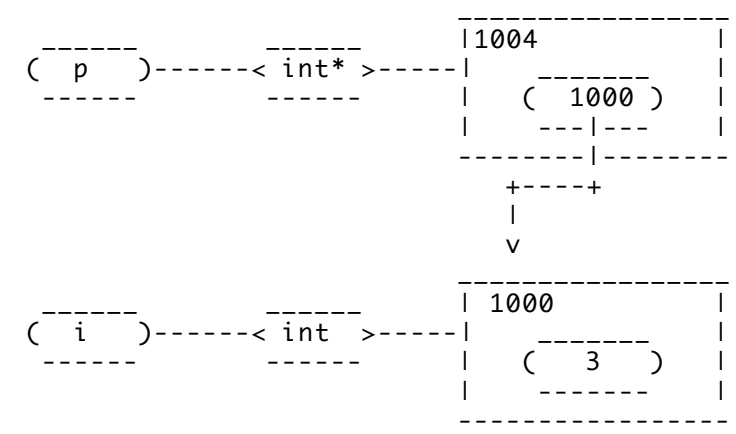
i. A variable's name is associated in the compiler's symbol table with its lvalue.

ii. When it is used in a context where an rvalue is needed, an explicit DEREFERENCING operation is done - i.e. code is generated that accesses the memory location specified by the lvalue, and the value found there is used. (Of course, many hardware instructions do this automatically.)

4. That brings us to some subtletes involving pointer or reference variables.

   a. A pointer variable is a variable use rvalue is the lvalue of some other variable. We can depict them using our diagramming technique as follows, where the diagram pictures the C code shown, assuming that i is at location 1000 and p at location 1004.

      ```
      int i = 3;
      int *p = &i;
      ```



   b. That is to say, pointer variables have reference semantics rather that value semantics - the lvalue of the variable is a reference to some other data object that contains the actual information. To get at the information, one must dereference the pointer - e.g. by using an operation such as * or -> (which combines * and .) in C/C++.

      (In Java, dereferencing is done by the . operator, which is roughly equivalent to C/C++'s ->)

   c. Another issue arises in languages which shield the programmer from some of the details of the differences between value and reference semantics.

      i. For example, Java supports various kinds of collections (e.g. Set) that can be used with any sort of objects. For example, all of the following are legal

         Set<Person>, Set<JFrame>, Set<Object> ...

         But the following is not legal

         Set<int>

         because primitive types are not objects (i.e. variables of primitive types use value semantics, not reference semantics)

      ii. To provide a way around this problem, Java defines 8 "wrapper" classes that correspond to the 8 primitive types (Integer, Double, Boolean, Character). Each has a single field of the corresponding primitive type.

Prior to Java 1.5, to store a primitive in a collection, one had to explicitly "wrap" it in the corresponding wrapper type - e.g. if one had

```
int i;
```

to store i in a Set, one would have to store new Integer(i)

and to use such a value, one would have to "unwrap" it - e.g. one would have to write something like

```
Integer gotten = ...
i = gotten.intValue();
```

iii. Java 1.5 introduced the notion of automatic boxing/unboxing; if one uses a primitive (value) type in a context where a reference type is needed (e.g. insertion into a collection), the compiler generates code to "box" ("wrap") the primitive in an object of the corresponding wrapper type; and if one uses a wrapper type in a context where a primitive type is needed (e.g. arithmetic or assignment), the comiler generates code to "unbox" ("unwrap") the wrapper to the corresponding primitive.

d. Finally, as we have already seen, pointer variables pose two special problems for programming language implementors

i. The problem of dangling references arises when the rvalue of a pointer variable is something that is no longer a valid lvalue. Consider the following C++ program fragment:

```
struct Node {
    int info;
    Node * next;
}
Node * p, q;
...
p = new Node();
p -> info = 3;
q = p;
...
delete p;
...
cout << q -> info;
```

Because of the dispose operation, the variable that p points to is destroyed - i.e. the rvalue of p is no longer a valid lvalue. But since q was set equal to p, q's rvalue is likewise invalid. The attempt to dereference it in the output statement is an erroneous DANGLING REFERENCE. Ideally, a reliable programming language would detect such errors; but in practice this is very hard to implement. [Of course, this is not an issue in languages that don't have an explicit deallocation operator like C++ delete.]

ii. The problem of garbage arises no variables rvalue refers to some object. For example, in the above if we said:

```
p = new Node();
p -> info = 3;
// No assignment to q
...
p = new Node();
```

then the first node's lvalue is no longer the rvalue of any variable.

C. Actually, our discussion of variables can be generalized to the notion of "data objects".

1. In general, a program can work with two kinds of data objects:

a. Variables

b. Constants

2. We can model constants by using the same kind of four-tuple we used with variables.

a. The distinctive feature of a constant is that its attributes (particularly its value) cannot change during program execution.

b. Another distinctive is that a given constant may not have to be associated with a specific storage allocation.

Example: (Java)_

```
private static final int MAX = 80;
...
for (int i = 0; i < MAX; i ++)
    ...
```

i. If the only reference to MAX were in the for loop, there would not need to be - at program run time - any storage location containing the value 80. (The compiler might, in fact, optimize this away since MAX is private, by "hard-wiring" the constant into the code instead of storing it somewhere and referencing it indirectly.

Indeed, even if MAX is used at several places in the clas, the value 80 may be stored at several points in the program code region of memory, but nowhere in the data region.

3. While some languages draw a fairly sharp syntactic distinction between variable declarations and constant declarations, in others the distinction is much less sharp.

a. In Ada, a constant is declared just like a variable, except that the word constant is used - e.g.

```
X: constant integer := 0;
```

declares X to be an integer constant whose value is 0.

b. C++ takes a similar approach. The reserved word const may preceed any declaration to "freeze" the value of the item being declared - e.g.

```
const int X = 0;
```

is semantically similar to the Ada example considered above.

c. Again, Java uses the word final with much the same (though not quite identical) meaning.

4. In addition to the variable and constant data objects visible to a programmer, programs often also work with additional invisible data objects. For example, in an instruction like:

```
y := (x+3) * (x-2)
```

the evaluation of the parenthesized expressions results in the creation of temporary data objects which are later multiplied to form a value to be assigned to y. The programmer never sees these, but they do exist at run time.

II. Binding Time

A. One very important concept that arises in connection with data objects is the concept of BINDING TIME, which we have already discussed. The way a given language handles the binding time of various attributes of a data object is one of its most important characteristics. We will consider the options for binding names, types, and storage allocations in turn. (All languages bind the values of constants at language design time or compile time and the values of variables at run time.)

1. The binding between an object and a name is dictated by the SCOPE rules of a language, as we have discussed. There are two basic possibilities. By way of review, what are they?

   ASK

   a. Static (lexical) scope

   b. Dynamic scope

2. We discussed earlier the difference between statically-typed languages and dynamically-typed languages. This is really a binding time issue: statically-typed languages bind a variable's type at compile time, and dynamically-typed languages do so at run time.

3. When a language binds a variable's storage allocation has a number of significant consequences.

   a. Storage allocation binding time determines whether or not the language can allow recursion.

      i. Languages like FORTRAN and COBOL bind all storage allocations at link time; thus, recursive functions must be forbidden because there is no way to have multiple incarnations of parameters and local variables.

ii. Because block-structured languages bind storage allocations for local variables at run time, they can support recursion.

b. Block-structured languages typically allow for three different classes of variables, distinguished by the binding time of their storage allocation. As we have previously discussd, each class of variable is said to have a different LIFETIME or EXTENT.

   What are they?

   ASK

   i. Static variables - storage is allocated at link time

   ii. Automatic (stack) variables - storage is allocated when a block is entered, and deallocated when it is exited.

   iii. Indefinite (heap) variables - storage is allocated for them by calling a storage allocator procedure such as new in C++ or Java or malloc in C.

c. Notice that we have dealt thus far with two key concepts related to various bindings: the SCOPE OF A NAME BINDING, and the LIFETIME OF A STORAGE ALLOCATION BINDING. These are separate but related concepts.

   i. For both static and automatic variables, scope and lifetime are synonymous in that a variable lives only while the scope in which its name is declared is still active.

   ii. For dynamic variables, there is often no name to have a scope - such variablws are often anonymous.

d. In LISP, scope and lifetime interact in a peculiar way to create what is known (for historical reasons) as the "FUNARG PROBLEM":

   i. This problem arises because LISP allows functions to return objects that are themselves functions, which may later be applied to a set of arguments.

      Example:

      ```
      (defun make-incrementer (increment)
              (lambda (x) (+ x increment))
      )
      ```

      Returns a FUNCTION whose effect is to add some increment to its argument. Therefore

      ```
      (setq f (make-incrementer 3))
      ```

      will make the value of f a function that adds 3 to its argument, and therefore

      ```
      (funcall f 7)
      ```

      should return 10

(Notice that the function created by make-incrementer will be different each time it is called, based on the parameter passed to it.)

ii. The problem here, however, is that the function that defines the new function exits BEFORE the function it defines is called. But the newly defined function depends on one of the parameters of the function that defined it. That is, the function assigned to f must use the variable increment; but since increment is local to make-incrementer and make-incrementer has exited, increment no longer exists!

iii. This problem has been solved in most LISPs by the concept of a CLOSURE. A closure is a combination of a function definition, plus a copy of the lexical environment that existed at the time it was defined. For example, the closure for the function returned by (make-incrementer 3) would include the lambda definition plus a retained copy of the binding of the parameter increment to 3 - which binding would only be visible within the function.

III. Expressions

A. Two of the most common uses for variables are in expressions and in assignment statements. As we have already seen, when a variable name appears in an expression, it is usually used as an rvalue and stands for the current value of the variable; but when it used on the left-hand side of an assignment it is used as an lvalue and stands for a reference to the storage allocation for the variable. Expressions and assignment statements involve some additional subtleties, though.

B. One desirable property of expressions is REFERENTIAL TRANSPARENCY.

1. Informally, an expression is referentially transparent if it does what one would expect it to do from looking at it.

2. This is perhaps best illustrated by a counter example. Suppose we have

```
int i;
int x[10];

int fact(int n)
{
   int f = 1;
   for (i = 2; i <= n; i ++)
       f = i * f;
   return f;
}

int main(int argc, char * argv[])
{
   for (i = 0; i < 10; i ++)
       x[i] = i;
   i = 1;
   x[i] = fact(2) + x[i];
}
```

a. One would expect this program to set x[1] to 3, since 2! is 2 and x[1] is 1 initially.

b. However, the function fact alters the value of the global variable i to 3 as a side effect. (The loop iterates until the end test fails.) Depending on the order in which the compiler generates code, all of the following outcomes are possible:

i. x[1] could indeed be set to 3

ii. x[1] could be set to 5 (2 + x[3]).

iii. x[3] could be set to 3.

iv. x[3] could be set to 5.

c. Clearly, surprises like this are not needed.

3. Referential transparency is lost when execution of portions of an expression has SIDE-EFFECTS that affect other parts of the expression. Typically, these arise as side effects of function calls embedded in the expression.

a. For this reason, it is generally recommended that functions should be so designed as to compute a single value and DO NOTHING ELSE - i.e. they should not alter their parameters or global variables. (This restriction does not apply to procedures, because procedure calls cannot be embedded in expressions, so referential transparency is not an issue with them.)

b. In fact, some programming languages have been designed to enforce this restriction on functions. For example, functions in Ada may only have in parameters - never out or in out parameters. (However, Ada functions may still access and alter global variables.)

C. Expressions are composed of two kinds of elements - operators and operands. Arithmetic (and other) expressions can be written using three different notation systems:

1. Infix - the system we normally use. Operators are written in between their operands: a+b. While this system is familiar to us, it has a couple of key limitations:

a. When an operand appears between two operators, it is not immediately clear which operator is done first - e.g.

```
a+b*c
```

This problem is handled in practice by some combination of:

- A left-to-right or right-to-left rule
- A table of operator precedence (e.g. * is usually done before +)
- Parentheses

Unfortunately, these rules are not always consistent from one system to another; and machine evaluation of such expressions is cumbersome since look ahead is needed before deciding whether a given operator can be applied now.

b. Infix notation can only use operators that have one or two operands - for three or more, an alternate notations such as functional notation (actually a form of prefix) must be used.

2. Prefix or Polish notation - invented by Lukasiewicz. An operator immediately preceeds its operands. Ex:

infix	prefix
a+b	+ab
a+b*c	+a*bc

Note: precedence is never an issue, parentheses are npt needed when the number of operands needed by a given operator is known (e.g. * always needs 2), and any number of operands can be specified for a given operator.

Trivia question: where have you been using prefix notation for years?

Answer: in functions like sin(x). Sin is the operator; x its argument.

3. Postfix or Reverse Polish notation (RPN):

infix	postfix
a+b	ab+
a+b*c	abc*+

Again: no precedence problems; parentheses are npt needed when the number of operands needed by a given operator is known; an operator can be defined to have any number of operands.

4. The latter is especially suited to machine evaluation of expressions. An expression written in postfix can be easily evaluated by using a stack, according to the following rules:

a. When an operand is encountered in the postfix, push it on the stack.

b. When an operator is encountered, pop the required number of operands, apply the operator, then push the result. If there are not enough items on the stack to supply the operands needed, then the expression is ill-formed.

b. At the end of the scan, the stack should contain a single value, which is the result of the expression. (If not, then the expression is ill-formed.)

d. Example: infix 1+(2+3)*(4-5) => 1 2 3 + 4 5 - * +

char scanned	Resultant stack	
1	1	
2	1 2	
3	1 2 3	
+	1 5	
4	1 5 4	
5	1 5 4 5	
-	1 5 -1	Note: second operand popped first
*	1 -5	
+	-4	

5. Moreover, an expression in postfix can easily be converted by a compiler into machine-language code for evaluation. This is especially easy for a stack architecture (like the one we used when we talked about operational semantics).

   Example: The above expression translates into the following stack code:

```
  PUSH    1
  PUSH    2
  PUSH    3
  ADD
  PUSH    4
  PUSH    5
  SUB
  MUL
  POP     result          - corresponds to :=
```

D. Converting an infix expression to postfix is obviously a necessary prelude to any of the above. This task is a bit more complex, but is not terribly hard. It also uses a stack - this time a stack of OPERATORS rather than operands. Note, then, that for direct interpretation of an infix expression two stacks are used: an operator stack to convert from infix to RPN, and an operand stack to evaluate the RPN.

   The desk calculator project you are doing in Ada makes use of this algorithm.

   1. We assign to each operator a precedence value. For example, the following would work for the common arithmetic operators of languages in the C family (though these are not the actual precendence values C uses, because these operators are integrated into a much larger set):

operator	precedence
+	1
-	1
*	2
/	2
%	2

   (We will assume that operators of equal precedence are evaluated left to right).

2. One special issue we must deal with is parentheses.

   a. Note that infix is the only one of the three notations that needs to use parentheses - e.g.

      The infix expressions (1 + 2) * 3 and 1 + (2 * 3) are clearly different, and the parentheses are needed in at least one case to specify the intended interpretation. The equivalent prefix and postfix forms are:

Prefix:	* + 1 2 3	+ 1 * 2 3
Postfix:	1 2 + 3 *	1 2 3 * +

   b. Thus, our algorithm will have to handle parentheses in the incoming infix expression, but will never output parentheses to the outgoing postfix expression.

   c. We will treat parentheses as a special kind of operator. In particular, the '(' will be given precedence value 0. (The ')' doesn't actually need a precedence.)

3. Our algorithm is as follows, assuming the expression is well-formed:

```
for each character in the input do
  switch character scanned of
     case operand:
       output it immediately to the postfix
     case operator:
       while stack is not empty and
             precedence (top operator on the stack) >=
               precedence (character scanned) do
          pop top operator from the stack and output it to postfix
       push character scanned
     '(':
       push character scanned
     ')':
       while top of stack is not a '(' do
          pop top operator from the stack and output it to postfix
       pop the '(' from the stack and discard it
at end of input
  while stack is not empty do
       pop top character from the stack and output it to postfix
```

Example: 1+(2+3)*(4-5):

Input char	Stack	Postfix
1		1
+	+	1
(	+(	1
2	+(	1 2
+	+(+	1 2
3	+(+	1 2 3
)	+(	1 2 3 +
	+	
*	+*	1 2 3 +
(	+*(	1 2 3 +
4	+*(	1 2 3 + 4
-	+*(-	1 2 3 + 4
5	+*(-	1 2 3 + 4 5
)	+*(	1 2 3 + 4 5 -
	+*	1 2 3 + 4 5 -
eol	+	1 2 3 + 4 5 - *
		1 2 3 + 4 5 - * +

4. Error-checking may be added as follows:

   i. Use a variable expected = (operand, operator), initially set to operand.

   ii. Use the following decision table:

Current value of expected	Input	Additional action taken (in addition to basic algorithm)
operand	operand	expected := operator
operand	+, -, *, /	error - operand expected
operand	(	(none)
operand	)	error - operand expected
operand	end-of-input	error - operand expected
operator	operand	error - operator expected
operator	+, -, *, /	expected := operand
operator	(	error - operator expected
operator	)	(none)
operator	end-of-input	(nome)
(either value)	Any character not listed above	error - invalid character

   iii. In addition, the following must be handled:

   - When a ')' is seen, if the operator stack becomes empty before a '(' is found then there is an error - '(' expected.
   - If the operator stack contains any '(' when the end of input is seen (i.e. a '(' is popped from the stack during the final loop), then there is an error - ')' expected.

   iv. An expression which passes all of the above tests will, when evaluated, yield a single value on top of the operand stack, as desired.

E. One issue that is sometimes important with expressions is the ORDER in which operations are done.

   1. We have just discussed how we can deal with a hierarchy of operator precedence levels, as is found in most programming languages. But this still leaves some choices open.

      a. In an expression like f(x)*g(x), the two functions must be called before the multiplication, but the ORDER in which they are called is usually not specified by the language.

      b. In a function call like f(g(x), h(x)), the two functions g and h must be called before f can be called - but again, the ORDER in which g and h is open.

      c. In general, questions like these are left to the compiler, which is free to perform the operations in the order that produces the most efficient code.

   2. Fortunately, questions of order like these usually do not affect the semantics of the program - provided the subexpressions involved do not have side effects. (This is another argument for referential transparency.)

   3. There is one case, though, where order of evaluation can still be important. This case is that of boolean expressions involving the operators and, or.

      a. In a boolean expression of the form e1 and e2, if the first of the two expressions to be evaluated is found to be false, then the second need not be evaluated. The overall expression has to be false. Likewise, in a boolean expression of the form e1 or e2, if the first of the two expressions to be evaluated is found to be true, then the second need not be evaluated. The overall expression has to be true.

      b. As you know, C, C++, and Java define the "and" and "or" operators (&& and ||) to do SHORT-CIRCUIT evaluation - i.e. the second operand is not evaluated if the value of the first (false or true as the case may be) determines the overall result.

         However, not all languages specify short-circult evaluation.

         i. In Pascal, the handling of and and or is implementation defined. In fact, some compilers under some circumstances may do short circult evaluation with the SECOND operand evaluated first (if this is more efficient.)

         ii. Other languages let the programmer have it both ways. For example, Ada defines the operators and and or which are defined to NEVER do short-circuit evaluation - they always evaluate both operands even if the first determines the value of the expression. On the other hand, the operators and then and or else are defined as ALWAYS doing short-circuit evaluation.

             Thus, in Ada, the following will always result in an error if I exceeds MAX:

```
MAX: constant INTEGER := something;
X: array 1..MAX of FLOAT;
...
if I <= MAX and X[I] > 0 then
...
```

But the following will never cause an error:

```
if I <= MAX and then X[I] > 0
```

[If code like the first example were written in Pascal, it behave either way!]

iii. Actually, the C family of languages also provides both alternatives. Though rarely done, & and | can be used for logical and and or, with evaluation of both operands being done.

D. Finally, we want to notice that some languages recognize a special kind of expression called a "COMPILE-TIME EXPRESSION". This is an expression constructed only of values known at compile time, so it can be fully evaluated by the compiler.

1. Example: suppose we want to declare a character string in C, capable of holding up to 10 characters PLUS a terminating null character. We may declare it as

```
#define MAX 10

char str[MAX+1];
```

This is always legal in C. C supports a wide variety of "constant expressions" that can be used in cases like this.

2. However, the corresponding construct in Pascal is not part of standard Pascal, though it is permitted by some compilers:

a. The following may or may not be allowed by a given compiler. It certainly will not be portable.

```
const
    Max = 10;
var
    str: packed array [1..Max+1] of char;
```

b. The following is a portable, but ugly, equivalent:

```
const
    Max = 10;
    MaxPlus1 = 11;
var
    str: packed array [1..MaxPlus1] of char;
```

3. Languages differ quite a bit as to whether they allow compile-time expressions to appear in place of constants, and if so what operations they allow. This is an important question to consider when looking at a new language design.

IV. Assignment Statements

A. One of the first statements most beginning programmers learn is the assignment statement - typically of the form

   variable = expression

   or

   variable := expression

   1. Actually, this statement is peculiar to two of the four language paradigms. Pure functional languages do not do assignment at all (though actually most functional languages do support some variant of it), and logic languages do unification, which is quite different from assignment, though it appears superficially siomilar.

   2. So familiar is this kind of statement in the procedural and oo paradigms that programmers using thse paradigms would be totally lost without it.

   3. Nonetheless, there are two questions about assignment statements we must consider - one practical and one philosophical.

B. The practical issue is the matter of how assignment is implemented by the language: ASSIGNMENT BY COPYING versus ASSIGNMENT BY SHARING, which is related to the two notions of a variable - a variable as a VALUE or a variable as a REFERENCE.

   1. With value semantics, the rvalue of a variable is a value, but with reference semantics, the rvalue of a variable is the lvalue of some object.

      a. For example, as we have previously noted, Java uses value semantics for primitive types, but reference semantics for objects and arrays (which it calls reference types.)

      b. C/C++ ordinary variables use value semantics, but pointers use reference semantics.

      c. C++ references actually involve both kinds of semantics in different contexts.

   2. When a variable has value semantics, the effect of assignment is COPYING. Consider the following Java and C++ examples.

      Java or C++:

      ```
      int a, b;
      ...
      a = b;
      ```

      The value of b is copied and becomes the value of a. Since a and b represent distinct copies of the same value, nothing that is subsequently done to a can affect b.

C++:

```
class Foo
...

Foo a, b;
...
a = b;
```

Again, the value of b is copied and becomes the value of a. Since a and b represent distinct copies of the same value, nothing that is subsequently done to a can affect b.

3. On the other hand, when a variable has reference semanctics, the effect of assignment is sharing. Consider the following Java example:

```
class Foo
...
Foo a, b;
...
a = b;
```

The variables a and b refer to the the SAME object in memory - hence anything done to a (any mutator call) also affects b. This differs dramatically from the C++ example above, because variables of class type have reference semantics, so assignment is done by sharing.

C++ equivalent

```
class Foo
...
Foo * a, * b;
...
a = b;
```

In this case, a and b have reference semantics, so assignment by sharing is done.

4. The semantics of C++ references is fascinating. Though they have reference semantics, assignment is done by copying

```
int & a = something, & b = something else;
...
a = b;
```

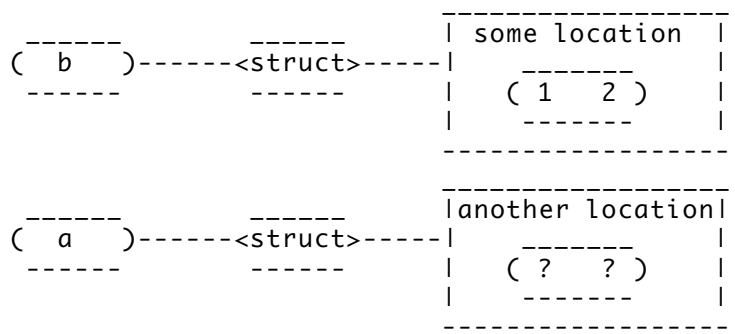
After the assignment, a and b refer to two different integers, which now have the same value (as in Java ints). Since they are two different objects, any subsequent operation on a will not affect b.

5. The differences can be depicted by using our diagramming technique, as follows (using C++ for examples):

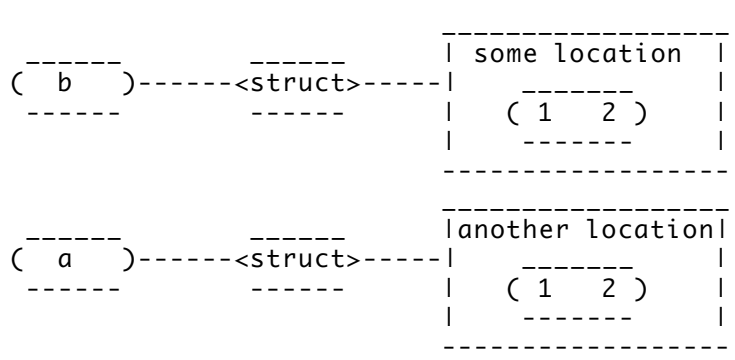
a. Assignment by copying

   Assume struct { int f1, int f2 } a, b; b.f1 = 1; b.f2 = 2;

   BEFORE DOING THE ASSIGNMENT a = b



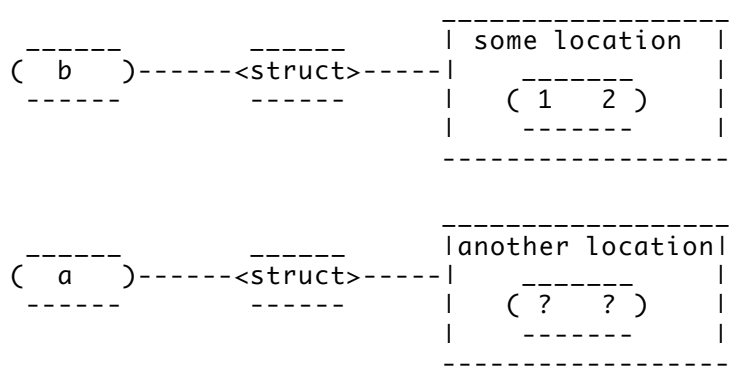
   AFTER ASSIGNMENT



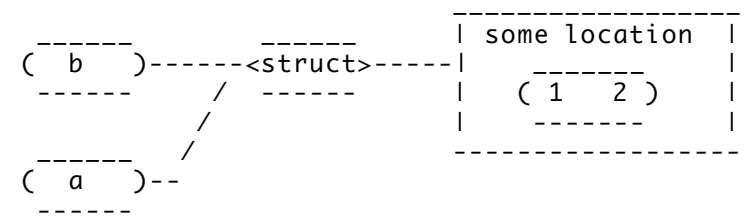
b. Assignment by sharing

   Assume struct { int f1, int f2 } * a, * b; b->f1 = 1; b->f2 = 2;

   BEFORE DOING THE ASSIGNMENT a = b



AFTER ASSIGNMENT



C. We close our discussion of variables and assignment statements on two philosophical notes.

1. There are some theorists who suggest that future progress in programming languages lies in the direction of eliminating them, or at least sharply curtailing their use - as in functional languages.

2. Apart from the question of models of computation, another fundamental problem with variables and assignment statements is this: the programming language notion of a "variable" is fundamentally different from the mathematical notion of a "variable".

   a. In mathematics, a variable is one of two things:

      i. As symbolic name for a possibly unknown quantity. Regardless of where the variable name appears, it always stands for the same unknown.

         Example: "solve x^2 + 2*x + 1 = 0"

         (x is an existential variable - there exist one or more value of x for which the above holds - the task is to find it/them.)

      ii. As a symblic name for a quantity which can assume any value (providing all occurrences use the same value)

         Example: the following is an identity: x^2 + 2*x + 1 = (x+1)^2

         (x is a universal variable)

   b. In a program, a variable can assume different values at different points in the program text, or even at different times at the same point.

   c. The distinction can be put succinctly as follows:

      x = x + 1

      is acceptable in a program (in C or FORTRAN), but mathematically absurd!

   d. The fact that a program variable is a different sort of creature from a mathematical variable greatly complicates program correctness proofs - and even understanding what programs do.

e. This observation has led to some programming languages being based on a SINGLE ASSIGNMENT RULE: each variable starts out unbound (having no value). An unbound variable can be bound by assigning it a value; but once a variable has been thus bound its value cannot be changed.

   i. In this way, a variable behaves like a mathematical variable. It always stands for a single value, which may or may not yet be known (bound).

  ii. One class of programming languages following this rule is the class of DATAFLOW LANGUAGES, which are associated with a non-VonNeumann computer architecture called DATAFLOW MACHINES. Unfortunately, we will not be able to look at these in this course.

 iii. Another language using what amounts to the single assignment rule is PROLOG. This we will look at later.