CSE305 Week 8 Thu.: Expressions (and Assignments)

Sebesta's agenda for *arithmetic expressions* is:

- Design issues for arithmetic expressions
  - Operator precedence rules?
  - Operator associativity rules?
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?

We have covered the first three points well; ,most of section 7.2 recaps them. Most programming languages follow the same rules for reading infix mathematical expressions. A venerable exception is the **APL** programming language: its "keep it simple" rule is that all binary operators have equal precedence and are **right**-associative. Things get funkier with logical operators and user-defined vector (etc.) operators and with extra unary or even ternary operators that some languages have, notably C/C++/Java:

1. Unary: post and pre `++` and `--`; negation `!`, bit-flip `~`, also pointer `*p` and `&x`.
2. Binary: besides `%` (modulus), the shifts `>>` and `<<` and bitwise operators. Plus assignment `=` and its relatives `+=` and `-=` etc. count as expression operators.
3. Ternary: `c ? x : y`, which is the "`if c then x else y` expression."

We will handle many of these in a novel manner---after covering other stuff related to this chapter. The most notable binary operator found in some other languages is `**` for powering (Python, Perl, Ruby) or `^` for powering (BASIC, Lua, R, MATLAB). OCaml uses `^` rather than `.` or `+` for string concatenation.

Array indexing `a[i]` is not really a binary operator between `a` and `i`. It is (IMPHO) best thought of as a unary function application. Function---and method---calls can be mixed into expressions too. In Scala, loops and other statement sequences can yield values---while in OCaml they too count as expressions. Let's call any block of code that (i) takes no inputs and (ii) yields a single value an "atomic expression."

Assignments and Side Effects (7.2 & 7.7)

In the C/C++/Java family, an assignment `x = v` counts as an expression that returns the value v as well as assigning v to `x`. This allows "multiple assignments" like `y = x = v` and that too knocks on the value v. The assignment action is a **side effect**. The `++` and `--` operators also have side effects, and we also count `+=` and similar increment operators as doing assignments. Function/method calls also have side effects---which can ultimately be traced to assignments inside them.

Definition: A program P is **referentially transparent** if for any two atomic expressions e_1 and e_2 within the program that yield the same value (and don't overlap each other), the program P' obtained by switching e_1 and e_2 in the code always has the same results. A (subset of a) programming language has **referential transparency** if every program written in (that subset of) it is referentially transparent.

Three main ways referential transparency can fail:

1. e_1 and/or e_2 have direct assignments inside them.
2. e_1 and/or e_2 make function calls that have side effects.
3. e_1 and/or e_2 and/or functions they call read from non-local variables---once that may have different values or even different bindings in their different scopes.

These reasons are interlinked: the side effects are generally assignments, and a function call may have different results because it reads non-local variables that were assigned different values. The non-local issue will be a focus of chapters 9 and 10. It is the main blocker to a related property that has maybe-more-clearly practical import: that a function call should give the same result when its code body is **inlined** at the point of call.

The **pure** subset of OCaml has been proved to enjoy referential transparency. But that subset does not include arrays since they are mutable. It doesn't include **`ref`** pointers and mutable class fields and a lot of other fun stuff ... thus ML and OCaml are regarded as "impure" functional languages on the whole. Proving logical properties of impure code has had some notable corporate successes.

One practical motive of referential transparency is that it enables both compiler optimizations and runtime **memoization**. The latter means that if a function `f(x)` is repeatedly called with the same value of `x`, then by transparency it will always give the same value v_x on the call, so one need not go through the function's motions again. Just save a record of the calls and return v_x immediately.

Overloading Operators (7.3)

I fork this into one more-general question and one more-specific question:

1. Allow user-defined operators? Practically speaking this mainly means user-defined infix binary operators, such as for classes of vectors and other mathematical objects. This used to be popular. A reason *not* to do this became fully appreciated in the 1990s: Summing up a bunch of vectors or matrices via operators like
$$M_1 \; + \; M_2 \; + \; M_3 \; + \quad \cdots \quad + \; M_n$$
involves making a lot of temporary copies of partial sums. Advances in processing have rendered many old-time efficiency foibles moot, but not this one. Code that loops over individual entries and sums them up without making copies is much more efficient, so much so that the **template metaprogramming** feature was added to C++ to help produce it. The text pushes this forward to a short section 9.11 in the context of overloaded subprograms in general.

2. Have truly polymorphic operators, even allowing users to create them? The most familiar representative example is using **+** to mean *catenation* (of strings and lists and arrays etc., plus unions of sets) as well as numerical addition. There is also the technicality that **+** is already polymorphic on basic integer versus floating-point types, and even those have top-level type subdivisions in C/C++/Java (`short`, `long`, `float` versus `double`, etc.) and many other languages. Note that OCaml shies away from this, using **^** for strings, **@** for lists, and **+.** for floating-point addition.

The current ethic seems to be to rein this in. Infix operators are nice to read but a bit of a pain to implement and mix in with other code.

Type Coercion (7.4 and 7.8)

There is also the point that an overloaded binary operator can have operands of different types, but chooses one of those types for its output. Generally the **wider** type is used, meaning the type whose underlying set is theoretically largest, thus giving a **widening conversion** from the other type. Type coercion is also called **implicit conversion** and means something that happens automatically, as opposed to **explicit conversions**, also called **casts**, written by the programmer. It can also happen just in a simple assignment statement between a variable of one type and a value or variable of a different type.

- Adding a normal `int` and a `long` generally yields a long (whether `int` = 32-bit and `long` = 64-bit or `int` = 64-bit and `long` = 128-bit; if both are 64-bit then *no hay de que*).
- If the result is then assigned back to an `int` variable, a **narrowing conversion** occurs, and since it is between integer types, it is also called a **truncation**. If the computed value was within the `int` range, OK; if not, trouble (*silent* trouble---not an overflow error).
- Adding `int` and `float`, regardless of size, produces a `float`. This happens even when the `float` has smaller or equal size, so that the integer **loses precision**. A 64-bit `double` represents fewer integers than a 64-bit `int`.
- Many C, C++, and Java implementations avoid having separate **+** etc. operations for each combination of `char`, `byte`, `short` and the like. Those are coerced up to `int` for the operation, then truncated back down. Usually this does not lose data, but it does take extra steps.
- Java, however, forbids an assignment that requires an implicit narrowing conversion. For instance, one cannot assign a `float` or `double` value to an `int` variable---unless one writes an explicit cast, that is.

```
double x = 3.5;
//int y = x;       //not OK
int y = (int)x;    //OK
double z = y;      //OK
```

- Type coercion can depend on compiler settings. Using casts is sometimes a "defensive driving" idea, such as writing `x/double(y)` to make double-sure this won't be integer division. Even if y is statically typed as `double`, the code with that declaration might change... Sometimes that helps readability too.
- Allowing implicit conversions makes a type system *weaker* and makes it harder to catch certain bugs.
- This is insanely true when **`size_t`** and **`int`** are mixed in C++. Scala and Python and OCaml duck this at the language-design level by not having first-class unsigned types.
- My story in the previous lecture of Perl implicitly converting strings denoting numbers into those numbers is echoed by the text regarding the old language PL/I. Python clamped down on that by requiring explicit casts `int`(...) and `str`(...) etc.

There is also the subclass-to-superclass kind of implicit conversion. When those classes are value types, actual loss of fields occurs(!). This was a prime motivation for Java allowing objects to be reference types only, so that assigning to a superclass variable never loses data. There are other issues besides data loss, as illustrated by the "Square Versus Rectangle" example in the first lecture. More on this when we hit OOP.

Boolean Expressions and Short-Circuit Evaluation (7.5-7.6)

Boolean expressions are a whole different vein from numerical ones. The one big type conversion issue is whether an `int` can be treated as `false` if 0, `true` otherwise. Perl and other scripting languages---Python too---allow the empty string `""` to test false and all nonempty strings to test true. This (plus assignments returning values) is a great convenience in writing code like

```
while(line = getline) { ... }
```

so as to stop when a completely empty line (not even carriage return) is returned at end-of-file, rather than having to handle an error. The following Python results are amusing:

```
>>> True and False
False
>>> True and ""
''
>>> False and ""
False
>>> True or ""
True
>>> True or "cat"
True
>>> False or "Cat"
'Cat'
```

Part of their logic is that in the third, fourth, and fifth cases, the string was not read at all. This seems to hold even when the second argument is logically more definite or both arguments are strings:

```
>>> x = "Cat"
>>> x or True
'Cat'
>>> y = "Dog"
>>> x or y
'Cat'
>>> y or x
'Dog'
```

What's going on here is (left-to-right) **short-circuit evaluation**. If the left argument to `and` is false, the right argument is not evaluated. Similar if the left argument to `or` tests true, even if the right-hand argument actually being `True` would be more definite.

Boolean short-circuit evaluation is the only option in many programming languages: Python, ML (and OCaml), Perl, and Ruby are mentioned by Sebesta. Even in languages with both kinds (see below), short-circuit is more popular. It is vital to be able to write simple code like

```
if (i >= n || arr[i] < max) { ... }
```

and not worry about `arr[i]` being executed when i is out of range. (Sebesta has a similar example.)

Ada has both the non-short-circuit **and** and **or** and the short-circuit **and then** and **or else**. Java, C#, and Scala use `&` and `|` for **strict** evaluation of both arguments (the term is not quite a synonym of **eager** evaluation here) and `&&` and `||` for the short-circuit ones. In C and C++, `&` and `|` are strict but have a different meaning: they are *bit-wise* **and** and **or** of integers treated as unsigned binary vectors. They have the intended logical effect only if their operands evaluate to the formal `false`/`true` values, or are guaranteed to evaluate only to 0 or to 1. The case to use them is when one or both sides of the operator are Boolean function calls that produce a *desired* side-effect.

Sebesta also mentions short-circuit numerical evaluation, which comes into play when one term of a multiplication evaluates to 0. Python doesn't do it:

```
>>> def f(x):
...    print(x)
...    return 7
...
>>> 0*f(3)
3
0
```

This isn't because Python detected that "print(x)" is a side effect in the function f; it's an instance of Sebesta's conclusion "...in arithmetic expressions this shortcut is not easily detected during execution, so it is never taken." Actually, it is often possible under "highest optimization" settings saying to expressly not care about unevaluated code. Moreover, possible detection at compile time, and any "just-in-time compilation" during runtime, is part of the recent **`constexpr`** feature of C++.

This finishes Sebesta's material in Chapter 7. However, we will mostly care about material that used to begin in the expression chapter but is now diffused into both earlier chapters (on compilation) and later chapters (especially coverage of Scheme in 15)---and in the case of postfix notation (also called RPN for "reverse Polish notation"), vanished altogether (at least from the index).

Morphing and Compiling and Executing Expressions (separate notes)

We have seen that generating a parse tree for an expression is tantamount to generating an expression tree for it too. It is worth noting a few wrinkles from richer kinds of expressions. The first definition comes originally from C but the idea applies more widely:

Definition. An **L-value** or **lvalue** is a variable or array entry or other entity in an expression into which a value can be stored. Whereas, an **rvalue** is a pure value. (Sebesta defines these back in section 5.3, but only in the context of a program variable.)

Example: Consider **`x++`** in C/C++. Here x is of course an lvalue in order for it to be incrementable. But the result of the expression is an rvalue---it is the value **`x`** had before it was incremented. It cannot be incremented again.

However, pre-increment **`++x`** returns not only the value of **`x`** after being incremented, it actually returns a reference to **`x`** which has that value. We can illustrate that by writing that it returns $\mathbf{x}^{\searrow}$ meaning the binding address of **`x`**. So it can be incremented again, as well as being the giver of a value to another variable. Illustrative code:

```
int main() {
    int x = 3;
    int y = ++(++x);
    //int z = (x++)++;
    //int z = ++(x++);
    //int z = ++x;  //OK, z = 6
    int z = (++x)++;
    cout << "Now x = " << x << " and y = " << y << " and z = " << z << endl;
```

This gives `y = 5`, `x = 7`, but `z =` only `6` because it got the value after the pre-increment but before the post-increment. The line marked `OK` works because the return of `x`↘ is **automatically de-referenced** to get the value.

The upshot is that we can allow the assignment operator `=` to appear in our expression trees, likewise `+=` and its other relatives, but only when its left-hand argument is an lvalue. And either kind of increment or decrement can be a unary operator, but only over an lvalue.

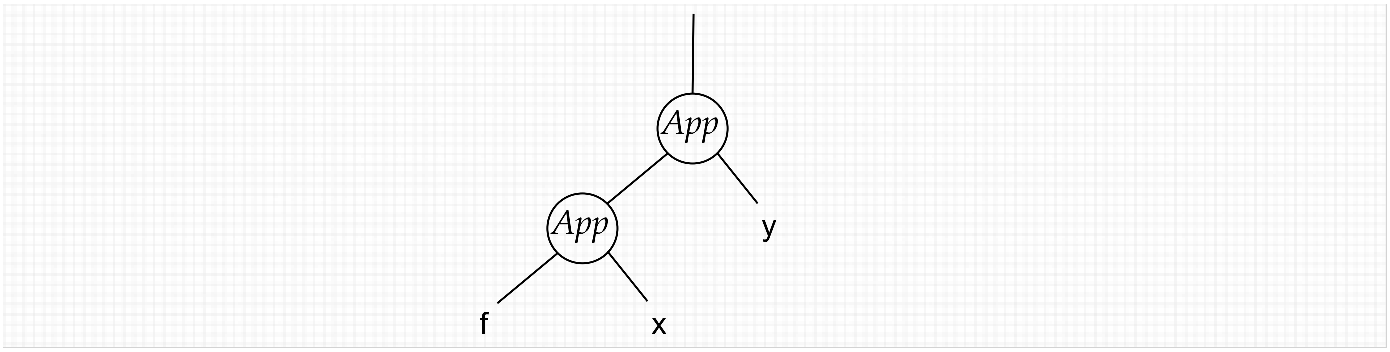
On the other hand, every constant and every result of a purely mathematical operator is an rvalue. Thus we need to mark our expression trees as to which nodes have lvalues or rvalues. This will become more nuanced when we include pointer operations.

General Functions in Expression Trees

A unary function `f(x)` can be treated as a unary node with `f` inside it and `x` below it. If `f` is evaluated on an expression E, then it can have the whole expression subtree for E below it.

A binary function `f(x,y)` can similarly have its own binary node, with `x` and `y` below it (or with subtrees for whatever expressions are passed to `f` in the line(s) of code being compiled). Similarly with a ternary function `f(x,y,z)` using three children, and so on. The if-then-else expression, whether written like in OCaml or with `:` and `?` as in C/C++/Java/etc., can be treated as a ternary node.

Aside: If we curry like `f x y`, then the "invisible binary application operator" should become a node in our tree, rather than treating this the same way as `f(x,y)`. Recall that application groups to the left, so this is the same as `(f x) y` (heavens, not `f (x y)`, what would `x` applied to `y` even mean? and recall from the HW3 TopHat problem referencing "Russell's Paradox" that (`x x`) won't type-check out, whereas `f x x` should be as good as `f(x,x)` would be). It is always important to keep in mind that `(f x) y` says that `x` is properly the only argument to `f` itself, but the call `(f x)` returns a function value that can take `y` as its argument. Well, this is how it would look in a "functional expression tree":



But rest easy---we will be interested in compiling C/C++/Java expressions, not OCaml ones, so we won't have to think of application so abstractly. The camel can take a back seat---well, actually it will be the pusher of our caravan.

One important feature we would like is that the result of our abstract evaluation of the `f` node should be the same if we make an expression sub-tree out of its whole body. (If we have the App node then we put the body there, so it can read the argument `x` part too.) Our evaluation scheme will use a stack architecture which will have the effect of popping out the body of `f` after we process it, leaving just the final value of the call `f(---)` atop the stack at the end, which will then be passed up to the parent node of `f`, whose own body and other children await further evaluation. That this should work the same either way goes hand-in-hand with inlining and referential transparency as discussed above.

A consequence of the stack-based scheme is that it will cause us to evaluate all (expressions given as) arguments to `f` before processing its body. This is called **eager evaluation**. It is opposed to short-circuit evaluation. If you do in Python:

```
def c():
   print("Cats", end="")
   return False
def d():
   print(" and Dogs! ", end="")
   return False

def myand(x,y):
   return x and y

print(myand(c(),d()))
```

then you get "Cats and Dogs!" even though the body of your **`myand`** function is short-circuit Boolean **`and`**. Well, this is a case where inlining the body of your function is NOT equivalent.

Incidentally, a zero-ary function **f()** just provides a value without hanging on any arguments, so it works like a leaf in the tree.

Postfix Conversion

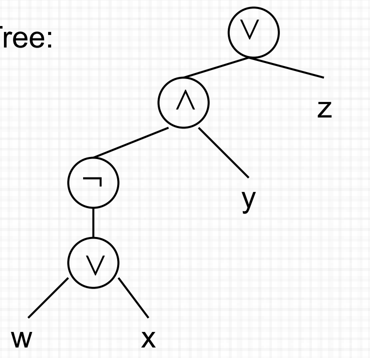
A stack-based scheme means we want to put operators last, rather than read them in infix. This means using **postfix** notation. To convert to postfix:

- Parse the infix notation to build the expression tree.
- Do a (left-to-right) **postorder traversal** of the tree, writing the tree elements out in that order. The operator in the root node will always come *last* (not first as with **preorder** traversal, and not midway as with **inorder** traversal used in binary search trees).
- As with expression trees themselves, you won't get any parentheses. But no worry: the resulting postfix notation is *always unambiguous*.

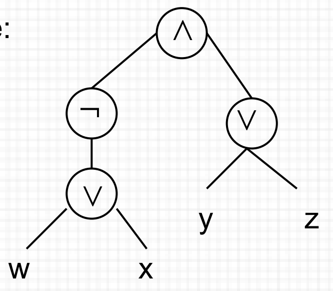
For an example, here is the postfix from the Boolean expression tree on the exam, compared to what you would get if you read $\wedge$ and $\vee$ with the wrong precedence:

Original infix: $\neg(w \vee x) \wedge y \vee z$ Infix with different grouping: $\neg(w \vee x) \wedge (y \vee z)$

Tree:



Tree:



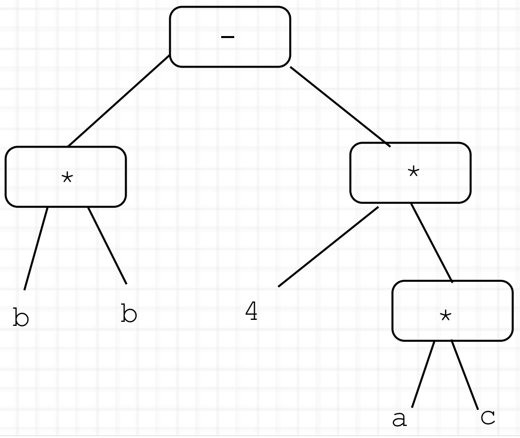
Postfix: w x ∨ ¬ y ∧ z ∨ Postfix: w x ∨ ¬ y z ∨ ∧

FYI Prefix: ∨ ∧ ¬ ∨ w x y z Prefix: ∧ ¬ ∨ w x ∨ y z

Cambridge Prefix: (∨ (∧ (¬ (∨ w x)) y) z) Camb. P.: (∧ (¬ (∨ w x)) (∨ y z))

Cambridge Prefix---used in Lisp---is what you get by doing the preorder traversal while sticking (and) around every non-leaf node. With pure prefix---as with postfix---you don't get any ()s, but with Cambridge prefix you get "Lots of Infuriatingly Stupid Parentheses." That prefix notation is unambiguous without using any parentheses was expounded on by the Polish logician Jan Łukasiewicz and some compatriots, so it was called "Polish notation"---and postfix became "Reverse Polish Notation" (more decorously abbreviated RPN).

The quadratic forumla expression tree from week 2, but with "4ac" grouped differently:



Infix: $b*b \;-\; 4*(a*c)$

Postfix: b b * 4 a c * * -

Prefix: - * b b * 4 * a c

Camb. P.: (- (* b b) (* 4 (* a c)))

To apply the unary **`sqrt`** function to that discriminant, the tree would have a unary node (sqrt) going to the (-) node. Since it is the new root, it goes last: **b b * 4 a c * * - sqrt** . The whole quadratic formula has more above the sqrt:

b - b b * 4 a c * * – sqrt ± 2 a * /

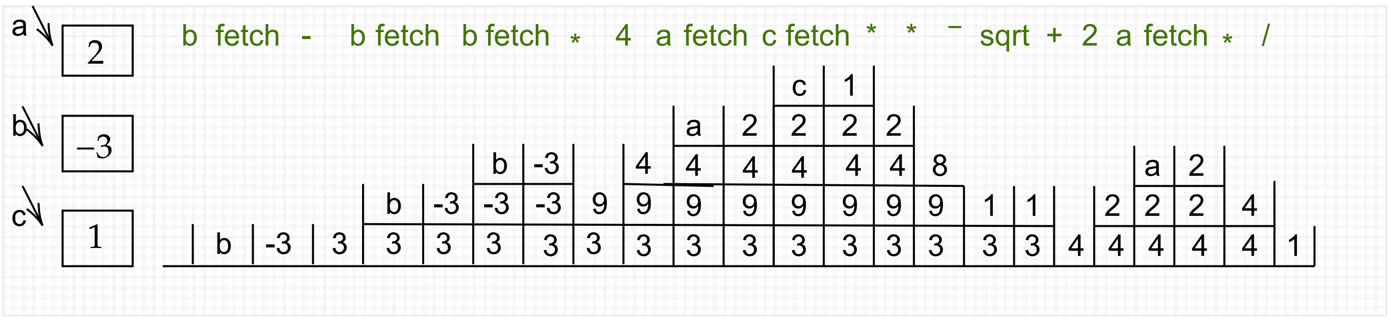
OK, "±" isn't really an operator, but it helps keep track of where things are. The first – sign has to be unary. We need not worry about the "infix gotcha" that `(-b + sqrt(b*b - 4*(a*c)))/2*a` gives the wrong result---automatically the **2 a * /** part multiplies the **2** and **a** together before doing the division.

Rudimentary Stack-based Object Code

We introduce two new postfix operators that work with variables in our postfix-converted expressions. One is unary and simply **`fetch`**es the value from a variable. We can just insert it into our postfix above since everything there is treated as rvalues (let's also replace ± by just + to take the upper branch of the square root):

b fetch - b fetch b fetch * 4 a fetch c fetch * * – sqrt + 2 a fetch * /

Suppose that our storage objects for the three variables hold the values shown at left. We can fetch them while processing the postfix code on a stack. The introduction of a new item in the postfix---as opposed to processing an operator or function---pushes it atop the stack. The fetch operation replaces the item atop the stack---if it is a variable---by the value stored there. (Later we will want to remind ourselves that what the stack really holds from a "variable" is not its name like "a", "b", "c" but rather its binding address, which I denote by $a^{\searrow}$, $b^{\searrow}$, $c^{\searrow}$. It's just cumbersome to keep writing those superscripted arrows on the stack trace.)


a
2
b
–3
c
1
b fetch - b fetch b fetch * 4 a fetch c fetch * * – sqrt + 2 a fetch * /

Note that a binary operation replaces the top two values on the stack by one new value. The binary minus sign – does `second` – `top`, not `top` – `second`. Another natural way to trace the stack is to picture the postfix items as going down---the way we picture lines of code---and the stack growing toward the right:

```
b
          b
fetch
          -3
-
```

```
         3
b
         3   b
fetch
         3   -3
b
         3   -3   b
fetch
         3   -3   -3
*
         3   9
4
         3   9   4
a
         3   9   4   a
fetch
         3   9   4   2
c
         3   9   4   2   c
fetch
         3   9   4   2   1
*
         3   9   4   2
*
         3   9   8
-
         3   1
sqrt
         3   1
+
         4
2
         4   2
a
         4   2   a
fetch
         4   2   2
*
         4   4
/
         1
```

Either way, the stack ends up holding just the final value.

The other operation is binary and works on the previous two elements as we process our postfix code. The element just before it must be a literal value, perhaps one that was just "fetch"ed, but the element before that *must be an lvalue*. It is called **`store`**. The "stack semantics" of store is entirely defined by saying what it does to the top two elements of the stack:

```
          ...    ...    x    v
store
          ...    ...    v                      now x↘ [ v ]
```

- It is important to re-draw the affected storage object to show its new value.
- The top two elements on the stack are replaced by just the *value* (not the variable).

The translation $tr(\cdots)$ of an assignment of the form x = E is simply stated as:

$$\texttt{x} \quad tr(E) \quad \texttt{store}$$

Since the value is left atop the stack, we can handle a "double assignment" y = x = E simply by

$$\texttt{y} \quad \texttt{x} \quad tr(E) \quad \texttt{store} \quad \texttt{store}$$

If we complete a simple assignment statement instead by x = E; then it means we're not going to make any further use of the assigned value, so we just pop it:

$$\texttt{x} \quad tr(E) \quad \texttt{store} \quad \texttt{pop}$$

That's right: the semicolon in C/C++/Java is translated as "pop". Also nifty here is that the translations of expressions can just be substituted in. So you can translate

```
upperSol = ((-b) + Math.sqrt(b*b - 4*a*c))/(2*a);
```

by throwing the entire postfix translation of the quadratic formula above in place of $tr(E)$. This freedom of substution makes it easy to plug-and-play the translations of other code elements, which can often be stated in shortcut schematic form. For example, **`x += `**E**`;`** is referentially equivalent to **`x = x + `**E**`;`** and noting that the *rvalue* use of **`x`** needs an extra `fetch`, we get:

$$\texttt{x} \quad \texttt{x} \quad \texttt{fetch} \quad tr(E) \quad \texttt{+} \quad \texttt{store} \quad \texttt{pop}$$

Decrement **`x -= `**E similarly comes out as `x x fetch` $tr(E)$ `- store pop`.

Other Operations

Both pre-increment and post-increment are unary operations. Don's get confused by "pre" versus "post"---they both come *after* the variable x in postfix notation. So in postfix, call them respectively **`x pre++`** and **`x post++`**. Now we can state the translation rules:

- **`x pre++`** becomes `x x fetch 1 + store pop`
- **`x post++`** becomes `x fetch x x fetch 1 + store pop`

The latter looks mysterious, especially because it has a `pop` without there being a semicolon to end the statement. The point is that we need to leave the original value v_0 of x (before the increment) as the final value atop the stack. The initial `x fetch` does that. Then we imitate the previous line. That leaves the incremented value v_1 above the original v_0, so we finally need to pop off the v_1.

Added: The **`x++`** versus **`++x`** example above says that the unary-operator node for **`(post++)`** in our expression trees must be labeled "**R**"value (of course the node below it, whether a leaf with "**`x`**" or something else, must be labeled "**L**"value), but the node for **`(pre++)`** can be either. Since the statement **`y = ++(++x);`** is legal, we get a tree with root node **`(=)`** whose right-child must be labeled **R**, but it is a **`(pre++)`** node. The latter has another **`(pre++)`** node as its single child, and it must be labeled **L** (along with the leaf x below it). The upshot is that a **`(pre++)`** node can "switch-hit", and *hmmm...*, does the above translation work in both cases?

Array accesses use the formula covered in the Tue. 3/28 lecture. If e is the size in bytes of the element type, and recalling that the name of the array stands for its base address, then:

- in an lvalue context, **`arr(i)`** becomes `arr e i lo - * +`
- in an rvalue context, **`arr(i)`** becomes `arr e i lo - * + fetch`

Thus **`arr[i]`** also "switch-hits"---and uses two different translations for those cases. If **`arr`** is 0-based then **`lo`** = 0, and if **`i`** is a variable then we need to fetch from it. Then we have:

- in an lvalue context, **`arr[i]`** becomes **`arr e i * +`**
- in an rvalue context, **`arr[i]`** becomes **`arr e i * + fetch`**

The C/C++ pointer operations & and * are translated this way:

1. **`&x`**, which is postfix is **`x &`** and can only be an rvalue, is translated as just `x`.
2. **`*x`** when being read from is `x fetch fetch`
3. **`*x`** in an lvalue context, e.g. as target of an assignment, is just **`x fetch`**.

The basic idea is that **`&`** always subtracts a `fetch` while **`*`** adds one. Since you only get `x fetch` to begin with when **`x`** is being read from, **`&`** can only occur in an rvalue context.