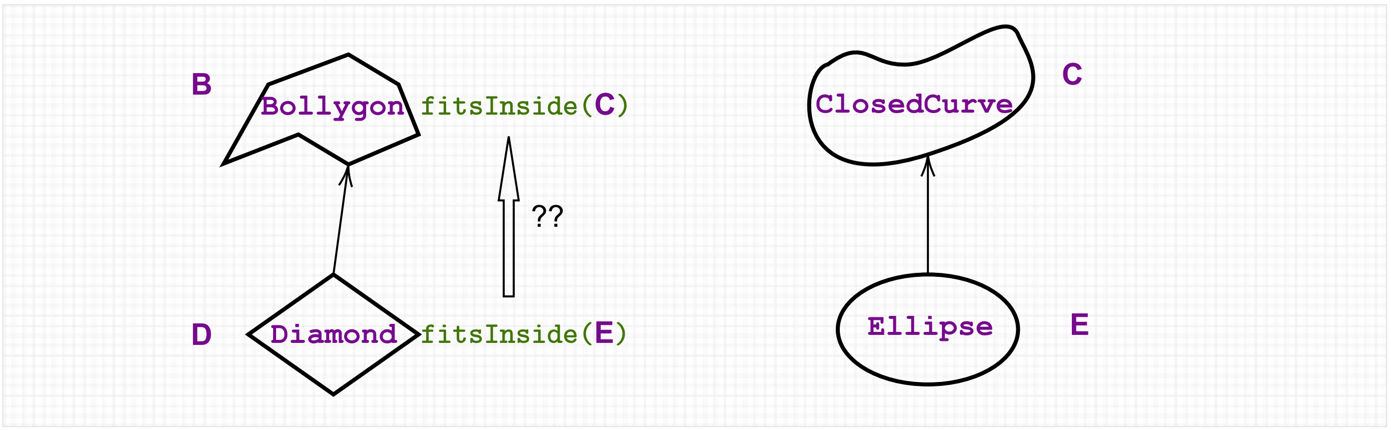
CSE305 Week 13: More Modern Design Issues For OOP Languages

We continue with some substantial topics not given focus in the text. Then we tie things back to OCaml and move on to Prolog.

Covariance and Contravariance

The issue of method compatibility is not just with binary methods. Consider the following object hierarchy. The previous class `Polygon` is renamed "`Bollygon`" so that it has the same letter "B" as "base class". Ellipse derives from ClosedCurve, but these



The code for whether a diamond fits inside an ellipse is much simpler and faster than whether a general polygon can fit inside an ellipse, or a diamond can fit inside an arbitrary (smooth and continuous) closed curve. Hence there is good reason to call it when applicable. The question is, can it be incorporated within the usual O-O dynamic-dispatch and override mechanism?

The problem is, what happens when we call `b.fitsInside(c)` when `b` holds a `Diamond` object `D` but `c` might hold only a basic `ClosedCurve C`? Dynamic dispatch says that finding `D` is the trigger to bind and call the code for `Diamond.fitsInside`. But that code relies on the argument being an `Ellipse`. In practice, the possibility of a code crash is averted because:

- `Diamond.fitsInside` is not considered a legal override of `Bollygon.fitsInside`.
- Newer compilers will probably warn that it is only an overload, and give a compile error if you try to mark it an `@Override`.
- So the code `b.fitsInside(c)` will be statically bound to be the base-class version (only). This will be perfectly safe, just not take advantage of better code for derived-class shapes that are more regular.
- Regardless, in OCaml terms, this public method prevents `Diamond` from being a true subtype of `Bollygon`. This is an example of a derived class being a "subclass but not a subtype." (Added: One can reach similar conclusions about a mutable-Square being incompatible with a mutable-rectangle, but this

The buzzwords for this situation are:

- Compatibility for return types is **covariant**, meaning it works in the same direction as extension/subclassing. For instance, having `Diamond.foo()` return an **`Ellipse`** would be fine to override a method `Bollygon.foo()` that returns a **`ClosedCurve`**.
- It is also covariant "by default" for the invoking object.
- But for parameters it is **contravariant**. For a general example, `Derived.bar(C arg)` can override `Base.bar(E arg)`.

(The last two points also exemplify a way the Ada95 syntax of having the invoking object be inside the parentheses with the other parameters was misguided.)

One "non-solution" is that programmers can always write (ugly) code like so (C++):

```
for (Bollygon* bp: bollygons) {
   Diamond* dbp = dynamic_cast<Diamond*>(bp);

   for (ClosedCurve* cp: curves) {
      Ellipse* ecp = dynamic_cast<Ellipse*>(cp);
      bool fitTest = (dbp && ecp) ? dbp->fitsInside(*ecp)
                                  : bp->fitsInside(*cp));
      if (fitTest) {
         ...
      }
   }
}
```

The call `dbp->fitsInside(*ecp)` can then be *statically bound* to the derived-class method body by the types given to the `dbp` and `ecp` pointers. This call will only be executed when the invoking object and the argument have the right types. Then it will give the desired time savings over the base-class code---which is run in all other cases.

Another idea is sometimes called "re-dispatching": have the body of `b.foo(c)` call `c.bar(...)` which does the actual work. This changes how the code is written, though.

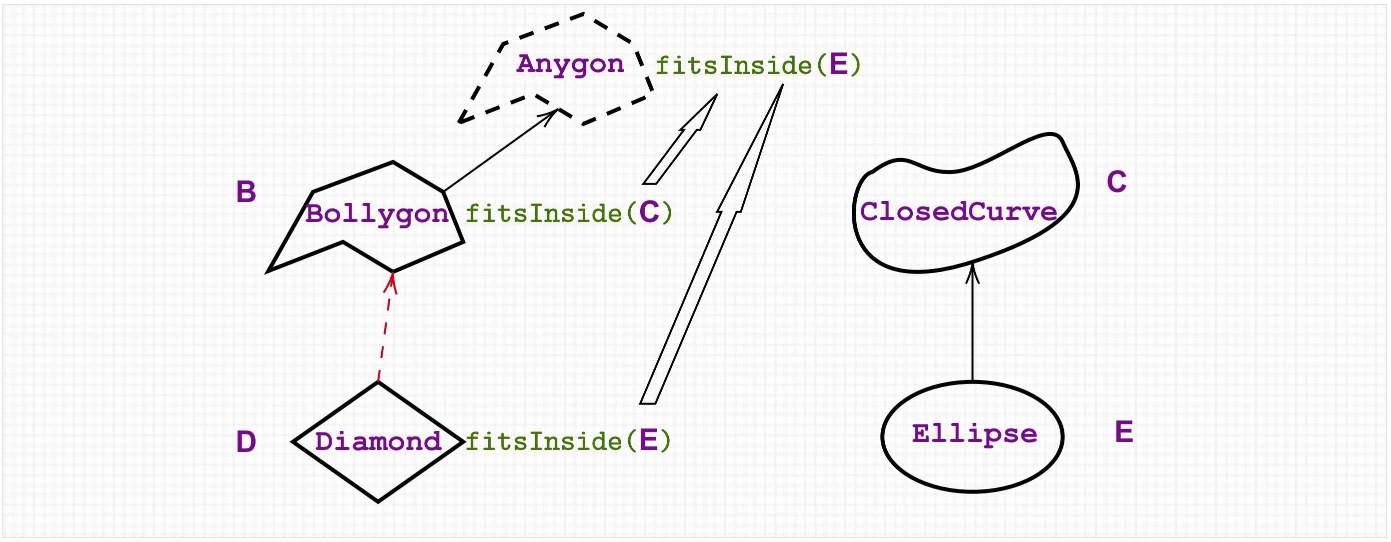
The desire to make this mechanism work more naturally and save programmer carpal tunnel syndrome (but ChatGPT may soon write such crufty-but-automatic code for us, like my `PolyOps.ml` file for the project) has led to numerous proposed solutions, including the following:

1. **Multiple Dispatch**: Here this would automatically poll *both* the invoking object and its argument for the actual types, calling the faster code if both are derived. The syntax might be something like **`(b,c).fitsInside()`**.
   (a) Pro: Seamless and effective

    (b) Con: Slow, complicates run-time system.
2. **"Tandem Objects"**: Make `(b,c)` an actual `Tuple` object, not just handy syntax. Prescribe rules for its full joint ownership of methods, versus individual ownership in the two classes.
    (a) Pro: More efficient than multiple dispatch because only one polling of the actual object need be done.
    (b) Con: more work for the programmer, clutters up class system, not seamless.
3. **Retroactive Abstraction**: This means allowing users to declare new *superclasses*. In this case we could declare (in made-up syntax):

```
class Anygon super_of Bollygon {
   ...
   bool fitsInside(Ellipse e) { ... }
}
```

This changes the Booch-style diagram so that `Diamond.fitsInside` and `Bollygon.fitsInside` are both legal overrides of `Anygon.fitsInside`:



Doing this makes the whole system 'legal"---possibly after removing the Booch arrow from `Diamond` to `Bollygon`. However:

- It does not solve the original problem of wanting to loop through arrays of various shapes, which have to be given the base-class type designators `Bollygon[]` and `ClosedCurve[]`, and call `b.fitsInside(c)` on each pair. You could do the explicit "**upcast**" `Anygon(b).fitsInside(c)` if the system doesn't upcast automatically, but you would still need to **downcast** the variable `c` to the class **E** just to invoke the base method.
- Writing `Anygon` in the code would clutter the class hierarchy and amplify the yo-yo problem.

The interesting question for us, however, is whether OCaml's subtype-compatibility system, which infers compatible types as needed, is already doing this kind of thing under-the-hood. Consider simply doing the following from the bare `ocaml` prompt---nothing preloaded:

```
# let f x = x#pop;;
val f: < pop : 'a; .. >  ->  'a  =  <fun>
```

Although this code is nothing but a "stub" (at most a prototype of a function), OCaml allows it. OCaml infers the existence of a type "`< pop : 'a; .. >`", which it says is the domain of the function `f`. The blue `..` means that any extension---here meaning subtype---of this type gives a valid argument to `f`. This will include your project object---presuming that you keep the name `pop` from the sample code covered in the Tue. Week 12 lecture.

The point to reflect on is that this domain type was not explicitly coded by me or anyone. It is determined by OCaml as the "least common denominator"---a more technical term is **meet**---of all types that can be the invoking object for this function. In a relatable but maybe different sense, the class **`Anygon`** is the minimum conjunction of **`Diamond`** and **`Bollygon`** needed to rectify the class hierarchy into one with legal overrides. Thinking in OCaml terms, it may be the "greatest compatible *supertype*" of **`Diamond`** and **`Bollygon`**. In that sense, it "is" **`Anygon`**---and OCaml may already be conjuring into existence a whole host of object types that would create too much clutter if coded explicitly. Is this a "yo-yo"-type danger, or is OCaml's way of leaving these types *implicit* OK?

The threshold of what I don't know is whether and how far OCaml really tries to infer something like a *common*-compatible supertype of **`Diamond`** and **`Bollygon`**. The "pop" example involves only a single descendant. This may be at the level of recent and current changes to the language.

Type Variance and Generalization

If this seems complicated enough with concrete shape types, imagine if **C** and **E** (not to mention **B** and **D**) are templated/generic type variables.

- Mainstream languages by-and-large have support for *covariance* **constraints**, exemplified by the Java syntax class **`Foo<E extends Bar> { ...`**
- Subtype constraints in newer languages including OCaml are written with **<** instead of "extends".
- Then you can also have "superclassing constraints" and write them like **S > Bar**. (I have no experience with them.)

OCaml also allows marking `'a` style type parameters `+` for covariant or `-` for contravariant as well as **!** for "injectivity"---which means operationally (?) that the type-inference mechanism is allowed to presume an actual subset relation on the data, not just compatibility of the interface. I skipped over this when giving the syntax for basic type declarations in the week 4 notes---here it is in full:

```
type-params      ::= type-param
                 |   ( type-param { , type-param } )

type-param       ::= [ext-variance] ' ident

ext-variance     ::= variance [injectivity]
                 |   injectivity [variance]

variance         ::= +
                 |   -

injectivity      ::=  !
```

Finally, to come back to the larger question of when and whether **type extension** is **specialization** or **generalization**, the essence seems to be that the onus for determining this is placed on the programmer rather than be borne by the language.

In particular, OCaml's extensible datatypes seem to punt this question, just because the name of the datatype stays the same. You don't get something like `expWithAssign` generalizing `exp`; the name stays `exp` (or `'a exp`) throughout.

Singleton Objects

[Language support varies widely, no clear picture. In C++ one can "disable the copy constructor" by declaring it private. Languages that elevate reference semantics tend to make it harder to clone objects.]

[Show project code and what is intended as singleton versus having multiple instances. Pure modules like **`Parser`** are clearly singleton. But how about the **`EvalStack`** and **`Storage`** objects?]

[The lecture finished by showing and discussing project code---illustrating mainly OCaml's handling of exceptions and `option` types, and coding quirks that go with using `match` to unpack multiple levels of `Some` and `None`. The latter two are literally existential, and if you've read *The Myth of Sisyphus* by Albert Camus, you know that recursion is the root of existential despair. The next lecture will tie in the parser code after flying over how a fundamental coding task appears in various languages.]

Programming Styles Across Languages---Including Prolog

Let's consider how the `insert` method for a binary search tree ADT might be coded in various languages we have seen. We'll start with a mashup of Java and Scala syntax:

```
class BST<E extends Comparable> {
   class Node(Node left, E item, Node right);  //Scala saves lines of code here
   Node root;
   Node base;   //as in "CL&R[S]" Algos text, instead of using null reference.
   ...
}
```

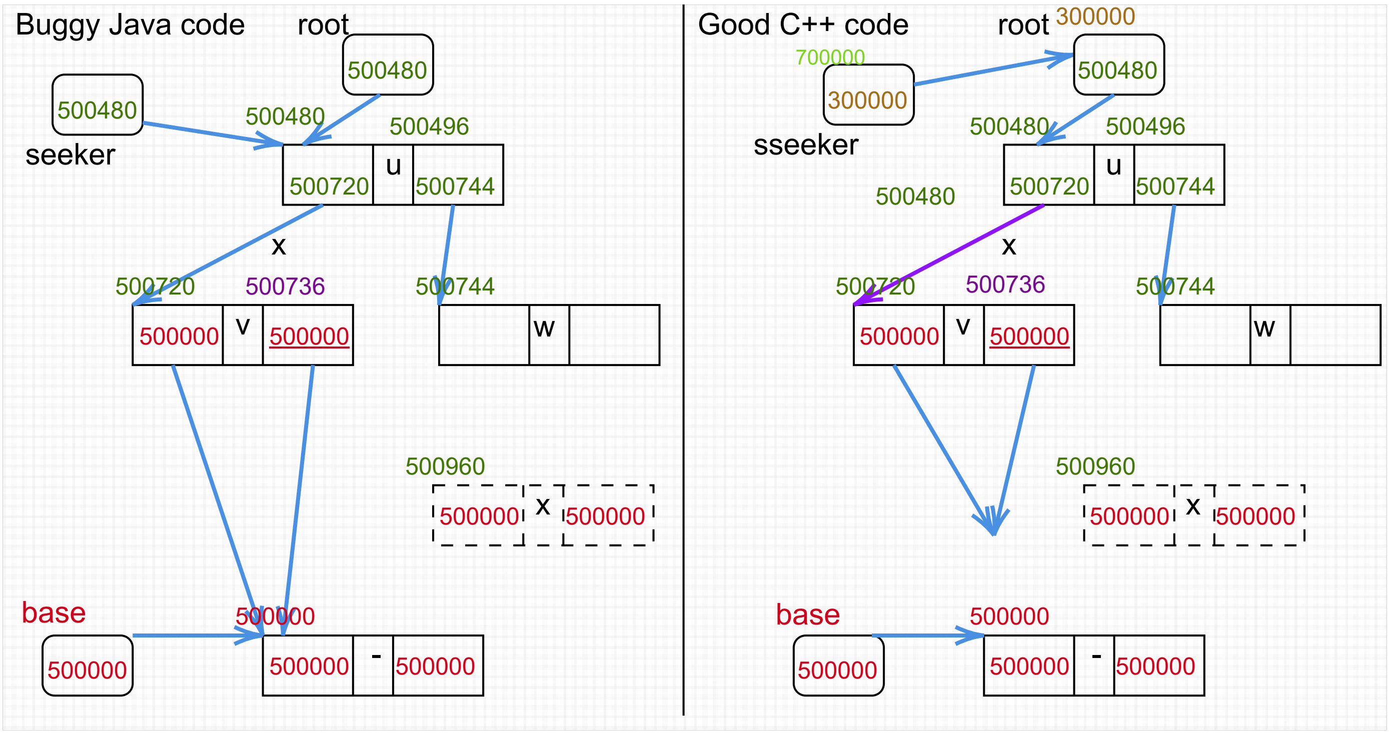
The following code is **buggy**---but *how it looks, and how the bug gets automatically fixed in similar-looking code,* are what we care about here:

```
   void insert(E x) {
      Node seeker = root;
      while (seeker != base) {
         if (x.compare(seeker.item) < 0) {
            seeker = seeker.left;
         } else {    //allow case item >< x, insert same-key items rightward
            seeker = seeker.right;
         }
      }
      seeker = new Node(base,x,base); //BUG: seeker not attached to tree
   }
```

One usual fix is to use a "`trailer`" reference/pointer that stays on the parent node of "`seeker`" and so that if `seeker` hits the base and last went left, then we do `trailer.left = new Node(...)`; if `seeker` last went right then `trailer.right` gets the new `Node` instead. Then the new node is attached to the tree. Or just use `trailer` for everything, or use `parent` links. In C++ one can use double indirection:

```
   //C++ code
   void insert(E x) {    //class has Node* root and Node* base
      Node** sseeker = &root;
      while ((*sseeker) != base) {
         if (x.compare((*sseeker)->item) < 0) {
            sseeker = &((*sseeker)->left);
         } else {
            sseeker = &((*sseeker)->right);
         }
      }
      (*sseeker) = new Node(base,x,base); //OK, (*sseeker) /is/ Node->left
   }                                      //or /is/ Node->right.
```

The reason this works where the other fails can be shown from storage-object diagrams. There we may take Node to be a record of three fields, each 8 bytes wide. Suppose the x to be inserted compares less than u but greater than v in the final leaf:



The difference in a nutshell is that when the C++ sseeker compares x with v, it does so through the purple arrow at right. The else branch gives it *the address of* the right field of the node with v, so it gets the value 500736. Now the ∗ *dereference* of that value equals the value of the base pointer, so the while-loop stops. At that moment:

- sseeker is 500736 as an *rvalue*, having initially been 300000 and then 500480 not 500496 (the latter being the address of the right field of the Node holding u), until it became 500736 after x compared >= v. (As an *lvalue,* sseeker is always 700000.)
- Hence (*sseeker) is the underlined 500000 as an *rvalue*, which makes it equal to base.
- But as an *lvalue*, (*sseeker) is still 500736. This deposits the address 500960 of the new Node into the storage object for the right field of the of the Node with v. So the new Node is connected to the tree.

Yuck. But 30 years ago I read sources saying this was the best way to code a BST in C++. And as review, note how the diagrams of storage objects, the levels of addressing, the notions of *lvalue* and *rvalue*, **and** the way they are translated without-or-with an extra fetch which we understand through the "rudimentary stack language", all come together to help understand this code. In the larger real-world picture, however, it must again be said: Yuck.

First note, however, how recursion fixes the bug organically:

```
//Java (C#, Scala, C++ etc. in this style would all work similarly.)
void insert(Node at, E x) {    //initially call insert(root, x)
   if (at == base) {
      at = new Node(base,x,base);
   } else if (x < at.item) {
      insert(at.left, x);      //at.left and at.right /are/ fields of a Node
   } else {
      insert(at.right, x);     //so a new Node here will connect to the tree.
   }
}
```

This recursive code is still **imperative**, however, because the recursive branches all speak the command verb **insert**, while the base case performs an action---an allocation---rather than return a value. Now let's see how OCaml will do it, first without tail recursion:

```
type 'e bst = Base | Node of 'e bst * 'e * 'e bst

let rec insert (subtree, x) = match subtree with
     Base -> Node(Base, x, Base)
   | Node(left, item, right) ->
         if compare(x,item) < 0 then
            Node(insert(left, x), item, right)
         else
            Node(left, item, insert(right, x));;
```

This works for the same reason the recursive Java code emulated the C++ bugfix of the original bad code: the recursive call works from the `left` or `right` *field* of the calling Node. There is a more primal way of seeing that this code is right, than delving into the C++ workings. *Read* the code this way:

- The result of inserting x into an empty tree is a node with x;
- The result of inserting x into a nonempty (sub)tree is a node that includes the result of inserting x into whichever of the left or right subtrees applies.

This code is **declarative** in the sense of *declaring results* as *nouns*. Whereas the recursive-but-imperative code has `insert` outside and node stuff inside on each branch, the declarative code has `Node` outside and the recursive calls to insert being inside.

However, *tail recursion* wants the calls to insert to be on the outside and to put the whole branch inside the (...) of insert:

```
let rec insert (subtree, x) = match subtree with
     Base -> Node(Base, x, Base)
   | Node(left, item, right) ->
         insert(if compare(x,item) < 0 then left else right, x)  (* Buggy *)
;;
```

Oops---this "throws away" the other half of the tree. We need to use the accumulator idea somehow to incorporate both subtrees of the node we're at into the resulting tree. So let's call the accumulator t for (new) tree rather than acc and dive right into the usual syntax:

```
let rec insert (subtree, x, t) = match subtree with
     Base -> t (?)  Or -> some function of (x,t) (?)
   | Node(left, item, right) ->
         insert(if compare(x,item) < 0 then left else right, x, update(x,t))
;;
```

When we were doing recursion over a linear data structure like summing over a list, the update function was just `acc -> x + acc`. Here, however, what can it be?

Life would be easier if we were programming a Boolean function `insertingGives` that just said t was equal to the tree we want. Then the base case is easy to write:

```
(** t is the result of inserting x into s *)
let rec insertingGives(s, x, t) = match s with
     Base -> t = Node(Base,x,Base)
```

How about the recursion case? For tail recursion, we need it to have the form

```
   | Node(left,item,right) -> insertingGives( ..., x, t' )
```

Let's use the `when` clause idea to divide this into the two comparison cases:

```
   | Node(left,item,right) when compare(x,item) < 0 ->
         insertingGives( ..., x, t' )
   | Node(left,item,right) when compare(x,item) >= 0 ->
         insertingGives( ..., x, t'' )
```

Now there is some more logic we can bring to bear to tackle the problem:

- Inserting x into a nonempty tree s can never make the resulting tree t be empty.
- Nor will x displace the original item: we are always inserting with a new leaf.

- So t will always have the form `Node(left',item,right')` for some possibly changed left and right subtrees.

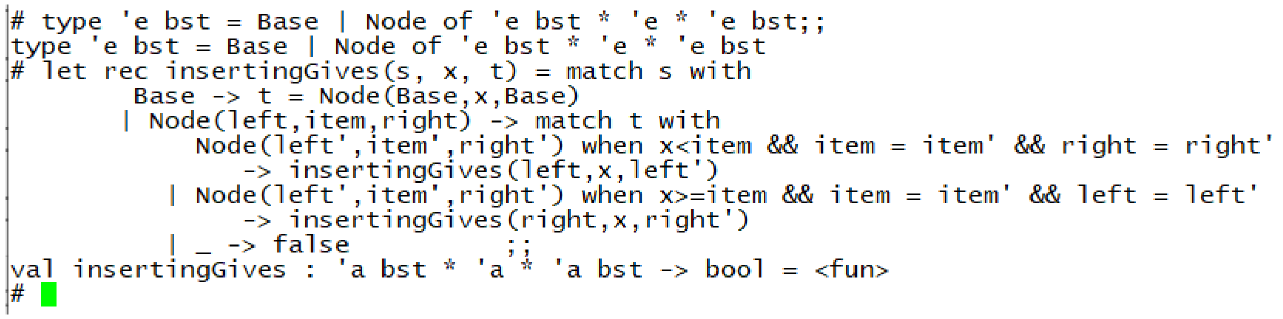
This means we should actually first be doing a nested match on t before applying our guarded-branch idea. Now the lightbulb flashes on: if x < item then what we need is that the result of inserting x into left gives left', with item and right staying the same. Whereas if x >= item then we need that the result of inserting x into right gives right', with left and item staying the same.

```
| Node(left,item,right) -> match t with
      Node(left',item',right') when x<item -> insertingGives(left,x,left')
    | Node(left',item',right') when x>=item -> insertingGives(right,x,right')
```

The logic here is not quite airtight because we need the "stays the same" part. But owing to how OCaml match-case behaves, we can just stick that too after `when` and use the wildcard feature to declare that any other case is false:

```
| Node(left,item,right) -> match t with
      Node(left',item',right') when x<item && item = item' && right = right'
         -> insertingGives(left,x,left')
    | Node(left',item',right') when x>=item && item = item' && left = left'
         -> insertingGives(right,x,right')
    | _ -> false
```

This is indeed legal---partly because apostrophe is allowed in an ordinary variable name whan it is not the first character:



```
# type 'e bst = Base | Node of 'e bst * 'e * 'e bst;;
type 'e bst = Base | Node of 'e bst * 'e * 'e bst
# let rec insertingGives(s, x, t) = match s with
          Base -> t = Node(Base,x,Base)
        | Node(left,item,right) -> match t with
              Node(left',item',right') when x<item && item = item' && right = right'
                 -> insertingGives(left,x,left')
            | Node(left',item',right') when x>=item && item = item' && left = left'
                 -> insertingGives(right,x,right')
            | _ -> false          ;;
val insertingGives : 'a bst * 'a * 'a bst -> bool = <fun>
# 
```

But---does this help us build the resulting tree? It would be nice if we could say the tree is built automatically. Then we would enjoy all of the following advantages:

- We didn't have to worry about how the algorithm operates and whether pointers "fall apart"---we just had to recurse the logic of: "left' is the result of inserting x into left or right' is the result of inserting x into right, whichever case applies."
- We got tail-recursive code---at least under the liberal "in any branch" definition.
- Although each branch begins with a verb again, it's not an imperative action verb like insert; it's a logical status verb, insertingGives.

All told, this code gives the appearance of only needing to declare the resulting logic. Well, there is a language in which you can write specifications and they actually execute---the system figures out how to execute them. Its *entire code* for this problem looks like this:

```
insert(base, X, node(base, X, base)).
insert(node(Left, Item, Right), X, node(LeftNew, Item, Right))
   :- X < Item, !, insert(Left, X, LeftNew).
insert(node(Left, Item, Right), X, node(Left, Item, RightNew))
   :- X >= Item, !, insert(Right, X, RightNew).
```

The ! means that the facts from the `:-` leading up to this stage, once brought about, are irrevocable. It is called "**cut**" and is both a boon and bane of Prolog. It is the only imperative feature built on top of Prolog's execution mechanism, which is called (**logical**) **unification** (via **resolution**). Unification is roughly similar to some combination of OCaml's type-inference checker and its subtype determinant.

For all of these reasons: declarativeness, conciseness, reliability, and harnessing a powerful deductive algorithm, Prolog ("*Programmation en Logique*") was part of a big "AI hype train" in the late 1970s and 1980s, under the banner of the "Fifth Generation Project." [show link] Like with ML/OCaml, there was Edinburgh-France synergy here too.

Basics of Prolog

We can make a bunch of observations right away---one can almost learn the whole language from this single example.

```
insert(base, X, node(base, X, base)).
insert(node(Left, Item, Right), X, node(LeftNew, Item, Right))
   :- X < Item, !, insert(Left, X, LeftNew).
insert(node(Left, Item, Right), X, node(Left, Item, RightNew))
   :- X >= Item, !, insert(Right, X, RightNew).
```

1. Variables are uppercase, matchable constants are lowercase---opposite of OCaml.
2. No need to define a separate `base | node(·,·,·)` datatype---the logic automatically takes care of it.
3. No reserved words here: my source wrote `nil` instead of `base`.
4. The symbol `:-` is like BNF ::= and works like ML/OCaml's match-case arrow except that the "flow of logic" is not strictly left-to-right. Best read as "**...is satisfied by...**"
5. A line without `:-` is hence satisfied by itself and works like an axiom. Called a **fact** in Prolog. Also called a **headless Horn clause** in the text.
6. Lines with `:-` are called **rules**. Both facts and rules end in a single period.
7. The comma is logical **and**. The right-hand side of a rule gives a list of stuff, all of which need to be made true in order to satisfy the left-hand side. Items separated by commas are called

terms. The use of just commas in a rule makes it logically equivalent to a **Horn clause**. (Red means that using those words is not required.)

8. Having two or more rules with the same left-hand side gives multiple ways to satisfy it, and so has the effect of logical **or**.
9. Here, you can verify the result of inserting `x` into `node(Left, Item, Right)` in either of two ways---depending on whether `x < Item` or `x >= Item`.
10. Prolog allows something that OCaml doesn't---matching the same variable twice to say their values in that case must be equal. Used with `Left`, `Right`, and `Item`. In this sense, variables give "matchable structure" too as well as the lowercase **atoms**.
11. As with OCaml, the rules are sequential. In OCaml, we've seen examples with mutually exhaustive `when` conditions where we could have skipped the second one. My source (subsection 8.1.4) actually does this:

```
insert(base, X, node(base, X, base)).
insert(node(Left, Item, Right), X, node(LeftNew, Item, Right))
   :- X < Item, !, insert(Left, X, LeftNew).
insert(node(Left, Item, Right), X, node(Left, Item, RightNew))
   :- insert(Right, X, RightNew).
```

12. The source also uses a different insertion policy: if x and the node's item compare equal, then silently don't insert x:

```
insert(base, X, node(base, X, base)).
insert(node(Left, Item, Right), Item, node(LeftNew, Item, Right)) :- !.
insert(node(Left, Item, Right), X, node(LeftNew, Item, Right))
   :- X < Item, !, insert(Left, X, LeftNew).
insert(node(Left, Item, Right), X, node(Left, Item, RightNew))
   :- insert(Right, X, RightNew).
```

The new second rule could also read:

```
insert(node(Left, Item, Right), X, node(LeftNew, Item, Right)) :- X = Item, !.
```

In both cases, the **cut** `!` says "We're done here---don't try any other possibilities to my left." It's like a one-way valve in the flow of logic. What it does operationally is cut off **backtrack** in the depth-first-search that underlies the mechanism of Prolog. This is not search in the tree we're thinking of, but in the "metaspace" of possible assoiations of Prolog's atoms according to the logic. This meta-algorithm is rich enough to incorporate the basic BST insertion algorithm via these rules.

13. Because `head :- body` is read as "if body then head"---or more precisely, "head is satisfied by body"---the left-hand side is called the **consequent** even though it comes first, and the right-hand side is the **antecedent.**

14. Not only does Prolog have a "REPL" (read-evaluate-print loop) like OCaml and Scala and Python, its programs are intended to be used as databases that one can submit **queries** to. If the query has no variables, it will get a simple true/false response. If it has variables and the variables can be set to make it true, Prolog will give (at least) one way to do so; if not, Prolog will reply false or maybe `fail`.
15. Prolog here makes a **closed-world-assumption** that if it cannot establish something from its world of given data, then that something is false. This policy is called **negation as failure**. It is "naff" in the British sense of being a little dodgy. To quote the Stones, it's like saying "if you can't get no satisfaction" then it's false.

[show simpler Sebesta examples] and/or [show analogy to Parser.ml in project code]

To reiterate the point about the bug in the latter: I "**refactored**" my older parser in Standard ML by separating off the grouping of unary operations to a separate function. This makes the part dealing with binary operations shorter and easier to read. I did not, however, create a separate datatype for what was originally a single function from "`gentree`" to "`ubtree`". Instead, I just added `Unop` and `Binop` variants to the original `Op` within the same datatype. So I created the tandem of `groupUnops: gentree list -> gentree list` and `group: gentree -> gentree` with the extra requirement of removing every `Op` not being enforced by the type checking. Whereas, the type-checking of `parse: gentree -> ubtree` ensures that no `gentree` symbols are left in the output.

Prolog does not have a similar notion of "type". So how could we do a similar check in Prolog? We could try to build checking logic for it directly. A basic fact is `hasOp(op X).`, and a basic rule is `hasOp(node(X)) :- hasOp(X).` We would need more rules to make it "compile-time check" the code, rather than just "run-time check" a single output. But it could be possible. Prolog's inference mechanism is powerful and general, but is it seamlessly applicable?

Whether Prolog would help implement and verify the OCaml parser leads to a simple gut-check issue. The code in `Parser.ml` already has some patterns that stretch across the whole screen. Giving a possible result pattern in Prolog would double that wingspan of a line of code. Well, the result patterns could be on different lines the way the `if-then-else` expressions already are, but heaviness of syntax remains an issue in many Prolog sources (in both senses of the word).