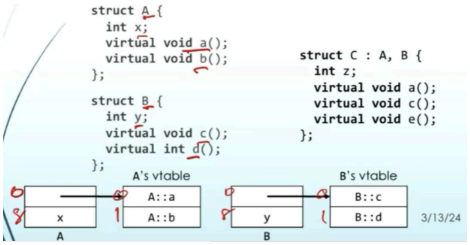
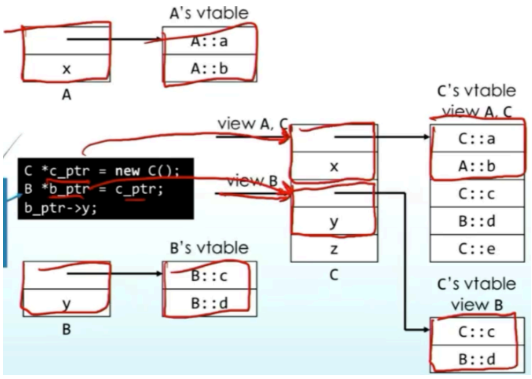
```
def emit_type_defs(self, ctx):
        """Emit definitions of user-defined types."""
        ctx.print(f'struct UC_TYPEDEF({self.name.raw}) ' + '{', indent=True)
        new_ctx = ctx.clone()
        new_ctx.indent += '  '
        for field in self.fielddecls:
            field.emit(new_ctx)
        new_ctx.print(f'UC_PRIMITIVE(boolean) operator==(const UC_TYPEDEF'
                      f'({self.name.raw}) &rhs) const = default;',
                      indent=True)
        ctx.print('};\n', indent=True)
```






```
void main(string[] args) {
    foo int = new foo(3);    //foo has one member var x
    int[] myArr = new int[] {1, 2, 3, 4};
    myArr << 5;
    println(int_to_string(myArr[4]));   //5
    myArr >> null;
    println(int_to_string(myArr.length));   //4
    println(true + "lol");  //truelol
    int i = 0;  //int i; is not allowed in uc
    for (;;) {
        if (i == 10) {
            break;
        }
        print("i is ");
        println(int_to_string(i));
        ++i;
    }
    return;
}
```

```
; solution using cond
(define (account balance)
  (lambda (message . args)
    (cond ((eq? message 'balance) balance)
          ((eq? message 'deposit)
           (set! balance (+ balance (car args)))
           balance)
          ((eq? message 'withdraw)
           (let ((amount (car args)))
             (if (> amount balance)
                 "Insufficient funds"
                 (begin (set! balance (- balance amount))
                        balance
                 )
             )
           ))
    )
  )
)
```

```
% merge(List1, List2, Result).
% True if Result is the sorted merge of the sorted lists List1 and
% List2.
% List1: in, must be a list of numbers in increasing order
% List2: in, must be a list of numbers in increasing order
% Result: out
merge([], List, List).
merge(List, [], List).
merge([First1|Rest1], [First2|Rest2], [First1|ResultRest]) :-
    First1 =< First2,
    merge(Rest1, [First2|Rest2], ResultRest).
merge([First1|Rest1], [First2|Rest2], [First2|ResultRest]) :-
    First2 < First1,
    merge([First1|Rest1], Rest2, ResultRest).

% mergesort(List, Result).
% True if Result consists of the numbers in List but in sorted,
% increasing order.
% List: in, must be a list of numbers
% Result: out
% base cases:
mergesort([], []).
mergesort([Item], [Item]).
% recursive case
mergesort(List, Sorted) :-
    writeln(List),
    % algorithm: split into halves, recursively sort halves, merge halves.
    length(List, ListLength),
    HalfLength is ListLength // 2,
    append(LeftHalf, RightHalf, List),
    length(RightHalf, HalfLength),
    mergesort(LeftHalf, LeftSorted),
    mergesort(RightHalf, RightSorted),
    merge(LeftSorted, RightSorted, Sorted).
```

```
list_contains([Item|_Rest], Item).
list_contains([_First|Rest], Item) :-
    list_contains(Rest, Item).

any_contains([List|_RestLists], Item) :-
    list_contains(List, Item).
any_contains([_List|RestLists], Item) :-
    any_contains(RestLists, Item).
```

```
all_even([]).
all_even([First|Rest]) :-
    First mod 2 =:= 0, all-even(Rest).

reverse(List, Reversed) :-
    reverse_tail(List, [], Reversed).
% reverse_tail(List, ResultSoFar, FinalResult) :-
reverse_tail([], ResultSoFar, ResultSoFar).
reverse_tail([First|Rest], ResultSoFar, FinalResult) :-
    reverse_tail(Rest, [First|ResultSoFar], FinalResult).

match_stars([], []).
match_stars([Item|Rest1], [Item|Rest2]) :-
    match_stars(Rest1, Rest2).
% match one or more items
match_stars([*|Rest1], [_Item|Rest2]) :-
    match_stars([*|Rest1], Rest2)
% match zero items
match_stars([*|Rest1], Rest2]) :-
    match_stars(Rest1, Rest2)
```

```
(define (make-stack)
  (let ((entries '()))
    (lambda (message . args)
      (cond ((eq? message 'push)
             (set! entries (cons (car args) entries)))
            ((eq? message 'pop)
             (let ((item (car entries)))
               (set! entries (cdr entries))
               item
             ))
             ((eq? message 'top) (car entries))
             ((eq? message 'size) (length entries))
      )
    )
  )
)
```

Note: built-in prolog predicates include append, length, permutations(thisIsAPermutationOf,This)

```
template<typename Container>
auto max_element(const Container &container) -> decltype(*container.begin()) {   //causes sub failure on arrays/ptrs
    auto max_iter = container.begin();
    for (auto iter = max_iter; iter != container.end(); ++iter) {
        if (*iter > *max_iter) {
            max_iter = iter;
        }
    }
    return *max_iter;
}

template<class T>
auto uc_add(T arg1, T arg2) {
  return arg1 + arg2;
}

UC_PRIMITIVE(string) uc_add(UC_PRIMITIVE(string) arg1, auto arg2) {
  UC_PRIMITIVE(string) num_str = std::to_string(arg2);
  return arg1 + num_str;
}
```

```
template <typename T, typename U>
auto mult(const T &x, const U &y) -> decltype(x * y) {
  return x * y;
}

std::string mult(const std::string &s, int count) {
  std::string result;
  for (int i = 0; i < count; ++i) {
    result += s;
  }
  return result;
}

std::string mult(int count, const std::string &s) {
  return mult(s, count);
}
```

In uC, there's UC_PRIMITIVE, UC_TYPEDEF, UC_FUNCTION, UC_VAR, uc_construct<ref>, uc_null_check(ref, pos), uc_array_{push/pop/index}(ref, item), uc_add(item1, item2)

Project 4 phases were:

- Finding declarations (add user-defined types and functions to the global environment with `def find_decls(tree, global_env):`)
- Resolving types (calculate types of arrays, make sure `void` is only used as a return type, set function parameter and return types)
- Resolving function calls (code in CallNode: `def resolve_calls(self, ctx):; self.func = ctx.global_env.lookup_function(ctx.phase, self.position, self.name.raw); super().resolve_calls(ctx))`
- Checking fields and variables and resolving names (create VarEnv's/scopes, add variable definitions to their local environment, set types of name expressions)
- Checking basic control flow (make sure break and continue are only used in loops)
- Computing and checking types (compute the type of every expression [first literals, then function calls, then operator results] , and make sure the appropriate types are used for the given functions/operators)

Project 5 phases were:

- Generating user-defined type declarations (ex. `struct UC_TYPEDEF(particle);`)
- Generating function declarations (ex. `UC_PRIMITIVE(double) UC_FUNCTION(max)(UC_PRIMITIVE(double) UC_VAR(a), UC_PRIMITIVE(double) UC_VAR(b));`)
- Generating user-defined type definitions (add {} to each member variable, set `UC_PRIMITIVE(boolean) operator==(const UC_TYPEDEF(particle) &rhs) const = default;`)
- Generating function definitions