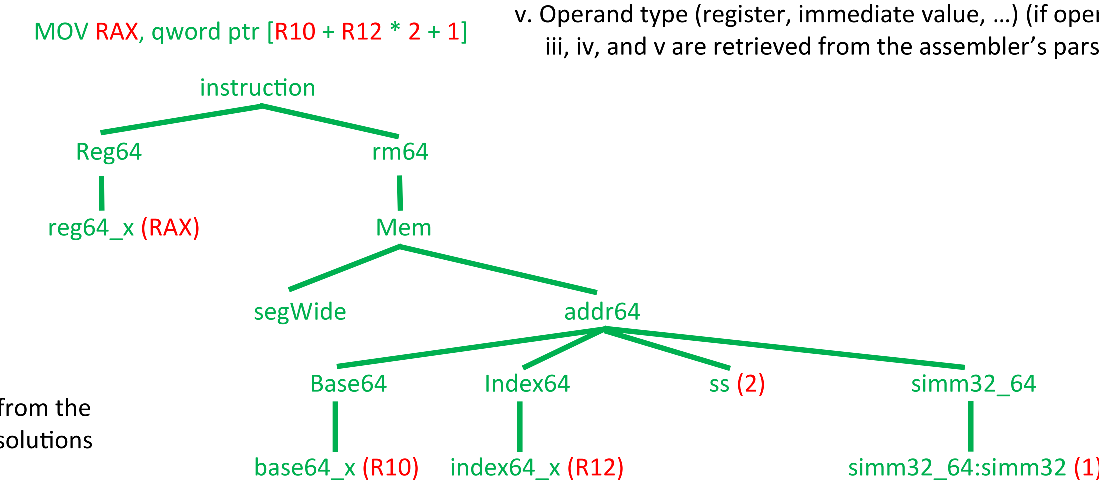
```java
/**
 * Construct a resolver for the given parse tree
 *
 * @param lang
 * @param at the address where the instruction will start
 * @param tree the parse tree
 * @param context the context expected at {@code instStart}
 * @param ctxGraph the context transition graph used to resolve purely-recursive productions
 */
public AssemblyTreeResolver(SleighLanguage lang, Address at, AssemblyParseBranch tree,
        AssemblyPatternBlock context, AssemblyContextGraph ctxGraph) {
    this.lang = lang;
    this.at = at;
    this.vals.put(INST_START, at.getAddressableWordOffset());
    this.tree = tree;
    this.grammar = tree.getGrammar();
    this.context = context.fillMask();
    this.ctxGraph = ctxGraph;
    this.operandData = AssemblyOperandData.buildAssemblyOperandDataTree(tree);
}

/**
 * Resolve the tree for the given parameters
 *
 * @return a set of resolutions (encodings and errors)
 */
public AssemblyResolutionResults resolve() {
    AssemblyResolvedPatterns empty = AssemblyResolution.nop("Empty");
    AssemblyConstructStateGenerator rootGen =
        new AssemblyConstructStateGenerator(this, tree, empty);

    Collection<AssemblyResolvedError> errors = new ArrayList<>();
    Stream<AssemblyGeneratedPrototype> protStream =
        rootGen.generate(new GeneratorContext(List.of(), 0));

    if (DBG == DbgTimer.ACTIVE) {
        try (DbgCtx dc = DBG.start("Prototypes:")) {
            protStream = protStream.map(prot -> {
                DBG.println(prot);
                return prot;
            }).collect(Collectors.toList()).stream();
        }
    }

    Stream<AssemblyResolvedPatterns> patStream =
        protStream.map(p -> p.state).distinct().flatMap(s -> {
            // add operand data to results structure
            empty.setOperandData(operandData);
            return s.resolve(empty, errors);
        });

    AssemblyResolutionResults results = new AssemblyResolutionResults();
    patStream.forEach(results::add);

    results = resolveRootRecursion(results);
    results = resolvePendingBackfills(results);
    results = selectContext(results);
    // TODO: Remove this? It's subsumed by filterByDisassembly, and more accurately....
    results = filterForbidden(results);
    results = filterByDisassembly(results);
    results = fillMasksVals(results);
    results.addAll(errors);
    return results;
}
```

Code located at src/main/java/org/mitre/pickledcanary/assembler/sleigh/sem/AssemblyTreeResolver.java

Obtaining suboperand info from Ghidra assembler v3

Goal: Get suboperand information from Ghidra's assembler and deliver it to users via the AssemblyOperandData object

What happens here?	What fields of AssemblyOperandData get populated here?
AssemblyOperandData buildAssemblyOperandDataTree() 5 • Build the skeleton of the operand data tree (for example, every node in the example instruction below gets an AssemblyOperandData object; basically, the structure of the tree is identical to that of the tree argument passed in) • This AssemblyOperandData tree will eventually be returned to the user in an AssemblyResolution object	i. Code name (e.g. reg64_x) ii. Operand name (e.g. RAX) iii. Wildcard name (e.g. Q1) (if operand is wildcarded) iv. Wildcard index (0-based idx of wildcard in instruction (if operand is wildcarded) v. Operand type (register, immediate value, …) (if operand is wildcarded) iii, iv, and v are retrieved from the assembler's parser



Brown numbers are function numbers from the Ultimate Diagram

Transfer AssemblyOperandData from the TreeResolver to the AssemblyResolutions (the results)

What happens here?	What fields of AssemblyOperandData get populated here?
AssemblyOperandState resolve() 14 • This method is where Ghidra assembles the (sub)operands one at a time by recursively looping over the tree • We get the AssemblyOperandData node that corresponds to the (sub) operand by checking that the code name (node names of the tree above) of the AssemblyOperandData matches that of the (sub)operand	vi. Partial operand mask (leading and trailing bytes added later) vii. Partial operand value (leading and trailing bytes added later) viii. Shift (for normal operands, helps determine operand byte within the instruction) ix. Expression (for normal operands, helps determine operand bits within a byte and the target for jump instructions)
AssemblyTreeResolver applyRecursionPath() 24 • Add the root shift here	x. shift (for the "instruction" code name, or root) * root mask, value, and expression are not populated because opcodes cannot be wildcarded in Pickled Canary
AssemblyResolvedBackfill solve() 30 • This is where Ghidra assembles the (sub)operands for those that could not be assembled above • We populate the ASsemblyOperandData fields for those operands here	xi. Partial operand mask (for backfills, leading and trialing bytes added later) xii. Partial operand value (for backfills, leading and trailing bytes added later) xiii. Shift (for backfills) xiv. Expression (for backfills)
AssemblyOperandData applyShifts() 32 • Add leading and trailing bytes to masks and values	
AssemblyOperandData fillMissingMasksVals() 33 • Some (sub)operands do not have bits of their own; instead, they affect bits in the opcode • Ghidra does not give these (sub)operands masks and values, so we give them empty byte arrays here	xv. Operand mask (for (sub)operands that do not have any bits of their own (quasi-opcodes in previous assembler)) xvi. Operand value (for (sub)operands that do not have any bits of their own (quasi-opcodes in previous assembler))